Verteilte Ausführung hybrider System-Modelle auf heterogenen Rechnerplattformen

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

von der Fakultät für Elektrotechnik und Informationstechnik

der Universität Fridericiana Karlsruhe

genehmigte

DISSERTATION

von

Dipl.-Ing. Rico Dreier

aus

SINSHEIM

Tag der mündlichen Prüfung: 14. Februar 2006

Hauptreferent: Prof. Dr.-Ing. Klaus D. Müller-Glaser

Korreferent: Prof. Dr.-Ing. Jörg Henkel

Karlsruhe: 17. Mai 2006

Kurzfassung

Wachsende Komplexität und Verbreitung eingebetteter Systeme in der Automobilindustrie, aber zum Beispiel auch in der Luft- und Raumfahrttechnik, machen neue, angepasste Entwicklungsverfahren notwendig. Der Entwurf elektronischer Steuergeräte für Automobile erfolgt heute unter Einsatz leistungsfähiger CASE-Werkzeuge. Wichtige Voraussetzungen für den durchgängigen Entwurf sind dabei vor allem Code-Generatoren, aber auch der frühzeitige Einsatz eines geeigneten Echtzeit-Betriebssystems auf dem Zielsystem zur Einhaltung zeitlicher Randbedingungen sowie eine Gesamtsystem-Simulation, damit Aussagen über Qualität und Kosten bereits in frühen Entwicklungsphasen möglich sind und die Markteinführungszeit verkürzt werden kann. Mit Hilfe der Simulation können Abläufe eines realen Systems anhand eines vereinfachten Modells untersucht und Erkenntnisse gewonnen werden, die Rückschlüsse auf das Verhalten des realen Systems zulassen. Der Grad an Übereinstimmung der Simulationsergebnisse mit dem tatsächlichen Verhalten des realen Systems hängt dabei entscheidend von der Genauigkeit ab, mit der das Modell relevante Aspekte der Realität nachbildet. Je umfangreicher und detaillierter das Modell ist, desto größer ist jedoch der Rechenaufwand für die Simulation. Darüber hinaus kann der Aufwand (zum Beispiel bei einer ereignisorientierten Simulation) nichtlinear mit der Modellgröße ansteigen, so dass selbst die schnellsten heutigen Rechner eine lange Ausführungszeit benötigen.

Viele Systeme sind durch ausschließlich zeitkontinuierliche oder ausschließlich zeitdiskrete, zustandsbasierte Modelle nicht adäquat darstellbar, sondern nur durch hybride System-Modelle. Für eine Gesamtsystem-Simulation ist es aufgrund unverzichtbarer oder spezieller Eigenschaften von CASE-Werkzeugen (einschließlich integrierter Simulatoren), die auch das Zielsystem betreffen können, oftmals notwendig, unterschiedliche Werkzeuge zu verwenden und miteinander zu koppeln, um eine Co-Simulation durchführen zu können. Solche Eigenschaften sind neben bestimmten Modellierungsparadigmen die Verfügbarkeit von zum Entwurf benötigten oder bereits existierenden, getesteten Modell-Bibliotheken, in die CASE-Werkzeuge integrierte Seriencode-Generatoren für das Hardware-Zielsystem und einer Unterstützung bei der modellbasierten Integration eines bestimmten Echtzeit-Betriebssystems wie zum Beispiel OSEK©/VDX. Abhängig von den eingesetzten CASE-Werkzeugen und den Komponenten des Zielsystems kann eine verteilte Simulation auf einer heterogenen Rechnerplattform basieren, bestehend zum Beispiel aus PC- und Mikrocontroller-basierten Systemen. Dies gilt auch, wenn FPGAs Komponenten des Zielsystems sind oder als Mittel zur Simulationsbeschleunigung eingesetzt werden, so dass Teilsysteme durch Emulation auf FPGAs ausgeführt werden.

Neben der Frage, wie die Markteinführungszeit verkürzt werden kann – nämlich durch Beschleunigung der modellbasierten Entwicklung eingebetteter elektronischer Systeme beim Einsatz heterogener Rechnerplattformen – einschließlich der Anwendung von Konzepten wie Hardware-in-the-Loop – stellt sich insbesondere auch die Problematik, wie sich Simulationsexperimente beschleunigen lassen.

In der vorliegenden Arbeit wird zur Gesamtsystem-Validierung sowohl eine Kommunikations-Infrastruktur für eine Hardware-in-the-Loop-Simulation zum schrittweisen Testen der Funktionalität unterschiedlich weit entwickelter Systemkomponenten entwickelt – bei Verwendung der realen Zielplattform und des realen Bussystems (CAN) zur Erzielung praxisnaher Ergebnisse – als auch vorhandene Rapid-Prototyping-Systeme in ein Gesamtkonzept integriert. Dabei werden zentrale Eigenschaften verschiedener OSEK/VDX-Echtzeit-Betriebssysteme und kommerzieller Seriencode-Generatoren berücksichtigt, da sich die Eigenschaften der automatisch generierten Quellcodes und der Echtzeit-Betriebssysteme zum Teil erheblich unterscheiden, so dass sie sich nicht für alle Applikationen gleichermaßen gut eignen. Darüber hinaus werden als Alternativen zur Ausführungsbeschleunigung zeitdiskreter, zustandsbasierter sowie hybrider System-Modelle Hardware-Emulation (per FPGA), Parallelisierung (per Partitionierung und Co-Simulation) und effizientere Co-Simulationsverfahren (per optimistischer Synchronisation) untersucht. Dazu dient u. a. eine im Rahmen der Arbeit entwickelte Java-basierte, plattformunabhängige Co-Simulations-Umgebung, die heterogene Ausführungsumgebungen integriert (PC-, Mikrocontroller- und FPGA-basierte Systeme). Darin sind

dedizierte Simulatoren für Logikschaltungen und Statecharts eingeschlossen, die optimistische Synchronisation unterstützen sowie ein Werkzeug zur automatischen Generierung von VHDL-Code aus Statecharts zur Konfiguration von FPGAs. Als Voraussetzung für eine verteilte Simulation resultiert ausgehend von Partitionierungsverfahren für Logikschaltungen ein Konzept zur Partitionierung von Statecharts. Weiterhin wird ein existierendes, auf optimistischen Synchronisationsalgorithmen basierendes Verfahren zur verteilten Simulation hybrider System-Modelle verbessert, so dass der Kommunikationsaufwand zwischen den einzelnen Partitionen wesentlich reduziert wird.

Abstract

Distributed Execution of Hybrid System Models on Heterogeneous Computer Platforms

Growing complexity and popularity of distributed systems in automotive as well as in aerospace industries require new, adapted design techniques. The design of electronic control units for automobiles is done today using powerful CASE tools. Important prerequisites for a seamless design flow are code generators as well as the early usage of an adequate real time operating system on the target system in order to meet time constraints and to simulate the complete system for predictions on the quality and the costs in early design phases; thus speeding up the product launch. Through simulation, the operation sequences of a real system can be studied on a simplified model to provide an insight into the behavior of the real system. The degree in which the simulation results comply with the actual behavior of the real system greatly depends on how exactly the model represents relevant aspects of the reality. The larger and more detailed the model is, the more complex are the calculations of the simulation. In addition to that, the computing complexity of an event based simulation may grow non-linearly with model size, so that even the most powerful computers of today require a long execution time.

Many systems cannot be adequately described with explicitly time-continuous or explicitly time-discrete, state based models, but only with hybrid system models. For a simulation of the complete system it is often necessary to use and link different tools with each other to carry out a co-simulation, because of essential or otherwise special properties of CASE tools (including integrated simulators), which possibly concern the target system as well. Such properties are, among certain modeling paradigms, e. g., the availability of model libraries necessary for the design or already existent and tested libraries, of production code generators integrated into the CASE tools for the hardware target system, and of support for the model-based integration of a certain real time operating system like OSEK©/VDX. Depending on the CASE tools deployed and the components of the target system, a distributed simulation may be based upon a heterogeneous computer platform. The platform may consist of, for example, PC and microcontroller based systems. It may also include reconfigurable hardware (FPGAs) as a part of the target system or as an emulator accelerating the simulation of a subsystem.

Along with the issue of how to speed up the product launch through quicker model-based design of embedded electronic systems when using heterogeneous computer platforms, including the usage of concepts like hardware-in-the-loop, the issue of how to accelerate simulation experiments must also be considered.

In the following, for a complete system validation, both a communication infrastructure for a hardware-in-the-loop simulation is developed to stepwise test the functionality of system components in different states of design – using the actual target platform and the real bus system (CAN) to achieve practical results – and already existing rapid prototyping systems are integrated into the over-all concept. Here the central properties of different OSEK/VDX real time operating systems and commercial production code generators are regarded, as the properties of the automatically generated source codes and of the real time operating systems differ in part substantially from each other, so that they are not equally suited for all applications. Furthermore, hardware emulation (with FPGAs), parallelization (through partitioning and co-simulation) and more efficient co-simulation methods (through optimistic synchronization) are examined as alternatives to speeding-up the execution of time dis-

crete, state based and hybrid system models. This is exemplary validated using a Java based, platform independent co-simulation environment to integrate heterogeneous execution environments (PC-, microcontroller- and FPGA-based systems), developed in the course of the dissertation. Dedicated simulators for logical circuits and Statecharts are included, supporting optimistic synchronization, as well as a tool for automatically generating VHDL code out of Statecharts to configure FPGAs. Starting from partitioning techniques for logical circuits, a concept for partitioning Statecharts results as a prerequisite for a distributed simulation. Furthermore, an already existing method for distributed simulation of hybrid system models, based on optimistic synchronization algorithms, is improved, so that the communication costs between the individual partitions are considerably reduced.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am FZI Forschungszentrum Informatik an der Universität Karlsruhe im Forschungsbereich Elektronische Systeme und Mikrosysteme (ESM).

Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. Klaus D. Müller-Glaser, der diese Arbeit ermöglichte und dessen konstruktive Kritik, wertvolle fachliche Anregungen und Unterstützung während der gesamten Bearbeitungszeit wesentlich zum Gelingen der vorliegenden Arbeit beigetragen haben. In gleicher Weise gilt mein Dank Herrn Prof. Dr.-Ing. Jörg Henkel für die freundliche Übernahme des Zweitgutachtens.

Meinen Kollegen danke ich für die angenehme kollegiale Zusammenarbeit bei der täglichen Projektarbeit – allen voran Thomas Stingl für die nicht nur fachlich interessanten Diskussionen – und meinem ehemaligen Abteilungsleiter Stefan Schmerler für die Vorarbeiten bei ESM im Bereich verteilter Simulation, aus denen sich zahlreiche Anregungen für die vorliegende Arbeit ergaben. Für die Zubilligung des notwendigen Freiraums zur Entwicklung eines eigenen wissenschaftlichen Forschungsthemas und die in den späteren Phasen der wissenschaftlichen Arbeit gewährten Freiräume gilt mein Dank meinen ehemaligen Abteilungsleitern Markus Kühl und Eric Sax.

Wesentliche Unterstützung bei der Realisierung der hier vorgestellten Arbeit verdanke ich meinen Kollegen, insbesondere Gerd Frick für die gemeinsamen Diskussionen über die Inhalte der Arbeit, die mir auf vielfältige Weise geholfen haben sowie „meinen“ studentischen Mitarbeitern, vor allem Georg Wendt und Amir Bukhari, die mir bei den Implementierungsarbeiten, zahlreichen Messreihen und Erprobungen von Ideen mit großem Engagement zur Seite standen.

Böblingen-Dagersheim, im Mai 2006

Rico Dreier

Inhaltsverzeichnis

Verwendete Abkürzungen

A/D	Analog/Digital-Wandler
ABS	AntiBlockierSystem
ACK	ACKnowledge
AMS	Analog und Mixed-Signal
ANSI	American National Standards Institute
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
AT	Activate Task
ATM	Asynchronous Transfer Mode
AutoSAR	AUTomotive Open Systems ARchitecture
BCCx	Basic Conformance Class x
BDM	Background Debug Mode
BEE	Berkeley Emulation Engine
BMFT	BundesMinisterium für Forschung und Technologie
BP	BackPlane
BSP	Board Support Package
BTB	Breathing Time Buckets
BTW	Breathing Time Warp
CAD	Computer Aided Design
CAN	Controller Area Network
CANoe	CAN open environment
CAPL	CAN Access Programming Language
CASE	Computer Aided Software Engineering, Computer Aided Systems Engineering
CCP	CAN Calibration Protocol
CI	Communication Interface
CISC	Complex Instruction Set Computer
ClearSim-MD	ClearSim-MultiDomain
Clk	Takt
CMB	Synchronisationsansatz nach Chandy-Misra-Bryant
COM	Communication
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off The Shelf
COW	Cluster Of Workstations
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CS	Continuous Simulation, Continuous Simulator
CSMA/CA	Carrier-Sense Multiple-Access with Collision Avoidance
CT	Chain Task
CTW	Clustered Time Warp
CUCB	Communication User Code Block
D/A	Digital/Analog-Wandler
DAG	Directed Acyclic Graph
DBS	SQL format DataBaSe
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DES	Discrete Event Simulation, Discrete Event Simulator
DESMO-J	Discrete-Event Simulation and MOdelling in Java
DESS	Differential Equation System Specification
DEVS	Discrete EVent System specification
DGL	DifferentialGLeichung
DLB	Dynamic Load Balancing
DLL	Data Link Layer, Dynamic-Link Library
DMA	Direct Memory Access
DoD	Department of Defense

DP	Dual Port
DPRAM	Dual-Port Random Access Memory
DRAM	Dynamic RAM
DS Toolbox	Distributed Simulink Toolbox
DSF	Distributed Simulation Framework
DSP	Digital Signal Processing/Processor
DTSS	Discrete Time System Specification
DV	DatenVerarbeitung
E/A	Eingabe/Ausgabe
EC	Embedded Coder
ECCO	in ASCET integrierter Embedded Code Creator and Optimizer
ECCx	Extended Conformance Class x
ECU	Electronic Control Unit
EDIF	Electronic Design Interchange Format
EMV	ElektroMagnetische Verträglichkeit
EPROM	Erasable Programmable ROM
EQ	Event Queue
ESM	Elektronische Systeme und Mikrosysteme
EVB	Evaluation Board
F	Synchronisationspunkt F
FES	First Event Synchronisation
FFT	Fast Fourier Transform
FIFO	First In First Out
FL	Fuzzy Logic
FM	Fiduccia und Mattheyses
FP	Floating Point (Fließkomma)
FPGA	Field Programmable Gate Array (programmierbarer Logikbaustein)
FSM	Finite State Machine
FZI	ForschungsZentrum Informatik
GA	Genetische Algorithmen
GTW	Georgia Tech Time Warp
GUI	Graphical User Interface
GVT	Global Virtual Time
HDL	Hardware Description Language
HF	HochFrequenz
HIL	Hardware In the Loop
HIS	Hersteller Initiative Software
HLA	High Level Architecture
HW	HardWare
I/O	Input/Output
ICE	In-Circuit-Emulator
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ILS	Iowa Logic Simulator
IMB	InterModule Bus
INTERACT	INtegration heTERogener Ausführungsumgebungen zur Co-SimulaTion
IP	Internet Protocol, Intellectual Property
IP Core	Intellectual Property Core
IPC	Inter Process Communication
IQ	Input Queue
IRA	Institut für Regelungs- und Automatisierungstechnik
IRB	Institut für Rechnerstrukturen und Betriebssysteme (Universität Hannover)
IRQ	Interrupt Request
ISCAS	International Symposium on Circuits And Systems
ISE	Integrated Software Environment

ISO	International Organization for Standardization
ISR	Interrupt Status Register, Interrupt Service Routine
IVZ	Integrierte Virtuelle Zeit
J	Synchronisationspunkt J
JIT	Just In Time
JNI	Java Native Interface
JTAG	Joint Test Action Group
JVM	Java Virtual Machine
K-AFM	K-FM mit azyklischer Anfangspartitionierung
K-FM	Algorithmus von Sanchis
Kfz	Kraftfahrzeug
KI	Künstliche Intelligenz
KL	Kernighan und Lin
LAN	Local Area Network
LB	Load Balancing
LBTS	Lower Bound on TimeStamp
LCD	Liquid Crystal Display
LDB	Local Database
LED	Light Emitting Diode
LGPL	Lesser General Public License
LIFO	Last In First Out
LIN	Local Interconnect Network
LP	Logischer Prozess
LSB	Least Significant Bit
LSI	Large Scale Integration
LTW	Local Time Warp
LVT	Local Virtual Time
M.I.S.R.A.	Motor Industry Software Reliability Association
MAN	Metropolitan Area Network
MAST	Analog, Mixed-Technology and Mixed-Signal HDL for Saber (Synopsys, Inc.)
µC	Mikrocontroller
MIL	Model In the Loop
MILAN	Model based Integrated simuLAtioN framework
MIMD	Multiple Instruction stream, Multiple Data stream
MIP	MATLAB Integration Package
MIT	Massachusetts Institute of Technology
MMI	Mensch-Maschine-Interaktion
MOST	Media Oriented System Transport
MPEG	Moving Pictures Experts Group
MPI	Message Passing Interface
MSB	Most Significant Bit
MSC	Message Sequence Chart
MTW	Moving Time Window
NM	Network Management
NN	Neuronale Netze
NOW	Network Of Workstations
OIL	OSEK Implementation Language
OMG	Object Management Group
OO	ObjektOrientiert
OQ	Output Queue
ORB	Object Request Broker
ORTI	OSEK Run-Time Interface
OS	Operating System
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
OSI	Open Systems Interconnection
PADOC	Probabilistic Adaptive Direct Optimism Control
ParaSol	Parallel Simulation Object Library
PARASOL	PARAllel Silos Oriented Logic simulator
PCI	Peripheral Component Interconnect
PDES	Parallel and distributed Discrete Event Simulation

PIL	Processor In the Loop
PLD	Programmable Logic Device
PMP	Partial Matching Predictor
PO	Predictive Optimism
PP	Physikalischer Prozess
PROM	Programmable Read Only Memory
PSK	Parallel Simulation Kernel
PSP	Previous State Predictor
PTW	Predictive Time Warp
PVM	Parallel Virtual Machine
PWM	PulsWeitenModuliertes Ansteuersignal
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RMI	Remote Method Invocation
ROM	Read Only Memory
ROSS	Rensselaers Optimistic Simulation System
RP	Rapid Prototyping
RPC	Remote Procedure Call
RT	Real Time
RTA	Real-Time Architect, Real Time Analysis Suite
RTAI	Real Time Application Interface
RTC	Real Time Clock
RTI	Real Time Interface, Runtime Infrastructure
RTL	Register Transfer Level
RTOS	Real Time Operating System
RTR	Remote Transmission Request
RTSJ	Real-Time Specification for Java
RTW	Real Time Workshop
S	State (Simulationszustand)
SDB	Shared Database
SDF	Standard Delay Format
SDL	Specification and Description
SE	Simulation Engine
SES	Second Event Synchronization
SIL	Software In the Loop
SIM	SIMulator C++ SIM
SimBa	Simulation Backplane
SIMD	Single Instruction stream, Multiple Data stream
SISD	Single Instruction stream, Single Data stream
SM	Shared Memory
SOAP	Simple Object Access Protocol
SPaDES	Structured Parallel Discrete-Event Simulation
SPEEDES	Synchronous Parallel Environment for Emulation and Discrete Event Simulation
SRAM	Static RAM
SS	State Stack, Synchronisationspunkt
SSHAFT	Simulink to Silicon Hierarchical Automated Flow Tools
STL	Standard Template Library
SW	SoftWare
TCP/IP	Transmission Control Protocol/Internet Protocol
TIP	Target Integration Package
TL	TargetLink
TLC	Target Language Compiler
TT	Terminate Task
TTA	Time Triggered Architecture
TTCAN	Time Triggered CAN
TTP	Time Triggered Protocol
TW	Time Warp
TWOS	Time Warp Operating System
UDP	User Datagram Protocol
UML	Unified Modeling Language
VDX	Vehicle Distributed eXecutive
VHDL	Very high speed integrated circuit Hardware Description Lan-

	guage
VLSI	Very Large Scale Integration
V-Modell	Vorgehensmodell
WAN	Wide Area Network
WCET	Worst-Case-Execution-Time
WDT	WatchDog Timer
WS	WorkStation
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Formelzeichen

A	availability
B	Zeitdauer
Δt	Zeitinkrement, Intervall
e	Ereignis
e_{min}	Ereignis mit der global kleinsten Eintrittszeit t_{min}
E	Menge aller auszuführenden Ereignisse
F_N	Normalkraft
F_S	Seitenführungskraft
F_U	Peripherkraft
F_R	Reibkraft
g	Erdbeschleunigung $9{,}82\frac{m}{s^2}$
G	Garantiezeit
K	Konstante, Kosten, Steigung
L	Lookahead, Leistungsbedarf des Systems, Ereignisliste
λ	Ausfallrate, Schlupf [%]
M	maintainability, Minimum der Zeitstempel aller Ereignisse
m	Message
$m+$	positive Nachricht
$m-$	negative Nachricht
μ_G	Gleitreibungskoeffizient
μ_H	Haftreibungskoeffizient
μ_{HF}	Bremskraftkoeffizient in Bewegungsrichtung
μ_R	Reibungskoeffizient
μ_S	Seitenkraftkoeffizient
$MTBF$	Mean Time Between Failure
$MTTF$	Mean Time To Failure
$MTTR$	Mean Time To Repair
n, N	Anzahl
N_{events}	Anzahl der Ereignisse
N_{states}	Anzahl der Zustände
NP	Anzahl der Prozessoren
ω	Drehfrequenz
P	physikalische Zeit, Prozessor, Partition
$p(HT)$	Wahrscheinlichkeit der Gültigkeit von Hypothesewerten
PK	Preis für die Anbindung des Prozessors an die Kommunikation
PL	Prozessorleistung
PP	Preis für den Prozessor
p_R	Wahrscheinlichkeit für eine Rücksetzung
PV	Preis für die Verbindungen zwischen den Recheneinheiten
R	reliability
r_R	Radradius
s_B	Bremsweg
ST	Space-Time Product
t, t', t''	Zeitpunkt, Zählerwert
T	Simulationszeit, Dauer
τ	Realzeitpunkt
t_{min}	global kleinste Eintrittszeit
T_{Null}	Zeitstempel einer Nullnachricht
v_0	Ausgangsgeschwindigkeit
v_F	Fahrzeuggeschwindigkeit
v_R	Radumfangsgeschwindigkeit
VT	Virtual Time (Simulationszeit)
X^{cont}	kontinuierliche Eingangsgröße
X^{discr}	diskrete Eingangsgröße
Y^{cont}	kontinuierliche Ausgangsgröße
Y^{discr}	diskrete Ausgangsgröße

1 Einleitung

1.1 Motivation

Wachsende Komplexität und Verbreitung eingebetteter Systeme in der Automobilindustrie, aber zum Beispiel auch in der Luft- und Raumfahrttechnik, machen neue, angepasste Entwicklungsverfahren notwendig. Die Größe und Komplexität von computergesteuerten Systemen ist in den letzten Jahrzehnten drastisch angestiegen, da Software (SW) in immer neuen Anwendungsgebieten eingesetzt wird und immer schwierigere Aufgaben unterstützt [Balz96]. Der Entwurf von Steuergeräten (Electronic Control Units, ECUs) für Automobile erfolgt heute unter Einsatz leistungsfähiger CASE-Werkzeuge (Computer Aided Software/Systems Engineering-Werkzeuge). Dabei sind Code-Generatoren eine wichtige Voraussetzung für den durchgängigen Entwurf [DrSc00] sowie der frühzeitige Einsatz eines geeigneten Echtzeit-Betriebssystems (Real Time Operating System, RTOS) auf dem Zielsystem zur Einhaltung zeitlicher Randbedingungen und eine Gesamtsystem-Simulation, damit Aussagen über Qualität und Kosten bereits in frühen Entwicklungsphasen möglich sind und die Markteinführungszeit verkürzt werden kann. Mit Hilfe von Simulation[1] können Abläufe eines realen Systems anhand eines vereinfachten Modells untersucht und Erkenntnisse gewonnenen werden, die Rückschlüsse auf das Verhalten des realen Systems zulassen. Dabei hängt der Grad an Übereinstimmung der Simulationsergebnisse mit dem tatsächlichen Verhalten des realen Systems entscheidend von der Genauigkeit ab, mit der das Modell relevante Aspekte der Realität nachbildet. Je umfangreicher und detaillierter das Modell ist, desto größer ist auch der Rechenaufwand für die Simulation. Darüber hinaus kann der Aufwand zum Beispiel bei einer ereignisorientierten Simulation nichtlinear mit der Modellgröße ansteigen [MaMe89] [Neel87], so dass selbst die schnellsten heutigen Rechner eine lange Ausführungszeit benötigen.

Ursachen für den schnell wachsenden SW-Umfang sind die Zunahme der Fahrzeugfunktionen, ihre Vernetzung, hohe Zuverlässigkeits- und Sicherheitsanforderungen sowie steigende Variantenvielfalt. Wie aus Bild 1 [Bosc05] ersichtlich, hat der Anteil der automatisch codierten Funktionen in der SW-Entwicklung für elektronische Steuergeräte in den letzten Jahren stark zugenommen [ScZu03].

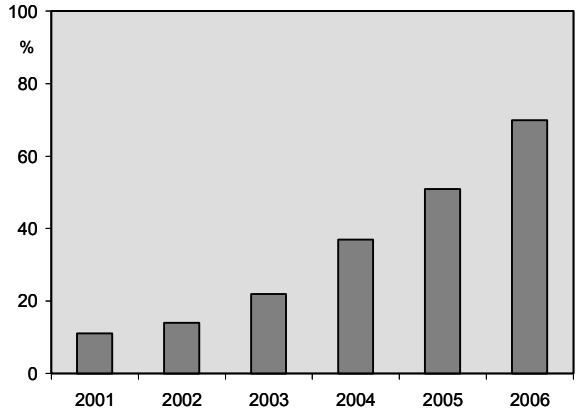


Bild 1: ECU-Anteil der automatisch codierten Funktionen in der SW-Entwicklung

Während sich bei den Motorsteuergeräten die Zahl der Sensoren und Aktoren in den letzten zehn Jahren nicht einmal verdoppelt hat, ist die Anzahl der Code-Zeilen um das 15-fache gestiegen [Bosc05].

Etwa ein Viertel der Herstellungskosten eines Kfz entsteht heute durch Elektrik und Elektronik. Der Ausstattungsgrad mit Elektronik bei Neuwagen ist seit 1990 kontinuierlich gestiegen (Bild 2) [VDI03].

[1] Die Simulation dient zur Verifikation der Implementierung einer Entwurfsidee.

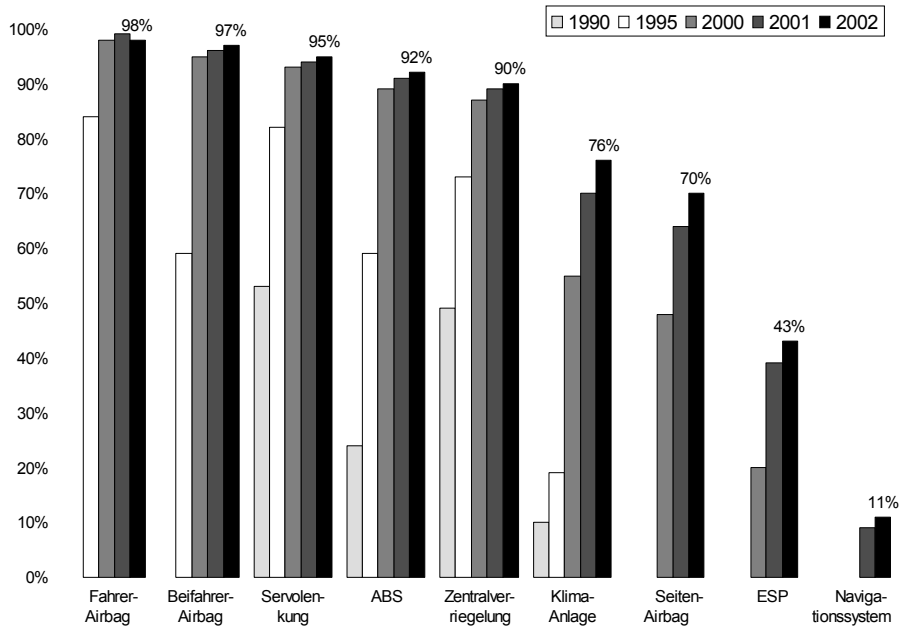


Bild 2: Verbreitungsgrad verschiedener elektronischer Systeme im Kfz

Ein wachsender Elektronikeinsatz im Fahrzeug verspricht einen höheren Grad an Komfort und Sicherheit für Fahrzeuginsassen und andere Verkehrsteilnehmer. Dieser Erwartung steht jedoch die steigende Fehleranfälligkeit des Fahrzeugs bei einer zunehmenden Zahl an elektronischen Komponenten entgegen. So stiegen zwischen 1998 und 2001 die festgestellten Elektrik- und Elektronikpannen um 23 %, während die mechanisch verursachten Pannen nur um 3 % zunahmen [Oswa03]. Eine Untersuchung der Elektrik- und Elektronikpannen zeigt, dass diese häufig durch Fehler der in den Steuergeräten eingebetteten SW verursacht werden [SaLi04]. Nicht nur aus Imagegründen ist es das Ziel der Autohersteller und Zulieferer, solche SW-Fehler in möglichst frühen Entwicklungsphasen zu identifizieren. Je früher ein Fehler entdeckt und dessen Ursache festgestellt wird, desto kostengünstiger kann er behoben werden.

In Bild 3 ist der typische prozentuale Anteil der Projektzeit nach [Berg02] dargestellt, der in jeder Phase des Entwurfszyklus verbracht wurde. Die Kurve zeigt die Kosten, die mit dem Beheben eines Defekts in jeder Stufe des Prozesses verbunden sind.

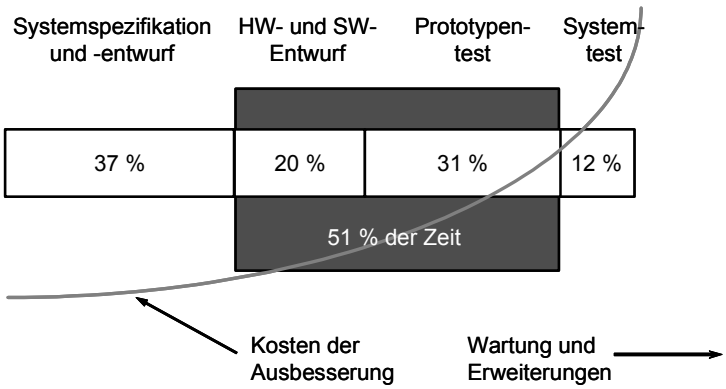


Bild 3: Kosten zum Beheben eines Defekts

Viele Systeme sind nicht adäquat durch ausschließlich zeitkontinuierliche oder durch ausschließlich zeitdiskrete, zustandsbasierte Modelle darstellbar, sondern nur durch hybride System-Modelle. Für eine Gesamtsystem-Simulation ist es aufgrund unverzichtbarer oder spezieller Eigenschaften von CASE-Werkzeugen (einschließlich integrierter Simulatoren), die auch das Zielsystem betreffen können, oftmals notwendig, unterschiedliche Werkzeuge zu verwenden und miteinander zu koppeln, um

eine Co-Simulation[2] durchführen zu können. Solche Eigenschaften sind neben bestimmten Modellierungs-Paradigmen [MGFS04] zum Beispiel die Verfügbarkeit von zum Entwurf notwendigen oder bereits existierenden, getesteten Modell-Bibliotheken, in die CASE-Werkzeuge integrierte Seriencode-Generatoren für das Hardware-Zielsystem und einer Unterstützung bei der modellbasierten Integration eines bestimmten RTOS wie zum Beispiel OSEK/VDX[3] (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug/Vehicle Distributed eXecutive). Abhängig von den eingesetzten CASE-Werkzeugen und den Komponenten des Zielsystems kann eine verteilte Simulation auf einer heterogenen Rechnerplattform basieren, bestehend zum Beispiel aus PC- und Mikrocontroller-basierten Systemen. Das gilt auch, wenn Field Programmable Gate Arrays (FPGAs) Komponenten des Zielsystems sind oder als Mittel zur Simulationsbeschleunigung eingesetzt werden, so dass Teilsysteme durch Emulation auf FPGAs ausgeführt werden.

1.2 Vorgehen und Ziele

Eine Simulation innerhalb eines Rechnerverbundes ist schneller als auf einem einzelnen Rechner mit sequenzieller Von-Neumann-Architektur[4], falls das zu simulierende Modell in geeigneter Weise parallelisierbar ist. Da Zeit zunehmend kostbarer und Hardware (HW) zunehmend günstiger wird, ist die Neu- und Weiterentwicklung von Algorithmen zur Beschleunigung verteilter Simulation erstrebenswert. Vorteile gekoppelter Standardrechner sind z. B. geringerer Preis und bessere Skalierbarkeit[5] im Vergleich zu Spezial-HW mit derselben Rechenleistung. Nachteilig ist die langsamere Kommunikation der Simulationsprozesse über das Local Area Network (LAN) im Gegensatz zur Kommunikation über einen gemeinsamen Speicher[6] (Shared Memory, SM).

Bei der modellbasierten SW-Entwicklung wird auf Basis der Anforderungen ein Modell der zu erstellenden SW entwickelt, das unter Beibehaltung der Eigenschaften und Funktionen des realen Systems dessen Komplexität reduziert. Das ausführbare Modell erhöht die Verständlichkeit und Transparenz innerhalb des Entwicklungsprozesses für alle Beteiligten und ist die technische Grundlage für weitere Aktivitäten im Rahmen der SW-Entwicklung. Mit Hilfe des Modells kann die Überprüfung der Anforderungen frühzeitig erfolgen, d. h. wichtige Tests zur Qualitätssicherung können schon vor der Erstellung des Steuergerätecodes durchgeführt werden. Darüber hinaus lässt sich mit Hilfe von Code-Generatoren aus dem Modell automatisch der ECU-Code erstellen. Ein aufwendiges und fehleranfälliges manuelles Programmieren ist mit Hilfe des Modells nicht mehr erforderlich [SaLi04].

Neben der Frage, wie die Markteinführungszeit durch Beschleunigung der modellbasierten Entwicklung eingebetteter elektronischer Systeme beim Einsatz heterogener Rechnerplattformen, einschließlich der Anwendung von Konzepten wie Hardware-in-the-Loop (HIL), verkürzt werden kann, stellt sich insbesondere auch die Frage, wie sich Simulationsexperimente beschleunigen lassen.

---

[2] Unter einer Co-Simulation versteht man die gemeinsame Simulation mit mehreren Simulatoren.

[3] Führende deutsche Automobilhersteller, Zulieferer und das Institut fur Industrielle Informationstechnik der Universität Karlsruhe schlossen sich zur sogenannten OSEK-Initiative zusammen, die zum Ziel hat, einen Standard für ein echtzeitfähiges Betriebssystem zu definieren, der den speziellen Erfordernissen der Automobilindustrie Rechnung trägt. Eine Initiative mit ähnlichen Zielen wurde unter dem Namen „Vehicle Distributed Executive“ (VDX) in Frankreich gegründet. 1994 schlossen sich die beiden Projektgruppen unter dem neuen Namen OSEK/VDX zusammen.

[4] Eine weit verbreitete Computerarchitektur, bei der ein Strom von Befehlen sequenziell abgearbeitet wird und Befehle auf einzelnen Datenworten arbeiten.

[5] Ein System ist skalierbar, wenn es seine Funktionalität auch bei quantitativen Veränderungen (z. B. die Anzahl an Clients oder Knoten oder eine größere Datenmenge) beibehält.

[6] Gemeinsamer Speicher (shared memory, SM): Alle Knoten nutzen gemeinsam einen Hauptspeicher (wobei jeder Knoten einen lokalen Cache-Speicher besitzt). Verteilter Speicher (distributed memory): Jeder Knoten hat einen eigenen Hauptspeicher und einen eigenen Cache, z. B. über das Intra- oder Internet gekoppelte Arbeitsplatzrechner (Workstation-Cluster) [ScWe04].

Im Folgenden werden zur Gesamtsystem-Validierung sowohl eine Kommunikations-Infrastruktur für eine HIL-Simulation zum schrittweisen Testen der Funktionalität unterschiedlich weit entwickelter Systemkomponenten entwickelt – bei Verwendung der realen Zielplattform und des realen Bussystems (Controller Area Network, CAN[7]) zur Erzielung praxisnaher Ergebnisse – als auch vorhandene Rapid-Prototyping-Systeme (RP-Systeme) ins Gesamtkonzept integriert. Dabei finden zentrale Eigenschaften verschiedener OSEK/VDX RTOS und kommerzieller Seriencode-Generatoren Berücksichtigung, da sich die Eigenschaften der automatisch generierten Quellcodes und der RTOS zum Teil erheblich unterscheiden, so dass sie nicht für alle Applikationen gleichermaßen gut geeignet sind. Darüber hinaus werden als Alternativen zur Ausführungsbeschleunigung zeitdiskreter, zustandsbasierter sowie hybrider System-Modelle HW-Emulation (per FPGAs), Parallelisierung (per Partitionierung und Co-Simulation) und effizientere Co-Simulationsverfahren (per optimistischer Synchronisation) untersucht. Dazu dient u. a. eine im Rahmen der Arbeit entwickelte Java-basierte, plattformunabhängige Co-Simulations-Umgebung, die heterogene Ausführungsumgebungen integriert (PC-, Mikrocontroller- und FPGA-basierte Systeme). Darin sind dedizierte Simulatoren für Logikschaltungen (JLogSim) und Statecharts (JStateSim) eingeschlossen, die optimistische Synchronisation unterstützen sowie ein Werkzeug zur automatischen Generierung von VHDL[8]-Code aus Statecharts zur Konfiguration von FPGAs. Als Voraussetzung für eine parallele und verteilte ereignisorientierte Simulation (Parallel and Distributed Event Simulation, PDES) resultiert ausgehend von Partitionierungsverfahren für Logikschaltungen ein Konzept zur Partitionierung von Statecharts. Weiterhin wird ein existierendes, auf optimistischen Synchronisationsalgorithmen basierendes Verfahren zur verteilten Simulation hybrider System-Modelle verbessert, so dass der Kommunikationsaufwand zwischen den einzelnen Partitionen wesentlich reduziert wird.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt gegliedert:

- Kapitel 1 enthält eine Motivation zur Aufgabenstellung, eine Beschreibung des Vorgehens bei der Arbeit und die Definition der Ziele. Es werden die Bedeutung von CASE-Werkzeugen, Code-Generatoren, Echtzeit-Betriebssystemen und verteilter Simulation von System-Modellen beim Entwurf elektronischer Steuergeräte für Automobile sowie von durchgängigem Entwurf und der Gesamtsystem-Simulation hybrider Systeme erläutert.
- In Kapitel 2 werden die für die Arbeit relevanten Grundlagen zu Systemen dargestellt, einschließlich Echtzeit-Betriebssysteme, eingebettete elektronische Systeme, Abtastung von Signalen, Regelung von Systemen und Anforderungen an die Verlässlichkeit. Es wird auf das Vorgehen bei der Systementwicklung eingegangen, einschließlich Modellbildung, Beschreibungsmittel wie erweiterte endliche Zustandsautomaten und der Analyse des Systemverhaltens mittels Simulation und verteilter Simulation. Weiterhin wird der Einsatz von Mikrocontrollern und FPGAs diskutiert.
- In Kapitel 3 wird der Stand der Technik vorgestellt. Dies umfasst die Bereiche Echtzeit-Betriebssystem OSEK/VDX, Bussysteme im Automobil-Bereich, Einsatz von CASE-Werkzeugen, Partitionierung, verteilte Ausführung von Systemen, Synchronisationsverfahren für die parallele und verteilte ereignisorientierte Simulation, Co-Simulations- und Emulations-Umgebungen und Simulatoren sowie eine Diskussion zum Einsatz der Java-Technologie.

[7] CAN ist eine weit verbreitete, ereignisgesteuerte Kommunikationstechnologie und arbeitet mit dem CSMA/CA (Carrier Sense Multiple-Access with Collision Avoidance) Zugriffsverfahren – einem Multimastersystem mit stochastischem wahlfreien Buszugriff. Bei der Kollision zweier Telegramme erfolgt eine bitweise Arbitrierung. Das Diagramm mit der höheren Priorität setzt sich durch, der niedriger priorisierte Teilnehmer zieht sich zurück [CANB91].

[8] VHDL steht für „Very high speed integrated circuit Hardware Description Language“ und ist neben Verilog die am weitesten verbreitete HW-Beschreibungssprache zur Spezifikation, Dokumentation, Simulation und Synthese von Schaltungen [4] (siehe 3.3.4).

- Das vierte Kapitel stellt den Kern der Arbeit dar. Hier wird der Entwurf des entwickelten Co-Simulations-Frameworks INTERACT einschließlich der zugehörigen Simulatoren beschrieben. Das Framework ermöglicht die Kopplung mehrerer Simulator-Instanzen. Darin eingeschlossen sind der in dieser Arbeit entwickelte Logik-Simulator JLogSim, der Stateflow-Chart-Simulator JStateSim, der Stateflow-Chart-Generator JStateGen, der VHDL-Code-Generator JVHDLGen sowie eine Erläuterung der Funktionsweise des Kontrollprozesses JSim-Control. Es wird ein Vorgehen zur Partitionierung für Statecharts, basierend auf Partitionierungsverfahren für Logikschaltungen, entwickelt, eine optimistische Co-Simulation mit Code, der mittels MATLAB Real-Time Workshop generiert wurde, untersucht, und ein existierender Mechanismus zur Co-Simulation mit optimistischen Synchronisationsalgorithmen verbessert. Weiterhin entsteht eine Infrastruktur für eine Prozessor-in-the-Loop-Simulation zur Echtzeit-Simulation auf dem Zielprozessor unter Einbindung des tatsächlich verwendeten Compilers und automatisch generiertem Code zur schrittweisen Verifikation des zu entwickelnden Systems.
- Kapitel 5 enthält Evaluierungs- und Messergebnisse, die aus der Analyse kommerzieller OSEK/VDX-Betriebssysteme, der Code-Analyse kommerzieller Produktionscode-Generatoren und der Verifikation der Funktionalität des INTERACT-Frameworks resultieren sowie eine Reihe von Performance-Messungen der in Kapitel 4 untersuchten, entwickelten und verbesserten Verfahren. Die dabei verwendeten Modelle werden erläutert.
- Nach der Zusammenfassung der Ergebnisse der Arbeit und einem Ausblick auf sich daraus ergebende weitere mögliche Forschungsschwerpunkte in Kapitel 6 schließt sich der Anhang an, der neben den Verzeichnissen (Literatur-, Abbildungs-, Tabellen- und Stichwortverzeichnis) auch eine Dokumentation zu den interessantesten Aspekten der Implementierungen, die im Rahmen der Arbeit entstanden sind, enthält.

2 Grundlagen

2.1 Systeme

Aus der Sicht der mathematischen Systemtheorie sind Systeme in eine Umwelt eingebettet. Zwischen der Umwelt eines Systems und dem System bestehen Wechselwirkungen, die als Eingabe in das System bzw. als Ausgabe aus dem System bezeichnet werden. Das System besteht aus einer Menge von Komponenten. Für jede Komponente ist eine Menge von Zuständen definiert. Die Gesamtheit der Zustände der Systemkomponenten bildet den Zustandsraum des Systems. Die betrachteten Systeme sind dynamisch oder zeitvariant, d. h. ihre Eingaben, Zustände und Ausgaben ändern sich mit der Zeit. Aussagen über das Verhalten von Systemen sind sowohl für bereits existierende als auch für geplante Systeme von Interesse. Sprachgeschichtlich wird der Begriff System von dem griechischen Wort „systema“ abgeleitet, was soviel bedeutet wie „das aus mehreren Teilen zusammengesetzte geordnete Ganze“ [Hofm55].

Es existieren in der Literatur unterschiedliche **Definitionen für den Systembegriff**, z. B.:

- „A system is an array of components designed to accomplish a particular objective according to plan” [JoKR63].
- Ein System ist die „Zusammenstellung von Einzelteilen oder Komponenten (Gegenständen, Vorgängen, Beziehungen) zu einer neuen Einheit, die durch das Zusammenwirken der Einzelteile mehr darstellt, als nur die Summe der Einzelteile“ [Schu78].
- „Die Ganzheit, die man betrachten will, wird als System bezeichnet, welches möglicherweise Bestandteil eines größeren Supersystems sein kann; Teile des Systems können als Subsystem aufgefasst werden“ [Ulri78].
- Unter dem Begriff System versteht man eine Einrichtung, die auf ein Eingangssignal mit einem Ausgangssignal antwortet [KiJä02].
- J. W. Forrester definiert ein System als eine Menge miteinander in Beziehung stehender Komponenten, die zu einem gemeinsamen Zweck interagieren [Forr72].
- Ein System ist ein natürliches oder künstliches Gebilde, das (mindestens) ein Eingangssignal entgegennimmt und (mindestens) ein Ausgangssignal abgibt [ScWe04].
- Ein System besteht aus verschiedenen Elementen. Die Elemente sind Teile eines Systems und haben bestimmte Merkmale. Zwischen den Elementen bestehen Relationen. Die Elemente und ihre Eigenschaften und Beziehungen bilden eine abgegrenzte Anordnung [MüGl00].
- Nach DIN 44300 T1 und DIN 19226 ist ein System eine Gesamtheit von Objekten, die sich, aufgrund der Beziehungen der Objekte untereinander, von ihrer Umwelt abhebt, davon abgegrenzt erscheint und daher ein als Einheit anzusehendes und gegliedertes Ganzes bildet. Aus Systemen können durch Unterteilen oder durch Zusammenfügen andere Systeme gebildet werden. Als Abgrenzung der so betrachteten Gesamtheit gegenüber ihrer Umwelt lässt sich eine entsprechende Hüllfläche denken, die von allen Beziehungen des Systems zur Umwelt durchdrungen wird. Die Orte der Durchdringung zusammen mit den Beschreibungen der dort geltenden Beziehungen zwischen dem System und seiner Umwelt lassen sich als Schnittstellen auffassen.

2.1.1 Betriebssysteme und Echtzeitsysteme

Definition Betriebssystem: Ein Betriebssystem (BS; Operating System) ist eine zusammenfassende Bezeichnung für alle Programme auf einem Rechner, die die Ausführung, das Zusammenwirken und den Schutz der Benutzerprogramme, die Verteilung der Betriebsmittel auf die einzelnen Benutzerprogramme und die Aufrechterhaltung der Betriebsart steuern und überwachen [DuIn01].

Nach [Gray99] können Computersysteme bezüglich ihres Zeitverhaltens in drei Kategorien eingeteilt werden:

- sequenzielle Systeme (sequential systems), in denen sich die einzelnen Aktionen zeitlich nicht überlappen können;
- nebenläufige Systeme (concurrent systems), in denen sich die Aktionen zeitlich überlappen können;
- Echtzeitsysteme (real-time systems), in denen sich die Aktionen nicht nur überlappen können, sondern auch in einem bestimmten Zeitintervall oder zu einem Zeitpunkt, auftreten müssen.

Ohne Werkzeuge, die über Wissen über das BS verfügen, ist es aufwändiger, BS-Objekte zu untersuchen, und den Status auf BS-Ebene festzustellen. Der Benutzer muss in diesem Fall zum Quellcode zurückkehren und z. B. Anzeigen disassemblieren, um den Speicherort nach Anhaltspunkten wie Alarm-Ablauf zu überprüfen.

Definition Echtzeitsystem: Ein Echtzeitsystem dient zur Steuerung und Abwicklung von Prozessen[9]. Hierbei besteht das System in der Regel aus HW- und SW-Komponenten, und die zu bearbeitenden Prozesse laufen in einer Umgebung ab, mit der das System in dauerndem Informationsaustausch steht. Die Zeitbedingungen versucht man durch Parallelverarbeitung oder verteilte Systeme einzuhalten; dadurch wird die Synchronisation zum vordringlichen Problem. Ein Echtzeitsystem muss unterbrechbar sein, um auf Ereignisse unverzüglich reagieren zu können [DuIn01]. Es müssen bestimmte Aktionen bzw. Reaktionen innerhalb eines vorgegebenen Zeitintervalls erfolgen und Berechnungen innerhalb einer bestimmten Frist abgeschlossen sein, d. h. ein beabsichtigtes Resultat muss innerhalb eines spezifizierten Zeitintervalls produziert werden. Im Gegensatz zu herkömmlichen Applikationen kann das Nichteintreten eines Ereignisses dazu führen, dass es zu einem schwerwiegenden Zwischenfall kommt. Weil Echtzeitsysteme inzwischen in Kraftfahrzeugen oder Flugzeugen Eingang gefunden haben, hat das Versagen derartiger Systeme oft größere Auswirkungen. Die Richtigkeit der Funktion eines Echtzeitsystems hängt also von seinem Verhalten im Wertebereich und im Zeitbereich ab. Da bei der Echtzeit-Verarbeitung Vorgänge in einem System durch den Computer überwacht und gesteuert werden, bestehen besondere Anforderungen an das zeitliche Verhalten, auch unter hoher Verarbeitungslast. Bild 4 zeigt den Zusammenhang zwischen Latenz-, Bearbeitungs- und Antwortzeit [ScWe04].

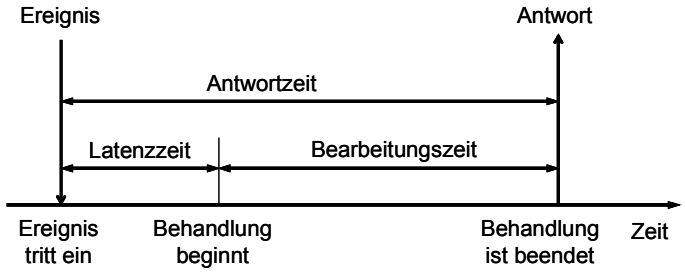


Bild 4: Latenz-, Bearbeitungs- und Antwortzeit

Ein Echtzeit-Computersystem ist meist Teil eines größeren Systems, eines Echtzeitsystems [Kope97]. Nach [IEEE96] wird der Begriff eines Echtzeitsystems wie folgt definiert: „A system in which the correctness of a computation depends not only upon the results of the computations but also upon the time at which the outputs are generated". Echtzeitsysteme können, je nach Anforderungen an das zeitliche Verhalten, in Systeme unter harten Echtzeitbedingungen und Systeme unter weichen Echtzeitbedingungen weiter unterteilt werden.

[9] Ein Prozess besteht aus einer sequenziellen Folge von Anweisungen, die eine abgeschlossene Aufgabe bearbeiten können.

ECUs in Automobilen decken einen weiten Bereich an Funktionalität ab, von der Komfortelektronik bis zur Elektronik für die Kernfunktionen, die die Basiskomponenten eines Automobils steuern. Um die korrekte Ausführung echtzeitkritischer Anwendungen sicherzustellen, werden in vielen elektronischen Systemen Echtzeitsysteme eingesetzt, die auch Unterstützung für die Kommunikation zwischen den ECUs bieten [SiR01b] [SiR01a]. Hinsichtlich der zeitlichen Anforderungen kann das Eintreten von Situationen, bei denen die Sicherheit nicht durch einen Defekt beeinträchtigt wird, bei weicher Echtzeit erlaubt werden, um eine Deadline[10] (einen Fristablauf) einhalten zu können, im Gegensatz zu harter Echtzeit[11] [Zahi99].

Definition Echtzeit-Betriebssystem (real-time operating system, RTOS): Ein Echtzeit-Betriebssystem ist ein Betriebssystem mit zusätzlichen Echtzeit-Funktionen für die Einhaltung von Zeitbedingungen und die Vorhersagbarkeit des Prozessverhaltens.

Definition Task, Deadline, harte und weiche Echtzeit: Eine Aktivität, im Folgenden auch Task[12] genannt, erfüllt harte Echtzeitbedingungen genau dann, wenn es eine feste obere Zeitschranke (im Folgenden Deadline genannt) gibt, an welcher die komplette Task abgearbeitet worden sein muss. Diese Deadline muss immer eingehalten werden, sonst wird die Abarbeitung der Task als fehlerhaft angesehen. Kommt das Ergebnis einer Berechnung auch nur geringfügig später an, als durch die Deadline vorausgesetzt, ist die Information (das Ergebnis) ungültig. Generell müssen sich in einem System alle Tasks mit harten Echtzeitanforderungen an ihre Deadlines halten, sonst befindet sich das System in einem Fehlzustand. In der Literatur [Kope97] werden Echtzeitsysteme meist anhand einer Nutzenfunktion des gelieferten Resultats definiert: Der Nutzen des Resultats, das ein Echtzeitsystem liefert, ist zeitabhängig und nur in einem begrenzten Zeitintervall größer als Null. Bei harten Echtzeitsystemen hat die Nutzfunktion bis zu einem bestimmten Zeitpunkt (der Deadline, s. o.) einen positiven Wert, und für alle späteren Zeitpunkte einen Wert von Null oder kleiner als Null. Ein Nutzen von weniger als Null ist ein Schaden oder bei realen Systemen eine Katastrophe, deren Eintreten verhindert werden muss. Bei weichen Echtzeitsystemen dagegen nähert sich der Nutzen nach der Deadline allmählich an Null an. Zu spät kommende Resultate haben damit zwar einen geringeren Nutzen als rechtzeitige (oder auch gar keinen mehr), sie verursachen aber keine Schäden. Bei weicher Echtzeit ist es nicht schlimm, wenn ein gewisser Anteil der Deadlines nicht eingehalten wird. Im Gegensatz zur harten Echtzeit ist Information, die verspätet eintrifft, immer noch gültig. Das nicht deterministische Zeitverhalten wird dem Fehlzustand durch eine verpasste Deadline vorgezogen. Außerdem können die zeitlichen Anforderungen an Tasks mehrdeutig sein. Anstelle der oberen Laufzeitschranke (worst-case execution time, WCET) treten dann z. B. best, better, worse und worst completion time. Das „rechtzeitige“ Verhalten (auch Timeliness genannt) eines Systems unter harter Echtzeit muss also deterministisch sein. Dieser Determinismus impliziert aber nicht notwendigerweise, dass die eigentlich benötigten Zeiten oder die Reihenfolge, in der die Tasks abgearbeitet werden, im Voraus bekannt sein müssen. Zudem wird oft fälschlicherweise behauptet, ein System arbeite unter harter Echtzeit, wenn es sehr schnell und optimiert arbeitet. Tatsächlich ist die Geschwindigkeit nicht relevant für die Beurteilung, ob ein System hartes Echtzeitverhalten aufweist. Es wird in der Definition auch keine Aussage über die Größenordnung der Zeit getroffen, es kann sich um Mikrosekunden oder Wochen handeln. Wichtig ist nur, dass die Deadlines immer eingehalten werden. Harte Echtzeit ist also eine Aussage über Vorhersagbarkeit, nicht über Geschwindigkeit. Durch den

[10] Eine Deadline legt den Zeitpunkt fest, zu dem ein Ergebnis vorliegen sollte (weiche Echtzeit) oder muss (harte Echtzeit).

[11] Bei harter Echtzeit müssen alle Deadlines unbedingt eingehalten werden. Bei weicher Echtzeit müssen nicht alle Deadlines unbedingt eingehalten werden; es reichen statistische Aussagen, z. B. dass 99 % aller Deadlines eingehalten werden.

[12] Eine Task ist eine sequenzielle Programmeinheit, die aus vier Komponenten besteht, die der Reihe nach ausgeführt werden. Zuerst erfolgt die Kommunikation mit dem Scheduler, danach die Übergabe möglicher Parameter, die Abarbeitung der eigentlichen Aufgabe und schließlich die Ausgabe der Ergebnisse. Innerhalb eines BS hat sie im Folgenden dieselbe Bedeutung wie ein Prozess.

notwendigen Determinismus der Timeliness müssen charakteristische Eigenschaften und Zeiten der beteiligten Tasks und der Umgebung a priori bekannt sein, z.B. die WCET einer Task. Mit diesem Wissen können dann die Ressourcenverteilung und die Ablaufplanung (Scheduling, s. u.) stattfinden, so dass alle Deadlines eingehalten werden.

Ein nicht-automotive-spezifisches Beispiel für den Einsatz von performance- und echtzeitkritischen Anwendungen (weiche Echtzeit) sind Anwendungen im Multimedia-Bereich. Z. B. werden bei einer Videokonferenz die Teilnehmer i. A. an einen zentralen Hochleistungscomputer angeschlossen. Die Hauptaufgaben dieses Computers oder Rechnerverbundes sind Audio- und Videomischung und die Verwaltung der Konferenzen, was einen enormen Rechenaufwand bedeutet, da mehrere komprimierte Ströme gemischt werden müssen, damit wieder ein komprimierter Strom entsteht. Um Kosten zu sparen, kann diese Aufgabe von handelsüblichen PCs erledigt werden, anstatt durch dedizierte Rechner oder Parallelrechner [Ritt99] [Waib99]. Videotelefonie stellt hohe Anforderungen an die Performance. Die Videobilder müssen vor der Übertragung komprimiert und nach dem Empfang dekomprimiert werden, da die Bandbreiten beschränkt sind[13]. Daraus resultiert die Notwendigkeit, die ankommenden komprimierten Videoströme zu dekodieren, die Bilder zu verkleinern, zusammenzusetzen und das entstandene Bild wieder zu kodieren. Die bei der Videokomprimierung und -dekomprimierung verwendeten Verfahren sind rechenintensiv. Ebenso sind die Anforderungen an das Netzwerk aufgrund der geforderten Echtzeitfähigkeit hoch. Echtzeitfähigkeit bedeutet hier, dass die Pakete in begrenzter Zeit übertragen werden müssen, d. h. eine möglichst niedrige Verzögerung bei der Kommunikation, Empfang in korrekter Reihenfolge und Synchronisierbarkeit mehrerer Audio- und Videoströme [Drei99].

Definition Ressource: Unter einer Ressource versteht man ein Betriebsmittel, das von einer Task zur Ausführung ihrer Funktion benötigt wird, z. B. Ein-/Ausgabe-Geräte (E/A-Ports, Kommunikationskanäle) oder Datenstrukturen im Speicher [BeHa01]. Aufgabe der Ressourcenverwaltung ist die koordinierende Vergabe von Nutzungsrechten und die Gewährleistung einer exklusiven Nutzung von Ressourcen.

Wie in Bild 5 dargestellt, kann eine Task die folgenden Zustände annehmen [BeHa01]:

- Ausführung (Running). Die Task wird momentan vom Prozessor bearbeitet.
- Rechenbereit (Ready). Die Task besitzt alle Voraussetzungen zur Bearbeitung, außer der Zuteilung des Prozessors.
- Aktiv. Dieser Zustand umfasst alle Tasks im Zustand Ready und Running.
- Schlafend (ruhend, sleeping). Die Task benötigt momentan keine Rechenzeit.
- Wartend (blockiert, suspendiert, waiting). Die Task wartet auf die Erfüllung einer bestimmten Bedingung, z. B. Eintreffen externer Ereignisse, Zuteilung einer angeforderten Ressource, Empfang einer Nachricht von einer anderen Task, Ablauf einer vorgegebenen Zeitspanne, explizite Reaktivierung durch eine andere Task.

Definition nebenläufige Verarbeitung: Mehrere Vorgänge, Programme, Objekte oder Prozesse heißen nebenläufig, wenn sie voneinander unabhängig bearbeitet werden können (Gegenteil: sequenzielle Verarbeitung). Zwei nebenläufige Prozesse sind nur bezüglich ihrer Abarbeitung unabhängig; sie sind jedoch indirekt voneinander abhängig, wenn sie auf gleiche Betriebsmittel zugreifen wollen. Weiterhin tauschen nebenläufige Prozesse im Allgemeinen auch Nachrichten untereinander aus; durch diese Synchronisation entstehen ebenfalls Abhängigkeiten. Die Benutzung von Betriebsmitteln durch nebenläufige Prozesse wird synchronisiert, d. h. gleichzeitig ausführbare Operationen werden

[13] Eine Möglichkeit, Bandbreite einzusparen, besteht im Einsatz von MPEG-4 (Moving Pictures Experts Group), da hier aufgrund des objektorientierten Szenenaufbaus nicht notwendigerweise das gesamte Bild, sondern nur bestimmte Objekte übertragen werden müssen [DrBS97] [DrSW98] [StBD97] [StDB97] [StDB98].

im Konfliktfall nacheinander (sequenziell) abgearbeitet, und durch weitere Überprüfungen werden Blockierungen verhindert [DuIn01].

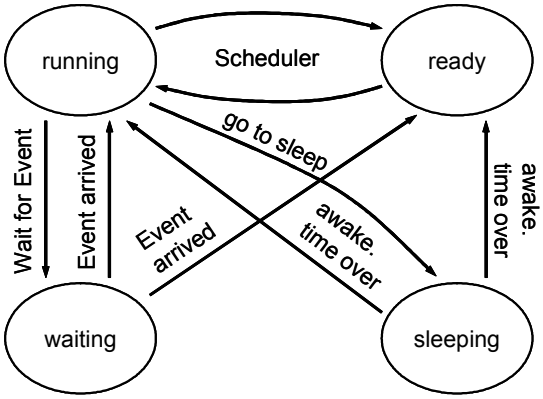


Bild 5: Task-Zustands-Modell eines typischen Echtzeitsystems

Definition Synchronisation: Unter Synchronisation versteht man die Abstimmung nebenläufiger Vorgänge aufeinander [DuIn01], für den Fall, dass zwei oder mehr dieser Vorgänge in gegenseitige Wechselwirkung treten [Schn97].

Definition Scheduling: Existieren mehrere aktive Tasks, muss bei einem Taskwechsel die nächste zu aktivierende Task ausgewählt werden. Dies wird als Scheduling bezeichnet. Die Auswahlroutine heißt Scheduler [BeHa01]. Der Hauptgrund für den Einsatz eines RTOS in eingebetteten Systemen ist die Ablaufplanung[14] (Scheduling) von Applikations-Tasks. Je komplexer die eingebettete Steuer-Software wird, desto nützlicher wird es, einem RTOS die Kontrolle über die Applikation zu geben. Der Scheduler bestimmt die Echtzeit-Fähigkeiten des Systems durch die Verwaltung der Gesamtausführung des Codes, einschließlich Ein- und Ausgabe-Verarbeitung, Datenübergabe zwischen Tasks und Sicherstellen, dass Berechnungen in Echtzeit ausgeführt werden. Der Scheduler muss auch Routinen zur Initialisierung, zum Neustart, und zum Herunterfahren des Controllers zur Verfügung stellen, zusammen mit einer Anzahl Fehlererkennungs- und -behandlungs-Routinen. Es ist wichtig, nicht einen Scheduling-Algorithmus als den besten oder ein Betriebssystem als das beste zu etablieren, sondern herauszufinden[15], welches am geeignetsten ist für die Aktivitäten, die es ausführen soll [Barr02].

Die zeitliche Ablaufsteuerung einer Applikation kann auf verschiedene Art und Weisen erfolgen [BeHa01]:

- zeitgesteuert: Eine Applikation ist zeitgesteuert (time triggered), wenn jede Aktivität des Systems zu vorher festgelegten Zeitpunkten erfolgt;
- ereignisgesteuert (Bild 6 [BeHa01]): Eine Applikation ist ereignisgesteuert (event triggered), wenn jede Aktivität zeitunabhängig (asynchron) durch äußere Ereignisse initiiert wird. Man unterscheidet unterbrechungsgesteuerte (Interrupt-gesteuerte) Ereignissteuerung und Ereignis-Polling.

Mögliche Scheduling-Strategien sind:

- First Come First Serve. Die Tasks werden in der Reihenfolge ihrer Aktivierung vom Scheduler ausgewählt und von der CPU (Central Processing Unit) bearbeitet.

[14] Bei der Ablaufplanung wird die zeitliche Reihenfolge bestimmter Operationen festgelegt. Dabei wird versucht, bestimmten Anforderungen oder Optimierungskriterien gerecht zu werden. Im Falle des Task Schedulings wird entsprechend versucht, die Start- und Endzeitpunkte der Tasks festzulegen.

[15] Wenn ein Echtzeitsystem entworfen anstatt ausgewählt werden soll, muss die Scheduling-Strategie also den Bedürfnissen des Echtzeitsystems besser angepasst werden, als nur die Abarbeitungsreihenfolge der Tasks zu berücksichtigen [Henk96].

- Round-Robin. Alle Tasks werden in einer festen Reihenfolge daraufhin geprüft, ob sie rechenbereit sind. Dies gewährleistet, dass alle aktiven Tasks reihum die CPU zugewiesen bekommen.
- Prioritätsgesteuertes Scheduling. Alle Tasks bekommen eine Priorität zugewiesen, die ihre Dringlichkeit ausdrückt. Der Scheduler wählt unter den aktiven Tasks die mit der höchsten Priorität aus.

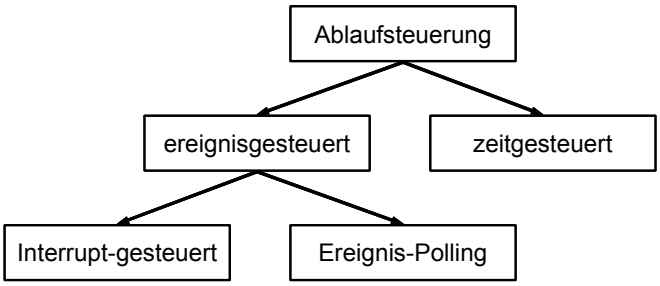


Bild 6: Arten der Ablaufsteuerung von Anwendungen

Definition Unterbrechung: Eine Unterbrechung (Interrupt) ist ein durch ein Ereignis ausgelöster, automatisch ablaufender Mechanismus, der die Verarbeitung des laufenden Programms unterbricht und auf eine Prüfroutine umschaltet. Diese Routine prüft, wie wichtig die Unterbrechung ist, und entscheidet dann, ob das bisherige Programm weiter bearbeitet oder zunächst eine andere Aktivität gestartet wird [DuIn01].

Bei zunehmender Prozessorgeschwindigkeit schrumpft das Unterbrechungs-Fenster, in dem Unterbrechungen deaktiviert sind, und verbessert so die Pünktlichkeit der Antwort, d. h. Unterbrechungen werden schneller verarbeitet. Deshalb kann sich die weiche Echtzeitperformance als ausschließliche Funktion der Prozessor-Geschwindigkeit verbessern. Diesem Trend wirkt die steigende Komplexität von Applikationen entgegen, die mehr Verarbeitung bei Unterbrechungen benötigen [Broo99]. Solche Applikationen sind herstellerunterstützte Werkzeuge wie Debugger, Profiler und Konfigurations-Werkzeuge, die dem Entwickler während der Implementierung und des Testens helfen, Codierfehler wie Speicherlöcher und falsche Speicherzeiger sowie logische Fehler wie z. B. Endlosschleifen und falsche Bedingungen zu finden. Je mehr Werkzeuge unter derselben graphischen Schnittstelle (Graphical User Interface, GUI) bedient werden können, desto geringer wird der Zeitraum für die fachliche Abstimmung der jeweiligen Werkzeuge. RTOS-Unterbrechungs-Routinen und -Gerätetreiber sind schwierig zu debuggen, und ein Fehler in denselbigen kann katastrophale Folgen haben. Einige RTOS versuchen das Potential für eine solche Katastrophe z. B. durch ein Änderungsverbot der Vektortabelle durch den Programmierer einzuschränken. Dies schützt vor Programmierfehlern aber führt zusätzliche indirekte Sprünge ein und führt deshalb zu Overhead und einer Reduzierung der Unterbrechungs-Verarbeitungsperformance. Kenntnis über das BS (OS awareness) ist nicht trivial und erfordert eine enge Integration zwischen der integrierten Entwicklungsumgebung (Integrated Development Environment, IDE), dem RTOS und der Werkzeuge zur Fehlersuche sowie Mittel zur Kommunikation mit solchen Werkzeugen. Deshalb wird ein Framework benötigt, das diesen verschiedenen Komponenten ermöglicht, zusammen zu arbeiten und zu kommunizieren, unabhängig von ihrem Ursprung – RTOS-Hersteller, Dritthersteller von Werkzeugen oder Applikationsentwickler, die betriebseigene Werkzeuge verwenden.

Definition Framework für das Gebiet der Simulation eingebetteter Systeme [Brie01]:
Ein Framework ist

- eine Sammlung von Bibliotheken, Schnittstellen, Werkzeugen und Methoden,
- ein Rahmen zur Entwicklung eines bestimmten Typs von Applikationen,
- ein System, welches Standardaufgaben übernimmt und verbirgt,
- eine Infrastruktur zur Laufzeitunterstützung komponentenbasierter verteilter Simulation.

Definition Multitasking: Unter Multitasking versteht man die Fähigkeit des BS, mehrere Prozesse parallel oder quasi-parallel (verzahnt) abarbeiten zu können.

Die Minimum-Funktionalität, die in einer Echtzeit-Multitasking-Umgebung benötigt wird, ist ein Scheduler und ein Basiselement zur Synchronisation. Alle anderen Synchronisations- und Kommunikations-Primitiven oder Systemaufrufe können darauf basieren. Je mehr Systemaufrufe existieren und je komplexer sie sind, desto weniger Code kann die Applikation enthalten. Bereits existierender Code, der im BS ausgeführt wird, ist wahrscheinlich effizienter und besser getestet vom Hersteller des BS als derselbe Code in der Applikation, wenn diese von Grund auf erst entwickelt wird. Moderne RTOS unterstützen Multitasking auf Basis eines prioritätsbasierten[16] präemptiven Schedulers. Jede konkurrierende Task erhält in einem Anwendungsprogramm eine bestimmte Priorität. Der Scheduler des RTOS stellt sicher, dass die höchstpriore Task, die sich im Zustand „bereit" befindet, in den Zustand „laufend" überführt wird. Er kann ferner eine laufende Task während der Ausführung stoppen, um eine höherpriore auszuführen.

Definition präemptiv, kooperativ und nichtpräemptiv: Zuteilungsstrategien, bei denen das BS dem Prozess den Prozessor entzieht, weil ein anderer Prozess bearbeitet werden soll, heißen präemptiv. In einigen BS muss diese Maßnahme explizit im Programm des Prozesses vorgesehen werden; diese Strategie wird als kooperativ bezeichnet (s. u. und Bild 7 [AuEl99]).

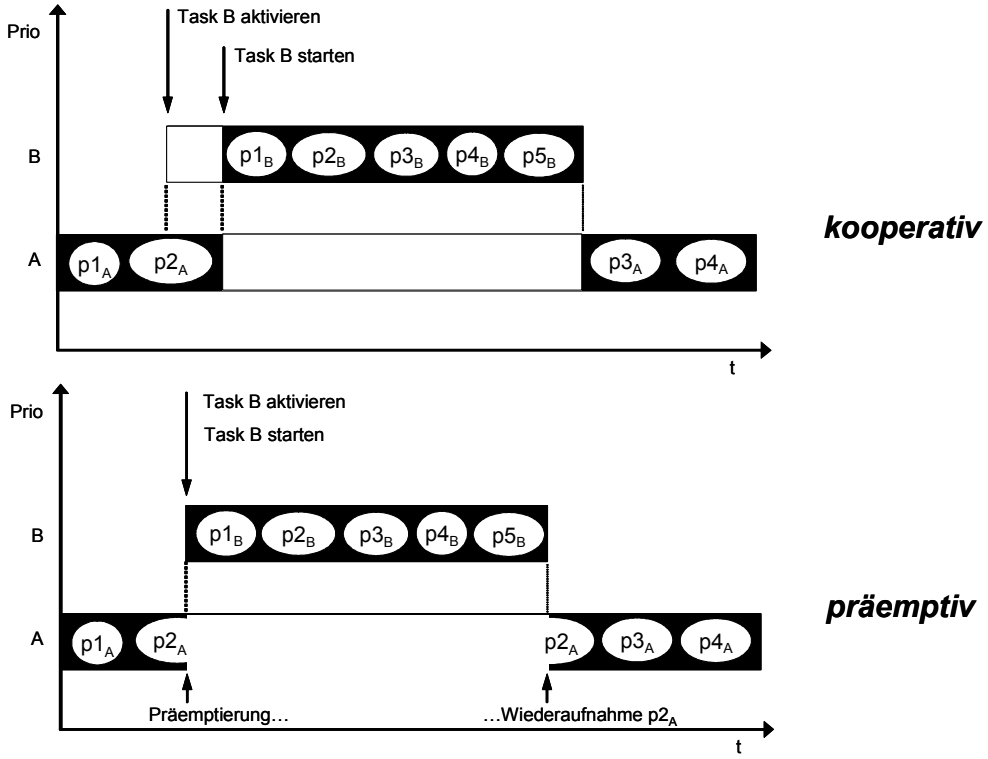


Bild 7: kooperative und präemptive Strategie

Definition Dispatching: Unter Dispatching versteht man den Vorgang des Umschaltens des Programmkontextes von einer Task auf eine andere. Nach dem das Dispatching auslösenden Faktor kann man folgende Unterscheidung treffen [BeHa01]:

- Kooperatives Multitasking. Alle Tasks geben freiwillig in kurzen Abständen die CPU frei.
- Zeitscheibengesteuertes Multitasking. Statt zu warten, bis der laufende Prozess von sich aus den Rechnerkern abgibt, ist es oft sinnvoll, die Laufzeit jedes Prozesses zu begrenzen. Jeder Task wird die CPU für eine bestimmte Rechenzeit (Zeitscheibe, time slice) zugeordnet. Nach

[16] Eine Priorität ist ein Maß für die Wichtigkeit einer Task oder Nachricht in Echtzeitsystemen.

Ablauf der Zeitscheibe wird der Task der Prozessor entzogen und einer anderen Task zugeteilt. Die Größe der Zeitscheiben kann sowohl von Task zu Task als auch zeitlich variieren. Die Steuerung erfolgt mit Hilfe der Systemuhr, die nach Beendigung des vorgegebenen Zeitintervalls eine Unterbrechung auslöst [Schn97].

- Unterbrechnungsgesteuertes Multitasking. Eine Aktivierung von Tasks durch eintreffende Ereignisse stößt gleichzeitig den prioritätsgesteuerten Scheduler zur Neubewertung der aktiven Prozesse an. Wird auch bei internen Ereignissen eine Neubewertung der Priorität durchgeführt, spricht man von präemptivem Multitasking.

Definition Laufzeit: Die Laufzeit eines parallelen Programms ist die Länge des Zeitintervalls, das vom Start des ersten Prozesses bis zur Beendigung des letzten Prozesses definiert ist [RePo99].

Definition Latenzzeit: Die Latenzzeit (latency time) ist die Zeit vom Auftreten eines externen Ereignisses bis zum Start der Behandlungsroutine [BeHa01].

Definition Ausführungszeit: Die Ausführungszeit (Bearbeitungszeit, service time) ist die Zeit zur reinen Berechnung einer Reaktion auf ein externes Ereignis. Sie wird durch den auszuführenden Algorithmus und die Geschwindigkeit des Prozessors vorherbestimmt [BeHa01].

Definition Reaktionszeit: Als Reaktionszeit (Antwortzeit) wird die Zeit bezeichnet, die vom Anlegen von Eingangsgrößen an ein System bis zum Erscheinen der Ausgangsgrößen benötigt wird. Die Frist (Deadline) kennzeichnet den Zeitpunkt, zu dem die entsprechende Reaktion am Prozess (Task) spätestens zur Wirkung kommen muss [BeHa01].

2.1.2 Eingebettete elektronische Systeme

In den letzten Jahren konnte ein stetiger Zuwachs an eingebetteter Elektronik im Automobil beobachtet werden. Der Wert dieser elektronischen Systeme stieg z. B. von 37 Milliarden Dollar 1995 auf 60 Milliarden Dollar im Jahr 2000 an, mit einer jährlichen Zuwachsrate von 10 % [AEE99] [VDA99]. Die Gründe für den zunehmenden Einsatz eingebetteter Systeme sind u. a. die Weiterentwicklung der Informationssysteme und die Erhöhung des Komforts und der Sicherheit. Der zunehmende Einsatz von Elektronik verläuft dabei vor dem Hintergrund fallender Preise und beschleunigter Produkterneuerung. Viele dieser Systeme arbeiten verteilt. Auch der Trend zu einer verteilten Architektur für die Elektronik in Automobilen ist steigend. Die ständig wachsende Bedeutung verteilter Systeme ist auf verschiedene Faktoren zurückzuführen. Zum einen sind die Kosten für die VLSI-Technologie (Very Large Scale Integration) (Prozessoren und Speicherkomponenten) kontinuierlich gefallen, zum anderen sind Netztechnologien mit hoher Bandbreite inzwischen problemlos verfügbar. Aber auch die wachsende Vielfalt von Anwendungen (komplexe Informationsverarbeitung, graphische Benutzeroberflächen) und die schlechten, unberechenbaren Antwortzeiten großer, zentraler Systeme, lassen verteilte Systeme immer attraktiver erscheinen. Der Trend führt also zum Einsatz dezentraler, verteilter Architekturen (siehe 3.5.2.1). Eine verteilte Architektur besteht in diesem Fall aus mehreren Steuergeräten, intelligenten Sensoren und Aktoren, die mittels geeigneter Schnittstellen an einen Bus angeschlossen sind. Dieses Netzwerk ist der Anfang neuer Top-Down Entwurfsansätze um eine höhere Performance, mehr Komfort und größere Sicherheit zu erreichen. Man kann in einer solchen Architektur drei Instanzen unterscheiden: Das Betriebssystem, das die Anweisungen an das Steuergerät ausführt, die Kommunikationssoftware, die für den Datentransport verantwortlich ist und das Netzwerkmanagement.

Ein eingebettetes System übernimmt komplexe Steuerungs-, Regelungs- und Datenverarbeitungsaufgaben in technischen Systemen. Es besteht aus einem Chip oder einer Platine und wird in der Regel für die Steuerung oder Überwachung des technischen Systems eingesetzt, in dem es eingebaut ist. Sein universeller Zweck ist es, eine konkrete Zielapplikation zu steuern oder zu regeln, indem es Steuersignale zu den Aktoren als Reaktion zu den Eingangssignalen schickt. Die Steuersignale werden vom Benutzer und von den Sensoren bereitgestellt, die die relevanten Zustandsparameter des

Systems erfassen. Die Ausführung bedarf in der Regel keiner oder nur geringfügiger Bedienung des Benutzers und erfolgt häufig in Echtzeit [Calv93]. Ein eingebettetes System besteht in der Regel aus HW- und SW-Einheit [Bohn00]. Die dort verwendete SW wird als „eingebettete" SW bezeichnet. Sie ist im Unterschied zur „normalen" SW für den Anwender des Systems nicht sichtbar. Der Benutzer sieht nur die Funktion, ohne feststellen zu können, ob sie z. B. durch einen anwendungsspezifischen integrierten Schaltkreis (Application Specific Integrated Circuit, ASIC[17]) oder durch einen eingebetteten Prozessor mit „festem" Programm bereitgestellt wird.

Definitionen eingebettetes System:

- An embedded system is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a specific function [Barr99].
- Unter einem eingebetteten System versteht man ein Mikrocomputer basiertes System, das als Teil eines technischen Gerätes oder Produktes in dieses integriert ist [Jazd98].
- Unter dem Begriff eingebettete Systeme fasst man alle Systeme zusammen, die einen oder mehrere Computer enthalten, ohne dass diese Computer vom Anwender explizit als solche wahrgenommen werden [BeHa01].

Definition Cross-Entwicklung: Bei eingebetteten Systemen findet die Entwicklung der SW in der Regel nicht auf dem eingebetteten System (Zielrechner) selbst statt, sondern auf einem konventionellen Rechner (Host-Rechner). Aufgrund der geringen Ressourcenausstattung des Zielsystems (Target) ist eine Trennung von Entwicklungssystem (Host) und Zielsystem erforderlich. Das Entwicklungssystem ermöglicht eine komfortable Arbeit bei der Erstellung, Generierung und dem Test der SW. Die Applikations-SW läuft ausschließlich auf dem Zielsystem. Verwenden Entwicklungssystem und Zielsystem unterschiedliche Prozessoren, spricht man von Cross-Compilation. Das fertige Programm wird dann mittels eines Cross-Compilers (ein Compiler, der auf dem Host-Rechner lauffähig ist) erstellt.

2.1.3 Signale und Abtastung von Signalen

Definition Signal: Unter einem Signal versteht man den sich zeitlich verändernden Verlauf einer beobachteten Größe, die eine für den Betrachter relevante Information enthält [KiJä02].

Das Signal wird von seiner physikalischen Trägergröße gelöst und als abstrakte Zahlenfolge dargestellt. Dazu führt die Messperipherie zu diskreten Zeitpunkten Eingabebefehle des Computers (eine Abtastung) aus. Alle zwischen diesen Abtastzeitpunkten liegenden Signalwerte werden vom Computer nicht wahrgenommen, so dass ein zeitdiskretes Signal entsteht. Die Umwandlung des Messwertes in eine Fest- oder Gleitpunktzahl wird durch einen Analog/Digital-Wandler (A/D-Wandler) ausgeführt. Wie Bild 8 zeigt, entsteht ein wertdiskretes Signal, da die Genauigkeit vom gewählten Zahlenformat mit seinem endlichen Wertevorrat begrenzt wird [ScWe04].

Definition zeit- und wertdiskretes Signal: Unter einem zeitdiskreten Signal wird eine aus unendlich vielen Elementen bestehende Zahlenfolge der Form $\{x(i)\} = \{\ldots, x(-1), x(0), x(1), x(2), \ldots\}$ verstanden, deren Argumentvariable i ausschließlich ganzzahlige Werte annehmen kann und diskreten Zeitpunkten $t = t_i, t_i < t_{i+1}$ zugeordnet ist. Sind diese Zeitpunkte äquidistant $[x(i) \rightarrow x(iT)]$, so nennt man die Folgeelemente $x(i)$ auch Abtastwerte, die Konstante T Abtastperiode und ihren Kehrwert $fa = \frac{1}{T}$ Abtastfrequenz. Ein Signal $x(t)$ ist wertdiskret, wenn seine abhängigen Variablenwerte x zu einer endlichen Menge von Zahlen (Wertevorrat) gehören [ScWe04].

Definition Abtastrate, zeit- und wertkontinuierliches Signal: Bei analogen Systemen sind alle auftretenden Signale kontinuierliche Funktionen der Zeit. Dabei ist einem Signal X zu jedem Zeitpunkt t ein Zustand $X(t)$ zugeordnet (Bild 8 a). Solche Signale werden als zeit- und wertkonti-

[17] Ein ASIC ist eine anwendungsspezifische Schaltung, die auf einem Chip integriert wird.

nuierliche Signale bezeichnet. Wird $X(t)$ nur zu bestimmten diskreten Zeitpunkten t_i abgetastet, entsteht ein zeitdiskretes, wertkontinuierliches Signal, das durch die Zahlenfolge $X(t_k)=\{X(t_1),X(t_2),\ldots\}$ mit $k=1,2,\ldots$ definiert wird (Bild 8 b). Die Zeitspanne $dTK=t_k-t_{k-1}$ wird als Abtastrate bezeichnet. Diese kann für alle k konstant sein oder sich ändern [ScZu03].

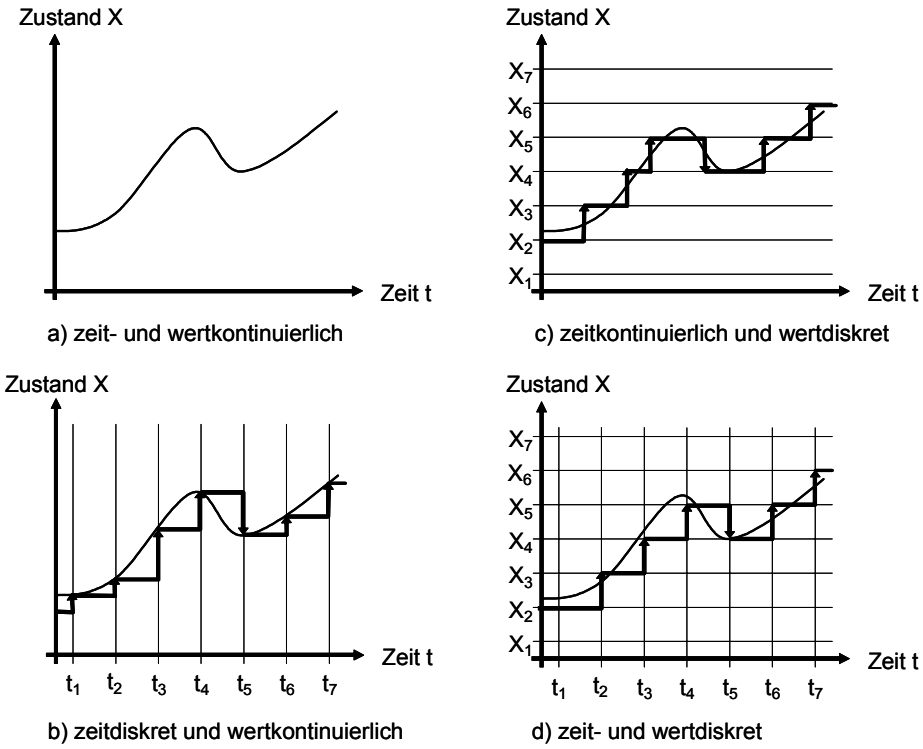


Bild 8: Abtastung eines kontinuierlichen Signals

Definition Quantisierungsfehler: Infolge der beschränkten Wortlänge der für die Erfassung der Eingangssignale eingesetzten A/D-Wandler, entsteht eine Amplitudenquantisierung oder ein wertdiskretes Signal (Bild 8 c). Jedem Zustand $X(t)$ wird genau ein diskreter Wert X_i der Menge $\{X_1,X_2,\ldots,X_n\}$ mit $X_{min}\le X_i\le X_{max}$ eindeutig zugeordnet. Die Differenz $X(t)-X_i(t)$ heißt Quantisierungsfehler. Tritt sowohl zeit- als auch wertdiskrete Quantisierung auf, entsteht ein zeit- und wertdiskretes Signal (Bild 8 d). Alle Größen, die als Eingangsgrößen in einem Programm verarbeitet werden, das auf einem Mikrocontroller einer ECU ausgeführt wird, sind zeit- und wertdiskrete Signale [ScZu03].

Abtasttheorem von Shannon: Wird ein sinusförmiger Signalverlauf der Frequenz *f* mit einer Abtastperiode *T* abgetastet (Abtastfrequenz *fa*), ist im Ergebnis der zeitkontinuierliche Originalverlauf weder eindeutig erkennbar noch wieder rekonstruierbar, sobald *f* die halbe Abtastfrequenz übersteigt. Informationsverluste dieser Art (aliasing) sind somit nur vermeidbar, solange *f* die folgende Bedingung erfüllt [Föll86]: *f < 1/2 fa*.

2.1.4 Steuerung und Regelung von Systemen

Bei einer offenen Steuerung (Bild 9 [ScWe04]) wirken eine Eingangsgröße oder mehrere Eingangsgrößen (Stellgrößen) *y* auf einen Prozess nach bekannten und diesem Prozess eigenen Gesetzmäßigkeiten ein, mit dem Ziel, das Verhalten anderer Ausgangsgrößen *x* in gewünschter Weise zu beeinflussen [BoSe94]. Kennzeichnend sind offene Wirkungswege bzw. offene Signalflusswege (Signalflussgraph ist zyklenfrei). Bei der Führungssteuerung wird der Steueralgorithmus von einem externen Signal (Führungsgröße *w*) geführt.

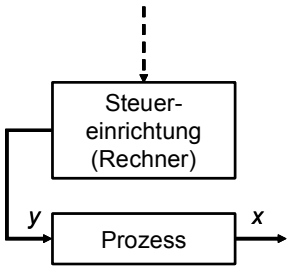


Bild 9: Offene Steuerung

Im Gegensatz zur offenen Steuerung sind bei einer Regelung geschlossene Wirkungsabläufe und zyklische Signalflussgraphen (Regelkreis) kennzeichnend. Die Größe, die zur Darstellung des vorgeschriebenen Prozesszustandes oder -ablaufes verwendet wird, heißt Regelgröße *x*. Der Prozess, dessen Zustand geregelt werden soll, wird Regelstrecke genannt. Grundprinzip aller Regelverfahren ist die Berechnung der Differenz zwischen der gewünschten Führungsgröße *w* und der tatsächlichen Regelgröße *x*, die als Regelabweichung oder Regeldifferenz *e* bezeichnet wird ($e = w - x$). Der berechnete Wert *e* wird dem eigentlichen Regelalgorithmus zugeführt. *z* in Bild 10 heißt Störgröße [ScWe04].

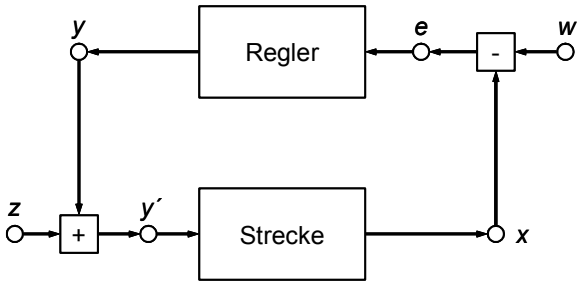


Bild 10: Regelkreis

2.1.5 Anforderungen an die Verlässlichkeit

Die Nachfrage und der Bedarf an immer komplexeren HW/SW-Systemen steigen schneller an als die Fähigkeit, diese systematisch zu entwerfen, zu implementieren und zu testen. Nach H. Balzert steigt die Nachfrage nach SW jährlich zwischen 10 und 25 Prozent, dem steht eine jährliche Produktivitätssteigerung bei der SW-Entwicklung von etwa 4 Prozent gegenüber [Balz96]. Teilweise wird auch von einer Softwarekrise gesprochen, die sich „in abgebrochenen Projekten, maßlosen Zeitüberschreitungen, Nichteinsetzbarkeit von SW, enormen Wartungskosten (über 200 % der Entwicklungskosten) und durch Fehler“ äußert [IIG99]. Hinzu kommt, dass auch viele sicherheitskritische Bereiche, u. a. in der Luftfahrt [DASA00] und im Kraftfahrzeug, aber auch in Kraftwerken und in medizinischen Systemen betroffen sind. Das Ausmaß der Schäden in einem Fehlerfall reicht daher von finanziellen Schäden bis hin zu schweren wirtschaftlichen Schäden, Personenschäden und dem Verlust von Leben. Aus diesem Grund steht der Aspekt der Sicherheit und Zuverlässigkeit von Computersystemen in diesen Bereichen mit an erster Stelle und das Erforschen von geeigneten Methoden und Technologien wird auch in den nächsten Jahren eine Herausforderung für Entwickler und Wissenschaftler bleiben.

Die Anforderungen an die Verlässlichkeit lassen sich in Anforderungen bezüglich Zuverlässigkeit, Sicherheit, Wartbarkeit und Verfügbarkeit untergliedern [Lapr92].

Definition Zuverlässigkeit, Sicherheit, Wartbarkeit und Verfügbarkeit:

- Die Zuverlässigkeit $MTTF$ (reliability) eines Systems ist die Wahrscheinlichkeit, dass ein System während einer bestimmten Zeitdauer gemäß seiner Spezifikation funktioniert. Hat ein System eine konstante Ausfallrate $\lambda = \frac{Ausfälle}{Stunde}$, so kann die Zuverlässigkeit zum Zeitpunkt *t*

wie folgt angegeben werden: $R(t) = e^{-\lambda(t-t_0)}$ wobei $t - t_0$ in Stunden gegeben ist. $MTTF = \frac{1}{\lambda}$ ist der Kehrwert der Ausfallrate – die mittlere Zeit bis zum Ausfall (Mean Time To Failure). Zuverlässigkeit bezeichnet also die Fähigkeit eines Systems, während einer vorgegebenen Zeitdauer bei zulässigen Betriebsbedingungen eine spezifizierte Funktion zu erbringen. Die Zuverlässigkeit gilt als Schlüsselfaktor für Softwarequalität [IEC91], einer Eigenschaft, die auch Funktionalität, Performance, Wartbarkeit, Dokumentation, Effizienz, Benutzbarkeit, Übertragbarkeit und andere Aspekte enthält. Die Zuverlässigkeit ist nicht zu verwechseln mit dem Begriff der Sicherheit, die das Nichtvorhandensein einer Gefahr bezeichnet. Ein System wie z. B. ein Schienenfahrzeug kann im Fehlerfall einfach anhalten; es droht keine Gefahr. Das System gilt als sicher, aber durch den unplanmäßigen Ausfall nicht als zuverlässig. 1994 bemängelte das Bundesministerium für Forschung und Technologie (BMFT), dass die Anwendung von Qualitätssicherungsmaßnahmen trotz anerkannter wissenschaftlicher Ergebnisse noch nicht den Stand erreicht hat, um geforderte Qualitätseigenschaften durch geeignete Methoden und Werkzeuge immer garantieren zu können, obwohl das in sensiblen Bereichen (etwa Flugzeug- und Fahrzeugsteuerung) absolut notwendig ist (Forderung nach Null-Fehler-Software) [BMFT94].

- Sicherheit (safety) ist Verlässlichkeit im Hinblick auf kritische Ausfälle. Ein kritischer Ausfall ist schädlich im Gegensatz zu einem nicht kritischen Ausfall, der als gutartig bezeichnet werden kann. Ein kritischer Ausfall kann eine Gefahr für Material, Umwelt oder Menschen darstellen. Die Kosten eines solchen Ausfalls können um Größenordnungen höher sein als die normalen Betriebskosten des Systems. Eingebettete Systeme finden immer mehr Einzug auch in sicherheitskritische Anwendungen, wie z. B. Antiblockiersystem (ABS) oder Airbag-Systeme im Kraftfahrzeug. Leider stehen bei der Entwicklung sicherheitsrelevanter Systeme Sicherheit und Funktionalität oft gegeneinander, so dass die Entwickler häufig abwägen müssen, wieviel Aufwand sie in Sicherheit und wieviel in Funktionalität eines neuen Produkts investieren [Mont99].
- Die Wartbarkeit (maintainability) ist ein Maß der Zeit, die benötigt wird, um ein System nach einem Ausfall wieder in den Zustand „betriebsbereit“ zu setzen. Die Wartbarkeit wird gemessen durch die Wahrscheinlichkeit *M(t)*, dass das System innerhalb der Zeit *t* nach Eintreten des Ausfalls wieder funktionsfähig ist. Die mittlere Zeit, die für die „Reparatur“ des Systems nach einem Ausfall benötigt wird, wird als Mean Time To Repair ($MTTR$) bezeichnet.
- Die Verfügbarkeit (availability) ist ein Maß der korrekten Funktionsfähigkeit eines Systems beim ständigen Wechsel zwischen funktionsfähigem und nicht funktionsfähigem Zustand. In Systemen mit konstanten Ausfall- und Reparaturraten besteht zwischen $MTTF$, $MTTR$ und A folgender Zusammenhang: $A = \frac{MTTF}{MTTF + MTTR}$. Die Summe $MTTF + MTTR$ wird als die mittlere Zeit zwischen Ausfällen $MTBF$ (Mean Time Between Failures) bezeichnet.

Um Fehler zu reduzieren gibt es drei Möglichkeiten: Vermeiden von Fehlern (Fehlerprävention), Finden und Korrigieren von Fehlern durch Tests und Fehlerbehandlung des Systems während des Betriebs. Oft werden die Fehlerprävention und das Finden und Korrigieren von Fehlern zum Begriff Fehlervermeidung zusammengefasst. Fehler können in allen Phasen des Systementwurfs auftreten (siehe 2.2.1). Die Fehlerbehandlung muss implementiert und selbst wiederum getestet werden, da sich auch hier wieder Fehler einschleichen können.

2.1.6 Ersetzung mechanischer Komponenten durch eingebettete Systeme

Um die Produktionskosten zu reduzieren, die Zuverlässigkeit intelligenter Produkte zu erhöhen, die Sicherheit zu verbessern und Energie zu sparen, werden mechanische Systeme immer häufiger durch eingebettete Systeme ersetzt. Z. B. werden durch die Einführung der X-by-Wire-Technologien im

Automobil mechanische und hydraulische Komponenten im Bereich von Lenkung, Bremse und Antriebsstrang durch rein elektronische Lösungen ersetzt [FESM05], siehe Bild 11 [ElA02a].

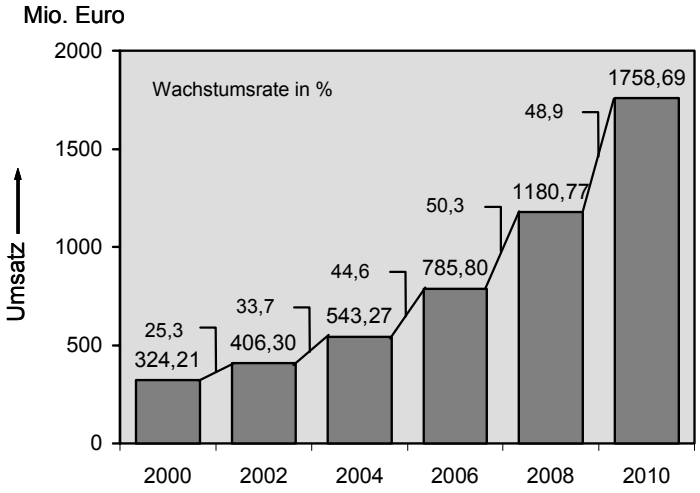


Bild 11: X-by-Wire-Technologie

Ein Brake-by-Wire-System kommt z. B. ohne hydraulisches System aus [Jütt02]. Die heute eingesetzte Hydraulik und Mechanik als Verbindung von Pedalen und Lenkrad einerseits sowie Motor und Fahrwerk andererseits könnte damit in künftigen Fahrzeuggenerationen vollständig durch elektrische Kabel ersetzt werden. Lenksysteme ohne Lenksäule und Bremssysteme ohne hydraulische Leitungen wären die Folge. Systeme dieser Art werden als X-by-Wire-Systeme bezeichnet, wobei das „X“ für eine beliebige fremdkraftbetätigte Funktion wie Lenken (Steer-by-Wire), Bremsen (Brake-by-Wire), Antriebssteuerung (Throttle-by-Wire, E-Gas) steht und „by-Wire“ für ihre elektronische Umsetzung. Werden diese Basissysteme mit entsprechenden elektronischen Sensoren zur Fahrzeugumwelt- und Fahrzeugzustandserfassung ergänzt und durch übergreifende Fahrzeugfunktionen intelligent miteinander verknüpft, spricht man von „Drive-by-Wire“-Konzepten. Fahrzeuge können den Autofahrer innerhalb von Millisekunden bei kritischen Fahrmanövern unterstützen [Vöhr00]. Eine Übersicht zum Entwurf und zur Konfiguration von X-by-Wire-Netzwerken am Beispiel von FlexRay (siehe 3.2) findet sich z. B. in [GaSp01].

2.2 Systementwicklung

2.2.1 Phasen der Systementwicklung

Heute erfolgt die Entwicklung von elektronischen Systemen in mehreren Entwurfsphasen. Üblicherweise beginnt der Entwurf mit der so genannten Analyse- und Spezifikationsphase und geht weiter über die Designphase und die Implementierung bis zur Integrations- und Applikationsphase. In der Analyse- und Spezifikationsphase wird das Verhalten der zu entwickelnden Funktionen mit ihren Anforderungen beschrieben. Hierzu können unterschiedliche Werkzeuge eingesetzt werden – z. B. Office-Werkzeuge, XML-Editoren und semiformale Engineering-Werkzeuge wie DOORS©[18] (Telelogic AB). In der Designphase werden dann formale Verhaltensmodelle entwickelt. Zu unterscheiden sind hier kontinuierliche (regelungstechnische) Systeme, die meist über algebraische Funktionen beschrieben werden, und diskrete (steuerungstechnische) Systeme, deren Beschreibung über Automatenmodelle erfolgt. Für den Entwurf dieser Modelle werden z. B. für diskrete Systeme die Werkzeuge „STATEMATE©“ (i-Logix Inc., [ILog05]), „MATLAB/Stateflow©“ (The MathWorks, Inc.) (Stateflow), „BetterState©“ (Wind River Systems, Inc.) und für kontinuierliche Systeme „MATLAB/Simulink©“ (The MathWorks, Inc.) (Simulink), „MATRIXx©“ (National Instruments Corporation) und

[18] DOORS ist ein Anforderungs-Erfassungs- und -Verwaltungssystem von Telelogic AB, das eine automatische Nachverfolgbarkeit der Anforderungen über das gesamte Projekt ermöglicht [Tele05].

„ASCET©“ (ETAS GmbH & Co.KG) eingesetzt. Nachdem das Verhalten beschrieben wurde und mit Hilfe der vorhandenen Modelle erste Simulationen[19] durchgeführt wurden, folgt die Implementierung. Hierbei entsteht der Code, entweder durch den Entwickler oder mit Hilfe von Code-Generatoren, die Code für die gewünschte Zielplattform erzeugen. Nachdem der Steuergerätecode erfolgreich auf dem Mikrocontroller implementiert wurde, wird das entwickelte System ins Fahrzeug integriert und appliziert [Hofm00].

Der Kernprozess zur Entwicklung elektronischer Systeme ist in Bild 12 [ScZu03] und der Prozess zur Erstellung einer Speicherabbilddatei (image file) für das Zielsystem in Bild 13 [Li03] dargestellt.

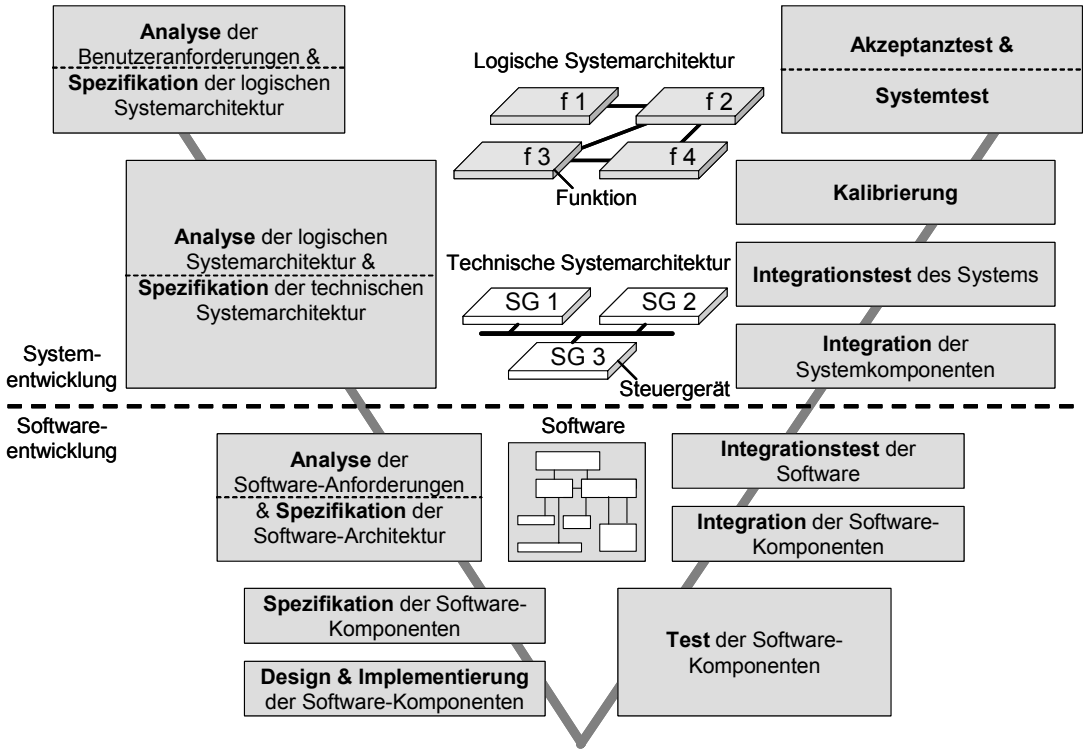


Bild 12: Kernprozess zur Entwicklung von elektronischen Systemen und Software

Die Erstellung eines SW-Produktes umfasst eine Reihe verschiedener Aktivitäten, die logisch aufeinanderfolgen [BeHa01] (Bild 12):

- Problemanalyse und Spezifikation. Diese umfassen die Bestimmung der durch die SW zu erreichenden Ziele und des gewünschten Systemverhaltens einschließlich der einzuhaltenden Randbedingungen.
- Entwurf (Design). Der Entwurf beinhaltet die Erstellung einer Systemarchitektur (Aufteilung zwischen HW und SW). Beim Entwurf wird festgelegt, wie das in der Analyse- und Spezifikationsphase bestimmte Verhalten des zu entwickelnden Systems durch eine Hierarchie geeigneter Bearbeitungsstufen ausgeführt werden soll.
- Implementierung. Aufgabe der Implementierung ist die Umsetzung der spezifizierten Bearbeitungsstufen in lauffähige Programme. Tests der einzelnen Komponenten gewährleisten, dass jede den Anforderungen entspricht.
- Integration und Systemtest. Diese Vorgänge umfassen das Zusammenfügen der einzelnen Programmkomponenten und den Test des kompletten Systems. Dabei ist nachzuweisen, dass die

[19] Simulation ist die Nachbildung der Funktionsweise eines Systems mit Hilfe von Modellen [BeHa01]. Beim Top-Down-Entwurf dient die System-Simulation der Überprüfung der Systemspezifikation auf Konsistenz und Vollständigkeit. Dabei werden die verschiedenen Schaltungsteile durch Blockschaltbilder, Formeln oder Signalverarbeitungsalgorithmen (z. B. in Simulink) spezifiziert [SiSi03].

laut Spezifikation geforderten Eigenschaften, hinsichtlich sowohl funktioneller Korrektheit (Verifikation[20]) als auch der erreichten Leistungen (Validierung[21]), erfüllt werden.

- Wartung. Der Systemlebenszyklus endet nicht mit dem System- und Akzeptanztest (Inbetriebnahme bzw. Abnahme), sondern umfasst den Betrieb mit Wartung ebenso wie den Rückbau. Die Wartung eines SW-Systems umfasst alle Arbeiten an dem System, die nach erfolgter Freigabe an den Nutzer auszuführen sind. Dies betrifft insbesondere die Fehlerbeseitigung, die Verbesserung der Leistung oder anderer Eigenschaften und die Anpassung des Systems an eine veränderte Umgebung.

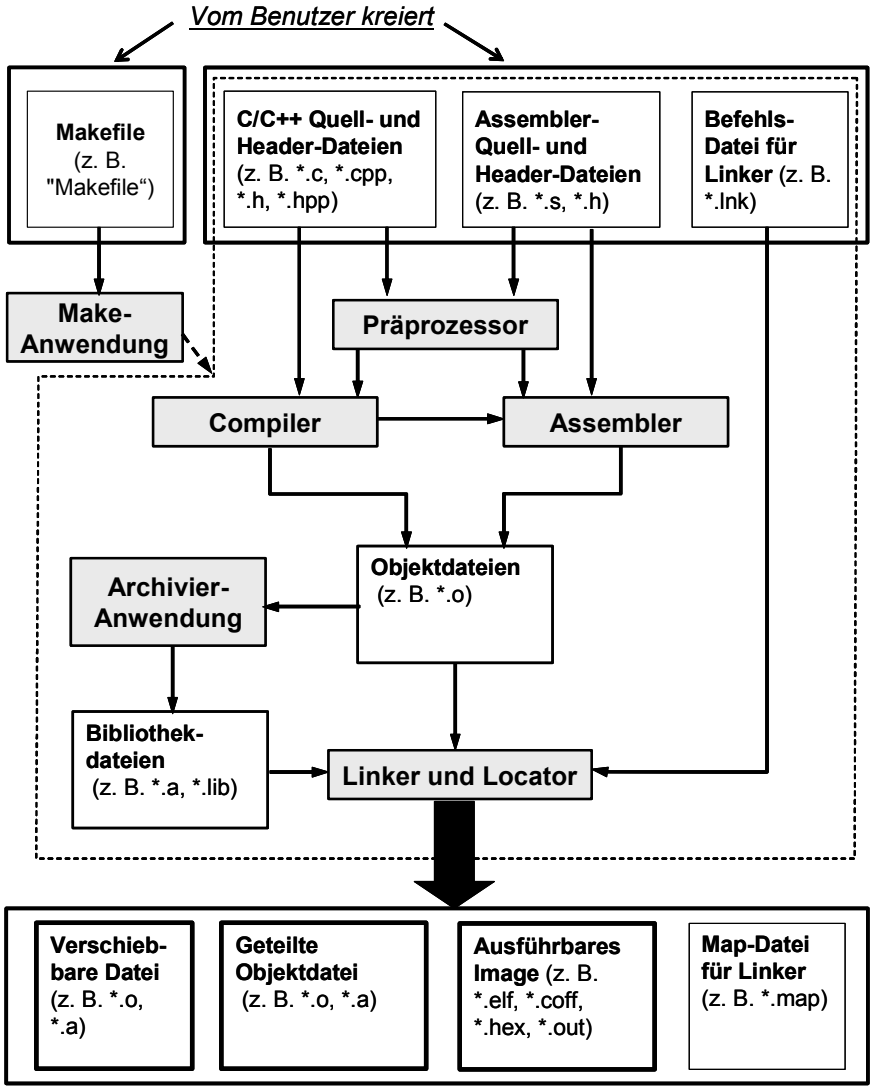


Bild 13: Prozess zur Erstellung einer Bilddatei für das Zielsystem

Zur Darstellung des Zusammenhangs dieser Aktivitäten wurde in der Vergangenheit eine Vielzahl unterschiedlicher Arten von Vorgehensmodellen entwickelt, z. B. das Wasserfall- und das V-Modell [VMod97] [VMXT05]. Solche Vorgehensmodelle sind Leitfäden für die SW-Entwicklung. Das klassische Wasserfallmodell versucht eine Relation zwischen den einzelnen Aktivitäten herzustellen und

[20] Überprüfung der Ergebnisse eines Entwurfsschrittes bezüglich der Vorgaben durch die vorangegangene Phase [BeHa01].

[21] Überprüfung der Ergebnisse eines Entwurfsschrittes bezüglich der initialen Anforderungen bzw. Überprüfung, ob die Spezifikation das Problem korrekt beschreibt.

die Schritte in eine zeitliche Reihenfolge zu bringen. Es unterteilt die SW-Entwicklung in einzelne Phasen und geht im Idealfall von einer getrennten rückkopplungsfreien Abfolge dieser einzelnen Phasen aus. In der Praxis überlappen sich die Phasen, und der scheinbar geradlinige Entwicklungsablauf bildet eine iterative Schleife [BeHa01]. Das V-Modell ist eine Weiterentwicklung des Wasserfall-Modells. Die Entwicklungsphasen auf dem linken Schenkel des V sehen Verfeinerungen im Sinne einer Top-Down-Zergliederung vor; die Phasen auf dem rechten Schenkel stellen Integrationsphasen im Sinne einer Bottom-Up-Integration dar. Das V-Modell erhält in Deutschland eine besondere Bedeutung, da es seit 1991 als verbindlicher Standard bei wehrtechnischen Projekten einzusetzen ist [ScWe04]. Es ist für den gesamten Bereich der Bundesverwaltung verbindlich und wird von vielen Firmen als interner Standard zur SW-Entwicklung verwendet. Der Systementwicklungsprozess wird nur aus funktionaler Sicht beschrieben; es werden keine Aussagen über die konkrete organisatorische Umsetzung getroffen [Calv93] [Vdeu97]. Das V-Modell lässt sich nicht nur auf SW, sondern auch auf Systeme anwenden. Es geht davon aus, dass Anforderungen zu Beginn nahezu vollständig erfasst werden und davon eine hinreichend genaue Spezifikation der technischen Systemarchitektur abgeleitet werden kann. In der Praxis sind Benutzeranforderungen jedoch nicht immer vollständig bekannt und werden während der Entwicklung geändert und ergänzt. Deshalb werden Schritte des V-Modells oder das gesamte V-Modell mehrmals durchlaufen [ScZu03].

2.2.2 Hardware-Entwurf: Y-Diagramm

Definition Entwurf: Unter Entwurf im technischen Sinne versteht man einen Teil des Prozesses, der die Spezifikation in eine technische Realisierung überführt.

Gajski [GaKu83] und andere [WaTh85] führen mit dem Y-Diagramm (Bild 14 [BeHa01], [LeWS-94]) eine schematische Darstellung ein, die sowohl verschiedene Entwurfssichten (Ausgangspunkte) als auch Entwurfsebenen (Hierarchien) darstellt.

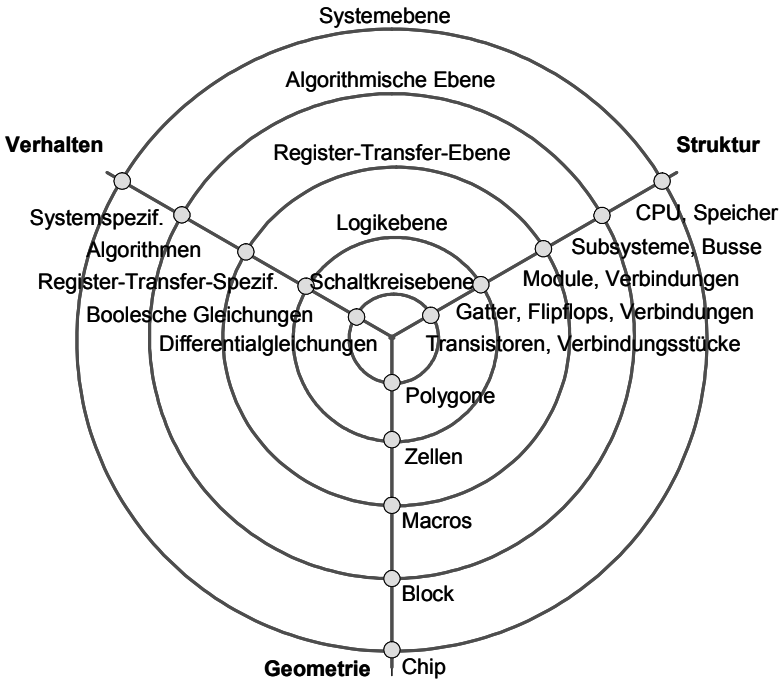


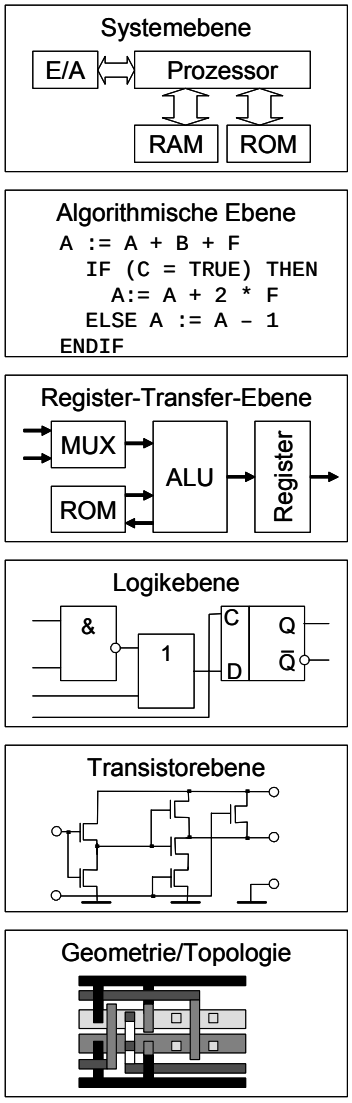
Bild 14: Gajski-Diagramm (Y-Diagramm)

Drei verschiedene Entwurfsansichten sind als Y-artig angeordnete Radialstrahlen dargestellt:

- Verhalten. Das System wird aus Sicht seines Verhaltens über die Zeit betrachtet.
- Struktur. Es ist dargestellt, aus welchen Subsystemen das System zusammengesetzt ist und wie diese miteinander verbunden sind.

- Geometrie. Bei dieser Entwurfssicht werden die geometrischen Eigenschaften des Systems und seiner Subsysteme betrachtet. Für den Entwurf sind vor allem die Realisierbarkeit und Länge von Verbindungen sowie deren relative Lage zueinander entscheidend.

In [Ramm89] wird als zusätzliche vierte Entwurfssicht die Testsicht eingeführt, so dass aus dem Y-Diagramm ein X-Diagramm wird. Konzentrische Kreise mit unterschiedlichen Radien symbolisieren die verschiedenen Hierarchieebenen im Entwurfsprozess, wobei ein großer Radius für einen hohen Abstraktionsgrad steht. Dieser ändert sich, wie in Bild 15 [BeHa01] gezeigt, mit der Entwurfsebene [Gose91]. Für Sichten und Beschreibungsebenen des Schaltungsentwurfs siehe auch [GaDW92].



Blöcke (Speicher, Prozessor(en), Schnittstellen, E/A-Blöcke), Prozesse, Kommunikationsprotokolle

Nebenläufige (parallele) Algorithmen, Funktionen, allgemeine Blöcke, die über Signale miteinander kommunizieren

Operationen, Datentransfer zwischen Registern, Register-Transfer-Module (Register, Multiplexer, ALUs,...)

logische Verknüpfungen, deren zeitliche Eigenschaften, Zusammenschaltung von Makros (Kombinatorik, Flipflops)

Netzwerkgleichungen, Zusammenschaltung von Transistoren, symbolisches Layout

Polygonzüge (für alle technologisch erforderlichen Ebenen)

Bild 15: Entwurfsebenen

Als Modellierungssprache wird insbesondere auf den beiden abstrakten äußeren Ebenen häufig die Programmiersprache C gewählt, da das algorithmische Schaltungsmodell in vielen Anwendungen mit dem Programmcode des digitalen Signalprozessors[22] (Digital Signal Processor, DSP) oder des Mikrocontrollers (µC) kommunizieren muss (eingebettetes System) und eine gemeinsame Sprachbeschreibung die Gesamtmodellierung erleichtert. Auf der Register-Transfer- sowie auf der Logikebene werden überwiegend HW-Beschreibungssprachen wie VHDL [LeWS94] und Verilog [Bhas99] [Veri05] eingesetzt. Diese bieten den Vorteil, dass eine wesentliche Eigenschaft von digitaler Hardware,

[22] Digitale Signalprozessoren sind programmierbare Mikroprozessoren, deren Architektur und Befehlssatz für die schnelle Verarbeitung komplexer Algorithmen der digitalen Signalverarbeitung optimiert sind.

nämlich deren Nebenläufigkeit, durch parallele Prozesse modelliert werden kann. Insbesondere auf der RTL-Ebene werden auch C und C++ im Zusammenhang mit speziellen Klassenbibliotheken eingesetzt (SystemC [Syst05]). Da das Echtzeit-Modell (Real-Time-, RT-Modell) nur VHDL-Konstrukte verwendet, die synthesefähig sind, lässt es sich durch ein Synthesewerkzeug auf die Gatterebene transformieren. Ein weiterer Transformationsschritt, das Place & Route, führt zur geometrischen Beschreibung durch Logikzellen.

2.2.3 Modellbildung

Während man Aussagen über das Verhalten existierender Systeme durch direkte Beobachtung und Messung dieser Systeme erhalten kann, muss man sich für die Ermittlung von Leistungsaussagen über zu entwickelnde Systeme zunächst geeignete Hilfsmittel (Modelle) schaffen. Modelle müssen die wesentlichen Eigenschaften eines betrachteten Systems erfassen. Was dabei wesentlich ist, orientiert sich an den Aussagen, die über das System gewonnen werden sollen [Kepp94].

Der Weg der Modellbildung führt über eine intensive Analyse des zu modellierenden Systems. Da die zu beschreibenden Systeme häufig noch nicht existieren oder für eine exakte Beschreibung zu komplex sind, beschränkt man sich in der Praxis bei der Modellbildung in der Regel auf Modelle, die ein System approximieren. Ein solches Modell beschreibt ein zu analysierendes System bezüglich der zu untersuchenden Eigenschaften hinreichend realitätsnah, so dass sich aus dem Modellverhalten Aussagen über das System ableiten lassen. Im nächsten Schritt müssen die wesentlichen Fakten und die erkannten Gesetzmäßigkeiten systematisch beschrieben werden. Für diese Beschreibung benutzt man eine Modellierungssprache, die eine exakte Definition eines Systems ermöglicht. Das Ergebnis ist ein Modell, für das die Hypothese aufgestellt wird, das es das System bezüglich der relevanten Aspekte des Systemverhaltens hinreichend realitätsnah modelliert.

Real existierende technische Systeme können durch Anwendung geeigneter Methoden in einem Modell dargestellt werden. Mathematische Modelle können in deterministische und stochastische Modelle unterteilt werden, die wiederum in diskrete und kontinuierliche Modellformen unterteilt werden können. Eine mögliche Taxonomie verschiedener Modell-Arten nach [Kepp94] ist in Bild 16 dargestellt.

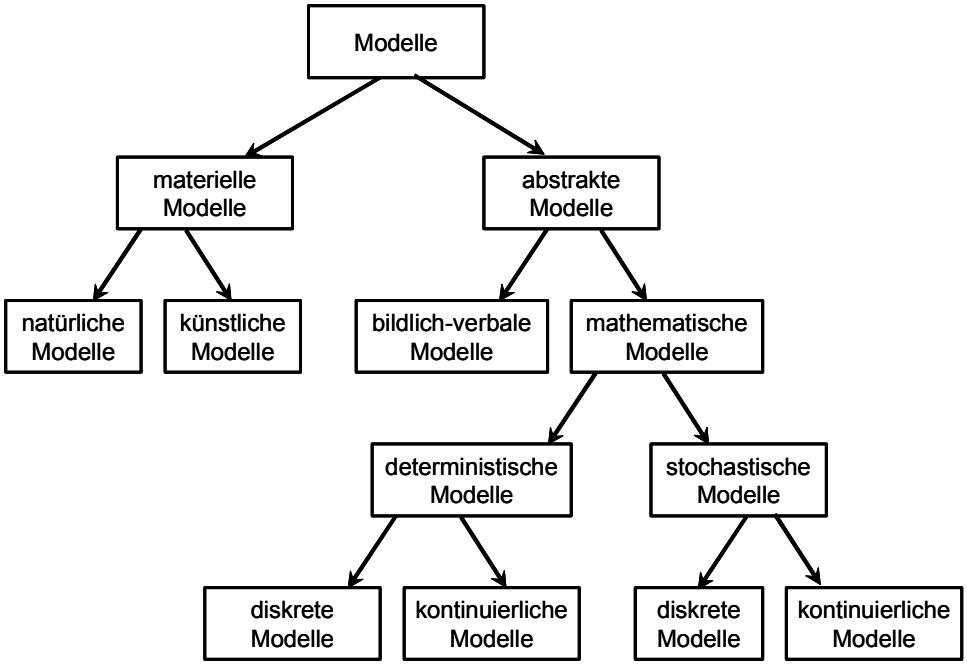


Bild 16: Taxonomie von Modell-Arten

Definitionen Modell:

- Modelle sollen dem Verständnis bzw. der Analyse von Problemen dienen und können auf verschiedenen Ebenen in Form von linguistischen oder graphischen Darstellungen im Rechner implementiert werden. Dabei ist ein Modell als eine ggf. abstrahierte Repräsentation eines (Teil-) Systems aufzufassen. Die Genauigkeit des Modells im Hinblick auf Vollständigkeit und Detaillierungsgrad wird durch das zu untersuchende Problem, den Wissensstand und die Modellumgebung bestimmt [Siem01].
- Unter einem Modell ist ein System zu verstehen, das als Repräsentant eines komplizierten Originals auf Grund mit diesem gemeinsamer, für eine bestimmte Aufgabe wesentlicher Eigenschaften von einem dritten System benutzt, ausgewählt oder geschaffen wird, um letzterem die Erfassung oder Beherrschung des Originals zu ermöglichen oder zu erleichtern bzw. es zu ersetzen [Wüst63].
- Ein Modell ist ein Abbild eines Originals, eines realen Systems. Je nach der Aufgabe des Modells sind unterschiedliche Abbildungsformen (Modellbeschreibungen) eines realen Systems möglich und notwendig, z. B. mechanische, elektrische, mathematische Beschreibungen aber auch Simulationsprogramme [Fish95].

Beschreibungen erfordern spezielle Gestaltungs- und Beschreibungsmittel. Durch das Beschreibungsmittel ist ein Modell maßgebend gekennzeichnet. Die Grundlage vieler Simulationsexperimente bilden mathematische Modelle. Ein mathematisches Modell ist die Gesamtheit formaler Beschreibungen des modellierten Systems, d. h. seiner Struktur und seines Verhaltens. Aus dem mathematischen Modell entsteht durch Hinzufügen von Methoden zur Problemlösung (z. B. numerische Integrationsverfahren) und zum Experimentieren das Simulationsmodell. Es ist die unmittelbare Vorlage für die Gestaltung des Rechenmodells, d. h. das unmittelbar abarbeitungsfähige Programm.

Mit einer Modellierung werden drei Ziele verfolgt: Simulation des Systemverhaltens, automatisierte Synthese des Systems und Dokumentation des Konkretisierungsschritts. Die Eingabe der Modelle erfolgt graphisch durch Editoren, durch die Eingabe von Zustandsautomaten, mittels Blockdiagrammen mit Hilfe von CASE-Werkzeugen wie z. B. Simulink oder durch Eingabe von Quellcode einer HW-Beschreibungssprache [SiSi03]. Ist das System-Modell eindeutig, kann es mit Hilfe von Rapid-Prototyping getestet und simuliert werden oder es ist möglich, automatisch, werkzeugunterstützt Code für Steuergeräte zu erzeugen [ScZu03].

2.2.4 Kontinuierliche und diskrete, zustandsbasierte System-Modelle

Ändert sich der Zustand des Modells stetig mit der Zeit, spricht man von einem Modell mit kontinuierlichen Ereignissen[23] (Events). Das Systemverhalten wird typischerweise durch eine Menge von Differentialgleichungen (Differential Equation System Specification, DESS) beschrieben, deren freie Variable die Zeit ist.

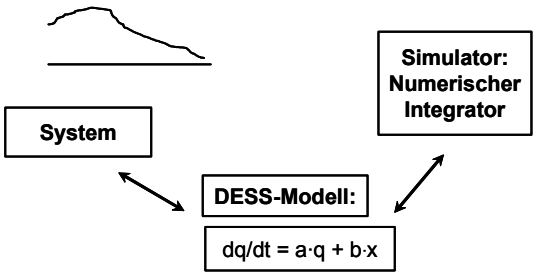


Bild 17: Simulation von DESS-Modellen

Ausgehend von einem Anfangszustand lässt sich der Zustand zu einem späteren Zeitpunkt berechnen. Die Simulation entspricht damit dem Lösen des Differentialgleichungssystems. Kontinuierliche

[23] Ein Ereignis ist z. B. das Auftreten eines Stimulus oder einer Unterbrechung.

Modelle, deren Verhalten mit Hilfe von Differentialgleichungen modelliert ist, können durch Zeitdiskretisierung mittels Differenzengleichungen (Discrete Time System Specification, DTSS) diskreten Modellen angenähert werden (siehe Bild 17 und Bild 18). Hierfür wurden parallele numerische Integrationsalgorithmen entwickelt [Kell88].

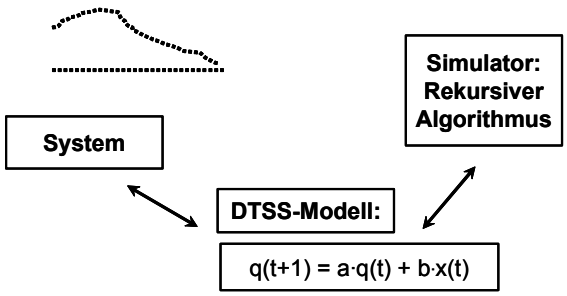


Bild 18: Simulation von DTSS-Modellen

Definition Diskretisierung: Unter Diskretisierung versteht man ein Verfahren, mit dem man bei analytischen Modellen mit kontinuierlichen Zeit- und/oder Ortskoordinaten endlich viele Zeit- bzw. Ortspunkte so auswählt, dass die in ihnen simulierten Prozessgrößen als repräsentativ für das Verhalten der zu simulierenden Prozessgrößen betrachtet werden können und die so gewonnenen Näherungen in einem näher zu definierenden Sinn gegen die wirkliche Lösung konvergieren. Die numerische Mathematik bietet zur Zeitdiskretisierung eine Reihe von Verfahren an, um die Diskretisierung in Gestalt der numerischen Lösungsalgorithmen (solvers) von gewöhnlichen DGLs (Differentialgleichungen) auszuführen (z. B. Euler- und Runge-Kutta-Verfahren) [ScWe04].

Lassen sich den Zustandsänderungen des Modells diskrete Zeitpunkte zuordnen, liegt ein Modell mit diskreten Ereignissen vor. Bei der diskreten Simulation erfolgen die Zustandsänderungen plötzlich und sprunghaft zu diskreten Zeitpunkten. Eine solche Zustandsänderung wird durch das Eintreten eines atomaren, d. h. keine Simulationszeit verbrauchenden, Ereignisses verursacht. Die klassischen Algorithmen zur Simulation diskreter Ereignisse sind entweder zeit- oder ereignisgesteuert. Bei einem ereignisgesteuerten System sind Kommunikation und Prozesse über das Auftreten von Ereignissen gesteuert.

DTSS (Bild 18) ist ein Sonderfall der ereignisdiskreten Systemspezifikation (Discrete Event System specification, DEVS), bei der ein Ereignis pro Zeiteinheit auftritt.

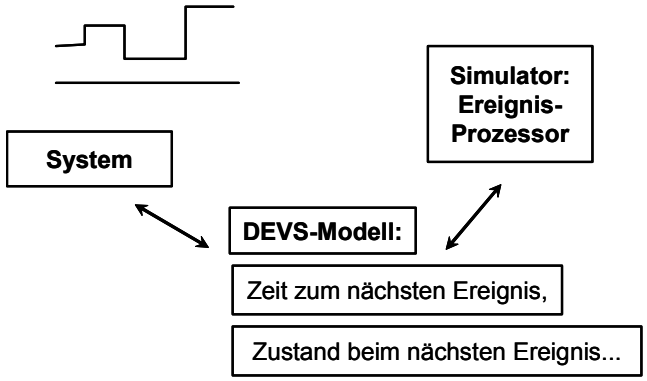


Bild 19: Simulation von DEVS-Modellen

Bei DEVS (Bild 19) ist das zeitliche Voranschreiten zeitauflösungs- oder ereignisbasiert mit bestimmter minimaler Zeitauflösung. Der Wert bleibt bis zum nächsten Ereignis konstant.

Hybride System-Modelle ergeben sich aus der Kombination von DEVS und DESS. Das dynamische Modellverhalten kann u. a. mittels endlicher Automaten des Typs Moore (*Ausgabe = f(Zeit, Zustand)*) oder Mealy (*Ausgabe = f(Zeit, Zustand, Eingabe)*), erweiterter endlicher Automaten (Harel Statecharts) oder Petri-Netzen beschrieben werden (siehe 2.2.5.3). Das System-Modell besitzt dis-

krete und kontinuierliche Eingangs- und Ausgangsgrößen (X^{cont} und X^{discr} bzw. Y^{cont} und Y^{discr}) (Bild 20, Bild 21).

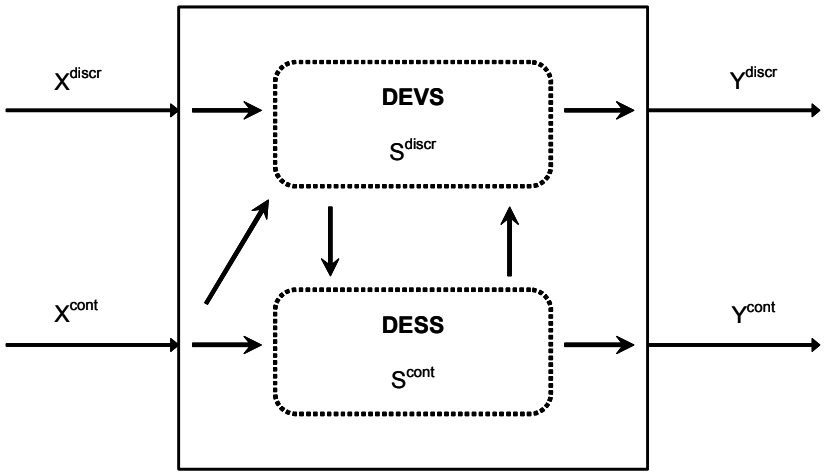


Bild 20: Kombination von DEVS und DESS

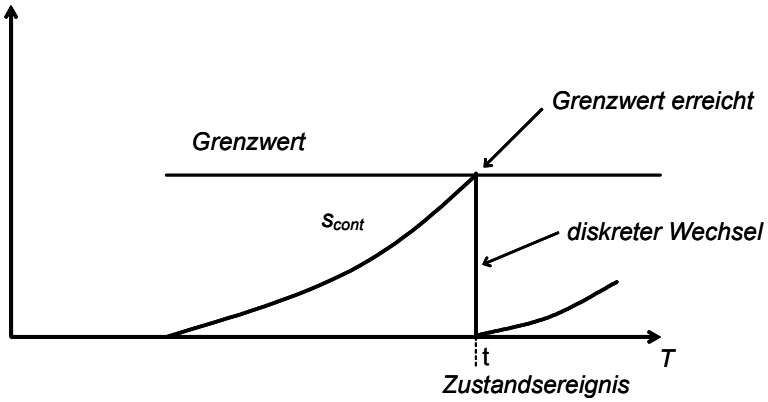


Bild 21: Kommunikation durch Schwellenwert-Erkennung

Definition ereignisdiskrete, zustandsbasierter Darstellung: Unter ereignisdiskreter, zustandsbasierter Darstellung wird nicht eine mathematische Darstellung verstanden, sondern eine Darstellung im Sinne von Zuständen und Ereignissen wie sie bei Statecharts und CASE-Werkzeug-basierter verteilter Simulation zum Einsatz kommt. Die Terme „Ereignisse“ und „Nachrichten mit Zeitstempeln“ werden im Folgenden synonym benutzt.

2.2.5 Beschreibungsmittel

Systeme lassen sich mittels natürlicher Sprache, mathematischer Formeln, Flussdiagrammen, objektorientierter Modellierung und graphischer Modellierung des Datenflusses beschreiben. Zustandsautomaten sind zur Verhaltensspezifikation diskreter ereignisgesteuerter Systeme geeignet und Petrinetze insbesondere zur Verhaltensspezifikation nebenläufiger Prozesse mit Synchronisationsanforderungen.

2.2.5.1 Endliche Zustandsautomaten

Abhängig von der Abstraktionsebene und den unterschiedlichen Sichten kann ein Modell auf unterschiedliche Weisen beschrieben werden. Endliche Zustandsautomaten (Finite State Machines, FSMs) eignen sich zur Spezifikation des Verhaltens diskreter ereignisgesteuerter Systeme. Ein Zustand (state) ist eine Situation, in der sich ein System oder ein Systemteil zu einem Zeitpunkt befinden kann. Hierbei bezieht man sich auf das Innere des Systems und ignoriert die externen Einflüsse wie Ein- und Ausgabewerte. Die Menge der Zustände ist eine Abstraktion eines realen Systems und heißt Zustandsraum [DuIn01]. Ein Automat gibt zu einer Eingabe ein bestimmtes Ergebnis aus. Ein endli-

cher Automat ist ein mathematisches Modell für Automaten, die Informationen Zeichen für Zeichen einlesen, das eingelesene Zeichen sofort verarbeiten und eine Ausgabe erzeugen. Ein endlicher Automat besitzt eine endliche Menge von Zuständen und keinen zusätzlichen Speicher. Mit Hilfe der Zustände kann er Informationen über die bereits eingelesene Eingabe zusammenfassen und diese zur Festlegung seines Verhaltens auf nachfolgende Eingabezeichen verwenden [DuIn01].

In Abhängigkeit davon, ob die Eingangssignale über eine Registerstufe auf Ausgangssignale abgebildet werden dürfen oder nicht, unterscheidet man Mealy-, Moore- und Medvedev-Automat [Siem-99]. Hieraus ergibt sich der grundsätzliche Aufbau eines endlichen Zustandsautomaten gemäß Bild 22 [SiSi03].

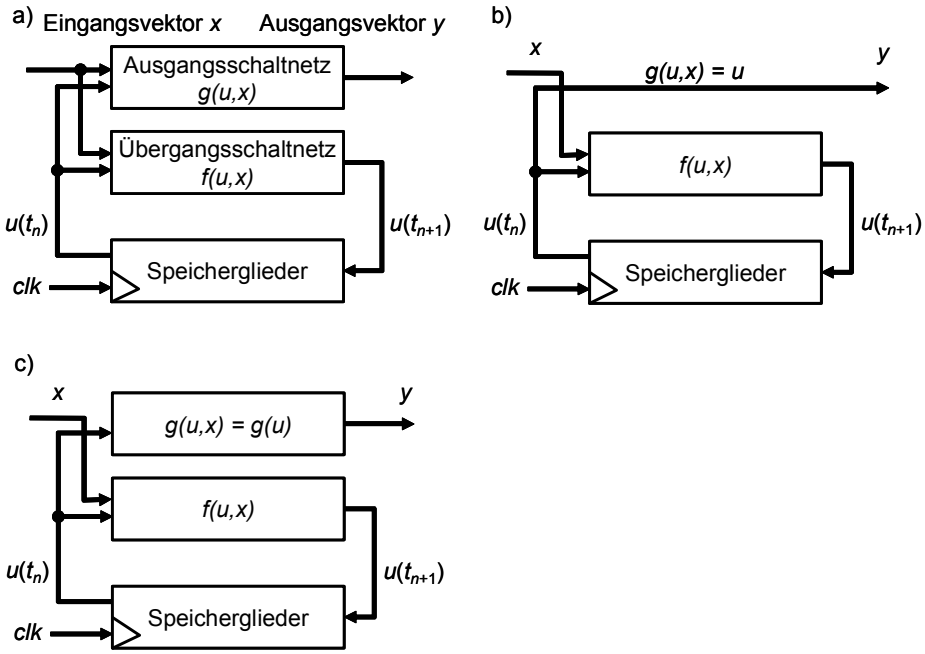


Bild 22: Einfache endliche Zustandsautomaten

In a) ist ein Mealy-Automat dargestellt, b) zeigt einen Medvedev- und c) einen Moore-Automaten.

Definition endlicher Automat:

Ein endlicher Automat A ist bestimmt durch:

- eine endliche Menge U von Zuständen,
- einen Startzustand $u_0 \in U$,
- ein Eingabealphabet X,
- eine Menge $Y \subseteq U$ von Endzuständen,
- eine Übergangsfunktion $f: U \times X \rightarrow U$,
- eine Ausgabefunktion $g: U \times X \rightarrow Y$.

Der Wert der Übergangsfunktion *f(u, x)* gibt den Zustand an, der ausgehend vom Zustand *u* beim Einlesen von *x* erreicht wird.

Eine wichtige Unterklasse ist die Klasse der Automaten mit Ausgabe. In ihr unterscheidet man Mealy-, Moore- und Medvedev-Automaten, mit $A = (U, X, Y, u_0, f, g)$. Ein Moore-Automat ist wie ein Mealy-Automat bestimmt, mit dem einzigen Unterschied, dass die Ausgabefunktion nur vom Zustand abhängt, also $g: U \rightarrow Y$. Ein Moore-Automat, bei dem die Ausgangswerte dem Zustandswert entsprechen, wird als Medvedev-Automat bezeichnet, also g: U = Y.

Graphentheoretisch werden Automaten als Zustandsdiagramme wie folgt dargestellt:

- Knoten = Zustände als Kreis,
- Endzustände als Doppel-Kreis,
- Kanten = Übergänge als Pfeile,
- Kanten beschriftet mit Eingabewort/Ausgabewort,
- Startzustand ist der Zustand mit einem Pfeil, der nicht von einem anderen Zustand kommt oder (Konvention) der erste Zustand ganz links und ganz oben.

Die Nachteile von endlichen Automaten und ihrer graphischen Darstellung für praktische Anwendungen lassen sich in zwei Punkte fassen:

- „Zustandsexplosion“ bei Modellen mit vielen parallelen Abläufen,
- keine Mittel zur Dekomposition großer Modelle.

Aus diesem Grund wurden 1987 von David Harel Statecharts eingeführt [Hare87].

2.2.5.2 Petrinetze

Im Jahre 1962 veröffentlichte Petri eine Netz-Theorie zur Modellierung komplexer Vorgänge, bei denen einzelne Abläufe nicht mehr streng seriell, sondern auch unabhängig voneinander parallel erfolgen können [Petr62]. Petrinetze stellen eine Erweiterung der Zustandsgraphen dar. Während bei Zustandsgraphen genau ein aktiver Zustand zugelassen wird, können bei Petrinetzen mehrere Zustände gleichzeitig aktiv sein, d. h. Petrinetze können dynamisch mehr als einen Folgezustand und mehr als einen Vorgängerzustand haben. Petrinetz-Modelle eignen sich daher insbesondere zur Beschreibung von Abläufen mit nebenläufigen Prozessen und nichtdeterministischen Vorgängen. Verbreitet ist der Einsatz von Petrinetzen für ereignisgesteuerte Vorgänge, für deren Kausalbeziehungen und Nebenläufigkeit ein solches Netz eine besonders geeignete Modellierung abgibt [Reis86]. Ein Petrinetz ist ein gerichteter Graph, der aus zwei verschiedenen Sorten von Knoten besteht: Stellen und Transitionen. Eine Stelle entspricht einer Zwischenablage für Daten, eine Transition beschreibt die Verarbeitung von Daten [DuIn01]. Bei größeren Problemen ist es nicht einfach, die Übersichtlichkeit bei Petrinetzen bei zu behalten.

2.2.5.3 Erweiterte endliche Zustandsautomaten (Statecharts)

Um ein komplexes Verhalten zu modellieren, erweiterte Harel FSMs zu Statecharts, einschließlich Konzepten wie Kombinationen von Mealy- und Moore-Automaten (hybride Zustandsautomaten), Zustandsvariablen[24], Zustandshierarchie, nebenläufigen Zuständen[25] und Kommunikation zwischen nebenläufigen Zuständen (Broadcast) [Hare87]. Den Übergängen, als Pfeile angezeigt, können Ereignisse und Bedingungen zugewiesen werden, die durch ihr Eintreten bzw. Erfülltsein den Zustandsübergang auslösen und dabei eventuell eine Aktion ausführen. Zustände, dargestellt durch Rechtecke mit abgerundeten Ecken, können Unterzustände enthalten, wodurch sich hierarchisch aufgebaute Prozessstrukturen beschreiben lassen. Mehrere nebenläufige Prozesse innerhalb eines übergeordneten Zustands werden durch gestrichelte Linien voneinander getrennt. In Bild 23 ist der Zustand D in nebenläufig ablaufende Zustände unterteilt.

Definition Statechart: Ein Statechart ist ein Zustandsdiagramm, dessen Zustände hierarchisch und parallel zerlegbar sind. Jeder Zustand in einem Statechart ist entweder elementar, durch ein anderes Statechart hierarchisch zerlegt oder parallel zerlegt in mehrere parallele Statecharts.

[24] Unter Zustandsvariablen sind Conditions und Data Items zu verstehen, zusätzlich zu den „diskreten Zuständen“, so dass bedingte und nach Bedingungen verzweigende Transitionen ermöglicht werden.

[25] Wenn nebenläufige Zustände als unterschiedliche Tasks realisiert sind, können sie quasi-parallel auf einer CPU bzw. parallel auf mehreren CPUs bearbeitet werden.

Ein wichtiges Merkmal von Statecharts ist die Möglichkeit zur hierarchischen Gliederung. Dabei gilt u. a.:

- Zustandsübergänge zu einem elementaren Zustand verhalten sich wie bei endlichen Automaten.
- Zustandsübergänge zu einem Zustand, der aus mehreren Statecharts besteht: dieser geht in den Initialzustand (außer ggf. bei Verwendung von History Junctions). Bei mehrstufiger Hierarchie gilt die Regel rekursiv.
- Verlassen eines elementaren Zustands erfolgt wie bei einem endlichen Automaten.
- Verlassen eines Zustandes, der aus mehreren Statecharts besteht: alle, auch hierarchisch in diesem Zustand enthaltenen Zustände, werden verlassen.

In Bild 23 sind die Zustände B1, C, R, U elementar. Die Zustände A, B, D sind in einen oder mehrere Unterzustände gegliedert.

Ein zweites wichtiges Merkmal ist die Möglichkeit, mehrere Zustände nebenläufig auszuführen. Diese Nebenläufigkeit bezieht sich nur auf die Simulationszeit. Bei der Ausführung der Simulation hängt es vor allem von der Rechnerarchitektur ab, inwieweit die Zustände echt parallel abgearbeitet werden. Insbesondere auf Einprozessorsystemen verläuft die Simulation streng sequenziell. Dabei gilt u. a.:

- Die Zustände der nebenläufigen Statecharts S_1,..., S_n bilden zusammen den Zustand Z.
- Beim Zustandsübergang nach Z gehen die Statecharts S_1,..., S_n in ihren jeweiligen Initialzustand.
- Beim Verlassen des Zustands Z werden alle Zustände aller Statecharts S_1,..., S_n verlassen.
- In nebenläufigen Statecharts können parallele Zustandsübergänge stattfinden.
- Zustandsübergänge zwischen nebenläufigen Statecharts sind verboten (graphisch: die gestrichelten Trennlinien dürfen nie von einem Zustandsübergangspfeil gekreuzt werden).
- Nebenläufige Statecharts können sich über Ereignisse durch Broadcasting gegenseitig beeinflussen.

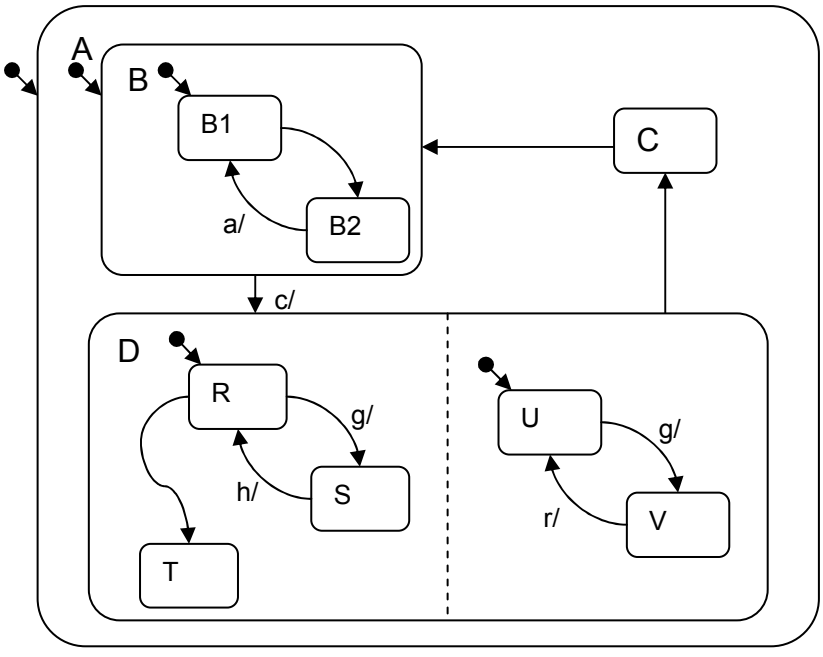


Bild 23: Hierarchisches Statechart

Einzelheiten zu Syntax und Semantik sind [Hare87] [Bort94] und [STAi95] zu entnehmen.

2.2.5.4 Activitycharts

Im Werkzeug STATEMATE [STAT95] stellen Activitycharts die funktionale Beschreibung eines Systems dar. Sie geben Funktionsstruktur und Informationsfluss ähnlich den Daten- bzw. Steuerflussdiagrammen der Strukturierten Analyse wieder (siehe z. B. [Bort94]). Die einzelnen Funktionen oder Prozesse (Activities) eines Systems werden durch Rechtecke repräsentiert, Informationsflüsse durch Pfeile (Flow-Lines). So können die Schnittstellen der Funktionen oder Subsysteme untereinander und nach außen festgelegt werden. Activities lassen sich hierarchisch aufbauen und unterstützen durch die so ermöglichte funktionale Dekomposition den Top-Down-Entwurfsprozess. Ein Activitychart kann ein Statechart als Steuerprozess (Control-Activity) enthalten und durch einen solchen aktiviert und deaktiviert werden. Das Verhalten von elementaren Activities (Basic Activities) wird durch ein Statechart, eine textuelle Beschreibung oder die Einbindung von externem Code festgelegt. Über die Flow-Lines lassen sich Ereignisse, Bedingungen und sonstige Daten zwischen bestimmten Activities austauschen. Wird ein Activitychart mehrfach verwendet, kann es als generisches Activitychart festgelegt werden, dessen Ein- und Ausgänge durch formale Parameter – ähnlich wie bei Prozeduren herkömmlicher Programmiersprachen – repräsentiert werden. Diese lassen sich dann bei der Instanziierung des generischen Charts durch die aktuellen Datenelemente ersetzen. In [Bort94] und [STAi95] sind detaillierte Beschreibungen der Activitycharts zu finden.

2.3 Analyse des Systemverhaltens eingebetteter Systeme

Es existieren verschiedene Analysemöglichkeiten für eingebettete Systeme, z. B.:

- Eine Möglichkeit zur Analyse eingebetteter Systeme ist die Interpretation und Analyse des Quelltextes der Anwendung. Hierbei wird der Versuch unternommen, anhand des Quelltextes auf die Laufzeit des resultierenden Programms zu schließen. Die Quelltextanalyse liefert im besten Fall eine WCET für das analysierte Teil- (System). Die tatsächlich auftretenden Laufzeiten können aber auch erheblich unter der WCET liegen [Brie01].
- Die formale Verifikation wird nicht auf das eingebettete System selbst angewendet, sondern arbeitet auf einer abstrakten Ebene. Werkzeuge wie STATEMATE und Rhapsody© von I-Logix sind in der Lage, aus den Modellen Programme oder Programmfragmente zu generieren. Die Überprüfung des Entwurfs auf der Ebene des Beschreibungswerkzeugs liefert bei diesen Werkzeugen eine Überprüfung der SW für das eingebettete System. Die Aussagekraft über das Zeitverhalten auf einer konkreten HW, genauso wie die Einbeziehung der Umgebung eines eingebetteten Systems, ist allerdings begrenzt [Brie01].
- HIL, SIL und MIL (siehe 2.3.1).
- Emulation (siehe 2.3.1).
- Bei der manuellen Spurverfolgung (tracing, Code-Instrumentierung) werden Ablaufdaten des eingebetteten Systems über eine Schnittstelle ausgegeben und auf einem separaten Rechner analysiert. Die Ablaufdaten werden durch spezielle Anweisungen in der SW des eingebetteten Systems generiert und nach außen transportiert. Dabei wird CPU-Zeit verbraucht und somit das Laufzeitverhalten des Systems beeinflusst. Darüber hinaus müssen die dafür zusätzlich notwendigen Anweisungen eingefügt und später wieder entfernt werden.
- Simulation.
- HW-Spurverfolgung mittels einer speziellen Zusatz-HW. Im Vergleich zur Emulation wird nicht nur der µC ausgetauscht, sondern Logik-HW zur Beobachtung auf dem Zielsystem integriert und im Feldeinsatz im Zielgerät belassen.

Eine Bewertung von Analysemöglichkeiten findet sich z. B. in [Brie01].

2.3.1 Analyse von Systemen mit Hilfe von In-the-Loop-Methoden

Das erfolgreiche Testen jedes Teilmodells impliziert keine umfassende Modellglaubwürdigkeit. Deshalb muss das gesamte Modell getestet werden, auch wenn jedes Teilmodell als ausreichend glaubwürdig eingestuft wurde.

Die entwickelten und verfeinerten Methoden sowie rechnergestützten Werkzeuge des Rapid Prototyping und Hardware-in-the-Loop tragen, eingebunden in eine durchgängige Entwicklungsmethode und -umgebung, maßgeblich zu einer Verkürzung der Entwicklungszeit und zur Erhöhung der Entwurfsqualität bei [MGBS99] [MüBS00]. HIL-Tests bieten die Möglichkeit, die Integration des ausführbaren Programms mit BS-Komponenten auf der Ziel-HW zu prüfen [TaBD03]. Der wesentliche Vorteil des Model-in-the-Loop (MIL)-Tests liegt in der Möglichkeit, System- und Funktionsspezifikation bereits im Prozessschritt Spezifikation zu testen. Software-in-the-Loop (SIL)-Tests werden als erste Tests des quantisierten Integer-Codes des übersetzten und ausführbaren Programms eingesetzt, unmittelbar nach oder während der Programmierung. Interne SW-Größen sind einfach zugreifbar, da die zu testende SW in eine Testumgebung eingebettet ist. HIL-Tests bieten im Gegensatz zu MIL- und SIL-Tests zusätzlich die Möglichkeit, die Integration des Codes mit BS-Komponenten und Treiber-SW auf der Ziel-HW zu prüfen [SaHS05].

Die In-Circuit-Emulation dient zum hardwareunterstützten Simulieren in Realzeit (Emulieren) und Austesten eines Zielsystems, das sich in Entwicklung befindet. Dabei werden die zu testenden Funktionen des Zielsystems, insbesondere der Prozessor und der Speicher, nachgebildet. Dazu wird der Prozessor durch einen In-Circuit-Emulator (ICE) ersetzt, der durch einen Entwicklungsrechner gesteuert wird. Der ICE kann zusammen mit dem Entwicklungsrechner dieselben Funktionen wie die ursprüngliche CPU des Zielsystems unter denselben Bedingungen ausführen [Schn97].

2.3.2 Analyse durch Simulation und Co-Simulation

Simulation ist eine Analysemethode, die es erlaubt, Abläufe eines realen Systems anhand eines vereinfachten Modells zu untersuchen. Ein Simulationsmodell spiegelt die wesentlichen Eigenschaften der zu simulierenden Vorgänge und ihre gegenseitige Beeinflussung wider. Die dabei gewonnenen Erkenntnisse lassen in gewissen Grenzen Rückschlüsse auf das Verhalten des realen Systems zu [Page91]. Alle Ergebnisse der Simulation beziehen sich nur auf das Modell. Inwieweit solche Ergebnisse auf die Wirklichkeit übertragen werden können, hängt daher entscheidend davon ab, wie gut die Wirklichkeit durch das Modell nachgebildet wird. So können Vorgänge untersucht werden, die in der Realität zu langsam oder zu schnell ablaufen und sich dadurch oftmals einer praktikablen Untersuchung entziehen. Es lassen sich aber auch Vorgänge analysieren, deren Untersuchung in der Realität zu teuer, gefährlich oder prinzipiell undurchführbar wäre. Darüber hinaus sind belastungsfreie Beobachtungen interner Systemknoten[26] möglich. Auch lässt sich mittels Simulation das Verhalten eines realen Systems approximativ vorhersagen, wodurch Simulation als Entscheidungshilfe interessant wird. Die Flexibilität von Simulationen zeigt sich auch daran, dass sie sich als Mittel zur Analyse komplexer Systeme noch einsetzen lassen, wenn andere, wie z. B. mathematisch-analytische, Methoden, bereits versagen [Neel87]. Ein weiteres Einsatzgebiet von Simulationen sind computergenerierte virtuelle Welten, in die Menschen und Ressourcen eingebettet werden. Die Qualität eines Simulationsergebnisses ist nicht nur vom Modell abhängig, sondern auch von der Erfahrung des Entwicklers, der weiß, welche Stimuli an das Modell zu legen sind und die Grenzen der zu Grunde liegenden Modelle kennt. Ist die korrekte Abbildung zwischen einem realen, dynamischen System und dem Modell gesichert, lassen sich die Abläufe des realen, dynamischen Systems im Modell nachvollziehen und Kenntnisse über das Modellverhalten sammeln, die schließlich auf das reale, dynamische System übertragen werden können. Zur Durchführung von Experimenten ist ein ausfüh-

[26] Ein Knoten ist eine Rechnereinheit im verteilten System, bestehend aus dem Host, seiner Applikation und einer Netzwerkanbindung.

rendes Gerät, bei Simulationsexperimenten ein Simulator, erforderlich. Ist der Simulator ein Computer einschließlich geeigneter SW, spricht man von Computersimulation [Grüt02].

Weitere Definitionen für Simulation:

- Simulation ist das Nachbilden eines Systems mit seinen dynamischen Prozessen in einem experimentierbaren Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind [VDIS00]. Wesentliche Bestandteile dieser Definition sind das Hervorheben der Systemdynamik (Modellierung der Zeit, Abbildung von Synchronisationen und Nebenläufigkeiten) sowie die Tatsache, dass das zu erstellende Modell zum Zwecke des Experimentierens entwickelt wird, damit die Ergebnisse aus den Experimenten am Modell auf das geplante oder reale System übertragen werden können [Wenz02].

Zwei Klassen von Simulations-Applikationen wird die meiste Aufmerksamkeit entgegengebracht: analytischen Simulationen und virtuellen Umgebungen (Tabelle 1) [Fuji00].

	Analytische Simulationen	Virtuelle Umgebungen
Voranschreiten der Ausführung	typischerweise so schnell wie möglich	in Echtzeit
Typisches Ziel	quantitative Analyse komplexer Systeme	erschaffen einer realistischen und/oder benutzbaren Repräsentation der Umgebung
Menschliche Interaktion	falls miteinbezogen ist der Mensch ein externer Beobachter des Modells	Menschen sind integraler Bestandteil und steuern das Verhalten von Modellelementen
Vorher-nachher-Beziehungen	es wird versucht, vorher-nachher-Beziehungen genau zu reproduzieren	Werden nur in dem Maße reproduziert, dass Menschen oder physikalische Komponenten, die in der Umgebung eingebettet sind, diese wahrnehmen können

Tabelle 1: Analytische Simulationen und virtuelle Umgebungen

Virtuelle Umgebungssimulationen mit menschlichen Teilnehmern (als Entitäten innerhalb der Simulation) werden als Human-in-the-loop-Simulationen bezeichnet, und Simulationen, die eingebettete physikalische Geräte miteinschließen, als HIL-Simulationen [Ledi01].

Bei einer Simulation wird ein bereits existierendes oder geplantes System im Modell nachgebildet, um Aussagen über das System machen zu können [BrFS87] [Fish78]. Um Experimente mit einem solchen Modell durchzuführen, wird das Modell in einem Programm spezifiziert. Die Durchführung des Experiments besteht in der Ausführung des Pogramms, d. h. in der Simulation der Ereignisse des Modells. Die Ausführung des Programms wird als Simulationslauf bezeichnet. Die Modellauswertung besteht in der Analyse der Ergebnisse, die sich durch Ausführung des Simulationsprogramms ergeben. Typische Schritte für das Erstellen und den Gebrauch von Simulationsprogrammen sind [Page91] [Spie82] [VoWo88]:

- Systemanalyse: Erfassung der Wechselwirkungen der Komponenten des zu simulierenden Systems;
- Modellbildung: Modellierung des zu simulierenden Systems unter Beachtung der wesentlichen Aspekte des Systemverhaltens und Vernachlässigung irrelevanter Details;
- Programmspezifikation: Abbildung des Modells auf ein Simulationsprogramm;
- Experimentalphase: Ausführung des Simulationsprogramms und Analyse der Ergebnisse.

Unter der digitalen Simulation versteht man diejenigen Simulationsarten, die an den mathematischen Modellen realer, dynamischer Systeme mit Hilfe von Computeranlagen durchführbar sind. Die unter Nutzung von Simulationswerkzeugen zu erstellenden Simulationsmodelle werden über eine Simulationsmethode und über ein Modellierungskonzept charakterisiert.

Man unterscheidet zwischen deterministischer und stochastischer Simulation. Bei der deterministischen Simulation sind alle an dem Modell beteiligten Größen exakt definiert oder aufgrund mathematischer Zusammenhänge aus Anfangswerten berechenbar. Bei der stochastischen Simulation werden im Modell auch zufallsabhängige Größen verwendet [DuIn01].

Definition Co-Simulation: Unter Co-Simulation wird die gemeinsame Simulation mit mehreren Simulatoren verstanden. Der Begriff Simulator umfasst hierbei alle Arten von SW und SW-Komponenten, die ein Modell ausführen können, z. B. einen Teil eines Werkzeugs, das auch zur Modellierung verwendet wird, oder insbesondere auch aus einem Modell generierten Code [ErST94].

Die Berücksichtigung von Interaktionen und der verschiedenen physikalischen Disziplinen erfordert ein durchgängiges „multidomain"-Designkonzept. Um ein möglichst vollständiges Bild vom Gesamtsystem zu bekommen, ist eine Kombination traditionell nur separat verfügbarer Simulationsalgorithmen innerhalb einer Simulationsumgebung erforderlich. Das Prinzip der Co-Simulation ermöglicht die gleichzeitige Verwendung verschiedener Simulationsalgorithmen, wie sie am besten für eine spezifische Simulationsaufgabe geeignet sind. Darüber hinaus bietet Co-Simulation vielfältige Modellierungsmöglichkeiten, da die Simulationsalgorithmen oftmals an eine bestimmte Art der Modellierung gebunden sind [Anso03]. Bei einer Co-Simulation kommunizieren beide Werkzeuge miteinander über Methoden der Interprozesskommunikation (IPC). Zur Synchronisation beider Simulatoren kann z. B. der Lock-Step- oder der Fixed-Time-Step-Algorithmus verwendet werden. Ein Simulator, der Master, übernimmt dabei die Führung; der Slave antwortet auf die Anfragen des Masters (Bild 24) [SeTh01].

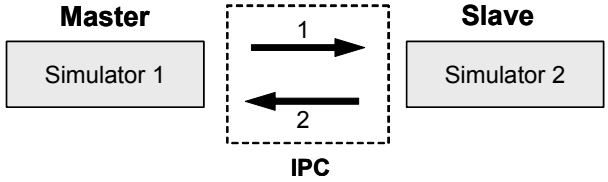


Bild 24: Co-Simulation mit Master und Slave

Diese Ansätze haben jedoch Nachteile bzgl. Genauigkeit und Performance. Effizientere Synchronisationsalgorithmen setzen für ihre Nutzung entsprechende Simulatorschnittstellen voraus [Drei00] [Drei01] [Drei02]. Um mehrere Funktionalitäten kombinieren und das Gesamtsystem betrachten zu können, kommt z. B. auch Modell-Export in Frage. Bei einem Modell-Export wird das im Quell-Simulator entwickelte Modell in ein Format konvertiert, das kompatibel zur Modellierungssprache des Target-Simulators ist, und anschließend in diesen importiert (Bild 25) [SeTh01].

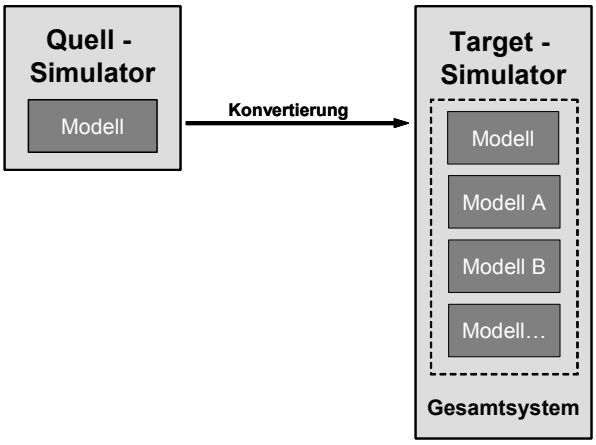


Bild 25: Modell-Export zur Gesamsystem-Simulation

Vorteilhaft ist hier, dass nur ein Simulator bei der Gesamtsimulation benötigt wird. Dadurch entfällt die Synchronisation zwischen verschiedenen Simulatoren und eine bessere Performance wird erhal-

ten. Der Anwender muss nur einen Simulator bedienen. Ein Nachteil ist der zusätzliche Konvertierungsprozess, der überdies nur in Ausnahmefällen gelingt, z. B. bei starker Einschränkung der Quell-Modellierungssprache [SeTh01].

2.3.2.1 Simulationsmethoden

Anhand der Art und Weise, wie die simulierte Realzeit, die im Folgenden virtuelle Zeit oder auch Simulationszeit genannt wird, voranschreitet, lassen sich Simulationsmethoden klassifizieren [MaMe89]. Die Simulationsmethode definiert das zu berücksichtigende Zeitverhalten und beschreibt damit die Zeitablaufsteuerung und das Voranschreiten der Simulationszeit sowie die Durchführung der damit verbundenen Zustandsänderungen des Modells (ereignisdiskret, zeitdiskret, kontinuierlich, hybrid) [Wenz02]. Aufgrund der verschiedenen Modellklassen, die bei einer mathematischen Beschreibung existieren, wurde im Laufe der Zeit eine große Zahl von Simulationswerkzeugen entwickelt, die auf die jeweiligen Themenbereiche zugeschnitten wurden. Die Simulatoren unterscheiden sich dabei in ihrer Zeitauflösung – die Simulation kann diskret oder kontinuierlich, d. h. in festen Takten oder durch Ereignisse gesteuert (z. B. diskret, kontinuierlich) erfolgen –, in ihrer Signalauflösung (z. B. reeller Wert, Bitwert) und in der Abstraktionsebene der verwendeten Modellbeschreibungsmöglichkeiten (z. B. Transistorebene, VHDL-Ebene) [Krüg75].

Definition Simulationszeit und Simulationsschritt: Die Simulationszeit wird in der Einheit Simulationssekunde angegeben und ist ein virtuelles Zeitmaß für den Simulationsfortschritt. Ein Simulationsschritt ist bei der Simulation kontinuierlicher Systeme die Berechnung der Ausgangswerte zu einer bestimmten Simulationszeit, während bei der Simulation ereignisdiskreter Systeme darunter die Verarbeitung eines Ereignisses und die damit verbundene Auswertung des Statecharts verstanden wird. Diese Definition ist nicht widersprüchlich, sondern berücksichtigt lediglich die Tatsache, dass bei ereignisdiskreter Simulation mehrere Ereignisse mit gleichem Zeitstempel auftreten können, die folglich innerhalb derselben Simulationszeit zu verarbeiten sind.

Definition kontinuierliche Simulation: Bei zeitkontinuierlicher (kontinuierlicher) Simulation schreitet die Simulationszeit (und damit die Folge der Zustandsänderungen) kontinuierlich voran, d. h. Zustandsänderungen erfolgen kontinuierlich mit der Zeit und sämtliche Werte des Systems werden in Abhängigkeit von der fortschreitenden Zeit berechnet. Das Systemverhalten wird typischerweise mit Hilfe von Differentialgleichungen beschrieben. Die numerische Berechnung von Differentialgleichungen, z. B. mit Rechteckintegration, stellt eine Näherungslösung für die kontinuierliche Simulation dar. Typische Beispiele von Anwendungsbereichen, bei denen kontinuierlich simuliert wird, sind die Modellierung von Wetter- oder Klimaverhältnissen, des Luftstroms um die Tragflächen eines Flugzeugs und Spannungsänderungen auf den Leiterbahnen eines elektronischen Schaltkreises.

Definition diskrete Simulation: Erfolgt das Voranschreiten der Zeit dagegen sprunghaft von einem Zeitpunkt zum nächsten, wird von zeitdiskreter (diskreter) Simulation gesprochen, d. h. Zustandsänderungen können bei der zeitdiskreten Simulation nur als Folge eines zu einem ausgewählten Zeitpunkt aufgetretenen Ereignisses vollzogen werden. Es werden lediglich ausgewählte Werte zu bestimmten klar unterscheidbaren Zeitpunkten berechnet. Das Simulationsmodell betrachtet das physikalische System wie wenn der Zustand nur an diskreten Zeitpunkten der Simulationszeit verändert würde. Ein Zustandswechsel in einem solchen Modell lässt sich durch das Eintreten von Ereignissen zu einem virtuellen Eintrittszeitpunkt t auffassen. Konzeptionell wird das System als von einem Zustand zum nächsten „springend“ betrachtet. Jede Änderung einer Zustandsvariablen tritt an einem bestimmten Zeitpunkt der Simulationszeit auf[27]. Auf diese Weise erreicht man eine Einsicht in das Verhalten eines komplexen Systems, in dem nur wenige wichtige Ergebnisse berechnet werden. Die

[27] In der Praxis gilt das auch für kontinuierliche Simulationen. Obwohl die Zustandsvariablen als sich kontinuierlich über der Zeit ändernd betrachtet werden, definiert der Simulator Zeitschritte und berechnet typischerweise neue Werte nur an den Grenzen der Zeitschritte.

Eintrittszeit t wird auch als Zeitstempel eines Ereignisses bezeichnet. Je nach Fortschreiten der Simulationszeit kann bei diskreter Simulation weiter zwischen zeit- und ereignisgesteuerter Simulation unterschieden werden (s. u.). Davon abzugrenzen ist die Unterscheidung in wertdiskrete und wertkontinuierliche Simulation.

Definition wertdiskrete und wertkontinuierliche Simulation: Bei wertdiskreter Simulation nehmen die an der Simulation beteiligten mathematischen Größen lediglich diskrete Werte an – beispielsweise natürliche Zahlen, binäre Werte oder alphanumerische Zeichen. Bei der wertkontinuierlichen Simulation sind kontinuierliche Größen beteiligt, die z. B. eine analoge physikalische Systemgröße als Entsprechung haben und durch reelle Zahlen dargestellt werden.

Definition ereignisgesteuerte Simulation: Bei der ereignisgesteuerten (ereignisorientierten, -getriebenen, -diskreten) Simulation (discrete event simulation, DES) bestimmt lediglich das Auftreten eines Ereignisses darüber, ob und welche Werte des Gesamtsystems neu berechnet werden; das Modell wird asynchron ausgeführt. Es wird auf die Berechnung der Werte zu jedem Zeitpunkt verzichtet; sie erfolgt nur bei bestimmten Zeitpunkten. Die Beobachtungszeitpunkte des simulierten Systems fallen mit dem Eintreten von Ereignissen zusammen, wobei diese während der Simulation sukzessive abgearbeitet werden. Dies wird dadurch ermöglicht, dass die Ereignisse mit Zeitstempeln versehen sind, die den Zeitpunkt ihres Eintretens wiedergeben, und bei ihrer Erzeugung (scheduling) in eine zeitlich geordnete Ereignisliste (event queue, EQ) eingereiht werden. Diese Methode eignet sich nicht für dynamische Vorgänge, bei denen dauernde zeitliche Veränderungen stattfinden. Die ereignisdiskrete Simulation basiert auf dem Verständnis von Ereignissen und Aktivitäten. Ereignisse (events) sind atomar und verbrauchen keine Zeit. Aktivitäten (activities) sind zeitbehaftete Operationen, die den Zustand eines einzigen Objektes transformieren [Wenz02]. Jedes Ereignis kann zu einer Veränderung einer oder mehrerer Zustandsvariablen führen. Bei der ereignisgesteuerten Simulation wird die Simulationszeit jeweils auf die Eintrittszeit des nächsten Ereignisses (d. h. auf die kleinste Eintrittszeit eines noch zu simulierenden Ereignisses) erhöht (siehe Bild 26), anstatt sie um ein festes Zeitquantum voranschreiten zu lassen.

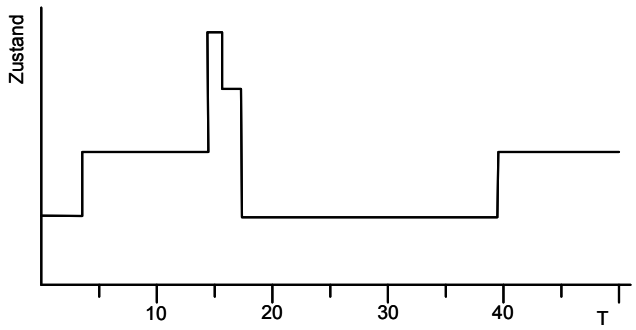


Bild 26: Ereignisgesteuerte Simulation

Ereignisse steuern also das Voranschreiten der Simulationszeit. Dadurch werden alle Totzeiten übersprungen, die in der zeitgesteuerten Simulation (s. u.) und auch in der Realität berücksichtigt werden müssten. Eine ereignisgesteuerte Simulation kann eine zeitgesteuerte Simulation durch die Definition von Ereignissen, die bei jedem Zeitschritt erzeugt werden, emulieren. Die Ereignisse werden vom Simulationsprogramm in einer üblicherweise nach der Eintrittszeit sortierten Ereignisliste gehalten. Das Simulationsprogramm entfernt aus der Ereignisliste den Eintrag mit der kleinsten Zeit. Diese Zeit ist der Zeitpunkt des Eintretens des Ereignisses. Die Simulationsuhr wird auf diesen Wert gesetzt, und das Ereignis wird simuliert. Durch das Bearbeiten des aktuellen Ereignisses können der Ereignisliste neue Elemente hinzugefügt oder bestehende Einträge gelöscht werden [Kepp94]. Dadurch ist der Rechenzeitaufwand bei ereignisgesteuerter Simulation trotz des durch die Ereignisverwaltung resultierenden Zusatzaufwandes kleiner als bei zeitgesteuerter Simulation, außer wenn die Ereignisgranularität im zeitlichen Durchschnitt sehr hoch ist.

Definition zeitgesteuerte Simulation: Bei der zeitgesteuerten (periodenorientierten, zeitgetriebenen) Simulation werden zu jedem Zeitschritt (Periode) alle Werte des Systems berechnet. Der Periodenabstand ist gleichbleibend, d. h. die Simulationszeit schreitet innerhalb eines Zeitrasters mit konstanter Schrittweite fort, wodurch die Beobachtung des simulierten Systems zeitlich in einheitlichen Intervallen diskretisiert wird. Wird der Periodenabstand klein genug gewählt, können ebenfalls dynamische Systeme mit Rückkopplungen exakt beschrieben und simuliert werden.

Bei zeitgesteuerter Simulation wird die Simulationszeit t in Inkrementen Δt fester Schrittlänge erhöht. Nach jeder Zeiterhöhung treten alle Ereignisse mit einem Zeitstempel $t' \in (t_{aktuell} - \Delta, t_{aktuell})$ in einer prinzipiell willkürlichen Reihenfolge ein, d. h. werden simuliert. Die Ereignisse können gleichzeitig oder zu verschiedenen Zeitpunkten eintreten. Das Zeitinkrement Δt muss dabei klein genug gewählt werden, damit ein in einem Zeitintervall eintretendes Ereignis nur Ereignisse in der simulierten Zukunft, nicht aber andere Ereignisse des gleichen Zeitintervalls beeinflussen kann. Es wird also vorausgesetzt, dass die Reihenfolge, in der die Ereignisse simuliert werden, keine Auswirkungen auf die Simulation hat. Die Simulation eines Ereignisses darf keine neuen Ereignisse bedingen, die Auswirkungen auf Elemente des Intervalls haben. Andererseits sollte Δt nicht zu klein gewählt werden, damit möglichst wenige Intervalle simuliert werden müssen, in denen überhaupt keine Ereignisse stattfinden. Die Größe des Zeitintervalls Δt ist von entscheidender Bedeutung für Korrektheit, Genauigkeit und Effizienz der Simulation. Es muss klein genug sein, damit Zustandsänderungen keine Auswirkungen auf Ereignisse des gleichen Intervalls haben. Andererseits sollte es möglichst groß sein, um zu verhindern, dass die Simulationszeit viele Male inkrementiert wird, ohne ein Ereignis zu simulieren. D. h. die Wahl der Schrittweite beeinflusst stark einerseits die Simulationsgenauigkeit, andererseits die Rechengeschwindigkeit, wobei hier ein Kompromiss zwischen erforderlicher Genauigkeit und noch vertretbarer Simulationsdauer gefunden werden muss. Bei zeitlich ungleichmäßig verteilt eintretenden Ereignissen sind die für die zeitgetriebene Simulation verwendeten Simulationsalgorithmen daher nicht besonders effizient. Die Simulation solcher Totzeiten stellt einen wesentlichen Nachteil zeitgesteuerter Simulationen dar. Ein typisches Beispiel für die zeitgesteuerte Simulation ist die Simulation von VLSI-Schaltungen [Spir90]. Die Ausführung einer zeitgesteuerten Simulation ist in Bild 27 dargestellt [Fuji00].

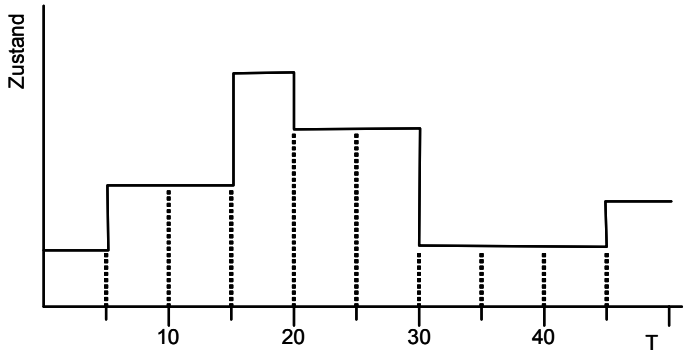


Bild 27: Zeitgesteuerte Simulation

2.3.2.2 Simulationsarten

Beim Entwurf von Schaltungen und Systemen der Mikroelektronik werden folgende Simulationsarten unterschieden [SiSi03]:

- Analogsimulation (z. B. mit PSpice© von Cadence Design Systems, Inc.): Gatter werden durch Transistormodelle ersetzt. Es wird das funktionale und zeitliche Verhalten der Signale durch Ströme und Spannungen repräsentiert.
- Digitalsimulation: Die möglichen Signalzustände werden auf der Register-Transfer-Ebene auf eine zwei- oder mehrwertige Logik abgebildet.

- Fehlersimulation: Während in der Digitalsimulation das Verhalten nur für einzelne Stimuli untersucht wird, besteht das Ziel der Fehlersimulation darin, die möglichst vollständige Prüfbarkeit der Schaltung nachzuweisen. Dazu werden systematisch viele Stimulikombinationen (Testvektoren) erzeugt, die einen möglichst großen Teil der Schaltung beeinflussen (hoher Fehlerüberdeckungsgrad).
- Mixed-Signal-Simulation (Mixed-Mode-Simulation): Es werden Schaltungen zusammen simuliert, die in Teilen auf unterschiedlichen Abstraktionsebenen modelliert sind. Besondere Bedeutung hat dabei die gemeinsame Simulation analoger (z. B. Transistoren, Dioden und Widerstände) und digitaler Schaltungsblöcke bzw. die gemeinsame Simulation mit diskret und kontinuierlich arbeitenden Simulationswerkzeugen. Digitale Systeme besitzen oftmals auch einen Anteil analoger Schaltungsblöcke wie z. B. A/D- und D/A-Wandler (Digital/Analog-Wandler), so dass insbesondere das Zusammenspiel dieser hybriden Komponenten (z. B. analoge Schaltungen aus der Transistorebene und digitale Schaltungen aus der Gatterebene) überprüft werden muss. Zur Standardisierung der D/A-Simulation wurde der VHDL-Sprachstandard um Elemente zur Analogmodellierung erweitert: Mit VHDL-AMS [VAMS05], einer Modellierungssprache für das Analog- und Mixed-Signal-Design, können über digitale Bauelemente hinaus z. B. auch Transistoren und Operationsverstärker auf Verhaltensebene modelliert werden (siehe 3.3.4). Eine besondere Klasse von Mixed-Signal-Simulatoren (z. B. Saber© der Firma Synopsys, Inc.) widmet sich der Simulation von elektrisch-mechanischen Systemen. Bei der Mixed-Mode-Simulation wird ein zeitgesteuertes Modell durch Verwendung konstanter Zeitschritte synchron ausgeführt. Simulatoren, die Schaltungen mit entweder digitalen oder analogen Komponenten aus zwei oder mehreren Abstraktionsebenen gemeinsam simulieren können, z. B. digitale Komponenten der System- bzw. der Gatterebene, werden als Mixed-Level-Simulatoren bezeichnet [Schn97].
- System- oder High-Level-Simulation: Unter diesem Begriff fasst man die Simulation sehr komplexer, oft auch heterogener Schaltungen auf hoher Abstraktionsebene zusammen [BeHa-01]. Die meisten Systemsimulatoren besitzen u. a. eine C-Schnittstelle, mithilfe derer der Anwender den Simulator durch eigene Implementierungen ergänzen kann [SiSi03].

2.3.2.3 Zeit und Kausalität

Bei jeder Simulation existiert ein gemeinsamer, globaler Zeitmaßstab, der von allen Komponenten der Simulation erkannt wird. Das stellt sicher, dass alle Teile der Simulation ein gemeinsames Verständnis von Vorher-Nachher-Beziehungen zwischen simulierten Aktionen haben, die zu bestimmten Zeitpunkten in der simulierten Zeit eintreten.

Definitionen des „Zeit“-Begriffes:

- Die physikalische Zeit bezieht sich auf die Zeit innerhalb des physikalischen Systems.
- Die Simulationszeit ist eine von der Simulation benutzte Abstraktion, um die physikalische Zeit zu modellieren. Die Simulationszeit ist eine vollständig geordnete Menge aus Werten, wobei jeder Wert einen Zeitpunkt im physikalischen System repräsentiert, das modelliert wird. Für zwei beliebige Werte der Simulationszeit T_1, die die physikalische Zeit P_1 repräsentiert, und T_2, die P_2 repräsentiert, gilt: wenn $T_1 < T_2$, dann tritt P_1 vor P_2 auf, und $(T_2 - T_1)$ ist gleichbedeutend mit $(T_2 - T_1) \cdot K$ für eine beliebige Konstante K. Wenn $T_1 < T_2$, tritt T_1 vor T_2 auf und wenn $T_1 > T_2$, tritt T_1 nach T_2 auf.
- Die „Wallclock“-Zeit bezieht sich auf die Zeit während der Ausführung des Simulators, der die Wallclock-Zeit i. A. aus einer HW-Uhr, die vom BS verwaltet wird, ausliest [Fuji00].

Alle drei Definitionen bedienen sich Skalierungen und Achsen zur Repräsentation bestimmter Zeitpunkte. Zwischen Intervallen der Simulationszeit und der physikalischen Zeit besteht eine lineare Beziehung.

Bei der Simulation eines Systems wird das als Programm spezifizierte Modell ausgeführt, wobei sich der Zustand des Modells mit dem Fortschreiten der Simulationszeit verändert. Die Simulationszeit entspricht dabei der im realen System voranschreitenden Zeit. Diese steht jedoch nicht in direktem Bezug zur Rechenzeit, die zur Ausführung des Simulationsprogramms benötigt wird. Der Fortschritt der Simulationszeit und die Durchführung der damit verbundenen Zustandsänderungen des Modells können auf unterschiedliche Art erfolgen und charakterisieren wesentlich die verschiedenen Simulationsmethoden [MaMe89] [Page91].

Bei Simulationen virtueller Umgebungen muss die Simulationszeit synchron zur Wallclock-Zeit sein, sonst erscheint die simulierte Umgebung unrealistisch. Simulationsausführungen, bei denen Fortschritte in der Simulationszeit denen in der Wallclock-Zeit entsprechen, heißen Echtzeitausführungen, und Simulatoren, die für diesen Modus entworfen wurden, heißen Echtzeit-Simulatoren. Eine Variation davon ist die skalierte Echtzeitausführung. Die Simulationszeit schreitet hierbei um einen konstanten Faktor schneller oder langsamer voran als die Wallclock-Zeit.

Mit Hilfe der folgenden Funktion wird die Wallclock-Zeit in die Simulationszeit konvertiert:

$$T_s = T_{Start} + Scale \cdot (T_w - T_{wStart})$$

mit

T_w	Wert der Wallclock-Zeit
T_{Start}	Simulationszeit, bei der die Simulation startet
T_{wStart}	Wallclock-Zeit am Simulationsbeginn
$Scale$	Skalierungsfaktor.

Ein Zeitschritt in der Simulationszeit kann in einem Teil der Simulation mehrere Sekunden in der Wallclock-Zeit benötigen, in einem anderen Teil Minuten und in einem dritten sogar Stunden. Diese Simulationen werden so schnell wie möglich ausgeführt, ohne dass eine feste Beziehung zwischen dem Foranschreiten der Simulations- und der Wallclock-Zeit besteht.

Definition Gesamtzeit zur parallelen Ausführung des Algorithmus: Es soll davon ausgegangen werden, dass ein Algorithmus gleichmäßig auf parallele Prozessoren aufgeteilt werden kann. Für die Gesamtzeit $T(N)$ zur parallelen Ausführung des Algorithmus gilt [FlKe89]:

$$T(N) = T_S + \frac{T_P}{N} + T_K(N)$$

mit

N	Anzahl der Prozessoren
T_S	serieller, nicht parallelisierbarer Zeitanteil
T_P	parallelisierbarer Anteil, der zwischen N parallelen Prozessoren aufgeteilt werden kann
$T_k(N)$	Zeitanteil zur Kommunikation, um Zwischenergebnisse zwischen parallelen Prozessoren auszutauschen

Eine Normierung auf T_E ergibt:

$$t(N) = t_S + \frac{t_P}{N} + t_K(N)$$

mit

$$t(N) = \frac{T(N)}{T_E}$$

$$t_S = \frac{T_S}{T_E}$$

$$t_P = \frac{T_P}{T_E}$$

$t_K(N) \quad = \frac{T_K(N)}{T_E}$

T_E Ausführungszeit eines Algorithmus auf einem Einzelprozessor

$T_E \quad = T_S + T_P$

Definition Speedup: Der Speedup (die Beschleunigung) $S(N)$ ist das Verhältnis der Ausführungszeit eines Algorithmus auf einem Prozessor zur Ausführungszeit auf N Prozessoren [SiSi03], definiert also den Zeitgewinn eines Programms bei der Ausführung auf einem Parallelrechner [RePo99][28]:

$$S(N) = \frac{T(1)}{T(N)}$$

mit

T Ausführungszeit

Mit $t(N)$ ergibt sich:

$$S(N) = \frac{1}{t_S + \frac{t_p}{N} + t_k(N)}$$

Mit $t_K(N) = 0$ ergibt sich die als Amdahls Law bekannte Aussage über die maximal erreichbare Beschleunigung mit p Prozessoren gegenüber einem Prozessor [Amda67]. Dabei gelten folgende Annahmen:

- Die Problemgröße ist fest, d. h. bei wachsender Prozessoranzahl wird das Programm mit gleichen Datensätzen ausgeführt.
- Das Programm enthält sequenzielle und parallele Programmteile, deren Verhältnis genau bekannt ist ($tseq$ bzw. $tpar$ Zeiteinheiten, mit einer Ausführungszeit von $tseq + tpar$ im Einprozessorfall; im Folgenden auf 1 normiert). Auf einem Parallelrechner bleibt die Ausführungszeit des sequenziellen Anteils unverändert; der parallele Teil hingegen kann durch ideale Lastverteilung verkleinert werden.

  $$\textit{Maximale relative Beschleunigung} = \frac{tseq + tpar}{tseq + \frac{tpar}{N}} = \frac{N}{1 + (N-1) \cdot tseq}$$

  Ist der sequenzielle Anteil etwa 10 %, ist nach Amdahl auch bei Einsatz von beliebig vielen Prozessoren keine Beschleunigung größer als 10 erreichbar.

Überlineare (super-lineare) Beschleunigung liegt dann vor, wenn bei Einsatz von *N* Prozessoren ein Beschleunigungswert resultiert, der größer als *N* ist. In der Praxis gibt es zwei Situationen, die eine überlineare Beschleunigung bewirken: bei der Bearbeitung von Branch-and-Bound-Algorithmen und durch den Einsatz eines in der Gesamtsumme größeren Cache-Speichers, so dass langsamere Hauptspeicherzugriffe reduziert werden [RePo99]. Trotz steigender Beschleunigung kann es bei wachsender Prozessoranzahl unökonomisch sein, weitere Prozessoren einzusetzen (siehe Kapitel 5).

Definition Effizienz: Die Effizienz $E(N) = \frac{S(N)}{N}$ ist der mit der Anzahl der Prozessoren normierte Speedup und drückt aus, welcher Anteil der Prozessorleistung tatsächlich nutzbar ist [RePo99].

[28] Je nach Anwendung hat man unterschiedliche Definitionen des Speedups. Eine für den HW/SW Coentwurf angepasste Definition ist z. B. in [Henk96] zu finden.

2.3.2.4 Physikalisches und logisches Modell

Definition physikalischer und logischer Prozess: Unter dem Begriff „physikalisches Modell“ wird hier die Modellierung eines zu simulierenden (realen) Systems durch ein physikalisches System verstanden. Es werden physikalische Systeme betrachtet, die aus einer endlichen Anzahl physikalischer Prozesse (PPs) bestehen. Jeder PP modelliert Einheiten des zu simulierenden Systems. Ein logisches System besteht aus einer Menge von logischen Prozessen (LPs), die physikalische Prozesse simulieren. Zur Simulation eines physikalischen Systems wird jedem PP genau ein LP zugeordnet. Unter einem LP wird die Ausführung eines Simulationsprogramms auf einem Prozessor verstanden. Dabei muss die Semantik des Simulationsprogramms das Verhalten des PP beschreiben [Kepp94].

Ein weit verbreitetes Modell zur Beschreibung von Parallelität bei der diskreten Ereignissimulation wurde von Chandy, Holmes und Misra [ChHM79] [ChMi81] vorgeschlagen. Dabei werden Systeme betrachtet, die aus einer endlichen Anzahl von PPs bestehen. Jeder PP modelliert eine Komponente des zu simulierenden Systems. Jeder PP wird durch eine Menge von Ereignissen mit zugehöriger Eintrittszeit beschrieben. Zustandsänderungen zu den Zeitpunkten t_i in dem zu simulierenden System werden durch Nachrichten zwischen den PPs, die von der Zustandsänderung betroffen sind, zu diskreten Zeitpunkten t_i modelliert. Ein physikalisches System kann simuliert werden, indem jeder PP des physikalischen Systems durch einen separaten LP simuliert wird. Die Menge aller LPs wird als logisches System bezeichnet. Die LPs kommunizieren durch Nachrichtenaustausch [Hoar85]. Es existiert nur dann eine gerichtete Verbindung von LP_i zum LP_j im logischen System, wenn im physikalischen System der PP_i Nachrichten an den PP_j sendet. Ein gerichteter Nachrichtenkanal von LP_i zum LP_j ist für LP_i ein Ausgangskanal und für LP_j ein Eingangskanal. Die Zeit, zu der im physikalischen System eine Nachricht gesendet wird (physikalische Zeit), ist Bestandteil der Nachricht im logischen System (logische Zeit). Eine Nachricht *m*, die im physikalischen System von PP_i an PP_j zur physikalischen Zeit *t* gesendet wird, wird durch eine Nachricht *(t, m)* simuliert, die im logischen System von LP_i an LP_j gesendet wird. Die logische Zeit *t* einer Nachricht *(t, m)* im logischen System wird auch als Zeitstempel der Nachricht bezeichnet. Um einen PP korrekt zu simulieren, muss der entsprechende LP die Nachrichten gemäß ansteigender Nachrichtenzeiten bearbeiten und nicht in der Reihenfolge ihrer Ankunft [Kepp94].

2.3.2.5 Sequenzielle ereignisgesteuerte Simulation

Simulationsmodelle lassen sich als Spezifikationen physikalischer Systeme auffassen, die durch Zustände und Ereignisse dargestellt werden. Bei der Durchführung einer solchen Simulation werden eintretende Ereignisse nachgebildet und die daraus folgenden Auswirkungen bestimmt, die sich in Zustandsänderungen und neu erzeugten Ereignissen äußern. Ein Programm, das ein ereignisgesteuertes Simulationsmodell ausführt, wird ereignisgesteuerter Simulator genannt.

Definition sequenzielle Simulation: Eine Simulation, die durch einen einzigen ereignisgesteuerten Simulator auf einem einzelnen Rechner ausgeführt wird, wird sequenzielle Simulation genannt und der entsprechende Simulator auch sequenzieller Simulator.

Ein sequenzieller ereignisdiskreter Simulator verwendet typischerweise drei Datenstrukturen:

- Zustandsvariablen zur Beschreibung des Systemzustandes;
- eine Ereignisliste, die Ereignisse enthält, die zu einem Zeitpunkt in der simulierten Zukunft eintreten;
- eine globale Uhr-Variable zur Anzeige der aktuellen Simulationszeit.

Alle Ereignisse in der Ereignisliste müssen einen Zeitstempel besitzen, der größer oder gleich der aktuellen Simulationszeit ist. Es können nur neue Ereignisse eingeplant werden, die in der simulierten Zukunft liegen. Die Ereignisse mit den kleinsten Zeitstempeln werden zuerst ausgeführt. Damit wird sichergestellt, dass Ereignisse in ihrer chronologischen Reihenfolge verarbeitet werden und dass

sich die Simulationszeit während der Simulation nicht vermindert. Dies verhindert eine Beeinflussung der Berechnung von Ereignissen, die einen kleineren Zeitstempel besitzen.

Die Simulation beginnt durch die Initialisierung der Zustandsvariablen und der Generierung von Anfangsereignissen. Vor Beginn der Simulation enthält die Ereignisliste mindestens ein Ereignis, das im Verlauf der Simulation eintreten (ausgeführt werden) soll. Ein sequenzieller Simulator lässt alle Ereignisse der Simulation chronologisch eintreten, indem er zyklisch das Ereignis mit der kleinsten Eintrittszeit t (nächstes Ereignis eines Simulators) aus der Ereignisliste auskettet, seine Uhr auf t setzt und anschließend die mit dem Ereignis assoziierte Ereignisroutine ausführt. An dieser Stelle sei angenommen, dass es keine zwei Ereignisse mit gleichem Zeitstempel gibt. Dadurch ist das nächste Ereignis immer eindeutig definiert. Durch das Ausführen der Ereignisroutine können Variablen des Zustandsraums gelesen und verändert und neue Ereignisse mit Zeitstempel $t' \geq t$ erzeugt und in die Ereignisliste eingefügt werden. Sobald der Simulator keine Ereignisse mehr in der Ereignisliste vorfindet oder eine eventuell vom Benutzer vorgegebene Abbruchbedingung (etwa eine bestimmte Simulationszeit) erreicht wird, beendet er die Simulation.

Simulationsverfahren mit sequenzieller Bearbeitung der Ereignisse nutzen nicht die Parallelität, die in einigen Simulationsmodellen vorhanden ist. Das gilt insbesondere für die Modelle, deren Komponenten in schwacher Wechselwirkung zueinander stehen und die sich somit für eine parallele Simulation besonders eignen. Parallele und verteilte Simulation ist deshalb interessant, weil Zeitschranken und Speicherengpässe sequenzieller Simulationsverfahren beseitigt werden können, indem die Arbeit auf mehrere Prozessoren verteilt wird [Reed83] [ReFu87] [Fuji01].

Viele Faktoren haben Einfluss auf die Performance der Simulation, z. B. die Eigenschaften des zu simulierenden Modells, das verwendete Synchronisationsprotokoll, die Simulatoren selbst, die Kommunikation über Netzwerk und Middleware, das BS und die eingesetzte HW. Ein trivialer Ansatz zur Beschleunigung einer Serie aus unabhängigen Einzelexperimenten besteht darin, jedes Einzelexperiment auf einem Prozessor durchzuführen. Sind die Einzelexperimente jedoch abhängig voneinander, ist dieser Ansatz nicht anwendbar.

Ein Ansatz, ein einzelnes Simulationsexperiment zu beschleunigen, besteht darin, häufig benötigte Hilfsfunktionen, wie das Generieren von Statistiken, die Ein-/Ausgabe und das Verwalten der Ereignisliste, auf andere Prozessoren auszulagern [Comf84] [WyYo83]. Jedes sequenzielle Simulationsmodell könnte damit ohne Veränderung potentiell beschleunigt werden. Um eine signifikante Beschleunigung der Simulation zu erreichen, müssten die Hilfsfunktionen allerdings einen großen Teil der Gesamtrechenzeit beanspruchen.

2.3.2.6 Parallele und verteilte Simulation

Ein weiterer Ansatz, ein einzelnes ereignisgesteuertes Simulationsexperiment zu beschleunigen, besteht darin, verschiedene Ereignisse nebenläufig auf unterschiedlichen Prozessoren auszuführen. Dazu wird das Gesamtmodell in mehrere kooperierende Teilmodelle partitioniert, die jeweils von einem ereignisgesteuerten Simulator ausgeführt werden. Für einen solchen, je ein Teilmodell ausführenden, kooperierenden ereignisgesteuerten Simulator (sequenzielle Simulation) hat sich in der Literatur die Bezeichnung logischer Prozess (LP, siehe oben) eingebürgert. Um bei diesem Ansatz eine Beschleunigung der Simulation zu erreichen, sollten folgende Anforderungen erfüllt werden:

- Partitionierung des Gesamtmodells in möglichst unabhängige Teilmodelle, die jeweils von einem LP ausgeführt werden;
- geeignete Zuordnung (Mapping) von Prozessoren auf LPs. Falls ein Prozessor mehrere LPs bedienen kann, ist darüber hinaus eine geeignete Auswahlstrategie (Scheduling) erforderlich;
- Synchronisation der Ereignisausführungen, so dass die Kausalität gewahrt bleibt und mit möglichst wenig Synchronisationsaufwand möglichst viele Ereignisse nebenläufig ausgeführt werden können.

Durch das Aufteilen des simulierten Modells in mehrere Prozesse entstehen weitgehend voneinander unabhängige Partitionen, die durch Abbildung auf die einzelnen Knoten eines Multiprozessorsystems die Ausnutzung der Modellparallelität ermöglichen sollen. Dabei ist zu beachten, dass sich in der Regel keine vollkommene Unabhängigkeit der Partitionen untereinander erzielen lässt, was insbesondere zeitliche Abhängigkeiten unter den einzelnen Prozessen schafft und damit Verfahren zur Synchronisation nötig macht. Die Prozesse können über einen Kommunikationsmechanismus untereinander Nachrichten austauschen.

Definitionen verteiltes System:

- Ein verteiltes System besteht aus unabhängigen, über ein Rechnernetz kommunizierenden Rechnern, wobei keine zentrale Systemsteuerung existiert und der Verteilungsaspekt für die Benutzer des Systems möglichst transparent ist [ScWe04].
- Ein verteiltes System besteht aus mehreren Prozessen, die mittels Nachrichten miteinander kommunizieren [Kien97].
- Ein System heißt verteilt, wenn es aus mehreren Komponenten besteht, die
    - unabhängig voneinander arbeiten könnten,
    - sich an räumlich getrennten Stellen befinden oder befinden könnten,
    - miteinander vernetzt sind und dadurch auf gemeinsame Ressourcen zugreifen oder sich gegenseitig Nachrichten schicken können und
    - gemeinsam an einer Aufgabe arbeiten [DuIn01].

Parallele Rechnerstrukturen lassen sich durch ihre Arbeitsweise in „Single Instruction stream, Multiple Data stream“ Systeme (SIMD[29]-Systeme) und „Multiple Instruction stream, Multiple Data stream“ Systeme (MIMD[30]-Systeme) klassifizieren. Die zentrale Eigenschaft dieser Rechner ist, dass alle Prozessoren zu jedem Zeitpunkt der Programmausführung dieselben Anweisungen, die von einer Zentraleinheit vorgegeben werden, auf unterschiedlichen Daten ausführen. Die Synchronisation der Prozesse wird z. B. durch Anwendung eines Lock-Step-Verfahrens erreicht, wobei die Prozesse zeitlich sehr eng aneinander gekoppelt sind und nur Ereignisse mit identischen Zeitstempeln nebenläufig abarbeiten dürfen. SIMD-Systeme werden häufig als SM-Architekturen realisiert, so dass die Prozesse gemeinsam eine zentrale Ereignisliste nutzen können. Die nebenläufige Simulation in SIMD-Umgebungen mit mehreren Prozessoren und zentraler Steuerung unter Anwendung hierfür geeigneter Synchronisationsmechanismen wird als parallele Simulation bezeichnet und der Spezialfall der DES als parallele und verteilte ereignisgesteuerte Simulation (Parallel and distributed Discrete Event Simulation, PDES). Durch die Verwendung verteilter Ereignislisten, die meist in MIMD-Umgebungen eingesetzt werden, lässt sich eine höhere Unabhängigkeit der einzelnen Simulationsprozesse voneinander erreichen, wodurch auch Ereignisse mit verschiedenen Zeitstempeln nebenläufig bearbeitet werden können. Das macht Protokolle zur lokalen Synchronisation der Simulationsprozesse notwendig, die unter Umständen einen erhöhten Kommunikationsaufwand bewirken. Die auf die Prozessoren eines MIMD-Systems verteilten Prozesse laufen asynchron zueinander und tauschen Nachrichten aus, die neben dem Datenaustausch auch die lokale Synchronisation nach einer dazu geeigneten Strategie durchführen.

Definition parallele und verteilte Simulation: Eine Simulation auf einem Mehrrechnersystem wird als parallele Simulation bezeichnet (siehe 2.3.2.6). Mehrrechnersysteme sind i. A. teurer als Systeme ohne gemeinsamen Speicher sowie schlechter skalierbar, da der gemeinsame Speicher mit steigender

[29] SIMD ist eine Computerarchitektur, bei der mehrere Daten gleichzeitig in mehreren Prozessoren mit ein und demselben Befehl bearbeitet werden (Vektorrechner).

[30] MIMD ist eine Sammelbezeichnung für eine größere Klasse von Parallelrechnern, die aus mehreren, durch verschiedene Befehlsflüsse gesteuerten Knoten (Teilrechnern) bestehen. Ein wesentliches Merkmal ist die Speicheranbindung der Knoten: Gemeinsamer Speicher (shared memory, SM).

Anzahl der Prozessoren zunehmend zu einem Flaschenhals wird. Bei einer moderaten Anzahl von Prozessoren ist jedoch die Kommunikation über den gemeinsamen Speicher sehr effizient. Eine Simulation, die auf einem verteilten Rechnersystem ausgeführt wird, d. h. auf einer Menge von Rechnern, die jeweils einen lokalen Speicher zur Verfügung haben und ausschließlich über Nachrichten miteinander kommunizieren können, heißt verteilte Simulation. Ein verteiltes Simulationsprogramm besteht aus zwei oder mehr zusammenwirkenden Prozessen, die ausschließlich durch Nachrichtenaustausch miteinander kommunizieren. Die Prozesse können auf einem Prozessor oder auf mehreren Prozessoren verteilt ausgeführt werden. Solange sie unabhängig voneinander ablaufen, können mehrere Prozesse gleichzeitig arbeiten. In diesem Fall spricht man von paralleler Simulation [Kepp94].

Parallele und verteilte Simulation wurde durch den Zusammenfluss von drei wesentlichen zu Grunde liegenden Technologien ermöglicht:

- integrierter Schaltungen bzw. günstiger Rechner;
- hoher Kommunikationsgeschwindigkeit zwischen Computern;
- Modellierung und Simulation: Technologien, die die Konstruktion von Modellen ermöglichen.

Parallele Simulationen werden auf einer Menge von Computern ausgeführt, die typischerweise durch ein einzelnes Gehäuse oder durch ein Zimmer begrenzt bzw. durch einen speziellen Hochgeschwindigkeits-Systembus verbunden sind, während verteilte Simulationen auf Rechnern ausgeführt werden, die sich geographisch in unterschiedlichen Gebäuden befinden können oder weltweit verteilt sind. Es existieren mehrere Vorteile bei einer Simulation über mehrere Rechner hinweg im Vergleich zu einer Simulation, die auf einer sequenziellen von Neumann-Architektur basiert:

- reduzierte Ausführungszeit, vor allem bei komplexen Systemen oder hochauflösenden Simulationen. Durch das Unterteilen umfangreicher Berechnungen in mehrere Teilberechnungen und das gleichzeitige verteilte Ausführen mit n Prozessoren lässt sich die Ausführungszeit theoretisch bis um den Faktor n reduzieren, falls das Modell parallelisierbar ist;
- geographische Verteilung. Die Simulation ist somit flexibler hinsichtlich der lokalen Verfügbarkeit von HW und SW, was z. B. auch für kooperierende Projektpartner interessant ist;
- Integration von Simulatoren, die auf Rechnern oder BS verschiedener Hersteller ausgeführt werden. Anstatt diese Programme auf einen einzelnen Rechner zu portieren, kann es kostengünstiger sein, die existierenden Simulatoren zu koppeln oder spezielle HW einzusetzen. Bestehende Modell-Bibliotheken können weiterverwendet werden;
- Fehlertoleranz. Es besteht für andere Prozessoren die Möglichkeit, die Arbeit des ausgefallenen Rechners trotz des Fehlers wiederaufzunehmen und fortzusetzen. Im Gegensatz dazu müsste eine auf einen einzelnen Prozessor abgebildete Simulation stoppen, wenn ein Fehler eintreten würde. Das spielt insbesondere bei sicherheitskritischen Einrichtungen wie z. B. militärischen oder zivilen Flugkontrolleinrichtungen oder allgemein bei zeitkritischen Entscheidungsfindungsprozessen eine Rolle;
- Skalierbarkeit.

Als Parallelrechner werden häufig homogene Rechner eingesetzt, die Prozessoren eines einzelnen Herstellers verwenden. Die Verzögerung bei der Übertragung einer Nachricht von einem Rechner zum anderen (die Kommunikationsverzögerung), ist bei Parallelrechnern relativ gering und liegt typischerweise im Bereich weniger Mikrosekunden. Diese Verzögerung hat einen großen Einfluss auf die Performance. Bei großen Verzögerungen können die Wartezeiten der Rechner auf auszuliefernde Nachrichten groß sein (Tabelle 2).

	Parallelrechner	verteilter Rechner
Physikalische Ausdehnung	Ein Raum	Einzelnes Gebäude oder global
Prozessoren und BS	homogene CPU- und BS-Landschaft	oftmals heterogen
Kommunikationsnetzwerk	individuell eingerichteter Verteiler	kommerzielles LAN oder WAN
Kommunikationsverzögerung	weniger als 100 µs	mehrere 100 µs bis einige s

Tabelle 2: Gegenüberstellung paralleler und verteilter Rechner

Heutzutage werden für verteilte Simulationen am häufigsten UNIX[31]-basierte Workstations und PCs eingesetzt. Verteilte Rechner verwenden Verbindungen, die auf verbreiteten Telekommunikationsstandards wie Asynchronous Transfer Mode (ATM) oder Ethernet basieren.

Die Antwortzeit verteilter Rechner ist wesentlich höher als die von Parallelrechnern, weil die Signale längere Strecken zurücklegen müssen und i. A. komplexe SW-Protokolle eingesetzt werden, um autonome Rechner verschiedener Hersteller miteinander zu verbinden, anstatt Hard- und SW, die für eine bestimmte Verbindungsart entworfen wurde. Der Preis für Allgemeingültigkeit ist Performance, da komplexe SW-Protokolle die Antwortzeiten im Vergleich zu denen von Parallelrechnern um eine oder zwei Größenordnungen erhöhen. LAN-basierte Systeme bestehen aus Rechnern, die sich in einem begrenzten geographischen Gebiet befinden, wie z. B. einem einzelnen Gebäude. MAN-basierte (Metropolitan Area Network) Systeme besitzen die physikalische Ausdehnung einer Stadt, und WAN-basierte (Wide Area Network) Systeme können über ein Land oder die Welt verteilt sein [Fuji00].

2.3.2.7 Simulation und künstliche Intelligenz

Zur Kopplung von Simulation und künstlicher Intelligenz (KI) werden biologisch inspirierte Techniken wie z. B. Neuronale Netze (NN), Genetische Algorithmen (GA) und Fuzzy Logic (FL) untersucht. Algorithmen aus der KI sind u. a. geeignet für Partitionierung und Scheduling. Die Verwendung solcher Algorithmen hat Vor- und Nachteile. Bei NN wird z. B. ein Training benötigt, bevor sie vorteilhaft eingesetzt werden können. Vor der Verwendung von GA muss ein Modell parallelisiert werden, oder es wird eine inhärente Modellparallelität vorausgesetzt, die bei den für diese Arbeit relevanten Anwendungen i. A. nicht gegeben ist. Ein genetischer Algorithmus verwendet Strategien aus der Evolutionstheorie, um zu einem Problem eine möglichst gute Lösung zu finden [DuIn01].

Bei der Partitionierung mit GA werden mehrere Lösungen verglichen, anstatt eine, wie bei NN, zu optimieren. Allerdings sind eine komplexere Repräsentation als bei einer graphentheoretischen Lösung sowie eine inhärente Parallelität vorhanden. Letzteres ist vorteilhaft bei Parallelrechnern mit vielen CPUs. Mit Ausnahme von FL reagieren oben genannte Techniken adaptiv auf nicht prädizierte Veränderungen. Sie sind intelligent, aber rechenintensiv, insbesondere das Online-Training bei NN.

2.4 Einsatz von Mikrocontrollern und FPGAs

Mikrocontroller (µCs) sind das dominierende Bauelement im Bereich eingebetteter Systeme. Ein µC ist ein vollständiges Mikrocomputersystem (Rechnersystem) auf einem Chip. Neben der CPU sind Programm- und Datenspeicher, Peripheriekomponenten und Unterbrechungssystem (Schnittstellen, Timer, A/D- und D/A-Wandler usw.) gemeinsam auf einem Chip integriert und über einen oder mehrere Busse miteinander verbunden. µCs sind i. A. von außen nicht direkt sichtbar und verfügen im Gegensatz zum PC nicht über eine direkte Bedienschnittstelle zum Benutzer. Im Gegensatz zu mechanischen, elektrischen und hydraulischen Bauteilen, die analoge Komponenten darstellen, werden

[31] Bei UNIX handelt es sich um eine Familie von Betriebssystemen, die 1973 von den AT & T Bell Laboratories entwickelt wurde. UNIX ist ein BS für mehrere Benutzer und Mehrprogrammbetrieb [DuIn01].

Eingangsgrößen von µCs digital verarbeitet. Bild 28 und Bild 29 zeigen den prinzipiellen Aufbau eines µC [BeHa01] [SiSi03].

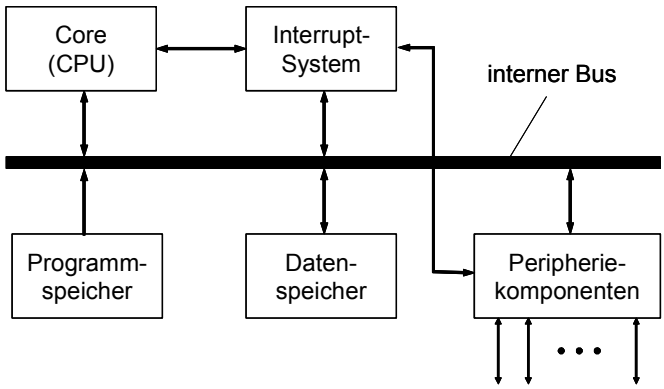


Bild 28: Prinzipieller Aufbau eines Mikrocontrollers (1)

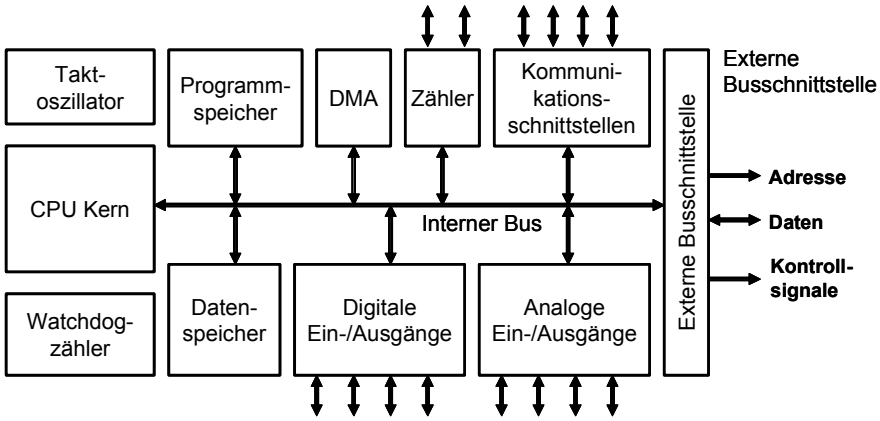


Bild 29: Prinzipieller Aufbau eines Mikrocontrollers (2)

Beispiele für integrierte Peripheriekomponenten:

- Takterzeugung: Auf dem Chip ist eine Oszillatorschaltung untergebracht, die nur wenige externe Komponenten benötigt (Quarz, Widerstände, Kondensatoren), um den Taktgenerator zu komplettieren.
- Ein- und Ausgabeports (E/A-Ports).
- Timer.
- Kommunikationsschnittstellen.
- A/D-, D/A-Wandler.
- Echtzeituhr (Real Time Clock, RTC).
- WatchDogTimer (WatchDog Zeitgeber, Watchdogzähler, WDT): Um das System eigenständig aus einem Fehlerzustand herauszuholen, wird ein Watchdogzähler verwendet, der von der SW regelmäßig mit großen Werten vorgeladen wird und eigenständig abwärts zählt. Bei funktionierender SW sollte er niemals den Wert 0 erreichen. Bei 0 erfolgt ein Rücksetzen (Neustart) des Systems.
- Ansteuerung von LCD- (Liquid Crystal Display) und LED (Light Emitting Diode)-Anzeigebauelementen.
- Stromsparfunktionen.
- Debug-Unterstützung.

- DMA (Direct Memory Access): Eine DMA-Einheit transferiert Daten eigenständig, schnell und ohne Eingriff der CPU zwischen Einheiten des Controllers.

In Bild 30 sind ein Mikroprozessor-basiertes eingebettetes System (links) und ein Mikrocontroller-basiertes eingebettetes System (rechts) gegenübergestellt.

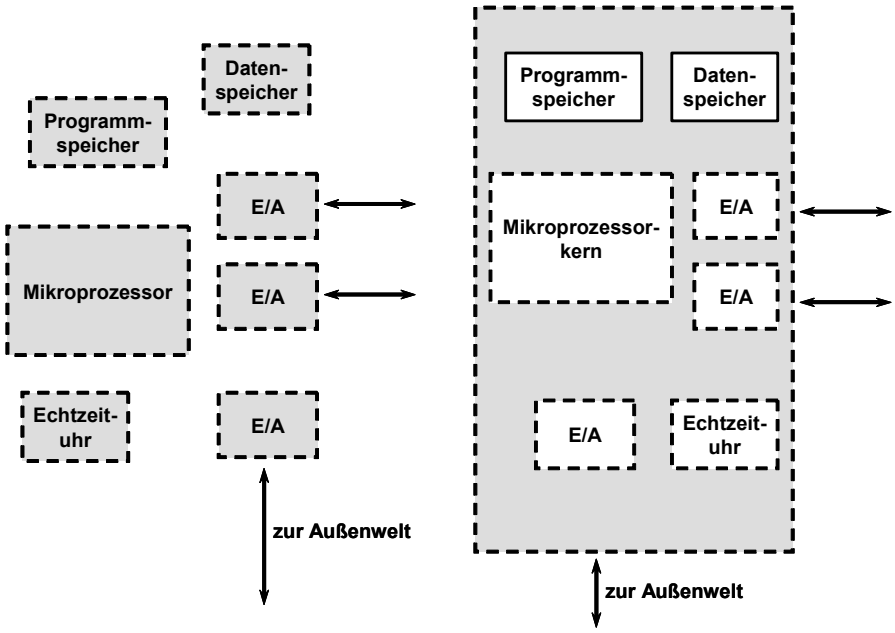


Bild 30: Mikroprozessor- und µC-basiertes eingebettetes System

Bei einem Mikroprozessor-basierten System sind CPU und die verschiedenen Ein-/Ausgabe-Funktionen getrennte ICs. Vorteile eines µC gegenüber einem Mikroprozessor und weiteren Zusatzkomponenten sind:

- höhere Integration, dadurch niedrigere Kosten;
- höhere Zuverlässigkeit, weniger Pakete, weniger und kürzere Verbindungen;
- bessere Performance, da Systemkomponenten für ihre Umgebung optimiert sind und die Signale auf dem Chip bleiben;
- weniger HF-Probleme (Hochfrequenz-Probleme) durch geringere Größe im Vergleich z. B. zu einer PC-Platine.

In der vorliegenden Arbeit kamen µCs des Typs Motorola HC12, Infineon C166 und Motorola PowerPC 555 zum Einsatz (siehe 4.2). Der HC12 ist eine 16-Bit-CISC-Architektur (Complex Instruction Set Computer). Die Datenwortbreite der Register und der Datenbusse beträgt 16 Bit. Ebenso weist der Adressbus 16 Bit Breite auf, was eine primäre Adressierfähigkeit von 64 KByte ergibt, die durch Paging-Verfahren auf über 4 MByte erweitert wird [Moto05]. Die C166-Familie ist eine 16-Bit-Architektur, die von Infineon Technologies [Infi05] entwickelt wurde. Sie ist eine Verschmelzung aus RISC (Reduced Instruction Set Computer) und CISC, um gleichzeitig eine hohe Performance und hohe Code-Dichte zu erzielen. Bei der Evaluierung des OSEK/VDX-BS kam der Typ C167 zum Einsatz [C16697] [Joha93]. Die PowerPC-Architektur ist eine RISC-Architektur. Die MPC500-Familie basiert auf einer 32-Bit-Wortbreite [MPC500]. Bild 31 zeigt eine komplette Single-Chip-Implementation für den MPC555 [BeHa01]. Für ausführlichere Beschreibungen sei an dieser Stelle auf die Reference-Manuals der jeweiligen Hersteller verwiesen. Der Einsatz eines µC, der sich als Standardschaltkreis zu günstigen Preisen erwerben lässt, stellt in vielen Fällen alternativlos die beste Wahl dar.

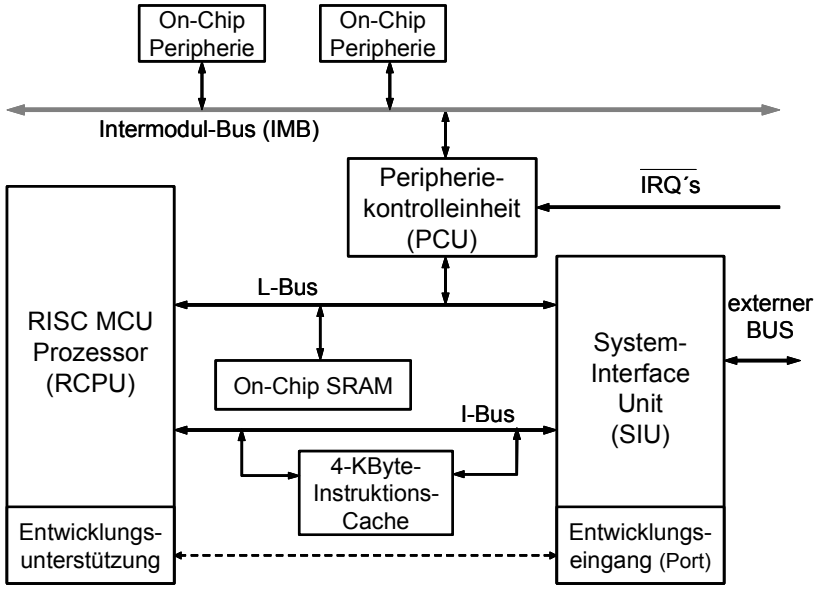


Bild 31: MPC555 Blockschaltbild

Bei der Umsetzung von sonstigen Schaltungsteilen, die sich alternativ aus der Zusammenschaltung von Standardlogikschaltkreisen und LSI- (Large Scale Integration) bzw. VLSI-Schaltkreisen auf der Leiterplatte realisieren lässt, wird man heute aus Gründen der höheren Zuverlässigkeit, des Energiebedarfs und der Miniaturisierung eine integrierte Lösung bevorzugen, die mittels FPGAs realisiert werden kann. Ob dies wirtschaftlich ist, und welcher Typ gewählt werden sollte, hängt neben der Komplexität der Schaltung und der benötigten Stückzahl an FPGAs auch von weiteren Faktoren ab [BeHa01]. Der Preisverfall hat dazu geführt, dass heute statt ASICs häufig Standard-Bausteine oder FPGAs eingesetzt werden, die über SW konfiguriert werden. Für Lösungen auf der Basis von System-on-Chip-Technologien sind FPGAs geeignet, da unterschiedliche Anpassungslogiken, Dekodierer bzw. Funktionen von DSPs im Baustein integrierbar sind. Ferner sind FPGAs den ASICs gegenüber im Vorteil, da sich FPGAs rekonfigurieren lassen [DeE02b], so dass Systeme auch noch spät im Entwicklungsprozess hinsichtlich geänderten Standards und Protokollen angepasst werden können. Weitere Vorteile, durch die sich geringere Entwicklungskosten ergeben [Zünd02]:

- hohe Integrationsdichte der Digitalschaltungen: FPGAs ermöglichen die Integration zahlreicher Systemfunktionen in Form einer Single-Chip-Lösung und tragen somit zu Platz- und Kosteneinsparungen bei;
- schnelle Integration;
- durch Rapid Prototyping ist Markteinführungszeit reduziert;
- kurze Design- bzw. Redesignzeiten;
- spezielle Testaufbauten entfallen;
- risikoarme Herstellung von Prototypen und Serienstückzahlen;
- Emulation mit programmierbarer HW anstelle zeitaufwendiger Simulation ist möglich;
- FPGAs können im Gegensatz zu ASICs vom Anwender selbst programmiert werden können. Dadurch können sie ASICs bei Kleinserien ersetzen oder bei ihrer Entwicklung als Prototypen dienen, da sich Fehler schnell und einfach korrigieren lassen.

FPGAs sind bei Kleinserien eine preiswerte Alternative zu ASICs. Nachteilig ist, dass FPGAs bei umfangreichen Schaltungen und Großserien im Vergleich zu ASICs noch teuer sind.

Die Logikblöcke eines FPGAs sind reihenweise angeordnet. Zwischen den Logikblöcken befinden sich die Verdrahtungskanäle. Aus Bild 32 ist der prinzipielle Aufbau ersichtlich [RePo99]. In einem

FPGA sind Ein-/Ausgabe-Logik, die Logikblöcke und die Verdrahtung frei programmierbar. Das Zeitverhalten wird weitgehend durch die Verdrahtung bestimmt.

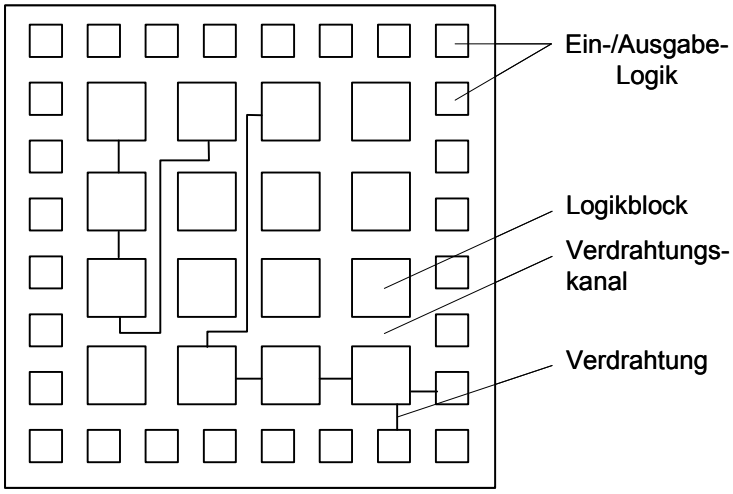


Bild 32: Aufbau eines FPGAs

2.5 Gesetz von Moore

Gordon Moore, Mitbegründer der Firma Intel Corp., sagte 1958 voraus, dass sich die Dichte der Transistoren und damit die Anzahl der auf einem integrierten Halbleiterbaustein unterzubringenden Transistorfunktionen alle zwölf Monate verdoppeln würden [Moor65]. Die Vorhersage, die als „Mooresches Gesetz“ berühmt wurde, musste zwar später auf 18 Monate korrigiert werden, hat ihre Gültigkeit aber prinzipiell behalten; die Entwicklung der Halbleitertechnologie folgt seit über 20 Jahren dem Gesetz von Moore. Dieses exponentielle Wachstum ermöglicht es – bei annähernd gleichbleibendem Preis pro Einheit –, die Komplexität von Grundkomponenten von Rechenanlagen, die Rechenleistung sowie die Zuverlässigkeit des Gesamtsystems durch Reduktion der Anzahl der für den Aufbau benötigten Grundbausteine, zu erhöhen. Im Vergleich dazu schreitet die Entwicklung von Algorithmen langsamer voran. Alleine durch den Anstieg von Hardwarekapazitäten lassen sich die Probleme nicht lösen. Es sind auch neue Wege im SW-Bereich nötig, um die Leistungsengpässe zu beheben.

Darüber hinaus schreitet der Zuwachs der Speichergeschwindigkeit wesentlich langsamer voran. Aus diesem Grund sind z. B. optimistische Synchronisationsalgorithmen aufgrund ihres inhärenten Speichermehrbedarfs durch Zustandsspeicherung (state saving, checkpointing) im Vergleich zu konservativen Algorithmen hinsichtlich ihrer Performance zunehmend benachteiligt, vor allem bei hoher Ereignisgranularität. Somit bietet die Optimierung optimistischer Algorithmen ein größeres Performance-Wachstumspotential als es bei konservativen Algorithmen der Fall ist (siehe 3.6).

3 Stand der Technik

3.1 Echtzeit-Betriebssystem OSEK/VDX

OSEK ist ein seitens der Automobilindustrie und -zulieferer initiiertes Projekt zur Standardisierung einer Architektur für verteilte Kontrolleinheiten im Automobil. VDX ist ein Projekt der französischen Automobilindustrie mit einem ähnlichen Ziel wie OSEK. Das gemeinsame Projekt heißt OSEK/VDX, siehe [Coom01] und [Tind00] und ist die Bezeichnung für die Standardisierungsbestrebungen, die vor allem auf die europäische Automobilindustrie zurückgehen. Ziel ist es, eine einheitliche Programmierumgebung für Steuer-SW im Automobil bereitzustellen. In öffentlich zugänglichen Dokumenten [OSEK03] definiert das OSEK-Kommittee die Schnittstellen für Applikationsprogrammierung (Application Programming Interfaces, APIs) sowie das Systemverhalten.

Die Spezifikation definiert eine Standard-BS-Schnittstelle für ECU-Applikationen im Automobil: das Verhalten des BS, die Laufzeitdienste (runtime services) und die API zu eingebetteten RTOS. Die API abstrahiert von spezifischer und darunterliegender HW und der Konfiguration von Automobil-internen Netzwerken [OSEK03]. Im Einzelnen definiert der OSEK/VDX-Standard APIs für ein RTOS (OSEK OS, Echtzeit-Ausführungseinheit für ECU-SW und Basis für andere OSEK-Module), Kommunikation (OSEK COM, Datenaustausch innerhalb und zwischen Steuergeräten) und Netzwerkmanagement (OSEK NM, Konfigurationserkennung und Überwachung) (Bild 33).

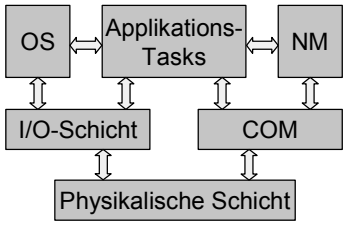


Bild 33: OSEK/VDX-Standards

Die Spezifikation gibt die Rahmenbedingungen für ein echtzeitfähiges Multitasking-BS vor. Der Standard macht Vorgaben bezüglich des Task-Managements, der Task-Synchronisation, des Unterbrechungs-Managements sowie des Umgangs mit Alarmen und der Fehlerbehandlung. OSEK unterstützt keine dynamische Erzeugung oder Vernichtung von Systemobjekten (Tasks, Ereignisse, Alarme) zur Laufzeit. Ein OSEK OS besitzt einen statischen Kern; eine OSEK-Applikation wird statisch zur Compilierungszeit mit der OSEK Implementation Language (OIL) konfiguriert und alle Kern-Objekte sind zur Compilierungszeit bekannt, d. h. die Anzahl der Tasks und ihrer Ressourcen muss bei der Systemerstellung feststehen. Als Ergebnis wird die Menge an Tests, die benötigt wird, um ein korrektes Verhalten der Applikation sicherzustellen, reduziert. Das ist wichtig für Applikationen, die eine Zertifizierung benötigen. Das zeitliche Verhalten der dynamischen Allokation und Deallokation würde sonst wesentlich von Faktoren abhängen, die schwierig oder gar nicht vorauszusagen sind, z. B. der Speicherfragmentierung. Es sind keine Funktionen vorhanden, die es erlauben, die Priorität eines Tasks zur Laufzeit zu verändern. Dadurch können die für das Prioritätsmanagement nötigen Parameter bereits bei der Systemkonfiguration berechnet werden. Dieses „Priority ceiling“ genannte Vorgehen verhindert Prioritätsinversionen unter OSEK OS.

Jede OSEK-Anwendung besteht aus dem eigentlichen Programmcode nebst einer Anwendungsdefinition. Für letztere steht die OIL zur Verfügung. Die Anwendungsdefinition legt alle Systemressourcen fest, die die Anwendung benötigt. Bei der Systemerzeugung liest ein System-Generator-Werkzeug diese Anwendungsdefinition aus und erzeugt C-Code mit den Datenstrukturen sowie dem Initialisierungscode für die Anwendung. Aus dem C-Programm im Verbund mit dem Anwendungscode sowie dem OSEK-Kernel entsteht dann das ausführbare Binär-Image. OSEK macht keine Angaben bzgl. der E/A-Funktionalität. Solche Tätigkeiten sind Aufgaben von Modulen, die nicht der Spezifi-

kation unterliegen. Dennoch spezifiziert OSEK eine Schnittstelle für die Unterbrechungsverarbeitung, die die Erstellung von fest in die Anwendung eingebundenen Gerätetreibern vereinfacht.

OSEK spezifiziert vier „Conformance Classes“. Jede dieser Klassen begrenzt die verfügbaren Systemressourcen wie Task-Prioritäten, Zeitmanagement und Systemdienste. Obwohl neben 8-Bit-Controller auch 64-Bit-Prozessoren mit mehreren Megabyte Speicher als Zielsysteme unterstützt werden, wird so überflüssiger Code reduziert [DeE02a]. Die Basic Conformance Classes BCC1 und BCC2 (BCCx) erfüllen die Anforderungen „tief“ eingebetteter Automotive-ECUs, während die Extended Conformance Classes ECC1 und ECC2 (ECCx) entworfen wurden, um „high-end“-Anforderungen zu genügen, bei denen eine intensivere Verwendung von Ressourcen nicht so kritisch ist. Die Unterstützung von ECCx vereinfacht die Abbildung von Ereignissen höherer Ebene in CASE-Werkzeugen im Kontext automatischer Code-Generierung [DrSM01] [Scho99]. BCC2 und ECC2 werden verwendet, wenn zwei oder mehr Klassen dieselben Prioritäten besitzen. Conformance Classes erlauben eine teilweise Implementierung entlang vordefinierten Linien, die als OSEK-konform zertifiziert werden können [Lemi00]. Eine typische OSEK-Entwicklungsplattform wird in [Blon01] beschrieben.

3.1.1 Verifikation zeitlicher Anforderungen mit OSEKtime

Der OSEKtime-Standard[32] definiert für jeden einzelnen Knoten im fahrzeugweiten Steuergerätenetz ein BS, das harte Echtzeitanforderungen durch das Binden der Tasks an feste Zeitpunkte erfüllt. Insbesondere stellt das System Dienste bereit, um für jede Task eine Überwachung von Deadlines zu ermöglichen. Unter OSEKtime ist die Aktivierung mit der sofortigen Ausführung gleichzusetzen. Im Gegensatz zu OSEK/VDX können Tasks nicht auf Ereignisse warten oder andere Tasks durch das Belegen von Ressourcen am Laufen hindern. So wird es möglich, für jede Task eine WCET für die Erledigung ihrer Aufgabe oder Lieferung eines Ergebnisses zu bestimmen. Für die Realisierung der X-by-Wire-Dienste stellt OSEKtime eine wichtige und standardisierte Basis dar [Scho01] [Schu02]. Z. B. ermöglicht OSEKtime die Überprüfung von Deadlines. Das Time Compiler-Werkzeug von LiveDevices erlaubt die Überprüfung, ob ein System seine Performance-Anforderungen einhält. Der „Time Compiler“ ist eine Komponente des SSX5-Toolsets, das die erforderlichen Berechnungen für das zu verifizierende System als Teil des Entwicklungsprozesses ausführt. Mit Hilfe dieser Methode lässt sich das Zeitverhalten im ungünstigsten Fall (worst-case) bestimmen. Die Echtzeit-Verarbeitung kann dahingehend untersucht werden, ob der Ablauf aller Aktivitäten (Tasks) auf jeder einzelnen CPU des Systems die Zeitanforderungen erfüllt. Der komplette Ablauf aller verschachtelten Tasks einschließlich der dazugehörigen Kommunikation über den seriellen Bus ist damit abschätzbar und als ausreichend bewertbar. Neben der Berechnung von Worst-Case-Antwortzeiten und der Verifizierung auf Einhaltung aller Deadlines kann der Time Compiler auch eine sogenannte „Empfindlichkeitsanalyse“ durchführen. Sie gibt an, wie nahe das System an das Verpassen einer Deadline herankommt, d. h. wieviel Reserve noch vorhanden ist [Tind99].

3.1.2 Profiling mit Hilfe von WindRiver WindView

WindView© ist ein Werkzeug zur SW-Diagnose und -Analyse, das auf den OSEK-Standard aufsetzt und in die SW-Entwicklungsumgebung Tornado for Automotive Control© integriert ist. WindView ermöglicht Einblicke in die dynamischen Prozesse eines eingebetteten Systems. Interaktionen zwischen Aufgaben, Unterbrechungsbearbeitungsroutinen und Systemobjekten können auf dem Bildschirm des Entwicklungsrechners visualisiert werden. Der Code kann während der Ausführung verfolgt und der zeitliche Ablauf der Prozesse überprüft werden [AuEl00].

WindView ermöglicht eine Logikanalyse von Echtzeit-SW und zeigt dynamische Informationen über das Echtzeitsystem in einem Graphikfenster an [WiVi01]. Ereignisse werden in Echtzeit erfasst, und es ist möglich, die dynamische Interaktion der Systemelemente zu studieren. Nach dem Test der

[32] OSEKtime ist eine Erweiterung von OSEK für den Einsatz zeitgesteuerter Technologien.

Systemintegration kann die WindView-Instrumentierung von der Anwendung entfernt werden. Die Simulation von Unterbrechungen und Ein-/Ausgabeoperationen ist kostspielig und aufwändig. Deshalb wird für Targets ohne Background Debug Mode (BDM)- oder JTAG[33]-Schnittstellen der Einsatz eines ROM-Monitors (Read Only Memory-Monitors) empfohlen. Durch den Einsatz eines Multitask-Debuggers können Systemressourcen wie z. B. Task-Listen, Nachrichten (messages), Zähler (counters), Alarms und Stacks angezeigt werden; bedingte Haltepunkte (conditional breakpoints) sollten unterstützt werden. Zusätzlich können durch die Verwendung von Dual-Port-Speicher eines Emulators diese Bereiche nicht-intrusiv in Echtzeit angezeigt werden. Es kann eine statistische Auswertung von Task-Laufzeiten und Task-Umschaltungen durchgeführt werden. Dieses Profiling ist notwendig, da die Performance des BS abnimmt, wenn ein Dienst viel Zeit benötigt und sehr oft aufgerufen wird.

Die im Rahmen dieser Arbeit entwickelte Profiling-Methode basiert im Vergleich zu WindView auf Simulink, so dass entsprechende Anzeige- und Weiterverarbeitungsmöglichkeiten der Daten vorhanden sind. Darüber hinaus wird beim Profiling das später im Fahrzeug eingesetzte CAN-Bussystem verwendet (siehe 3.2.1).

3.2 Bussysteme im Automobil-Bereich

In der Vergangenheit kamen oftmals proprietäre Lösungen zum Einsatz, die heute im Antriebsstrang und Karosseriebereich durch CAN-Module mit ihren Varianten High- und Low-Speed ersetzt werden. High-Speed-CAN [CAN94] kann z. B. im Bereich des Antriebsstrangs und Fahrwerks eingesetzt werden, Low-Speed-CAN [Seri94] z. B. im Komfortbereich. In der Vergangenheit gab es verschiedene ereignisgesteuerte Kommunikationssysteme wie A-BUS, VAN und M-Bus. Keines dieser Protokolle wurde zu einem weit verbreiteten Standard [Buhl02]; die meisten blieben herstellerspezifisch oder verschwanden nach kurzer Zeit wieder vom Markt. Zwar findet die CAN-Technologie in weiten Bereichen Anwendung, sie kann aber nicht jede Anforderung an Steuergerätevernetzungen ausreichend abdecken. Im preissensitiven Sektor der Body-Steuergeräte (Karosserie- und Komfort-Elektronik) stellt LIN (Local Interconnect Network) eine weit verbreitete Lösung dar. Sie ist im Vergleich zu Low-Speed-CAN etwa halb so teuer pro Knoten [LIN05]. Der Multimedia- oder Infotainment-Sektor verlangt nach hohen Datenraten und streng zeitsynchroner Datenübertragung. CAN ist hier überfordert. Aus Gründen der elektromagnetischen Verträglichkeit (EMV) werden hier bevorzugt Lichtwellenleiter eingesetzt. Multimedia-Systeme, die hohe Datenraten verlangen, können z. B. mit MOST© (Media Oriented System Transport) [MOST] verbunden werden. Für sicherheitsrelevante X-by-Wire-Systeme (wie Brake-by-Wire und Steer-by-Wire) werden Hochleistungsbusse benötigt, die fehlertolerant und daher überwiegend redundant und verteilt ausgelegt sind. Sie müssen Daten mit geringem zeitlichen Jitter[34] in streng deterministischer Weise übertragen. Ein für diese Anwendung geeignetes Protokoll ist z. B. FlexRay [DaBe01]. Die Kommunikation erfolgt im Rahmen eines Kommunikationszyklus, der aus einem statischen und einem dynamischen Teil besteht. Die Grenze zwischen beiden Teilen kann flexibel festgelegt werden, wobei auch einer der Teile leer sein kann. Der statische Teil des Übertragungszyklus besteht aus einer frei konfigurierbaren Anzahl von Sendeslots identischer Länge. Eine gute Übersicht zu FlexRay findet sich z. B. in [ElA01a] [Baue02] [HeSc02] und [Raus02]. Das LIN-Protokoll wurde für geringe Datenraten, wie z. B. bei Schaltereinstellungen, bei geringstmöglichen Kosten entwickelt. Es wird für einfache Ein/Aus-Kontrollen – wie Sitzkontrolle, Türverriegelung, Schiebedach, Regensensor, Lüftungsklappen, Tempomat, Scheibenwischer und Außenspiegel – und für andere Anwendungen zur Kommunikation mit mechatronischen Satelliten von Subsystemen eingesetzt, bei denen keine hohen Übertragungsraten erforderlich sind. Zur Verwendung im Bereich kostensensibler Karrosserieanwendungen kommt

[33] JTAG (Joint Test Action Group) ist ein Standard nach IEEE 1149.1. Das JTAG-Konzept ermöglicht es, die funktionellen Anschlüsse (Ein-/Ausgänge) eines integrierten Schaltkreises auf serielle Art und Weise abzufragen bzw. Testmuster auszugeben [Walt01].

[34] Jitter sind Laufzeitschwankungen zwischen maximaler und minimaler Latenzzeit.

z. B. LIN [LIN05] für Subnetze in Frage [ElAu00] [Lasc02]. Das Hauptnetzwerk läuft unter dem CAN-Standardprotokoll, Unterknotenpunkte basieren auf dem LIN-Protokoll. Z. B. wird nach der Bestätigung des Empfangscodes der Befehl zur Türentriegelung über den CAN-Bus und das CAN-LIN-Gateway an den LIN-Busmaster ausgegeben, der dann seinerseits die Daten an den entsprechenden Knotenpunkt in seinem Netzwerk übermittelt [Schm00]. Der LIN-Bus ist ein Sub-Bussystem, das auf einem seriellen Kommunikationsprotokoll basiert. Der Bus ist ein Single-Master/Multiple-Slave-Bus, der zur Datenübertragung nur eine Leitung verwendet. Das bedeutet, dass allein der Master eine Kommunikation initiieren kann. Direkte Kommunikation zwischen Slaves ist nicht möglich [Gril02].

Zukünftige Fahrwerkregelsysteme stellen typische verteilte Regelungssysteme dar. Für den Einsatz im Bereich der sicherheitsrelevanten Systeme wie Brake- und Steer-By-Wire-Systeme sind deterministische Kommunikationssysteme notwendig. Mögliche Vernetzungstechnologien hierfür sind FlexRay [FlRa05], TTP [TTP05] und TTCAN (Time Triggered CAN) [TTCA02]. Charakteristische Probleme sind u. a. das Einhalten der Abtastzeiten, die Applikations-Synchronisation sowie die zeitliche Wirkungskette des verteilten geschlossenen Regelkreises. Zeitgesteuerte Systemarchitekturen eignen sich für die Realisierung verteilter Regelungen. Sie besitzen eine global synchronisierte Zeitbasis. Zu festgelegten Zeitpunkten macht das System eine Momentaufnahme der Systemumgebung und aktualisiert die Zustandsgrößen bzgl. der synchronisierten Systemzeit. So können zeitsynchron Sensoren ausgelesen und Aktuatoren angesteuert werden [ElA01a] [Drei03]. FlexRay wurde von DaimlerChrysler und BMW initiiert.

Bild 34 zeigt einen kostenbasierten Vergleich zwischen LIN, CAN, FlexRay und MOST [DeE01a].

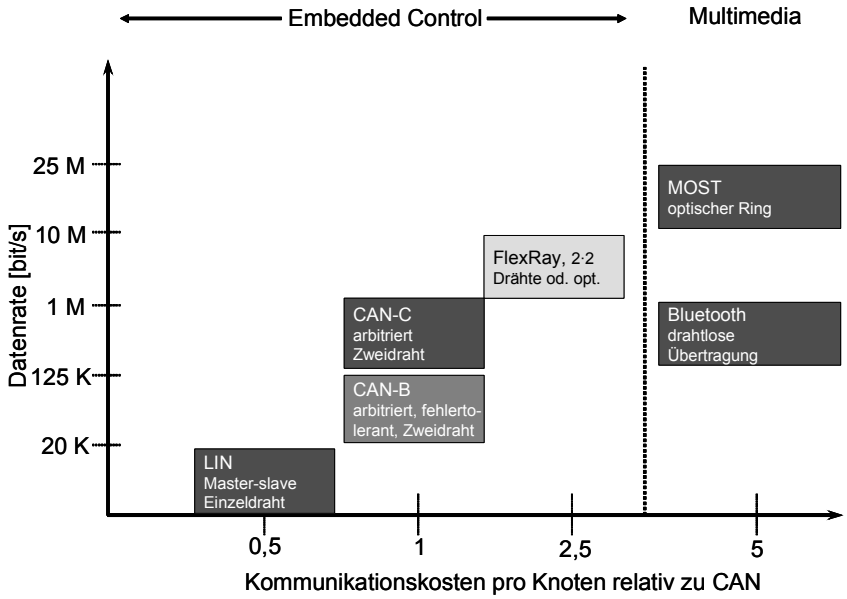


Bild 34: Kostenvergleich zwischen den Protokollen

3.2.1 CAN

Ein Protokoll, das sich im Kraftfahrzeug und in der Automatisierungstechnik etabliert hat, ist das CAN-Protokoll. CAN wird deshalb als Beispiel für den Einsatz eines realen Bussystems für die weiteren Untersuchungen ausgewählt (siehe 4.4). Dieses von der Robert Bosch GmbH (Bosch) entwickelte und in ISO (International Organization for Standardization) 11898 international standardisierte Bus-Protokoll wird auf Grund seiner Flexibilität und Robustheit für verschiedene Klassen von Fahrzeugnetzwerken eingesetzt. CAN wurde in Europa zum Standard. Diese Rolle wurde in den USA durch das Protokoll J1850 ausgefüllt. Wesentliche Vorteile von CAN sind das breite Geschwindigkeitsspektrum von 10 KBit/s bis 1 MBit/s, Störfestigkeit sowie die Fehlerbehandlungsmechanismen [Gril02]. In einer CAN-Botschaft können bis zu 8 Datenbytes übertragen werden. Jede

Nachricht verfügt über einen Identifier, der die Nachricht eindeutig charakterisiert. Der Identifier ist mit einer Priorität behaftet, d. h. die Nachricht mit dem niedrigeren Identifierwert erhält bei gleichzeitigem Buszugriff das Senderecht. Die im CAN-Protokoll definierte Busarbitrierung erfolgt zerstörungsfrei, so dass die höchstpriore Nachricht ohne Verzug den Buszugriff bekommt. Das CAN-Protokoll unterstützt eine ereignisgesteuerte Kommunikation. Wenn ein Ereignis eintritt, das über den Bus kommuniziert werden soll, wird ein Sendeversuch gestartet. Hochpriore Nachrichten kommen schnell durch, und Nachrichten mit niedriger Priorität können erfolgreich gesendet werden, sobald keine hochprioren Sendeanforderungen mehr anstehen [Müll01].

CAN beruht auf einer reinen Ereignissteuerung, d. h. Informationen werden immer beim Eintreten eines Ereignisses ins Kommunikationssystem eingestellt. Die Steuerung der Priorität erfolgt mit Hilfe einer der Nachricht vorangestellten Identifikation. Je höher die Priorität ist, umso größer ist die Wahrscheinlichkeit, dass sich diese Nachricht am Bus durchsetzen wird. Der Vorteil liegt in der Flexibilität des Systems. Es ist einfach, neue Nachrichten in ein vorhandenes Netz einzufügen, und der Zugriff auf den Bus kann ohne Verlust von Botschaften stattfinden. Dieses Verfahren verhindert in großen Netzen eine Erhöhung der Übertragungsrate auf über 1 MBit/s. Durch die Prioritätssteuerung kommt es zu zeitlichen Abhängigkeiten innerhalb eines Netzes. Je höher die Auslastung des Busses ist, desto größer wird der zeitliche Versatz des Informationstransports und damit die Unsicherheit, ob und wann eine Nachricht beim Empfänger ankommt [Buhl02].

Im Antriebs- und Fahrwerksbereich sind heute CAN-Datenbussysteme im Einsatz, die bis zur Geschwindigkeit von 1 MBit/s spezifiziert sind. Aufgrund fahrzeugtypischer Randbedingungen sind die Datenraten im Serieneinsatz auf 500 KBit/s begrenzt. Die Evolution der Vernetzung im Automobil stößt damit bereits heute an die Performance-Grenzen (Bild 35 [ElA01a]).

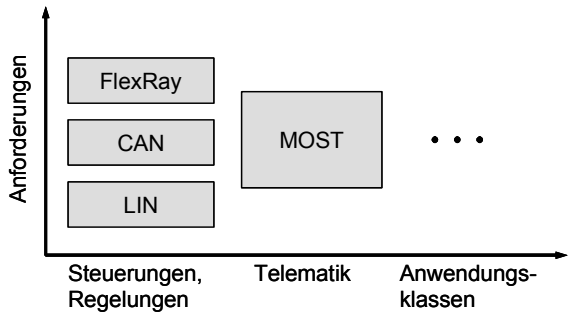


Bild 35: Anforderungen und Anwendungsklassen von Bussystemen

Wesentliche Merkmale des CAN-Protokolls:

- Ausgetauscht werden Nachrichten in einem festen Format, die durch einen „Identifier“ eindeutig gekennzeichnet sind und eine eindeutige Priorität haben. Dieser Identifier beschreibt den Inhalt der Nachricht, sagt aber nichts über deren Sender oder die potentiellen Empfänger aus.
- Jeder Knoten am Netz hört alle Nachrichten und entscheidet, welche für ihn wichtig sind.
- Ein Knoten versucht dann zu senden, wenn ihm sein zugehöriger Rechner dazu einen Auftrag gibt und die Daten geliefert hat. Der Zeitpunkt dafür wird von der Applikation bestimmt. Dieses Vorgehen heißt ereignisgesteuerte Kommunikation.
- Wenn ein Knoten eine Nachricht nicht korrekt empfangen hat, meldet er dies an alle Knoten. Dadurch wird erreicht, dass alle Netzteilnehmer das gleiche Bild über die Gültigkeit eines Datums haben.
- Ein Knoten kann mit dem Senden nur beginnen, wenn der Bus frei ist. Er kann also keine Nachrichten anderer Knoten zerstören, um selber zu senden. Wenn mehrere Knoten gleichzeitig versuchen, Nachrichten zu senden, legt die Priorität der Nachricht den Vorrang fest. In diesem Fall wird durch den Mechanismus der zerstörungsfreien Arbitrierung dafür gesorgt, dass

sich die Knoten, die weniger wichtige Nachrichten haben, vom Sendeversuch zurückziehen und ihn auf später verschieben.

- Die Nachrichten werden durch zusätzliche Bits (Cyclic Redundancy Check, CRC) gegen fehlerhafte Übertragung gesichert. Ist eine Nachricht nicht bei allen Knoten korrekt angekommen, was durch „Error Frames“ oder fehlendes „Acknowledge“-Signal signalisiert wird, versucht der Sender eine erneute Übertragung.
- Jeder Knoten bewertet seinen eigenen Zustand durch Führen interner Zähler – jeder auftretende Fehler erhöht den Zählerstand. Werden Schwellwerte überschritten, hält der Knoten sich selbst für defekt und schränkt seine Aktivität ein.
- Bei der Auslegung eines CAN-Busses ist zu beachten, dass jeder Identifier nur einmal vergeben wird. Das erlaubt es, das Netz flexibel zu erweitern, ohne dass bereits definierte Knoten nachträglich zu ändern sind.
- Die Bitrate beträgt bis zu 1 MBit/s. Das ist eine physikalisch bedingte Grenze, die von dem gewählten Verfahren der bitweisen Arbitrierung herrührt; sie ist bei Netzen größerer räumlicher Ausdehnung niedriger. Jedoch steht netto nur ein Bruchteil dieser Bitrate zur Verfügung, da zu den reinen Informationsbits noch die für das Übertragungsprotokoll erforderlichen Daten (Protokoll-Overhead) hinzugefügt werden müssen. Der Bus muss zu einem großen Teil der Zeit frei sein, damit ein Knoten zu einer gegebenen Zeit mit einer vertretbaren Wahrscheinlichkeit senden kann [WeFü01].
- Der Informationsaustausch erfolgt objektorientiert. Ein System kann maximal 2032 Objekte nach CAN-Spezifikation 2.0A bei 11-Bit-Identifier verwalten. Es kann wahlweise ein erweiterter Objektumfang bei 29-Bit-Identifier nach CAN-Spezifikation 2.0B genutzt werden. Bild 36 zeigt den typischen Telegrammaufbau eines Rahmens nach CAN-Spezifikation 2.0A [BeHa01].

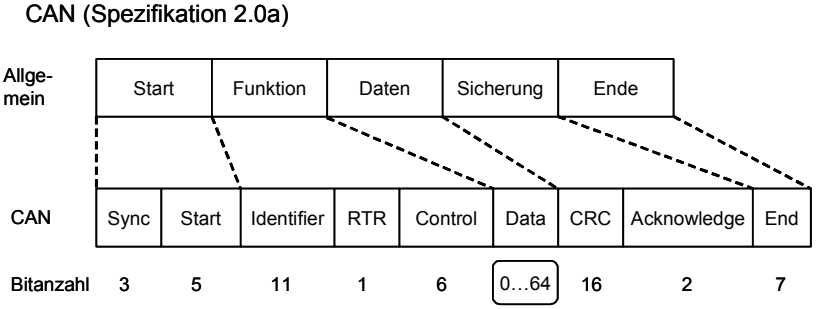


Bild 36: CAN-Telegramm

Das CAN-Telegramm enthält:

Sync:	rezessiver Pegel; danach kann gesendet werden,
Start:	dominantes Startbit; Beginn eines Frames; Synchronisation der Busteilnehmer,
Identifier:	Objekt (Adresse) und die Priorität der gesendeten Nachricht,
RTR:	Remote Transmission Request; in Telegrammen, die eine Aufforderung zum Datensenden enthalten,
Control:	Steuerinformationen (nachfolgenden Datenlänge),
Data:	Datenfeld (0-8 Byte),
CRC:	Prüfsumme der vorangegangenen Bits mit CRC-Polynom,
ACK:	Acknowledge; Busteilnehmer quittieren bei korrektem Empfang,
End:	7 rezessive Bits.

3.3 Einsatz von CASE-Werkzeugen

3.3.1 Modellierung und Simulation

Die mechanischen Komponenten des Automobils werden mehr und mehr durch elektronische Systeme abgelöst. Zukünftig werden 90 % aller Innovationen im Automobil in ihrer wesentlichen Funktionalität durch Elektronik geprägt sein [Hofm00]. Für die Realisierung wird aus Kostengründen überwiegend auf Standard-Komponenten zurückgegriffen, während die eigentliche Funktionalität in SW implementiert wird. Dieser Paradigmenwechsel – Ersatz von mechanischen durch elektronische Komponenten – wirkt sich nicht nur auf einzelne Systeme aus, sondern auf die gesamte Produktentstehungskette. Somit bestimmt die Elektronik zukünftig im Wesentlichen die Entwicklung, die Produktion, den Vertrieb und mit steigender Tendenz auch die Interaktion zwischen dem Fahrzeug und der Umwelt. Sehr komplexe Systeme müssen in immer kürzeren Entwicklungszeiten unter Berücksichtigung der hohen Qualitätsansprüche (Sicherheit, Wartbarkeit, Wiederverwendung) entworfen werden. Darüber hinaus sind eine steigende Baureihenvielfalt und eine Zunahme an neuartigen Systemen (z. B. Fahrerinformationssysteme) festzustellen, die sich wesentlich auf die Struktur der Systeme auswirken. Für eine effiziente durchgängige Entwicklung derartig komplexer Systeme ist eine modellbasierte Entwicklung unumgänglich.

Modelle ermöglichen neben einer leichten Änderbarkeit und der Simulation in frühen Phasen die Möglichkeit des Austauschs zwischen den Entwicklern. Außerdem kann aus den Zustandsdiagrammen automatisch der Implementierungscode generiert werden. Zu diesem Zweck wurden in den vergangenen Jahren unterschiedliche Werkzeuge, wie z. B. STATEMATE, MATRIXx, MATLAB und SDT auf ihre Eignung für den Einsatz in der Automobilindustrie untersucht und in vielen Projekten praxisnah eingesetzt. Die prinzipielle Eignung konnte sehr schnell nachgewiesen werden. Es blieben allerdings viele Wünsche offen, vor allem bei der Beschreibung automobilspezifischer Funktionen und der Durchgängigkeit des Entwicklungsprozesses, insbesondere hinsichtlich des Austausches von Entwurfsdaten. Bei der Entwicklung eines elektronischen Systems wird heute eine Vielzahl verschiedener Werkzeuge eingesetzt. Jedes dieser Werkzeuge verwendet neben einer eigenen Benutzeroberfläche meist ein spezifisches Datenformat, so dass ein effizienter Datenaustausch nicht gewährleistet und die Zusammenarbeit der Werkzeuge nur bedingt möglich ist. Wünschenswerte Anforderungen wie Offenheit und Erweiterbarkeit der Werkzeuge werden nicht berücksichtigt, auch eine Anpassung an spezifische Systembeschreibungssprachen ist nicht vorgesehen. Darüber hinaus verhindern die oben angeführten Mängel und die inhärent mitgelieferten Abläufe der Werkzeuge den Einsatz eines übergreifenden, durchgängigen Entwicklungsprozesses [Hofm00].

3.3.1.1 Simulink und Stateflow

MATLAB ist eine Sprache und eine Oberfläche für technische Berechnungen. Es besteht aus einem mathematischen Kern und Grafikwerkzeugen für technische Berechnungen, Datenanalyse, Visualisierung und für die Entwicklung von Algorithmen und Anwendungen. MATLAB kann mittels Toolboxen erweitert werden. Sie sind Sammlungen anwendungsspezifischer Funktionen, die Anwendungen wie die Signal- und Bildverarbeitung, den Entwurf von Regelungssystemen, Optimierungen, finanztechnische Anwendungen, neuronale Netze usw. unterstützen. Simulink ist eine Entwicklungsplattform für den Entwurf, die Simulation und die Analyse von dynamischen Systemen und Prototypen. Simulink bietet eine mit Blockdiagrammen operierende, graphische Programmierumgebung zur Modellierung von Systemen, die auf der mathematischen Kernfunktionalität von MATLAB aufbaut, und kann mit Hilfe von Blocksets erweitert werden. Blocksets sind Bibliotheken anwendungsspezifischer Simulink-Blöcke für unterschiedliche Anwendungsgebiete, z. B. für Steuerungen und Kommunikationssystemen, für die digitale Signalverarbeitung, für die Entwicklung von Festkomma-Algorithmen usw.

In Simulink wird eine als mathematische Gleichung vorliegende Differentialgleichung als Integralgleichung im Laplace-Bereich u. a. mit Integratorblöcken dargestellt. Außerdem muss zur Ausführ-

barkeit ein Algorithmus für die numerische Integration ausgewählt werden. Bild 37 zeigt zwei Darstellungsmöglichkeiten[35] der Differentialgleichung

$\frac{dx}{dt} = -2x(t) + u(t)$ mit $u(t) = sin(t)$ und $t \in [0, \infty]$ [TaWi04].

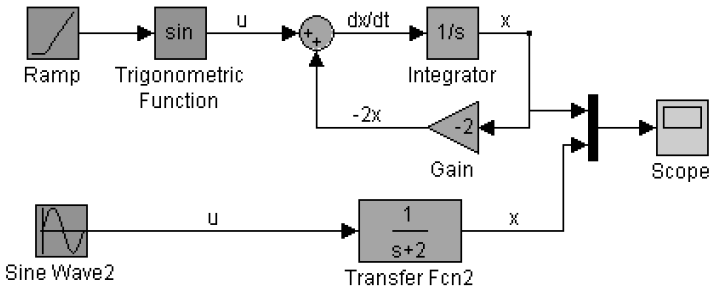


Bild 37: Darstellungsformen einer linearen Differentialgleichung in Simulink

Simulationsablauf in Simulink: Die Simulation besteht aus den beiden Etappen Initialisierung und Simulation. Bei der Initialisierung werden die Parameter der Blöcke initialisiert und danach in eine Reihenfolge sortiert, die die Aktualisierung vom Eingangsvektor *u*, dem Ausgangsvektor *y* und dem Zustandsvektor *x* ermöglicht. In dieser Phase werden auch die so genannten algebraischen Schleifen entdeckt. Es handelt sich hier um Schleifen über Blöcke, die keine Dynamik simulieren, wodurch der laufende Ausgang von sich selbst abhängig wird. Weiter wird geprüft, ob die Größe des Ausgangsvektors eines Blocks gleich der Größe der Eingangsvektoren aller Blöcke ist, die an ihn angeschlossen sind. In der Simulationsphase wird ein numerisches Integrationsverfahren eingesetzt, um die Vektoren der Blöcke zu aktualisieren. Zuerst wird der Ausgangsvektor berechnet und danach wird die Ableitung des Zustandsvektors mit Hilfe des laufenden Zustandsvektors, des laufenden Eingangsvektors und der laufenden Zeit ermittelt. Mit dieser Ableitung berechnet der Gleichungslöser entsprechend dem gewählten numerischen Verfahren den neuen Zustandsvektor für das nächste Zeitintervall. Erst jetzt werden die Zustände der diskreten Blöcke und die Darstellungen (z. B. für die Scope-Blöcke) aktualisiert [Hoff99].

Das CASE-Werkzeug Stateflow (The MathWorks) ist eine graphische Simulationsumgebung zur Modellierung von Zustandsautomaten für den Entwurf ereignisgesteuerter Systeme. Als Zusatz zu Simulink bietet es eine Lösung zur Entwicklung von Steuer- und Protokoll-Logiken. Stateflow wird zur Entwicklung von Statecharts (Stateflowcharts) verwendet, beginnend an dem Punkt, an dem das Modell oder ein Teil davon auf HW abgebildet und dann in die Simulation eingebunden werden soll. Stateflow ist eine graphische Erweiterung für Simulink und ermöglicht die Modellierung und Simulation von Zustandsautomaten wie es z. B. auch BetterState tut [StDr99]. Im Gegensatz zu Simulink-Systemen ist Stateflow ereignisgesteuert, d. h. ein Chart wird nur beim Eintreten vorher definierter Ereignisse abgearbeitet. Zustandsübergänge werden durch Ereignisse und Bedingungen spezifiziert. Innerhalb eines Simulink-Modells wird ein Stateflow-Diagramm durch einen Stateflow-Block repräsentiert. Ein Modell kann mehrere Stateflow-Blöcke enthalten, die über ihre Schnittstellen kommunizieren können. Eine Stateflow-Machine ist als die Menge aller Stateflow-Blöcke in einem Modell definiert. Während der Simulation können Signale mit Simulink ausgetauscht werden. Das Stateflow-Konzept erweitert Harel-Statecharts u. a. durch Junctions (diese ermöglichen die Darstellung von logischen Strukturen wie `for`-Schleifen), Gültigkeitsbereiche für Ereignisse und Daten, gerichteter Ereignis-Versand (erlaubt ein Ereignis explizit an einen bestimmten Zustand zu senden), implizite Ereignisse und bedingungshaftete Aktionen [StDr99].

In Zuständen und an Zustandsübergängen können Aktionen ausgeführt werden. Diese werden in Zuständen beim Eintreten, während des Aufenthalts oder beim Verlassen ausgeführt und wie folgt no-

[35] Eine andere Darstellungsform ist die in dieser Arbeit angewandte Repräsentation des Modells unter Verwendung von C-Code, wie er beispielsweise mit RTW erzeugt werden kann. Sobald das Modell in geeigneter Form vorliegt, kann es unter Verwendung des entsprechenden Simulators simuliert werden.

tiert: `entry:`, `during:`, `exit:`. Bei Übergängen wird eine Aktion als Beschriftung in geschweiften Klammern notiert. Allgemein besteht eine Beschriftung aus `Ereignis[Bedingung]{bedingte Aktion}/Übergangs-Aktion.` Bei der Simulation empfängt das Stateflowchart Daten und Ereignisse aus seiner Simulink-Umgebung und kann selber Daten und Ereignisse an diese senden (Bild 38).

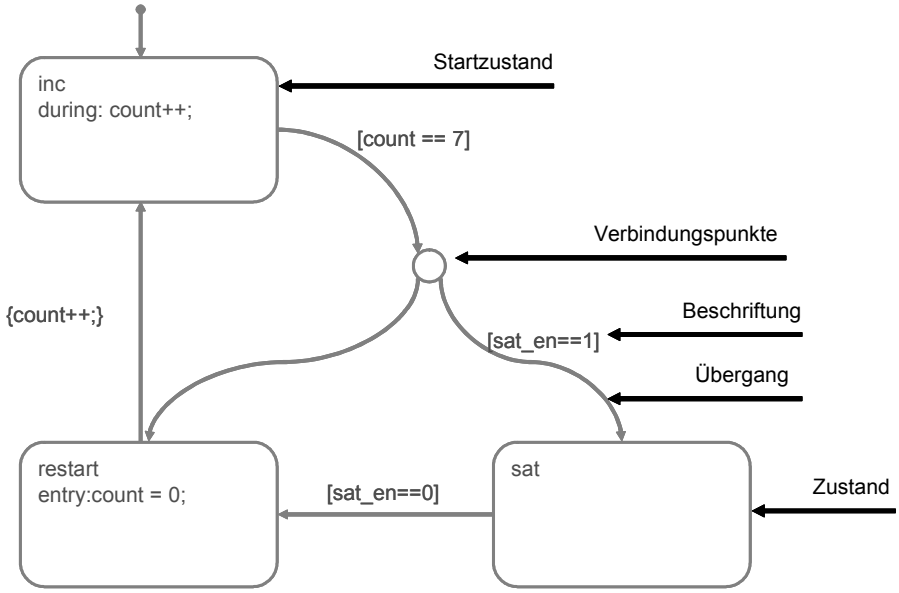


Bild 38: MATLAB-Stateflow

Zeitverhalten bei Simulink: Anhand folgenden Bildes (Bild 39) wird das Zeitverhalten veranschaulicht.

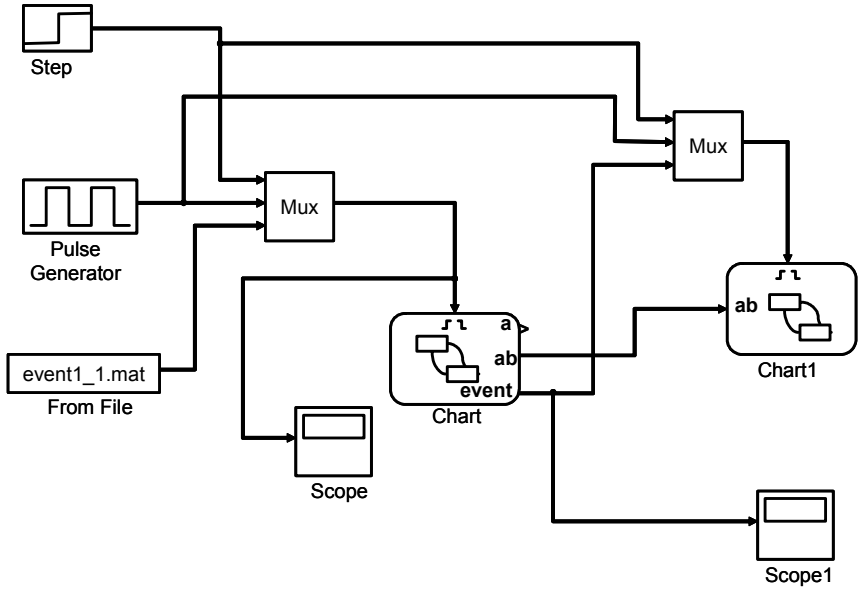


Bild 39: MATLAB mit mehreren Charts

- Simulink betrachtet jeden Block als System mit Eingängen und Ausgängen.
- Die Entscheidung, welche Blöcke zuerst ausgeführt werden, hängt von ihren Eingängen ab. Sobald Daten an allen Eingängen eines Blockes vorhanden sind, wird er ausgeführt.

Bei der Partitionierung eines Modells müssen die Eingangsprioritäten beachtet werden, so dass sich die Teilmodelle genauso verhalten wie das Gesamtmodell. Anhand eines Beispiels wird gezeigt, wie die Eingangsprioritäten berücksichtigt werden. Es wird angenommen, dass ein Eingangsereignis (ein

Puls) aufgetreten ist und die Zustände `A`, `B`, `C`, `A2` und `B1` aktiv sind. Das Modell verhält sich folgendermaßen:

- Es wird zuerst `B` ausgeführt (höhere Priorität als `C`).
- Die Transition „`/d=a2;event1;`“ wird nun ausgeführt (Transitionsbedingung erfüllt). `d` wird auf `a2` gesetzt, und dann wird das Ereignis `event1` erzeugt und ausgeführt. Es muss beachtet werden, dass das vorherige Ereignis `Puls` noch nicht komplett ausgeführt wurde. Es wird nun die Ausführung von `Puls` unterbrochen, so dass das neue Ereignis `event1` ausgeführt werden kann.
- Bevor `event1` ausgeführt wird, wird der Zustand `A1` inaktiv, so dass es in dem parallelen Zustand `B` keinen aktiven Zustand gibt.
- Da es in dem Zustand `B` keinen aktiven Zustand gibt, wird `C` und darin die Transition „`event1/f=b1;`“ ausgeführt. Danach wird `B2` aktiviert.
- Nun ist die Ausführung des Ereignisses `event1` beendet. Die Ausführung des vorherigen Ereignisses läuft weiter, d. h. die Zustand `A2` wird aktiviert und das Ereignis `Puls` wird im Zustand `C` weiter ausgeführt.

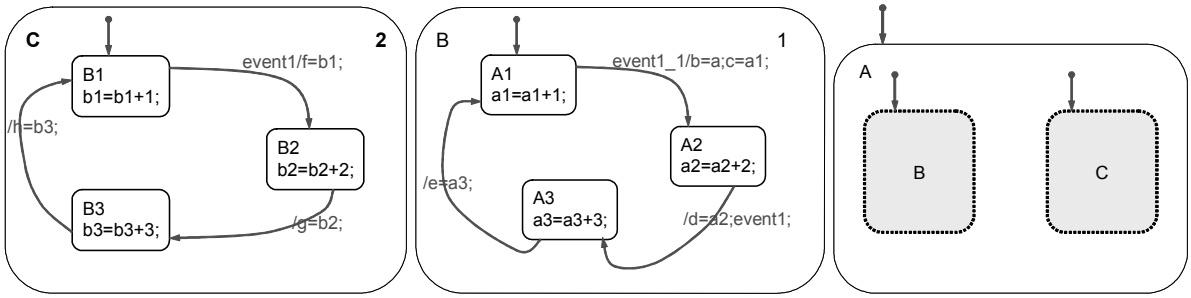


Bild 40: Testmodell zur Beschreibung der Eingangsprioritäten

Bearbeitung eines Ereignisses: Wenn ein Ereignis aufgetreten ist, wird es von oben nach unten entlang der Zustandshierarchie bearbeitet. Auf jeder Ebene der Hierarchie werden die `during`- und `on event_name`-Aktionen der aktiven Zustände ausgeführt. Dann wird die Gültigkeit der Transitionen zwischen Unterzuständen des jeweiligen aktiven Zustands geprüft.

Ausführung einer Transition: Die Gültigkeit der Transitionen aus derselben Quelle wird nach den folgenden Kriterien geprüft, die nach einer Prioritätsfolge vorgegeben sind:

- Hierarchieebene des Endpunkts: Transitionen, die zu einer höheren Ebene der Hierarchie führen, werden zuerst geprüft.
- Label: Transitionen, deren Endpunkte die gleiche Hierarchieebene haben, werden nach dem Label geprüft. Labels mit sowohl Ereignissen als auch Bedingungen haben die höchste Priorität. Es folgen Labels mit Ereignissen, danach Labels mit Bedingungen. Transitionen ohne einen Label werden am Ende geprüft.
- Position des Startpunkts: Wenn die Ausführung einer Transition nach den beiden oben angegebenen Kriterien noch nicht entschieden werden kann, müssen die Transitionen nach den Positionen ihrer Startpunkte im Quellobjekt im Uhrzeigersinn geprüft werden. Die Prüfung beginnt rechts oben im Quellobjekt.

Ausführung von Zuständen: Die `entry`-Aktion eines Zustands wird ausgeführt, wenn der Zustand aktiv wird. Die Eintrittsfolge von nebenläufigen Zuständen wird nach ihren geometrischen Positionen im Chart von oben nach unten, von rechts nach links geordnet. Diese Folge bestimmt also die Priorität der Ausführung von nebenläufigen Zuständen. Die Eintrittsfolge der nebenläufigen Zustände in Bild 41 ist `A`, `B`, `D`, `C`.

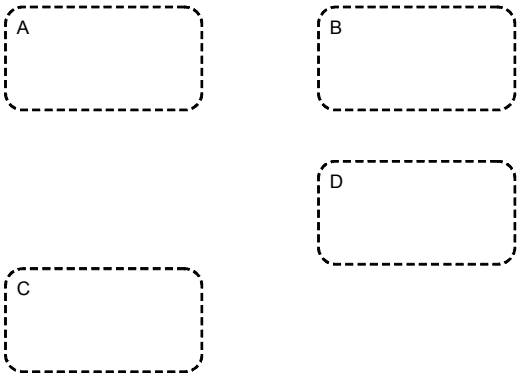


Bild 41: Eintrittsfolge von nebenläufigen Zuständen

Wenn ein aktiver Zustand ein Ereignis empfängt, und kein Zustandsübergang ausgelöst wird, werden die `during`- und `on event_name`-Aktionen ausgeführt. Die Ausführungsfolge von nebenläufigen Zuständen ist wie ihre Eintrittsfolge. Führt ein Ereignis zum Zustandsübergang, wird die `exit`-Aktion ausgeführt, und der Zustand wird inaktiv. Nebenläufige Zustände werden in umgekehrter Reihenfolge verlassen.

3.3.1.2 STATEMATE, VisSim und MATRIXx

Das kommerzielle CASE-Werkzeug STATEMATE ist ein Spezifikationswerkzeug für reaktive Systeme, das sich insbesondere in den frühen Phasen des Entwurfsprozesses für die graphische Beschreibung von Verhalten, Funktion und Struktur elektronischer Systeme eignet. Die entstandenen Spezifikationen lassen sich durch einen eingebauten Simulator ausführen, wodurch eine frühzeitige Verifikation und Validierung ermöglicht wird. Durch die Kombination von STATEMATE mit anderen kommerziellen Werkzeugen wie dem Programmpaket MATRIXx, das u. a. zur Modellierung von analogen Systemen und regelungstechnischen Algorithmen dient, lassen sich beispielsweise gemischt analoge und digitale Schaltungen gemeinsam mit ihrer meist analogen Umwelt zu einem System zusammenfassen und als solches simulieren. Angewendet werden solche Mixed-Mode-Simulationen z. B. in der Automobiltechnik.

Reaktive Systeme könen sowohl in HW als auch in SW realisiert werden und sind durch komplexe Reaktionen auf diskrete Ereignisse der Systemumgebung charakterisiert [Bort94]. Durch die graphische Darstellung können innerhalb kurzer Zeit anschauliche und eindeutige Spezifikationen erstellt werden, die sich gut als Dokumentation eignen und eine einfache Fehlersuche ermöglichen [McCl89] [ScNe90]. Aus diesen Gründen wird das Werkzeug in vielfältigen Industriebereichen wie Luft- und Raumfahrttechnik, Automobilindustrie [Weis96] und Telekommunikation häufig in Kombination mit anderen Werkzeugen zur Systemspezifikation eingesetzt. Bei einer rein gedanklichen Durchdringung komplexer Systeme bleiben mögliche unerwünschte Betriebszustände oder Fehlerfälle in der Entwurfsphase leicht unberücksichtigt. Je später aber solche Versäumnisse und Fehler entdeckt werden, desto zeitaufwendiger und vor allem kostspieliger ist ihre Beseitigung. Deshalb und im Hinblick auf die Qualität des Endprodukts ist es für den Entwickler äußerst erstrebenswert, möglichst viele Fehler schon in frühen Stadien des Entwurfsprozesses zu erkennen und zu beheben [KuZO95]. Die entstandenen Modelle können mit Hilfe des eingebauten Simulators ausgeführt und so auf ihre Korrektheit überprüft werden. Fertiggestellte Modelle lassen sich in den entsprechenden C- oder Ada-Code umsetzen. Die Spezifikationserfassung erfolgt mit Hilfe der graphischen Beschreibungssprachen Statechart, Activitychart und Modulechart [STAi95] (siehe 2.2.5).

Steuerung und Kommunikation erfolgen in STATEMATE über verschiedene Arten von Informationselementen und Aktionen. So lassen sich Informationen zwischen Activities austauschen, Prozesse aktivieren und deaktivieren und zeitliche Zusammenhänge beschreiben [Bort94] [STAi95]. Ereignisse haben keinen Wert und dienen als Steuersignale. Sie ähneln einem Impuls und sind verloren, wenn sie nicht sofort erfasst werden. Bedingungen sind kontinuierliche Steuersignale und können die

Werte `true` und `false` annehmen. Mögliche Aktionen sind das Auslösen von Ereignissen, das Setzen von Bedingungen, das Modifizieren von Variablen und das Starten oder Anhalten einer Activity. Aktionen können sowohl unmittelbar als auch um eine bestimmte Zeitspanne verzögert ausgeführt werden.

Ein Beispiel für die Integration von DEVS und DESS (siehe 2.2.4) bei CASE-Werkzeugen ist STATEMATE mit Visual Solutions VisSim, einer graphischen Sprache für die nichtlineare, dynamische Simulation, die auf Blockdiagrammen basiert [ScMu97]. Das Funktionsprinzip der Kommunikation zwischen den beiden Werkzeugen ist in Bild 42 und Bild 43 dargestellt.

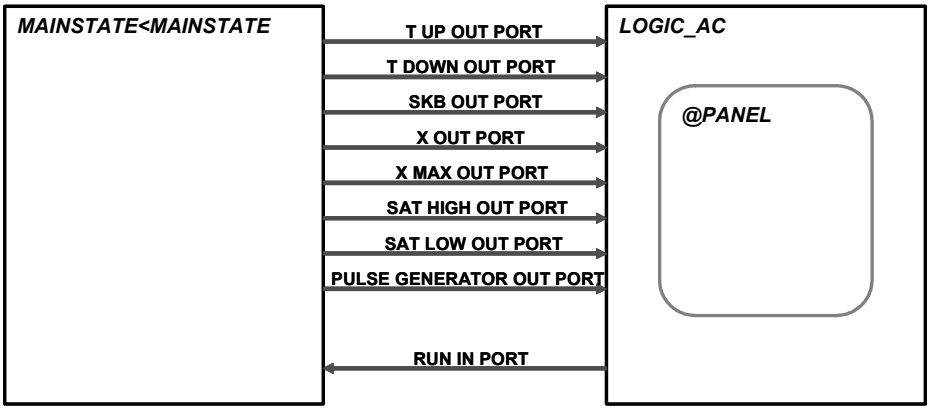


Bild 42: Kommunikation zwischen STATEMATE und VisSim (in STATEMATE)

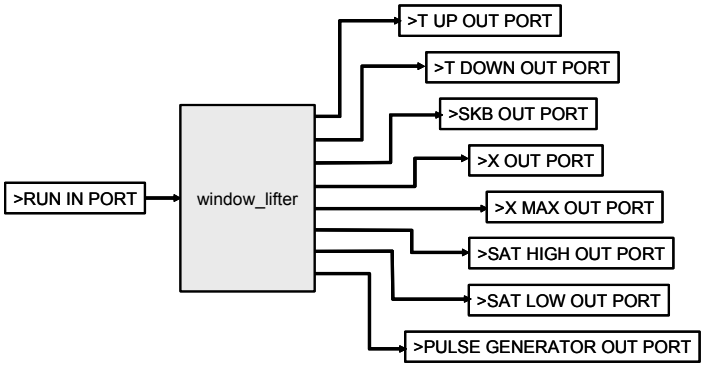


Bild 43: Kommunikation zwischen STATEMATE und VisSim (in VisSim)

Die Kommunikation erfolgt durch Variablen (in STATEMATE: `data items`) sowie speziell durch Boole'schen Variablen (in STATEMATE: `conditions`). Eine Übergabe von Ereignissen von STATEMATE nach VisSim und von Dirac-Impulsen von VisSim nach STATEMATE ist nicht möglich.

3.3.1.3 ASCET

ASCET ist eine Entwicklungsumgebung für eingebettete Steuerungs- und Regelungssysteme im Automobilsektor der Firma ETAS GmbH & Co.KG [ASCE00]. ASCET unterstützt einfache Zustandsautomaten, die die Modellierung komplexerer Systeme nur bedingt unterstützen. Im Bereich der Entwurfsmethoden fehlen folgende Erweiterungen von Statecharts bei der diskreten Modellierung: Hierarchie, Nebenläufigkeit und Zeitbedingungen.

3.3.1.4 Einsatz der Unified Modeling Language

Die Sprache UML (Unified Modeling Language) dient zum einen der Veranschaulichung und dem Dokumentieren von Ideen, Modellen, Vorgehensweisen und Lösungen, zum anderen der Spezifikation und dem Entwurf von Systemen. UML ist unabhängig von der Zielsprache und von der Zielplattform, unterstützt aber den objektorientierten (OO) Entwurf von Systemen. Daher lassen sich vie-

le Diagramme leicht z. B. in JAVA übertragen, so dass aus einer UML-Spezifikation schnell ein Prototyp zum Experimentieren entstehen kann [DuIn01].

Das für die Definition und Weiterentwicklung der UML verantwortliche Gremium ist die Object Management Group (OMG)[36]. Ideen aus anderen etablierten Modellierungssprachen wie Specification and Description Language (SDL) und Message Sequence Charts (MSC) wurden bei der Weiterentwicklung von UML berücksichtigt. Die Version 1.x von UML erlaubt eine Beschreibung des Verhaltens nur in Form von Quellcode, der als Fragment an geeigneter Stelle im Modell platziert und bei der Code-Generierung später übernommen wird. Das Modell selbst ist nicht ausführbar; es werden zusätzlich eine Programmiersprache und ein Compiler benötigt, um letztendlich ein ausführbares Programm zu erhalten. UML 2.0 ist eine ausführbare Sprache. Ein System kann beschrieben werden, ohne auf Code-Fragmente einer Programmiersprache zurückgreifen zu müssen. Das UML-Modell kann direkt in Objekt-Code übersetzt werden [BjKo02].

Bei den Verhaltensdiagrammen fehlen z. B. Beschreibungsmöglichkeiten für eine maximale Ausführungsdauer und für Scheduling-Mechanismen. Weitere Anforderungen, die nur umständlich beschrieben werden können, sind z. B. die CAN-Kommunikation und die Mensch-Maschine-Interaktion (MMI). Es fehlen Mechanismen zur Kopplung mit der Graphikdarstellung sowie zur Weiterleitung von Zustandsänderungen der Geräte zum Menü-System. Für eingebettete Echtzeitsysteme und für graphische Benutzerschnittstellen im Fahrzeug sind Anpassungen und Erweiterungen der Standard-UML erforderlich. Modellierung von Echtzeitsystemen unter Verwendung von UML ist nicht vollständig ohne UML-Erweiterungen für die Darstellung der Echtzeitsystem-Anforderungen aus der Sicht des Gesamtsystems, der HW- und SW-Schnittstellen sowie der Aspekte im Zusammenhang mit Multitasking und Nebenläufigkeit [Rao01].

Neben den Modellierungssprachen, die graphische Sichten auf das zu entwickelnde System darstellen, definiert der UML-Standard ein Datenmodell, in dem alle Entwurfsinformationen und ihre Abhängigkeiten abgelegt werden können. In diesem sogenannten UML-Metamodell wird festgelegt, wie die UML-Elemente (Klasse, Aktor, Use-Case[37]) und die Beziehungen zwischen diesen (Vererbung, Assoziation, Benutzung) gespeichert werden. Das Konzept des UML-Metamodells bietet die Möglichkeit, graphische Sichten in Form von Modellierungssprachen hinzuzufügen, die ihre Informationen ebenfalls im Format des UML-Metamodells ablegen. Daher kann das UML-Metamodell als einheitliches Austauschformat zwischen unterschiedlichen Beschreibungssprachen eingesetzt werden. Durch das auf dem UML-Metamodell basierende Extensible Markup Language (XML)-Dateiformat XML Metadata Interchange (XMI) können auch Modelle zwischen den Werkzeugen unterschiedlicher Hersteller ausgetauscht werden. XML ist eine Metasprache zur Beschreibung und Instanziierung von Auszeichnungssprachen für allgemeine Dokumente [DuIn01]. Die Entwicklung elektronischer Systeme im Automobil erfordert eine durchgängige Prozesskette, UML bietet zwar einen internationalen Standard für Modellierungssprachen, ist für die in der Fahrzeugentwicklung tätigen Funktionsexperten jedoch zu softwarelastig [Hofm00] [Balz96] [FoSc97] [HaGe97]. Noch fehlt die Verbindung zwischen der Beschreibung der Anforderungen und der Modellierung derselben in UML 2.0. Dies kann durch die unterstützenden Werkzeuge wie z. B. DOORS [Tele03] geleistet werden.

3.3.2 Verteilte Steuergeräte-Entwicklung

Die Industriepartnerschaft einiger großer Automobilhersteller und Systemlieferanten mit dem Namen AutoSAR (Automotive Open Systems Architecture) hat das Ziel, ein standardisiertes Elektrik-/Elek-

[36] Systemanbieter und Anwender objektorientierter Techniken haben sich 1989 zur OMG zusammengeschlossen. Die OMG verfolgt das Ziel, Standards und Spezifikationen für verteilte objektorientierte Anwendungen zu schaffen.

[37] Use-Cases erfassen die wichtigsten Funktionspunkte einer Anwendung. Sie stellen spezifische Anwendungen des geplanten Systems dar und sind für Programmierer ein Weg, spezifische Funktionsaspekte zu kommunizieren. Das Use-Case-Diagramm lässt sich von Technikern und Nichttechnikern gleichermaßen leicht verstehen. Es bildet daher die zentrale „Landkarte“ der Funktionsanforderungen an ein System [Leim01].

tronik-Architektur-Konzept zu entwickeln. Im Einzelnen befasst sich die Initiative mit der Modularisierung und Konfiguration von SW-Komponenten, mit der Vereinheitlichung von Schnittstellen und mit der Entwicklung einer Laufzeitumgebung. In Abgrenzung zur Hersteller Initiative Software (HIS), die eher kurzfristige Standardisierungen existierender Systeme ins Auge fasst, eröffnet AutoSAR eine längerfristige Perspektive (länger als drei Jahre) [ElAu03].

Ein funktionaler Zugang erlaubt das Design verteilter Systeme und die Integration von eingebetteter SW auf der ECU. Der modulare Entwurf von SW-Komponenten mit definierten Signal-Schnittstellen vereinfacht die Wiederverwendung von getesteten Funktionen in verschiedenen Netzen. Wie die im Rahmen der vorliegenden Arbeit entwickelten Werkzeuge, können auch die im Folgenden erläuterten Werkzeuge Titus©, DaVinci© und INTECRIO© im Bereich verteilter Steuergeräte-Entwicklung eingesetzt werden. Jedoch liegt der Schwerpunkt dieser drei Werkzeuge weder bei der Durchführung einer Gesamtsystem-Simulation, noch ist ihr Ziel, eine Simulationsbeschleunigung zu erreichen.

- Ziel der Titus-Entwurfsmethode ist es, den Entwicklungsprozess von verteilter Steuergeräte-SW durch Trennung von Funktions- und HW-Design zu optimieren. Durch die Wiederverwendung von SW-Bestandteilen und automatisierte Generierung ist es möglich, auf die immer kürzeren Entwicklungszeiten zu reagieren [Wern01]. Die Titus-Tools Suite ermöglicht eine Wiederverwendung von SW in unterschiedlichen HW-Topologien [WeFr01].
- Das Werkzeug DaVinci dient als Integrationsplattform für die Entwicklung vernetzter Steuergeräte und bietet, mit Blick auf das Gesamtsystem, auch die Möglichkeit, verteilte Fahrzeugfunktionen zu spezifizieren und die Kommunikation zwischen den Steuergeräten zu definieren. Es sind Schnittstellen zu MATLAB/Simulink/Stateflow vorhanden, so dass z. B. eine Code-Generierung mittels TargetLink© (siehe 3.3.3.2) möglich ist. In DaVinci enthalten sind Editoren, eine Werkzeug-Architektur mit verschiedenen Schnittstellen (Simulink, DBS, OIL), ein Code-Integrations-Framework sowie eine Schnittstelle zu externen Konfigurations-Management-Werkzeugen. DaVinci unterstützt den graphischen Entwurf von verteilter Applikationsstrukturen, die Spezifikation von Signal-Schnittstellen für den Datenaustausch zwischen SW-Komponenten sowie die modellbasierte Implementierung von SW-Komponenten. Eine Spezifikation der HW-Topologie mit ECUs, Sensoren, Aktoren und CAN-Bussen lässt sich damit vornehmen [Vect05].
- INTECRIO enthält keine eigenen Entwicklungswerkzeuge, sondern ist eine Integrationsplattform und Experimentierumgebung zum Rapid-Prototyping elektronischer Systeme und Komponenten, basierend auf einer Kombination aus ASCET-Modellen, MATLAB/Simulink-Modellen und C-Code-Modulen [ETAS05]. INTECRIO ermöglicht das Rapid Prototyping von kombinierten MATLAB/Simulink- und ASCET-Modellen mit der „ES1000“ und „ETK“ Rapid Prototyping HW von ETAS. In den Code-Generator von ASCET sind optimierte, in C- oder Assembler-Code geschriebene, targetspezifische Serviceroutinen integriert [ETHa05].

3.3.3 Automatische Code-Generierung von C-Code

3.3.3.1 Code für Rapid-Prototyping-Systeme

Unter Prototyping versteht man die Entwicklung eines Modells, das bereits wichtige Eigenschaften des künftigen Systems besitzt. Am Prototyp werden Probleme studiert, die in der jeweiligen Arbeitsumgebung entstehen werden, wie Schnittstellen, Funktionsumfang, Sicherheits- und Geschwindigkeitsanforderungen usw. Das „normale“ Prototyping modifiziert nun die Anforderungen und baut den Prototypen systematisch zum gewünschten System aus. Beim „schnellen“ Prototyping (Rapid Prototyping, RP) dient der Prototyp ausschließlich dazu, möglichst schnell einen vollständigen Überblick über alle Wünsche und Probleme zu erhalten, die mit dem angestrebten Einsatz des zu erstellenden Systems verbunden sind [DuIn01]. Der RP-Code ist im Vergleich zu automatisch generiertem Seriencode (Produktionscode) für möglichst viele Freiheiten beim Testen ausgelegt und somit nicht

für die bei Seriensteuergeräten limitierten Ressourcen wie Speicher oder Ein- und Ausgänge optimiert.

Ein Beispiel für einen Code-Generator für ein RP-System ist Real-Time Workshop© (RTW) von The Mathworks. RTW ist ein in Simulink integrierbarer Code-Generator, mit dem optimierter, portabler und auf die Anwenderanforderungen anpassbarer ANSI-C-Code (American National Standards Institute) aus Simulink-Modellen erstellt werden kann, um selbständig ausführbare Versionen des Modells zu erhalten, die auf unterschiedlichen HW- und SW-Plattformen ausgeführt werden können (PC mit Windows© oder Linux, DSPs, µCs, usw.). Zu Beginn der Code-Generierung wird das als mdl-Datei vorliegende Modell analysiert und eine temporäre RTW-Datei erzeugt, die das Modell in hierarchischer Strukturbeschreibung enthält (Bild 44).

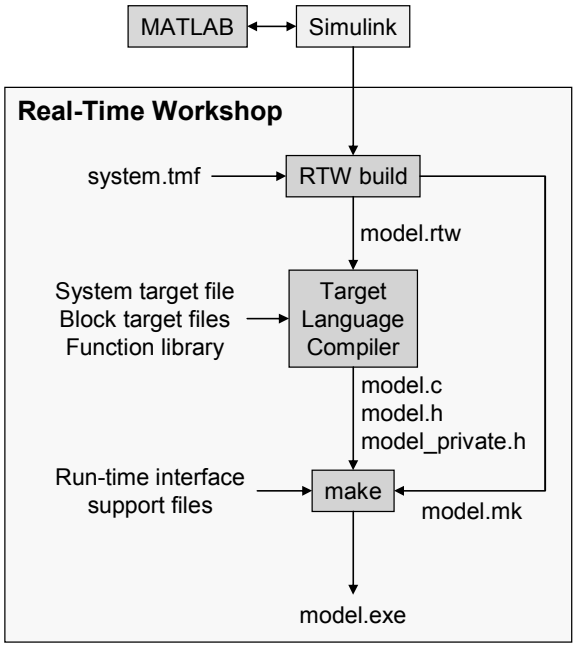


Bild 44: Ablauf der RTW-Code-Generierung

Der Target Language Compiler (TLC) interpretiert ein Skript, das TLC-Programm, das für die Code-Generierung zuständig ist und aus der rtw-Datei den C-Code erstellt [TLCR02]. Unter Verwendung einer ebenfalls generierten Make-Datei und einigen modellunabhängigen C-Dateien, die die Laufzeitumgebung für das Modell zur Verfügung stellen, wird anschließend der Code compiliert und zu einer binären Datei zusammengelinkt, die auf der entsprechenden Plattform ausgeführt werden kann [RTWU02]. Der RTW ergänzt das Simulationswerkzeug Simulink, um aus Simulationsmodellen echtzeitfähigen ANSI-C-Quellcode zu erzeugen [AuEl99]. RTW und Stateflow Coder sind Werkzeuge zur Generierung von C-Code aus Simulink-Modellen und Stateflow-Diagrammen, der dann zum Rapid Prototyping, für HIL-Simulationen und in eingebetteten Systemen eingesetzt wird.

3.3.3.2 Produktionscode

dSPACE TargetLink (TL) wird im Rahmen der Untersuchungen dazu verwendet, aus Simulink- und Stateflow-Modellen C-Code zu generieren [KöTS01] [TLdS01]. Aus Simulink-Blockschaltbildern oder Stateflow-Zustandsdiagrammen kann mit TL C-Code nach dem ISO/IEC (International Electrotechnical Commission) 9899-Standard erzeugt werden. Für die in dieser Arbeit verwendeten µC-Knoten werden mit dSPACE TargetLink automatisch generierter Produktionscode und ein OSEK RTOS eingesetzt (siehe 5.1).

TL integriert sich in MATLAB und ermöglicht eine Umsetzung des in Form von Simulink/Stateflow-Modellen vorliegenden SW-Designs in C-Code. Die Korrektheit dieser Umsetzung lässt sich

mittels Simulation überprüfen. Die Verifikation kann schrittweise erfolgen: zunächst durch eine Offline-Simulation auf dem Host-PC (SIL-Simulation), dann durch eine Echtzeit-Simulation auf dem Zielprozessor unter Einbindung des tatsächlich verwendeten Compilers (PIL-, Processor-in-the-loop-Simulation) [TaBD03]. Bild 45 zeigt die Integration von mit TL automatisch generiertem Code und OSEK [TLdS02] [TL2d02][38].

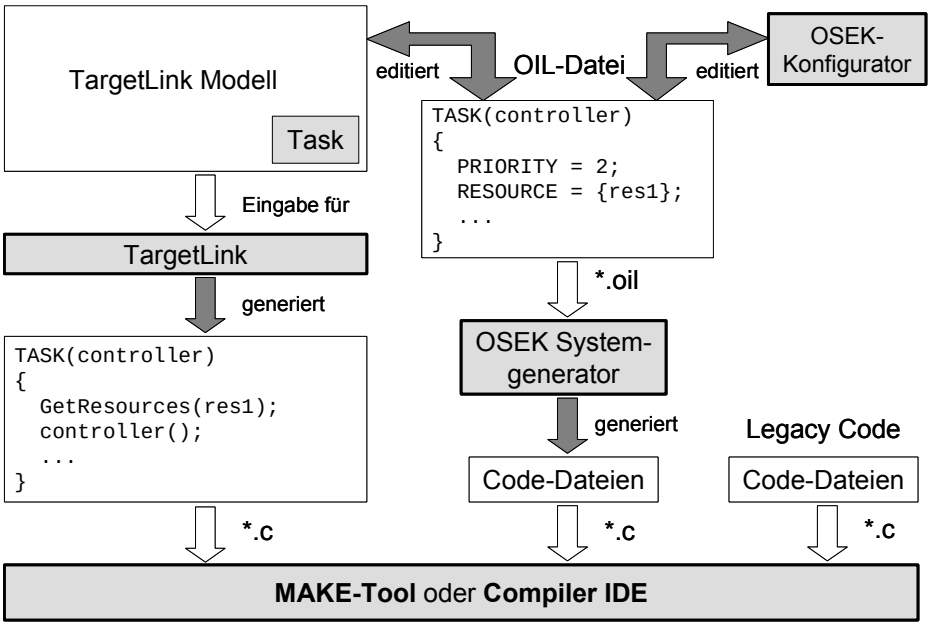


Bild 45: Integration von TargetLink-Code und OSEK/VDX

3.3.4 Hardware-Beschreibungssprachen

Die HW-Beschreibungssprache VHDL (Very high speed integrated circuit Hardware Description Language) wurde 1987 durch das Institute of Electrical and Electronics Engineers (IEEE) verabschiedet (Standard IEEE 1076-1987, VHDL'87) [VHDLor]. 1993 wurde eine überarbeitete Version (VHDL'93) vorgestellt, siehe [Perr91] [VRef93] und [IRef99]. Sie ermöglicht eine systemunabhängige, diskrete Modellierung; durch VHDL wird die Beschreibung von digitalen Schaltungen sowohl auf der Struktur- als auch auf der Verhaltensebene ermöglicht. Die wesentlichen Eigenschaften von VHDL sind:

- Anweisungen zur sequenziellen und nebenläufigen Bearbeitung von Zuweisungen, die nacheinander bzw. quasiparallel abgearbeitet werden;
- Trennung von Schnittstelle und Verhalten eines Modells durch Entity und Architecture;
- ereignisorientierte Simulation, diskrete Zeitachse;
- modulare und flexibel änderbare Modelle auf Registertransfer- oder Gatterebene.

VHDL ermöglicht als HW-Beschreibungssprache einen Top-Down-Entwurf eines Systems auf abstrakter Ebene, wobei der Einsatz von Synthese[39] als wesentliches Hilfsmittel dient, Komplexitätsprobleme handhabbar zu machen [LeWS94]. Aufgabe der Synthese ist es, ein in einer Hochsprache beschriebenes Modell einer digitalen Schaltung in eine untere Abstraktionsebene zu übersetzen. Ausgangspunkt der Register Transfer Level (RTL)-Synthese ist eine Modellbeschreibung auf Register-Transfer-Ebene, also eine aus kombinatorischen und sequenziellen Schaltungsblöcken bzw. Zustandsautomaten bestehende Beschreibung der Schaltung. Unter Berücksichtigung der Zielkriterien

[38] Unter Legacy Code ist handgeschriebener Code zu verstehen; OSEK Configurator ist ein Synonym für OSEK Builder.

[39] Synthese ist die rechnergestützte Transformation von einer höheren zu einer niedrigeren Abstraktionsebene, verbunden mit einem Wechsel der Entwurfssicht [BeHa01].

(constraints) erhält man nach einigen Optimierungsschritten eine symbolische Gatternetzliste [SiSi-03].

Da VHDL sich nicht zur Simulation analoger Systeme eignet, wurde VHDL-AMS (VHDL-Analog und Mixed-Signal) entwickelt. VHDL-AMS ist eine simulatorunabhängige, im Jahre 1999 standardisierte (IEEE 1076.1) Beschreibungssprache für sowohl gemischt analog-/digitale als auch Multidomain-Systeme [VHDLAM]. Wesentliches Ziel war es, ein einheitliches, standardisiertes Format für analoge Modelle bereitzustellen [ElA01b]. Ein Vorteil von VHDL-AMS gegenüber herstellerspezifischen HW-Beschreibungssprachen wie MAST[40] ist die Möglichkeit zur Erstellung simulatorunabhängiger Modelle für digitale, analoge und gemischte (mixed-signal) Systeme (siehe 2.1.3). VHDL-AMS Modelle haben eine hohe Portabilität und Wiederverwendbarkeit [ElA02b]. Der VHDL-Entwurfsfluss (Bild 46) wird in drei Bereiche eingeteilt: Designerstellung, Implementierung und Verifikation.

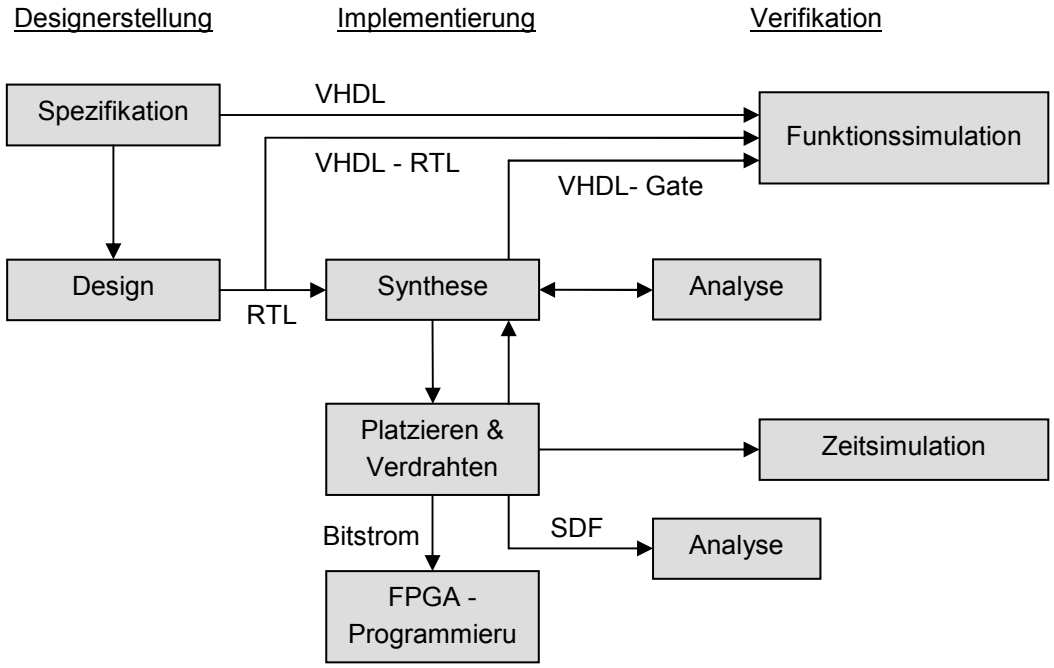


Bild 46: VHDL-Entwurfsfluss

Der Ausgangspunkt jedes Entwurfs ist seine Spezifikation. In VHDL könnte diese auf Abstraktionsebenen wie Algorithmen und Funktionsblöcken erfolgen oder von einer nicht formalen Beschreibung konvertiert werden. Bereits auf dieser Ebene sollte die Korrektheit des Entwurfs durch Logiksimulation getestet werden. Diese Beschreibung auf hoher Ebene wird in die Register-Transfer-Ebene und dem Synthesewerkzeug übergeben. Die von diesem Werkzeug unterstützte Sprache ist die erste der Abhängigkeiten, die in der Werkzeugkette auftritt. Das Synthesewerkzeug selbst konvertiert die VHDL-RTL-Beschreibung in eine im Wesentlichen technologieunabhängige Logikbeschreibung, z. B. indem spezielle Eigenschaften eines bestimmten FPGAs verwendet werden; auch der RTL-Code ist nicht target-abhängig. Diesem Schritt folgt Platzieren und Verdrahten, bei dem der Entwurf auf die Technologie eines bestimmten Herstellers abgebildet wird. An dieser Stelle ist eine präzise Taktinformation in SDF (Standard Delay Format) verfügbar, und es ist möglich, das Zeitverhalten zu simulieren, um zu verifizieren, ob alle Randbedingungen erfüllt werden. Wenn die Zielplattform ein FPGA ist, wird ein Bitstrom für seine Konfiguration generiert. Die Module werden auf dem FPGA platziert und miteinander verbunden.

[40] Mixed-Technology und Mixed-Signal HDL von der Firma Analogy für das Werkzeug Saber.

Alternativen zu VHDL sind Verilog und SystemC [SySC00]. 1995 wurde Verilog als Standard IEEE 1364 verabschiedet und 2001 aktualisiert, siehe [VCad94] [Golz96] und [Suth00], und Mitte 2000 wurde Version 1.0 von SystemC vorgestellt. Seit 2001 ist Version 2.0 verfügbar, die insbesondere Methoden der Modellierung der Systemebene unterstützt [Syst05]. SystemC ist eine Klassenbibliothek von C++, die es erlaubt, HW zu beschreiben. Zur Simulation der Programme ist ein ANSI-C++-Compiler notwendig.

3.3.4.1 Synthese endlicher Automaten

Im Allgemeinen kann ein endlicher Automat bei einer Umsetzung in Hardware in drei Teile unterteilt werden (Bild 47). Der Zustandsspeicher beinhaltet ein Register, das zur Speicherung des aktuellen Zustands benötigt wird. Der Zustandswechsel erfolgt synchron zu einem globalen Taktsignal. In Abhängigkeit vom aktuellen Zustand und den Eingangssignalen ermittelt die kombinatorische Zustandsübergangslogik den nächsten Zustand. Der letzte Block ist die Ausgangslogik, die aus dem aktuellen Zustand und, in Abhängigkeit von der Art des Automaten, die Werte der Ausgangssignale ermittelt.

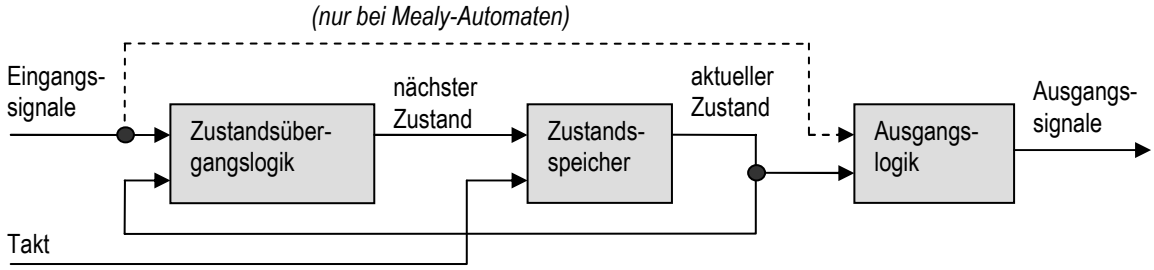


Bild 47: Synthetisierte Elemente eines endlichen Automaten

Wie ein endlicher Automat in VHDL beschrieben wird, soll exemplarisch in Bild 48 gezeigt werden.

```
architecture BEHAVIORAL of STATE_MACHINE is
…
begin
   Zustandsuebergangslogik: process (ACTUAL_STATE, IN_DATA)
   begin
    case ACTUAL_STATE is
      when q0 =>
        NEXT_STATE <= q1;
      …
   end process;

   Zustandsspeicher: process (CLK)
   begin
      if (CLK'event and CLK = '1') then
        ACTUAL_STATE <= NEXT_STATE;
      end if;
   end process;

   Ausgangslogik: process (ACTUAL_STATE, IN_DATA)
   begin
      case ACTUAL_STATE is
        when q0 =>
          OUT_DATA <= “00”;
        …
   end process;
end BEHAVIORAL;
```

Bild 48: Umsetzung eines endlichen Automaten in VHDL

Hierbei wird jeder Block aus Bild 47 in einem eigenen Prozess beschrieben. Im Gegensatz dazu können aber auch die Zustandsübergangs- und Ausgangslogik in einen Prozess zusammengefasst werden [Geiß00].

Für die Umsetzung der Zustände, dem so genannten state encoding, gibt es eine Vielzahl von Möglichkeiten. So kann jedem Zustand ein bestimmter Binärcode zugeordnet werden. Mit n Registern können dann 2^n Zustände kodiert werden. Bei der Gray-Kodierung wird sichergestellt, dass sich immer nur ein Bit bei einem Übergang ändert. Es wir die minimale Anzahl an Leitungen ($\log_2$ Zustände) gebraucht [AgHH99]. Eine einfache Kodierung ist die One-Hot-Codierung. Dabei werden so viele Bits (Register) verwendet, wie es Zustände gibt. Bei einem Zustandswechsel verändern sich immer 2 Bits. Nachteilig dabei ist die große Anzahl der benötigten Register. Allerdings wird keine kombinatorische Logik benötigt, um den aktuellen Zustand zu dekodieren. Daher ist diese Methode die schnellste Implementierung. Mit Hilfe von VHDL-Attributen kann der Compiler den Zustandsvektor identifizieren und entsprechend synthetisieren.

3.3.5 Code-Generatoren für Hardware-Beschreibungen

Das IDE von Mentor Graphics ist eine Oberfläche, von der aus FPGA-Entwicklungswerkzeuge von Xilinx und Altera[41] erreicht werden können. ModelSim von derselben Firma ermöglicht eine Logiksimulation des Entwurfs [Ment05]. Dabei können Design-Blöcke in unterschiedlichen Beschreibungssprachen wie SystemC, Verilog und VHDL vorliegen. AccelChip DSP Synthesis kann MATLAB- (mit Dateierweiterung .m) und Simulink-Modelle (mit Dateierweiterung .mdl) einlesen und synthetisierbare RTL-Modelle und RTL-Testumgebungen (Testbenches) zur Simulation in VHDL oder Verilog generieren [Acce05]. Die Xilinx-eigene FPGA-Entwicklungsumgebung heißt ISE (Integrated Software Environment) [Xili05], diejenige von Altera Quartus II [Alte05]. Bei Xilinx verknüpft der „System-Generator“ [XiSG03] ISE mit MATLAB/Simulink, bei Altera Quartus II übernimmt diese Funktion der „DSP-Builder“. Der Xilinx SystemGenerator ist ein SW-Werkzeug für den FPGA-basierten Entwurf von DSP-Systemen. Es ist als Blockset in Simulink eingebettet. Mit dem SystemGenerator beschriebene Systeme können in hardwarebeschreibende Codes (VHDL) und Netzlisten übersetzt werden, die nach einer Synthese auf einem FPGA konfiguriert werden können. Definierte Systemparameter in Simulink werden für eine spätere HW-Realisierung des SystemGenerators in Entities, Architectures, Ports, Signale und Attribute abgebildet. Darüber hinaus werden automatisch Dateien für die Synthese, Simulation und Implementierung generiert. Mit dem Altera DSP Builder Blockset können ebenfalls signalverarbeitende Designs unter Simulink entworfen werden, mit denen ein Altera FPGA konfiguriert werden kann. Der DSP Builder verfügt über ähnliche Funktionalitäten wie der Xilinx SystemGenerator. SystemGenerator und DSP Builder können in Simulink zusammen mit dem TargetLink Blockset (siehe 5.2.2.1) verwendet werden.

Für die Systementwicklung in FPGAs sind Standard-IP-Cores (Softcores, Standard-Intellectual Property Cores), u. a. Prozessoren, Kommunikationsfunktionen, optimierte DSP-Funktionen, Schnittstellen und Peripherieeinheiten, verfügbar. Der Nios-Embedded-Prozessor von Altera ist ein so genannter Softcore-Prozessor, der in Form einer parametrisierbaren HW-Beschreibung sowie einer automatisch angepassten SW-Entwicklungsumgebung existiert. Die gesamte Funktionalität ist wahlweise in VHDL oder Verilog beschrieben. Das hat den Vorteil einer hohen Flexibilität, da das System-On-Chip mit den exakt benötigten Eigenschaften entworfen und auch nachträglich verändert und erweitert werden kann. Die Excalibur-Nios-Familie beruht auf einem konfigurierbaren 16-Bit- oder 32-Bit-RISC-Prozessor, der als synthetisierbare HW-Beschreibung erzeugt wird. Der Befehlssatz des Prozessors lässt sich durch selbst definierte Instruktionen erweitern, die in der programmierbaren Logik ausgeführt werden [Zimm02]. Nios kann in FPGAs und ASICs implementiert werden. Je nach eingesetzter HW-Plattform lassen sich damit sowohl die Kosten für eine CPU als auch die Leistungsfähigkeit an die jeweiligen Anforderungen anpassen.

[41] Die FPGA-Typen Xilinx Virtex-II [XiVi01] und Altera Stratix [Zeid02] wurden für Erprobungen im Rahmen dieser Arbeit verwendet [DeE02c] (siehe 4.2).

Neben Werkzeugen von Xilinx, Altera, AccelChip und Mentor Graphics dient der VHDL-Code-Generator SF2VHD im Rahmen dieser Arbeit als Basis für die Entwicklung von JVHDLGen (siehe 4.3.4). SF2VHD wurde im Rahmen des SSHAFT-Projekts (Simulink to Silicon Hierarchical Automated Flow Tools) des Berkeley Wireless Research Centers der University of California in Berkeley entwickelt. Die Ergebnisse flossen in das Nachfolgeprojekt BEE (Berkeley Emulation Engine) [BEE03] ein, das durch Rapid Prototyping mit Hilfe mehrerer Xilinx-Virtex-E-FPGAs das Ziel der Simulationsbeschleunigung verfolgt [ChKR03].

Die wichtigsten Einschränkungen von SF2VHD sind das Nichtvorhandensein von [Came00] [Came-01]:

- AND-Dekomposition und History-Konnektor. Diese Einschränkung betrifft vor allem die Verwendung hierarchischer Statecharts.
- Transitionen über Hierarchieebenen.
- Priorisierung der Reihenfolge der Transitionen eines Zustandes (Sie werden in der Reihenfolge des Erscheinens in der MATLAB-Datei behandelt).
- Konnektoren an Starttransitionen.
- „exit“- oder „transition-taken“-Aktionen.
- arithmetischen Operationen auf Boole’schen Werten oder umgekehrt.
- in VHDL reservierten Wörtern außerhalb von Kommentaren.
- Ereignissen.

Die Einschränkungen des Programms beziehen sich einerseits auf Zeichenkettenlängen, die in der Stateflow-Syntax verwendet werden, andererseits auf die Anzahl der verwendeten Objekte (siehe Tabelle 3). So darf der Name von Stateflow-Daten nicht mehr als 256 Zeichen, der von Transitionen und Zuständen nicht mehr als 2048 Zeichen enthalten.

Beschreibung	Wert
Anzahl der Stateflow-Objekte in einer mdl-Datei	1000
Anzahl der Statecharts in einer mdl-Datei	100
Anzahl definierbarer Daten in einem Statechart	500

Tabelle 3: Einschränkungen von SF2VHD

SF2VHD bildet verschiedene Datentypen in Stateflow in unterschiedliche Datentypen in VHDL ab (Tabelle 4). Andere Datentypen wie z.B. `float` und `double` werden nicht unterstützt.

Stateflow-Typ	VHDL-Typ
boolean	std_logic
unit8	unsigned (7 downto 0)
unit16	unsigned (15 downto 0)
unit32	unsigned (31 downto 0)
int8	signed (7 downto 0)
int16	signed (15 downto 0)
int32	signed (31 downto 0)

Tabelle 4: Abbildung von Datentypen in SF2VHD

Stateflow-Operatoren werden wie in Tabelle 5 in VHDL-Operatoren umgesetzt. Die Division, die Array-Operation und der Funktionsaufruf werden nicht unterstützt.

Operatoren in Stateflow	Operatoren in VHDL	Beschreibung
+, -, *, /	+, -, *, /	Arithmetik
%%	mod	Modulo
<<, >>	sll, sra	Schieben
&, |, ^, ~	and, or, xor, not	Logisch (Bit)
&&, ||	and, or	Logisch (Boolean)
<, <=, >, >=	<, <=, >, >=	Vergleich
==	=	Gleichheit
~=, !=, <>	/=	Ungleicheit

Tabelle 5: Abbildung von Operatoren in SF2VHD

3.4 Partitionierung

Die Rechenzeit für eine Simulation ist die Summe aus der reinen Applikations-Simulationszeit, dem Zeitbedarf für Rücksetzungs und Wartezeiten sowie der Kommunikationszeit zwischen den LPs. Die Partitionierung eines Modells beeinflusst die Effizienz in erheblichem Maße. Je abhängiger die Teilmodelle voneinander sind, desto geringer ist der potentiell verfügbare Grad an Parallelität, der ausgenutzt werden kann. Die Hoffnung besteht jedoch darin, dass sich das Modell in vielen Fällen in möglichst lose gekoppelte Teile partitionieren lässt. Eine automatische Partitionierung ist z. B. denkbar bei der Simulation von VLSI-Schaltungen auf Gatterebene. Das Gesamtmodell lässt sich bei dieser Anwendungsklasse als Graph betrachten, dessen Knoten Gattern entsprechen und dessen Kantenmenge der Netzliste der zu simulierenden Schaltung entspricht. Eine Partitionierung könnte daraus resultieren, *n* knotendisjunkte, zusammenhängende Teilgraphen zu bestimmen, so dass benachbarte Teilgraphen möglichst wenige Kanten gemeinsam haben. Hierbei wird angenommen, dass zwei Teilschaltungen umso unabhängiger voneinander sind, je weniger Signalwege sie verbinden. Tatsächlich kann es in manchen Fällen auch sinnvoll sein, auf die strikte Knotendisjunktheit zu verzichten [MaLo93]. In diesen Fällen werden bestimmte Teile des Modells mehrfach (und damit redundant) ausgeführt. Weil sich die Unabhängigkeit der Teilmodelle dadurch erhöht, wird möglicherweise trotz des höheren Gesamtrechenaufwandes eine weitere Beschleunigung erreicht. Für viele Anwendungen muss jedoch eine Partitionierung manuell durch den Modellierer gefunden werden. Analog zur Modellierung [Neel87] stellt dieser Vorgang i. A. eher eine Kunst als eine Wissenschaft dar [Meh94b].

Als entscheidendes Partitionierungskriterium bei einer Performance-Optimierung kann eine über die Partitionen betrachtete gleichmäßige maximale Systemauslastung angesehen werden. D. h. es ist erstrebenswert, möglichst zu allen Zeitpunkten alle Teilsysteme ausgelastet zu haben, um so die Leerlaufzeiten zu minimieren. Hier sollte einerseits nach Abschätzung des Rechenaufwands für die Teilsysteme eine Partitionierung gewählt werden, die die Leistungsfähigkeit der jeweiligen HW mit einbezieht; andererseits muss in die Überlegung die Datenabhängigkeit der Teilsysteme einfließen, da es sonst zu Verzögerungen kommen kann, wenn ein Teilsystem auf die Eingaben eines anderen Teilsystems wartet. An dieser Stelle muss die Dimensionierung der Schnittstellen betrachtet werden: Nach einer quantitativen Analyse sollte die Anzahl auszutauschender Daten minimiert werden, da ein geringer Kommunikationsaufwand über die Zeit zu einer weiteren Performance-Steigerung führen kann. Ein derart optimiertes System kann allerdings störanfällig sein, da Redundanz zwecks Performance-Optimierung eliminiert wurde. Beim Einsatz eines FPGAs muss gerade bei der Verwendung von 32 Bit breiten Variablen beachtet werden, dass die Funktionseinheiten auf dem FPGA entsprechend groß synthetisiert werden, was dazu führen kann, dass der Speicher auf dem FPGA nicht ausreicht [300]. Für ein mit ausreichend Rechenleistung ausgestattetes System gilt:

$$L \leq \sum_{i=0}^{n} NP(i) \cdot PL(i)$$

mit

L: Leistungsbedarf des Systems

NP(i): Anzahl der Prozessoren des Typs *i*
PL: Prozessorleistung des jeweiligen Prozessortyps *i*

Die Ungleichheit nimmt mit wachsender Redundanz bzw. mit sinkender Systemauslastung zu. Auch bei der Auswahl der Prozessortypen gibt es unterschiedliche Einsatz-Szenarien: Zur Performance-Maximierung werden teurere, aber auch schnellere FPGAs eingesetzt. Im Gegensatz dazu reicht bei Systemen mit geringen Leistungsanforderungen und hohen Stückzahlen möglicherweise ein preiswerter 8-Bit-Mikrocontroller aus. Bei der Verteilung eines Systems auf mehrere Partitionen müssen auch die notwendigen Investitionen in die Kommunikation betrachtet werden. Die Partitionierung sollte mit möglichst wenigen Verbindungen zwischen den Teilsystemen auskommen. Eine grobe Gesamtkostenfunktion lässt sich folgendermaßen angeben:

$$K = \left[\sum_{i=0}^{n} NP(i) \cdot (PP(i) + PK(i)) \right] + PV$$

mit

K: Kosten
PP(i): Preis für Prozessor des Typs *i*
PK(i): Preis für die Anbindung des Prozessors an die Kommunikation
PV: Preis für die Verbindungen zwischen den Recheneinheiten

Die Wahl einer bestimmten Partitionierungslösung beeinflusst Systemeigenschaften wie Speicherbedarf, Ausführungsgeschwindigkeit, Buslast, Energieverbrauch, Zuverlässigkeit und Kosten. So kann z. B. durch eine Verteilung in Teilpartitionen mit unterschiedlicher Leistungsfähigkeit ein erhöhter Speicherbedarf entstehen, wenn die Ergebnisse einer schnellen Komponente zwischengespeichert werden müssen, bis sie von einer langsameren Einheit verarbeitet werden können. Die Systemperformance hängt neben der ausgewählten HW vom Grad der Datenabhängigkeit der Partitionen und bei hohem Datenaufkommen von der Leistungsfähigkeit des Kommunikationssystems ab. Eine hohe Datenabhängigkeit wirkt sich auch durch eine gesteigerte Buslast aus. Wird ein System z. B. von einer leistungsstarken Komponente mit hoher Stromaufnahme auf mehrere schwächere Einheiten mit deutlich geringerem Energiebedarf verteilt, kann der Verbrauch gesenkt werden.

Mit wachsender Komplexität und Verteilung integrierter Schaltungen und eingebetteter Systeme steigen auch die Anforderungen an ihre Entwicklung. Vor allem die mehrere Stunden bis Tage dauernde Verifikation großer Entwürfe erweist sich bei konventioneller Computer-basierter Simulation als Flaschenhals im Entwicklungsprozess. Ein möglicher Ansatz zur Simulationsbeschleunigung liegt darin, verschiedene Ereignisse auf mehreren Prozessoren zur selben Zeit zu bearbeiten. Eine Ein-Prozessor-Architektur vorausgesetzt, erlaubt dies eine echte Parallelität der Ausführung. In einem zweiten Schritt wird das Modell oder ein Teil des Modells auf rekonfigurierbarer HW ausgeführt. Bei dieser Methode muss zwar der einmalige Partitionierungsaufwand aufgebracht werden, allerdings birgt sie den Vorteil deutlich höherer Ausführungsgeschwindigkeit und echter Parallelität auf einem Chip.

3.4.1 Hardware/Software-Co-Design

Der Entwurf eines gemischten HW-SW-Systems wird auch als HW/SW-Co-Design bezeichnet. Durch die Parallelisierungsmöglichkeit von HW bietet er ein größeres Leistungspotential als SW, ist aber weniger flexibel hinsichtlich der Anpassung und Erweiterung der Funktionalitäten im Laufe des Produktlebenszyklus. Ein möglicher Ablauf eines Systementwurfs mittels HW/SW-Co-Design ist in Bild 49 dargestellt [Teic97]. Zentraler Bestandteil des HW/SW-Co-Designs ist die HW-/SW-Partitionierung. Hierbei sind verschiedene Vorgehensweisen möglich. Partitionierungskriterien können u. a. Kosten, Fläche, Verlustleistung und Performance eines Systems sein. Entwurfsbeschränkungen (Constraints) sind durch die resultierenden Kosten und die geforderte Leistungsfähigkeit eines Systems vorgegeben.

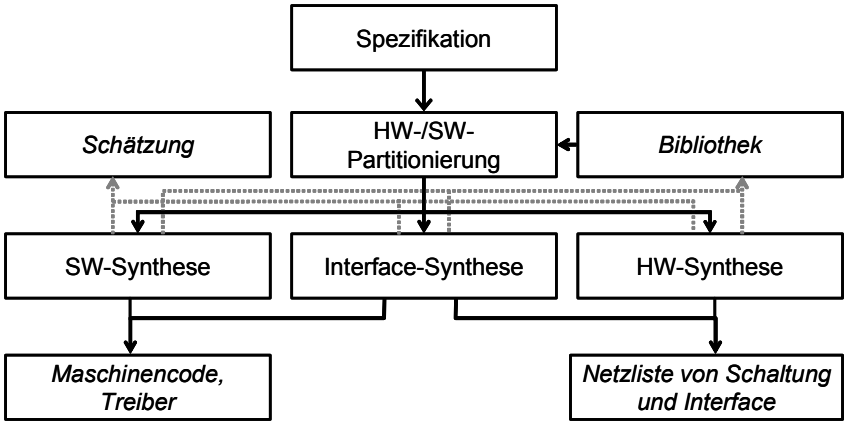


Bild 49: Systementwurf mittels HW/SW-Co-Design

3.4.2 Mapping und Scheduling

Stehen mindestens so viele Prozessoren zur Verfügung, wie Teilmodelle verwendet werden sollen, so ist es nahe liegend, je ein Teilmodell von einem anderen Prozessor ausführen zu lassen. Wenn mehr als zwei LP auf einer CPU laufen, ist der Einsatz eines Scheduling-Verfahrens (Zuteilung, Ablaufplanung) notwendig, bzw. wird dadurch erst ermöglicht. Wird der Kooperationsaufwand zwischen zwei Teilmodellen im Vergleich zum Rechenaufwand in diesen Teilmodellen zu hoch, kann es auch günstiger sein, beide Teilmodelle zusammenzufassen und auf dem gleichen Prozessor auszuführen. Für viele Simulationen ist jedoch die Anzahl verfügbarer Prozessoren wesentlich kleiner als die Anzahl der ein Teilmodell ausführenden LP. Bei paralleler Simulation ist kein explizites Mapping erforderlich. Es bleibt lediglich ein Scheduling-Problem für die Auswahl des als nächsten zu bedienenden LP. Bei verteilter Simulation sind prinzipiell Mapping- und Scheduling-Strategien erforderlich. Man unterscheidet jeweils statische und dynamische Verfahren.

Die Frage nach einem geeigneten Scheduling stellt sich dann, wenn mehrere LPs pro CPU zum Einsatz kommen. Zwei prinzipielle Fragen sind:

- Wann soll die Ausführung von einem LP zum nächsten wechseln?
- Welcher LP soll bei einem Wechsel der nächste sein?

Abhängig vom verwendeten Synchronisationsverfahren sind verschiedene Scheduling-Kriterien denkbar. Beispiele für Scheduling-Strategien sind:

- Verarbeitung aller noch nicht verarbeiteten Ereignisse vor einem Wechsel. Dadurch kann ein Overhead durch zu weite Progression der lokalen Simulationszeit[42] (Local Virtual Time, LVT) entstehen;
- Wechsel nach jedem Ereignis. Es entsteht ein Overhead durch Kontextwechsel;
- Wechsel nach einigen Ereignissen. Kriterien können z. B. sein: die CPU-Zeit oder eine große LVT-Differenz gegenüber anderen LPs;
- Priorität für wartende Tasks;
- die Ereignisdichte;
- Reihenfolge der Task-Aktivierungen und Task-Prioritäten: In OSEK kann jeder Task eine Priorität zugeordnet werden (dabei entspricht eine höhere Zahl einer höheren Priorität). Tasks mit gleicher Priorität werden nach dem FIFO-Prinzip verwaltet, so dass die „älteste“ Task mit der höchsten Priorität zur Ausführung kommt;

[42] Die lokale Zeit ist der Zeitpunkt des letzten vom Simulator abgearbeiteten Ereignisses.

- präemptive Zuteilung: Bei einer präemptiven Zuteilungsstrategie kann eine höherpriore Task die Ausführung einer niederprioren Task unterbrechen;
- nichtpräemptive Zuteilung: Bei einer nichtpräemptiven Zuteilungsstrategie kann der Wechsel von einer niederprioren zu einer höherprioren Task nur zu bestimmten Zeitpunkten erfolgen, z. B. nach Abschluss der niederprioren Task [ScZu03].

Bei statischem Mapping werden alle LPs bei ihrer Erzeugung einem Rechner fest zugeteilt. Hierbei muss ein guter Kompromiss zwischen erwarteter Rechenlast einzelner LPs und dem erwarteten Kommunikationsaufkommen zwischen LPs gefunden werden. Ein Weg, diese Parameter zu ermitteln, besteht in einer zeitbegrenzten Vorabsimulation des Modells, bei dem die Rechenlast und das Kommunikationsaufkommen mitprotokolliert werden. Bei dynamischem Mapping kann diese Zuordnung zur Laufzeit der Simulation etwa aufgrund des Systemverhaltens (z. B. zu hoch werdende Last eines Rechnerknotens) geändert werden. Bei Änderung der Zuordnung migriert ein LP von einem Rechner zu einem anderen.

Statisches Scheduling wendet immer die gleiche Strategie zur Auswahl des als nächsten zu bedienenden LP an. Als gute Strategie hat sich herausgestellt, die in der Simulationszeit am weitesten zurückliegenden LP zu bevorzugen [Meh94b]. Bei dynamischem Scheduling kann die Scheduling-Strategie adaptiv vom bisherigen Systemverhalten abhängen. Es wird angenommen, dass nach einer Initialisierungsphase, in der eine konstante Anzahl LPs erzeugt wird, keine weiteren LPs generiert werden. Durch diese Forderung wird die Klasse statisch-verteilter Simulatoren beschrieben. Ein Beispiel für dynamisch-verteilte Simulatoren, bei der durch die Ausführung von Ereignissen, also während der Simulation, neue LPs erzeugt werden können, wird in [AgTi91] [TiAg89] gegeben.

3.4.3 Partitionierung von Graphen

Partitionierungs-Algorithmen können z. B. aus der Design-Automatisierung für VLSI-Systeme stammen. Bei Computer Aided Design (CAD) ist der erste Schritt die Beachtung von Kriterien hinsichtlich Fläche und Anschlussbedingungen. Die Algorithmen sind allgemeingültig und können auch für die PDES angewandt werden. Hinsichtlich Rücksetzungen müssen sie an Time Warp (TW) angepasst werden. Dabei sind heuristische[43] Algorithmen einfacher als numerische an PDES anpassbar [Spor94]. In Predictive Time Warp (PTW) kommt z. B. ein statischer, heuristischer, rekursiver „k-way“-Algorithmus zum Einsatz [ScTM97]. Ein Maß für die Güte der Partitionierung bei optimistischer Synchronisation ist die Rücksetzungshäufigkeit. Falls vorhanden, können bei der Partitionierung Informationen aus einer Vorsimulation berücksichtigt werden.

Anforderungen an Partitionierungsverfahren:

- die hierarchische Modellstruktur sollte möglichst genau erkannt werden;
- Komponenten mit hohem Kommunikationsaufkommen sollten nicht getrennt werden, auch wenn sie nicht direkt zusammenhängen;
- Partitionierungsgrenzen sollten nicht durch eine Struktur zusammenhängender Elemente verlaufen;
- das Kommunikationsaufkommen zwischen den Partitionen sollte reduziert werden;
- die Kommunikationskanäle zwischen den Partitionen sollten unidirektional sein, so dass die Ausbreitung von Rücksetzungen gehemmt wird.

In [BaRR01] werden die Performance-Faktoren, die eine PDES-Simulation betreffen, in sechs Klassen eingeteilt, die nicht unabhängig voneinander sind (Bild 50). Die Performance-Faktoren jeder Klasse müssen für maximalen Durchsatz optimiert werden.

[43] Heuristik ist die Bezeichnung für ein Lösungsverfahren, das nur zum Teil auf wissenschaftlich gesicherten Erkenntnissen, vorwiegend aber auf Hypothesen, Analogien oder Erfahrungen aufbaut [DuIn01].

WORKLOAD	Granularität, Speicher, Netzwerktopologie, E/A, Ereignis-Population, Ereignis-Wahrscheinlichkeit, Ereignis-Verzögerung, Startkonfiguration, Anzahl Simulationsobjekte, Anzahl Prozessoren
PROTOKOLL	konservativ: Lookahead, Verklemmungs-Aufhebung optimistisch: GVT-Berechnung, Annihilierung, Zustandsspeicherung, Speicherverwaltung
SIMULATION ENGINE	Ereignislisten-Handhabung, Zeitmodell, Speicherreservierung, Ablaufplanung
KOMMUNIKATION	Netzwerklatenzzeit, Protokoll-Schnittstelle, blockierende und nichtblockierende Kommunikation
SOFTWARE-SCHNITTSTELLE	Betriebssystem, Programmiersprache, Compiler-Optimierungen
HARDWARE	Prozessorgeschwindigkeit, Speicher- und Cache-Latenzzeit

Bild 50: Performance-Faktoren von PDES

Die Performance ist abhängig vom Partitionierungsgrad; bei zu vielen Partitionen sinkt sie wieder (Bild 51).

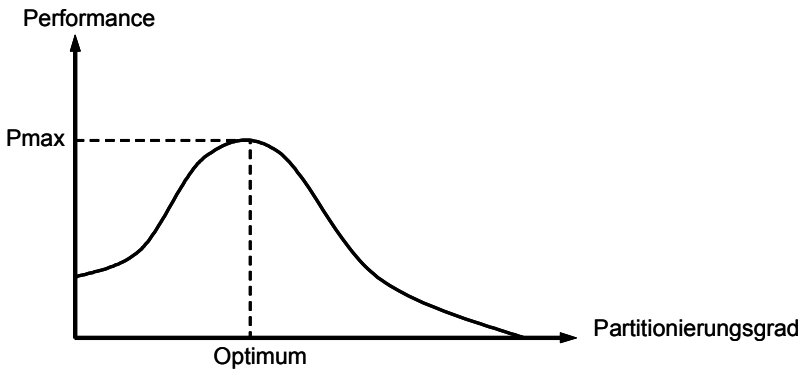


Bild 51: Performance abhängig vom Partitionierungsgrad

3.4.3.1 Heuristische Partitionierungsalgorithmen

Man unterscheidet stochastische, deterministische und heuristische Partitionierungsverfahren. Im Rahmen der Arbeit kommen heuristische Verfahren zum Einsatz. Alle unten spezifizierten iterativen Methoden modellieren einen Schaltkreis als einen ungerichteten Graphen oder Hypergraphen, ohne die Richtung von Signalen zu beachten. In einigen Situationen ist die Information über die Richtung von Signalen nützlich, um die Partitionierung zu verbessern [ChJB97] [CoZB94].

Der Algorithmus von Kernighan und Lin (KL) [KeLi70] ist ein Algorithmus zur Bi-Partitionierung. KL beginnt mit einer zufälligen Partitionierung und tauscht die Knoten der beiden Partitionen paarweise aus, so lange der Schnitt (cut size) verkleinert werden kann. Der KL-Algorithmus ist nur auf Knoten mit einheitlichen Gewichten anwendbar, und die Größe einer Partition ist fest. Dieser Algorithmus kann nicht für die Partitionierung von Hypergraphen eingesetzt werden. Der Algorithmus von Fiduccia und Mattheyses (FM) [FiMa82] ist eine Erweiterung des KL-Algorithmus. Da der FM-Algorithmus die Verschiebung einzelner Knoten zwischen Partitionen erlaubt, ist die Größe einer Partition veränderbar. Beim FM-Algorithmus ist die Auswahl des Knotens aus denjenigen, die als nächstes verschoben werden können und denselben Gewinn bringen, beliebig. Krishnamurthy [Kris-84] führte das Konzept der „level gains" ein, um zwischen Knoten entsprechend ihrer Gewinne durch spätere Verschiebungen zu unterscheiden. Sanchis [Sanc89] erweiterte das Konzept von Krishnamurthy zu einer k-way-Partitionierung (K-FM). Ein Knoten wird als frei markiert, wenn er nicht innerhalb eines Durchgangs des Algorithmus verschoben wird. Sonst wird er als gesperrt markiert. Der K-FM-Algorithmus von Sanchis kann für eine beliebige Anzahl von Partitionen und für Knoten mit unterschiedlichen Gewichten eingesetzt werden (Bild 52).

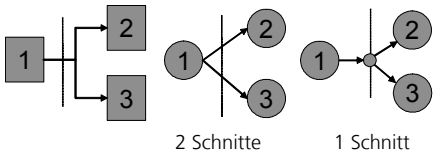


Bild 52: Partitionierung nach Sanchis

3.4.3.2 Clustering

Ansätze zur Optimierung von Time Warp basieren auf der Partitionierung des simulierten Systems in mehrere Cluster selbständiger LPs, die jeweils auf einem eigenen Prozessor ablaufen. Ziel dabei ist, die für eine Rücksetzung benötigte Rechenzeit entscheidend zu verringern, indem nur einige dieser LPs zurückgerollt werden müssen, anstatt eine sehr umfangreiche Partition innerhalb eines einzelnen LP zurückzurollen. Eine Folge davon ist, dass bei einer notwendig werdenden Rücksetzung die ganze Partition zurückgerollt und anschließend erneut simuliert werden muss, selbst wenn nur wenige Elemente betroffen waren. Durch Aufteilung des Gesamtsystems in mehr als einen logischen Prozess pro Simulator und Bildung von Clustern durch Zusammenfassung der LPs lässt sich die benötigte Reevaluierung nach einer Rücksetzung auf kleine Regionen begrenzen, ohne den Verwaltungsaufwand entscheidend zu erhöhen. Zum Beispiel basiert das Protokoll Clustered Time Warp (CTW) [AvTr95] auf einer solchen Clusterbildung. Innerhalb der einzelnen Cluster erfolgt eine sequenzielle Synchronisation der enthaltenen LPs, während die auf verschiedenen Prozessoren befindlichen Cluster untereinander mittels TW synchronisiert werden.

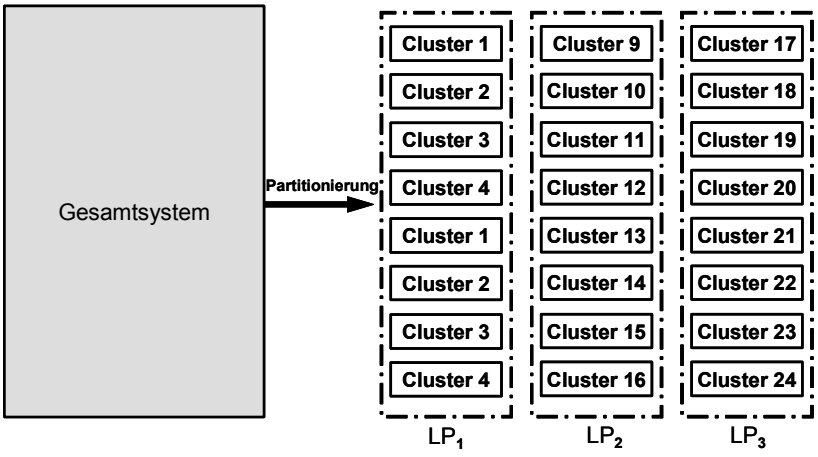


Bild 53: Partitionierung des Gesamtsystems in LP und Cluster

Die Partitionierung des Modells in LPs und Cluster (siehe Bild 53) kann in mehreren Stufen erfolgen: Partitionierung des Modells in LPs, Aufteilung der LPs auf die zur Verfügung stehenden CPUs und Partitionierung der LPs in Cluster. Die Cluster können wiederum in größere Cluster zusammengefasst werden (bottom-up Clustering). In der Literatur findet man einen Performance-Gewinn bis zu 60 % durch Clustering [AvTr95].

3.4.4 Lastenausgleich in verteilten Systemen

Je besser die statische Partitionierung ist, desto geringer ist der Kommunikationsaufwand bei der Simulation, woraus eine geringere Netzwerk- und CPU-Last und geringere Verzögerungen resultieren. Eine dynamische Neupartitionierung von LPs und Clustern innerhalb von LPs ist aufgrund des Aufwands für eine Datenmigration nicht immer vorteilhaft. Eine dynamische Datenmigration ist vor allem bei SM-Architekturen vorteilhaft. Das BS ist kein guter Entscheidungsträger, da es nur die Auslastung und nicht das gleichmäßige Voranschreiten der LVT der Prozesse berücksichtigt.

Zur Optimierung der Flusssteuerung existiert u. a. das Space-Time-Produkt-Verfahren von Tropper, das Anzahl und Größen von Ereignissen und Zuständen berücksichtigt [AvTr95].

$$ST_i(t) = \left(sizeof(event) \cdot Nevents + sizeof(state) \cdot Nstates\right) \cdot min\left(LVT_i\right)$$

$$\forall \left(P_i, P_j\right) \in P : \left|ST_i(t) - ST_j(t)\right| \leq \Delta$$

Das Ziel von Lastenausgleichsverfahren (load balancing) ist die Verringerung des Kommunikations-Overheads zwischen CPUs, LPs und Clustern durch deren Entkopplung sowie die Aufhebung eines eventuellen Flaschenhalses. Bei optimistischen Verfahren ist ein weiteres Ziel die Aufhebung oder Reduzierung von Rücksetzungen. Bei statischen Verfahren erfolgt die Partitionierung des Modells in Teilmodelle vor dessen Simulation in LPs und Cluster, bei dynamischen Verfahren währenddessen. Eine dynamische Partitionierung ist u. a. dann sinnvoll, wenn vor der Simulation keine ausreichenden Informationen vorliegen oder bei dynamischer Änderung der CPU- bzw. LP-Lasten infolge von Änderungen ihrer Belastungscharakteristik.

Mit Hilfe von dynamischem Lastenausgleich (dynamic load balancing, DLB) ist es z. B. möglich, unterschiedliche Lasten bei einer verteilten Simulation zu kompensieren. Dabei sind zentrale Verfahren für kleine und mittlere Systeme effizienter als dezentrale und einfache effizienter als komplexe. Es können sowohl Daten als auch LPs verschoben werden. Allerdings ist der entstehende Overhead bei langsamen Netzwerken nur schwer zu kompensieren. Ein Verfahren ähnlich dem bei CTW wird ergänzt, indem einzelne LPs von einem Cluster in einen anderen verschoben werden, falls dies aus Gründen der Prozessorauslastung für den weiteren Simulationsverlauf aussichtsreich erscheint. Ein solcher dynamischer Lastausgleich führt über die Reduzierung der Rücksetzungsdauer hinaus zu einer Verringerung der Anzahl sowie der zeitlichen Distanz von Rücksetzungen. Dies wird dadurch erreicht, dass sich ein oder mehrere Cluster von einem Simulationsprozess in einen anderen verschieben lassen und somit anschließend auf einem anderen, bis dahin weniger ausgelasteten Prozessor ablaufen, wodurch sich die gesamte Auslastung gleichmäßiger auf alle beteiligten Prozessoren verteilt und zu einem zügigeren Simulationsfortschritt beiträgt. Allerdings nimmt der Verschiebevorgang beträchtlich Zeit in Anspruch, so dass Geschwindigkeitssteigerungen vor allem bei schnellen Netzwerken oder Parallelrechnern erzielt werden.

3.5 Verteilte Ausführung von Systemen

Der Einsatz paralleler und verteilter Simulation hat gezeigt, dass bedeutende Fortschritte bezüglich der Reduzierung der Rechenzeit erzielt werden können. Gerade bei Simulationen sehr komplexer Modelle, die zudem einen sehr hohen Speicherbedarf besitzen, und bei stochastischen Simulationen, die häufig viele Simulationsdurchläufe erfordern, um Streuung und Zuverlässigkeit der Ergebnisse (siehe 2.1.5) zu ermitteln, ist die Rechendauer von ausschlaggebender Bedeutung. Dies gilt ebenso beispielsweise für Parameteroptimierungen mittels iterierter Simulation. Die Mehrzahl der zu simu-

lierenden technischen Systeme lässt sich in eine Anzahl von Komponenten zerlegen, die in der Regel gleichzeitig arbeiten. Naheliegend ist daher die Ausnutzung dieser inhärenten Parallelität der simulierten Modelle durch ihre Abbildung auf parallele Rechnerstrukturen, um so die bedeutend höhere Rechenkapazität im Vergleich zu Einzelprozessoren zu nutzen und den Zeitbedarf zu verringern. Dabei werden – neben den üblichen Multiprozessorsystemen wie massiv-parallele Architekturen, die auf solche parallel durchzuführenden Aufgaben spezialisiert sind und dazu über extrem schnelle Kommunikationsmechanismen verfügen –, auch günstigere, skalierbare Workstation-Cluster[44] eingesetzt, bestehend aus Workstations, die miteinander über ein herkömmliches LAN verbunden sind.

Zur Erzielung einer höheren Rechenleistung mittels paralleler Simulationsstrategien existieren verschiedene Protokolle, konservative und optimistische, die im Folgenden vorgestellt werden. Das simulierte Modell wird dabei in mehrere Prozesse aufgeteilt, die parallel zueinander auf den einzelnen Knoten der verteilten Rechenumgebung ablaufen. Ein solches Vorgehen wird als Co-Simulation bezeichnet (siehe 1.1 und 2.3.2).

Allgemeine Angaben zur Überlegenheit von konservativem oder optimistischem Protokoll können nicht formuliert werden, da Performance – aufgrund eines sehr hohen Grades an Verflechtungen von einflussnehmenden Faktoren – durch Modelle charakterisiert wird. Seitens der Plattform wird die Performance von der Hardware in sich (z. B. Verhältnis der Kommunikation zur Berechnungsgeschwindigkeit), dem Kommunikationsmodell (z. B. FIFO, Routing-Strategie, Netzwerktopologie, Möglichkeiten von Broadcast- und Multicast-Operationen) und dem Synchronisationsmodell (z. B. globale Steuereinheit, asynchrone verteilte Speicher, gemeinsame Variablen) beeinflusst, wodurch Protokolle über Plattformen hinweg weitgehend unvergleichbar werden. Protokollspezifische Optimierungen, d. h. Optimierungen in einem Protokoll, die kein Gegenstück im anderen Protokoll haben (z. B. lazy cancellation, siehe 3.6.2.3), verhindern einen „gerechten“ Vergleich. Daher ist es ohne experimentelle Evaluierung schwierig vorherzusagen, ob ein optimistischer oder ein konservativer Algorithmus für eine bestimmte Applikation eine bessere Performance ergibt. Wenn die Applikation allerdings schlechte Look-Ahead-Eigenschaften (siehe 3.6.1.2) besitzt, ist es wahrscheinlicher, dass mit optimistischen Methoden eine wesentliche Beschleunigung erreicht wird. Wenn erwartet wird, dass die Kosten für die Zustandsspeicherung dominieren, werden konservative Methoden bevorzugt.

3.5.1 Logische Prozesse

Die inhärente Parallelität eines simulierten Modells soll durch Aufteilung des Modells in mehrere LPs (siehe Bild 80 und [Fuji89]) und ihre nebenläufige Ausführung in möglichst hohem Maße ausgenutzt werden. So lassen sich Komponenten des simulierten Systems, die in der Realität gleichzeitig arbeiten, auf parallele Rechnerstrukturen abbilden, um Geschwindigkeitsvorteile zu erzielen. Jeder Prozess simuliert dabei einen Teil der Raum-Zeit-Ebene, der auch Ereignisstrukturregion genannt wird. Eine solche Region stellt die Menge aller Ereignisse eines Abschnitts der Simulationszeitachse oder eines Unterbereichs des Simulationsraums dar. In paralleler Weise werden alle eintretenden Ereignisse von einer Menge LPs synchron oder asynchron abgearbeitet. Dabei ermöglicht eine Simulationsumgebung den Austausch lokaler Daten zwischen den Prozessen und die Synchronisation lokaler Aktivitäten durch globale oder externe Ereignisse in Form von Nachrichten. Den einzelnen LP LP_i wird dabei jeweils eine Region R_i des Simulationsmodells zugeordnet, deren lokale oder interne Ereignisse ähnlich wie bei der herkömmlichen DES von einer Simulation Engine SE_i abgearbeitet werden. Dabei können globale Ereignisse erzeugt und die LVT des Prozesses angepasst werden. Jeder logische Prozess kann nur auf den Teil der Zustandsvariablen zugreifen, der der ihm zugewiesenen Region entspricht. Kommunikationsschnittstellen CI_i (Communication Interfaces) übernehmen die Übertragung der Nachrichten zwischen den Simulation Engines (siehe Bild 80).

[44] Ein Cluster ist allgemein eine Gruppe miteinander vernetzter Rechner (Knoten), die gemeinsam an einem Problem arbeiten. Ein Cluster of Workstations (COW) ist ein Verbund von Workstations über ein Netzwerk. Als Network of Workstations (NOW) bezeichnet man vernetzte Rechner, die – anders als die Knoten in einem Cluster – auch als unabhängige Workstations genutzt werden, wie COW, aber auch ohne engere Netzwerkanbindung.

Bei der synchronen LP-Simulation ist zu jedem Echtzeitpunkt für alle LPs dieselbe globale Simulationszeit als LVT „sichtbar“, während bei der asynchronen Simulation i. A. unterschiedliche LVTs gelten. Ein bedeutender Nachteil der synchronen Simulation ist, dass bei einer unterschiedlichen Auslastung der einzelnen LPs schnellere Prozesse auf langsamere Rücksicht nehmen und zeitweise ihre Ausführung blockieren müssen, worunter die Performance leiden kann. Dies soll bei der asynchronen Simulation vermieden werden, indem Ereignisse mit verschiedenen Zeitstempeln, die einander nicht beeinflussen, zur nebenläufigen Verarbeitung zugelassen werden, wodurch sich die Simulation häufig stark beschleunigen lässt. Dabei muss auf die Erhaltung der Kausalität geachtet werden, was unter Umständen ein erhöhtes Kommunikationsaufkommen zur Folge hat. Ein Kausalitätskonflikt liegt vor, wenn ein LP ein globales Ereignis erhält, das im Nachhinein Einfluss auf den bisherigen Simulationsverlauf genommen hätte und sich somit in seiner virtuellen Vergangenheit [Jef85a] befindet. Dies kann dann eintreten, wenn der Zeitstempel eines eintreffenden globalen Ereignisses vor der lokalen virtuellen Zeit des empfangenden LP liegt und das Ereignis somit aus Sicht des Prozesses zu spät ankommt, weil es nun nachträglich berücksichtigt werden müsste. Verschiedene Synchronisationsmechanismen werden eingesetzt, um in einem solchen Fall einen Kausalitätsverlust zu vermeiden oder aber Fälle dieser Art grundsätzlich zu umgehen und so die Kausalität zu jedem Zeitpunkt zu erhalten. Ein Kausalitätsverlust lässt sich ausschließen, wenn jeder logische Prozess seine Ereignisse in monoton steigender zeitlicher Reihenfolge ihrer Zeitstempel abarbeitet, was auch der lokalen Kausalitätsbedingung entspricht [Fuji90]. Diese hinreichende Bedingung ist allerdings dann nicht notwendig, wenn mehrere Ereignisse desselben LP voneinander unabhängig sind und somit parallel abgearbeitet werden können.

3.5.2 Kopplung von Ausführungsumgebungen

Die Notwendigkeit, unterschiedliche Werkzeuge einzusetzen ergibt sich durch

- Mangel an durchgängiger Unterstützung des gesamten Entwicklungsprozesses bei den Werkzeugen;
- die notwendige Zusammenarbeit verschiedener Entwicklungsabteilungen mit ihren gewachsenen Strukturen (Werkzeuge, Wissen und Modelle);
- die gemeinsame Entwicklungsarbeit von Hersteller und Zulieferer mit unterschiedlichen Werkzeugen;
- den Wunsch, für jede Entwicklungsaufgabe das am besten geeignete Werkzeug einzusetzen.

Eine perfekte und lückenlose Schnittstellenlösung kann es wegen der zum Teil sehr unterschiedlichen Arbeitsweise und der speziellen Eigenschaften der Werkzeuge, durch die sie sich im Wettbewerb behaupten müssen, nicht geben. Es gibt jedoch Möglichkeiten zur Werkzeugkopplung und zum Modellaustausch [WoMB01].

Im Bereich des rechnergestützten Entwurfs elektronischer Systeme erlangen Co-Simulations-Konzepte besonders in den frühen Phasen des Entwurfsprozesses zunehmende Bedeutung. Um den Forderungen nach steigender Qualität, kürzeren Entwicklungszeiten und geringeren Kosten nachzukommen [Bark96], müssen Entwurfsfehler bereits frühzeitig im Entwurfsablauf, möglichst schon während der Spezifikation, entdeckt und beseitigt werden. Dies erfordert geeignete Werkzeuge zur parallelen Simulation heterogener elektronischer Systeme, die aus verschiedenen Aspekten heraus und auf unterschiedlichen Abstraktionsebenen modelliert wurden.

3.5.2.1 Steuerung

Man unterscheidet zentrale und dezentrale Steuerung. Bei zentraler Steuerung werden alle Simulatoren durch einen zentralen Kontrollprozess gesteuert. Der Vorteil ist, dass das System mit weniger Aufwand erweiterbar ist und der zentrale Kontrollprozess eine bessere Systemkenntnis als verteilte Kontrollprozesse besitzt. Allerdings ist das System nicht beliebig skalierbar, da der Knoten des Kontrollprozesses zum Flaschenhals werden kann.

3.5.2.2 Middleware

Als Middleware wird eine Kommunikations-SW bezeichnet, die zwischen der Anwendungs-SW und der BS-SW angesiedelt ist und verschiedene Applikationskomponenten integriert, ohne dass dabei der Anwender eine spezifische Kommunikations-Infrastruktur erzeugen muss. Kommunikations-Middleware dient der vereinfachten und standardisierten Kommunikation der Komponenten eines verteilten Systems [ScWe04]. Mit ihrer Hilfe kann bei einem geschichteten SW-Modell die SW in den Schichten darüber mit SW in den Schichten darunter kommunizieren und die Anzahl der Schnittstellen wird minimiert. Hiermit lassen sich Komponenten nutzen und zusammenfassen oder Dienste, die über klar definierte Schnittstellen verfügen, auffinden und ansprechen [DuIn01]. Durch den Einsatz von Middleware steigt der Kommunikationsaufwand. Im Gegensatz dazu erfordert die direkte Kommunikation über IPC einen wesentlich höheren Entwicklungsaufwand, ist aber schneller, da kein Kommunikationsaufwand durch Middleware entsteht. Außerdem ist bidirektionale Kommunikation (asynchroner Remote Procedure Call, RPC) ohne ein fixes Client-Server-Prinzip möglich.

Als Middleware kommen z. B. in Frage:

- Common Object Request Broker Architecture (CORBA) der Object Management Group [OMG05]. Im Gegensatz zu Remote Method Invocation (RMI) o. ä. ist CORBA kein Produkt, sondern lediglich eine Spezifikation der OMG. CORBA ist plattform- und programmiersprachenunabhängig [ScWe04].
- Distributed Component Object Model (DCOM) von Microsoft.
- Java Remote Method Invocation (Java-RMI) von Sun. RMI ist ein speziell für die Programmiersprache Java von der Firma Sun entwickeltes Konzept. RMI funktioniert als Client-Server-Architektur zwischen lokalen und entfernten Objekten. Die Kommunikation zwischen diesen wird von RMI übernommen und ist dadurch für den Anwender transparent. Für die Übergabe von Objektparametern gibt es zwei Möglichkeiten: Referenzübergabe (call by reference) und Wertübergabe (call by value) [ScWe04].
- Message Passing Interface (MPI) und Parallel Virtual Machine (PVM). Eine Möglichkeit der Parallelisierung über Rechnergrenzen hinweg sind Nachrichtenbibliotheken wie MPI oder PVM. MPI eignet sich besonders für homogene Cluster, während PVM seine Stärken bei heterogenen Clustern mit unterschiedlichen HW-Architekturen und BS ausspielen kann [Görz00]. Mit PVM können unterschiedliche Systeme miteinander verbunden werden. Dabei wird auf jedem Prozessor eine betriebssystemseitige Erweiterung, der PVM-Dämon, gestartet. Alle Dämonen der beteiligten Rechner bilden die jeweilige virtuelle Maschine [PVM05]. MPI ist eine Spezifikation einer Bibliothek zur Programmierung von Mehrprozessorsystemen mit verteiltem Speicher [Gropp 95]. MPI hat sich an PVM orientiert [Hempel 96]. Allerdings sind die Systeme nicht kompatibel [MPI05].
- Transmission Control Protocol (TCP)- und User Datagram Protocol (UDP)-Sockets. Die Socket[45]-Schnittstelle ist der De-facto-Standard einer Programmierschnittstelle für die direkte Nutzung von TCP/IP (Transmission Control Protocol/Internet Protocol) und UDP/IP zur Kommunikation[46] und Grundlage für viele Middleware-Systeme.
- JavaSpaces. Die 1998 freigegebene JavaSpaces-Spezifikation in der Version 1.0 [JaSp98] beschreibt einen Mechanismus zur Erstellung asynchron arbeitender verteilter Anwendungen. Dieser beruht auf Austausch von Objekten, so dass damit sowohl Daten als auch Funktionalität zwischen den Teilen einer verteilten Anwendung ausgetauscht werden können. Die auszutauschenden Objekte sind Java-Objekte, die eine spezielle Schnittstelle implementieren (z. B.

[45] Ein Socket ist ein Kommunikationsendpunkt innerhalb einer Anwendung. Er wird durch eine IP-Adresse und eine Port-Nummer adressiert.

[46] Eine Bewertung von TCP-/UDP-/UNIX-Sockets, Pipes, SM u. a. Kommunikationsalternativen sowie eine vergleichende Übersicht von MPI, PVM, ClearSim (3.7.4), SimBa (3.7.1), HLA (3.5.2.2) u. a. findet sich z. B. in [Brie01].

`net.jini.space.Entry`) müssen. Weiterhin dürfen die öffentlichen Attribute der Objekte nur Objektreferenzen enthalten, und die referierten Objekte müssen serialisierbar sein.

- SOAP (Simple Object Access Protocol). Middleware auf Basis von XML und HTTP.
- High Level Architecture (HLA). Die HLA ist ein Interoperabilitätsstandard für Simulationsanwendungen, der den Anspruch hat, nicht auf eine bestimmte Kategorie von Simulatoren beschränkt zu sein [DaKW98]. Hauptbestandteil von HLA ist eine Spezifikation zur Beschreibung der Schnittstelle zwischen verschiedenen Simulatoren (Federates) und einer Infrastruktur (Runtime Infrastructure, RTI). Die HLA stellt eine Architektur zur Verfügung, die sowohl analytische Simulationen als auch Simulationen virtueller Umgebungen überspannt und wurde im September 1996 zur Standardarchitektur für alle Modellierungs- und Simulationsaktivitäten des Verteidigungsministeriums der USA (Department of Defence, DoD) [DMSO05] [HLAM05]. Im Unterschied zur DEVS ist HLA ein allgemeiner Modellspezifikationsformalismus und dient zur allgemeinen Standardisierung von Schnittstellen, während der Schwerpunkt von DEVS nur die Integration ereignisgesteuerter, kontinuierlicher Systeme ist [SaZe00]. Die HLA-Laufzeitumgebung nutzt CORBA als interne Kommunikationsinfrastruktur und definiert darauf Schnittstellen, die zum Arbeiten mit verteilten Objekten in der Simulation nützlich sind [Bach00]. Obwohl dabei die Möglichkeit der Kopplung an sich, d. h. Interoperabilität und Portabilität, und nicht der Geschwindigkeitsgewinn durch Parallelisierung der Modellausführung im Vordergrund steht, ist das RTI-RTI-Schnittstellenprotokoll nicht spezifiziert, so dass nur RTI-Komponenten eines Herstellers miteinander kompatibel sind. HLA unterscheidet sich von anderen Middleware-Standards für allgemeine Applikationen durch spezielle, für Simulationsanwendungen unbedingt erforderliche Dienste. Das in HLA integrierte Zeitmanagement erlaubt die Sychronisation von Federates hinsichtlich Zeit- und Datenaustausch zur Laufzeit, und es sind intelligente Verteilungsmechanismen für Daten vorhanden [Stra99]. Obwohl HLA aus dem militärischen Bereich stammt, ist der Einsatz im zivilen Bereich ebenfalls möglich. Verschiedene HLA-Schnittstellen für Simulatoren wurden innerhalb von Forschungsprojekten entwickelt (z. B. SLX und SIMPLEX) [StSc01].

RMI und JavaSpaces haben den Vorteil, dass sie kompatibel zu Java und im Gegensatz zu CORBA frei erhältlich sind; aber den Nachteil, dass das Java Native Interface (JNI) zur Kommunikation mit C-Funktionen benötigt wird. Zur Steigerung der Performance kann die Koordination der Nachrichtenübertragung zwischen Remote-Prozessoren mit Hilfe von JavaSpaces [Hupf00] alternativ zu RMI zentralisiert erfolgen. Die RMI-Performance ist nicht signifikant geringer als die Java-Sockets-Version derselben Client-Server-Applikation. Allerdings ist die Entwicklungszeit der RMI-Applikation signifikant kleiner. Die Java Sockets sind etwa 30 Prozent schneller als RMI und 15 Prozent schneller als JavaSpaces. Wenn Client und Server auf dem selben Host-Rechner laufen, unter der Randbedingung, dass die Programmiersprache Java zum Einsatz kommt, ist die Latenzzeit von Java-RMI ca. 25 % kleiner als die von CORBA, und die von SOAP etwa eine Größenordnung größer [DaPa02]. Für Situationen, bei denen legacy-Simulationsmodelle in verschiedenen Sprachen involviert sind, hat die HLA aufgrund ihrer Simulationsorientierung in Form von simulationsbezogenen Diensten Vorteile. Wenn legacy-Datenbanken involviert sind, wird CORBA zur Implementierung der Middleware empfohlen. Und wenn die Implementierung größtenteils neu ist, ist RMI aufgrund seines engen Bezuges zu der Programmiersprache Java und deren Objektorientierung von Vorteil [BuJa98].

Netzwerkbezogene Performance-Aspekte sind u. a. bei der Flusskontrolle und der eingesetzten Netzwerkprotokolle zu beachten. Wenn zu viele Ereignisse versendet werden, so dass der Empfänger sie nicht verarbeiten kann, wird er überlastet, und zu viele bereits LP-externe Ereignisse müssen berücksichtigt werden, beispielsweise bei Rücksetzungen. Hinsichtlich der Netzwerkprotokolle muss ein Kompromiss zwischen Geschwindigkeit, Zuverlässigkeit (siehe 2.1.5) und Risiko gefunden werden. Es werden in der Literatur u. a. auch Techniken wie Broad- und Multicast anstatt rekursive Negativ-Nachrichten in Erwägung gezogen. Wenn die Paket- oder Fenstergröße bei der Datenübertragung zu klein ist, ist die Systemperformance im Vergleich zur optimalen Fenstergröße ebenfalls

deutlich kleiner. Ist die Fenstergröße zu groß, nimmt die Verzögerung, bis ein Ereignis empfangen bzw. verarbeitet wird, wieder zu, und damit die Systemperformance ab.

Automotive Middleware ist z. B. OSEK/VDX COM 3.0, das einen Ansatz für eine einheitliche Lösung im Kommunikationsbereich darstellt. Wie OSEK OS nutzt auch OSEK COM OIL als standardisierte Konfigurationssprache und hebt sich damit von proprietären Lösungen ab. OSEK COM ist offen für verschiedene Data Link Layers (DLLs). Neben CAN sind z. B. auch Anbindungen an LIN und FlexRay möglich [Scho03].

3.5.2.3 Kopplung auf Code-Ebene

Eine Kopplung von Werkzeugen durch die Integration von generiertem Programmcode kann den Einsatz spezialisierter Simulatoren ausschließen und erfordert nach jeder Modelländerung einen erneuten Bindevorgang, evtl. mit manueller Code-Aufbereitung [ErST94] [ScTM96]. Durch die Ankopplung der Werkzeuge an eine Simulationsumgebung entfallen diese Einschränkungen [Stre96].

Bei einer Kopplung auf Code-Ebene ist evtl. zusätzliche manuelle Codierung notwendig (siehe Bild 54).

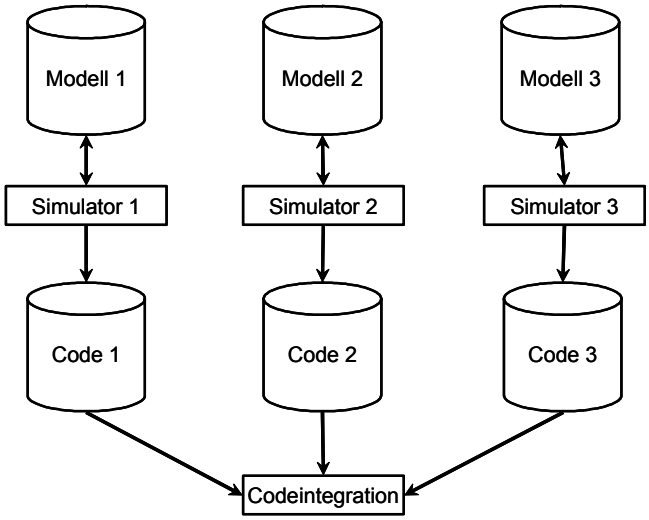


Bild 54: Kopplung auf Code-Ebene

Dieses Prinzip der Kopplung ist zwar bei den meisten Werkzeugen, die Benutzercode unterstützen, möglich, jedoch ist die Performance aufgrund der impliziten indirekten Übertragung der Daten im Vergleich zur Kopplung über eine dedizierte Schnittstelle nicht optimal.

Ein Beispiel für die Kopplung auf Code-Ebene unter Einbeziehung der Simulationsumgebung ist die Kopplung der Rapid Controller Prototyping-Umgebung RTI von dSPACE [dSPA05] und den CASE-Werkzeugen STATEMATE und MATLAB [Häfn98]. Dabei dient die HW als CPU-basierter „Simulationsbeschleuniger“. Das Herunterladen des Modellcodes auf die dSPACE-HW ist in Bild 55 dargestellt, die MATLAB/dSPACE-Umgebung in Bild 56. Die dSPACE-Umgebung kann für Rapid-Prototyping-Zwecke für HIL-Tests (siehe 2.3.1) dienen, oder auch zum Zweck der Simulationsbeschleunigung durch spezielle HW, z. B. als Alternative zu FPGAs. Die Simulation mit FPGAs würde zwar i. A. schneller ablaufen, aber der Entwurfsprozess wäre auch unflexibler, da der Entwurf des Modells in einer HW-Beschreibungssprache, oder die Konvertierung in eine solche, notwendig wäre. Darüber hinaus müsste eine Platzierung und Wegfindung angewendet werden[47].

[47] Da bei der Simulationsumgebung die Variante ohne Middleware implementiert ist, steht die Simulationsumgebung ausschließlich für die UNIX-Plattform zur Verfügung; es besteht keine Möglichkeit zur Kommunikation mit Windows-Prozessen. Da dSPACE keine UNIX-Bibliotheken für die vorhandene Alpha-CPU zur Verfügung stellt, kann nur der DSP zur Simulation eingesetzt werden.

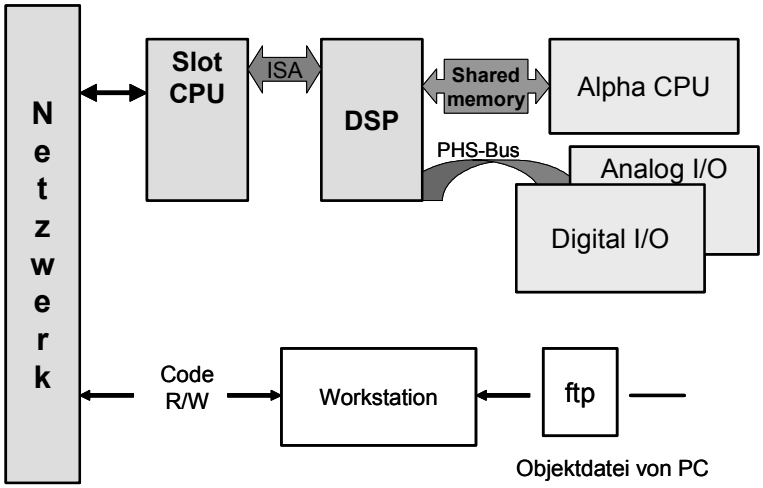


Bild 55: Kommunikation mit dSPACE HW

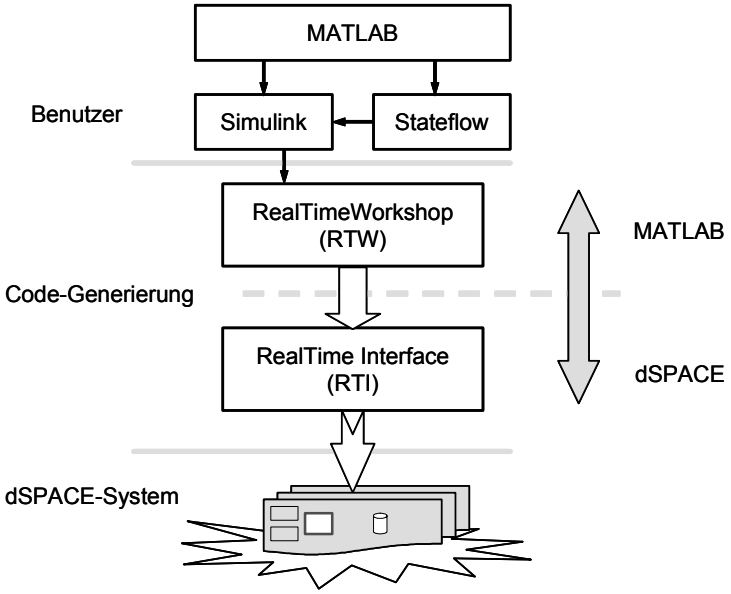


Bild 56: MATLAB/dSPACE-Umgebung

3.5.2.4 Kopplung auf CASE-Werkzeug-Ebene (bilaterale Kopplung)

Bei der bilateralen Kopplung der Simulatoren werden individuelle, proprietäre Schnittstellen gekoppelt (siehe Bild 57). Beispiele sind das MATLAB Integration Package (MIP) von ETAS, mit dem es möglich ist, ASCET und MATLAB zu koppeln[48]. Beim MIP findet die Kopplung unter Einbindung des von MATLAB generierten Codes in ASCET statt, d. h. die Simulatoren werden nicht direkt miteinander gekoppelt.

[48] MIP wurde von ETAS durch INTECRIO (siehe 3.3.2) ersetzt.

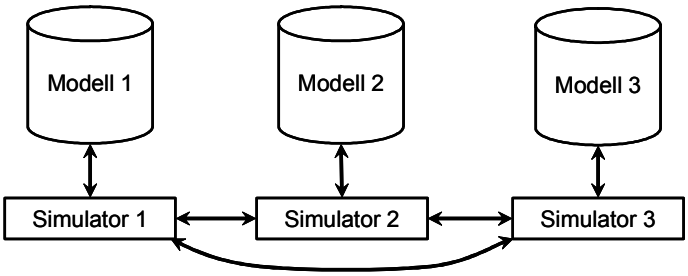


Bild 57: Bilaterale Kopplung

3.5.2.5 Kopplung über ein Framework

Bei der Simulatorkopplung ist das Ziel der Kopplung die Simulation des Gesamtentwurfs. Die dabei bestimmten Simulationsergebnisse müssen den Werten entsprechen, die bei der Simulation des gesamten Entwurfs durch nur einen (hypothetischen) Simulator berechnet werden würden. Daraus ergeben sich die folgenden Anforderungen an eine Co-Simulations-Umgebung [ErST94]:

- Die an der Co-Simulation beteiligten Simulatoren müssen in ihrer Simulationszeit synchronisiert werden. So wird z. B. der ereignisdiskrete Simulator wegen seines implementierten Algorithmus schneller in der Simulationszeit voranschreiten können als der kontinuierliche Simulator. Bei einem Datenaustausch zwischen den beiden Simulatoren muss der ereignisdiskrete Simulator auf die Simulationszeit des kontinuierlichen Simulators zurückgesetzt und die Simulation ab dieser Stelle mit den neuen Eingabewerten weitergeführt werden. Diese Anforderung setzt die Möglichkeit voraus, dass die Simulatoren in der Simulationszeit zurückgesetzt werden können [Pril93].
- Es muss eine Kontrolle aller Simulatoren durch die Kopplung möglich sein. Die einzelnen Simulatoren müssen von außen gesteuert werden können, um die Synchronisation zu realisieren.
- Die Datenkonsistenz zwischen allen beteiligten Simulationssystemen muss garantiert sein. Das geschieht durch den Austausch von Daten während der Simulation.

Die Simulatoren werden nicht direkt, sondern über eine den Simulatoren übergeordnete Instanz, die das Framework darstellt, gekoppelt. Dieses Framework übernimmt die Synchronisation der Simulatoren. Dadurch können unterschiedliche Synchronisationsalgorithmen verwendet werden, sofern die angekoppelten Simulatoren diese unterstützen und die Simulationsumgebung diese anbietet. Die Umgebung besitzt weiterhin die Aufgabe, die entsprechenden Ein- und Ausgabewerte zum richtigen Simulationszeitpunkt zwischen den Simulatoren auszutauschen. Es existieren definierte Schnittstellen zwischen den Simulatoren und der Backplane. Dadurch wird nur ein geringer Arbeitsaufwand für die Integration eines Simulators in die Co-Simulations-Umgebung notwendig. Durch eine Schnittstellendefinition besteht die Möglichkeit, nicht nur existierende, sondern auch noch nicht entwickelte Simulatoren in die Co-Simulationsumgebung einzubinden. Bei einem Backplane-Konzept können die jeweils besten Werkzeuge für eine Aufgabenstellung eingesetzt und die während jahrelanger Benutzung eines Werkzeugs evtl. erstellten Modellbibliotheken weiterverwendet werden. Durch den Einsatz unterschiedlicher Simulatoren verschiedener Hersteller können auch deren Modell-Bibliotheken beim Systementwurf eingesetzt werden. Durch die zusätzlich vorhandene Instanz zur Koordinierung der Simulatoren geht Performance verloren, wenn die gesamte Co-Simulationsumgebung auf einem Rechner betrieben wird. Wenn die Simulatoren nur über die Backplane miteinander kommunizieren können, kostet das ebenfalls Performance, im Vergleich zur direkten Simulatorkopplung. Dieser Performance-Verlust äußert sich in einer längeren Rechenzeit und einem größeren Speicherbedarf.

3.6 Synchronisationsverfahren für die PDES

Die Synchronisation der Ereignisausführungen stellt das zentrale Problem verteilter und paralleler Simulation dar. Verteilte Simulation versucht, unabhängige Ereignisse nebenläufig auszuführen und so die Simulation zu beschleunigen. Die Beschleunigung verteilter und paralleler Simulation hängt entscheidend davon ab, wie viele Ereignisausführungen nebenläufig ausgeführt werden können und wie gering der benötigte Synchronisationsaufwand ist. Verteilte Verfahren lassen sich direkt zur parallelen Simulation einsetzen, und die Umkehrung gilt in den meisten Fällen.

Wesentlicher Bestandteil einer verteilten Simulation ist ein Synchronisations- und Simulationsalgorithmus, der dafür sorgt, dass die Kausalordnung zwischen den auszuführenden Ereignissen entsprechend der vom Benutzer vorgegebenen Modellbeschreibung eingehalten wird. Fehler, die aus einer falschen Ereignisverarbeitungsreihenfolge resultieren, werden als Kausalitätsfehler bezeichnet. Eine ereignisdiskrete Simulation, die aus LP besteht, die ausschließlich durch den Austausch von Nachrichten mit Zeitstempeln interagieren, gehorcht der lokalen Kausalitätsbedingung nur dann, wenn jeder LP Ereignisse in nicht fallender Reihenfolge ihrer Zeitstempel verarbeitet. Dies garantiert jedoch nicht, dass die Simulation sinnvolle Ergebnisse liefert. Jedes Simulationsmodell muss validiert werden, bevor den Ergebnissen vertraut werden kann [Fuji00].

Das Problem der Gewährleistung, dass Ereignisse in der Reihenfolge ihrer Zeitstempel verarbeitet werden, heißt Synchronisationsproblem. Wenn die Simulation über mehrere Prozessoren verteilt ist, wird für die gleichzeitig ablaufende Ausführung ein Mechanismus benötigt, so dass exakt dieselben Ergebnisse erzielt werden wie bei einer sequenziellen Ausführung. Der Synchronisationsalgorithmus muss nicht gewährleisten, dass Ereignisse in verschiedenen Prozessoren in der Reihenfolge ihrer Zeitstempel verarbeitet werden, sondern nur, dass das Ergebnis dasselbe ist, als wäre dies der Fall.

Da Simulatoren die ihnen zugeteilten Modelle in der Regel unterschiedlich schnell simulieren, müssen sie in ihrer lokalen Simulationszeit synchronisiert werden, damit beim Austausch von Nachrichten keine Kausalitätsfehler auftreten können. Ein Kausalitätsfehler liegt dann vor, wenn ein Simulator von einem anderen eine Nachricht erhält, die eigentlich zu einem Zeitpunkt in seiner Simulationsvergangenheit hätte eintreffen müssen und somit die implizit getroffenen Voraussetzungen ungültig macht. Das Synchronisationsproblem wird dadurch entscheidend erschwert, dass diese Kausalordnung nur implizit definiert ist und sich die Menge aller auszuführenden Ereignisse erst im Verlauf der Simulation ergibt. Bei unabhängigen Modellen ist jede Simulationsreihenfolge möglich. Z. B. sind SS_1 (State Stack, Synchronisationspunkt) und SS_2 in Bild 56 unabhängig [Kepp94]. Synchronisationspunkte sind hier *F* und *J*.

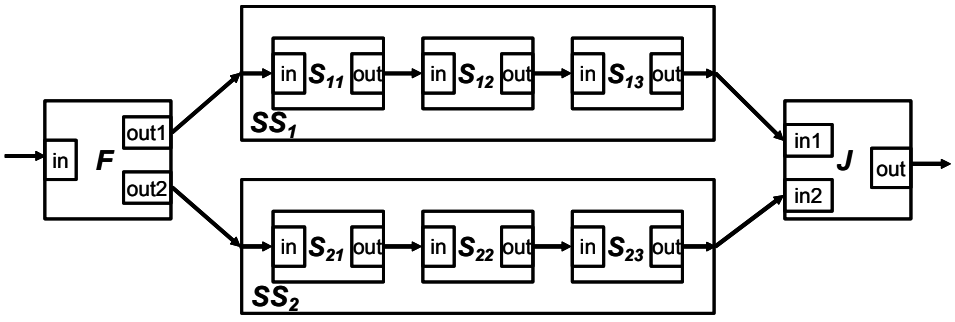


Bild 58: Synchronisation bei unabhängigen Modellen

Die zur Synchronisation verteilter Systeme verwendeten Algorithmen (Simulationsstrategien, Synchronisationsprotokolle) lassen sich in drei Gruppen aufteilen: konservative, optimistische [TaSM95] [ScTM95] [MiLD92] und aus konservativen und optimistischen kombinierte (hybride). Der Erfolg konservativer und optimistischer Verfahren ist an die Güte der Modellpartitionierung geknüpft (siehe 3.4). Optimal sind identische Progressionsraten aller LVT.

Bei konservativen Synchronisationsprotokollen (siehe 3.6.1) werden Ereignisse ausschließlich in der Reihenfolge ihrer Zeitstempel verarbeitet. Es wird von vornherein garantiert, dass alle Simulationsschritte im logischen System mit dem physikalischen System übereinstimmen. Jeder Simulationsschritt wird so lange aufgeschoben, bis alle notwendigen Informationen dafür vorliegen. Die konservativen Methoden können ineffizient sein, weil die LPs warten, bis die für den nächsten Simulationsschritt relevanten Informationen eingetroffen sind [Reed83] [ReM88a] [Seet78]. Optimistische Synchronisation (siehe 3.6.2) basiert auf der Annahme, dass jederzeit alle notwendigen Informationen für die Durchführung des nächsten Simulationsschritts vorliegen. Es wird kein Simulationsschritt aufgeschoben, obwohl dafür relevante Informationen fehlen können. Bei optimistischer Synchronisation werden Fehler während der Ausführung entdeckt und durch bestimmte Mechanismen wieder korrigiert. Jeder LP kann jede Nachricht sofort nach ihrem Eintreffen solange bearbeiten, bis ein Fehler eintritt. Ein Fehler liegt vor, wenn eine Nachricht eintrifft, die in der Simulationsvergangenheit des LP liegt. Im Fehlerfall muss der LP seinen Zustand auf einen Zeitpunkt vor Eintreffen der Nachricht zurücksetzen und mit der Simulation von diesem Zeitpunkt aus fortfahren. Alle Aktivitäten, die der LP fälschlicherweise ausgeführt hat, müssen rückgängig gemacht werden. Dies kann das Löschen von Nachrichten, die der LP bereits gesendet hat, erfordern und damit das Zurücksetzen anderer LPs bewirken [Brya79] [SoBW88] [JeSo82] [JeSo83] [Jef85b] [Jef85a]. Das Erkennen und die Beseitigung der Auswirkungen falscher Simulationsschritte kann bei optimistischen Simulationsstrategien erheblichen zusätzlichen Aufwand mit sich bringen [JeSo85] [LaMu83]. Die Performance einer verteilten Simulation kann durch Optimismusbegrenzung bei optimistischer Synchronisation erhöht werden. Alternativ sind z. B. auch eine Protokollumschaltung und eine Reduktion der CPU-Zeit von in Rücksetzungen involvierten CPUs möglich.

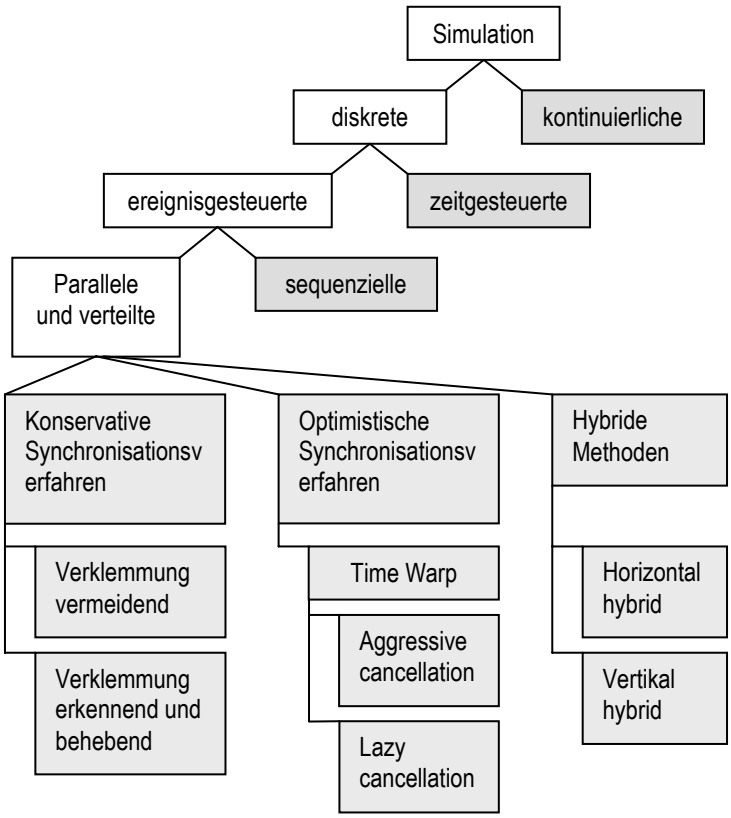


Bild 59: Parallele und verteilte Simulation

Eine Klassifikation von Mechanismen optimistischer Synchronisationsprotokolle ist in [SpSS99] enthalten. Hybride Synchronisationsprotokolle umfassen das Spektrum, das zwischen den oben beschriebenen Extremen liegt. Das grundsätzliche Entwurfsmerkmal dieser Klasse ist es, einen Ausgleich zwischen den extremen Strategien zu wählen, mit dem Ziel, die Leistungsfähigkeit der verteilten Simulation physikalischer Systeme zu steigern. Diese dritte Klasse kann realisiert werden, in-

dem begrenzter Optimismus in die rein konservative Strategie integriert wird, oder der Optimismus der rein optimistischen Strategie eingeschränkt wird[49]. In Bild 59 wird eine Auswahl paralleler und verteilter Simulationsmethoden gezeigt.

3.6.1 Konservativer Ansatz nach Chandy-Misra-Bryant

Die Grundidee konservativer verteilter Simulationsalgorithmen besteht darin, bei der Ausführung von Ereignissen die Kausalordnung grundsätzlich einzuhalten: Ein Ereignis darf erst dann ausgeführt werden, wenn vorab sichergestellt wurde, dass es kausal unabhängig von allen anderen zukünftig noch auszuführenden Ereignissen ist. Kausalitätsverletzungen werden strikt vermieden. D. h. bei der konservativen Methode [ChHM79] [ChMi81] [Misr86] wird eine Nachricht erst dann von einem LP bearbeitet, wenn sichergestellt ist, dass an diesen LP keine Nachricht mit einer kleineren Zeit ankommen kann. Konservative Simulationsverfahren wurden bereits 1977 von Chandy und Misra [ChMi79] [ChMi81] sowie von Bryant [Brya77] [Brya79] (CMB-Verfahren) vorgestellt.

Es werden nur sichere Ereignisse abgearbeitet, d. h. nur Ereignisse mit einem Zeitstempel bis zu einem Zeithorizont, vor dem sicher keine globalen Ereignisse mehr eintreffen können. Die lokalen und globalen Ereignisse müssen in chronologischer Reihenfolge entsprechend ihrer Zeitstempel abgearbeitet werden. Weil für den Fortschritt der Simulation LPs auf die Nachrichten anderer LPs warten müssen, besteht die Gefahr der Entstehung von Verklemmungen[50] (Deadlocks), wenn alle Prozesse auf das Eintreffen von Nachrichten warten, gleichzeitig aber in ihrer Ausführung blockiert sind. Es wurden zahlreiche Verfahren zur Vermeidung von Verklemmungen vorgestellt. Eine Alternative zur Verklemmungserkennung und -beseitigung ist z. B. die Verklemmungsvermeidung durch die Nullnachrichten-Methode (siehe 3.6.1.3) oder ähnliche Hilfsnachrichten, die lediglich der Synchronisation oder der Blockierungserkennung dienen [ChMi79] (s. u.). Dabei schickt jeder LP beim Senden einer Nachricht *(t, m)* über einen Ausgangskanal eine so genannte Nullnachricht *(t, NULL)* mit der gleichen Zeit *t* über alle anderen Ausgangskanäle. Dadurch wird der nachfolgende LP darüber informiert, dass über diesen Kanal keine Nachricht mit kleinerer Zeit mehr ankommen wird. Ist über jeden Eingangskanal eines LP eine Nachricht *(t, m)* oder eine Nullnachricht *(t, NULL)* eingetroffen, kann die frühestmögliche Nachricht des LP ermittelt und gegebenenfalls simuliert werden. Diese Methode kann das Nachrichtenaufkommen stark erhöhen, insbesondere dann, wenn die Anzahl von Verzweigungen im Kommunikationsgraphen und damit auch die Anzahl erzeugter Nullnachrichten hoch ist [LuSh89].

Da die Kausalordnung vor Ausführung der Ereignisse in der Regel nur teilweise bekannt ist, führt das Bestreben, solche Verletzungen von vornherein zu vermeiden, oft zu einer starken Einschränkung der potentiell ausnutzbaren Parallelität. Je mehr Garantien (s. u.) aufgrund externen Wissens vorhanden sind, desto weniger muss auf Nullnachrichten gewartet werden. Empirische Messungen, beispielsweise in [Fuj88b] [Fuj88a] [Nico88], bekräftigen dies. Die Abhängigkeit von externem Wissen führt auch dazu, dass aus kleinen Änderungen am Simulationsmodell möglicherweise große Änderungen der Performance resultieren [Fuj88a] [LeCl89]. Dies macht effiziente verteilte Simulationsanwendungen schwerer wartbar.

Der Simulationsalgorithmus entscheidet für jeden LP des Simulators, wann dieser sein nächstes Ereignis ausführt und welche Maßnahmen er zur Erhaltung der Kausalität durchzuführen hat. Die Kausalordnung auf der Menge E aller auszuführenden Ereignisse ist eine transitive, antisymmetrische und antireflexive Relation $\rightarrow$ auf E.

[49] [Mehl94] und [Fuji00] geben einen umfassenden Überblick über Methoden und Algorithmen bei der verteilten Simulation.

[50] Eine Verklemmung ist eine Situation, bei der zwei oder mehr Tasks beteiligt sind und jede Task eine andere derart blockiert, dass das System aus diesem Zustand nicht wieder fortschreiten kann, d. h. während sie selbst noch auf Betriebsmittel warten, bereits Betriebsmittel belegt halten, die der andere Prozess benötigt, um fertig zu werden [Schn97].

Definition: Ein Ereignis e_2 heißt *kausal* von e_1 abhängig (geschrieben: $e_1 \rightarrow e_2$), wenn

(1) der Zeitstempel von e_1 kleiner als der von e_2 ist und

- e_1 eine Zustandsvariable verändert, die von e_2 gelesen wird,
- e_1 eine Zustandsvariable liest, die von e_2 verändert wird, oder
- e_1 eine Zustandsvariable verändert, die auch von e_2 verändert wird, oder

(2) es ein Ereignis e_x gibt derart, dass $e_1 \rightarrow e_x$ und $e_x \rightarrow e_2$ (Transitivität).

Gilt $e_1 \rightarrow e_2$, dann muss e_1 vor e_2 ausgeführt werden. Die Ausführung in umgekehrter Reihenfolge würde i. A. ein anderes Ergebnis liefern.

Sobald ein LP sicher ist, keine Ereignisse mehr mit einer kleineren Eintrittszeit als t ausführen zu müssen, erhöht er seine lokale Uhrzeit auf t. Erreicht die Uhr die Eintrittszeit des nächsten Ereignisses e in seiner Ereignisliste, ist dieses Ereignis garantiert und kann ausgeführt werden. Blockierungen können erkannt und behoben werden, indem der LP_i gefunden wird, der zurzeit das Ereignis e_{min} mit der global kleinsten Eintrittszeit t_{min} in seiner Ereignisliste besitzt. Da ein Ereignis mit Zeitstempel t nur Ereignisse mit Zeitstempel $t' \geq t$ erzeugen kann, gibt es in der gesamten restlichen Simulation keine Ereignisse mehr mit kleinerer Eintrittszeit als t_{min}. Die Uhrzeit von LP_i kann daher auf t_{min} erhöht werden.

Beispiel: Ein Simulator blockiert, wenn nicht garantiert ist, dass nicht an allen Eingängen von J Nachrichten mit Zeitstempeln $> t_j$ vorhanden sind (Bild 60). Das nächste Ereignis in J zum Zeitpunkt t_j kann verarbeitet werden, wenn gilt: $t_j \leq t_{in1} \wedge t_j \leq t_{in2}$ [Zeig00].

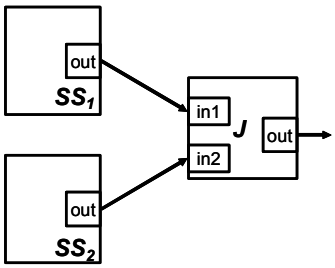


Bild 60: Block mit zwei Eingängen und einem Ausgang

3.6.1.1 Garantien

Zum Erkennen von garantierten Ereignissen ist der Austausch von Garantien notwendig. Garantien lassen sich aus lokalem, globalem oder externem Wissen ableiten. Eine Garantie G von LP_i an LP_j ist die Zusicherung von LP_i an LP_j, dass LP_j während der restlichen Simulation keine Ereignisse mehr von LP_i erhalten wird, deren Zeitstempel kleiner als G sind. Damit das Versenden solcher Garantien mittels Nachrichten sinnvoll ist, wird für jeden Kanal (d. h. jede unidirektionale, direkte Kommunikationsverbindung zwischen zwei LPs) gefordert, dass alle Nachrichten in der Reihenfolge empfangen werden, in der sie gesendet wurden. Außerdem wird angenommen, dass es zwischen zwei LP höchstens einen Kanal in jede Richtung gibt. Durch diese Forderungen wird sichergestellt, dass eine Garantie, die beim Senden noch galt, auch beim Empfang noch gelten wird. Ohne sie könnte eine Garantie G ein bereits früher versendetes Ereignis e mit Zeitstempel $t < G$ auf dem Kanal „überholen“ [PrLo91].

Die lokale Uhrzeit eines konservativen LP lässt sich als eine mögliche Garantie von diesem an alle anderen LP betrachten. Ein konservativer LP setzt seine lokale Uhrzeit auf die virtuelle Zeit t, wenn er sicher ist, keine Ereignisse mit einer kleineren Eintrittszeit als t mehr ausführen zu müssen. Bei Ausführung eines Ereignisses mit Zeitstempel t könnten jedoch allenfalls Ereignisse mit Zeitstempel $t' \geq t$ erzeugt werden. Folglich könnte ein LP prinzipiell immer bei Erhöhung seiner Uhrzeit auf die

virtuelle Zeit t die Garantie $G = t$ an alle anderen LPs senden. Das Ereignis mit dem global kleinsten Zeitstempel kann, genau wie bei einem sequenziellen Simulator, immer sofort ausgeführt werden. Die entsprechende virtuelle Zeit heißt globale virtuelle Zeit (Global Virtual Time, GVT) und ist eine monoton steigende Funktion der Realzeit.

Die GVT zum Realzeitpunkt τ ist definiert durch

$$GVT(\tau) = \min_{LP_i}\{T_{lokal}(\tau), T_{in\_transit}(\tau)\}$$

mit

$T_{lokal} = \begin{cases} Uhr_i(\tau), & \text{wenn } LP_i \text{ führt zur Zeit } \tau \text{ ein Ereignis aus} \\ Next(\tau), & \text{sonst} \end{cases}$

$Uhr_i(\tau)$ = Uhrzeit des logischen Prozesses LP_i zur Zeit τ.

$Next_i(\tau)$ = Minimum von ∞ und den Zeitstempeln aller Ereignisse von LP_i, die zum Zeitpunkt τ bereits erzeugt, aber noch nicht in die Ereignisliste von LP_i eingefügt wurden.

3.6.1.2 Lookahead

Kennt ein LP_i mit virtueller Simulationszeit t alle Ereignisse, die er einem LP_j bis zur virtuellen Zeit $t + L_j$ einplanen wird, so hat LP_i bzgl. LP_j den Lookahead L_j. Kennt LP_i alle Ereignisse, die er beliebigen anderen LP bis zum Zeitpunkt $t + L$ einplanen wird, dann hat LP_i den Lookahead L. Es gilt $L = \min_j L_j$, wobei der Index j über alle LPs des verteilten Simulators läuft. Große Lookaheads wirken sich in konservativen Simulationen günstig auf die Effizienz aus. Der Lookahead kann sich während der Ausführung dynamisch ändern. Er kann zum Beispiel abgeleitet werden aus:

- der Interaktionsgeschwindigkeit zweier PPs;
- der Reaktionsgeschwindigkeit eines LP auf ein Ereignis;
- der Toleranz von Ungenauigkeiten;
- Nichtpräemptivem Verhalten, das den Lookahead erhöht;
- der Vorausberechnung von Simulationsaktivitäten: Wenn die Ereignisse, die ein LP über eine bestimmte Zeit erzeugt, nicht von externen Ereignissen, sondern nur von internen Berechnungen, abhängen, können sie im Voraus berechnet und so der Lookahead verbessert werden.

3.6.1.3 Nullnachrichten

Neben impliziten Garantien können Garantien auch explizit in einer Nachricht versendet oder angefordert werden. Eine explizite Garantie wird auch als Nullnachricht bezeichnet, da eine solche Garantie als implizite Garantie eines Pseudoereignisses angesehen werden kann, das „nichts tut“ (daher „Null“), außer mit seinem Zeitstempel die Garantie weiterzuleiten [ChMi78]. Nullnachrichten dienen ausschließlich der Synchronisation und korrespondieren nicht mit einer Aktivität im physikalischen System. Allgemein gilt, dass eine Nullnachricht mit dem Zeitstempel T_{Null}, die von LP_i nach LP_j gesendet wird, im Wesentlichen ein Versprechen von LP_i ist, dass LP_i dem LP_j keine Nachricht mit einem Zeitstempel kleiner als die untere Grenze T_{Null} sendet.

Beispiel: Nullnachrichten „reisen“ durch das gesamte System. Für die aktuelle virtuelle Zeit $t = 10$ werden keine Ereignisse von den LPs erzeugt (Bild 61). F und J können $(e,14)$ bzw. $(e,13)$ zur Zeit $t = 10$ nicht verarbeiten.

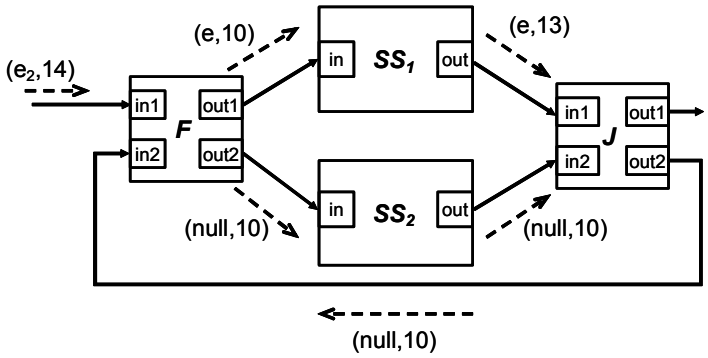


Bild 61: Vermeidung von Blockierungen durch Nullnachrichten

Frühestmögliche Versendung: Explizite Austauschschemata lassen sich weiter danach unterscheiden, wann die Garantien versendet werden. Der höchste Parallelitätsgrad, aber auch der höchste Nachrichtenaufwand, wird erreicht, wenn explizite Garantien möglichst früh versendet werden. Wenn ein Prozess die Verarbeitung eines Ereignisses abgeschlossen hat, sendet er an jeden seiner Ausgänge eine Nullnachricht, die die untere Grenze anzeigt. Der Empfänger der Nullnachricht kann dann neue Grenzen für seine abgehenden Verbindungen berechnen, diese Informationen zu seinen Nachbarn senden usw. Nullnachrichten können z. B. nach der Verarbeitung jedes Ereignisses gesendet werden. Das garantiert, dass die Prozesse hinsichtlich der Zeitstempel zukünftiger Nachrichten, die von anderen Prozessen empfangen werden können, immer auf dem neusten Stand sind. Ankommende Nullnachrichten werden wie andere Nachrichten verarbeitet, d. h. die Uhr des LP wird aktualisiert, aber es wird kein Applikationscode ausgeführt, um die Nachricht zu verarbeiten. In der Regel führt ein solches Verhalten daher zu einem „Überfluten“ des Kommunikationsmediums mit Garantienachrichten.

Versendung bei Blockaden: Das Überfluten des Kommunikationsmediums mit Nachrichten lässt sich abschwächen, indem explizite Garantien nur noch „sporadisch“ versendet werden. Beispielsweise könnte ein LP nur dann seine bestmöglichen Garantien an alle Nachbarn senden, wenn er blockiert, d. h. wenn er selbst keine weiteren garantierten Ereignisse in seiner Ereignisliste besitzt, die er ausführen kann.

Garantien auf Anfrage: Das „blinde“ Versenden bestmöglicher Garantien eines LP, wie es zum Teil bei Blockaden, vor allem aber bei frühestmöglicher Versendung expliziter Garantien vorherrscht, birgt auch das Problem, dass viele der versendeten Garantien überhaupt nicht benötigt werden [WaLa88]. Wartet ein LP z. B. auf eine Garantie $G \geq 1000$ von einem logischen Prozess LP_i, um sein nächstes Ereignis auszuführen, so nutzen ihm von LP_i versendete Garantien 1, 2, 3, ..., 999 nichts, da sie lediglich die Anzahl versendeter Nachrichten erhöhen. Aus diesem Grund wurden Verfahren vorgeschlagen, bei denen jeweils nur die wirklich benötigten Garantien angefordert werden [BaSc88] [Misr86] [SoGu91] [SuSe89]. Dieser Ansatz hilft, die Anzahl der Nullnachrichten zu reduzieren. Allerdings kann eine längere Verzögerung bis zum Empfang der Nullnachrichten auftreten, da zwei Nachrichten-Übertragungen notwendig sind.

3.6.2 Optimistischer Ansatz nach Jefferson-Sowizral

Optimistische Protokolle veranlassen ein zeitliches „Zurückspringen“ – eine Revidierung eines Teils der bereits durchgeführten Simulation – der Simulation Engine, falls eine vorzeitige Abarbeitung lokaler Ereignisse zu Inkonsistenzen bei den Kausalitätsverhältnissen anderer LPs geführt hat. So kann die Chance, dass möglicherweise kein kausales Problem auftreten wird, wirkungsvoll genutzt werden. Allerdings muss ein vorläufiger Kausalitätsverlust in Kauf genommen und sofort nach dem Auftreten behoben werden. Bei der optimistischen Synchronisation können Wartezeiten der Simulatoren reduziert werden und im günstigsten Fall sogar verschwinden, wodurch eine insgesamt geringere Rechenzeit entsteht. Die am häufigsten eingesetzten optimistischen Synchronisationsprotokolle basieren auf dem von Jefferson [Jef85a] und Sowizral [JeSo85] im Jahr 1985 vorgestellten, Time

Warp (TW) genannten Mechanismus, der auch den Begriff der virtuellen Zeit geprägt hat. Die Synchronisation erfolgt, ähnlich wie bei den konservativen CMB-Verfahren, durch das Versenden von globalen Ereignissen, die mit Zeitinformationen versehen sind. TW sieht jedoch eine Rücksetzung in der Simulationszeit vor, um eine die lokale Kausalitätsbedingung erfüllende Simulation zu ermöglichen. Dazu ist das gelegentliche Speichern des Systemzustands (Checkpointing, State Saving) nötig. Eine Voraussetzung zur Anwendung optimistischer Synchronisationsverfahren ist die Unterstützung durch die Simulatoren.

Anstatt zu warten, d. h. nichts zu tun, führt ein optimistischer LP prinzipiell immer sein lokal nächstes Ereignis e aus. Er hofft dabei, dass ihm im restlichen Verlauf der Simulation kein weiteres Ereignis, das er entsprechend der Kausalordnung vor e auszuführen hat, mehr eingeplant wird. Erweist sich diese Hoffnung als unberechtigt, so werden die Effekte verfrüht ausgeführter Ereignisse rückgängig gemacht. Kausalitätsverletzungen sind also erlaubt, werden entdeckt und repariert. Der Aufwand optimistischer Verfahren liegt daher im Verwalten von Rücksetzinformationen und im Durchführen von Rücksetzungen (Rollbacks).

Beispiel: In Bild 62 ist die Simulationszeit bis zu VT=9 fortgeschritten. Zur Zeit VT=4 tritt ein Straggler-Ereignis (Nachzügler-Ereignis) auf, das in Bild 63 dargestellt ist. Aufgrund dieses Ereignisses wird eine Rücksetzung zurück zur Zeit VT=4 durchgeführt, falsche Ausgaben werden rückgängig gemacht und Eingangsereignisse erneut berechnet. Der Zustand nach der Durchführung der Rücksetzung ist in Bild 64 dargestellt.

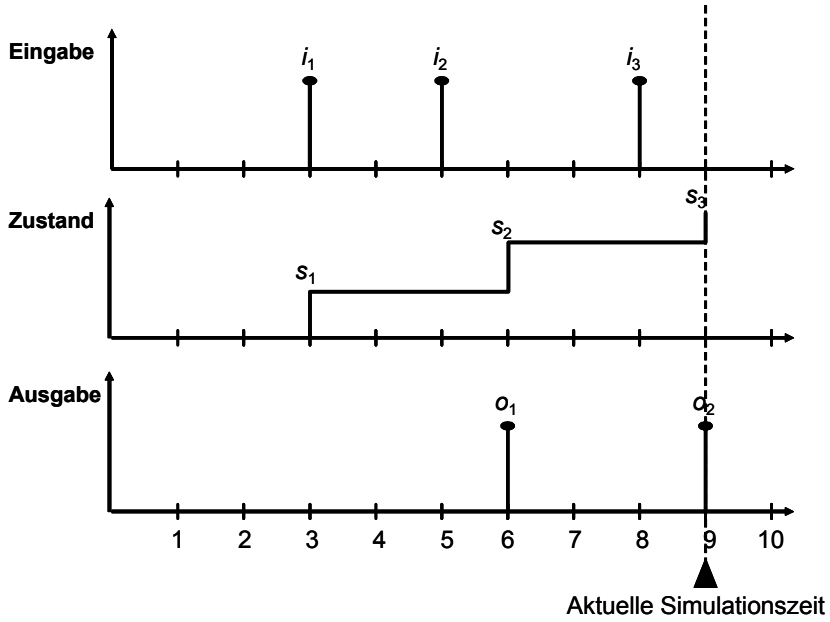


Bild 62: TW – Zustand vor Straggler

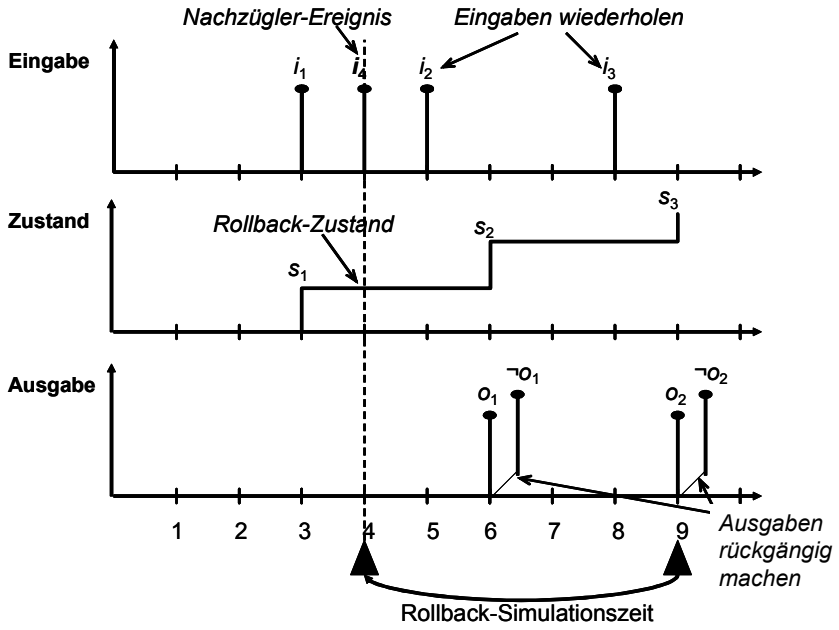


Bild 63: TW – Zustand nach Straggler

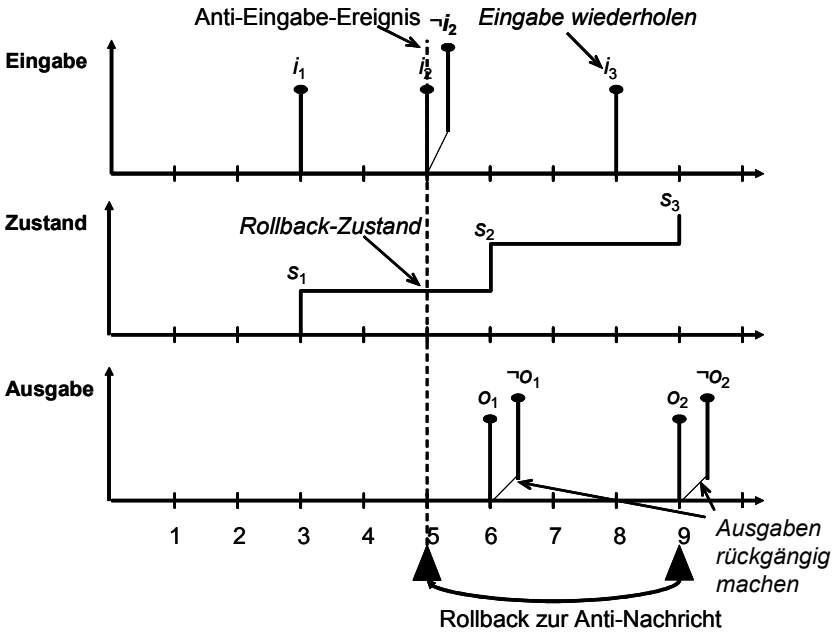


Bild 64: TW – Zustand nach Rollback

Empirische Messungen mit TW zeigen in der Regel eine gute Performance. Beispielsweise wurden auf Mark-III-Hypercube-Rechnern ein Speedup von 16 bei 32 Prozessoren für eine Rechnernetzsimulation [PrEb89], ein Speedup von 28 bei 60 Prozessoren für eine Gefechtsfeldsimulation [WiHa89] oder ein Speedup von 12 bei 32 Prozessoren für eine vereinfachte Verhaltensstudie aus dem Bereich der Biologie [EbDi89] erreicht. TW kann also für realistische Modelle eine beträchtliche Beschleunigung ermöglichen, auch wenn diese für die Anzahl der Prozessoren vergleichsweise gering ausfällt. Der maximal erreichbare Speedup ist letztendlich durch die im Modell vorhandene Parallelität beschränkt.

In vielen Messungen wurde TW mit konservativen Verfahren verglichen. Untersuchungen auf einem Transputer-basierten Mehrprozessorsystem deuten an, dass TW für die meisten Modelle den konservativen Verfahren überlegen ist [Prei90]. In den Modellen, in denen jedoch viele externe Garantien gegeben werden können, zeigten sich konservative Verfahren TW überlegen. Diese Ergebnisse wur-

den ebenfalls für parallele Simulation bestätigt [Fuji89]. Auch können konservative Verfahren besonders dann besser als optimistische sein, wenn das Modell einen großen Zustandsraum besitzt. Das häufige Anlegen von Zustandskopien kann die Performance optimistischer Simulation erheblich reduzieren.

3.6.2.1 Zurücksetzen von Zustandsvariablen (Rollback)

Ein bedeutender Teil der Rechenzeit trägt nicht zum eigentlichen Fortschreiten der Simulation bei, sondern wird für Rücksetzungen in Anspruch genommen. Beim Eintreffen eines globalen Ereignisses mit einem Zeitstempel in der lokalen virtuellen Vergangenheit (Straggler-Ereignis) rollt die Kommunikationsschnittstelle die Simulation Engine zurück. Dies geschieht durch Wiederherstellen des letzten gesicherten Systemzustands vor dem Zeitstempel des Stragglers. Anschließend beginnt die Simulation erneut. Durch dieses Vorgehen kann der begangene Kausalitätsfehler behoben und ein Kausalitätsverlust vermieden werden. Um jedoch bereits erzeugte Folgeereignisse, die an andere Prozesse geschickt wurden, wieder rückgängig zu machen, müssen geeignete Maßnahmen ergriffen werden, die häufig die Performance stark beeinflussen. Zur Verringerung des Aufwands haben viele Forschungsansätze wie [Fers95] und [NoCh95] eine Reduktion der Rücksetzungshäufigkeit zum Ziel. Die Rücksetzungsmethode ist ein rein optimistisches Verfahren für die verteilte Simulation diskreter Ereignisse auf der Grundlage der Modellbeschreibung des physikalischen und des logischen Systems. Vermeidung und Behandlung von Rücksetzungen und Blockierungen sind seit 1988 ein wichtiger PDES-Forschungsschwerpunkt. Es existieren verschiedene Verfahren der Optimismusbegrenzung zur Vermeidung von Rücksetzungen durch eine einheitliche Progressionsrate der LVT eines Simulators (von *n* Simulatoren). Erreicht wird dies z. B. durch Zeitfenster, lokale Rücksetzungen, Prädiktoren [ScTM97] usw.

Die optimistische Annahme, jederzeit über alle notwendigen Informationen zu verfügen, kann einen LP dazu veranlassen, eine falsche Ereignisnachricht *(t, m)* zu versenden. Eine Ereignisnachricht *(t, m)* von LP_i nach LP_j ist falsch, wenn im physikalischen System keine äquivalente Nachricht m zur Zeit *t* von PP_i nach PP_j gesendet wird. Umgekehrt ist eine Ereignisnachricht *(t, m)* von LP_i nach LP_j korrekt, wenn im physikalischen System eine äquivalente Nachricht *m* zur Zeit *t* von PP_i nach PP_j gesendet wird. Um Abweichungen des logischen Systems vom physikalischen System durch falsche Ereignisnachrichten *(t, m)* zu beseitigen, wurde bei der Rücksetzungsmethode als neuer Nachrichtentyp die Negativnachricht *('-', t, m)* (anti message) eingeführt. Sie ist die Komplementärnachricht zu einer falsch gesendeten Ereignisnachricht *(t, m)* und hat die Aufgabe, diese zu eliminieren. Das Rücksetzen eines LP kann Auswirkungen auf andere LPs haben. Deshalb muss ein LP, der zurückgesetzt wird, zu allen in dem durch die Rücksetzung verlorenen Zeitintervall gesendeten Nachrichten *(t, m)* Negativnachrichten *('-', t, m)* senden. Eine einmal ausgelöste Rücksetzung kann sich baumartig über alle LPs ausbreiten. In Abhängigkeit von der Topologie der LPs ist dieser „Baum“ evtl. in Wirklichkeit ein zyklischer Graph; Eine von einem LP ausgelöste Rücksetzung kann dazu führen, dass dieser LP selbst wieder Negativnachrichten empfängt. Da die Zeitstempel der von einem LP gesendeten Negativnachrichten immer größer oder gleich sind als der Zeitstempel der Nachricht, der die Rücksetzung des LP verursacht hat, können diese Negativnachrichten jedoch nicht zu einer weiteren Rücksetzung des LP führen. Es gibt also keinen Domino-Effekt, bei dem das gesamte Simulationssystem immer weiter in die Vergangenheit zurückgesetzt wird. Im schlimmsten Fall werden alle LPs auf die Simulationszeit zurückgesetzt, in der sich der auslösende Prozess befindet.

Zur Realisierung des Rücksetzungsmechanismus in jedem LP sind eine Reihe von Verwaltungsstrukturen notwendig:

- Eingangswarteschlange (input queue, IQ): In ihr werden alle ankommenden Ereignisnachrichten in der Reihenfolge ihrer Ankunftszeiten einsortiert. Sie ist die lokale Ereignisliste des LP.
- Ausgangswarteschlange (output queue, OQ): Sie enthält die Negativnachrichten zu den von diesem LP schon gesendeten positiven Ereignisnachrichten. Die Warteschlange ist gemäß steigenden Abgangszeiten der Nachrichten geordnet.

- Zustandswarteschlange: Darin wird der Prozesszustand von Zeit zu Zeit gesichert, um eine Rücksetzung zu ermöglichen.

Jede positive Ereignisnachricht wird vom empfangenden LP ihrer Ankunftszeit entsprechend in die Ereignisliste eingefügt. Ein Fehler liegt vor, wenn die Ankunftszeit der zuletzt empfangenen '+'-Nachricht kleiner ist, als die Ankunftszeit der zuletzt gesendeten '+'-Nachricht.

Jeder LP hat seine eigene Eingangswarteschlange, die die lokale Ereignisliste dieses Prozesses repräsentiert, und seine eigene LVT. Jede an einem LP ankommende Nachricht *(t, m)* wird gemäß ihrer Zeit *t* in dieser Warteschlange gespeichert. Die LVT eines LP ist gleich der Zeit *t* der zuletzt von ihm gesendeten Nachricht *(t, m)*. Während der Simulation speichert jeder LP von Zeit zu Zeit seinen Zustand, bestehend aus der LVT und allen Variablen, die seinen Zustand zur Zeit LVT beschreiben, in einer Zustandswarteschlange. Diese ist gemäß ansteigenden LVTs geordnet. Der Zustand wird jeweils nach dem Einfügen von $k >= 1$ neuen Ereignissen in die lokale Ereignisliste gesichert (state saving, checkpointing, s. u.). Je kleiner dabei *k* ist, desto schneller kann eine Rücksetzung durchgeführt werden, aber desto größer ist auch der Speicherverbrauch. Die Festlegung von *k* stellt folglich einen Kompromiss zwischen Rechenzeitaufwand und Speicherbedarf dar. Im Fehlerfall wird der LP auf die letzte Zustandssicherung vor Eintritt des Fehlers zurückgesetzt. In der Ausgangswarteschlange speichert jeder LP Negativnachrichten *('-', t, m)* zu den Nachrichten *(t, m)*, die er schon gesendet hat. Die Ausgabewarteschlange ist gemäß ansteigenden Zeiten *t* der bereits gesendeten Nachrichten *(t, m)* geordnet. Jeder LP bearbeitet die Nachrichten seiner Ereignisliste in zeitlich aufsteigender Reihenfolge solange, bis eine Nachricht mit einer Zeit eintrifft, die in der Simulationsvergangenheit des Prozesses liegt (Rücksetzungszeitpunkt). Tritt ein solcher Fehler auf, muss der LP auf einen Zustand zurückgesetzt werden, der vor dem Rücksetzungszeitpunkt bestand.

Die Rücksetzung eines LP durch eine Nachricht, die in seiner Simulationsvergangenheit liegt, gliedert sich in drei Phasen: Restauration, Korrektur und Wiederholung. Die Restaurationsphase beginnt mit dem Einfügen der Nachricht aus der Simulationsvergangenheit in die Eingangswarteschlange. Ist die Nachricht eine Negativnachricht zu einer Nachricht in der Eingangswarteschlange, so führt das Einfügen zum gegenseitigen Auslöschen der Nachrichten. Danach werden der Zustand und die LVT des LP aus der letzten Sicherung (in der Zustandswarteschlange) vor der Rücksetzung auslösenden Nachricht rekonstruiert (Restaurationszeitpunkt). Auf den zurückgesetzten Zustand nachfolgende Einträge in der Zustandswarteschlange müssen ggf. gelöscht werden. In der Korrekturphase werden alle Auswirkungen, die in anderen LPs durch falsches Versenden von Nachrichten aufgetreten sind, rückgängig gemacht (s. u.). Dazu versendet der LP alle Negativnachrichten in der Ausgangswarteschlange mit einer Zeit größer als die der Nachricht aus der Simulationsvergangenheit, und entfernt die gesendeten Nachrichten aus der Ausgangswarteschlange. Die Aufrechterhaltung der Nachrichtenreihenfolge des Kommunikationssystems vorausgesetzt, gibt es zwei Möglichkeiten, wie sich der Empfang einer Negativnachricht bei LP_j auswirken kann:

- Die originale (positive) Ereignisnachricht wurde noch nicht bearbeitet und befindet sich in der Ereignisliste von LP_j. Das Einfügen der empfangenen Negativnachricht führt zum gegenseitigen Auslöschen der Nachrichten. In LP_j hinterlassen die Nachrichten keine Spuren. Es ist so, als hätte LP_j die Nachrichten nie empfangen.
- Die originale (positive) Ereignisnachricht wurde bereits bearbeitet. Auch hier führt der Empfang der Negativnachricht zum gegenseitigen Auslöschen der komplementären Nachrichten, nur hat die Bearbeitung der originalen Nachricht ihrerseits schon Auswirkungen auf den Zustand des LP_j und auf andere LPs. Die Folge ist eine sekundäre Rücksetzung von LP_j. Dies führt zum Aussenden von Negativnachrichten durch LP_j, die aber spätere Zeitstempel als die initiale Negativnachricht haben, so dass sich dieser Rücksetzungsprozess nicht endlos fortsetzen kann.

Durch eine Zustandssicherung nach jeweils $k > 1$ Ereignissen kann ein LP in der Restaurationsphase weiter in die Vergangenheit zurückversetzt werden, als dies nötig gewesen wäre. In diesem Fall muss

der LP in der Wiederholungsphase, vom restaurierten Zustand ausgehend, durch die Bearbeitung der Nachrichten der Eingangswarteschlange den letzten Zustand vor dem Rücksetzungszeitpunkt nochmals berechnen. Das Aussenden von Nachrichten mit Zeiten zwischen dem Restaurationszeitpunkt und dem Rücksetzungszeitpunkt wird in dieser Phase unterdrückt, da diese Nachrichten in den Warteschlangen des Absenders und des Empfängers schon vorhanden sind.

3.6.2.2 Zustandssicherung

Um Rücksetzungen zu bestimmten vergangenen Systemzuständen zu ermöglichen, müssen diese entweder vollständig (copy state saving) oder nur inkrementell (Änderungen des Systemzustands) (incremental state saving) gespeichert werden. Die zur Durchführung von Rücksetzungen benötigte Protokollierung des Simulationsverlaufs durch Zustandssicherung nimmt sowohl Rechenzeit als auch Speicherkapazität in Anspruch. Bei inkrementeller Speicherung erhöht sich die Ausführungskomplexität bei Rücksetzungen, da die zu restaurierenden Systemzustände durch Verfolgung der einzelnen Inkremente entlang eines Pfades gebildet werden müssen, der bis vor den Zeitstempel des Stragglers zurückreicht. Es wird also mehr Speicher für eine vollständige Speicherung benötigt; die Zustände können aber schneller wiederhergestellt werden (siehe Bild 65). Eine inkrementelle Zustandsspeicherung ermöglicht hingegen eine schnellere Speicherung der Zustände aber bedeutet eine langsamere Wiederherstellung (siehe Bild 66).

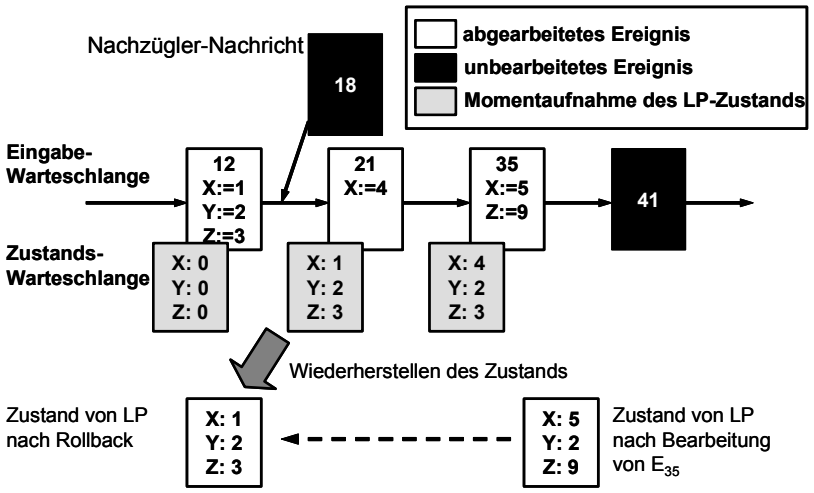


Bild 65: Rollback bei vollständiger Zustandsspeicherung

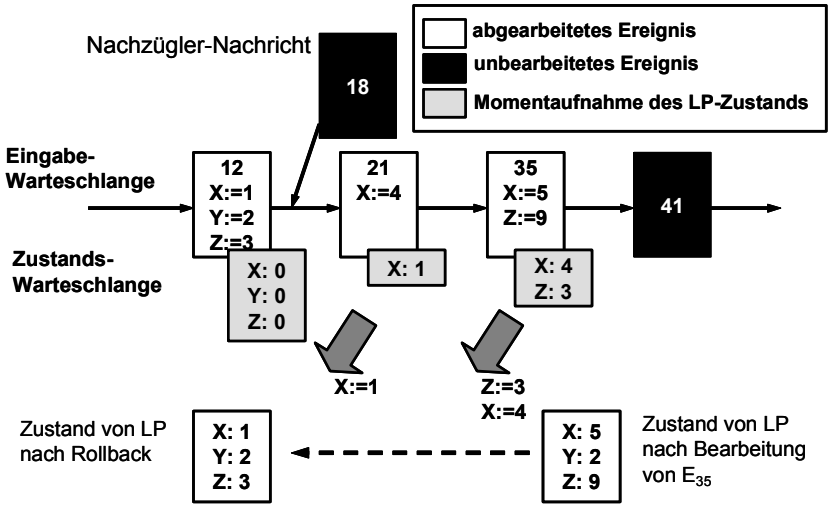


Bild 66: Rollback bei inkrementeller Zustandsspeicherung

In der Literatur findet man, dass eine inkrementelle Zustandsspeicherung im Durchschnitt 20 % schneller ist als eine vollständige Zustandsspeicherung. Eine Kopie des kompletten lokalen Zustan-

des ist dann aufwändig, wenn der lokale Zustandsraum mehrere Megabytes groß ist oder etwa Zeiger auf dynamisch angelegte Speicherbereiche enthält [JeBe87]. So fand Fujimoto bei einer Messung an einem Warteschlangennetz-Simulator heraus, dass die Performance von TW halbiert wird, wenn der Zustandsraum von ca. 100 Byte auf 2 KByte vergrößert wird [Fuji89].

Zahlreiche Verfahren protokollieren nicht jeden einzelnen Berechnungsschritt und vergrößern so das Checkpointing-Intervall (interleaved state saving), um Zeit und Speicher zu sparen. Die Folge ist jedoch, dass bei der Restaurierung eines früheren Systemzustands nicht immer eine entsprechende Kopie im Zustandsspeicher vorhanden ist, sondern oft zunächst ein davor liegender Zustand wiederhergestellt werden muss. Dieser Zustand muss dann durch einen Coast-forward – der bereits durchgeführte Berechnungen reproduziert und somit eine vermeidbare Inanspruchnahme von Rechenkapazität darstellt – in den benötigten Zustand überführt werden. Auf der anderen Seite konnten bis dahin u. U. bereits erhebliche Einsparungen an Zeit und Speicher erzielt werden, so dass sich ein gelegentliches Coast-forward weniger stark auswirkt. In [PaWi93] wird diese Methode „weniger Zustandskopien“ mit der Methode inkrementeller Zustandssicherungen analytisch verglichen. Dabei wurde festgestellt, dass keines der Verfahren immer überlegen ist.

In [LiPr93] wird eine adaptive Variante periodischer Zustandssicherungen vorgestellt, in der zyklisch eine gewisse Anzahl von Ereignissen ausgeführt und anschließend jeweils ein neuer Wert ermittelt wird. Solche adaptiven Verfahren scheinen sinnvoll zu sein, da die Qualität eines Verfahrens typischerweise stark vom Modellverhalten abhängt.

Für die Durchführung von optimistischer Simulation setzen CASE-Werkzeuge die Möglichkeit einer Zustandsspeicherung und Rücksetzung voraus. Bei den meisten auf dem Markt befindlichen CASE-Werkzeugen haben die Hersteller keine Möglichkeiten für eine Zustandsspeicherung des kompletten Systemzustandes oder eine Rücksetzung auf einen bestimmten Zeitpunkt vorgesehen[51].

3.6.2.3 Fehlerbehandlung (Ereignis-Annullierung)

Zur Annullierung (cancellation) von bereits gesendeten Ereignissen werden Negativnachrichten gesendet. Noch nicht verarbeitete Ereignisse in der Eingangs-Warteschlange werden annulliert (Bild 67). In Abhängigkeit davon, wann Negativnachrichten versendet werden, lassen sich zwei Strategien unterscheiden: frühe Annullierung (aggressive cancellation) und späte Annullierung (lazy cancellation) von Ereignissen [Gafn85]. Bei der frühen Annullierung werden Negativnachrichten sofort versendet, d. h. die Rücksetzung startet unmittelbar nach Empfang des Stragglers.

Bei der späten Annullierung werden Negativnachrichten nur versendet, wenn die Ausgaben nicht identisch zu den alten Ausgaben sind. Im Vergleich zur frühen Annullierung verringert dies die Rücksetzungshäufigkeit, weil ein Straggler oft nur einige oder keine der zurückgerollten Ereignisse berührt und meist zumindest ein Teil der unter irrtümlichen Voraussetzungen erzeugten Nachrichten trotzdem gültig ist. Dieses Verhalten ergibt sich dadurch, dass nicht sofort nach Ankunft eines Stragglers eine Negativnachricht für eine in der Ausgabe-Warteschlange enthaltene positive Nachricht verschickt wird. Stattdessen überprüft das Protokoll während der an die Rücksetzung anschließenden Re-Simulation, ob bei Erreichen des Zeitstempels der möglicherweise fehlerhaften Nachricht erneut eine identische Nachricht erzeugt wird. In diesem Fall kann die bis dahin verzögerte Versendung einer Negativnachricht fallengelassen werden, andernfalls wird wie üblich verfahren. Die Folge ist, dass Nachrichten erst dann widerrufen werden, wenn sie sich als fehlerhaft erwiesen haben. Auf diese Weise vermeidet späte Annullierung die überflüssige Annullierung korrekter Nachrichten, verursacht jedoch zusätzlichen Speicherbedarf und Verwaltungsaufwand und zögert die Aufhebung tatsächlich fehlerhafter Nachrichten hinaus. Im Zusammenhang mit später Annullierung kann TW einen

[51] Z. B. schlägt der Versuch einer optimistischen Simulation mit MATLAB fehl. Die Systemvariablen lassen sich zwar bei der Simulation von $t = t_0$ bis $t = t_2$ mit Hilfe eines Skriptes in einer Matrix zeitabhängig festhalten, jedoch ergibt eine erneute Simulation ab dem Zeitpunkt $t = t_1$, $t_0 < t_1 < t_2$, bei variabler Schrittweite, die für eine asynchrone Ausführung notwendig ist, nicht reproduzierbare Ergebnisse.

so genannten superkritischen Speedup [JeRe91] hervorrufen, d. h. eine Umgehung und Abkürzung des kritischen Pfades der Simulation, falls ungültige Berechnungen dennoch korrekte Ergebnisse liefern. Bei später Annullierung kann dies wegen der sofortigen Revidierung aller Berechnungen nach Rücksetzungen nicht ausgenutzt werden. Es besteht eine starke Abhängigkeit zwischen der erzielbaren Performance und der Art des simulierten Modells, so dass sich keine grundsätzliche Entscheidung zugunsten eines der beiden Verfahren treffen lässt.

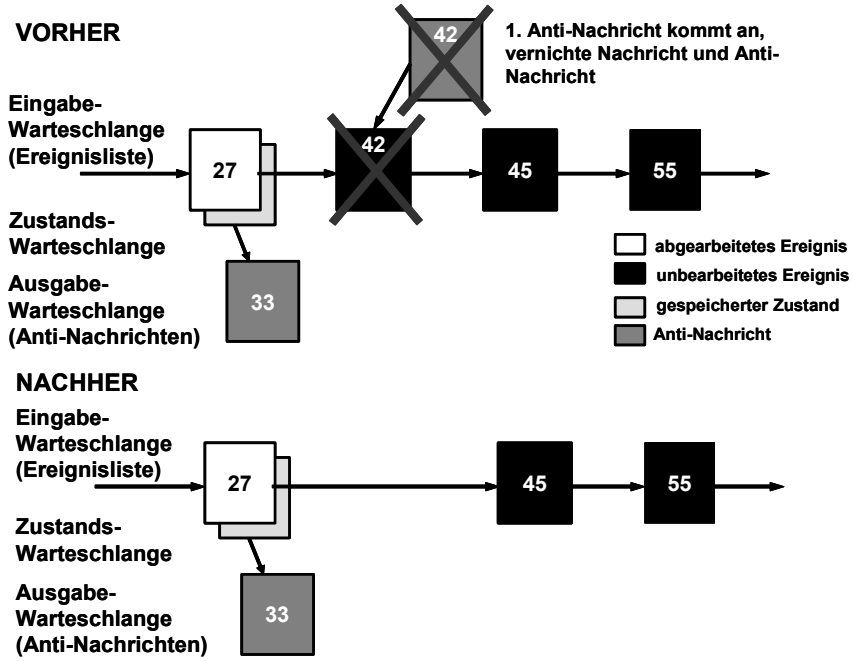


Bild 67: Annullierung einer Nachricht

Es lassen sich Modelle konstruieren, bei denen frühe Annullierung der späten Annullierung deutlich überlegen ist, und umgekehrt. Mit der späten Annullierung ist es theoretisch sogar möglich, eine Simulation in kürzerer Zeit korrekt zu beenden, als es nötig wäre, um alle Ereignisse auf dem kritischen Pfad [BeJe85] auszuführen. Die Ursache hierfür liegt darin, dass falsche Ereignisse möglicherweise korrekte Folgeereignisse erzeugen können, die bei später Annullierung nicht annulliert werden. Aus diesem Grund können Ereignisse hier bereits früher als bei der schnellstmöglichen verteilten Ausführung, die zu keinem Zeitpunkt die Kausalität verletzt, erzeugt werden. Dieser Effekt wird auch superkritischer Speedup genannt [JeRe91]. In [JeRe91] wird gezeigt, dass neben später Annullierung z. B. auch das Jump-forward-Schema (siehe unten) prinzipiell in der Lage ist, superkritischen Speedup zu erzeugen. In realistischen Modellen wird superkritischer Speedup allerdings so gut wie nie erreicht [Lin92].

Während späte Annullierung die Annullierung externer Auswirkungen nach Ankunft eines Stragglers verzögert, versucht späte Neuevaluierung (layz re-evaluation) die Revidierung von Einträgen im Zustandsspeicher zu vermeiden [Fuji90]. Wird nach erfolgter Rücksetzung während der Neuberechnung ein Zustand erreicht, der einem gespeicherten vollständig entspricht und auch eine identische zugehörige Eingangswarteschlange aufweist, kann sofort zur vor der Rücksetzung gültigen LVT vorgesprungen werden, ohne die verbliebenen Berechnungen korrekter Systemzustände erneut durchführen zu müssen (jump forward). Dies ist insbesondere dann erfolgreich, wenn häufig Ereignisse eintreten, die den Systemzustand nicht beeinflussen. Allerdings müssen ein erhöhter Speicherbedarf und eine zunehmende Komplexität des Algorithmus in Kauf genommen werden, u. a. weil bei jeder Zustandsspeicherung eine Kopie der Eingangswarteschlange enthalten sein muss, um später die Gleichheit der Warteschlangen nachweisen zu können.

3.6.2.4 Berechnung der globalen virtuellen Zeit

Neben der LVT wird bei TW eine globale virtuelle Zeit verwendet, die eine untere Grenze aller LVT und der Zeitstempel aller noch nicht abgearbeiteten Ereignisse darstellt. Damit stellt sie den spätesten Zeitpunkt dar, vor den mit Sicherheit kein Prozess zurückrollt. Die GVT hat die Bedeutung einer „gemeinsamen“ Simulationszeit und ist ein Maß für den gesamten Simulationsfortschritt. Mit Speicherverwaltung wird die Ermittlung und Freigabe von später definitiv nicht mehr benötigtem Speicher während der Simulation bezeichnet. Dies spielt insbesondere in Umgebungen mit begrenztem Speicher eine ausschlaggebende Rolle, da TW im Vergleich zu CMB-basierten Protokollen aufgrund der benötigten Zustandsspeicherung meist einen deutlich höheren Speicherbedarf aufweist. Während der Speicherverwaltung (fossil collection) dürfen alle vor der GVT liegenden im SS gespeicherten Zustände und in der Ausgabewarteschlange enthaltenen globalen Ereignisse verworfen werden und der von ihnen beanspruchte Speicher freigegeben werden, da keine Rücksetzung zu einem Zeitpunkt vor der GVT notwendig werden kann. Die Bestimmung der GVT ist notwendig zur Optimierung der Speicherverwaltung und zur Steuerung der Ausgabe und des Simulationsablaufs.

Die globale Simulationszeit zum realen Zeitpunkt *r* ist definiert als das Minimum über allen lokalen Simulationszeiten aller LPs zum Zeitpunkt *r* und über allen Zeitstempeln aller vor dem Zeitpunkt *r* gesendeten aber noch nicht empfangenen Nachrichten. Da die Simulation bei der Ermittlung einer neuen GVT weiterlaufen soll, ist es nur möglich, eine untere Schranke für diesen Wert zu finden. Die untere Schranke muss in bestimmten Abständen durch einen verteilten Algorithmus ermittelt und allen LPs mitgeteilt werden, damit diese aus ihren Warteschlangen alle Informationen entfernen, die älter sind als die untere Schranke der GVT. Durch diesen Mechanismus wird der Speicherbedarf des Simulationssystems reguliert. Entsprechend der optimistischen Simulationsstrategie kann es zu keiner Verklemmung auf der Simulationsebene kommen, weil kein LP auf eine bestimmte Nachricht eines anderen LP wartet. Gerichtete Zyklen im Graphen der LPs und ihren Kanälen sind somit nicht möglich [Kepp94].

Kann auch die Speicherverwaltung nicht genügend Speicher zur Fortführung der Simulation verfügbar machen, lassen sich bereits erhaltene Nachrichten zurückschicken (Message Sendback, Cancelback) oder künstliche Rücksetzungen (Artificial rollback) erzeugen, um Eingangs- oder Ausgangswarteschlange teilweise zu entleeren und so deren Speicherbedarf zu reduzieren. Entsprechend der Speicherverwaltung darf eine als sicher geltende Ausgabe von Simulationsergebnissen erst dann durchgeführt werden, wenn die GVT den zugehörigen Simulationszeitpunkt überschritten hat, weil nur in diesem Fall die Gültigkeit der Ausgabe garantiert ist. Ereignisse, deren Zeitstempel vor der GVT liegen und deren Revidierung daher weder möglich noch notwendig ist, werden als ausgeführte Ereignisse bezeichnet. Demgegenüber stehen die bearbeiteten Ereignisse, die möglicherweise durch eine Rücksetzung wieder annulliert werden müssen. Des Weiteren wird die GVT zur Bestimmung des Simulationsendes verwendet. Das monotone Ansteigen der GVT mit der Echtzeit garantiert, dass TW die Simulation trotz lokaler Rücksetzungen vorantreibt und letztendlich zu Ende führt. Da die exakte Bestimmung der GVT aufwändig[52] und nicht ohne Unterbrechung des Simulationsablaufs möglich ist, wird in der Regel ein Schätzwert als Näherung eingesetzt. Dieser ist kleiner oder gleich der exakten GVT und stellt somit eine Unterschätzung dar, weil nur dadurch verhindert werden kann, dass später noch benötigte Zustandsspeicherungen fälschlicherweise verworfen werden. Um den Speicherbedarf zu minimieren, ist eine möglichst genaue Schätzung anzustreben, d. h. ein nur geringer Unterschied zwischen tatsächlicher und geschätzter globaler virtueller Zeit.

Die Ermittlung der GVT erfolgt dezentral innerhalb der einzelnen LP oder aber zentral durch einen GVT-Manager (dieser heißt in der vorliegenden Arbeit JSimControl, siehe 4.3.5). Da die Bestimmung Rechen- und Kommunikationskapazität beansprucht, ist es in der Regel – solange ausreichende Speicherressourcen zur Verfügung stehen – nicht ratsam, ständig die aktuellste GVT zu er-

[52] Das Kernproblem hierbei sind die noch im Kommunikationssystem befindlichen Nachrichten, deren Zeitstempel sich nicht ohne weiteres bestimmen lassen und die möglicherweise eine Rücksetzung auslösen werden.

mitteln, weil dies, über eine effizientere Speicherausnutzung, nur indirekte Auswirkungen auf die Performance hat. Wird die GVT zu selten berechnet, werden z. B. nicht mehr benötigte Ereignisse und Zustandskopien entsprechend spät durch die Speicherverwaltung erkannt und der Speicher lange nicht freigegeben. Wird andererseits die GVT (genauer eine Approximation der GVT) zu häufig berechnet, steht der Nutzen in keinem Verhältnis zu dem Aufwand für die GVT-Berechnung. In den in der Literatur beschriebenen Verfahren variiert der gewählte Abstand zwischen zwei aufeinander folgenden GVT-Berechnungen stark; Zeitabstände im Mikrosekundenbereich [Reyn91] [RePa93] bis hin zum Sekundenbereich [HoBe89] werden genannt. Optimal wäre es, wenn sich die Berechnung adaptiv an das Modellverhalten anpassen würde. Z. B. ist SPEEDES (Synchronous Parallel Environment for Emulation and Discrete Event Simulation, siehe 3.6.4) ein Verfahren, das adaptiv den Zeitpunkt für die nächste GVT-Berechnung bestimmt [Stei91]. Der auf dem Konzept von Jefferson und Sowizral [JeSo83] basierende Algorithmus wurde u. a. von Samadi [Sama85] in einer verbesserten Version veröffentlicht.

3.6.2.5 Performance-Risiken bei Time Warp

Das Rücksetzungsverhalten ist abhängig von der Partitionierung des Modells. Eine Rücksetzung muss für den gesamten LP durchgeführt werden (siehe Bild 68).

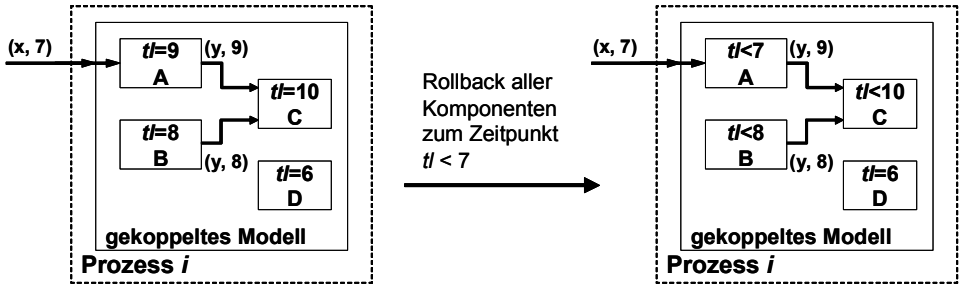


Bild 68: Rollback aller Zustände bei einem LP

Sind Modelle in mehrere Prozesse partitioniert, müssen nicht notwendigerweise alle Modellkomponenten von der Rücksetzung betroffen sein (siehe Bild 69).

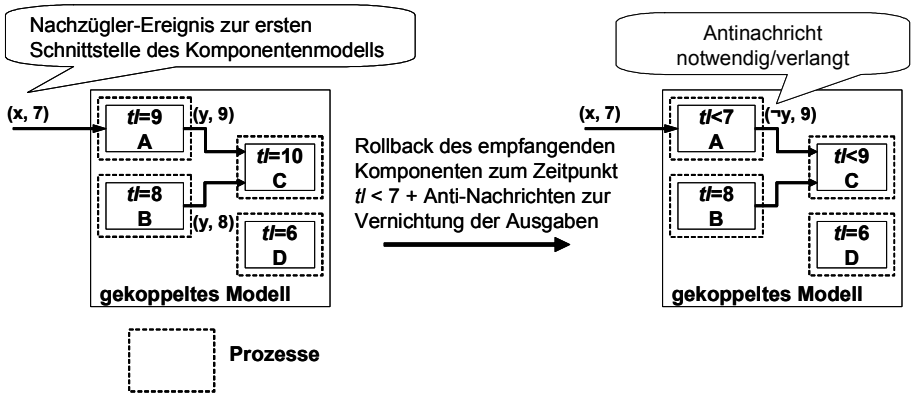


Bild 69: Rollback relevanter Zustände bei mehreren LP

In TW werden die durch verfrühte Ereignisausführungen erzeugten Ereignisse durch das Versenden von Negativnachrichten zurückgezogen. Tatsächlich kann es vorkommen, dass Negativnachrichten zugehörige normale Ereignisse erst dann neutralisieren, wenn diese bereits ausgeführt wurden und dabei neue Ereignisse erzeugt haben. Es kommt dann zu sog. Secondary Rollbacks (Bild 70).

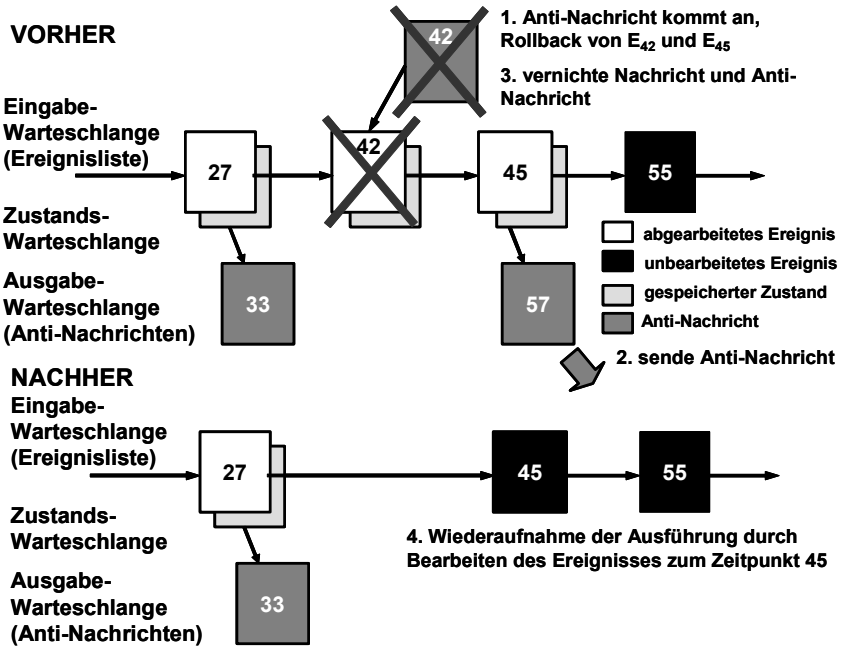


Bild 70: Annullierung von Nachricht und Negativnachricht

Löst jede Negativnachricht in diesem Fall eine Rücksetzung im empfangenden LP aus, so entstehen ganze Kaskaden von Rücksetzungen. Bild 71 illustriert diesen „Dog-chasing-its-tail“ genannten Effekt [Abra88] [Fuji90]. Dabei kommt es nicht zu einem „Domino-Effekt“, da sich Rücksetzungen relativ zur GVT in die Zukunft ausbreiten.

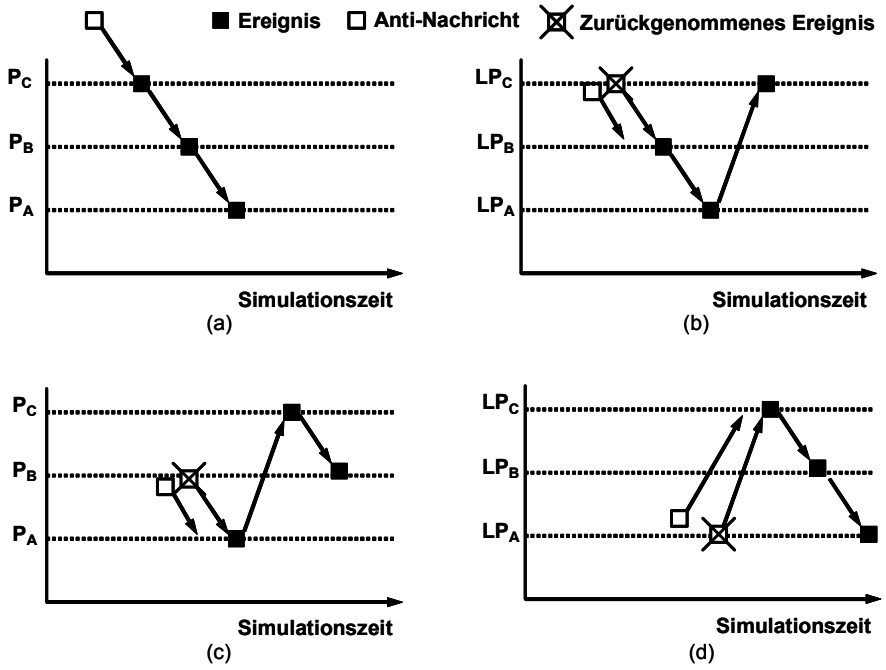


Bild 71: Rücksetzungswellen

Es wurden viele Ansätze vorgeschlagen, um diesen Effekt zu vermeiden und die Ausbreitung potentiell falscher Folgeereignisse einzudämmen, da sie und ihre Rückgängigmachung die Performance stark hemmt.

Bei zu geringer Verarbeitungsgeschwindigkeit der Rücksetzung können ferner sog. Rollback-Echos auftreten (siehe Bild 72).

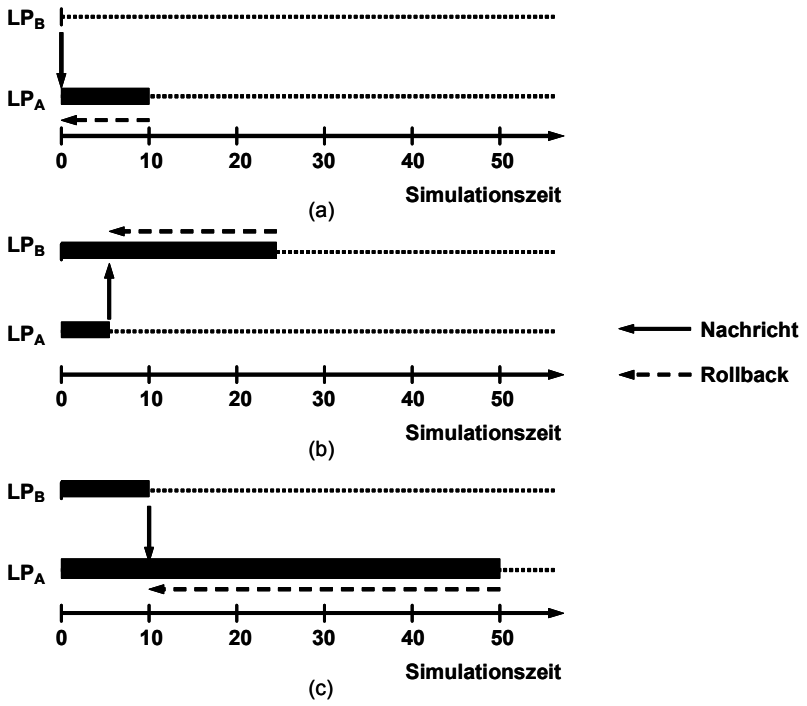


Bild 72: Rollback-Echos

In [MaWa88] und [MaWa89] wird vorgeschlagen, bei der Auslösung einer Rücksetzung die Ausführung aller LPs anzuhalten, die ein zurückzusetzendes Folgeereignis erhalten haben könnten. Anschließend neutralisieren Negativnachrichten die zugehörigen normalen Ereignisse, und die Simulation kann fortgesetzt werden. Dieses Verfahren wird Wolf-Algorithmus genannt. Einerseits begrenzt es das Auftreten von Rücksetzungskaskaden, da das Einfrieren endlich vieler LPs in endlicher Zeit erfolgt, Rücksetzungskaskaden sich jedoch prinzipiell endlos fortsetzen können. Andererseits reduziert es die Performance durch das typischerweise häufige Einfrieren vieler (oft sogar aller) LPs.

3.6.2.6 Verfahren zur Optimismusbeschränkung

Der durch Ausführen von Rücksetzungen aufgrund eines überoptimistischen Fortschreitens der Simulation verursachte Kommunikations-Overhead von TW ist der dominierende Faktor der Simulationsperformance. Auf TW aufbauende Protokolle garantieren zwar selbst bei Rücksetzungskaskaden großer Länge oder großer rekursiver Tiefe ein Fortschreiten der Simulation, jedoch verursacht der durch solche Effekte entstehende Overhead einen großen Performance-Verlust, der sich im Wesentlichen auf zwei Weisen verringern lässt. Es wurden mehrere Verfahren vorgeschlagen, die die Wahrscheinlichkeit für Rücksetzungskaskaden durch Senkung des Optimismusgrades bzw. durch Angleichung der Progressionsraten der LVT_i zu reduzieren versuchen. Zum einen wird versucht, Rücksetzungsketten möglichst früh zu durchbrechen und so die Auswirkungen eines Stragglers in Grenzen zu halten. Zum anderen wurden Protokolle entwickelt, die durch eine „Begrenzung“ des Optimismus den Rücksetzungsoverhead verringern können. Die Idee beruht auf der Annahme, dass die Wahrscheinlichkeit der Fehlerfortpflanzung umso höher ist, je weiter in der virtuellen Zeit ein LP anderen LPs voraus ist. Die Verfahren versuchen dafür zu sorgen, dass ein LP nicht beliebig weit in die Zukunft Ereignisse ausführen darf, die Asynchronität andererseits aber auch nicht zu stark eingeschränkt wird. Die Wahrscheinlichkeit für das Auftreten einer Rücksetzung steigt mit der zeitlichen Distanz zwischen einzelnen Simulationspartnern, die bei unterschiedlichen Simulationsgeschwindigkeiten entsteht. Durch Verminderung dieser Distanz lässt sich die Rücksetzungshäufigkeit reduzieren.

- Dies wird z. B. mittels Durchführung der Simulation innerhalb sich mit der Simulationszeit bewegender Zeitfenster erreicht, so dass die unter Umständen überoptimistisch abgelaufene

Simulation weniger optimistisch fortschreitet und Straggler und somit Rücksetzungen seltener auftreten. Dabei dürfen beispielsweise nur Ereignisse innerhalb dieser für jeden logischen Prozess individuellen Zeitfenster, die sich während der Simulation verschieben und dabei eventuell auch ihre Fensterbreite anpassen, abgearbeitet werden. Im Moving-time-window-Verfahren *(MTW)* wird durch *[GVT, GVT+L]* ein Fenster konstanter Länge L definiert. Jeder LP darf nur Ereignisse mit einem Zeitstempel t ausführen, wenn t in diesem Fenster liegt [SoSt89] [SoBW88]. Ein Problem bei diesem Ansatz ist jedoch, eine geeignete Fensterlänge L zu finden. Offensichtlich kann es keine konstante optimale Länge L für alle Modelle geben. Eine nahe liegende Lösung dieses Problems besteht darin, die Fenstergröße nicht über virtuelle Zeitabstände festzulegen, sondern über die Anzahl der Ereignisse. Einem Simulator könnte beispielsweise erlaubt werden, maximal k Ereignisse mit einem Zeitstempel größer als die aktuelle GVT auszuführen. Der Wert von k könnte auch adaptiv vom Systemverhalten abhängig gemacht werden.

- Im Filtered-bounded-lag-Verfahren wird wie beim MTW-Verfahren das Zurückhinken („Lag“) einzelner LPs durch eine virtuelle Zeitspanne begrenzt [LuSh89].
- In Probabilistic-synchronization (Concurrent-resynchronization) lösen LPs in zufällig bestimmten Realzeitabständen eine Rücksetzung einer ganzen Gruppe von LPs aus [MaHa92] [MaWa91]. Eine Gruppe von LPs könnte aus allen LPs des Simulators oder auch nur aus solchen LPs bestehen, die häufig miteinander interagieren. Natürlich findet nur dort tatsächlich eine Rücksetzung statt, wo die Simulationszeit des Empfängers größer als die Rücksetzzeit ist. Dieses Schema unterscheidet sich von MTW und Filtered-bounded-lag vor allem dadurch, dass LPs durch die Synchronisation nicht blockiert werden.
- Ein anderer Ansatz wurde in [AgTi91] für TW vorgeschlagen. Dabei können sich einzelne LPs periodisch Ereignisse einplanen, die wie eine Schranke mit Zeitstempel t wirken. Solange eine solche Schranke in einem LP eingeplant ist, führt dieser keine Ereignisse mehr mit Zeitstempel $t' \geq t$ aus, bis $t = GVT$ gilt. Das Einplanen solcher Schranken könnte an das Systemverhalten gekoppelt werden.
- Das Breathing-time-buckets-Protokoll (BTB) basiert darauf, dass die Wahrscheinlichkeit, dass ein Ereignis im weiteren Lauf der Simulation revidiert werden muss, mit zunehmendem Abstand seiner Zeitstempel von der globalen virtuellen Zeit größer wird. Daher werden Nachrichten mit großem zeitlichem Abstand von der GVT verzögert ausgesendet [Stei93]. Bei Verwendung derartiger, zur Vermeidung von Instabilitäten bei der Performance von TW eingesetzter Protokolle fehlt die Ausnutzung der optimistischen Strategie jenseits der oberen Fenstergrenze, was sich in einem zu pessimistischen Verhalten und Performance-Verlusten äußern kann (siehe 3.6.4).
- Der Breathing-time-warp-Algorithmus (BTW) von Steinman [Stei93] kombiniert die Time-warp- [Jef85a] und BTB-Algorithmen [Stei91].
- Die Anzahl der Rücksetzungen kann drastisch reduziert werden, indem Rücksetzungskaskaden vermieden werden. Dies lässt sich dadurch erreichen, dass anderen LPs immer nur sichere Ereignisse eingeplant werden. Unter einem sicheren Ereignis wird ein Ereignis verstanden, das durch eine Ereignisausführung erzeugt wurde, die bis zum Ende der Simulation nicht mehr rückgängig gemacht wird. Wird z. B. ein Ereignis e erzeugt, so ist e sicher, sobald die GVT größer ist als die Eintrittszeit von e. Werden ausschließlich sichere Ereignisse in andere LPs eingeplant, müssen keine Negativnachrichten mehr versendet werden und es können keine Rücksetzungskaskaden auftreten. Rücksetzungen hingegen können noch auftreten, etwa wenn zwei sichere Ereignisse in umgekehrter Zeitstempelreihenfolge im gleichen LP eingeplant werden. Beispiele auf dieser Idee beruhender Verfahren sind etwa TW-ohne Risiko [Bell93], SRADS/LR [DiRe90], SPEEDES [Stei91] [Ste92b], LTW [RaAy93] sowie spekulative Simulation [Mehl91].

- Zur Begrenzung des Optimismus bei TW können Prädiktoren eingesetzt werden, die die möglicherweise in der Zukunft eintretende Ereignisse vorhersagen. Bei Predictive-time-warp (PTW) wird die Umgebung der einzelnen Teilsimulatoren während der Simulation ständig analysiert, um auf das nachfolgende Verhalten schließen und dieses für den weiteren Simulationsablauf möglichst präzise vorhersagen zu können. Im Gegensatz zu anderen, bereits bestehenden Ansätzen, die die Rücksetzungshäufigkeit durch Blockieren eines Teilsimulators zu vermindern suchen und somit Rechenkapazität ungenutzt lassen, schreitet bei PTW die Simulation ständig fort [ScTM97]. Das Protokoll wurde als Synchronisationskern für die Co-Simulationsumgebung SimBa implementiert [Stre97] und in Verbindung mit einem Logiksimulator eingesetzt [ScTM95].

  Im Rahmen der Arbeit wurde gemessen, dass PTW, insbesondere bei sehr komplexen Modellen und wenn der Rechenaufwand zur Berechnung der GVT klein ist, schneller als BTW ist. Ein Grund könnte sein, dass die Prädiktion im Vergleich zu dem bei BTW zum Einsatz kommenden Verfahren zur Optimismusbegrenzung verhältnismäßig viel Rechenzeit benötigt[53].

3.6.3 Erweiterung nach Frey

In der vorliegenden Arbeit wird auf die in [FrCW98] und [FrRa00] veröffentlichten Konzepte für die optimistische Co-Simulation hybrider Systeme zurückgegriffen. Es handelt sich hierbei um Synchronisationsverfahren, mit denen zeitkontinuierliche (Continuous simulator, CS) und ereignisdiskrete (Discrete event simulator, DES) Simulatoren zur optimistischen, parallelen Simulation gekoppelt werden können. Voraussetzung ist, dass ein hybrides Modell so partitioniert wird, dass keine kontinuierlichen Teilmodelle entstehen, die direkt von Simulationsergebnissen anderer kontinuierlicher Teilmodelle abhängen. Andernfalls würde der hohe interne Kommunikationsaufwand, der bei der Simulation eines kontinuierlichen Modells lokal entsteht, bei verteilter Ausführung dieses Modells eine hohe Netzwerklast erzeugen. Bei diesen atomaren kontinuierlichen Teilmodellen findet ein Datenaustausch also nur mit diskreten Teilmodellen statt. Dabei wird jedes Teilmodell auf einem eigenen Simulator berechnet und die CS mit den DES synchronisiert. Hinsichtlich der Synchronisation wird der CS von außen wie ein DES betrachtet, der bei Empfang eines Ereignisses, dessen Zeitstempel die Start- oder Stoppzeit des vom CS simulierten Zeitintervalls festlegt, aktiviert wird. Abhängig davon, ob der CS bei Empfang der Start- oder der Stoppzeit des jeweiligen Simulationsintervalls aktiviert wird, wird zwischen First Event Synchronization (FES) und Second Event Synchronization (SES) unterschieden. Die Grenzen des Simulationsintervalls des CS stellen somit die Synchronisationszeitpunkte zwischen kontinuierlichen und diskreten Domänen dar, an denen beim CS eine Zustandsspeicherung durchgeführt wird. Bei der FES besteht die Schwierigkeit darin, eine angemessene Stoppzeit für den CS zu wählen, da zur Aktivierung des CS nur die Startzeit benötigt wird. [FrCW98] enthält Messergebnisse, aus denen hervorgeht, dass die SES eine schnellere Simulationsdurchführung ermöglicht. Aus diesen Gründen wurde in der vorliegenden Arbeit auf der SES aufgebaut.

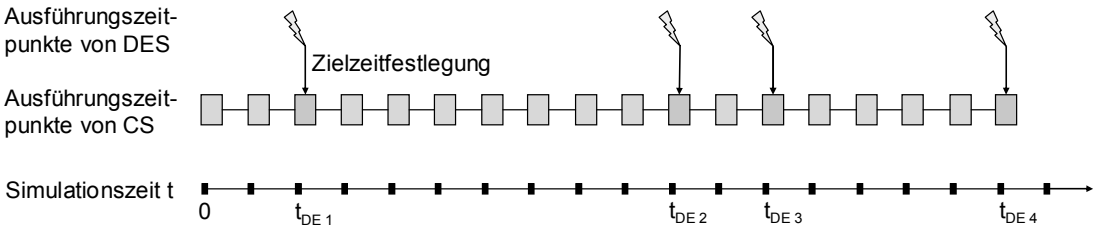


Bild 73: Bestimmung der Zielzeiten für den CS durch den DES

[53] Der Vergleich wurde mit mehreren lokalen Simulatorinstanzen auf einem Sun SuperSPARC IIi mit 333MHz und 512 MB RAM mit den ISCAS'89[53] Modellen s1238, s1494, s1423 und s1488 durchgeführt (Parameter: ein Prädiktor des Typs Previous State Predictor (PSP), maximale Vorhersageordnung drei).

Bild 73 verdeutlicht, wie die Zielzeiten für den CS vom DES bestimmt werden, wobei in diesem Beispiel der CS mit einer festen Schrittweite voranschreitet, so wie es auch in der vorliegenden Arbeit der Fall ist.

Hat der CS beispielsweise bis zum Zeitpunkt $t_{DE\ 1}$ simuliert und hat das nächste auszuführende Ereignis des DES den Zeitstempel $t_{DE\ 2}$, so wird dem CS vom DES dieser Wert als neue Zielzeit mitgeteilt und der reaktivierte CS kann von $t_{DE\ 1}$ bis $t_{DE\ 2}$ simulieren. So schreitet die Simulation von einer Zielzeit $t_{DE\ n-1}$ zur nächsten $t_{DE\ n}$ voran, vorausgesetzt, seitens des CS wird kein Schwellwert überschritten. Wie diese Schwellwerte definiert sind und wie eine Schwellwertüberschreitung festgestellt wird, ist weder in [FrCW98] noch in [FrRa00] ausgeführt. Konzeption und Implementierung dieses Aspekts ist eine wesentliche Teilaufgabe der vorliegenden Arbeit. An dieser Stelle wird davon ausgegangen, dass ein Schwellwert existiert und eine Überschreitung eintreten und festgestellt werden kann. Bei Erreichen der Zielzeit wird der Zustand des CS gespeichert und so ein Synchronisationspunkt erstellt. Durch dieses Vorgehen kann der Speicherbedarf für die Zustände des CS reduziert werden, vorausgesetzt, der zeitliche Abstand zwischen den Synchronisationspunkten ist größer als die Schrittweite des CS. Bei Erreichen der Zielzeit $t_{DE\ n}$ ohne Schwellwertüberschreitung gilt das simulierte Intervall $[t_{DE\ n-1}, t_{DE\ n}]$ als synchronisiert, da sowohl DES als auch CS bis $t_{DE\ n}$ simuliert haben und mögliche äußere Kausalitätsverletzungen aufgrund der optimistischen Simulation vom DES aufgefangen werden.

Tritt innerhalb des Intervalls $[t_{DE\ n-1}, t_{DE\ n}]$ eine Schwellwertüberschreitung auf, so hält der CS an dieser Stelle an, speichert seinen momentanen Zustand und sendet ein Ereignis mit dem errechneten Wert und dem Zeitpunkt des Simulationsabbruchs an den DES. In Bild 74 ist diese Zeit mit $t_{CS,SW}$ gekennzeichnet.

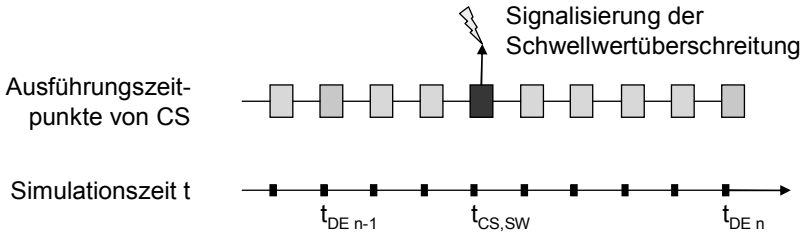


Bild 74: Schwellwertüberschreitung beim CS

Durch das vom CS erzeugte Ereignis wird beim DES eine Rücksetzung zum letzten Synchronisationspunkt bei $t_{DE\ n-1}$ ausgelöst, von wo ab die Simulation unter Einbeziehung von $t_{CS,SW}$ als zusätzliche Zielzeit neu startet. Der zusätzliche Synchronisationspunkt verhindert eine mögliche Blockierung, wie er in Bild 75 dargestellt ist: ohne den Synchronisationspunkt würden DES und CS zu $t_{DE\ n-1}$ zurückspringen und CS hätte erneut $t_{DE\ n}$ als Zielzeit. So käme es bei $t_{CS,SW}$ wieder zur Schwellwertüberschreitung, und die Rücksetzung zu $t_{DE\ n-1}$ würde abermals ausgelöst.

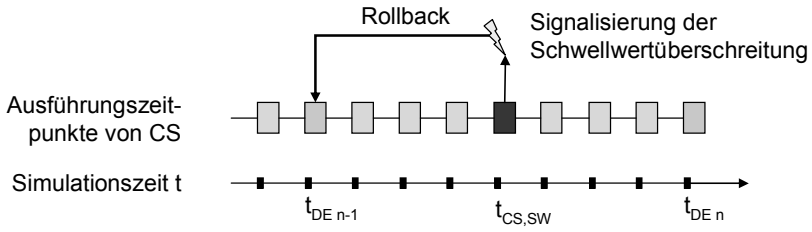


Bild 75: Blockade durch Schwellwertüberschreitung

Durch die zusätzliche Zielzeit $t_{CS,SW}$ tritt in den Intervallen $[t_{DE\ n-1}, t_{CS,SW}]$ und $[t_{CS,SW}, t_{DE\ n}]$ nun keine Schwellwertüberschreitung mehr auf und die Simulation kann über $t_{CS,SW}$ hinaus fortfahren.

Um das Fortschreiten der Simulation bis zur vom Benutzer vorgegebenen Endzeit sicherzustellen, muss der DES für diese Endzeit ein Ereignis haben, dessen Zeitstempel dem CS als Zielzeit dient. Liegen während der Simulation keine anderen Ereignisse vor, und aktiviert der DES daher den CS mit dem Ereignis der Endzeit, kann daraus ein großes Simulationsintervall resultieren, was die Wahrscheinlichkeit einer notwendigen Rücksetzung erhöht und zu größerem Rechenaufwand führt. Aus diesem Grund wird in [FrCW98] als zusätzliche Option eine wählbare Begrenzung der Intervallgröße vorgeschlagen.

3.6.4 Hybride Verfahren

Hybride Simulationsverfahren kombinieren konservative und optimistische Verfahren. Es werden horizontal- und vertikalhybride Verfahren unterschieden. Bei horizontalhybriden Verfahren koexistieren konservative und optimistische Ansätze. Beispielsweise könnte ein Teil aller LPs konservativ arbeiten, die restlichen LPs jedoch optimistisch [ArSm92] [PrRa88] [Reyn88]. Bei vertikalhybriden Verfahren verwenden alle LPs das gleiche rein konservativ und rein optimistisch angesiedelte Simulationsverfahren. Welche Variante dieser hybriden Verfahren am besten ist, hängt von der Anwendung ab.

Es wurden diskrete und kontinuierliche sequenzielle Simulationsmethoden auf einer Ebene miteinander kombiniert (siehe z. B. [Schm81]). Eines der ersten horizontalhybriden verteilten Simulationsverfahren kombiniert eine diskret ereignisgesteuerte konservative Methode mit einem Verfahren für kontinuierliche Simulation [HaDo88]. Auch Verfahren, die konservative ereignisgesteuerte Simulation mit zeitgesteuerter Simulation kombinieren, wurden vorgeschlagen [ChGo93].

Bei BTB (SPEEDES) führt jeder LP fortlaufend alle lokalen Ereignisse chronologisch aus [Stei91] [Ste92b]. Die für den eigenen LP erzeugten Ereignisse werden dabei sofort, diejenigen für andere LPs dagegen erst mit Erhalt der nächsten GVT-Approximation eingeplant. Bis dahin werden die letztgenannten Ereignisse in einer LP-lokalen Liste L gespeichert. Die bei BTB notwendige GVT-Berechnung wird von einer Zentrale durchgeführt. Zur Berechnung der GVT erhält diese von jedem logischen Prozess LP_i einen Zeitstempel größer oder gleich dem aktuellen Minimum M_i der Zeitstempel aller Ereignisse, in dessen Liste L ist. Die Mitteilungsnachricht enthält auch den Wert von M_i. Sobald die Zentrale diese Nachricht von allen LPs empfangen hat, können im restlichen Verlauf der Simulation keine Ereignisse mehr mit Eintrittszeit $t < \min_i\{M_i\}$ erzeugt werden. Dies ist offensichtlich gleichbedeutend damit, dass die GVT gleich $\min_i\{M_i\}$ ist. Infolgedessen sendet die Zentrale den Wert dieses Minimums als neue GVT, entfernt jeder LP alle Ereignisse mit einer Erzeugungszeit (Send-time) $t < GVT$ aus seiner Liste L und plant diese Ereignisse den empfangenden LP ein. Die Einfachheit der GVT-Berechnung in dieser Simulationsmethode ist vor allem darauf zurückzuführen, dass während der GVT-Berechnung keine Ereignisse auf dem Kommunikationsmedium unterwegs sein können. Das Verfahren hat eine entfernte Ähnlichkeit mit konservativen Verfahren, indem es anderen LPs nur sichere (d. h. nicht mehr annullierbare) Ereignisse einplant. Dazu werden GVT-Garantien verwendet derart, dass Verletzungen der Kausalität a priori reduziert werden. Umgekehrt ist das Verfahren auch optimistisch, da ein LP immer umgehend das lokal nächste Ereignis ausführt. Wenn der Zeitstempel eines ausgeführten Ereignisses sich im Nachhinein als größer als die nächste GVT herausstellt, so sind lokale Rücksetzvorgänge notwendig. Daher müssen wie bei TW entsprechende Rücksetzinformationen (wie etwa Kopien des lokalen Zustandsraums) fortlaufend angelegt werden. Negativnachrichten sind jedoch nicht erforderlich, da nur sichere Ereignisse ausgetauscht werden. Ein Nachteil dieser Methode offenbart sich für solche Simulationsmodelle, in denen zwischen zwei aufeinander folgenden GVT-Berechnungen nur wenige Ereignisse auszuführen sind: Hier muss die GVT unverhältnismäßig oft berechnet werden. Dies kann die Effizienz der Simulation senken, wie in [Bell93] gezeigt wird. Im Vergleich zu TW unterscheidet sich BTB in folgenden drei Punkten:

- es ist weniger optimistisch, da anderen LPs nur sichere Ereignisse eingeplant werden;

- es treten keine Rücksetzungskaskaden auf, weil keine Negativnachrichten benötigt werden;
- der GVT-Approximationsalgorithmus ist einfacher, weil Ereignisse nur zu bestimmten Zeiten über das Kommunikationsmedium ausgetauscht werden können.

Andere vertikalhybride Verfahren auf dem Gebiet verteilter, ereignisdiskreter Simulation werden u. a. in den Arbeiten von Aahlad und Browne [AaBr89], Chandy und Sherman [ChSh89] sowie Mehl [Mehl91] vorgestellt. Hierbei handelt es sich um so genannte „ausgewogene Simulationsstrategien", die zwischen den rein pessimistischen und der rein optimistischen Simulationsmethoden liegen.

3.7 Co-Simulations- und Emulationsumgebungen und Simulatoren

Die folgende Übersicht erhebt nicht den Anspruch der Vollständigkeit, sondern stellt lediglich eine Auswahl verfügbarer Werkzeuge dar. Verbreitete und wichtig erscheinende Systeme wurden ausgewählt. Ältere Werkzeuge, die längere Zeit nicht weiterentwickelt wurden und nicht als Basis für die vorliegende Arbeit dienten, werden nicht berücksichtigt. Reine Komponentenmodelle wie CORBA, DCE (Distributed Computing Environment) oder JavaBeans sind hier nicht explizit aufgeführt worden[54].

3.7.1 Frameworks für die Co- und Standalone-Simulation

Durch die Kopplung über eine flexible Schnittstellen-SW, wie z. B. die bei FZI/ESM (Forschungszentrum Informatik/Elektronische Systeme und Mikrosysteme) entwickelte Simulationsumgebung SimBa (Simulation Backplane), wird es möglich, verschiedene CASE-Werkzeuge, wie z. B. STATEMATE, MATRIXx und Verilog Saber, durch gegenseitige Integration funktionaler Blöcke miteinander zu koppeln, so dass eine Co-Simulation durchgeführt werden kann [ScTM95] [TaSM95] [ScTM95] [Diec96]. SimBa ermöglicht die gekoppelte Simulation auf verteilten Systemen sowohl mit kommerziellen als auch mit eigenentwickelten Simulatoren. Mit Hilfe entsprechender Simulatoren lassen sich auch gemischt analoge und digitale Systeme simulieren [TaSM95] [ScTM95] [Nied96]. Die Kommunikation erfolgt über ein auf TCP/IP aufgebautes Kommunikationsmodul. SimBa ist zuständig für die Steuerung des Simulationsablaufs, die Synchronisation der einzelnen Simulatoren sowie für die Kommunikation der Simulatoren untereinander. Durch den modularen Aufbau können Synchronisationsmodule, in denen verschiedene Algorithmen implementiert sind, gegeneinander ausgetauscht werden. Die objektorientiert in C++ implementierte, für SunOS entwickelte Simulationsumgebung ist in drei Versionen vorhanden: einer kommerziellen zeitgesteuerten Version mit einfachem, synchronen Lock-Step-Algorithmus[55] [Diec96] sowie ereignisgesteuerten Versionen mit den optimistischen Algorithmen TW und PTW. Der Aufwand für die Portierung für das BS Windows wäre nicht unerheblich aufgrund der verwendeten proprietären UNIX-IPC. Deshalb wurde eine neue Testumgebung entwickelt (siehe 4.2).

Der Kern ist ein im Hintergrund laufender Backplane-Prozess, der über eine graphische Benutzeroberfläche gesteuert wird. Nachdem die Simulatoren an die Backplane angekoppelt wurden, können sie von ihr kontrolliert werden und über sie miteinander kommunizieren. Sie haben Zugriff auf eine globale Datenbank (Shared Database, SDB) und auf ihre eigenen lokalen Datenbanken (Local Databases, LDB) mit den ihnen zugewiesenen Modelldaten als Partitionsbeschreibung. Die logische Verbindung der Simulatoren untereinander erfolgt über globale Netze, an die sie mit ihren Ein- und Ausgängen (Ports) angeschlossen werden und über die sie Änderungen ihrer Ausgangswerte als Ereignisse versenden können. In der globalen Datenbank sind Daten zur Simulationsumgebung wie Namen von Programmen, lokalen Datenbanken und Simulatoren und deren Verteilung auf einzelne Rechner abgelegt. Die lokalen Datenbanken enthalten neben den Daten des zu simulierenden Modells auch Angaben über die Verbindung der Ports des Modells mit den globalen Netzen. Die Simu-

[54] CORBA wird in 3.5.2.2 angesprochen.

[55] Der Lock-Step-Algorithmus ruft jeweils den Simulator auf, in dessen Modell die nächsten Änderungen möglich sind, und verhindert so die Entstehung von zeitlichen Widersprüchen.

lationsumgebung wird durch die GUI aufgebaut, die den Backplane-Prozess und die einzelnen Simulatoren aufruft. Nach einer Ankopplungs- und Initialisierungsphase fordert die Backplane die Simulatoren auf, sich ebenfalls zu initialisieren und ihre Ports bei ihr anzumelden [Stre96]. Nach abgeschlossener Initialisierung kann die gekoppelte Simulation durchgeführt und von außen durch die GUI gesteuert werden. In Bild 76 ist der Aufbau von SimBa dargestellt.

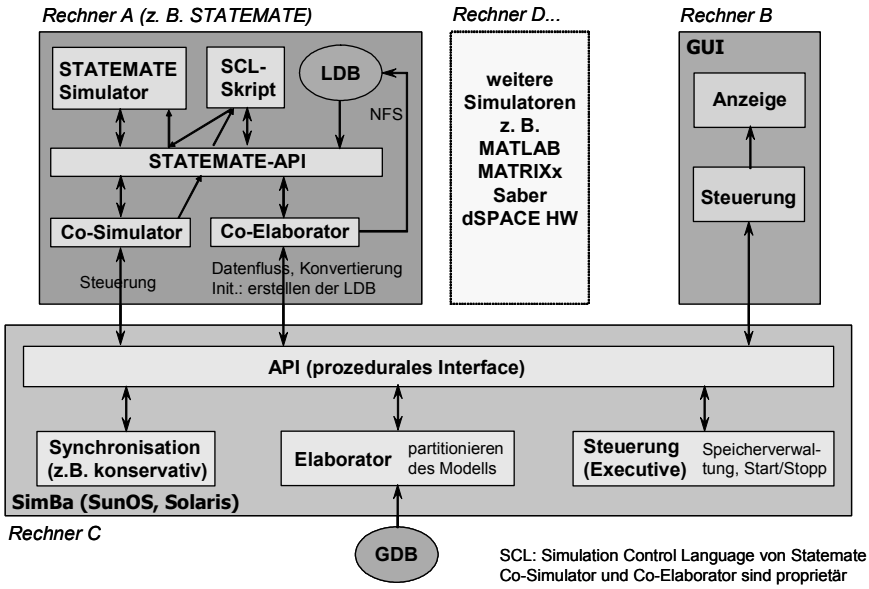


Bild 76: Prinzip der Simulation Backplane SimBa

Im Gegensatz zu den konservativen Verfahren überlässt SimBa bei optimistischer Synchronisation den einzelnen Simulatoren weitgehend die Verantwortung für die lokale Simulationsablaufsteuerung. Die Aufgaben von SimBa beschränken sich im Wesentlichen auf die Weiterleitung der zwischen den Simulatoren ausgetauschten Nachrichten sowie die Bestimmung der globalen virtuellen Zeit [SiBa-97].

Weitere Frameworks für die Co-Simulation sind z. B.

- NovaSim© von MicroNova: Das Testsystem NovaSim ermöglicht einen Verbundtest durch Kopplung von HIL-Simulatoren. Der Simulator wurde unter Verwendung verschiedener E/A-Komponenten, wie z. B. rekonfigurierbare E/A-FPGA-Karten, aufgebaut [RaDe05].
- MILAN: Ziele von MILAN (model based integrated simulation framework) sind die Integration heterogener Simulatoren und die Entwicklung von Modellen zur formalen Repräsentation der Systemstruktur und des Systemverhaltens. MILAN ermöglicht die Evaluierung von Performance-Metriken wie Energiebedarf [BaPL01].
- IDES: IDES ist eine Infrastruktur und ein Framework zur verteilten Simulation mit Zielanwendungen im medizinischen Bereich, das sowohl in Java als auch in C++ entwickelt wurde [NiJo97]. Die Synchronisation basiert auf dem optimistischen BTB-Protokoll [siehe 3.6.2.6].

Ein Framework für die Standalone-Simulation ist z. B. Ptolemy II. Ptolemy II ist ein frei verfügbares SW-Framework (inklusive Simulatoren), das vom Department of Electrical Engineering and Computer Sciences der University of California at Berkeley im Rahmen des Ptolemy-Projekts entwickelt wurde. Ptolemy II besteht aus Java-Paketen und unterstützt Modellierung und Entwurf heterogener und nebenläufiger Systeme. Es können z. B. ereignisdiskrete Systeme, Datenflüsse, Prozessnetzwerke, synchrone und reaktive Systeme sowie kommunizierende sequenzielle Prozesse dargestellt werden. Ähnlich wie Simulink ist Ptolemy II eine graphische Sprache, die auf Blockdiagrammen basiert [Ptol03]. Der Schwerpunkt von Ptolemy II liegt bei digitaler Signalverarbeitung; Ptolemy wurde nicht für Co-Simulations-Aufgaben entworfen.

3.7.2 Logiksimulatoren für die Co- und Standalone-Simulation

Der Logiksimulator oLogic wurde bei FZI/ESM in der Arbeit [Diec96] an SimBa angekoppelt, um mittels eines einfachen Lock-Step-Synchronisationsprotokolls gekoppelte Simulationen durchzuführen. Für die einzulesenden Schaltungsnetzlisten verwendet er ein Format ähnlich dem der ISCAS-Benchmark-Schaltungen (International Symposium on Circuits and Systems) [BrBK89] [BrFu85], die für Vergleichsmessungen bei Logiksimulationen eingesetzt werden. Als Schaltungselemente stehen Inverter, AND-, OR-, NAND- und NOR-Gatter sowie D-Flipflops zur Verfügung, die sich über Signale miteinander verbinden lassen. Zur Kopplung mehrerer Schaltungspartitionen untereinander über die Backplane wurde das ISCAS-Format für die Ein- und Ausgangsblöcke derart ergänzt, dass sich Verbindungen von Schaltungsausgängen mit den zugehörigen Eingängen einer anderen Partition festlegen lassen. Offen gebliebene Eingänge können als Stimuli-Muster-Generatoren eingesetzt werden, wobei die Wahl zwischen periodischen Rechtecksignalen und stochastischen Verläufen besteht. Alle Gatter besitzen eine wählbare Schaltungsverzögerung. Der Logiksimulator ist ein Simulator zur Simulation diskreter Ereignisse, der mit ganzzahligen Simulationszeitwerten arbeitet. Um die Rücksetzungshäufigkeit von vornherein niedrig zu halten, wurde faule Annullierung (siehe 3.6.2.3) zum Widerrufen von Nachrichten nach einer Rücksetzung eingesetzt. Die Zustandsspeicherung erfolgt nach jedem abgeschlossenen Simulationszeitintervall, womit sich –durch das so erzielte Interleaved State Saving – Rechenzeit und Speicher einsparen lassen. Der im Rahmen dieser Arbeit entwickelte Logiksimulator JLogSim (4.3.1) basiert auf oLogic.

Eine andere Umgebung zur verteilten Logiksimulation, die bereits 1992 vorgestellt wurde, ist z. B. PARASOL (parallel silos oriented logic simulator) [LaBa92].

Logiksimulatoren für die Standalone-Simulation sind z. B.

- ILS: Der Logiksimulator Iowa Logic Simulator (ILS) ist zu Schaltungsbeschreibungen der „Iowa Logic Specification“-Sprache kompatibel [Jone01] und als Quellcode verfügbar. Der ILS wurde 1983 in C entwickelt und verfügt über keine Netzwerkschnittstelle.
- C++ SIM: C++ SIM wurde 1994 von der University Newcastle in Großbritannien entwickelt und verfügt über keine Netzwerkschnittstelle.

3.7.3 Simulink-basierte Lösungen

Simulink-basierte Lösungen für die Co-Simulation sind z. B.

- Extessy-Blocksatz: Die EXTESSY AG, eine Ausgründung des IRA der TU Braunschweig, bietet unter dem Namen EXITE© (EXTESSY Inter Tool Engineering) eine Plattform zur verteilten – nicht optimistischen – Simulation mit unterschiedlichen CASE-Werkzeugen an. U. a. stehen Schnittstellen zu Simulink, ASCET und Rhapsody in C++ zur Verfügung [EXCI03]. Die Distributed Simulink Toolbox (DS-Toolbox) ist die EXITE-Schnittstelle zu Simulink; EXITE bietet keine Schnittstellen zu anderen Werkzeugen an [EXDS02]. Die DS-Toolbox greift auf den EXITE-Service im PC-Netzwerk zurück, der die Vernetzung der Simulation koordiniert und von jedem EXITE-Block aus per Konfigurationsoberfläche erreichbar ist. Die Kommunikation wird auf der Basis von CORBA durchgeführt [Schu03].

  Wie die im Anhang aufgeführten Messergebnisse zeigen, lässt sich mit EXITE bei verteilter Simulation kleiner Modelle keine Ausführungsbeschleunigung erreichen. Die Handhabung in Simulink ist komfortabel, es gilt lediglich zu beachten, dass bei Verwendung mehrerer EXITE-Blöcke in einem Modell diese jeweils in Subsysteme gekapselt werden müssen. Ein großer Unterschied im Vergleich zu dem im Rahmen dieser Arbeit implementierten Framework ist der Zugriff auf die Werte der Simulationsdaten: diese werden bei EXITE durch eine Simulatorschnittstelle abgegriffen, wobei Abtastfehler entstehen können oder bei zu hoher Abtastrate Performance-Einbußen möglich sind. Bei der im Rahmen dieser Arbeit realisierten Lösung wird direkt auf die Werte im jeweiligen Simulator intern zugegriffen, was eine äußere Signalabtastung überflüssig macht. Dieses Vorgehen macht allerdings eine simulatorspezifische Im-

plementierung für jeden Simulator notwendig. Generalisierte Schnittstellen, wie sie bei EXITE verwendet werden, können bei dem im Rahmen dieser Arbeit implementierten Framework in dieser Weise nicht verwendet werden. Diese Schnittstellen machen EXITE zu einem flexiblen System, das mit vertretbarem Aufwand an viele Simulatoren angekoppelt werden kann. Beim implementierten Framework hingegen ist der Aufwand für die Ankopplung deutlich größer, aber die Leistung auch deutlich höher, da die Ankopplung für den jeweiligen Simulator optimiert ist. Einige Aspekte des realisierten Frameworks können mit EXITE kaum umgesetzt werden, so beispielsweise die Simulation mit optimistischen Algorithmen, da EXITE auf simulatorinterne Systemzustände nicht zugreifen kann. Ein Lösungsansatz wäre eine Middleware, die zwischen dem eigentlichen Simulator und EXITE geschaltet wird. Dies käme dann aber wieder der im Rahmen dieser Arbeit realisierten Lösung nahe.

- xPC TargetBox©: Die xPC TargetBox von The MathWorks ist ein PC-basiertes HW-System für das Rapid Prototyping mit graphischer Entwicklungs-SW [xPCT02]. Mit Simulink erstellte Simulationen können auf der xPC TargetBox in Echtzeit ablaufen [ScSc03].
- Link for ModelSim©: Link for ModelSim [Math05] ist eine Co-Simulations-Schnittstelle, das Simulink in den HW-Entwicklungsprozess für FPGAs und ASICs einbindet. Es stellt eine bidirektionale Verknüpfung zwischen Simulink und Model Technology ModelSim zur Verfügung. Dadurch können ModelSim-Modelle auf RTL-Ebene gegen Simulink verifiziert werden [LiMo04].

3.7.4 Sequenzielle Simulatoren für die Co-Simulation

Es existieren eine Reihe sequenzieller Simulatoren für die Co-Simulation. Beispiele sind:

- DESMO-J: DESMO-J (Discrete-Event Simulation and MOdelling in Java) [LePa99] sind Java-Bibliotheken zur sequenziellen verteilten Simulation ereignisdiskreter Systeme. DESMO-J erweitert Java mit Eigenschaften zur Konstruktion von Modellen zur ereignisdiskreten Simulation (siehe [PaLe00] und [DESM05]).
- ClearSim, ClearSim-MultiDomain und ClearSim 2000: ClearSim [AlDD98], ClearSim-MultiDomain (ClearSim-MD) [KrBE01] und ClearSim 2000 werden am Institut für Rechnerstrukturen und Betriebssysteme (IRB) des Fachbereichs Elektrotechnik der Universität Hannover entwickelt. ClearSim kann erweiterte endliche Automaten einlesen und benötigt dafür als Eingabeformat C++-Code [BrMS99]. Funktionalität und Zeitverhalten können bei ClearSim-MD als formloser Text oder graphisch durch eine UML- oder SDL-Beschreibung spezifiziert werden. Wenn die Beschreibung mittels UML erfolgt, ist ein Klassendiagramm die Basis für eine System-Simulation. Zustandsdiagramme können entweder in erweiterte Zustandsdiagramme mit Spezifikation des Zeitverhaltens oder in SDL konvertiert werden. Nach Konvertierung in SDL können daraus u. a. C-Code oder Simulink-Modelle generiert werden. Eines der Hauptziele von ClearSim 2000 ist die Modellierung und Simulation mechatronischer Systeme und hier insbesondere die Kopplung verschiedener Simulatoren über eine Backplane. Es handelt sich allerdings nicht wirklich um eine verteilte Simulation, da alle Ereignisse über eine einzige globale Ereigniswarteschlange verwaltet werden und die einzelnen Simulationskomponenten nicht gleichzeitig, sondern nacheinander rechnen. ClearSim 2000 besteht im Wesentlichen aus drei Teilen [ScMS99]: dem ClearSim-Kern, der die Simulation steuert, der Schnittstelle zur Ankopplung von Simulationsmodulen und der graphischen Benutzeroberfläche. Der ClearSim-Kern ist für den Ablauf der Simulation verantwortlich. Er startet die einzelnen Simulatoren und führt während des Ablaufs den Datenaustausch und evtl. nötige Datenkonvertierungen durch. Mit dem Kern wird durch eine Schnittstelle kommuniziert und so vom konkret benutzten Kommunikationsmechanismus abstrahiert. Die graphische Oberfläche integriert Modellierungs- und Analysewerkzeuge, mit denen das zu modellierende System zusammengestellt und analysiert werden kann [Brie01].

3.7.5 Sprachen und Bibliotheken für die PDES

Eine Simulationssprache stellt einen vollen Satz genau abgegrenzter Sprachkonstrukte zur Verfügung, um Simulationsmodelle zu entwerfen. Im Vergleich dazu stellt eine Bibliothek nur eine Gruppe von Routinen zur Verfügung, die mit einer Basis-Programmiersprache (z. B. C oder C++) eingesetzt werden [LoLC99].

Sprachen sind u. a. APOSTLE [WoBr96], Parsec [Bagr97] [MeBa99], ModSim [WeMu88] und YADDES [PreY90]. APOSTLE und ModSim unterstützen nur ein optimistisches Synchronisationsprotokoll und bieten keine Statistikfunktionalitäten an. Parsec und YADDES unterstützen verschiedene konservative und optimistische Synchronisationsprotokolle sowie Statistikfunktionalitäten. Von den vier oben aufgeführten Sprachen verfügt nur Parsec über eine visuelle Programmierumgebung.

Bibliotheken sind u. a. CPSim [Gros95], GTW (Georgia Tech Time Warp) [DaFP94], ParaSol (Parallel Simulation Object Library) [MaKR95], PSK (Parallel Simulation Kernel) [RoLA96], SimKit [GoFU95] [Simk05], SPaDES (Structured Parallel Discrete-Event Simulation) [TeTK96] [SPaD01], SPEEDES [Ste92a], TWOS [Pete90] und WARPED [DaWT95]. Mit Ausnahme der kommerziellen, C-basierten, konservativen PDES-Bibliothek CPSim basieren alle genannten Bibliotheken auf optimistischen Protokollen. GTW und TWOS basieren ebenfalls auf C, die sonstigen Bibliotheken auf C++. SPaDES ist mittlerweile auch als RMI-basierte Java-Bibliothek, die nach CMB einen konservativen, auf Nullnachrichten basierten Algorithmus unterstützt, verfügbar [TeNg02]. Von der Firma Metron vertriebene SPEEDES implementiert das BTB-Protokoll, bei dem alle Rücksetzungen für jeden LP lokal bleiben und nicht zu anderen LPs propagiert werden (siehe 3.6.2.6) [SPEE03]. Mit Ausnahme von SPEEDES unterstützen alle genannten Bibliotheken die vollständige Zustandsspeicherung, bei der das System den vollständigen Zustand eines LP nach jedem Zeitinkrement speichert. SPEEDES unterstützt nur die inkrementelle Variante, bei der die Veränderungen der Zustandsvariablen zum vorher gespeicherten Zustand gespeichert werden. CPSim, GTW und TWOS unterstützen Statistikfunktionalitäten. Von den oben aufgeführten Bibliotheken verfügen nur GTW, PSK und TWOS über eine Visualisierungsmöglichkeit.

- Versionen von GTW [GTW94] existieren für SM-Architekturen und heterogene Workstation-Netzwerke (Network Of Workstations, NOW). Die SM-Version ist als Quellcode für Forschungszwecke frei erhältlich [GTTW05]. GTW wurde 1997 von Fujimoto (Georgia State University, Atlanta) entwickelt und besitzt einen monolithischen Kern. GTW ist zwar C-basiert und verwendet UNIX-Funktionen, besitzt aber eine C++-Schnittstelle: GTW++ [GTWP-96].
- WARPED wurde 1996 von Radhakrishnan (University of Cincinnati) entwickelt, basiert auf MPI und ist als Quellcode frei erhältlich [WARP99].
- Parsec ist „mächtiger“ als C, C++ und Java, da z. B. Funktionen zur Berechnung von Statistiken und für das Ereignis-Scheduling integriert sind. Parsec ist nur in Binärform frei verfügbar.
- ParaSol wurde von Rego an der Purdue University in Indiana (USA) entwickelt und unterstützt PVM und MPI.

3.7.6 Simulatoren für die PDES

Simulatoren für die PDES sind z. B.

- ROSS: ROSS (Rensselaer's Optimistic Simulation System) wurde von Carothers am Rensselaer Polytechnic Institut, NY, entwickelt und ist ein modulares, C-basiertes Time-Warp-System [CaBP00]. ROSS wurde für Telekom-Anwendungen und SM-Architekturen entwickelt und besitzt einen modularen Kernel.
- ProperVHDL: Krishnaswany und Banerjee beschreiben in [KrBa96] den Entwurf und die Implementierung des Time-Warp-basierten Simulators ProperVHDL für VHDL-Beschreibungen.

3.7.7 Emulatorsysteme

Der Ausführungsgeschwindigkeit wird sowohl im industriellen als auch im universitären Umfeld große Bedeutung zugemessen. Neben Gemeinsamkeiten, wie der Verwendung der gleichen Simulations- und Synthesewerkzeuge, werden auch die Unterschiede sichtbar. Einige Systeme erreichen in ihren jeweiligen Einsatzgebieten mit hohem HW-Einsatz wesentlich höhere Leistungen, bieten jedoch nicht die gleiche Flexibilität wie das in der vorliegenden Arbeit entwickelte System.

3.7.7.1 Emulink

Emulink ist ein kommerzielles Programmpaket zur Simulationsbeschleunigung von Chipentwürfen, das auch mit dem Synthesewerkzeug LeonardoSpectrum von Mentor Graphics [Ment05] verknüpft werden kann. Unter der Bezeichnung Simulation-Speedboard ist ein Komplettpaket zur Simulationsbeschleunigung erhältlich. Dieses umfasst sowohl die SW Emulink-Mapper und die Schnittstelleneinheiten zu ModelSim der Firma Model Technology [MoTe05] und LeonardoSpectrum, als auch die gesamte erforderliche Hardware auf einer PCI-Karte (Peripheral Component Interconnect-Karte). Emulink-Mapper unterstützt die Implementierung der in der HDL-Simulation (Hardware Description Language-Simulation) zu beschleunigenden Designkomponente auf der PCI-Karte. Die mitgelieferte HW ist zur Erreichung einer hohen Beschleunigung an ModelSim angepasst [PDEC05].

3.7.7.2 Berkeley Emulation Engine (BEE)

BEE ist die Abkürzung für Berkeley Emulation Engine. Dabei handelt es sich um einen Echtzeit-HW-Emulator für digitale Kommunikations- und DSP-Systeme. Verwendung finden 20 Xilinx-Virtex-E-FPGAs, die es der BEE erlauben, über 600 Giga-Operationen pro Sekunde auszuführen und somit ein zum 10-Millionen-Gatter-ASIC äquivalentes System zu emulieren. Weiterhin steht eine E/A-Bandbreite von über 200 GBit/s für hochkommunikative Anwendungen zur Verfügung. Mit einem integrierten Entwurfsfluss von Simulink zur Implementierung kann der Anwender sein System gleichzeitig im BEE-System oder ASIC entwickeln. Das Werkzeug SF2VHD [Came01] entstand in einer Master-Arbeit aus dem Umfeld der BEE. Es bildet die Grundlage für das in dieser Arbeit entwickelte Werkzeug JVHDLGen (siehe 4.3.4).

3.7.7.3 VIKING CSM

Das VIKING-CSM-Co-Simulator-System von Mentor Graphics bietet in Zusammenarbeit mit dem NSIM-System (Hardware) und dessen enger Integration in ModelSim eine HW-Simulationsbeschleunigung von bis zu 16 Millionen Gattern. Der dabei verwendete Compiler übersetzt dabei bis zu 50.000 Gatter pro Minute. Es können mit diesem System, wie in ModelSim üblich, sowohl Logik- als auch Zeitsimulationen durchgeführt werden. Die Beschleunigung, die mit diesem System erreicht wird, erstreckt sich von Faktor 10 im Bereich der Module (ca. 500.000 Gatter) über den ASIC-Bereich (ca. 1.000.000 Gatter) bis hin zum SoC-Bereich mit mehr als 1.500.000 Gatter und einem Faktor größer 70 [Ment02].

3.8 Einsatz der Java-Technologie

Java-Programme werden mittels einer Java Virtual Machine (JVM) ausgeführt, die nominell plattformunabhängigen Bytecode interpretiert. Neuere Entwicklungen wie statische Analyse, Just-In-Time-Compilierung (JIT-Compilierung), JVM-Optimierungen und Optimierungen auf Instruktions-Ebene haben die Ausführungseffizienz verbessert, so dass Java bei einigen Applikationen und auf einigen Plattformen mittlerweile vergleichbar mit C und C++ ist. Mehrere Projekte zielen darauf, u. a. die RMI-Performance weiter durch Einsatz neuer Techniken zu verbessern [GhPa03].

Vorteile der Java-Technologie sind höhere Produktivität im SW-Entwicklungsprozess, hohe Zuverlässigkeit der SW, die in Java entwickelt wurde und die Flexibilität, die durch Mechanismen wie dynamisches Klassenladen möglich wird. Nachteile sind der hohe Speicherbedarf, die oft schlechte Laufzeitperformance und der Mangel an Echtzeitgarantien. Einer der größten Vorteile von Java, die

sichere Speicherverwaltung durch automatische Speicherbereinigung, ist zugleich auch das größte Problem beim Einsatz in Echtzeitsystemen. Die automatische Speicherbereinigung ist jedoch als Basis für die Sicherheitsmechanismen zwingend erforderlich. Sie findet automatisch Speicher, der von der Anwendung nicht mehr referenziert wird, und gibt ihn wieder frei, damit er erneut genutzt werden kann. In klassischen Java-Implementierungen mit einem eigenen Thread für die Speicherbereinigung werden alle anderen Threads angehalten, solange die Speicherbereinigung läuft. Die sich ergebenden Pausen sind aus der Sicht des Programmierers nicht planbar und machen die Vorhersage des Zeitverhaltens der Anwendung unmöglich. Aus diesem Grund wurde die Real-Time Specification for Java (RTSJ) [Boll00] [RTSJ05] von der Real-Time for Java Expert Group definiert, mit dem Ziel, die Java-Sprachdefinition und der Standardbibliotheken um Echtzeit-Threads zu erweitern, deren Ausführungszeit bestimmten zeitlichen Kriterien unterliegen. Z. B. ist die JamaicaVM [Jama05] eine Java-Implementierung, die Echtzeitverhalten für alle Threads anbietet [Sieb03]. In Systemen mit automatischer Speicherbereinigung werden Objekte nicht vom Programmierer, sondern vom BS freigegeben. Das System findet selbst heraus, welche Objekte noch über Zeiger referenziert werden, und gibt alle nicht mehr erreichbaren Objekte frei [Krüg03].

Eine empirische Performance-Evaluierung von Java-RMI und einen Vergleich mit dem Java Sockets API findet man in [AhQu00]. In Bild 77 ist die mittlere Antwortzeit für Leseoperationen in Abhängigkeit von der Anzahl der Clients, die auf einen Server zugreifen, dargestellt. Bild 78 zeigt die mittlere Antwortzeit für Aktualisierungsoperationen.

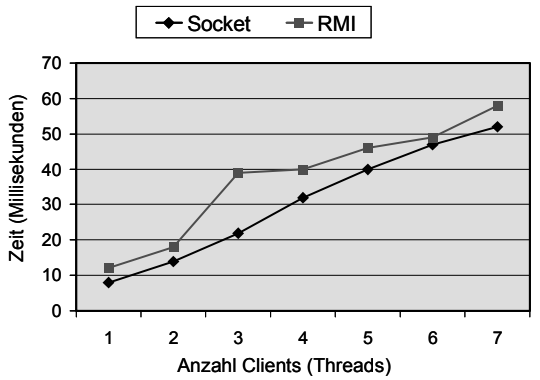


Bild 77: Performance-Evaluierung von Java-RMI

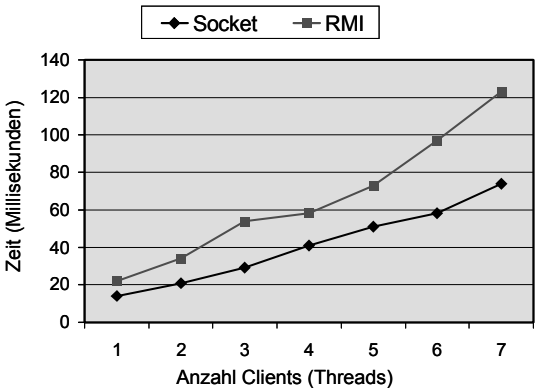


Bild 78: Mittlere Antwortzeit für Aktualisierungsoperationen

4 INTERACT-Framework

4.1 Anforderungen und Anwendungsszenarien

Eine Voraussetzung zur Kopplung von Simulatoren ist ein geeignetes Framework, das bestimmte Voraussetzungen erfüllen sollte [Drei04]. Dabei spielt auch die verwendete Middleware eine zentrale Rolle. Anforderungen an eine Testumgebung zur verteilten Simulation sind u. a. Plattformunabhängigkeit, modularer Aufbau und eine möglichst geringe Anfälligkeit für Implementierungsfehler. Die erreichbare Performance des Gesamtsystems hat nicht die höchste Priorität. Basierend auf diesen Anforderungen bietet sich z. B. die Programmiersprache Java an, die mit Java-RMI bereits eine Implementierung für eine abstrakte, plattformunabhängige Client-/Server-basierte IPC beinhaltet. Weiterhin sollte die Testumgebung konservative und optimistische Synchronisationsverfahren unterstützen, damit eine flexible Kopplung von Simulatoren möglich ist und untersucht werden kann. Als Programmiersprache wurde Java gewählt, da es weniger fehleranfällig als C++ und besonders hinsichtlich der Netzwerkprogrammierung plattformunabhängig ist.

4.2 Überblick

Um diesen Anforderungen gerecht zu werden, wurde ein Co-Simulations-Framework entwickelt (Bild 79), das heterogene Ausführungsumgebungen integriert [DrMG03].

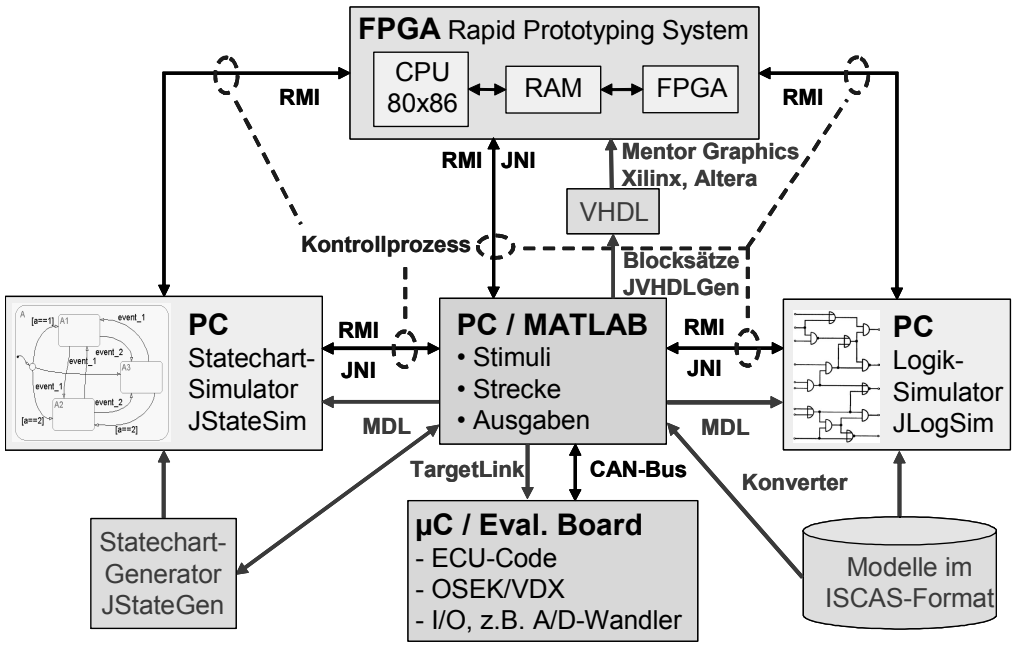


Bild 79: Co-Simulations-Framework

OSEK und ECU-Code werden auf einem Mikroprozessor ausgeführt, während die Testsignale, die Anzeige von Eingabesignalen und Ergebnisse und der interaktive Teil der Umgebung in Simulink verbleiben. OSEK ist heute der RTOS-Standard in der Automobilindustrie. Rechenzeitintensive Teile der Umgebung werden mit einem FPGA emuliert und mit dedizierten Statechart- und Logiksimulatoren simuliert. Deshalb wurde eine FPGA-basierte Simulationsbeschleunigung, ein Statechart-Simulator (JStateSim) und ein Logiksimulator (JLogSim) für eine plattformunabhängige Ausführung in Java entwickelt, sowie eine Target/Host[56]-Kommunikations-Infrastruktur zur Realisierung einer Processor-in-the-Loop-Simulation (PIL) zwischen dem CASE-Werkzeug Simulink und 32-Bit-Mikrocontroller-Targets wie z. B. Motorola PPC 555.

[56] Host ist die Bezeichnung für den Rechner eines Knotens in einem verteilten System.

Eine Anforderung zur Ausführung einer Co-Simulation sind passend partitionierte Modelle, so dass jeder Teilnehmer so effizient wie möglich ist und der zusätzliche Kommunikations-Overhead, der durch die Verteilung eines Modells erzeugt wird, so klein wie möglich bleibt. Deshalb wurde ein Vorgehen zur Partitionierung von Statecharts vorgeschlagen, das bewährte Partitionierungsalgorithmen für Logikschaltungen mit einbezieht [DrDZ03]. Eine andere Möglichkeit, die Simulationsgeschwindigkeit zu erhöhen, ist der Einsatz dedizierter Simulatoren wie JStateSim mit optimistischen Synchronisationsprotokollen als Option. Da das Framework flexibel entworfen wurde, kann es erweitert werden, um auch kontinuierliche Systeme zu unterstützen.

Zur schnellen Simulation komplexer Statecharts (erweiterte endliche Automaten) [StDr99] und Logikschaltungen sollen beliebig viele Simulatorinstanzen (LPs) gekoppelt werden können (Bild 80). Die Ausführungsgeschwindigkeit einer Simulation kann durch die Konvertierung eines Stateflow-Modells in Java-Code weiter erhöht werden: Java-Bytecode kann nach der Konvertierung des Modells generiert werden, so dass das Modell wie eine Standard-Java-Klasse innerhalb der JVM ausgeführt und dynamisch während der Simulation geladen werden kann.

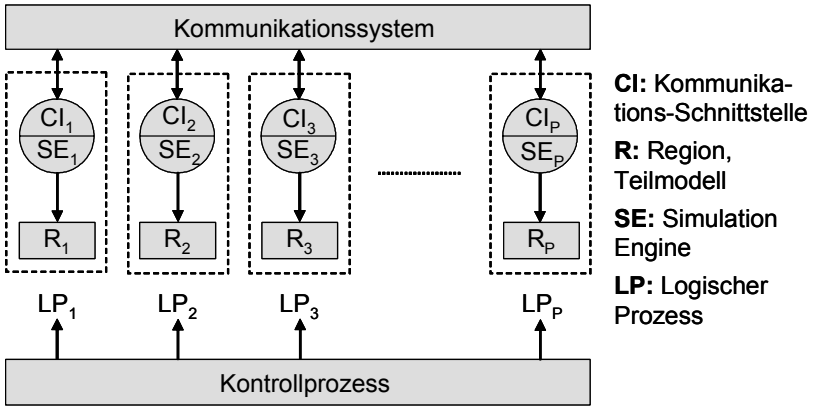


Bild 80: Kopplung mehrerer Simulatorinstanzen

Als Middleware zur Steuerung sowie zum Datenaustausch dient Java-RMI. Die Kommunikation zwischen der C-Schnittstelle von MATLAB (S-Funktion) und Java erfolgt über das JNI. Da eine geeignete Partitionierung der Modelle und das Synchronisationsprotokoll einen wesentlichen Einfluss auf die Simulationsgeschwindigkeit haben, wurden Untersuchungen zur Partitionierung von Statecharts durchgeführt. Ziel ist die Reduktion von Rechenzeit und Kommunikationsaufwand. Ein Kontrollprozess steuert die Simulatoren und den Datenaustausch. Mit Hilfe des in Java entwickelten Werkzeugs JStateGen können komplexe Testmodelle automatisch generiert und ISCAS-Performance-Schaltungen eingelesen werden [ISCA89].

Als Middleware für die Kontrolle und den Datenaustausch wird Java RMI genutzt. Die Kommunikation zwischen der C-Schnittstelle von MATLAB (S-Funktion) und Java ist mit Hilfe des Java Native Interface (JNI) realisiert (Bild 81).

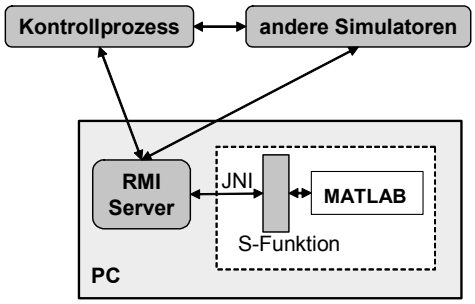


Bild 81: Schnittstelle zu MATLAB

Ein Kontrollprozess kommuniziert mit dem RMI-Teil der in Java implementierten Simulatoren sowie mit dem RMI-Server, der seinerseits mit MATLAB kommuniziert. Dieselbe Kommunikationsstruk-

tur kommt bei der Einbindung weiterer Simulatoren zum Einsatz. Der RMI-Server läuft auf dem Host-PC, auf dem auch MATLAB läuft, und etabliert die Kommunikation zwischen der MATLAB-C-Schnittstelle und der externen Umgebung.

Die Simulationsumgebung wird in drei Schichten eingeteilt: die Simulatorschicht, die Synchronisationsschicht und die RMI-Schicht, die jeweils unabhängig voneinander implementiert werden können (Bild 82). Für eine Beschreibung der Implementierung der Simulator-, Synchronisations- und RMI-Schicht sowie der MATLAB-Schnittstellen siehe [Bukh03].

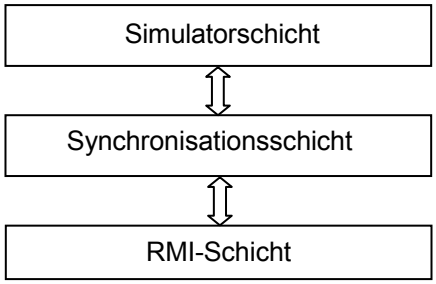


Bild 82: Schichtstruktur

Die Simulatorschicht ist der Simulator selbst. Die Aufgaben der Simulatorschicht sind wie folgt:

- Durchführung der Simulation des Modells.
- Die Simulatorschicht stellt Schnittstellen (Methoden), die den Zugriff darunter liegender Schichten auf die Simulatorschicht ermöglichen, zur Verfügung.
- Beim Einsatz von MATLAB wird JNI verwendet, damit MATLAB mit der darunter liegenden Schicht kommunizieren kann.

Die Synchronisationsschicht ist dafür verantwortlich, die Simulatoren zu synchronisieren. Man kann ihre Aufgaben wie folgt definieren:

- Diese Schicht kann sich für ein konservatives oder optimistisches Verfahren für die Synchronisierung der Simulatoren entscheiden, außer beim Einsatz von MATLAB in der Simulation. Hier wird das konservative Verfahren verwendet, da MATLAB keine Rücksetzung durchführen kann.
- Für den Datenaustausch zwischen dieser Schicht und der RMI-Schicht wird ein Stellvertreterobjekt (Proxy) verwendet, da die beiden Schichten nebenläufig laufen (Threads).

Die RMI-Schicht ist dafür verantwortlich, dass die Simulatoren miteinander kommunizieren können. Man kann ihre Aufgabe wie folgt definieren:

- Diese Schicht implementiert die Remote-Schnittstelle, die vom Client als Remote-Objekt angesehen wird. Sie stellt den oberen Schichten Methoden und Klassen zur Verfügung, um Nachrichten zur Remote-Schicht eines anderen Simulators zu senden.

Das Rapid-Prototyping-System RP.2002 [RaPr02] sowie FPGA Advantage von Mentor Graphics werden als Evaluierungsplattform für die Entwicklung von FPGAs eingesetzt. Der CORE-Generator von Xilinx stellt parametrisierbare COREs zur Verfügung, die für Xilinx-FPGAs optimiert sind. Eine Eigenschaft von RP.2002 ist der umfangreiche Einsatz von Standard-HW-Komponenten (Commercial Off The Shelf, COTS). Dies führt nicht nur zu geringen Kosten, das System kann auch einfach in ein bestehendes Netzwerk integriert werden und eignet sich für eine verteilte Simulation. Eine Backplane integriert einen PCI-Controller und ein FPGA und erlaubt den Zugriff auf externe Auslöser und Sensoren. Als BS wird Linux eingesetzt, zusammen mit der Echtzeitschnittstelle RTAI (Real Time Application Interface). Durch spezielle Blocksätze wie SystemGenerator (Xilinx) und AccelFPGA (AccelChip) sowie das Werkzeug JVHDLGen kann VHDL-Code aus Stateflow-Modellen für die Konfiguration eines FPGAs generiert werden; dabei werden auch nebenläufige Charts berücksichtigt.

AccelFPGA benötigt Festkomma-MATLAB- oder Simulink-Dateien als Eingabe. Das Modell muss entweder in dieser Form entwickelt oder manuell in diese überführt werden. Bei MATLAB-Modellen kann dies automatisch mit dem `Auto Quantize`-Kommando erledigt werden. Zusätzlich müssen Compileranweisungen angegeben werden, um zu spezifizieren, welcher Teil des Modells nach VHDL konvertiert werden soll. Sie stellen dem Compiler modellspezifisches Wissen zur Verfügung und bieten die Möglichkeit zur Optimierung. AccelFPGA generiert eine VHDL-Beschreibung auf Register-Transfer-Ebene vom modifizierten Simulink- oder MATLAB-Entwurf, der dann synthetisiert und simuliert werden kann. Stateflow-Blöcke werden nicht unterstützt. Ähnliche Einschränkungen existieren für SystemGenerator. Deshalb wird JVHDLGen eingesetzt, zusammen mit den Werkzeugen von Mentor Graphics und Xilinx.

4.3 Software-Kernkomponenten

4.3.1 Logiksimulator JLogSim

JLogSim ermöglicht die Simulation von logischen Gattern, einschließlich Flipflops. Die Verzögerungszeit kann für jede Komponente festgelegt werden und kann auch null sein. JLogSim ist mit MATLAB-mdl-Dateien (Logikblöcken) und ISCAS'89-Netzlisten [BrBK89], die für Benchmarkmessungen verwendet werden können, kompatibel. In einer früheren Arbeit wurde eine Simulation-Backplane (SimBa) unter C++ für SunOS entwickelt [ScTM95]. SimBa ermöglichte es, mehrere CASE-Werkzeuge mit Hilfe einer zeitgesteuerten Lock-Step-Synchronisation miteinander zu koppeln. Ferner wurde ein Logiksimulator (oLogic) in C++ entwickelt, der als Basis für die Forschungsarbeiten zu prädiktivem Optimismus diente [ScTM97]. JLogSim ist eine Weiterentwicklung von oLogic, von dem zwei Versionen existieren: die Standalone-Version und die in SimBa integrierte Version mit Zusatzfunktionen (u. a. für Netzwerkkommunikation) [Stre97]. Die Codegröße wurde aufgrund der durch Java zur Verfügung gestellten Methoden wesentlich reduziert und die UNIX-Socket-Netzwerkschnittstelle [Stev92] durch Java-RMI ersetzt.

Hinsichtlich des Parse-Prozesses einer Logikschaltung (bestehend aus 8191 AND-Blöcken), ist der Aufwand von JLogSim O(n); durch die Verwendung einer Standard-Hashtabelle beträgt die Beschleunigung, verglichen mit oLogic ($O(n^2)$), Faktor 29. Hinsichtlich der Ereignisverarbeitung ist der Aufwand von JLogSim O(log(n)) durch die Verwendung eines geordneten Binärbaums, im Gegensatz zum Aufwand von O(n) von oLogic durch das lineare Einfügen in eine verkettete Liste. Als Beschleunigung resultiert Faktor 9 für eine Standalone-Simulation ohne den Einsatz eines optimistischen Algorithmus. Jedem Block wird zusätzlich eine Verzögerung (Delay) zugeordnet. Dadurch wird beim Wertewechsel am Eingang der entsprechende Ausgangswert erst mit der jeweiligen Verzögerung geschrieben. Beim optimistischen Synchronisationsverfahren können die Ereignisse optional auch inkrementell gespeichert werden. Ein Block wird nach folgendem Schema beschrieben:

```
Blockname = BlockTyp(Eingang1[, Eingang2, ...]) ['Delay] [:Netzangabe]
```

`Delay` ist eine Zahl größer oder gleich 0. Alle Blöcke haben das Standard-`Delay` 1. Als Schaltungselemente (Blocktypen) stehen zur Verfügung: `AND`, `NAND`, `OR`, `NOR`, `NOT`, `DFF`, `INPUT`, `INPUT_NET`, `OUTPUT.` Der `INPUT`-Block dient als Stimuli am Eingang einer Schaltung und realisiert je nach Konfiguration eine von zwei möglichen Funktionalitäten. Im ersten Fall entspricht er einem Pulsgenerator: er wechselt seinen Wert von 0 nach 1 oder umgekehrt und stößt die Simulation durch das erzeugte Ereignis immer wieder neu an. Im zweiten Fall werden über den `INPUT`-Block extern erzeugte Stimuli aus einer Datei gelesen und als Ereignisse der Simulation zugeführt. Der `DFF`-Block ist bei beschaltetem Takteingang ein vorderflankengesteuertes D-Flipflop, bei fehlender Takteingangsbeschaltung realisiert er ein Verzögerungsglied. Ein `OUTPUT`-Block ohne Netzangabe schreibt die Eingangssignale in eine Datei. Ein `OUTPUT`-Block mit Netzangabe sendet Signale an den `INPUT_NET`-Block im anderen Teilmodell. Die Netzangabe beschreibt die Information über die Schnittstelle mit einem anderen Teilmodell. Bei der Simulation wird ein Teilmodell einem Rechner

zugewiesen; die Netzangabe muss durch Verwendung einer Konfigurationsdatei auf einen Rechner mit Hostname und Portnummer abgebildet werden[57].

Folgende ISCAS-Beschreibung entspricht dem in Bild 83 dargestellten Schaltnetz:

```
Gen1 = Input() '4
Gen2 = Input() '3
And1 = And(Gen1, Gen2) '2
Inv = Not(Nor) '1
Gen3 = Input() '5
Nor = Nor(Gen2, Gen3) '1
And2 = And(Nor, Gen3) '3
```

`Gen1`, `Gen2` und `Gen3` sind `INPUT`-Blöcke mit einer Verzögerung von jeweils vier, drei bzw. fünf Simulationszeitschritten. `And1` ist ein `AND`-Block mit einer Verzögerung von zwei Zeitschritten und zwei Eingängen aus den Blöcken `Gen1` und `Gen2`.

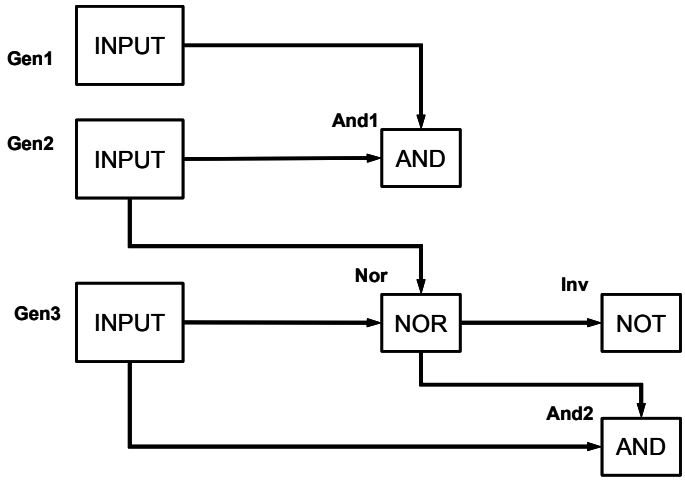


Bild 83: Logikschaltung im ISCAS-Format

4.3.2 Statechart-Simulator JStateSim

JStateSim erlaubt, analog zu Stateflow, eine verteilte ereignisdiskrete Simulation von Stateflowcharts, die ähnlich zu Harel-Statecharts sind [Hare87], bietet aber zusätzlich die Möglichkeit einer verteilten Simulation mit konservativen sowie optimistischen Protokollen [ScTM97], da die Basiskomponenten eines optimistischen Synchronisationsalgorithmus [Fuji00] [Mehl94], wie Rücksetz-Mechanismus, Zustandsspeicherung und GVT-Berechnung, implementiert sind. JStateSim ist in Java implementiert und aus diesem Grund plattformunabhängig. Die Kommunikation wird mit RMI realisiert. Es können beliebig viele Simulatorinstanzen gekoppelt werden. Transitions-Notationen sind auf die folgende formale Beschreibung eingeschränkt:

- `if` oder `if/then`
- `if := [condition]` oder `event`
- `then := (event)*` und/oder `(Gleichung)*` (getrennt durch „`;`“)
- `Gleichung := VK ° VK`
- `VK := Variable` oder `Konstante`
- `° :=` + oder –

[57] Die Konfigurationsdatei enthält außer der Zuordnung der Teilmodelle auf Rechner folgende Informationen: Zuordnung der Stimulidateien, Kennzeichnung der Debug-Ausgaben und Name der Ausgabedatei.

Um JStateSim als Teil einer verteilten Simulation mit HW-Beschleunigung einzusetzen, wurde ein Teil des Simulators modifiziert, so dass die HW vollständig vor den anderen Simulatoren und dem Kontrollprozess verborgen ist. In diesem Fall dient dem Simulator kein Modell als Eingabe, da es geladen wird, während das FPGA konfiguriert wird. Ansonsten ist das Verhalten äquivalent zu dem einer Instanz des Simulators von JStateSim, kann aber nur mit entsprechender HW (FPGAs) und SW (Bibliothek für den FPGA-Zugriff) gestartet werden. Die Kommunikation zwischen den Objektinstanzen von JStateSim sowie zwischen der CPU auf dem FPGA-RP-System und den anderen Simulatoren, die bei der Simulation beteiligt sind, wird von Java-RMI übernommen.

JStateSim wertet das Stateflowchart aus, solange Ereignisse vorliegen. Die Ereignisse können aus Stimuli-Listen stammen, von anderen Simulatoren empfangen werden, intern bei der Auswertung des Stateflowcharts entstehen oder durch einen zusätzlichen internen Trigger erzeugt werden. Dieser schreitet in der Simulationszeit um das in JStateSim kleinste Zeitinkrement von einer Simulationssekunde voran und erzeugt ein Ereignis, wenn sich der Wert an einem Systemeingang oder der Zustand des Stateflowcharts bei der vorherigen Triggerung geändert hat. Allerdings kann dadurch auch leicht eine quasi zeitgesteuerte Simulation hervorgerufen werden: dies ist der Fall, wenn sich der Zustand des Stateflowcharts bei jeder Auswertung ändert, z. B. durch bedingungsfreie Transitionen, die bei jeder Triggerung einen Zustandsübergang ermöglichen, oder wenn am Systemeingang ständig Wertänderungen auftreten. Ergeben sich zu jedem Abtastzeitpunkt Wertänderungen, wird JStateSim zeitgesteuert.

Die Struktur von JStateSim ist in Bild 84 dargestellt.

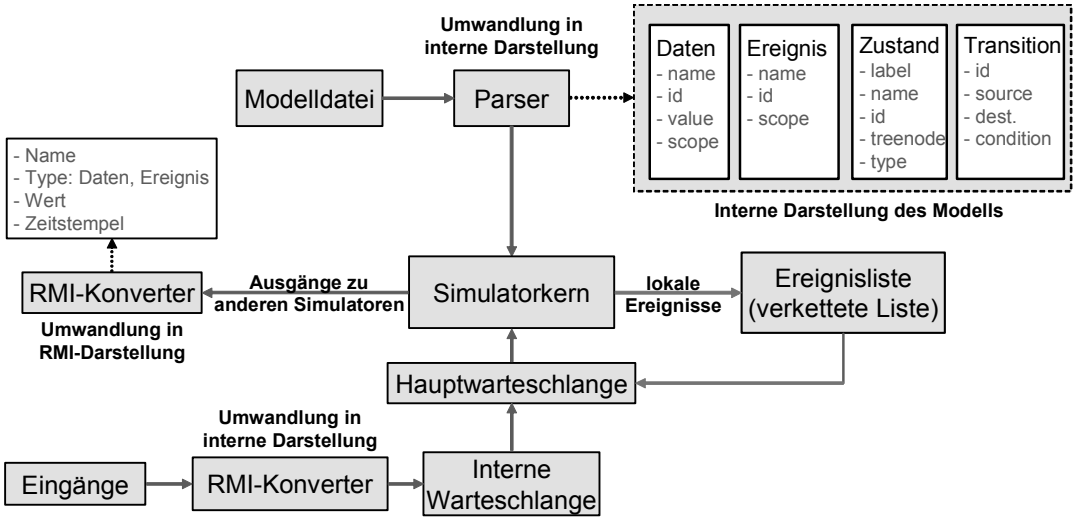


Bild 84: Struktur von JStateSim

4.3.3 Stateflowchart-Generator JStateGen

JStateGen ermöglicht die automatische Generierung von Stateflowcharts, entsprechend einer Spezifikation ihrer Struktur und ihres Verhaltens. Hierzu wird auf einen bei FZI/ESM entwickelten mdl-Parser und Generstor aufgebaut, der von Simulink erstellte Modelldateien im mdl-Format einlesen und in eine Java-Struktur überführen kann. Der Generator schreibt die Modellobjekte wieder in eine Modelldatei zurück. Die generierten Stateflowcharts sind in Simulink ausführbar.

Die von diesem Werkzeug unterstützten Simulink-Komponenten zur Modellierung sind Funktionsblöcke, Subsystem, Masked-Subsystem und Stateflow-Subsystem. Die Bibliothek von Funktionsblöcken wird durch eine XML-Datei definiert. Man kann benötigte Funktionsblöcke nachträglich selbst in diese Bibliothek aufnehmen. Ein SW-Paket bietet die Struktur von verschiedenen Objekten wie z. B. State, Transition, Event, Data und Junction eines Stateflow-Modells sowie die Funktionalität zur Platzierung von graphischen Komponenten und zur Verbindung dieser Komponenten. Dadurch kann eine von diesem Werkzeug generierte Modelldatei in Simulink graphisch dargestellt werden.

JStateGen kann folgende Operationen auf ein bestehendes Chart durchführen:

- Veränderung des Dekompositionstyps von Statecharts auf OR bzw. AND.
- Veränderung der Anzahl von Zuständen.
- Veränderung der Nebenläufigkeit des Modells: für Modelle mit gleicher Anzahl von Zuständen kann ihre Nebenläufigkeit die Simulationsdauer beeinflussen.
- Veränderung Zustandshierarchie.
- Veränderung der Abhängigkeit zwischen Zuständen: Je mehr auf gemeinsame Variablen nebenläufiger Zustände zugegriffen wird, desto größer ist der Kommunikationsaufwand.
- Beschriftung von Transitionen: mit und ohne Bedingungen, mit und ohne Transitionsaktionen.
- Beschriftung von Zuständen: mit und ohne In-State-Aktionen.
- Beliebige Vervielfachung eines Zustands einschließlich Transitionen und Daten: Ein Chart kann einem hierarchisch übergeordneten Zustand zugeordnet, mit JStateGen vervielfacht und nebenläufig verschachtelt werden (Bild 85).

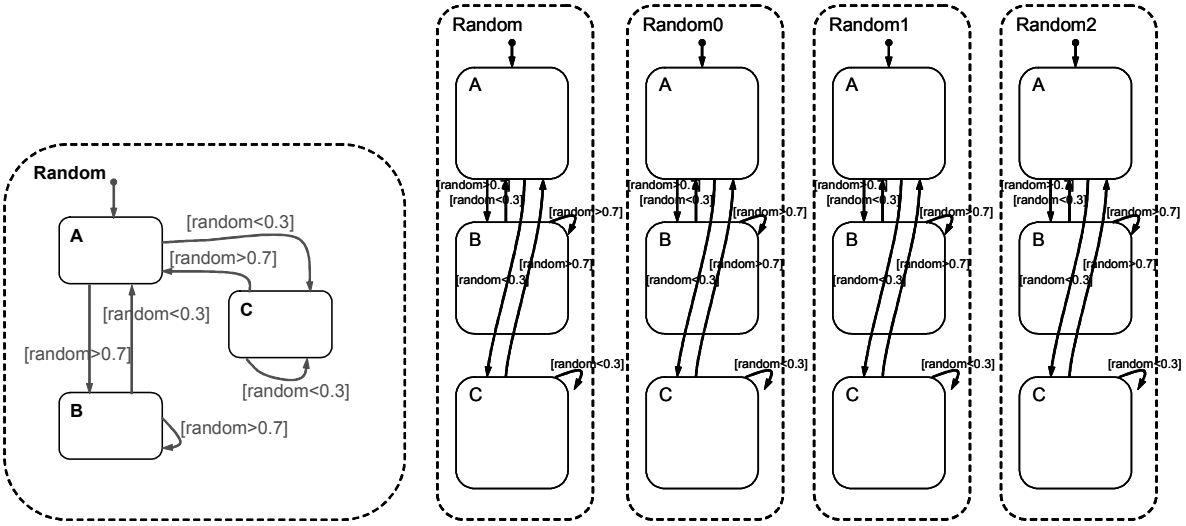


Bild 85: Verfielfachung eines Charts

- Automatische Generierung eines endlos laufenden Statecharts: JStateGen kann ein endlos laufendes Statechart beliebiger Länge erstellen. Hierzu werden Zustände erstellt, die mit Transitionen so verbunden werden, dass sie einen Kreis bilden. Diese Funktionalität kann zur Erstellung großer Statecharts mit OR-Dekomposition verwendet werden (Bild 86).

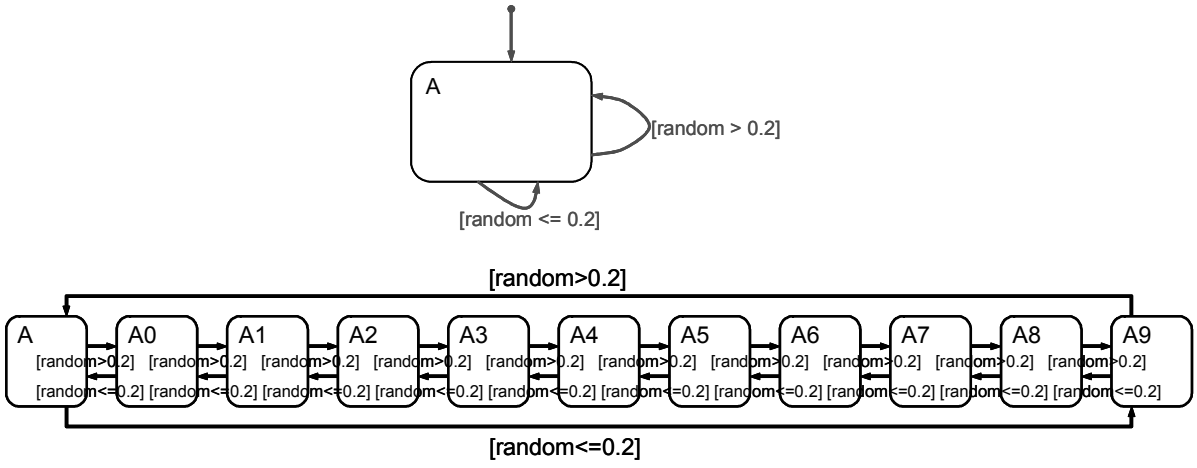


Bild 86: Generierung eines endlos laufenden Statecharts

Mit dem folgenden Befehl wird ein Modell mit 500 endlos nacheinander laufenden Zuständen erzeugt:

```
java –jar jstategen.jar -i einstate.mdl -o s500_snake.mdl -snakes 499 -w e -cd p
```

- Verkapselung von Zuständen in einen übergeordneten Zustand: Hiermit lässt sich der momentane Stand der Generierung in einen übergeordneten Zustand kapseln, so dass sich z. B. einheitliche Hierarchien erstellen lassen (Bild 87).

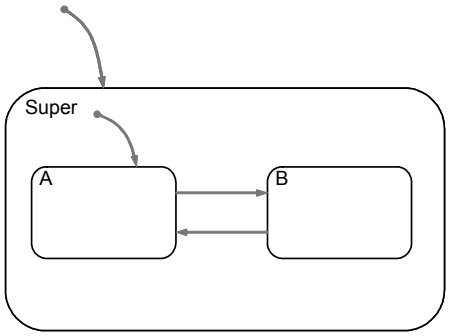


Bild 87: Verkapselung von Zuständen

Bei der Generierung großer Statecharts mit mehr als 10.000 Zuständen (siehe Tabelle 6) muss die maximale Größe von Speicher für die Java-VM durch den Parameter `Xmx` neu zugewiesen werden, da sonst der Fehler `java.lang.OutOfMemory` auftritt. Für die Generierung eines Modells mit 100.000 Zuständen laut der Befehl z. B.

```
java –Xmx128m –jar jstategen.jar –i s500.mdl –o s100000_p200_h0.mdl
-c 199 -w p -cd e
```

Anzahl von Zuständen	Xmx
20.000	128 MB
50.000	256 MB
100.000	512 MB
200.000	1 GB
300.000	1,5 GB

Tabelle 6: Speicherbedarf für JStateGen

Für die Erstellung eines Statecharts mit 300.000 Zuständen mit JStateGen wird ein Rechner mit mindestens 2 GB Arbeitsspeicher benötigt.

4.3.4 VHDL-Code-Generator JVHDLGen

In Bild 161 ist ein exemplarisches Stateflowchart dargestellt, aus dem JVHDLGen VHDL-Code generieren kann. JVHDLGen basiert auf demselben Parser wie JStateGen.

In Statecharts können nebenläufige Zustände miteinander durch Broadcast und gemeinsame Daten kommunizieren. In VHDL wird Nebenläufigkeit durch Prozesse unterstützt, die durch Signale miteinander kommunizieren können. In VHDL'93 kann der Datentyp `shared variable` zur Kommunikation verwendet werden sowie die Anweisung `wait`, um Prozesse zu synchronisieren. Eine alternative Methode ist es, nebenläufige Zustände in einen Prozess einzufügen, der in einem Taktzyklus ausgeführt wird: Ein Aufzählungstyp wird für jeden nebenläufigen Zustand definiert. Die möglichen Werte dieses Typs sind die Namen der Basiszustände. Zusätzlich wird ein virtueller Zustand hinzugefügt, um den Quellzustand der Default-Transition zu repräsentieren. Ein Zustand innerhalb des virtuellen Zustands bedeutet, dass dieser Zustand ebenfalls inaktiv ist. Es werden alle Default-Transitionen der nebenläufigen Zustände ausgeführt, wenn der entsprechende Zustand betreten wird. Wenn der Zustand mit der AND-Dekomposition verlassen wird, werden alle nebenläufigen

Unterzustände auf den entsprechenden virtuellen Zustand gesetzt, so dass beim nächsten Eintritt in den Zustand mit AND-Dekomposition immer die Default-Transitionen ausgeführt werden [Para04].

4.3.5 Kontroll-Prozess JSimControl

JSimControl ist ein Java-Programm zur Konfiguration und Steuerung einer verteilten Simulation. Der Kontrollprozess kann sowohl die konservative als auch die optimistische Simulation steuern. Bei der optimistischen Simulation berechnet er die GVT.

Der Kontrollprozess verbindet sich mit jedem Simulator über einen Kanal, der die Kommunikation mit dem Simulator übernimmt. Er kann Steuerbefehle zu den anderen Simulatoren abschicken: INIT, START, PAUSE, CONTINUE, STOP und QUIT. Eine graphische Benutzeroberfläche ermöglicht eine komfortable Anwendung. Einstellbar sind u. a. Rechnername, Modelldatei, Simulationsart, E/A-Konfiguration, Stimuli-Dateien und Simulatortyp (Bild 88).

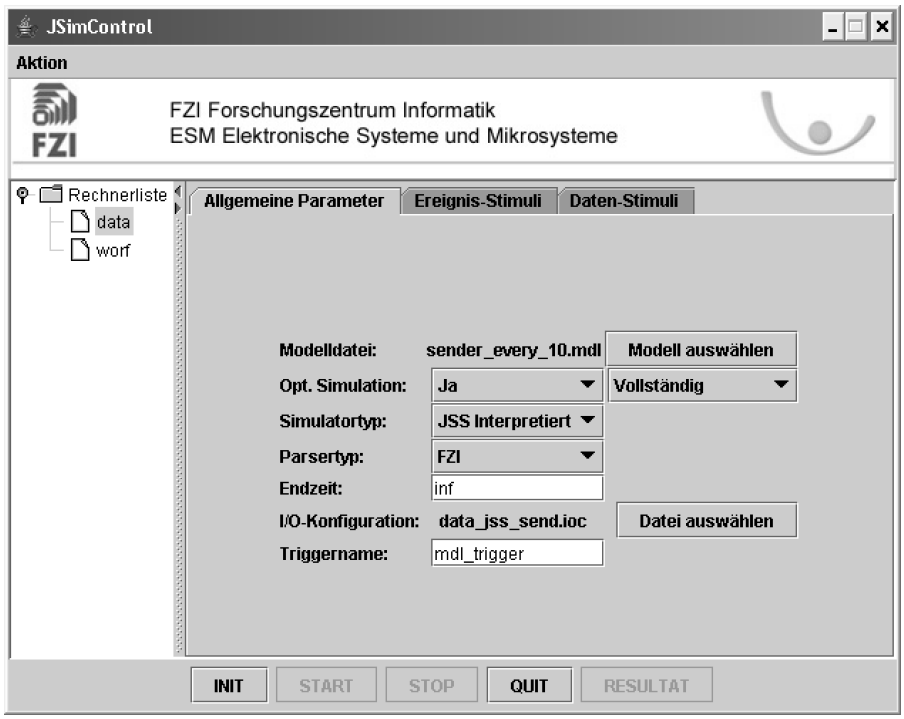


Bild 88: Benutzeroberfläche von JSimControl

4.4 Kopplung mehrerer Simulator-Instanzen

Es wurde eine Methodik zur verteilten Ausführung von Modellen auf mehreren Mikrocontrollern und einem FPGA entwickelt [DrWM04] [Wen04a]. Mithilfe eines FPGA kann die Systemperformance durch Parallelverarbeitung wesentlich erhöht werden. Im Blickpunkt stehen dabei hybride elektronische Systeme, die sowohl zustandsbasierte als auch kontinuierliche Modellteile enthalten. Für die Messung der Systemperformance wurde eine Methode zur Analyse des Zeitverhaltens und Optimierung des Task-Schedulings entwickelt, die eine graphische Repräsentation des Ausführungszeitintervalls und der Ausführungszeitpunkte der Tasks sowie die Erkennung von Leerlaufzeiten, ermöglicht. Der Datenaustausch ist mittels CAN [CANB91] realisiert. Da die Kommunikation der Geräte untereinander einen integralen Bestandteil der Gesamtfunktionalität der Systeme darstellt, muss ihre Systemeinbindung während der gesamten Entwicklungsphase begleitet werden. Dies gilt umso mehr, da heute Systemelemente unterschiedlichster Ausprägung – wie z. B. µCs und FPGAs – in derartige Gesamtsysteme eingebunden sind.

Die bestehende Lösung zur Co-Simulation zwischen Simulink und dem Mikrocontroller MPC555 über einen CAN-Bus wurde um weitere Mikrocontroller-Knoten sowie die Integration eines FPGA über CAN in diese Co-Simulations-Umgebung erweitert (Bild 89).

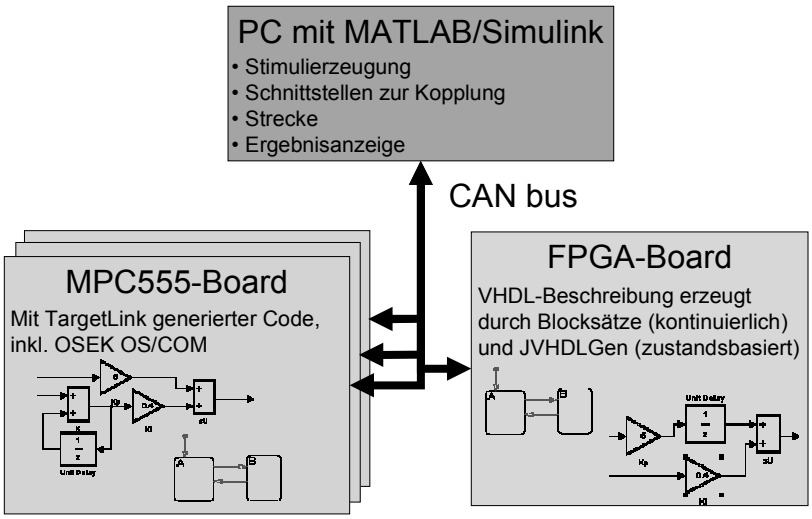


Bild 89: Erweiterte Co-Simulations-Umgebung

4.5 Partitionierungsverfahren für Statecharts

Um eine verteilte Simulation ausführen zu können, muss das Modell in mehrere nebenläufige Teilmodelle aufgeteilt werden. Jedes Teilmodell wird durch einen eigenen Simulator ausgeführt. Ziel der Partitionierung ist es, parallele Ausführbarkeit zu maximieren und gleichzeitig den Kommunikationsaufwand zu minimieren. Weiterhin werden der statische Lastenausgleich und die jeweilige Synchronisation der Simulatoren berücksichtigt [Mehl94]. Da Statecharts eine komplexe Semantik besitzen, können sie nicht direkt in Graphen konvertiert werden, wie das bei Logikschaltungen der Fall ist. Mögliche Partitionen, der Rechenaufwand pro Partition und die Kommunikation zwischen den Partitionen müssen zunächst analysiert werden. Dann werden die Statecharts als Hypergraphen modelliert, so dass Partitionierungsalgorithmen, die sich für Logikschaltungen als nützlich erwiesen haben, auch für Statecharts angewendet werden können.

- **Definition Hypergraph:** Ein Hypergraph G = (V, E) besteht aus einer Menge von Knoten V und einer Menge von Hyperkanten [Leng90]. Eine Hyperkante ist eine Teilmenge der Knoten V. Die Hyperkante e ist verbunden mit dem Knoten v, wenn $v \in e$. Der Grad eines Knotens v ist die Anzahl der mit v verbundenen Hyperkanten. Ein Graph $G' = (V', E')$ ist ein Teilgraph von G, wenn $V' \subseteq V$ und $E' \subseteq E$.
- **Definition Graph:** Wenn jede Hyperkante von G nur zwei Knoten enthält, spricht man von einem Graph. Wenn (v, w) eine Kante ist, heißen die Knoten v und w benachbart. Ein Pfad p von Knoten v_0 nach Knoten v_k in G ist eine Folge von Knoten $(v_0, v_1, ..., v_k)$ mit $v = v_0$, $w = v_k$ und $(v_i, v_{i+1}) \in E$ für $0 \leq i < k$. Wenn ein Pfad von Knoten v_0 nach Knoten v_k existiert, ist Knoten v_k von Knoten v_0 erreichbar. Ein Knoten ist immer von sich selbst erreichbar. Wenn $v_0 = v_k$ gilt, heißt p ein Zyklus. Je nach Problemstellung können die Paare von Knoten geordnet oder ungeordnet sein. Im ersten Fall wird der Graph gerichteter Graph genannt, im zweiten Fall ungerichteter Graph. Ein gerichteter azyklischer Graph (directed acyclic graph, DAG) ist ein gerichteter Graph, der keine Zyklen enthält.

4.5.1 Partitionierungsverfahren für Logikschaltungen

Vor der Partitionierung einer Logikschaltung wird seine Netzliste in Graphen modelliert. Die Partitionierung von Logikschaltungen kann als Partitionierung von Graphen oder Hypergraphen realisiert werden. Die Partitionierung eines Hypergraphs wird in [Leng90] formalisiert. Wenn eine gegebene Netzliste sequenzielle Komponenten enthält (z. B. Flipflops), müssen diese zuerst entfernt werden. Nach der azyklischen k-way-Partitionierung werden diese Komponenten sowie die Verbindungen zwischen sequenziellen Komponenten und Partitionen geeigneten Partitionen auf Basis derselben Rechenlast zugewiesen.

4.5.1.1 Modellierung als Graph und Hypergraph

Da die betrachteten Partitionierungsalgorithmen für Graphen entworfen wurden, müssen Logikschaltungen vor der Partitionierung in Graphen umgesetzt werden. Für die Partitionierung von Logikschaltungen wird ein Logikblock durch Knoten, und eine Verbindung zwischen Logikblöcken mit gerichteten Hyperkanten modelliert. Die primären Ein- und Ausgänge werden auch durch Knoten repräsentiert. In Bild 90 ist eine Logikschaltung mit vier Gattern in Simulink modelliert. Der entsprechende Hypergraph ist in Bild 91dargestellt.

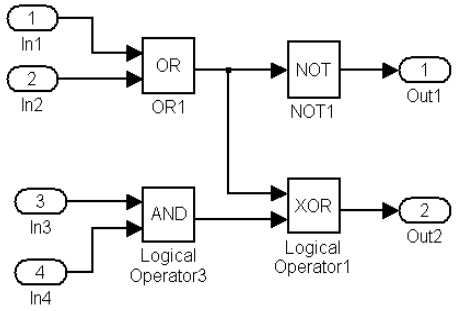


Bild 90: Logikschaltung mit vier Gattern

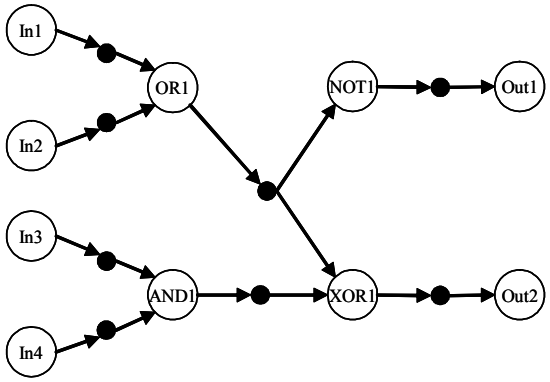


Bild 91: Gerichteter Hypergraph der Logikschaltung von Bild 90

Für eine Hyperkante wird ein zusätzlicher (durch einen schwarzen Punkt markierter) Knoten hinzugefügt. Die gerichteten Kanten zwischen den Knoten eines Logikgatters und dem Knoten einer Hyperkante repräsentieren die Ein- und Ausgabeknoten dieser Hyperkante.

4.5.1.2 Partitionierung von Modellen im ISCAS-Format

Um eine Logikschaltung, die im ISCAS-Format vorliegt, zu partitionieren und auch mit Simulink zu simulieren, muss sie zuerst ins mdl-Format umgewandelt werden. Nach der Partitionierung werden die Teilmodelle im mdl-Format gespeichert. Die Informationen über die Schnittstelle zwischen Teilmodellen werden bei der Partitionierung hinzugefügt. Um die Teilmodelle im mdl-Format mit JLogSim verteilt zu simulieren, müssen sie in `JLSchaltnetz`-Objekte von JLogSim konvertiert werden. In Bild 92 ist dieser Vorgang dargestellt.

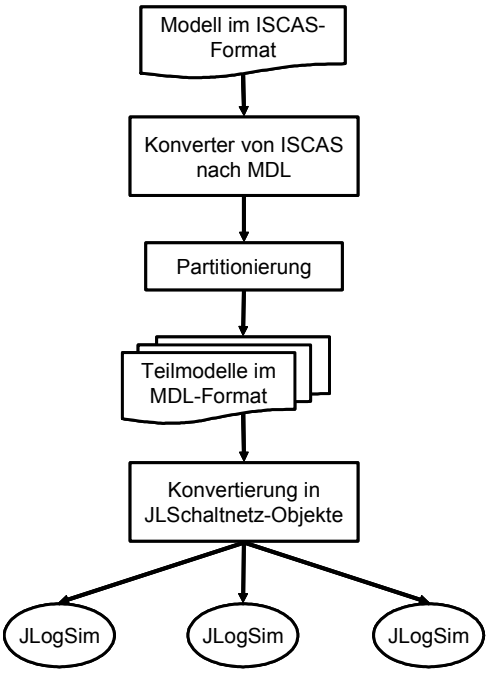


Bild 92: Simulation von Logikschaltungen im ISCAS-Format

4.5.2 Anforderungen an den Partitionierungsalgorithmus

4.5.2.1 Nebenläufigkeit eines Modells

Die Nebenläufigkeit einer verteilten Simulation hängt von der Nebenläufigkeit des Modells ab. Eine Stateflowmachine kann mehrere Charts beinhalten. Innerhalb von Statecharts hängt die Nebenläufigkeit von der Zustandshierarchie ab. In Bild 93 ist eine Stateflowmachine mit zwei Chartblöcken dargestellt. Die beiden Charts sind in Bild 94 und Bild 95 dargestellt.

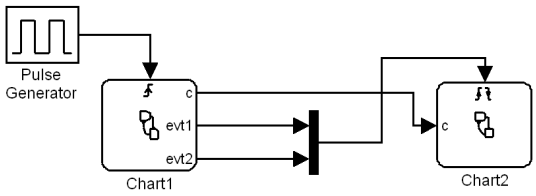


Bild 93: Stateflowmachine mit zwei Chartblöcken

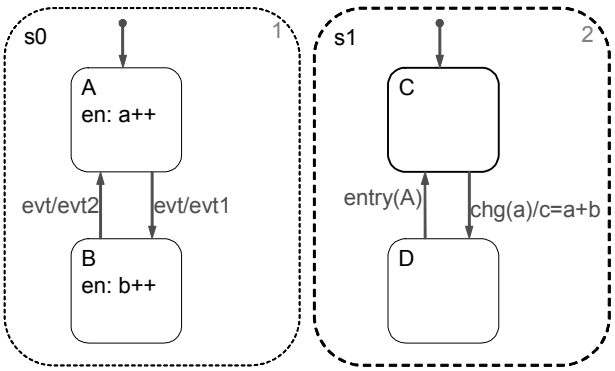


Bild 94: Chart1 der Stateflowmachine von Bild 93

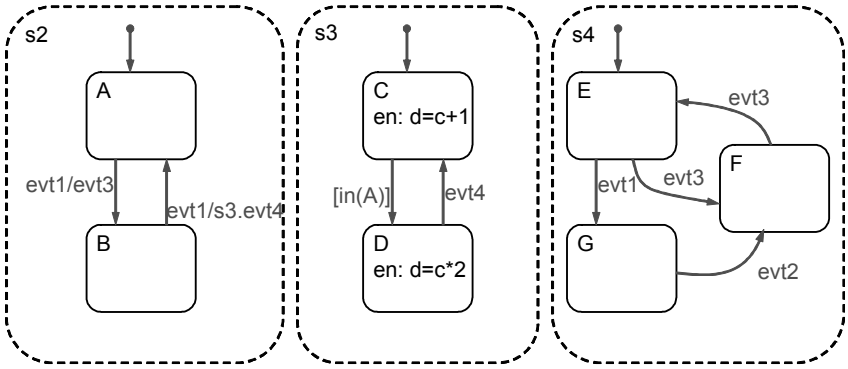


Bild 95: Chart2 der Stateflowmachine von Bild 93

Die Zustandshierarchie kann als Baum repräsentiert werden, wobei der Wurzelknoten der Knoten der Stateflowmachine ist. Die Nachfolger sind die Knoten für die Charts. Die Nebenläufigkeit eines Charts ist als die Anzahl der Unterzustände des Charts definiert, wenn das Chart ein Zustand mit UND-Dekomposition ist. Wenn das Chart ein Zustand mit XOR-Dekomposition ist, ist seine Nebenläufigkeit zu eins definiert, d. h. dieses Chart kann nur auf einem Simulator simuliert werden. Die Nebenläufigkeit eines Modells ist die Summe der Nebenläufigkeiten aller seiner Charts. Die maximale Anzahl von Partitionen eines Modells wird durch seine Nebenläufigkeit festgelegt. Da nur ein Unterzustand in einem Zustand mit XOR-Dekomposition gleichzeitig aktiv sein kann, wird die Partitionierung eines Zustands mit XOR-Dekomposition keinen Parallelismus produzieren. Die mögliche Partition kann entweder ein Chart oder ein Unterzustand eines Charts sein, abhängig davon, ob das Chart eine XOR- oder eine UND-Dekomposition besitzt. Bild 96 illustriert die Zustandshierarchie des in Bild 93 dargestellten Modells.

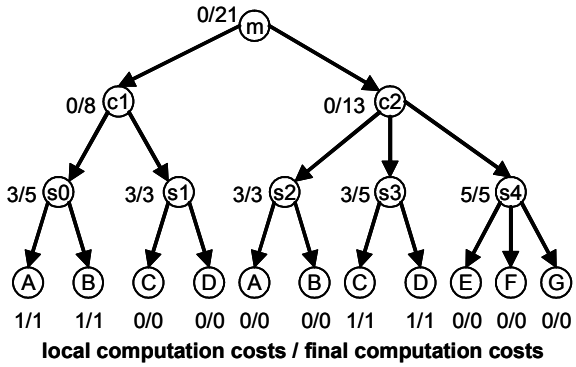


Bild 96: Zustandshierarchie eines Modells mit Nebenläufigkeit fünf

Der Knoten `m` repräsentiert die Stateflowmachine; die Knoten `c1` und `c2` repräsentieren die Charts `Chart1` und `Chart2`. Die Nebenläufigkeiten der Statecharts `Chart1` und `Chart2` sind deshalb zwei und drei, und die Nebenläufigkeit dieses Modells ist fünf; d. h. dieses Modell kann höchstens in fünf Partitionen aufgeteilt werden. Der Berechnungsaufwand eines Modells wird hauptsächlich durch die Ausführung von Aktionen und Transitionen von Zuständen verursacht. Es wird angenommen, dass alle Transitionen und Zustandsaktionen mit derselben Häufigkeit ausgeführt werden, der Rechenaufwand jeder Transition derselbe ist und die Zustandsaktion dieselbe ist. Der Rechenaufwand eines Zustands ist als die Summe der Anzahl seiner Zustandsaktionen und Transitionen definiert. Da Transitionen zwischen Zuständen auf unterschiedlichen Hierarchieebenen existieren können, wird der Rechenaufwand in zwei Schritten berechnet. Zuerst wird der lokale Rechenaufwand, definiert als die Summe von Zustands-Aktionen und der Anzahl der Transitionen zwischen direkten Unterzuständen, berechnet. Dieses Ergebnis wird zuerst als Gewicht eines Knotens im Zustandsbaum gekennzeichnet. Der endgültige Rechenaufwand eines Zustands wird durch Akkumulation der endgültigen Kosten seiner direkten Unterzustände und seiner lokalen Kosten berechnet. Diese Berechnung wird von den Blättern zur Wurzel durchgeführt. Für die Blätter entsprechen die lokalen Kosten auch den

endgültigen. Bild 96 zeigt, wie der Rechenaufwand des Modells von Bild 93 berechnet wird. Da der Wurzelknoten keine Zustandsaktionen und keine Transitionen enthält, sind seine lokalen Kosten null. Die Unterzustände `A` und `B` von Zustand `s0` beinhalten jeweils eine Zustandsaktion. Der Zustand `s0` beinhaltet drei Transitionen einschließlich der Default-Transition. Der endgültige Rechenaufwand von Zustand `s0` ist die Summe der Kosten der Knoten `A`, `B` und seinen lokalen Kosten drei.

4.5.2.2 Rechenaufwand eines Modells

Weil ein Simulator das Verhalten von Statecharts simulieren muss, wird der Rechenaufwand eines Modells mit Statecharts zum größten Teil von der Bearbeitung von Zustandsaktionen sowie Transitionen verursacht. Angenommen, es werden alle Transitionen und Zustandsaktionen mit gleicher Häufigkeit bearbeitet, dann ist der Rechenaufwand jeder Transition und Zustandsaktion gleich, und der Rechenaufwand eines Zustandes wird als die Summe der Anzahl seiner Zustandsaktionen und Transitionen definiert. Die obige Annahme kann das Ergebnis der Partitionierung beeinflussen. Weil Transitionen zwischen Zuständen auf verschiedenen Hierarchieebenen auftreten können, wird der Rechenaufwand in zwei Schritten berechnet:

- Es wird zuerst ein lokaler Rechenaufwand berechnet. Der lokale Rechenaufwand wird als die Summe der Anzahl von Zustandsaktionen und der Anzahl von Zustandsübergängen zwischen direkten Unterzuständen definiert. Diese Zahl wird zuerst als Gewicht eines Knotens im Zustandsbaum markiert.
- Der endliche Rechenaufwand eines Zustands wird durch die Summierung des endlichen Aufwands seiner direkten Unterzustände und seines lokalen Aufwands berechnet. Diese Berechnung wird von den Blättern zur Wurzel sukzessiv durchgeführt. Für Blätter ist der lokale Aufwand auch ihr endlicher Aufwand.

4.5.2.3 Kommunikationsaufwand zwischen Partitionen

Die Partitionen aus Zuständen oder Charts müssen miteinander kommunizieren, wenn die Ereignisse, die von einer Partition generiert wurden, zu einer anderen Partition weitergegeben werden müssen, oder wenn zwei Partitionen auf gemeinsamen Variablen zugreifen. Dadurch unterscheiden sich die Datenobjekte, die zwischen zwei Partitionen ausgetauscht werden, von Ereignissen und Variablen. Auf ein Datenobjekt kann entweder durch eine Zustandsaktion oder eine Transition zugegriffen werden. Um die Kommunikationskosten zwischen Partitionen herauszufinden, wird ein Kommunikationsgraph für eine Partitionierung definiert. Ein Kommunikationskanal existiert zwischen zwei Partitionen `P1` und `P2`, wenn eine der folgenden Bedingungen erfüllt ist:

- Eine Variable wird von einer Zustandsaktion oder Transition in `P1` verändert, gleichzeitig ist sie Operand eines Ausdrucks einer Zustandsaktion oder Transition in `P2`.
- Ein normales oder implizites Ereignis in `P1` muss über Broadcast zu `P2` gesendet werden.
- Ein gerichtetes Ereignis einer Transition in `P1` wird zu einer anderen Transition in `P2` gesendet.
- `P1` muss `P2` über die Gültigkeit eines Zustands innerhalb von `P1` als Zustandsbedingung informieren.

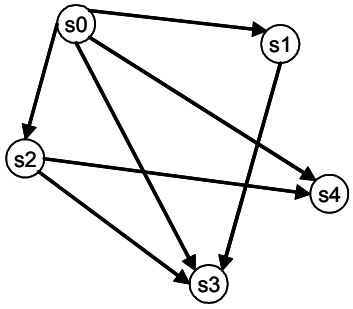


Bild 97: Kommunikationsgraph zwischen Partitionen

Die Knoten des Kommunikationsgraphen repräsentieren eine mögliche Partition. Wenn eine der oben spezifizierten Bedingungen erfüllt ist, wird als Kommunikationskanal eine gerichtete Kante von `P1` nach `P2` hinzugefügt. Sie beinhaltet alle Datenobjekte, die von `P1` nach `P2` gesendet werden sollen. Es wird angenommen, dass alle Datenobjekte mit derselben Frequenz zwischen den Partitionen ausgetauscht werden. Der Rechenaufwand des Austausches zwischen zwei Partitionen in eine Richtung wird als die Anzahl unterschiedlicher Datenobjekte definiert. Bild 97 zeigt den Kommunikationsgraph des Modells des vorangegangenen Abschnitts.

4.5.3 Auswahl und Anpassung des Partitionierungsalgorithmus

4.5.3.1 Erstellung des Hypergraphs für ein Stateflowchart-Modell

Bei der Partitionierung eines Modells für Simulink bestehen einige Einschränkungen:

- Es darf keine Rückkopplung zwischen Charts auftreten, da in Simulink diese nicht erlaubt sind. Es kann eine Rückkopplung auftreten, wenn es eine Verzögerung gibt [Simu04]. Aber dies würde das Verhalten der Teilmodelle ändern und das Endergebnis würde dann nicht mehr mit dem Hauptmodell übereinstimmen.
- Eine Variable darf nur auf einem Chart geändert und auf den anderen Charts gelesen werden, da der Eingang an einem Chart nur lesbar ist. Das Chart darf den Wert des Eingangs nicht ändern, da dieser aus anderen Charts oder anderen Simulink-Blöcken stammt [Stat04].

Diese Punkte werden bei dem verwendeten Partitionierungsprogramm, das in der Arbeit [Zhan03] entwickelt wurde, berücksichtigt.

4.6 Optimistische Co-Simulation mit RTW-Code

Es wurde ein Verfahren entwickelt, mit dem es möglich ist, eine Co-Simulation hybrider elektronischer Systeme unter Einsatz optimistischer Synchronisationsverfahren durchzuführen [DrWM05] [Wen04b]. Für die optimistische Simulation zustandsdiskreter Teilmodelle dient der Simulator JStateSim (4.3.2) und für die zentrale Simulationsstatus-Überwachung und -Steuerung JSimControl (4.3.5) (Bild 98).

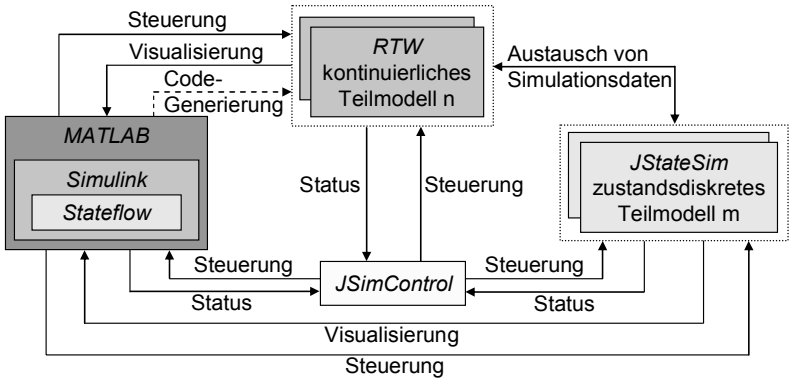


Bild 98: Optimistische Co-Simulation mit RTW-Code

Der in [FrCW98] und [FrRa00] vorgestellte Synchronisationsalgorithmus SES zur verteilten, optimistischen Simulation hybrider Systeme wird im Rahmen der vorliegenden Arbeit mit JStateSim als ereignisdiskretem (DES) und RTW als kontinuierlichem (CS) Simulator implementiert. Es werden zunächst einige Aspekte angesprochen, die in [FrCw97] [FrCW98] und [FrRa00] zum Teil implizit oder gar nicht formuliert sind. Es handelt sich hierbei insbesondere um Einschränkungen hinsichtlich der Partitionierung, die für eine korrekt ablaufende Simulation und einen effizienten Einsatz der SES gemacht werden müssen. Weiterhin wird auf die Problematik der Schwellwertüberschreitung eingegangen, die in [FrCW98] und [FrRa00] wenig thematisiert wird. Außerdem wird ein Konzept entwickelt, das eine flexible Handhabung von Schwellwertüberschreitungen unter Minimierung des Kommunikationsaufwands ermöglicht.

4.6.1 Anforderungen

Das Gesamtsystem soll erweiterbar und konfigurierbar sein. Dies erfordert u. a. eine einheitliche Kommunikationsstruktur hinsichtlich Konfiguration, Steuerdaten und Simulationsdaten, eine Schnittstelle für die Steuerung, ein einheitliches Steuerungsprotokoll, sowie Synchronisationsschnittstellen zwischen zeitgesteuerten und ereignisgesteuerten Simulatoren. Die Simulationsdurchführung darf nicht unnötig durch Kommunikation ausgebremst werden. Daher ist auf eine geringe Kommunikationslast durch minimale Datenmengen bei niedriger Senderate zu achten. Soweit es die jeweiligen Synchronisationsalgorithmen erlauben, können konfigurierbare Puffer und Zeitlimits zum Sammeln der Daten und zum verzögerten Senden eingesetzt werden. Um unabhängig von Plattform und Programmiersprache Komponenten einbinden zu können, wird als Kommunikationsprotokoll TCP/IP verwendet.

Um eine Anbindung des Simulators kontinuierlicher Systeme an das INTERACT-Framework durchführen zu können, muss es möglich sein, auf simulatorinterne Strukturen und Abläufe Einfluss zu nehmen. Notwendig ist die Möglichkeit, auf den jeweiligen Systemzustand und die Simulationszeit lesend und schreibend zugreifen zu können, da nur so eine optimistische Simulation mit Rücksetzung realisierbar ist. Es soll möglich sein, bei der Ausführung eines Simulink-Modells Daten über TCP/IP an angeschlossene Simulatoren zu senden und Daten von diesen Simulatoren zu empfangen und darzustellen. Die zu sendenden Daten sollen auch während der Simulation manipulierbar sein (z. B. durch einen Schalter im Modell), so dass eine Interaktion während der Simulation möglich ist. Die dargestellten Daten sollen Gültigkeit haben, d. h. bei optimistischen Simulationsverfahren werden nur die Daten dargestellt, deren Zeitstempel kleiner oder gleich der GVT ist. In einem Puffer mit konfigurierbarer Größe werden die Daten gesammelt, und gesendet, wenn der Puffer gefüllt oder ein einstellbares Zeitintervall überschritten ist. Wird die Simulation durch ein Stopp-Signal beendet, werden noch alle Daten, die von den anderen Simulatoren vor Auftreten des Stopp-Signals gesendet wurden, empfangen und dargestellt. Ein sicheres Beenden der Simulation ist so jederzeit möglich.

4.6.2 Entwurf eines optimistischen Simulations-Frameworks für RTW-Code

4.6.2.1 Kommunikation

Die Kommunikation basiert auf TCP/IP. Es werden zunächst auf dem Konfigurationskanal die relevanten Konfigurationsdaten empfangen. Auf dem Kontrollkanal werden Steuerbefehle empfangen und auf dem Datenkanal Simulationsdaten empfangen und gesendet. Der Datenkanal wird durch Verbindungen zu den als Sender und/oder als Empfänger konfigurierten benachbarten Simulatoren realisiert, wobei für jeden Simulator ein Sende- und/oder ein Empfangskanal erzeugt wird.

4.6.2.2 Elemente für optimistische Simulation

Das Simulations-Framework von RTW erhält die gleichen Funktionalitäten zur optimistischen Simulation wie bei JStateSim.

Zustandsspeicherung und Rücksetzung: Da der Simulator teilweise basierend auf hypothetischen Eingangswerten rechnet, muss es möglich sein, einen früheren Zustand wiederherzustellen, um falsche Hypothesewerte zu korrigieren und mit dem richtigen Wert weiterzusimulieren. Hierfür werden die Zustandspeicherung und der Rücksetz-Mechanismus eingesetzt. Dabei entspricht der mit Zeitstempel *t* gespeicherte Zustand dem Systemzustand nach der Durchführung des Simulationsschritts zum Zeitpunkt *t*. Daher muss bei einer Rücksetzung, der durch ein Ereignis mit Zeitstempel *t* ausgelöst wird, der Zustand mit Zeitstempel *t-1* wieder hergestellt werden, so dass der Zustand mit Zeitpunkt *t* neu berechnet werden kann.

Folgende Informationen müssen bei einer Zustandsspeicherung gesichert werden:

- Zeitstempel des gespeicherten Zustands: über den Zeitstempel wird bei einer Rücksetzung der wiederherzustellende Zustand identifiziert. Die Zustände sind dabei mit aufsteigendem Zeitstempel angeordnet in einer Liste gespeichert.
- Speicherungsart (vollständig (engl. total), inkrementell): bei inkrementeller Zustandsspeicherung werden vollständig gespeicherte Zustände genutzt, um von dort ausgehend die Veränderungen inkrementell zu speichern. Daher muss bei jedem Zustand erkennbar sein, welche Speicherungsmethode angewandt wurde.
- Zeitstempel der letzten totalen Zustandsspeicherung: Bei einer Rücksetzung wird bis zu dem letzten vollständig gespeicherten Zustand zurückgesprungen, dessen Zeitstempel kleiner als die Rücksetzungszeit ist. Von dort an werden die Zustände bis zur Rücksetzungszeit inkrementell wieder hergestellt.
- Systemeingänge (Inports) `rtU`: dies sind die Werte, die zum jeweiligen Zeitpunkt an den Systemeingängen anliegen.
- Systemausgänge (Outports) `rtY`: dies sind die Werte, die zum jeweiligen Zeitpunkt an den Systemausgängen anliegen.
- Innere Blockzustände (Blockausgänge) `rtB`: dies sind die Werte, die zum jeweiligen Zeitpunkt an den Blockausgängen des Modells anliegen.
- Der Zeitpunkt des letzten Sendevorgangs für jeden Empfänger: wird beim Senden mit Zeitlimits gearbeitet, müssen diese Werte gespeichert werden, um das Zeitlimit korrekt auswerten zu können.

Beim Eintreffen einer Nachricht mit Zeitstempel kleiner als die aktuelle Simulationszeit wird eine Rücksetzung ausgelöst. Dabei muss feststellbar sein, auf welchem Eingangsport die Rücksetzung ausgelöst wurde, da dieser Port mit dem empfangenen neuen Eingangswert aktualisiert werden muss.

Negativnachrichten: Für jede erzeugte Nachricht wird auch eine Negativnachricht generiert. Bei einer Rücksetzung werden für alle Nachrichten mit Zeitstempel größer als die Rücksprungzeit die

entsprechenden Negativnachrichten gesendet, um so die zugehörigen Nachrichten zu annullieren. Wird eine Negativnachricht empfangen, sind zwei Situationen möglich:

- Die entsprechende Nachricht wurde bereits ausgeführt: In diesem Fall muss eine Rücksetzung durchgeführt werden und nach Zustandswiederherstellung die der Negativnachricht zugehörige Nachricht gelöscht werden.
- Die entsprechende Nachricht wurde noch nicht ausgeführt: In diesem Fall kann die zugehörige Nachricht aus der Nachrichteneingangsliste direkt gelöscht werden.

GVT und Speicherverwaltung: Der GVT-Mechanismus und die zugehörige Methode der Speicherverwaltung (fossile collection) ist eine notwendige Optimierung bei Algorithmen, die mit Zustandsspeicherung arbeiten, um beim Fortschreiten der Simulation belegten Speicherplatz wieder verwenden zu können. Da auf die Zustände mit Zeitstempel kleiner als GVT nicht mehr zugegriffen werden muss, können sie gelöscht werden und so Speicher freigegeben werden. Fossile collection findet also immer im Anschluss an die GVT-Aktualisierung statt und betrifft alle Listen, die notwendige Daten für eventuelle Rücksetzungen speichern.

4.6.3 Ausgetauschte Daten

Zwischen DES und CS werden für unterschiedliche Zwecke Daten über mehrere virtuelle Kanäle ausgetauscht (Bild 99):

- Alle Ausführungszeitpunkte des DES werden dem CS durch ein jeweils über den virtuellen Aktivierungskanal übertragenes Ereignis mitgeteilt. Es lässt sich als Aktionsereignis charakterisieren, da es lediglich einen Zeitwert darstellt, der der nächsten Zielzeit der kontinuierlichen Simulation entspricht. In diesem speziellen Kontext wird im Folgenden der Begriff Aktivierungsereignis verwendet. Die Ausführungszeitpunkte des DES lassen sich aus den Zeitstempeln der abzuarbeitenden Ereignisse bestimmen. Für Ereignisse des DES, die den gleichen Zeitstempel aufweisen, sendet der DES nur ein gemeinsames Aktivierungsereignis an den CS; das Senden mehrerer Aktivierungsereignisse mit gleicher Zielzeit ändert nicht das Simulationsverhalten, sondern erzeugt nur redundanten Kommunikationsoverhead.
- Sobald sich Werte an Datenausgängen des DES, die mit dem CS verbunden sind, ändern, werden die neuen Werte mit Zeitstempel als Datenereignisse über den virtuellen Datenkanal von DES nach CS übertragen.
- Ausgangswerte des CS können als Datenereignisse mit Zeitstempel über einen virtuellen Datenkanal an den DES übermittelt werden. In 4.6.7 wird näher darauf eingegangen, unter welchen Bedingungen die Werte übertragen werden müssen und auf welche Weise sie vom DES verarbeitet werden können.

Bild 99 zeigt die virtuellen und realen Datenverbindungen zwischen CS und DES. Für jede Kommunikationsrichtung besteht eine unidirektionale TCP/IP-Verbindung.

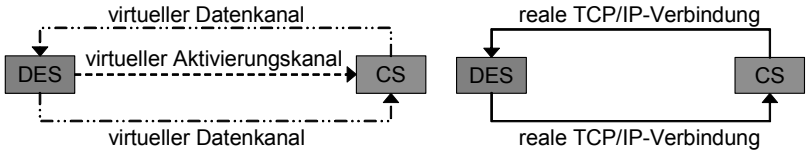


Bild 99: Virtuelle und reale Datenverbindungen zwischen CS und DES

Die virtuellen Kanäle werden durch Typisierung der über die realen TCP/IP-Verbindungen ausgetauschten Ereignisse mit Hilfe von Typ-IDs, die in Tabelle 7 aufgelistet sind, realisiert.

	Typ-ID
Von CS empfangenes Aktivierungsereignis	RTW_ACTION
Von CS empfangenes Datenereignis	RTW_DATA
Von DES empfangenes Datenereignis	JSS_DATA

Tabelle 7: Typ-IDs der ausgetauschten Daten

4.6.4 Einschränkungen hinsichtlich der Modellverteilung

Zunächst ist zu beachten, dass gemäß [FrCW98] und [FrRa00] kontinuierliche Teilmodelle als atomar angesehen werden, d. h. ein hybrides Modell darf nicht derart partitioniert werden, dass direkte Kommunikationskanäle zwischen kontinuierlichen Teilmodellen existieren. Begründet wird dies mit dem hohen Kommunikationsaufwand und der daraus resultierenden Simulationsverlangsamung, die eine derartige Aufteilung mit sich bringt. Festzuhalten bleibt, dass bei der SES ein CS nur mit DES Daten austauschen darf, nicht aber mit anderen CS.

Wie in 4.6.3 erwähnt, sind bei der Kopplung von DES und CS drei Verbindungsarten zu unterscheiden (Bild 100):

- Datenkanal von DES zu CS,
- Datenkanal von CS zu DES,
- Aktivierungskanal von DES zu CS (siehe 4.6.3).

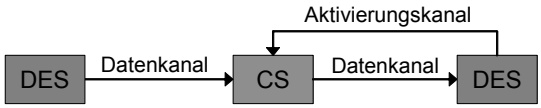


Bild 100: Verbindungsarten zwischen CS und DES

Für das Problem, wie hinsichtlich der Aktivierung zu verfahren ist, wenn ein CS mit mehreren DES verbunden ist, sind unterschiedliche Ansätze denkbar:

- Es wird ein DES festgelegt, der als einziger die Zielzeiten des CS bestimmen und diesen dadurch aktivieren kann. In diesem Fall wird es jedoch keine Synchronisationspunkte für den CS mit den anderen DES geben, was aber eine Grundvoraussetzung der SES ist.
- Wenn mehrere oder alle DES Zielzeiten an den CS senden können, ergibt sich ein Synchronisationsproblem, da die DES wegen des optimistischen Simulationsverfahrens in ihrer Simulationszeit nicht synchron voranschreiten. So ist es möglich, dass der CS durch die Aktivierung eines in seiner lokalen Simulationszeit weit vorangeschrittenen DES Simulationsintervalle mit großen Zielzeiten berechnet, bevor von einem DES mit kleinerer lokaler Simulationszeit niedrigere Zielzeiten gesendet werden. In diesem Fall erhält der CS Zielzeiten, die kleiner sind als seine momentane Simulationszeit. Da in [FrCW98] und [FrRa00] auf dieses Szenario nicht eingegangen wird, bleibt offen, wie darauf zu reagieren ist. Möglich ist z. B. eine Rücksetzung auf die niedrigere Zielzeit; dies würde im Endeffekt aber dazu führen, dass der in seiner lokalen Simulationszeit jeweils am langsamsten voranschreitende DES die Ausführungsgeschwindigkeit des CS steuert, was sich in Geschwindigkeitsverlusten auswirkt. Werden die niedrigen Zielzeiten hingegen ignoriert, so werden hierfür keine Synchronisationspunkte angelegt, was wieder eine Grundvoraussetzung der SES verletzt.

Diese Überlegungen zeigen, dass es bei Verwendung der SES ohne zusätzliche Synchronisationsmechanismen nicht sinnvoll ist, einen CS von mehreren DES aus zu aktivieren. Des weiteren kommt die Einschränkung hinzu, dass der CS nur an den ihn aktivierenden DES Daten senden darf, da nur für diesen DES die Schwellwerte bekannt sind.

4.6.5 Überlegungen zur Schwellwertprüfung

Ein wesentlicher Aspekt der Synchronisationsmethode nach [FrCW98] und [FrRa00] ist die Berücksichtigung von Schwellwertüberschreitung. Diese wurde im Rahmen der vorliegenden Arbeit genauer spezifiziert. Im Rahmen der weiter oben genannten Bedingungen lassen sich folgende Aussagen treffen, die u. a. Sinn und Zweck der Schwellwerte klären und definieren, welche Signale auf Schwellwertüberschreitung überprüft werden müssen:

(1) Es werden solche Datensignale auf Schwellwertüberschreitung geprüft, die vom CS zum DES übertragen werden.

(2) Die in (1) charakterisierten Signale stammen von einem kontinuierlichen Modell, das oft analoge Systeme repräsentiert und typischerweise komplexe, teilweise stochastische Ausgangssignale erzeugt. Daher kann davon ausgegangen werden, dass der Signalwert nur sehr selten konstant ist, und bei der überwiegenden Zahl von Simulationsschritten vom vorherigen Wert abweicht.

(3) Die in (1) charakterisierten Signale lassen sich weiter nach der Art des Eingangs, an dem sie auf DES-Seite anliegen, unterteilen: Signale, die am Triggereingang des DES anliegen, können bei entsprechender Wertänderung das diskrete System triggern. Werte von Signalen, die an einem Dateneingang des DES anliegen, werden als Operanden bei Relationsoperationen (z. B. <, >) und / oder arithmetischen Operationen (z. B. +, -) verwendet. Die Unterscheidung in Signale am Triggereingang und Signale an Dateneingängen wurde in [FrCW98] und [FrRa00] nicht gemacht, womöglich weil sie hinsichtlich der SES keine Auswirkung auf die Synchronisation und die Simulation hat. Dennoch ist sie zum besseren Verständnis und zur Verifikation des Algorithmus von Bedeutung.

(4) Wird das Ausgangssignal des CS als Triggersignal des DES verwendet, muss der DES alle Wertänderungen dieses Signals kennen, da jede Wertänderung eine Triggerung des DES bewirken kann.

(5) Bei optimistischer Simulation muss dem DES auch jede Wertänderung an seinen Dateneingängen signalisiert werden, da diese durch eine Zustandsspeicherung protokolliert werden muss, um bei einer Rücksetzung für jeden Rücksprungszeitpunkt eine korrekte Zustandswiederherstellung sicherzustellen.

(6) Aus (4) und (5) geht hervor, dass jede Wertänderung am Ausgang des CS signalisiert werden muss, unabhängig davon, ob der Wert beim DES am Trigger- oder Dateneingang anliegt. Signale, wie sie in (2) beschrieben sind, können im schlimmsten Fall bei jedem Ausführungsschritt des CS ihren Wert ändern. In diesem Fall kann es ohne weitere Maßnahmen zu einer starken Abbremsung der Simulation kommen: so kann ein stark dynamisches Ausgangssignal des CS, das auf den Triggereingang des DES geschaltet ist, dazu führen, dass der DES zu jedem Simulationszeitpunkt des CS getriggert wird und somit quasi zeitgesteuert simuliert. Optimistische Simulation basiert auf Annahmen hinsichtlich der Werte an den Eingängen der Simulatoren. Die Quote für gültige Hypothesewerte kann, abhängig vom verwendeten Verfahren zu deren Bestimmung, bei stark dynamischen Eingangssignalen auf Null sinken, was an jedem Zeitschritt eine Rücksetzung notwendig macht. Zusätzlich kommt es zu einem großen Kommunikationsaufwand, der die Gesamtsimulation weiter ausbremst. Diese Ausführungen machen deutlich, dass zusätzliche Maßnahmen notwendig sind, um eine effiziente optimistische Co-Simulation hybrider Systeme durchführen zu können.

(7) Durch den Einsatz eines Schwellwerts können die in (6) genannten Schwierigkeiten ausgeräumt werden: es kommt zu einer Reduzierung der auszutauschenden Ereignissen, was dazu führt, dass die Zeitsteuerung aufgehoben, die Zahl der Rücksetzungen verringert und die zu übertragende Datenmenge eingeschränkt wird. Definition und Einsatz des Schwellwerts wird in den folgenden Punkten beschrieben.

(8) Bezüglich des Eingangssignals eines beliebigen Systems kann durch Festlegung einer oberen und unteren Toleranzgrenze ein Toleranzband eingeführt werden, innerhalb dessen das Eingangssignal variieren darf, ohne dass dem System eine Änderung des Eingangswerts signalisiert wird.

Diese Toleranzgrenzen sind gleichbedeutend mit dem zitierten Schwellwert. Es gibt demnach eigentlich zwei Schwellwerte: der eine bildet die Toleranzgrenze nach oben, der andere diejenige nach unten. Die Toleranzgrenzen werden in Relation zum jeweils letzten bekannten Eingangswert ausgewertet; dieser dient also als Referenzwert. Dabei wird zwischen absoluter und relativer Toleranz unterschieden (siehe Bild 101 und Bild 102). Obere und untere Toleranzgrenzen können in Art und Wertigkeit unabhängig voneinander festgelegt werden.

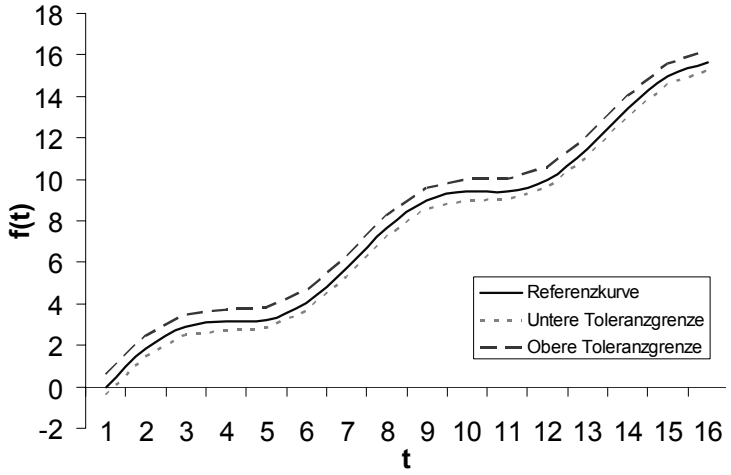


Bild 101: Toleranzband mit Referenzkurve und absoluten Toleranzgrenzen

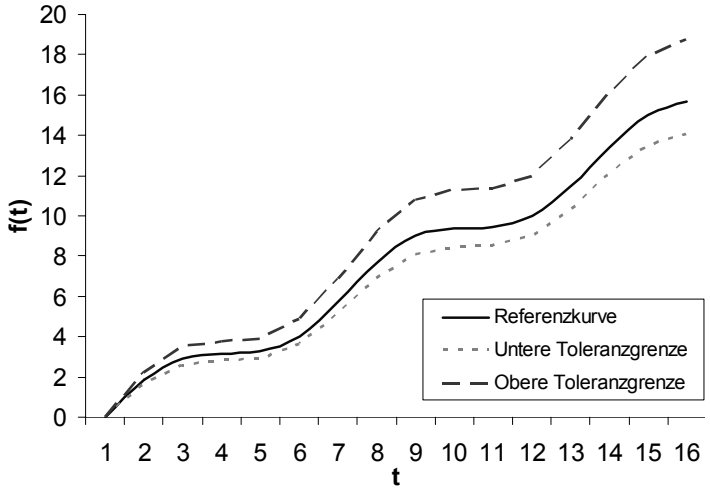


Bild 102: Toleranzband mit Referenzkurve und relativen Toleranzgrenzen

(9) Unterschiedliche Referenzkurven für das Toleranzband können bei gleicher Toleranzbandbreite bei einem vorgegebenen Signalverlauf zu einer unterschiedlichen Zahl von Schwellwertüberschreitungen führen (siehe Bild 103 und Bild 104). Durch die Wahl einer dem Signalverlauf ähnlichen Referenzkurve lässt sich die Zahl der Überschreitungen deutlich reduzieren.

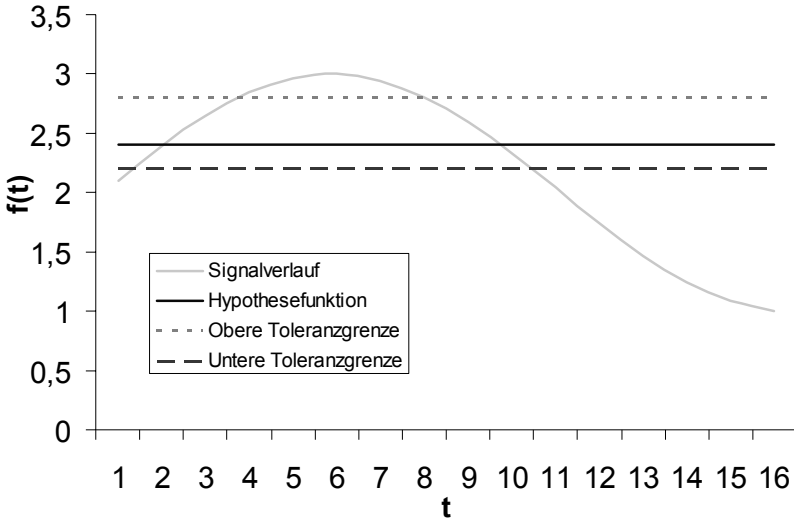


Bild 103: Ungünstige Referenzkurve für vorgegebenen Signalverlauf

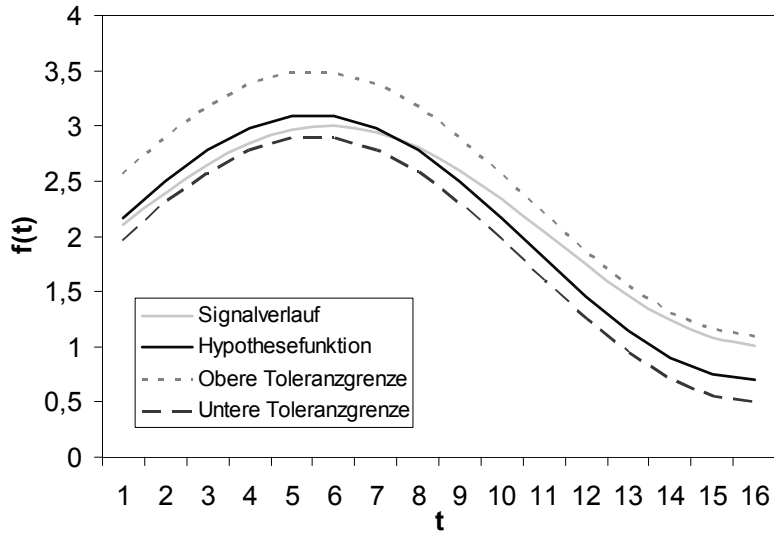


Bild 104: Günstige Referenzkurve für vorgegebenen Signalverlauf

(10) Es gibt zwei Aspekte, in deren Zusammenhang Toleranzbänder eingesetzt werden können: Zum einen können am Eingang des DES Toleranzen verwendet werden, um die Sensibilität des Systems zu mindern und so die Systemdynamik zu verringern. So lässt sich die Zahl der Systemtriggerungen und der Zustandsspeicherungen reduzieren. Zum anderen lässt sich der Kommunikationsaufwand zwischen CS und DES reduzieren: unterscheidet sich der neue Ausgangswert des CS nur gering vom Hypothesewert auf Seiten des DES, und liegt er somit im Toleranzbereich, so kann der Hypothesewert akzeptiert werden, was eine Übertragung des tatsächlichen Ausgangswertes unnötig macht. Da umso mehr Werte zum Toleranzbereich gezählt werden, je breiter das Toleranzband ist, ist der Kommunikationsaufwand antiproportional zur Toleranzbandbreite.

(11) Der Einsatz von Schwellwerten bringt einen Eingriff in das Simulationsverhalten mit sich, der Abweichungen und Ungenauigkeiten in den Simulationsergebnissen zur Folge haben kann. Dieser Aspekt muss bei der Auswertung der Resultate unbedingt berücksichtigt werden.

(12) Die Genauigkeit der Simulationsergebnisse wird bei gleich bleibender Referenzkurve für das Toleranzband maßgeblich von dessen Breite beeinflusst. Je breiter das Toleranzband, desto ungenauer sind die Ergebnisse.

(13) Aufgrund der in (10) aufgeführten Antiproportionalität von Toleranzbandbreite und Kommunikationsaufwand und der in (12) erwähnten Antiproportionalität von Toleranzbandbreite und Ergebnisgenauigkeit, muss ein Kompromiss zwischen Kommunikationsaufwand und Ergebnisgenauigkeit akzeptiert werden. Eine Optimierung des einen Kriteriums führt stets zur Verschlechterung beim anderen.

(14) Einen scheinbaren Widerspruch bildet die Feststellung, dass die Schwellwerte auf Seiten des DES festgelegt werden, aber auf Seiten des CS ausgewertet werden müssen, da eine Auswertung auf Seiten des DES eine Übertragung der Daten voraussetzen würde – und gerade das soll verhindert werden. Die Frage ist also, woher der CS die Schwellwerte des DES kennt, denn auch eine Übertragung der Schwellwerte führt zu einem hohen Kommunikationsaufwand.

4.6.6 Konzeption der Schwellwertprüfung: Synchrone Hypothese

Nachdem im vorhergehenden Abschnitt 4.6.5 Sinn und Zweck der Schwellwerte erläutert, ihre Notwendigkeit aufgezeigt und die Hauptschwierigkeit einer effizienten Realisierung formuliert wurde, wird im Folgenden ein Konzept entwickelt, das eine flexible Handhabung von Schwellwertüberschreitungen unter Minimierung des Kommunikationsaufwands ermöglicht. Dabei ist die Verwendung von Referenzkurven, wie sie in 4.6.5 vorgestellt wurden, ein wesentliches Grundprinzip. Diese Referenzkurven werden als Hypothesefunktionen sowohl am Ausgang des CS als auch am Eingang des DES mit jeweils identischem Verhalten eingesetzt (siehe Bild 105). Auf Seiten des DES liefern die Funktionen Hypothesewerte für die optimistische Simulation.

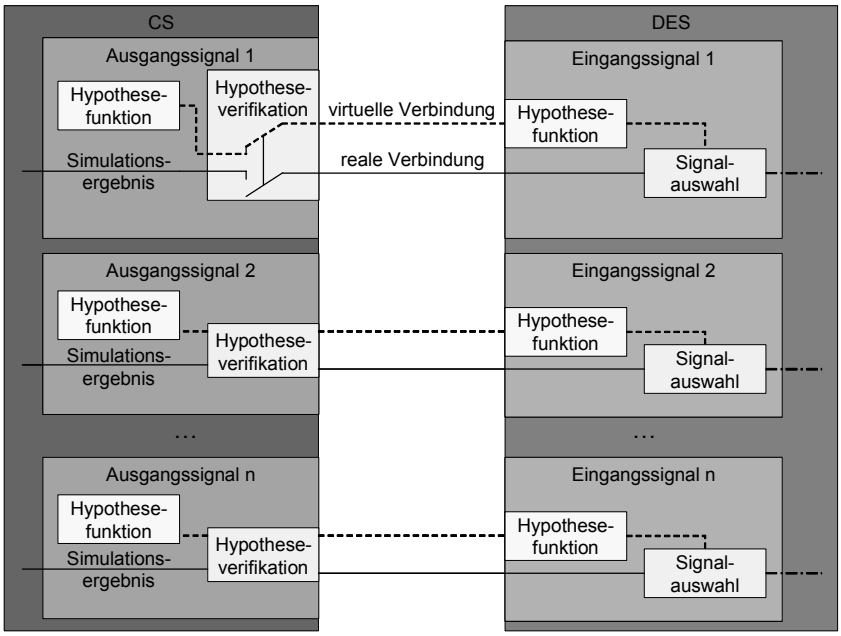


Bild 105: Datenaustausch zwischen CS und DES mit Hypothesefunktion

Aufgrund des identischen Verhaltens der Hypothesefunktionen auf beiden Simulatoren kennt der CS die Hypothesewerte des DES und kann diese auf Schwellwertüberschreitung prüfen. Bild 106 zeigt schematisch das hierfür eingesetzte Verfahren.

Ist der Simulationswert y größer als der Hypothesewert h ($d=y-h>0$), so muss auf Überschreitung der oberen Toleranzgrenze geprüft werden. Für $a_o-y>0$ liegt y unterhalb der oberen Toleranzgrenze und somit im Toleranzband. Der Hypothesewert des DES wird auf Seiten des CS als gültig akzeptiert, eine Übertragung des tatsächlichen Simulationswerts ist nicht notwendig. Ähnlich verhält es sich, wenn der Simulationswert y kleiner als der Hypothesewert h ($d=y-h<0$) ist. Nun muss auf Überschreitung der unteren Toleranzgrenze geprüft werden. Für $y-a_u>0$ liegt y oberhalb der unteren Toleranzgrenze und somit wiederum im Toleranzband. Der Hypothesewert des DES wird auch in diesem Fall auf Seiten des CS als gültig akzeptiert.

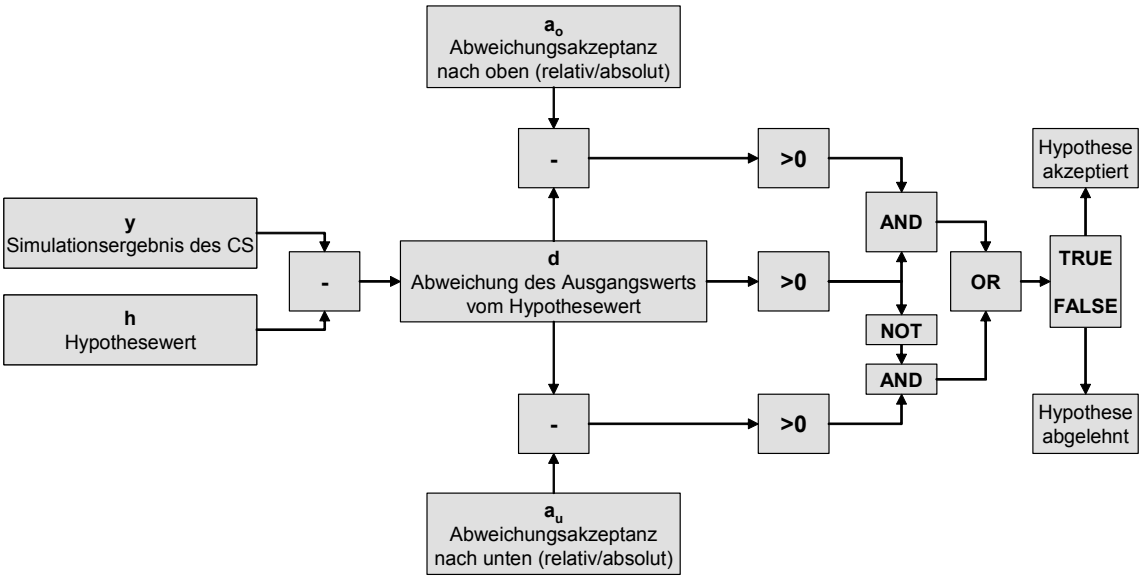


Bild 106: Verfahren zur Überprüfung der Gültigkeit der Hypothesefunktion

Art und charakteristische Eigenschaften der Hypothesefunktion werden in einer Konfiguration festgelegt, die den Funktionstyp (z. B. lineare Funktion, trigonometrische Funktion, e-Funktion, Interpolationsfunktion, usw.) und die jeweils benötigten Parameter (z. B. Amplitude, Offset und Frequenz bei der Sinus-Funktion) enthält. Kommunikationsaufwand und Simulationsgeschwindigkeit sind abhängig davon, wie gut sich die Hypothesefunktion an das tatsächliche Ausgangssignal des CS anpasst. Die Anpassungsgüte wird durch die Form der Hypothesefunktion bestimmt, die in der Konfiguration festgelegt ist. Da die Funktionstypen als austauschbare Module bereitgestellt werden, lässt es sich jedem Ausgangssignal des CS eine andere Konfiguration und somit eine andere Hypothesefunktion zuweisen. Eine passende Konfiguration für die Hypothesefunktionen kann gewöhnlich nicht a priori bestimmt werden; sie kann aber durch Testläufe, in denen die Charakteristik des Ausgangssignals bestimmt und Konfigurationsparameter variiert werden, iterativ angenähert werden. Denkbar ist auch der Einsatz komplexerer Interpolationsverfahren, wodurch ebenfalls eine Annäherung der Hypothesefunktion an den realen Signalverlauf ermöglicht wird – in diesem Fall sogar online während eines Simulationsdurchlaufs, der auf Werten aus bereits ausgeführten Simulationsschritten basiert.

Eine dynamische Rekonfiguration der Hypothesefunktion zur Simulationslaufzeit kann ebenfalls zu einer höheren Erfolgsquote bei den Hypothesewerten führen. In diesem Fall ist es wichtig, dass die Hypothesefunktionen auf Seiten des CS und des DES synchron rekonfiguriert werden, andernfalls werden die beiderseits berechneten Hypothesewerte nicht mehr übereinstimmen, was zu Simulationsfehlern führen kann. Der Erweiterbarkeit der Hypothesefunktionsmodule sind kaum Grenzen gesetzt; es gilt lediglich zu beachten, dass komplexe Hypothesefunktionen höheren Bedarf an Rechenleistung und Speicher haben, was ab einem gewissen Komplexitätsgrad die Simulationsgeschwindigkeit beeinträchtigen kann. Im Rahmen dieser Arbeit werden drei Hypothesefunktionen eingesetzt: die einfachste ist die Quasi-Konstante, deren Wert immer dem zuletzt übermittelten Ausgangswert des CS entspricht. Zusätzlich wird eine lineare Funktion, deren y-Achsenabschnitt und Steigung parametrisierbar sind, zur Verfügung gestellt. Außerdem steht eine Sinus-Funktion mit anpassbarer Amplitude, Frequenz, Phasenverschiebung und Offset bereit. Die Quasi-Konstante weist bereits laufzeitdynamische Eigenschaften auf, denn ihr Wert ändert sich während des Simulationsdurchlaufs.

Aus der Tatsache, dass während einer Simulation womöglich keine Simulationsergebnisse vom CS an den DES gesendet werden, darf keinesfalls geschlossen werden, dass sich die Ausgangswerte des CS nicht ändern würden und somit die Werte an den Eingängen des DES konstant seien. Im Gegenteil, die Signale können starke dynamische Eigenschaften aufweisen, und dennoch muss kein Datentransfer stattfinden. Dies ist der Fall, wenn für alle Signale jeweils eine Hypothesefunktion so opti-

mal konfiguriert wurde, dass die tatsächlichen Simulationswerte von den Hypothesewerten immer nur so schwach abweichen, dass sie noch im Toleranzband liegen. Ein umgekehrter Kausalzusammenhang scheint näher liegend, nämlich dass eine Hypothese darauf geprüft wird, ob sie in der Akzeptanzumgebung des tatsächlichen Werts liegt. Hier aber ist die erste Formulierung korrekt, denn durch die optimistische Simulation wird der Hypothesewert als Ist-Wert vom DES vorgegeben und das Simulationsergebnis des CS wird auf Abweichungen vom Hypothesewert geprüft. Die in 4.6.5 unter Punkt (10) aufgeführte Möglichkeit, ein Toleranzband am DES-Eingang einzusetzen, um eine gewisse Abweichung des realen Signals bzw. des Hypothesewerts vom vorherigen Eingangswert zuzulassen, wird im vorgestellten Konzept nicht umgesetzt. Da der als DES eingesetzte JStateSim in der aktuellen Fassung keine wertkontinuierlichen Signale als Trigger akzeptiert, sondern nur mit Aktionsereignissen getriggert werden kann, ist die angesprochene Reduzierung von Triggerungen nicht relevant. Gegen die Regel, bei jeder Änderung des Eingangswertes eine Zustandsspeicherung durchzuführen, kann an dieser Stelle verstoßen werden, da die Eingangswerte von der Hypothesefunktion stammen und daher ohne Zustandsspeicherung reproduzierbar sind. Nur wenn die Hypothese als falsch ausgewertet wird und der CS einen Korrekturwert sendet, muss dieser beim DES gespeichert werden. Somit wird in dieser Arbeit nur ein Toleranzband am Ausgang des CS eingesetzt, das Abweichungen des realen Signals von der Hypothese zulässt und somit das Datenaufkommen verringert.

4.6.7 Zeitliche Aspekte der Eingangssignale

An dieser Stelle soll die Frage geklärt werden, zu welchen Zeiten welche Daten von CS und DES gesendet werden müssen, um eine möglichst schnelle Simulationsausführung zu erreichen. Dieser Aspekt wird in [FrCW98] und [FrRa00] nicht detailliert behandelt. Wir gehen davon aus, dass der letzte Synchronisationspunkt von CS und DES bei t_{n-1} liegt. DES hat also zuletzt ein Ereignis mit Zeitstempel t_{n-1} ausgeführt und CS hat das Intervall von t_{n-2} bis t_{n-1} simuliert. Für die Festlegung der Sendezeiten der vom DES an den CS gesendeten Aktivierungsereignissen sind drei Prinzipien denkbar:

(1) Das Aktivierungsereignis wird vor Ausführung des ersten Ereignisses mit Zeitstempel t_n gesendet. Hier läuft der CS dem DES immer etwas hinterher. Man kann daher davon sprechen, dass der CS bei der Simulation des Intervalls t_{n-1} bis t_n die Hypothesewerte der mit dem CS verbundenen Eingänge des DES zum Zeitpunkt t_n verifiziert. Simuliert der CS schneller als der DES, so muss er nach Berechnung des Simulationsintervalls warten, bis der DES ein neues Aktivierungsereignis sendet. Simuliert der DES schneller als der CS, so werden sich beim CS die Aktivierungsereignisse sammeln, da sie schneller eintreffen, als sie abgearbeitet werden können. In diesem Fall steigt die Wahrscheinlichkeit für eine Rücksetzung auf Seiten des DES, da sein Operieren immer stärker auf Hypothesewerten basiert, die vom CS noch nicht verifiziert wurden. Wird $p_{H,T}$ als die Wahrscheinlichkeit definiert, mit der alle Hypothesewerte des DES in einem Zeitschritt gültig sind, ergibt sich die Wahrscheinlichkeit p_R für eine Rücksetzung, unter der Bedingung, dass der CS das Intervall t_{n-1} bis t_n simuliert und der DES den Zeitschritt t_{n+m}, zu $p_R = 1 - (p_{H,T})^{m+1}$. Die Wahrscheinlichkeit für eine Rücksetzung strebt also gegen 1, wobei die durch Zustandsspeicherung verursachten Gesamtkosten mit zunehmender Zahl an Simulationsschritten des DES linear steigen. Durch Überlagerung von Wahrscheinlichkeitsfunktion und normierter Kostenfunktion lässt sich bestimmen, wie weit ein DES einem CS vorauseilen darf, bis der Kostenaufwand der Wahrscheinlichkeit nach unrentabel wird. Bild 107 zeigt drei Wahrscheinlichkeitsfunktionen für p_R mit unterschiedlichen Werten für $p_{H,T}$, sowie zwei Kostenfunktionen mit unterschiedlicher Steigung K, die die normierten Kosten pro Simulationsschritt wiedergibt. Die Schnittpunkte von Wahrscheinlichkeitsfunktionen und normierter Kostenfunktion geben die Rentabilitätsgrenzen an.

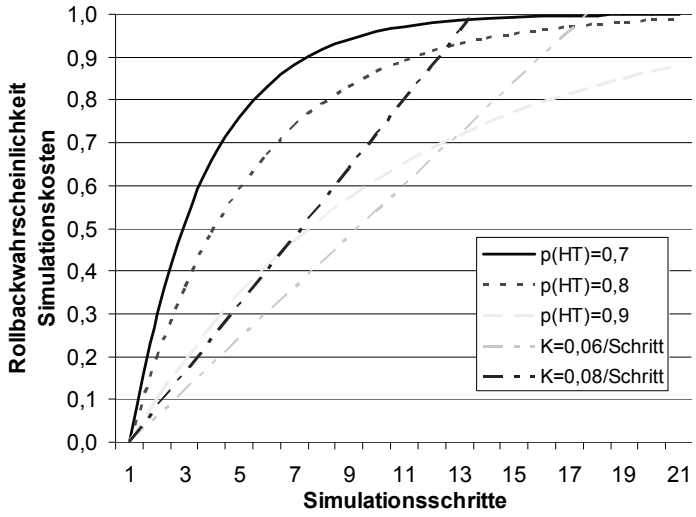
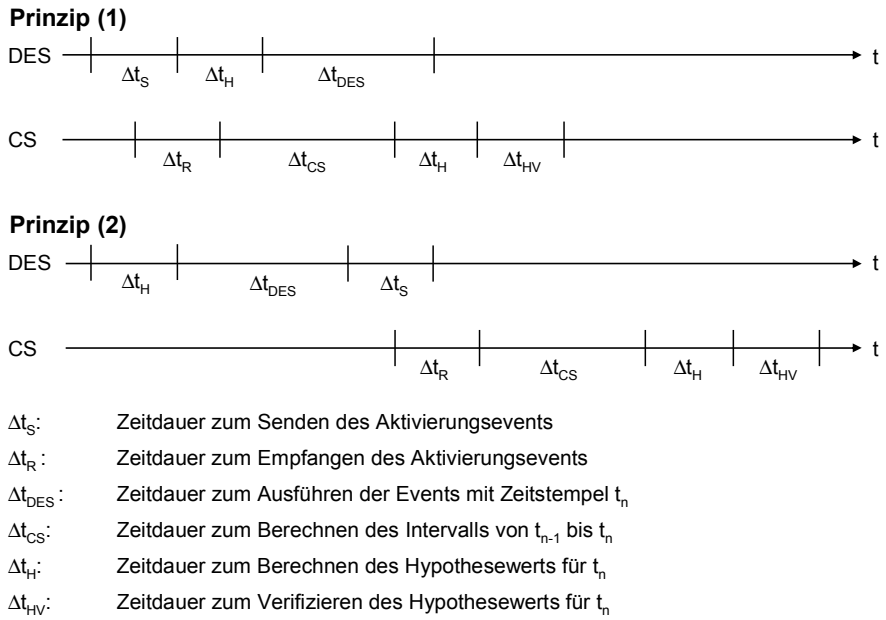


Bild 107: Kosten- und Wahrscheinlichkeitsfunktion

Dieser Aspekt wird beim vorgestellten Konzept aufgrund des zusätzlich notwendigen Implementierungsaufwands nicht beachtet. Dennoch wäre es interessant zu sehen, in wie weit die Simulationsgeschwindigkeit beeinflusst wird, nicht zuletzt durch zusätzlichen Kommunikationsaufwand, da der CS den DES darüber informieren muss, wie weit er momentan simuliert hat. Außerdem benötigt die vom DES durchgeführte Auswertung von Kosten- und Wahrscheinlichkeitsfunktionen zusätzliche Rechenleistung. Die Erfolgsquote der Hypothese des CS ist identisch mit der des DES: der CS berechnet das Intervall von t_{n-1} bis t_n, und an seinen Eingängen liegen ausschließlich Werte des DES vom Zeitpunkt t_{n-1} an, die bis t_n konstant sind. Sind diese Werte vom DES auf Grundlage einer korrekten Hypothese bestimmt worden, so ist automatisch auch die Hypothese des CS korrekt. Erweist sich hingegen beim DES eine Hypothese als falsch und wird dadurch eine Rücksetzung ausgelöst, so geschieht dies auch beim CS.

(2) Das Aktivierungsereignis wird nach Ausführung des letzten Ereignisses mit Zeitstempel t_n gesendet. Dieses Prinzip führt zu einem gleichen Synchronisationsverhalten wie (1). Wie dem Bild 108 entnommen werden kann, wird es allerdings zu einer langsameren Ausführung kommen, da der CS erst nach Ausführung der Ereignisse auf Seiten des DES aktiviert wird und daher nicht parallel zum DES simulieren kann. Aus diesem Grund ist Prinzip (1) vorzuziehen.

(3) Das Aktivierungsereignis wird gesendet, sobald der DES ein Ereignis, für den noch kein Aktivierungsereignis an den CS gesendet wurde, mit einem Zeitstempel t_n, erzeugt oder empfängt. Dieses Prinzip unterscheidet sich deutlich von (1) und (2), da hier der CS dem DES vorauseilen kann. Dies ist der Fall, wenn der CS die durch die Aktivierungsereignisse festgelegten Simulationsintervalle schneller berechnet, als der DES die entsprechenden Ereignisse ausführt. Nun simuliert der CS zusätzlich mit einem vom DES unabhängigen Optimismus: er muss seine Dateneingänge mit hypothetischen Werten belegen, da am Ausgang des DES noch keine gültigen Signale anliegen. Zusätzlich gilt die Hypothese, dass der DES kein Aktivierungsereignis sendet, dessen Zeitstempel kleiner als die Simulationszeit des CS ist. Da die in (1) angegebene Rücksetzungswahrscheinlichkeit seitens des DES bei diesem Prinzip auch noch gültig ist, führt dies zu einer nachvollziehbar minimalen Wahrscheinlichkeit, dass sich alle Hypothesen als korrekt erweisen. Aufgrund dessen bietet dieses Prinzip wenig Aussicht auf Erfolg.



Δt_S: Zeitdauer zum Senden des Aktivierungsevents

Δt_R: Zeitdauer zum Empfangen des Aktivierungsevents

Δt_{DES}: Zeitdauer zum Ausführen der Events mit Zeitstempel t_n

Δt_{CS}: Zeitdauer zum Berechnen des Intervalls von t_{n-1} bis t_n

Δt_H: Zeitdauer zum Berechnen des Hypothesewerts für t_n

Δt_{HV}: Zeitdauer zum Verifizieren des Hypothesewerts für t_n

Bild 108: Auswirkung des Sendezeitpunkts des Aktivierungsereignisses

Für die vom DES zum CS gesendeten Datenereignisse ist es ideal, wenn die Eingangswerte des CS für den Zeitpunkt t_{n-1} spätestens anliegen, wenn der CS das Aktivierungsereignis für das Simulationsintervall $[t_{n-1}, t_n]$ empfängt und daraufhin die Simulation des Intervalls startet. Allerdings kann dies nicht garantiert werden, da die Datenereignisse nicht vom aktivierenden, sondern ausschließlich von anderen DES an die Dateneingänge des CS gesendet werden dürfen (siehe 4.6.4). Somit ist bei verteilter, optimistischer Simulation diesbezüglich keine Kontrolle der Sende- bzw. Empfangszeitpunkte möglich. Trifft ein Datenereignis verspätet, also nach Empfang des zugehörigen Aktivierungsereignisses ein, so muss eine Rücksetzung durchgeführt werden.

Wie weiter oben erklärt, sendet der CS dem DES nur Datenereignisse, wenn sich eine Eingangswerthypothese als falsch erweist. Da der DES in diesem Fall mit falschen Werten simuliert, ist eine möglichst frühe Übertragung des korrekten Werts sinnvoll, damit der DES zurückspringen und möglichst bald wieder mit korrekten Wert weiterrechnen kann. Diese Aussage kann in Kapitel 5 mit Messergebnissen bestätigt werden.

Als letzter Punkt wird erläutert, zu welchen Zeiten die Hypothesefunktion aktiviert wird, um Hypothesewerte für die Belegung der Eingänge des DES zu liefern. Dieser Aspekt ist abhängig davon, wie und wann ein DES getriggert werden kann und wird an dieser Stelle für den verwendeten JStateSim betrachtet. Beim verwendeten JStateSim kommt ein interner Trigger zum Einsatz, der ausgelöst wird, solange die Ereignisliste nicht leer ist, oder sich der Zustand des Stateflowcharts bei Triggerung ändert, wie beispielsweise bei bedingungslosen Transitionen, die bei jeder Triggerung einen Zustandsübergang ermöglichen. Der Grund für dieses Triggerkonzept ist, möglichst viele Schritte simulieren zu können, um so einen hohen Grad an Optimismus zu erreichen. Für die Zeiten, in denen JStateSim keine vorrätigen Ereignisse hat, müssen alle Eingangswerte auf Wertänderungen untersucht werden, was bedeutet, dass alle Hypothesefunktionen ausgeführt werden. Dies ist erforderlich, weil vom aktiven Zustand des Stateflowcharts Transitionen mit Bedingungen ausgehen können, in denen Eingangswerte als Operanden einer Relationsoperation verwendet werden. Jede Eingangswert-Änderung kann somit dazu führen, dass die Transitionsbedingung wahr wird und das Stateflowchart aus seinem aktiven Zustand in einen anderen Zustand wechseln kann. Da die Triggerung von JStateSim darauf angelegt ist, jeden möglichen Zustandsübergang möglichst früh durchzuführen, um so den Simulationsfortschritt zu garantieren und nicht unnötig zu verlangsamen, müssen also alle Hypothesefunktion dann ausgeführt werden, wenn keine Ereignisse in der Ereignisliste von JStateSim stehen. Vom Prinzip her wird zu diesen Zeiten die Hypothesefunktion mit der Frequenz des Triggers abgetastet und

bei Wertänderungen das Stateflowchart getriggert. Ergeben sich zu jedem Abtastzeitpunkt Wertänderungen, wird JStateSim zeitgesteuert, was mit den Erkenntnissen aus Punkt (6) in 4.6.5 übereinstimmt. Für alle Zeitpunkte, an denen JStateSim vorrätige Ereignisse ausführt, muss die jeweilige Hypothesefunktion nur ausgeführt werden, wenn auf den entsprechenden Eingangswert zugegriffen wird. Allerdings ist diese differenzierte Auswertung mit zusätzlichem Rechenaufwand verbunden; hinzukommt, dass je nach Realisierung die Hypothesefunktion mehrmals aufgerufen werden müsste. Außerdem wäre ein tieferer Eingriff in die Struktur von JStateSim notwendig. Aus diesen Gründen wird die einfachere Lösung bevorzugt, wonach vor jedem Simulationsschritt zunächst die Dateneingänge mit Hypothesewerten belegt werden. Dies ist auch für die in diese Arbeit integrierte zweite Triggerungsvariante von JStateSim für optimistische Simulation sinnvoll: hier ist es möglich, mit externen Ereignisse ohne internen Trigger den DES zu aktivieren, was dem Grundprinzip der ereignisgesteuerten Simulation entspricht. In diesem Fall müssen vor Ausführung des getriggerten Stateflowcharts die Eingangswerte mit Hypothesewerten belegt werden, soweit keine berechneten Eingangswerte bekannt sind.

4.6.8 Erweiterungsmöglichkeiten

Wie in 4.6.6 angesprochen, lässt sich die Bibliothek der Hypothesefunktionsmodule uneingeschränkt erweitern; allerdings sollte bei der Implementierung eines neuen Moduls beachtet werden, dass komplexe Hypothesefunktionen höheren Bedarf an Rechenleistung und Speicher haben, was ab einem gewissen Komplexitätsgrad die Simulationsgeschwindigkeit beeinträchtigen kann. Eine mögliche Variante von SES wird in 4.6.7 erwähnt: der Optimismus auf Seiten des DES wird durch Berücksichtigung von Rücksetzungswahrscheinlichkeit und Kosten pro Zustandsspeicherung eingeschränkt. Auch diese Erweiterung benötigt zusätzliche Rechenleistung und Arbeitsspeicher, weswegen eine genauere Kosten/Nutzen-Betrachtung vor der Umsetzung angebracht ist.

4.7 Infrastruktur für eine PIL-Simulation

Es wurde eine Target-Host-Kommunikationsinfrastruktur für die Realisierung einer PIL-Simulation entwickelt. Ausgehend von einer Anforderungsbeschreibung werden Schritt für Schritt Funktionen des späteren Steuergerätes in Simulink modelliert und mit einer virtuellen Umgebung, dem Modell der Strecke, getestet. Das in der Simulation getestete Steuergerätemodell wird in einem weiteren Schritt durch Einsatz des Code-Generators TargetLink (TL) in C-Code transformiert. Ein OSEK-konformes RTOS, wie z. B. Wind River OSEKWorks, wird integriert und um eine CAN-Schnittstelle sowie eine Zeitsynchronisation erweitert und das Resultat zur Ausführung auf dem Zielsystem gebracht. Durch eine Auslagerung geprüfter SW-Stände auf die Ziel-HW lassen sich konkrete Erkenntnisse über den bereits erreichten Entwicklungsstand gewinnen, so dass Aussagen über Qualität und Kosten bereits in frühen Entwurfsphasen möglich sind. Die Kommunikation zwischen Simulink auf dem Host-PC und dem in der Schleife befindlichen Zielsystem erfolgt über einen CAN-Bus. Für die µC-Knoten kommt mit TL automatisch generierter Produktionscode, sowie OSEK für die Echtzeitanforderungen, zum Einsatz.

4.7.1 PIL-Simulation mit PC (Simulink) und Mikrocontroller über CAN

Zur Kommunikation mit externen CAN Komponenten: Ist das OSEK-System entsprechend konfiguriert, kann auf die Funktionalität des CAN-Kommunikationssystems in C-Code über die in der API von OSEK COM spezifizierten Funktionen zugegriffen werden. Über die API-Funktionen des CAN-Treibers kann innerhalb einer Anwendung direkt auf die vom CAN-Controller zum Senden und Empfangen von CAN-Nachrichten bereitgestellten CAN-Nachrichtenpuffer zugegriffen werden. Wird OSEK COM nicht verwendet, sondern direkt auf die Funktionen des CAN-Treibers zugegriffen, können OIL-Objekte `MESSAGE`, `CANOBJECT`, `CANADDRESS` und `NETWORK` nicht zum Einsatz kommen. Bei dieser Vorgehensweise wird direkt auf Funktionen und Datentypen des CAN-Treibers zugegriffen, die in [BSPW02] ausführlich beschrieben sind. Messungen haben ergeben, dass mit dieser Methode eine deutlich höhere Datenrate erzielt werden kann. Die Messungen zur Busauslas-

tung haben gezeigt, dass bei Verwendung der von OSEK bereitgestellten externen Kommunikation wesentlich weniger Daten gesendet werden können, als dies bei direktem Zugriff auf Funktionen des CAN-Treibers möglich ist; allerdings ist die Programmierung aufwendiger. Es wird davon abgeraten, innerhalb eines Projekts sowohl über OSEK COM als auch mit direktem Zugriff auf den CAN-Treiber [CANL03] CAN-Nachrichten zu versenden, da es sonst zu Inkompatibilitäten kommen kann, die zu Kommunikationsfehlern führen.

Auf der Simulink-Seite verwenden S-Funktionen das API der PC-CAN-Karte zum Lesen und Schreiben. Ferner wird dort die chronologische Reihenfolge zwischen Lese- und Schreibzugriff spezifiziert. Da die Nachrichtengröße über den CAN-Bus fest ist und ein `Char`-Feld benutzt wird, wird die Umwandlung zwischen ganzzahligen und Fließkomma-Datentypen, die im Modell verwendet werden, durch S-Funktionen wie auch durch das Target durchgeführt.

Der Datenaustausch zwischen individuellen Tasks, die durch TargetLink generiert oder optional manuell eingebunden werden können, findet über OSEK COM (Kommunikationsschicht) statt. Eine der Tasks wird für die Kommunikation mit dem PC über den CAN-Bus benutzt, und eine weitere für die Initialisierung von OSEK-Alarms. Die Initialisierungs-Task hat die höchste Priorität und wird nur einmal ausgeführt. Weitere Tasks beinhalten Funktionalitäten des Modells und werden entweder periodisch von OSEK-Alarms aktiviert oder aktivieren sich gegenseitig. Eine einfache Applikation kann z. B. aus vier Tasks bestehen, die auf dem µC laufen: Scheduler, Controller (Bild 109) und Communication und mit TL generiert werden sowie einer Kontrollgröße, die mit dem A/D-Konverter des µC verbunden ist.

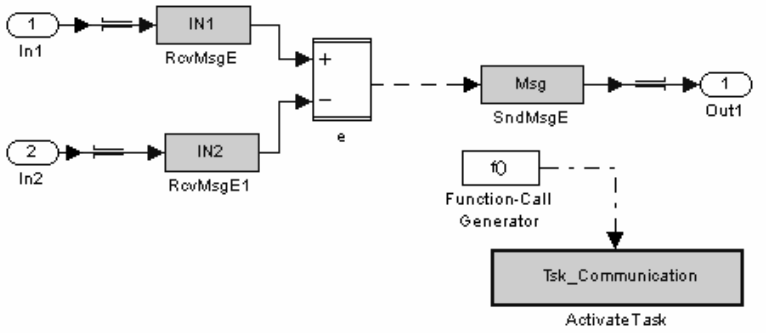


Bild 109: Beispiel einer Controller-Task

Die letzte Task in dieser Kette aktiviert die Kommunikations-Task, die durch TL-Custom-Code-Blocks auf den CAN-Bus zugreift (Bild 110).

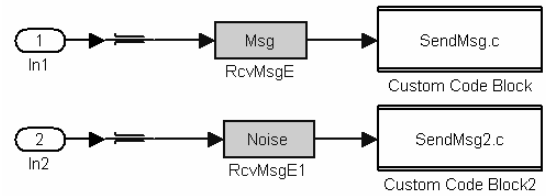


Bild 110: Beispiel einer Kommunikations-Task

Sie muss eine niedrigere Priorität haben, weil die vorige Task nicht unterbrochen werden darf, wenn präemptives[58] Scheduling eingesetzt wird. Die Zeitintervalle zwischen OSEK-Alarms müssen möglichst groß gemacht werden, um alle Tasks komplett abzuarbeiten. Wenn eine CAN-Nachricht am Zielsystem ankommt, wird die Task-Verarbeitung durch eine Unterbrechung unterbrochen. Eine Unterbrechungs-Verarbeitungsroutine benutzt das CAN-API und das OSEK-COM-API um die Nach-

[58] Tasks heißen nichtpräemptiv, wenn sie nicht unterbrochen werden können. In diesem Fall belegt die Task eine Ressource, solange sie diese benötigt. Tasks heißen präemptiv, wenn der Zugriff einer Task auf eine Ressource von einer wichtigeren Task unterbrochen werden kann (siehe 2.1.1).

richt weiterzuleiten. Eingaben, Strecke und die Anzeige der Ausgaben verbleiben in Simulink (Bild 111).

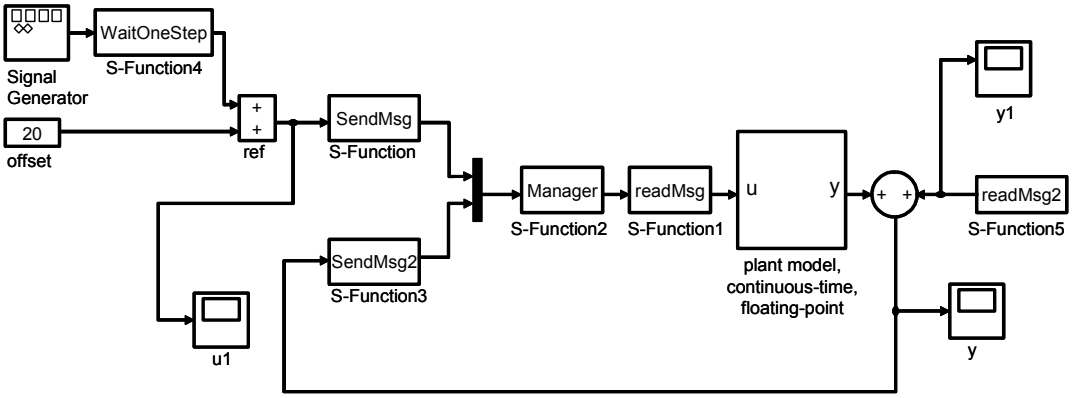


Bild 111: Angepasstes Modell mit Sende- und Empfangsfunktionen

Zuerst wird die Simulation des Simulink-Modells gestartet, anschließend die Applikation auf dem µC. Die S-Funktion `WaitOneStep` wartet auf die Start-Nachricht des µC, erst dann schaltet sie in den transparenten Zustand. Die S-Funktionen `SendMsg` und `SendMsg2` senden die Simulationsdaten zum Controller, `readMsg` liest die ankommenden Daten, und `Manager` definiert die chronologische Reihenfolge: zuerst wird gesendet, dann gelesen. Somit ist die Kommunikation auf dem CAN-Bus bidirektional. Während der Sendephase kann Simulink weitere Berechnungen durchführen und Ergebnisse zum CAN-Bus senden. Während der Empfangsphase muss Simulink warten, solange keine Nachricht vom CAN empfangen wird. `Manager` prüft ununterbrochen, ob eine Nachricht auf dem CAN-Bus ist. Wenn das der Fall ist, empfängt Simulink diese mittels `readMsg` und startet dann wieder von vorne. Neben der inkrementellen Methode ist eine Co-Simulation von Modellen zwischen dem Host-PC und mehreren Targets über CAN möglich. Die SW auf dem Zielsystem läuft in einer Zeitscheibe. Automatisch generierter Code, der in Großserien verwendet werden kann, wird in Verbindung mit einem OSEK-kompatiblen BS verwendet. Der größte Teil des gesamten Setup-Aufwandes wird nur einmal für jedes Zielsystem benötigt und ist ohnehin notwendig, wenn der generierte Code getestet und in Echtzeit in der realen Umgebung ohne Kopplung mit der Entwicklungsplattform ausgeführt wird.

4.7.2 Fehlersuche für Zeitanforderungen auf einer µC-Zielplattform

Es existieren verschiedene Methoden zum Messen und Darstellen der Ausführungsdauer von Tasks. Eine Möglichkeit ist die Zeitmessung mit OSEK-Alarms. Diese sind gemäß der OSEK-OS-Spezifikation an einen OSEK-Counter gebunden. Ein Alarm wird ausgelöst, wenn der Counter einen vorgegebenen Wert erreicht hat. Beim Auslösen des Alarms kann z. B. eine Task aktiviert werden. Da der Counter zyklisches Verhalten aufweist, ist es nicht möglich, allein über die Differenz der Counter-Werte zu zwei Zeitpunkten die Dauer zwischen diesen zwei Zeitpunkten zu bestimmen. Vielmehr muss zusätzlich die Anzahl $n = n_1 - n_0$ der zwischenzeitlich abgelaufenen Zyklen in die Berechnung mit eingehen. Die Zeitspanne zwischen zwei Zeitpunkten t_0, n_0 und t_1, n_1 berechnet sich demnach aus:

$$t = (t_1 - t_0) + (n_1 - n_0) \cdot T$$

mit

t_0: Zählerwert zum Beginn der Messung
t_1: Zählerwert zum Ende der Messung
n_0: Anzahl der bereits durchlaufenen Counter-Zyklen zum Beginn der Messung
n_1: Anzahl der durchlaufenen Counter-Zyklen zum Ende der Messung
T: Zeitdauer eines vollständigen Counter-Zyklus

Das Erreichen des Maximalwerts des Counters wird durch einen Alarm signalisiert. Dieser aktiviert eine Task, in der eine Variable *n* zum Zählen der durchlaufenen Zyklen jeweils um eins erhöht wird. Messungen haben gezeigt, dass diese Methode nur bei Tasks mit langer Ausführungsdauer anwendbar ist, da zum einen die Zeitauflösung des verwendeten `SystemTimer` zu gering ist und zum anderen es eine Grenzfrequenz gibt, bei der die vom Alarm gestartete Task nicht mehr vollständig ausgeführt wird. Aus diesem Grund wurde bei den Messungen eine andere Methode gewählt. Hierbei wird unter Umgehung von OSEK direkt über Assembler-Befehle auf das Timebase-Register des MPC555 zugegriffen. Die resultierenden Werte werden in Zähleinheiten (Ticks) gemessen, die jeweils einer bestimmten Zeiteinheit entsprechen. Durch Einführen einer weiteren CAN-Nachricht können die Aktiv-/Passiv-Zeiten der Tasks in Simulink graphisch dargestellt werden. Die Nachrichten haben unterschiedliche IDs und werden in Simulink von den zugehörigen S-Funktionen empfangen und an ein XY-Scope weitergegeben, wobei die Zeitangaben (Simulationsschritte) auf der X-Achse und die Aktiv-/Passiv-Signale auf der Y-Achse aufgetragen werden (Task ist aktiv (*y=1*) oder inaktiv (*y=0*)).

Die fallende Gerade in Bild 112 symbolisiert die Dauer der Task-Ausführung, während der steigenden Gerade ist die Task passiv.

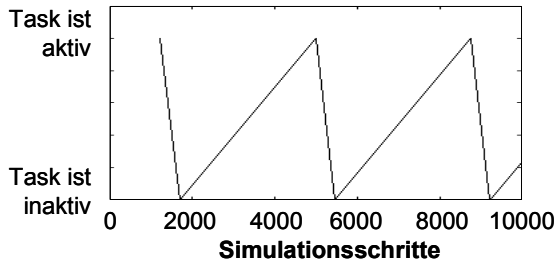


Bild 112: Zyklisch ausgeführte Task

Sollen die Ausführungszeitpunkte von zwei Tasks so festgelegt werden, dass beide sich nicht unterbrechen, so muss die eine Task stets in einem Zeitraum ausgeführt werden, innerhalb dessen die andere passiv ist. Wie in Bild 113 dargestellt, kommt es zu Überschneidungen der Ausführungszeiten und somit zu Task-Unterbrechungen. Letztere lassen sich an den waagerechten Geradenabschnitten erkennen.

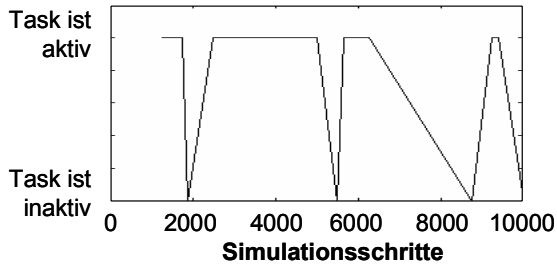


Bild 113: Zwei sich gegenseitig unterbrechende Tasks

4.7.3 FPGA-Integration

Zur Entwicklung von FPGAs kommt das RP-System RP.2002 [RaPr02] zum Einsatz, das u. a. über FPGAs, CPU, RAM und eine Ethernet-Schnittstelle verfügt. Mit Hilfe spezieller Blocksätze wie z. B. Xilinx SystemGenerator und AccelChip AccelFPGA, und des Werkzeugs SF2VHD der University of California at Berkeley [Came01] kann zur Konfiguration eines FPGAs aus Stateflow-Modellen VHDL-Code generiert werden. Das MATLAB-M-Skript SF2VHD wurde hinsichtlich der unterstützten Stateflow-Eigenschaften in wesentlichen Punkten erweitert und objektorientiert in Java für einen MATLAB-unabhängigen Einsatz neu entwickelt (JVHDLGen). Zum FPGA-Entwurf kommen die Werkzeuge von Mentor Graphics und Xilinx zum Einsatz.

Bei einer Verteilung eines Systems auf mehrere ausführende HW-Komponenten ist ein Kommunikationsmedium notwendig, über das die einzelnen Systemkomponenten Informationen austauschen können. Da CAN eine weit verbreitete Lösung darstellt, ist in dieser Arbeit der Datenaustausch mit CAN (ISO 11898) realisiert, das speziell für den Datenaustausch zwischen elektronischen Steuergeräten entwickelt wurde. Die entworfenen Teile des FPGA-CAN-Frameworks sind ebenfalls in VHDL implementiert; der Softcore des CAN-Controllers ist in der HW-Beschreibungssprache Verilog geschrieben. Für die CAN-Busanalyse wird das Werkzeug CANoe (CAN open environment) verwendet, das das Verhalten von Steuergeräten in einem CAN-Netzwerk simuliert [Broc99] [CAIn03]. Es stellt eine Entwicklungs-, Test- und Analyseumgebung für CAN-Bussysteme für den Kommunikationsentwurf, die Modellerstellung, -Diagnose und -Validierung zur Verfügung. FPGA Advantage von Mentor Graphics wird zum Einbinden von mit JVHDLGen erzeugtem VHDL-Code in das FPGA-CAN-Framework verwendet. Es ist eine Lösung für den Entwurf von FPGA-Systemen und beinhaltet die drei Werkzeuge HDL Designer Series (Entwurfsentwicklung und -verwaltung), ModelSim (Entwurfsverifikation, Fehlersuche und Analyse) und Leonardo Spectrum[59] (Entwurfssynthese) und integriert sie in den VHDL-Entwurfsfluss (siehe Bild 46).

Die Code-Generierung aus Simulink/Stateflow-Modellen erfolgt mit TL: Unter Verwendung von TL lässt sich aus Simulink- und Stateflow-Modellen C-Code generieren, der in die OSEK-Umgebung eingebunden und nach Compilierung auf der Ziel-HW ausgeführt werden kann. Um die beschriebene Methodik der Zeitmessung möglichst weitgehend zu automatisieren, wurde die dort aufgeführte Funktionalität in zwei Custom-Code-Blöcke von TL integriert, wobei der erste für die Initialisierung und den Start der Messung und der zweite für deren Beendigung und die Ergebnisübertragung zuständig ist. Der zu messende Modellteil kann somit zwischen diesen zwei Blöcken geschaltet werden, was die Handhabung der Zeitmessung erleichtert. Ein weiterer Block enthält einen in Verilog implementierten, frei erhältlichen Softcore-CAN-Controller, der eine Schnittstelle für den µC 8051 von Intel besitzt [OPCO05]. Gleichzeitig übernimmt es die Koordination des Simulationsablaufs, indem es das mit JVHDLGen in VHDL konvertierte Stateflowchart über die Kontrollsignale `clk` (Clock), `reset` und `ce` (Chip enable) ansteuert. Die Verwendung dieser drei high-aktiven Signale ist durch JVHDLGen vorgegeben, das bei jedem Modell diese Signale als Teil der Schnittstelle im `port`-Eintrag der `entity` generiert und sie u. a. als sensitive Signale im VHDL-Prozess-Kopf einsetzt.

Da es für den vorgesehenen Einsatz überdimensioniert gewesen wäre, einen Softcore für einen solchen µC 8051 in das System zu integrieren, wurde ein auf die Schnittstelle des 8051 angepasstes Steuerwerk für die Ansteuerung des CAN-Controllers entworfen.

4.8 Integration des Rapid-Prototyping-Systems RP.2002

Das Stateflow-Modell wird in ein VHDL-Modell umgewandelt. Nach der Synthese kann es auf einem FPGA ausgeführt und in eine verteilte Simulation eingebunden werden. Es wurde untersucht, wie effizient Beschreibungsmittel von ereignisdiskreten Modellen auf einem FPGA implementiert werden können. Modelle unterschiedlicher Größe und Komplexität wurden verwendet. Kriterien sind die Ressourcennutzung und die Laufzeit. Nicht die ganze Semantik von Statecharts (Stateflowcharts) wird von SF2VHD unterstützt; speziell die AND-Dekomposition wird im Vergleich zu JVHDLGen nicht unterstützt. Im Vergleich zu STATEMATE führt Stateflow nebenläufige Statecharts nicht unabhängig voneinander aus. Es wurde auch untersucht, in welchem Umfang eine nebenläufige Ausführung von Statecharts auf einem FPGA beschleunigt wird [Dumm03]. AccelFPGA generiert vom modifizierten Simulink- oder MATLAB-Entwurf eine VHDL-Beschreibung auf Register-Transfer-Ebene, die dann synthetisiert und simuliert werden kann. Stateflow-Blöcke werden nicht unterstützt. Ähnliche Einschränkungen existieren für SystemGenerator.

[59] Das PLD-Synthesewerkzeug (Programmable Logic Device-Synthesewerkzeug) Leonardo Spectrum ist auch Bestandteil des kompletten FPGA-Design-Flow-Paketes FPGA Advantage von Mentor Graphics [DeE01b].

Deshalb wird das Werkzeug JVHDLGen eingesetzt, zusammen mit den Werkzeugen von Mentor Graphics und Xilinx.

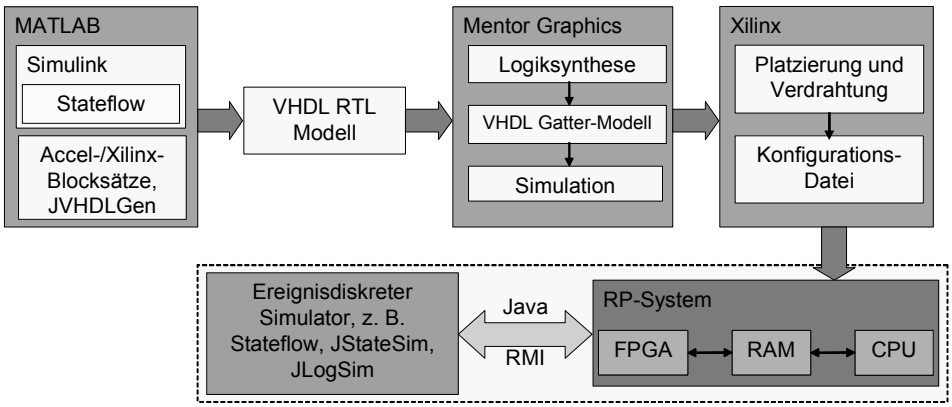


Bild 114: VHDL-Entwurfsfluss

4.8.1 Schnittstelle zwischen Modell (FPGA) und Netzwerk (RMI)

4.8.1.1 Hardwarezugriff

Die Simulationsstruktur ist in Bild 115 dargestellt.

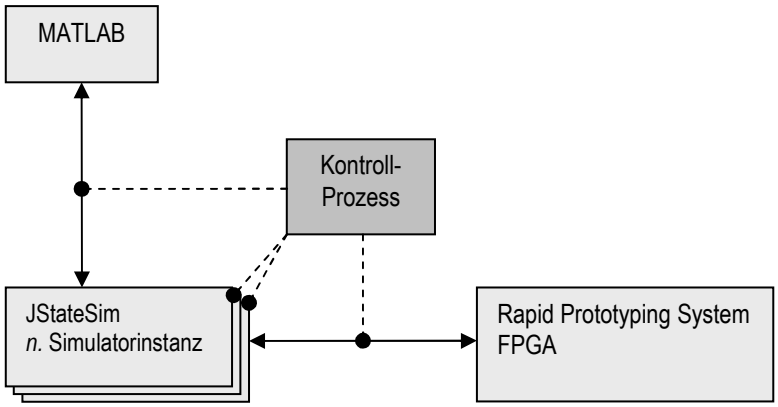


Bild 115: Simulationsstruktur

Die Kommunikation zwischen der C-Schnittstelle von MATLAB (S-Funktion) und Java ist mit Hilfe des JNI realisiert. Ein in Java implementiertes Kontrollprozess kommuniziert mit dem RMI-Teil des Simulators. Eine bidirektionale Verbindung zwischen MATLAB und dem RP-System ist ebenfalls möglich, da die Schnittstelle dieselbe wie zwischen MATLAB und JStateSim ist. Das JNI wird als Schnittstelle zwischen Java und der HW, sowie für die Kommunikation zwischen JStateSim und MATLAB verwendet. Sie erlaubt Java-Code, der in einer Java Virtual Machine läuft, die Zusammenarbeit mit Bibliotheken, die in C/C++ geschrieben sind. Die SW-Kommunikation auf dem RP.2002-System ist in Bild 116 dargestellt.

Die Kommunikation zwischen den Objektinstanzen von JStateSim sowie der CPU des FPGA-RP-Systems wird von Java-RMI übernommen. Eine modifizierte Version von JStateSim kommuniziert über RMI mit den anderen Simulatoren, die an der Simulation beteiligt sind. Die dynamische Bibliothek `fpga.so` ist mittels JNI eingebunden, die Anfragen in ein FIFO schreibt und die Antworten aus einem FIFO ausliest. Die FIFOs dienen als Kommunikationsschnittstelle zwischen Benutzer- und Kernel-Adressraum. Im Kernel-Adressraum liest das Modul `fpga_sim.o` die Befehle des Simulators und übersetzt sie in Lese- und Schreiboperationen des PCI-Treibers. Er schreibt die Rückgabewerte nach der Ausführung auf dem FPGA für den Simulator in den entsprechenden FIFO. Die HW-

Zugriffe auf das FPGA über den PCI-Bus und lokalen Bus werden für den Benutzer durch den PCI-Treiber transparent gemacht.

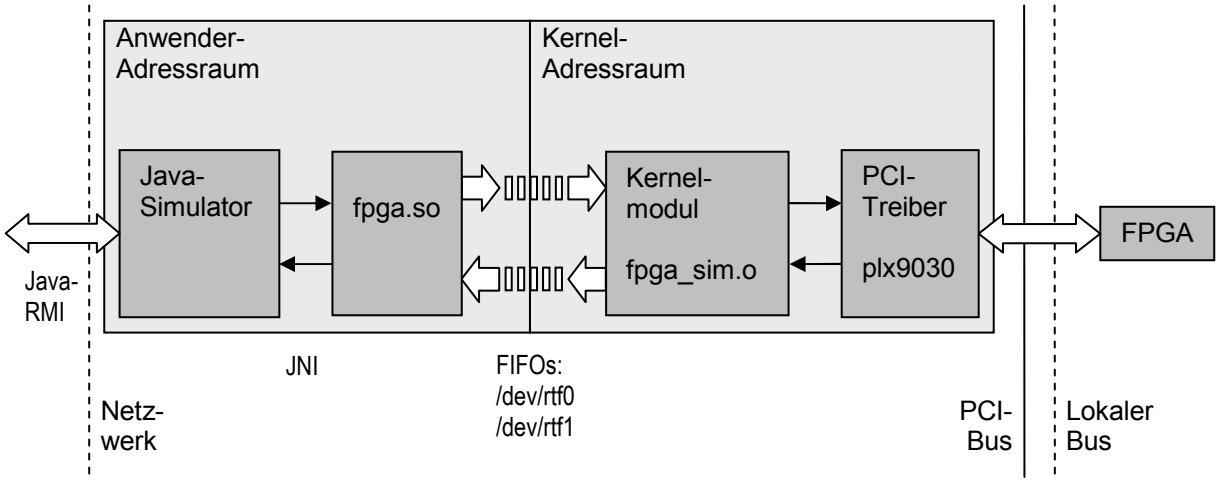


Bild 116: Kommunikationsschema auf dem RP.2002

Bild 117 zeigt eine schematische Übersicht der Module, die innerhalb des FPGA realisiert sind. Die Konfiguration des FPGA sowie die Kommunikation mit den Modulen ist über den lokalen Bus realisiert – ein synchroner Bus zwischen dem PCI-Controller und dem Dual-Port Random Access Memory[60] (DPRAM) innerhalb des FPGA. Der Taktspeicher bekommt seinen Takt vom PCI-Bus und versorgt den lokalen Bus und das FPGA mit dem Takt. Der FPGA-Takt wird von einem Taktgenerator erzeugt, der zwei interne Systemtakte für das FPGA generiert. Die Steuereinheit übernimmt die Datenvorbereitung und die Kontrolle des Simulink-Modells und sollte so flexibel wie möglich sein, um ohne Modifikation auch von anderen Modellen verwendet werden zu können.

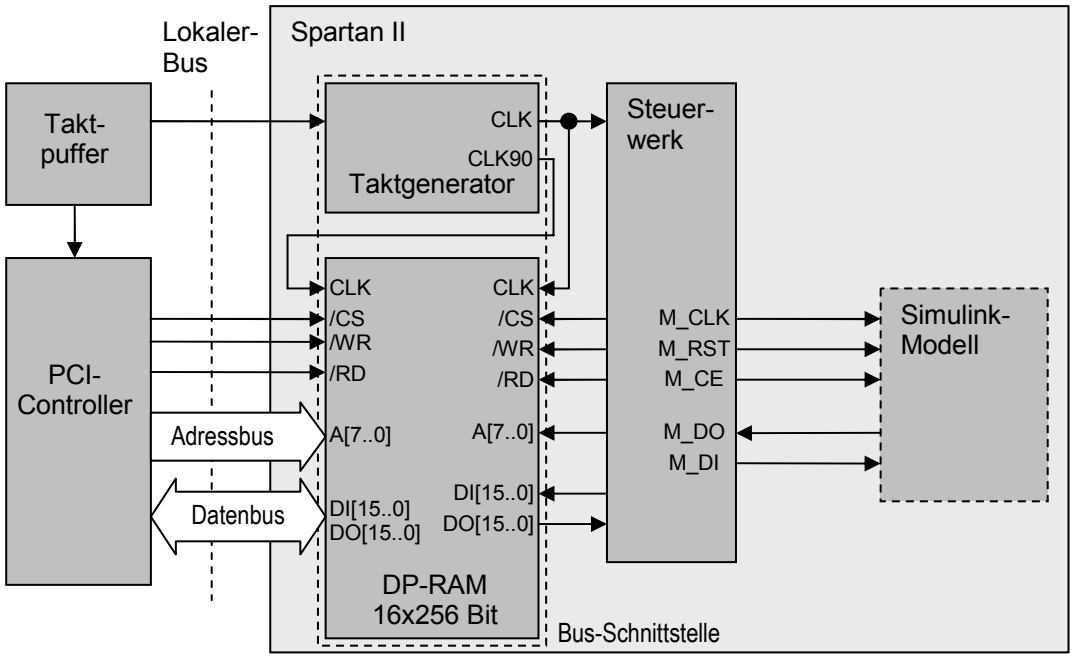


Bild 117: Das MATLAB-Modell innerhalb des FPGAs

Eine gemeinsame Bibliothek (shared library) von JNI stellt dem Teil des Simulators, der auf dem FPGA läuft, eine Schnittstelle zur Verfügung. Es übersetzt und leitet die Nachrichten von JStateSim weiter zum Kernelmodul. Deshalb sind zwei FIFOs für Kommunikationszwecke installiert und wer-

[60] DPRAM ist ein HW-Baustein, auf den über zwei Schnittstellen zugegriffen werden kann.

den am Ende eines Kommunikationsschrittes gelöscht. Ein Kernelmodul etabliert die Kommunikation zwischen dem Simulator und dem FPGA und verwendet die Funktionalität des PCI-Treibers und der FIFOs. Das Kernelmodul stellt zur Initialisierung und zum Aufräumen Funktionen zur Verfügung, die nach dem Laden bzw. Entladen des Moduls ausgeführt werden. Die anderen Funktionen stellen Lese- und Schreibzugriff für das DP-RAM sowie Kommandos zum Zurücksetzen des Modells and der Ausführung eines Simulationsschrittes zur Verfügung. Diese wiederkehrenden Operationen sind auf dieser niedrigen Ebene implementiert, um den Kommunikationsoverhead zu reduzieren. Die Bus-Schnittstelle ist die Verknüpfung zwischen dem lokalen Bus, dem Simulator und dem FPGA. Grundsätzlich besteht sie aus einem Taktgeber und einem Dual-Port-Speicher und wird mit Hilfe des Xilinx-CORE-Generators instanziiert. Dieser RAM-Typ wird verwendet, da der lokale Bus und der Simulator gleichzeitig vom RAM lesen bzw. in das RAM schreiben müssen.

4.8.1.2 Integration von RTAI

Das Realtime Application Interface [RTAI05] wurde am Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano entwickelt, um ein System für die am Institut anfallenden Echtzeitaufgaben aus dem Bereich der Luftfahrttechnik zu haben. Das System wurde 1998 größtenteils unter der LGPL-Lizenz (Lesser General Public License) [GNU05] als freie SW zur Verfügung gestellt. Im Rahmen dieser Arbeit werden RTAI-Funktionen verwendet, um im Bereich der Kommunikation eine möglichst schnelle Abarbeitung zu erreichen. Beim RTAI wurde Echtzeit-Funktionalität im Nachhinein in ein konventionelles BS integriert. Spezielle Teile des Kernels kümmern sich um die zeitkritischen Aufgaben, während wie gewohnt die „normalen“ Funktionen des Systems für zeitunkritische Aufgaben in vollem Umfang zur Verfügung stehen.

Zwischen den Echtzeitfunktionen und konventionellen Programmen müssen Kommunikationsmechanismen geschaffen werden, mit deren Hilfe Daten die Grenze zwischen Echtzeit- und konventionellem Bereich passieren können [Schw02]. Der Mechanismus, der bei der Implementierung der Kommunikation Verwendung findet, ist der der FIFOs. Ein FIFO ist ein unidirektionaler Lese-/Schreibpuffer, der für den asynchronen Transfer von Daten zwischen verschiedenen Prozessen benutzt wird. Da FIFOs für die Echtzeitkommunikation gedacht sind, werden sie im Folgenden RT-FIFOs (Real Time FIFOs) genannt. Ein RT-FIFO wird auf der einen Seite der Anwender-/Kernel-Adressraumgrenze zum Schreiben, auf der anderen Seite zum Lesen geöffnet. Die RT-FIFOs erlauben das Schreiben ohne auf das Lesen der Gegenseite zu warten. RT-FIFOs können folgend charaktisiert werden:

- RT-FIFOs können auf beiden Seiten des Anwender-/Kernel-Adressraums erstellt werden.
- RT-FIFOs können benannt und über den Namen referenziert werden.
- Die Größe von RT-FIFOs kann auch nach dem Erstellen geändert werden.
- FIFOs speichern Daten in einer Warteschlage. Anwendungen müssen sich um leere und volle FIFOs kümmern.
- Grenzen zwischen aufeinander folgenden Daten werden nicht aufrechterhalten. Anwendungen müssen sich um das Erkennen der Grenzen kümmern.
- RT-FIFOs erscheinen als Geräte `/dev/rtf0`…`/dev/rtf63` im Dateisystem. Es gibt keine obere Schranke für die verwendete Anzahl an FIFOs oder der Größe der in sie geschriebenen Daten.
- RT-FIFOs unterstützen die `/proc`-Dateisystem-Schnittstelle.
- RT-FIFOs unterstützen Semaphoren zur Synchronisation.
- RT-FIFOs unterstützen unterschiedliche Leser und Schreiber.
- RT-FIFOs unterstützen asynchrone Ereignissignale (Ankunft von Daten in einem leeren FIFO) [RTAI00].

In den beiden folgenden Quellcode-Fragmenten wird exemplarisch die Verwendung der FIFOs zur Kommunikation über die Kernelgrenze hinweg gezeigt.

```
int main(int argc, char** argv) {

  int fd1; /* lese-fifo */

  if ((fd1 = open("/dev/rtf1", O_RDONLY)) < 0) {
    fprintf(stderr, "CAL: error opening fifo /dev/rtf1\n");
    exit(1);
  }
  n = read(fd1, &von_fpga, sizeof(von_fpga));
  if (close(fd1) != 0)
      fprintf(stderr, "fifo /dev/rtf1 nicht geschlossen");
}
```

Bild 118: Lesen eines FIFOs im Anwenderadressraum

Bild 118 zeigt einen Ausschnitt aus dem Anwenderadressraum. Der FIFO wird, wie beim Umgang mit Dateien, zum Lesen geöffnet (`open("/dev/rtf1", O_RDONLY)`). Dann werden mit dem Befehl `read` die Daten ausgelesen, wonach der FIFO mit `close(fd1)`wieder geschlossen wird. Ähnliches geschieht im Kernelmodul (Bild 119). Hier werden jedoch die einzelnen Schritte (Erstellen, Schreiben/Lesen, Entfernen) in verschieden Methoden ausgeführt. Die speziellen Methoden `init_module` und `cleanup_module` werden beim Laden und Entladen des Moduls ausgeführt und eignen sich damit für das Erstellen und Entfernen der FIFOs. Ihre eigentliche Verwendung finden sie in anderen frei definierten Methoden. Im Quellcode in Bild 119 ist dies beispielsweise `schreibeFIFO()`.

```
#include <rtai_fifos.h>
#define fifo_s 1

void schreibeFIFO (){
  rtf_put(fifo_s, &zu_user, sizeof(zu_user));
}

int init_module(void) {

    rtf_create(fifo_s, 1024);  // Schreib-FIFO
    return 0;
}

void cleanup_module(void) {

  rtf_destroy(fifo_s);
}
```

Bild 119: Schreiben eines FIFOs im Kerneladressraum

4.8.1.3 Implementierung der Schnittstellen

FPGASim: Für den Einsatz in einer verteilten Simulation kann auf JStateSim zurückgegriffen werden. Da dieser komplett in Java implementiert ist, sind die nötigen Änderungen durch die objektorientierte Struktur einfach durchzuführen. Die Klassen, die sich um die Kommunikation mit den anderen Simulatoren kümmern, bleiben erhalten und geben somit die Schnittstellen vor. Ausgetauscht werden muss lediglich die Klasse, die die eigentliche Simulation durchführt. Sie wird durch eine Klasse ersetzt, die die Kommunikation zum FPGA über JNI aufbaut (Bild 120).

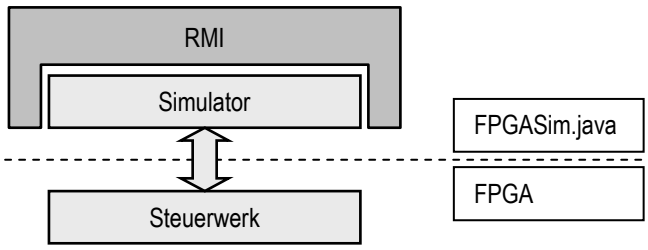


Bild 120: Änderungen bei JStateSim

In dieser Klasse wird zunächst nach den Vorgaben von JNI die verwendete Bibliothek mit `System.loadLibrary("fpga");` geladen (Bild 121). Da die Kommunikation mit dem Simulator über den Austausch einzelner Werte hinausgeht, werden die Daten über die Klasse `Simdaten` ausgetauscht. Erkennbar ist das bei der Deklaration der Methode `fpga` in Bild 121.

```
public class FPGASimulator extends Thread {

  public static native int fpga(Simdaten simdaten);
  static {  System.loadLibrary("fpga"); }

  public FPGASimulator(EventSimPar argument) {…}

    public void InitSimulator() {
    …
    simdt.befehl = RUECKSETZEN;
    simdt.nr_zu_fpga = 1;
    if (fpga(simdt) == 1) {
        System.out.println(simdt.nr_von_fpga);
        } else
            System.out.println(" FEHLER BEIM RUECKSETZEN ");
    }
    …
    public void run() {
      … // Aufruf der eigentlichen Simulation
    }
}
```

Bild 121: FPGASimulator.java

Die Klasse `Simdaten` (Bild 122) wird beim Aufruf der JNI-Bibliothek übergeben. Nach der Simulation werden ebenfalls über diese Klasse die Rückgabewerte übermittelt. Sie findet ihr Pendant in der C-Struktur `Nachricht` (Bild 123), die über die Kernelgrenze übergeben wird. In Bild 121 ist beispielhaft nur das Rücksetzen gezeigt, das während der Initialisierungsphase der Simulatoren ausgeführt wird. Die eigentliche Simulation erfolgt in gleicher Weise und ist in einem eigenen Thread implementiert. Sie wird somit in der Methode `run()`ausgeführt.

```
public class Simdaten {
    int befehl;
    int nr_zu_fpga;
    int daten_zu_fpga;
    int nr_von_fpga;
    int daten_von_fpga;
    public Simdaten(){}
}
```

Bild 122: Klasse "Simdaten"

Hier werden auch die Kommandos, die der Kontrollprozess den verschiedenen Simulatoren sendet, abgearbeitet. Eine Besonderheit im Hinblick auf die Zusammenarbeit mit dem Steuerwerk (Bild 122) ist die Behandlung der Simulationszeit. Dem Steuerwerk wird eine maximale Zeit übergeben, bis zu

der es das Modell ausführen darf. Zurück geschrieben wird die tatsächlich erreichte Zeit. Der Simulator muss also bei jedem Durchgang den Sprung in der Simulationszeit neu berechnen.

JNI-Bibliothek „fpga.c“: Mit jedem Aufruf dieser Bibliothek aus dem Simulator heraus wird ein kompletter Kommunikationszyklus abgearbeitet, bestehend aus dem Schreiben der Anfrage und Lesen der Antwort. Dafür werden zunächst zwei FIFOs für die Kommunikation mit dem Kernelmodul erstellt und am Ende wieder gelöscht. Mit `fd0 = open("/dev/rtf0", O_WRONLY);` wird FIFO `fd0` zum Schreiben, FIFO `fd1` mit der Option `O_RDONLY` zum Lesen geöffnet. Auf diesen kann mit `read` und `write` operiert werden. Beim Aufruf bekommt der Bibliothek vom Simulator eine Klasse übergeben, die den Daten- und Steuerfluss kapselt. Auf diese Werte muss nun zugegriffen werden. JNI stellt dafür Möglichkeiten bereit. Mit `jclass jclass = (*env)->GetObjectClass( env, obj );` wird die Klassendefinition in `jclass` geschrieben. Um auf die einzelnen Attribute der Klasse zuzugreifen, muss zunächst der Zeiger auf den Speicherplatz geholt werden:

```
jfieldID jfid;
jfid = (*env)->GetFieldID( env, jclass, "befehl", "I");.
```

Der zweite Parameter `jclass` ist ein Zeiger auf die Klassendefinition. Der dritte Parameter kennzeichnet den Namen der Variablen, der letzte Parameter definiert den Typ der Variablen. In diesem Fall steht „I“ für Integer. Der letzte Schritt ist das Auslesen bzw. Setzen der Werte. Mit

```
jint befehl;
befehl = (*env)->GetIntField( env, obj, jfid);
```

wird der Variablen `befehl` der Wert der entsprechenden Klassenvariablen übergeben. Das Schreiben funktioniert auf ähnliche Weise. Nachdem der Zeiger der gewünschten Variable geholt wurde, kann mit `(*env)->SetIntField(env, obj, jfid, (jint)von_fpga.nr);` der Wert geschrieben werden.

Die Strukturen `zu_fpga` und `von_fpga` sind vom Typ `nachricht` (Bild 123) und kapseln ähnlich wie die Klasse `Simdaten` (Bild 122) die Kommunikation zum Kernelmodul. Die Variable `command` enthält ein Kommando an das Kernelmodul. Bisher sind die Befehle „Lesen und Schreiben von FPGA-Registern“, „Rücksetzen des Modells“ und „Ausführen eines Simulationsschrittes“ implementiert. Der Wert `adresse` wird nur bei Lese- und Schreibkommandos gebraucht. Die beiden anderen Befehle greifen selbständig in richtiger Reihenfolge auf die nötigen Speicherplätze zu. Die Variablen `wert` und `nr` enthalten die Daten, die zum Modell geschickt werden sollen, und eine Nummer, die den Simulationsschritt eindeutig kennzeichnet. Bei der Antwort enthalten diese beiden Werte die Ergebnisse der Simulation.

```
struct nachricht {
  unsigned long command;
  unsigned int adresse;
  unsigned long var1;
  unsigned long var2;
  unsigned long var3;
  unsigned long nr;
};
```

Bild 123: Struktur „Nachricht“

Um für JNI verfügbar zu sein, muss `fpga.c` als dynamisch ladbare Bibliothek compiliert werden. Da auf dem RP-System Linux als Betriebssystem läuft, wird hierfür der frei verfügbare GNU-Compiler `gcc` verwendet [HeAr04]. Bild 124 zeigt beispielhaft den Compileraufruf von der Kommandozeile. Die Option `-shared` gibt an, dass die Ausgabe eine dynamische Bibliothek sein soll. Mit `-Wall` werden alle Warnungen ausgegeben. Die mit `-I` eingebundenen Suchpfade für die Header-Dateien sind vom Aufbau des jeweiligen Systems abhängig. Mit `fpga.c` wird dem Compiler der

Name der zu compilierenden Datei übergeben. Bei dem mit `-o` angegebenen Name der Ausgabedatei ist folgende Konvention zu beachten: Dynamische Bibliotheken erfordern das Präfix „lib“ sowie die Dateierweiterung „.so“. Nur so findet der Aufruf `System.loadLibrary("fpga");` in Java die richtige Bibliothek.

```
gcc    -Wall -shared
-I /usr/java/j2sdk1.4.1_01/include
-I /usr/java/j2sdk1.4.1_01/include/linux
-I /home/dummer/fpga_test
-I /home/dummer/fpga_test/jni
fpga.c -o libfpga.so
```

Bild 124: Compileraufruf für JNI-Bibliotheken

Kernelmodul „fpga_sim.c“: Beim BS des RP.2002 handelt es sich um ein Linux der SuSE-Distribution. Neue Kernelmodule können also dynamisch hinzugefügt und wieder entfernt werden. Das für die Kommunikation zwischen Simulator und FPGA entworfene Kernelmodul benutzt Funktionalitäten der Module „plx9030“ (PCI-Treiber), „rtai_fifos“ (FIFOs) und „rtai“, die somit vorher geladen sein müssen.

Als Kernelmodul besitzt `fpga_sim.c` zunächst zwei Funktionen `init_module` und `clean-up_module`, die beim Laden bzw. Entladen des Moduls ausgeführt werden. Hier werden vor allem die FIFOs für die Kommunikation angelegt und wieder entfernt. Als Funktionalität bietet es dem Anwender zunächst zwei Funktionen zum Lesen und Schreiben auf bestimmte Register des Modells im FPGA; desweiteren zwei für die Simulation wichtige Funktionen des Rücksetzens des Modells und des Ausführens eines Simulationsschrittes. Diese immer wiederkehrenden Befehle sind schon auf dieser Ebene realisiert, um das Kommunikationsaufkommen zu reduzieren. Sie können jedoch bei Bedarf auch vom Simulator nachgebildet werden.

Um auf den Speicher im FPGA zuzugreifen, der über einen PCI-Controller am PCI-Bus angekoppelt ist (Bild 117), nutzt das Modul die beiden Funktionen des plx9030-Kernelmoduls

`plx9030_read(card, base, *data, offset, size)` und
`plx9030_write(card, base, *data, offset, size)`.

Den Funktionen werden die Kartennummer und die Basisadresse als Parameter übergeben. Die Daten zum Lesen oder Schreiben werden als Zeiger auf den Speicherbereich `*data` übergeben. Beide Funktionen beginnen an der Stelle `offset` und behandeln Größen von `size` Bytes. Die beiden letzten Parameter sind somit von der Implementierung des Speichers auf dem FPGA abhängig. Bild 125 beschreibt die Abbildung von `offset` auf die Register. Sie ist in ihrer Belegung mit dem Steuerwerk (Bild 130) abgestimmt. Da jede Speicherstelle auf dem FPGA 16 Bit umfasst, ist der Parameter `size` eine Konstante der Größe zwei.

```
#define CONTROL_REG 0x0
#define ANFRAGE_REG 0x02
#define ANFRAGE_NR_REG 0x04
#define ANTWORT_REG 0x08
#define ANTWORT_NR_REG 0x0A
```

Bild 125: Registerdefinition

Um aus `fpga_sim.c` ein Kernelmodul zu erzeugen, müssen beim Compileraufruf einige Optionen gesetzt sein (Bild 126). Die Option `-D__KERNEL__` wird gebraucht, um auf interne Kerneldefinitionen in den Header-Dateien zugreifen zu können. `-DMODULE` definiert die Flags für Module. Dadurch wird dem Modul der Zugriff auf Funktionen des laufenden Kernels ermöglicht. Die Option `-O2` setzt die Optimierungsstufe so, dass fast alle Optimierungen ausgeführt werden. Die Performance des ge-

nerierten Codes wird somit gesteigert. Die übrigen Optionen haben dieselbe Bedeutung wie oben erwähnt.

```
gcc    -D__KERNEL__ -DMODULE -O2 -Wall
-I/lib/modules/2.4.17-rthal5/build/include
-I/home/rp/rtai/include/
-I/home/rp/io/plx9030/
-I/home/dummer/fpga_test/
fpga_sim.c -o fpga_sim.o
```

Bild 126: Compileraufruf für Kernelmodule

Der erstellte Objektcode kann mittels `insmod` in den laufenden Kernel geladen und mit `rmmod` wieder entladen werden. Zum Ausführen dieser Befehle muss der Anwender über Administratorrechte im System verfügen. Für das Einhalten der Abhängigkeiten unter den Kernelmodulen hat es sich als sinnvoll erwiesen, das Laden bzw. Entladen von einem Skript ausführen zu lassen. Das Skript zum Entladen der Module entfernt sie in umgekehrter Reihenfolge. Mit dem `sync`-Befehl werden veränderte Datenblöcke aus dem Cache unabhängig vom automatisch laufenden `update`-Dienst auf die Festplatte zurück geschrieben. Er dient der Sicherheit vor Datenverlust.

Ausgaben der Kernelmodule werden nicht auf die Standardausgabe sondern in eine Protokolldatei geschrieben. Für die Funktionsüberwachung und für die Fehlersuche kann man sich mit `tail –f /var/log/messages` das ständig aktualisierte Ende diese Datei anzeigen lassen (gilt für die verwendete SuSE-Distribution).

Bus-Schnittstelle: Da für eine sichere Kommunikation an einer Hardwareschnittstelle das Einhalten von Zeitbedingungen unumgänglich ist, soll im Folgenden näher auf das Timingdiagramm (Bild 127) beim Lese- bzw. Schreibzugriff des PCI-Controllers auf das RAM im FPGA eingegangen werden. Der Takt des lokalen Busses entspricht mit einer Phasenverschiebung dem des PCI-Busses. Zum Schreiben legt der PCI-Controller die Adressen und Daten auf den Adress- und Datenbus und setzt `chip select` (`CS`) und `read` (`RD`). Mit der nächsten steigenden Taktflanke werden die Daten in die Adresse im RAM geschrieben. Beim Lesen setzt er `CS` und `WR` und legt die Adresse auf den Adressbus. Bei der zweiten steigenden Taktflanke übernimmt er die Daten vom Datenbus.

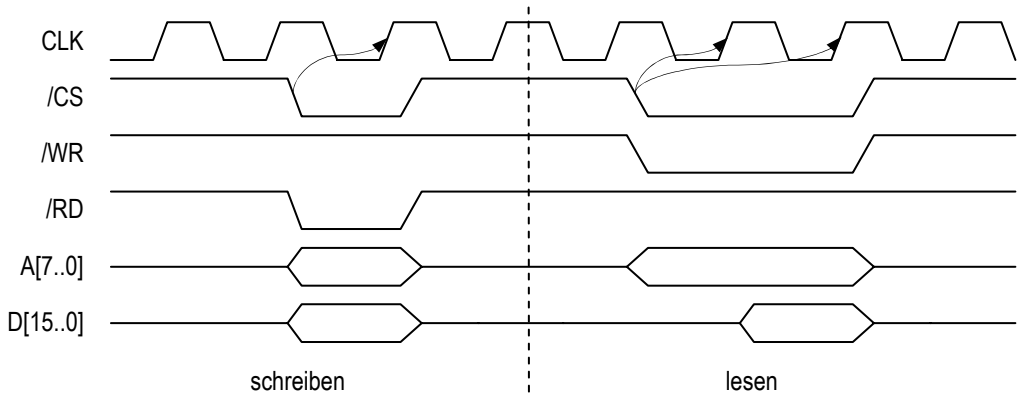


Bild 127: Schreib- und Lesezugriff

Die Bus-Schnittstelle (Bild 128) übernimmt die Kommunikation zwischen lokalem Bus und dem Simulator mit angehängtem Modell im FPGA und besteht im Wesentlichen aus einem DP-RAM und einer Taktaufbereitung. Als RAM (16 x 256 Bit) wurde mittels Xilinx-CORE-Generator ein Xilinx-IP-Core parametrisiert und instanziiert. Er ist über eine Anpasslogik mit dem lokalen Bus verbunden und verhält sich, da das `READY`-Signal (`dp_ready`) konstant `HIGH` ist, an diesem rein passiv. Der einzige aktive Teilnehmer ist der PCI-Controller.

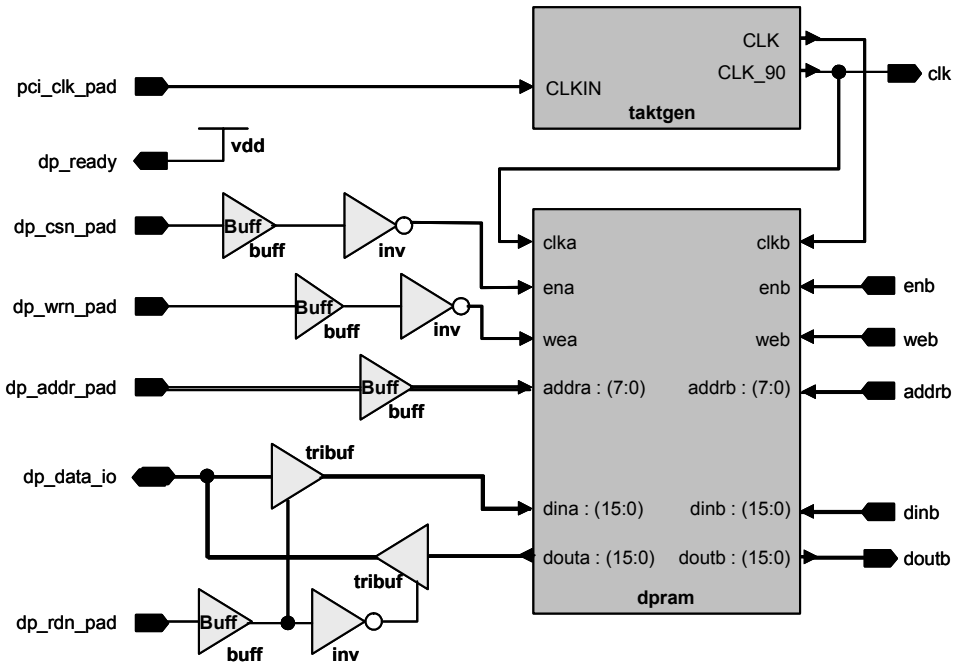


Bild 128: Bus-Schnittstelle

Das DP-RAM besitzt für jede Seite separate Ein- und Ausgänge. Diese sind nur insofern unabhängig, als dass nur gleichzeitiges Lesen auf dieselbe Adresse erlaubt ist (Gleichzeitigkeit bezieht sich auf die steigende Taktflanke an `clka` und `clkb`). Alle anderen gleichzeitigen Zugriffe führen zu ungültigen Daten. Es muss also eine Konfliktauflösung geben, die die eben genannten Zustände ausschließt. Dafür kommen verschieden Möglichkeiten in Betracht. Der verwendete IP-Core bietet die Möglichkeit, einen Ausgang `RFD[A|B]` (Ready for data) zu instanziieren, der angibt, ob Daten geschrieben werden können oder nicht. Eine andere Möglichkeit besteht darin, für `clka` und `clkb` Frequenzen unterschiedlicher Periodendauer zu benutzen [XiSp01]. Für diese Arbeit wurde eine dritte Möglichkeit verwendet, da sie zum einen in ähnlicher Weise schon in vorherigen Arbeiten erfolgreich eingesetzt wurde [Zöll02] und andererseits in ihr Standardkomponenten des FPGAs verwendet werden. Laut Spezifikation [XiSp02] reicht eine Zeitdifferenz T_{BCCS} (`clka` → `clkb`) von 4 ns aus, um sicher auf dieselbe Speicherzelle zugreifen zu können. Des Weiteren besitzt der verwendete SpartanII spezielle Funktionsblöcke zur Taktaufbereitung. Diese ermöglichen den Eingangstakt mit einer Phasenverschiebung von 0°, 90°, 180° und 270° über den FPGA zu verteilen sowie zu vervielfachen und zu teilen. Bei einem Bustakt von 33 MHz reicht eine Phasenverschiebung von 90° (≅ 7,5 ns) zwischen `clka` und `clkb` aus, um die Zeitbedingung zu erfüllen. Bild 129 zeigt die Umsetzung. Es handelt sich dabei um eine Standardschaltung aus [XiSp01]. Die Bausteine sind Primitive aus der `Unisim_vital`-Bibliothek von Xilinx und können über die mitgelieferten so genannten „Generics“ simuliert werden.

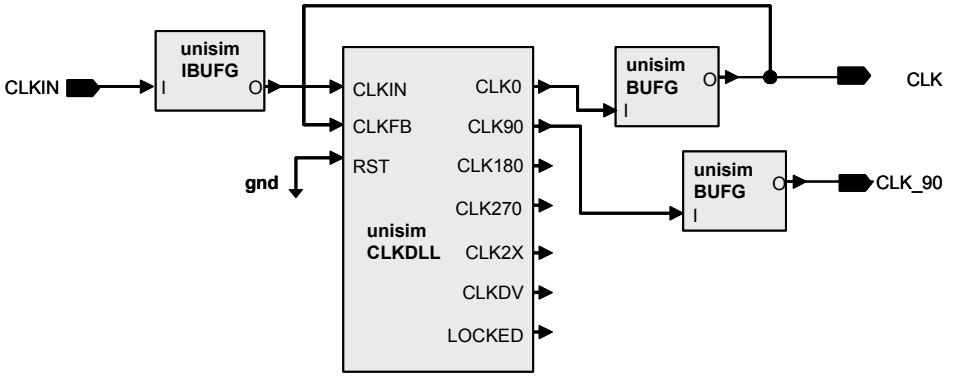


Bild 129: Taktgenerator

Steuerwerk: Das simulierte Modell ist an das Steuerwerk angefügt. Per Standard besitzt das Steuerwerk einen Takteingang sowie einen synchronen Reset- und Freigabe-Eingang. Datenein- und ausgänge sind abhängig vom jeweiligen Modell. Das Steuerwerk ist der Teil des Simulators, der in HW implementiert ist. Entsprechend den Daten, die im RAM gespeichert werden, stellt es dem simulierten Modell Steuersignale und Daten zur Verfügung. Das Steuerwerk besteht aus unterschiedlichen Zuständen. Der initiale Zustand `warten` liest solange das Kontrollregister (Adresse: `0x00`) aus, bis sich Bit 0 verändert. In diesem Fall setzt der Zustand `sperren` das Bit 3 im Kontrollregister, um zu verhindern, das die SW ungültige Werte liest oder der Steuereinheit neue Kommandos sendet. Danach liest der Zustand `lesen` die Variablen und übergibt sie dem Modell. Weiterhin wird eine maximale Simulationszeit von einem bestimmten Register im RAM ausgelesen. Der Zustand `ausfuhren` simuliert das Modell einen Zeitschritt lang und inkrementiert die Zeit um eins. Der Zustand `weiter` entscheidet, ob die Simulation fortgeführt werden kann oder nicht. Abbruchbedingungen sind die Simulationszeit und modifizierte Variablen. Anschließend schreibt der Zustand `schreiben` die Resultate und die Simulationszeit zurück. Nach dem Rücksetzen von Bit 3 im Zustand `freigeben` kehrt die Steuereinheit zum Anfangszustand zurück und startet einen neuen Zyklus. Die Struktur der Steuereinheit entsteht aus den Anforderungen der verteilten Simulation und des Modells. JStateSim stellt die Anforderung an die Steuereinheit, das Modell bis zu einer bestimmten Zeit zu simulieren. Wenn sich die Ausgangsvariablen ändern, muss die Simulation abgebrochen werden und die neuen Werte und der Zeitstempel müssen dem Kontrollprozess übergeben werden. Zustandsautomaten eignen sich für die Beschreibung dieser Anforderungen in VHDL. Die Zustandsaktionen (state actions) und Bedingungen (conditions) sind VHDL-Anweisungen. Die eingekreisten Zahlen `warten` und `weiter` am Anfang der ausgehenden Transitionen repräsentieren die Prioritäten. In Bild 130 wird nur eine Variable beispielhaft überprüft. Die Entscheidung, den Status der Steuereinheit in ein Zustandsbit zu schreiben, kann zukünftig durch eine Implementierung eines Unterbrechungs-Mechanismus für das FPGA des RP.2002 revidiert werden.

Der Aufbau des Steuerwerks ergibt sich aus den Anforderungen der verteilten Simulation und des Modells. In der verteilten Simulation verlangt JStateSim, dass das Modell eine bestimmte Zeit lang weiter simuliert werden kann. Ändern sich in dieser Zeit die Ausgänge des Modells, soll die Simulation abgebrochen und die Zeit und Werte den entsprechenden Prozessen übergeben werden. Diese Aufgabe wird direkt in Hardware realisiert, um eine möglichst hohe Geschwindigkeit zu erreichen. Die Zeit, bis zu der maximal simuliert werden darf, wird im Zustand `nr_le` gelesen. Sie wird im Zustand `weiter` als Abbruchkriterium mit einer Laufvariablen verglichen. Ein weiteres Abbruchkriterium ist die Änderung der Ausgänge des Modells. In diesem Fall wird exemplarisch nur eine Variable (`VAR1`) überprüft.

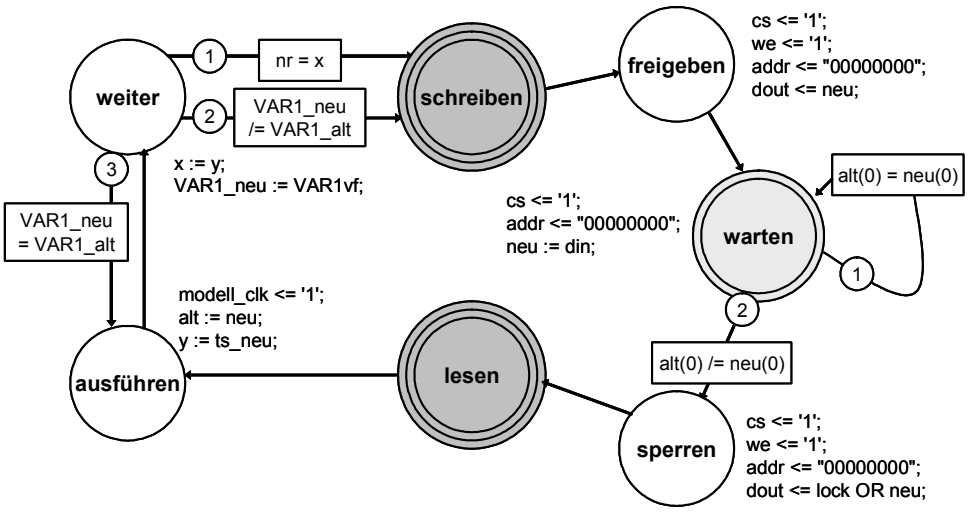


Bild 130: Steuerwerk

Die anderen Variablen veranschaulichen den linearen Aufwand bei Zunahme der Variablen. Die Entscheidung, dem Simulator über ein Zustandsbit in der Steueradresse den aktuellen Zustand des Steuerwerkes zu übergeben, resultiert aus der fehlenden Implementierung eines Interrupts für den FPGA auf dem RP.2002. Der FPGA ist bei diesem RP-System nicht mit dem Interrupteingang des Prozessormoduls verbunden. Das Verfahren der expliziten Abfrage (polling), das deswegen angewendet wird, hat den Nachteil des aktiven Wartens (permanente Busaktivität und Prozessorlast). Damit bei einem Fehler der Kerneltreiber des Simulators nicht die gesamte Prozessorleistung beansprucht, ist die Abfrage zeitlich begrenzt. Des Weiteren wird nach jeder Abfrage für eine kurze Zeit die CPU für andere Prozesse freigegeben.

Da sich für die Beschreibung dieser Aufgabe endliche Automaten gut eignen, wurde diese Möglichkeit genutzt. Nach der Definition aus 2.2.5.1 handelt es sich hierbei um einen Moore-Automaten. Die Entwicklungsumgebung FPGA-Advantage bietet für die Arbeit mit Automaten einen graphischen Editor an. FPGA-Advantage generiert aus diesem Zustandsdiagramm automatisch den entsprechenden VHDL-Code. Es muss dazu zunächst ein Takteingang spezifiziert werden. Bei Verwendung von Variablen, in diesem Fall zum Speichern von Werten, scheint es notwendig zu sein, den Automaten in 2 anstatt in vorgegebenen 3 Prozessen generieren zu lassen (der Prozess für die Ausgabe und den nächsten Zustand werden dabei zusammengefasst). Ansonsten treten trotz korrekter Deklaration der Variablen entsprechende Fehlermeldungen beim Compilieren auf.

Als entscheidend für die korrekte Funktion des Steuerwerkes hat sich die Art der Kodierung der Zustände herausgestellt. FPGA-Advantage bietet hierfür mehrere Möglichkeiten an. Je nach Anzahl der verwendeten Zustände hatte die Wahl der Zustandsdekodierung Fehler bei der Verdrahtung (nicht eingehaltene Zeitbedingungen), ein nicht funktionierendes Modell oder ein korrekt arbeitendes Modell zur Folge.

Steueradresse: Mit Hilfe der Bits 0 bis 3 der Steueradresse kann das Modell angesteuert werden. Ein Umschalten von Bit 0 erzeugt einen Taktimpuls am CLK-Eingang des Modells. Mit gesetzten Bits 1 und 2 werden die Reset- und Enable-Eingänge des Modells auf „1“ gesetzt. Bit 3 zeigt dem Simulator, ob das Steuerwerk bereit für einen Simulationsschritt ist. Die restlichen Bits sind frei und können anderweitig benutzt werden (Bild 131).

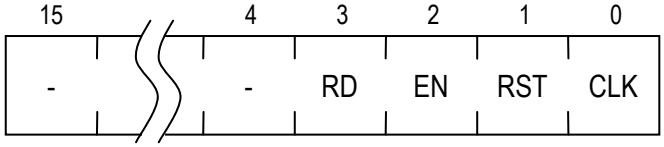


Bild 131: Bitbeschreibung der Steueradresse

Speicherbelegung: Die Bedeutung der Speicherstellen im RAM ist vom Steuerwerk vorgegeben. Tabelle 8 beschreibt die verwendeten Adressen. Die Zuordnung ist beliebig und spiegelt sich in den dem PCI-Treiber übergebenen Adressen wieder (Bild 125).

Adresse	Beschreibung
0x0	Steueradresse
0x1	Daten zum Modell (Variable 1)
0x2	Daten zum Modell (Variable 2)
0x3	Daten zum Modell (Variable 3)
0x4	Simulationsnummer zum Modell
0x5	Daten vom Modell (Variable 1)
0x6	Daten vom Modell (Variable 2)
0x7	Daten vom Modell (Variable 3)
0x8	Simulationsnummer vom Modell

Tabelle 8: Speicherbelegung im RAM

Programmierung des FPGA: Die in den vorangegangenen Abschnitten beschriebenen Module des FPGA werden zusammengefügt (Bild 132).

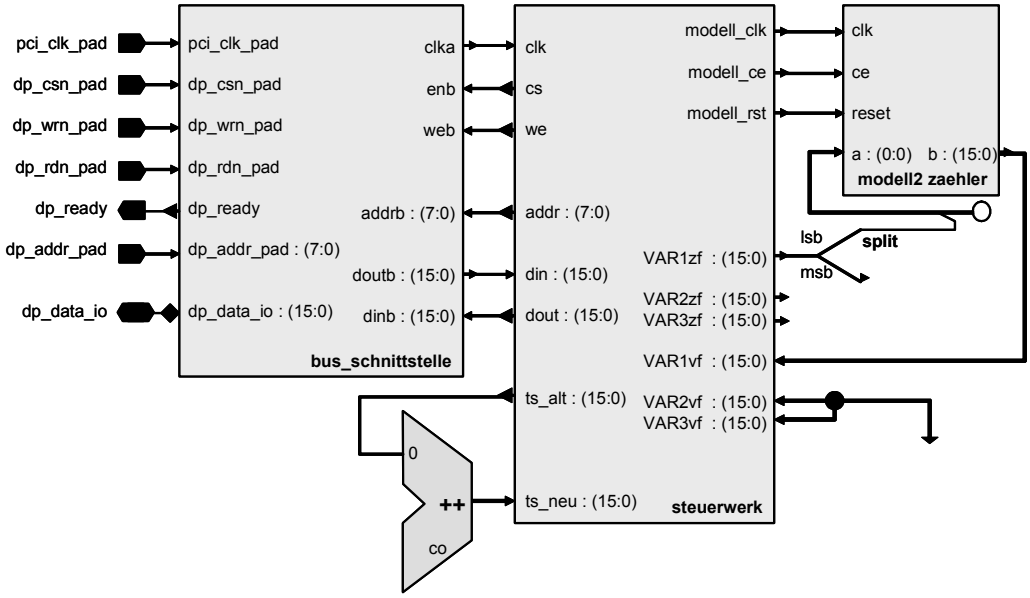


Bild 132: Konfiguration des FPGAs

Mit den Anschlüssen auf der linken Seite greift das Modul `bus_schnittstelle` auf den lokalen Bus zu. Sie müssen also auf die entsprechenden Pins des FPGA abgebildet werden. Diese Aufgabe übernimmt die Constraints-Datei (Bild 133). In dieser Datei werden auch die vom Taktgenerator (Bild 129) verwendeten Elemente denen des FPGA zugeordnet. Den mit `NET` bezeichneten Signalen entsprechen die Signalbezeichnungen aus Bild 132. Die einzelnen Signale eines Busses werden durchnummeriert. Mit dem `TNM_NET-` und `TIMESPEC-`Attribut wird die Frequenz am Eingang `pci_clk_pad` mit 30 MHz und einem Tastverhältnis von 50 % spezifiziert.

Für die Konfiguration des FPGAs kommt ein Skript zum Einsatz. Es verlangt als Eingabe eine Konfigurationsdatei (*.rbt) im ASCII-Format. Diese wird vom Platzierungs- und Verdrahtungswerkzeug erstellt und mittels FTP auf das RP-System übertragen. Die Übertragung der Daten zum FPGA verläuft ähnlich der Kommunikation des Simulators mit dem FPGA. Zunächst wird ein Kernelmodul geladen, das die FIFOs installiert und die Kommunikation mit dem PCI-Treiber übernimmt. Ein C-Programm liest die .rbt-Datei ein und schreibt den Datenstrom in die FIFOs. Abschließend wird das Kernelmodul wieder entladen. Es besitzt dieselben Abhängigkeiten wie `fpga_sim.c`. Um eine sichere Arbeit des Simulators zu gewährleisten, muss dessen Kerneltreiber nach jeder Konfiguration des FPGA entladen und wieder geladen werden.

```
NET "dp_addr_pad(0)" LOC = "P3";
NET "dp_addr_pad(1)" LOC = "P4";
...
NET "dp_addr_pad(6)" LOC = "P9";
NET "dp_addr_pad(7)" LOC = "P10";
...
NET "pci_clk_pad" LOC = "P80";
NET "pci_clk_pad" TNM_NET = "pci_clk_pad";
TIMESPEC "TS_pci_clk_pad" = PERIOD "pci_clk_pad" 30 ns HIGH 50 %;
NET "pci_clk_pad" LOC = GCLKPAD0;
#INST I1_notri_I0_I1 LOC = DLL0;
#INST I1_notri_I0_I5 LOC = GCLKBUF0;
#INST I1_notri_I0_I4 LOC = GCLKBUF2;
```

Bild 133: Ausschnitt aus Constraints-Datei

Synchronisation: Die Vorgaben von JStateSim erlauben nur zyklenfreie Modelle, so dass keine Verklemmungen auftreten können [Fuji00]. Als Kontrollprozess kommt der gleichnamige Prozess von JStateSim zum Einsatz. Von ihm erhalten alle Simulatorinstanzen über RMI globale Befehle, wie Initialisierung der Modelle, Starten und Stoppen der Simulation sowie Beenden der LPs. Die Mächtigkeit der Menge P ist hier im Beispiel von Bild 80 mit n gegeben. Die LPs stellen nach der Partitionierung des Gesamtmodells die Einheiten von Simulatorinstanz SE und Teilmodell R dar. Dies kann z. B. der Aufruf von JStateSim und der ihm übergebenen Modelldatei, aber auch der FPGA-Simulator im Zusammenspiel mit dem konfigurierten FPGA sein. Hauptsächliches Kommunikationssystem zwischen den LPs ist ebenfalls RMI, zu dem jede Simulatorinstanz eine Schnittstelle CI haben muss. Über dieses werden zeitmarkierte Botschaften (m, t_m) zwischen den einzelnen LPs ausgetauscht. JNI, wie es in dieser Arbeit verwendet wird, gehört nicht zur Kommunikationsschnittstelle nach Bild 80, sondern wird zwischen dem Simulator SE und dem Modell R eingesetzt.

5 Ergebnisse

5.1 Analyse kommerzieller OSEK/VDX-Betriebssysteme

Die wichtigsten Aspekte von RTOS, speziell OSEK, sowie Aspekte integrierter Entwicklungsumgebungen dienen als Basis, wovon Anforderungen an BS und Richtlinien für Tests abgeleitet werden. Messungen, Tests, Methoden und Kriterien für eine RTOS-Evaluierung werden spezifiziert, und wesentliche Eigenschaften existierender BS-Implementierungen identifiziert; außerdem werden Charakteristiken, Zeit- und Speicheranforderungen untersucht [DrMG04]. Es ist wichtig, nicht ein bestimmtes BS als das Beste darzustellen, sondern herauszufinden, welches am geeignetsten für die Aktivität ist, die es ausführen soll [Barr02]. Die evaluierten OSEK-Derivate sind von den Herstellern Vector Informatik, Wind River[61], LiveDevices[62], Metrowerks und 3SOFT[63].

In Anbetracht der Umfänge und der engen Endbenutzer-Preiskalkulation eingebetteter Systeme ist RAM eine kostbare Ware. Das BS muss diesen Speicher effizient nutzen. Der RAM-Bedarf erhöht sich nicht linear mit der Größe der Applikation, weshalb er bei einer großen Applikation weniger ins Gewicht fällt. Der zusätzliche Aufwand eines RTOS hinsichtlich ROM hängt von der Anzahl der verwendeten Funktionen ab. Eine minimale Instrumentierung muss während des Build-Vorgangs des Systems hinzugefügt werden. Emulatoren, Werkzeuge zur Fehlersuche im Hintergrund und monitorbasierte Debugger können nicht ohne Unterstützung arbeiten.

Ein Hauptmangel beim Testen sicherheitskritischer Systeme mit dynamisch allokiertem Speicher ist, dass keine vollständige Testfolge erstellt werden kann, um jede mögliche Kombination von Ereignissen und Eingaben abzudecken, die einen möglichen Fehler erzeugen könnten. Ein OSEK OS ist ein statischer Kern, d. h. eine OSEK-Applikation wird zur Compilierungszeit mit der OIL konfiguriert und alle Kernobjekte sind zur Compilierungszeit bekannt. Als Ergebnis wird die Menge an Tests, die benötigt wird, um ein korrektes Verhalten der Applikation sicherzustellen, reduziert. Das ist wichtig für Applikationen, die eine Zertifizierung benötigen.

5.1.1 Anforderungen und Kriterien

Ein gutes RTOS besteht nicht nur aus einem guten Kernel, sondern sollte gut dokumentiert sein und mit guten Werkzeugen für die Anwendungsentwicklung und Feineinstellung ausgeliefert werden [Barr02]. BS eignen sich für verschiedene Zwecke unterschiedlich gut. Deshalb sind Auswahlkriterien wichtig, wobei es Kriterien geben kann, die in jedem Fall erfüllt werden müssen. Abhängig von der Applikation hat ein RTOS, das viele Geräte unterstützt, manchmal mehr Vorteile als ein einfacher, sehr guter, kleiner Kernel. Hauptmerkmale existierender BS-Implementierungen werden identifiziert und analysiert. Ihre Charakteristiken, Speicher und Zeitanforderungen werden untersucht. Wichtig sind die Verfügbarkeit der OSEK-API-Implementierungen der entsprechenden Hersteller, ihre Versionen, ob stattdessen proprietärer Ersatz verfügbar ist, unterstützte Conformance Classes, zertifizierte Targets, minimale Kernelgröße, Verfügbarkeit eines Analysewerkzeugs zur statischen Überprüfung des Stack-Verbrauchs sowie die Charakteristiken des zugehörigen OIL-Konfigurationswerkzeugs. Dies sollte benutzerfreundlich und GUI-basiert sein, automatische Tests nach Vollständigkeit und Konsistenz sowie Projekt-Arbeitsbereiche, die die verschiedenen Dateitypen und Pfade zusammenfassen, erlauben. Mehr als eine OIL-Datei sollte gleichzeitig geöffnet sein können, und Tasks und Ressourcen sollten dupliziert werden können. Die Unterstützung von Eigenschaften in Bezug auf zeitliche Vorhersehbarkeit ist wichtig.

[61] Tornado von Wind River Systems ist eine Entwicklungsumgebung für elektronische Geräte im Automobil. Die Plattform enthält das RTOS OSEKWorks [Seru02].

[62] Realogy Real-Time Architect (RTA) von LiveDevices (ETAS) basiert auf dem RTOS SSX5. RTA verfügt über Weiterentwicklungen der OSEK-Spezifikation, z. B. „leichtgewichtige“ Tasks mit Reduzierung der Stack-Anforderungen je Task auf rund ein Drittel des standardmäßigen Task-Stack-Bedarfs [Trau02].

[63] Diese Reihenfolge korreliert aus Gründen der Geheimhaltung absichtlich nicht mit der Nummer der RTOS.

Implementierungen können auf eine bestimmte Plattform eingeschränkt sein. Beispielapplikationen sollten zur Verfügung gestellt werden. Die Qualität eines BS hängt gewöhnlich davon ab, wie lange das BS auf dem Markt etabliert ist und auf welchem RTOS es basiert. Weitere nicht-funktionale Kriterien sind die Antwortzeit und die Antwortqualität des Hersteller-Supports. Verfügbar sein sollten: Analysewerkzeuge und eine Laufzeitfehler-Überprüfung zur Erkennung von Zeiger- und Speicherverwaltungsfehlern, ein Profiler, ein Analysator für den Stackverbrauch, um die maximale Stackgröße zu ermitteln, ein Analysator zur Erzeugung der Bindetabelle, zum Editieren und zur Optimierung von Speichertabellen-Setups, und ein Compiler-Optionsdialog für die Verwaltung von Compiler-Einstellungen.

Wird eine Applikation auf einem Prozessor ausgeführt, der eine Fließkommaeinheit (Floating Point, FP) besitzt oder kommen FP-SW-Bibliotheken zum Einsatz, und es wird somit Gebrauch von FP-Operationen gemacht, ist es wichtig zu verstehen, welche RTOS-Unterstützung für FP angeboten wird. I. A. sind drei Ansätze möglich:

- Das RTOS behandelt den FP-Registersatz wie die Register, die bei Kontextumschaltungen gespeichert werden. Das erlaubt jeder Task die Verwendung von FP. Ein Overhead, sowohl bei der Laufzeit als auch im Stack-Verbrauch, entsteht für Tasks, die kein FP verwenden.
- Das RTOS speichert nicht automatisch den FP-Registersatz beim Umschalten des Kontextes. In diesem Fall kann nur eine Task oder eine Unterbrechung FP-Verarbeitung einsetzen.
- Der Anwendungsentwickler sorgt dafür, dass der FP-Kontext gegen Präemptierung der Task oder Unterbrechungen geschützt wird.

Das RTOS und der Compiler müssen perfekt zusammen passen. Das ist eine harte Anforderung, besonders wenn keine Modifikationen des Quellcodes gemacht werden können. Die BS-Hersteller empfehlen insbesondere bestimmte Compiler aber lassen i. A. die Anwender über die IDE, den Debugger und die BDM-HW entscheiden. Wenn das BS in Form von Bibliotheken und nicht als Quellcode geliefert oder durch ein Konfigurationswerkzeug generiert wird, können nur die spezifizierten Compiler verwendet werden. Kommerzielle Compiler für ein bestimmtes Target generieren stärker optimierten Code für dieses Target. Je weniger Hersteller involviert sind, desto effizienter sind i. A. die Problemlösungen und desto kleiner ist das Risiko von Inkompatibilitäten. Nicht alle BS- und Debugger-Hersteller bieten eine integrierte Entwicklungsumgebung an. Wenn diese nur von Drittherstellern verfügbar oder nicht vom BS-Hersteller getestet ist, sollte der Aufwand, mögliche Inkompatibilitäten auszuräumen, mitberücksichtigt werden. Die Bewertung der Werkzeugkette kann auf der Anzahl der Hersteller, der OSEK-Awareness und der Kompatibilität mit speziellen Code-Generatoren für Optimierungszwecke, z. B. mit TL, basieren. Wenn die Vektortabelle vom OIL-Konfigurator generiert wird, kann sie implizit durch Modifikation der OIL-Konfiguration modifiziert werden, und es wird nicht empfohlen, die Vektortabelle mit Hilfe der IDE zu generieren. Die Sprache C ermöglicht es, direkt auf die HW zuzugreifen, sowie effiziente OSEK-basierte Systeme mit niedrigem Speicherbedarf zu entwickeln. Aber das Modifizieren der Umgebung, z. B. der CPU-Register ist riskant, da man sich den Modifikationen durch das BS und den Auswirkungen von Modifikationen auf das BS bewusst sein sollte, so dass Konflikte vermieden werden. Nur diejenigen Kriterien, die eine Differenzierung der OSEK-RTOS-Derivate ermöglichen, sind für einen Vergleich nützlich.

Ein Debugger, mit dem eine OSEK-Applikation getestet werden soll, muss wissen, wie Informationen über das BS zu beschaffen sind, d. h. der Debugger muss u. a. Kenntnis über die interne Struktur des Kerns haben, um z. B. den Zustand einer Task im Klartext („Ready“, „Suspended“, ...) anzuzeigen. Zusätzlich muss er berücksichtigen, welche Struktur der Kern in Abhängigkeit von der Applikation gerade hat. Diese Informationen hängen zusätzlich auch von dem Hersteller des BS und von der verwendeten µC-Architektur ab. Ein Debugger, mit dem Applikationen für OSEK-BS unterschiedlicher Hersteller und für unterschiedliche µCs getestet werden soll, müsste für jede denkbare Konstellation von µC, OSEK-Hersteller und Kern-Konfiguration die notwendige Information besitzen, was praktisch nicht möglich erscheint. Um dem Gedanken der offenen Systeme beim Testen von

OSEK-Applikationen Rechnung zu tragen, bedarf es einer standardisierten Debug-Schnittstelle zwischen OSEK und Debugger.

Ein Verfahren zum Test von OSEK-Applikationen basiert auf einer Datei, der so genannten „OSEK-Runtime-Interface“-Datei (ORTI-Datei), die die OSEK-spezifischen Debug-Informationen zu einer OSEK-Applikation enthält. Das OSEK Run Time Interface (ORTI) ist eine standardisierte Schnittstelle, besonders zur Fehlersuche während der Laufzeit [OSEK03]. Wenn die Werkzeugkette bekannt und eine proprietäre Lösung verfügbar ist, ist ORTI evtl. nicht notwendig. Das ORTI muss von beiden Seiten unterstützt werden. Eine ORTI-Datei wird zur Generierungszeit der OSEK-Applikation durch die Werkzeuge des OSEK-Systems (i. A. durch den System Generator des OSEK OS) erzeugt, die auch die Konfiguration des OSEK-Kerns vornehmen [Büch00], und anschließend vom Debugger geladen. Wenn die Implementierung von OSEK ORTI unterstützt wird, kann jeder Debugger, der das ORTI berücksichtigt, verwendet werden. Genau genommen kann jeder Debugger verwendet werden, aber wenn er das ORTI nicht berücksichtigt, ist die Fähigkeit, den momentanen Status des RTOS auf hoher Ebene anzuzeigen, nicht gegeben. Informationen über die vorliegende OSEK-Applikation wären z. B. Anzahl, Name und ID-Nummer der Tasks dieser Applikation. Die ORTI-Datei enthält z. B. die Adresse im Speicher des Zielsystems, in der zu jeder Zeit die Identifikation der Task steht, die gerade im Zustand „Running“ ist.

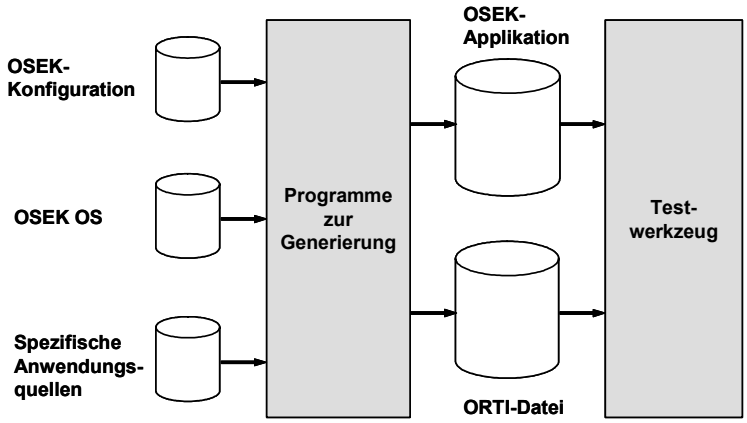


Bild 134: Test von OSEK-Applikationen mit Hilfe des ORTI-Standards

Die ORTI-Datei wird von den Testwerkzeugen[64] verwendet, um spezielle Test- und Analysefunktionen für die betreffende OSEK-Applikation zur Verfügung zu stellen (Bild 134 [Büch01]) und beschreibt, auf welche Weise der Debugger dynamisch Informationen über den aktuellen Systemzustand ermitteln kann. Eine OSEK-Applikation, deren BS sich an diese Schnittstelle hält, kann mit jedem Debugger getestet werden, der sich ebenfalls an diese Schnittstelle hält.

Hinsichtlich IDE-basierter Projektverwaltung sollte die Gruppierung von Make-, C-, Header- und OIL-Dateien möglich sein und Projektbereiche zur Verwaltung ganzer Projekte existieren. Mehrere Dateien sollten zur selben Zeit geöffnet sein können, und Drag-und-Drop sollte unterstützt werden. Eine automatische Generierung der Make-Dateien sollte möglich sein. Die Installation sollte unkompliziert und deren Benutzung ohne Administratorrechte möglich sein. Der BS-Hersteller sollte ein Board Support Package (BSP) für die eingesetzte Evaluierungsplatine (Evaluation Board, EVB) zur Verfügung stellen.

[64] Allgemeine Bezeichnung für ein Werkzeug zum Test von Applikationen. Kann z.B. ein In-circuit-Emulator (ICE), ein BDM- oder JTAG-Debugger, eine Monitorlösung oder ein Simulator sein. Ein ICE ist ein Testwerkzeug, das den µC des Zielsystems ersetzt. Signifikante Bestandteile eines ICE sind Trace, Trigger und Emulationsspeicher. Hauptmerkmale eines ICE sind Echtzeitfähigkeit und dass er keine Ressourcen (etwa Speicher) des Zielsystems benötigt. Die aufgezeichneten Zyklen werden in einem Speicher (Trace-Speicher) des ICE abgelegt.

5.1.2 Setup der Messung

Für eine realistische Einschätzung der gemessenen Werte werden die Messungen wiederholt. Start- und Make-Dateien werden für die Untersuchungen modifiziert, so dass dieselbe IDE (Wind River Tornado, Single Step BDM Debugger, Diab Compiler v4.4b/5) verwendet werden kann. Alle Untersuchungen konzentrieren sich auf den Target-µC Motorola PowerPC 555 (MPC555). Die OIL-Dateien des untersuchten BS sind untereinander inkompatibel. Die folgende OIL-Konfiguration wird benutzt: `Status = Standard`, `Hooks = False`, `Systemstacksize = 512`, `Activation = 1`, `Stacksize = 128`. Die Tasks werden in die Ready-Warteschlange (ready queue) eingestellt, indem die `Autostart`-Option in der OIL-Datei auf `true` gesetzt wird. Dieselbe ROM/RAM-Partitionierung wird benutzt und auch ohne Code zur Zeitmessung verifiziert. Alle Tests werden mit angeschlossenem Debugger durchgeführt (das Target läuft langsamer, wenn der Debugger angeschlossen ist). Hinsichtlich der Zeitbasis wird in allen Konfigurationen die Zeit 0,803 µs pro Zähleinheit (Tick) angenommen, da 315197637 Einheiten, die mit `delay(30000)` generiert werden, 253 s benötigen. Je höher die Verzögerung (delay) ist, desto besser ist die Konvergenz zu einer unteren Grenze von etwa 0,803 µs. Diese Werte sind relativ und hängen von der individuellen Initialisierung der Zeitbasis und dem Einfluss des BS ab. Ohne ein BS variieren die Werte, genauso wie dies beim Einsatz eines anderen BS sein kann. Die `delay()`-Funktion hat den Effekt, dass weiterer Fortschritt im C-Code für eine bestimmte Zeit verhindert wird. Durch Messen der Verzögerungszeit sowie das Auslesen der Anzahl der Zähleinheiten der Zeitbasis, die für diese Verzögerungszeit benötigt werden, kann die Zeit, die für jede Zähleinheit benötigt wird, berechnet werden. Alle durchgeführten Zeitmessungen basieren auf Zähleinheiten. Deshalb ist die verwendete Relation: `[Wert in µs] = 0,803·[Wert in Ticks]`, wobei die Taktrate des µC 40 MHz ist.

Die Messungen werden mit den Original-Make-Dateien durchgeführt, so wie sie vom Hersteller zur Verfügung gestellt werden. Im Fall von RTOS 1, RTOS 3 und RTOS 5, sind Compileroptimierungen (Parameter `-XO`) aktiviert, da sie ursprünglich in den Make-Dateien der entsprechenden Hersteller aktiviert sind. Das resultiert in einem Performance-Gewinn von etwa 22 %. Um mit minimalem Overhead zu messen, kommt auch Assembler-Code für die Zeitmessungen zum Einsatz, und Prozessorregister werden direkt beschrieben. Diese Umgehung des BS kann dazu führen, dass einige Messungen wiederholt werden müssen. Manuelle Messungen werden mit Hilfe eines prellfreien Schalters durchgeführt, so dass doppelte Auslöser vermieden werden.

5.1.3 Performance und Belastungstests

Bei Analyse der Ausführungsgeschwindigkeit muss zunächst zwischen zwei Systemtypen unterschieden werden. Auf der einen Seite stehen Systeme, deren Performance durch entwurfsspezifische Aspekte wesentlich beeinflusst wird. So findet bei einem Modell mit OSEK und Verwendung von OSEK-Alarms direkter Eingriff in das Zeitverhalten statt, so dass die Performance nicht mehr durch die HW, sondern durch den Entwurf beeinflusst wird. Ähnlich ist der Fall bei Systemen, bei denen eine Triggerung von außen notwendig ist, um die Ausführung zu starten. Hier kann während der Ausführung zwar eine maximale Prozessorauslastung erreicht werden; wird aber das System nicht ständig getriggert, kommt es zu häufigem Leerlauf, und ein Betrieb mit durchgängig voller Auslastung ist nicht mehr gegeben. Bei Verwendung von OSEK-Alarms muss darauf geachtet werden, dass nicht durch eine überhöhte Taktung die Ausführung der Tasks vorzeitig abgebrochen wird.

Gegenstand der Betrachtung ist die Messung von Durchsatz, Reaktionsfähigkeit und Determinismus. Eine absolute Zeitreferenz wird benötigt, um Zeitintervalle zu messen. Messungen können durchgeführt werden, indem viele Ereignisse gemessen werden, d. h. während eines längeren Zeitraums, so dass eine ausreichende Genauigkeit erreicht wird. Im Rahmen der Evaluierung werden typischerweise spezielle Echtzeit-Eigenschaften des RTOS gemessen [DeSE01]. Performance-Tests werden ausgeführt, um die Performance zu untersuchen. Belastungstests sind relevant, um das Systemverhalten unter hoher Last, und damit die Effizienz eines Teilausfalls, zu untersuchen.

5.1.3.1 Latenzzeit beim Umschalten der Tasks

Die Latenzzeit beim Umschalten der Tasks (thread switch latency) ist das Zeitintervall zwischen der letzten Instruktion des momentan laufenden Tasks vor der Rückgabe des Prozessors, und der ersten Instruktion der nächsten Task im Zustand `Ready`; d. h. die Zeit, die das BS benötigt, um die aktuelle Task zu unterbrechen (preempt), die Task im Zustand Ready, die als nächste laufen soll, zu reaktivieren und ihre Ausführung zu starten. Zweck ist die Verifikation, ob die Zeit, die das BS zur Ausführung einer Task-Umschaltung braucht, von der Anzahl der Tasks in der Ready-Warteschlange abhängig ist. Ein Variieren der Task-Anzahl und Prioritätsebenen zeigt, ob die Verwaltung der Systemwarteschlangen effizient ist.

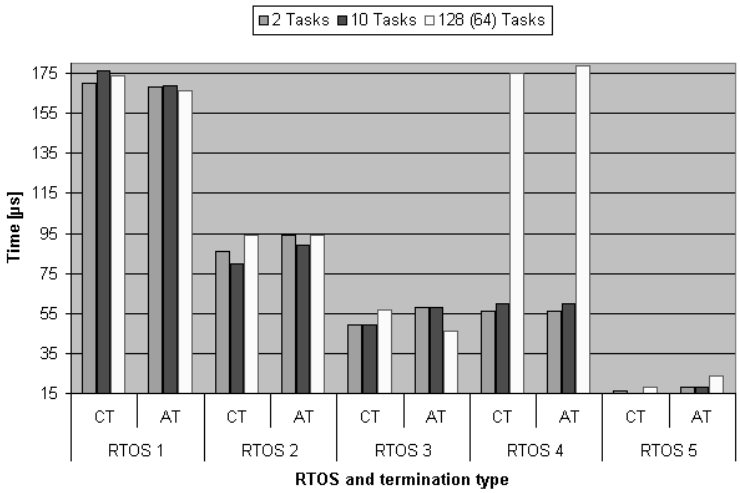


Bild 135: Vergleich mit unterschiedlicher Anzahl von Tasks, BCC1

Das Beenden einer Task-Funktion ohne den Aufruf von `TerminateTask()` oder `ChainTask()` ist nicht erlaubt und kann das System in einem undefinierten Zustand zurücklassen. Die Kombination von `ActivateTask()` (AT) und `TerminateTask()` (TT, in dieser Reihenfolge) korrespondiert mit `ChainTask()` (CT). Um vergleichbare Werte zu erhalten, werden die Messungen deshalb vor `ActivateTask()` durchgeführt. In Bild 135 ist die Conformance Class konstant (BCC1), in Bild 136 werden BCCx Conformance Classes in Verbindung mit dem CT- und AT-Terminierungstyp verglichen, und in Bild 137 ist die Task-Anzahl konstant (10 Tasks).

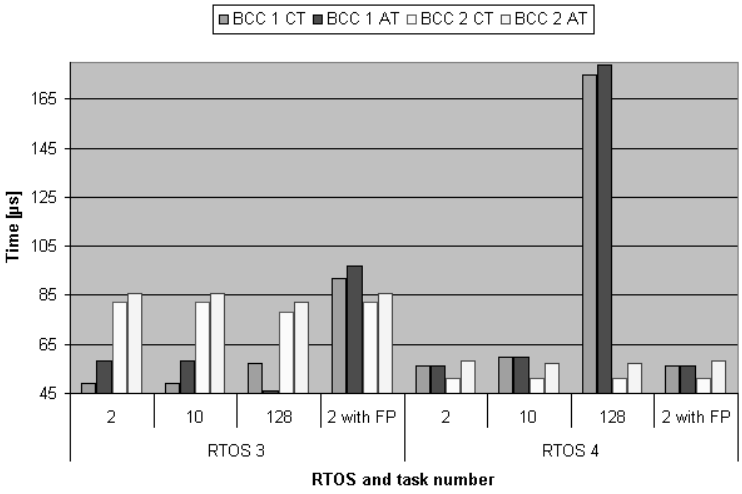


Bild 136: Vergleich mit Basic Conformance Classes und FP

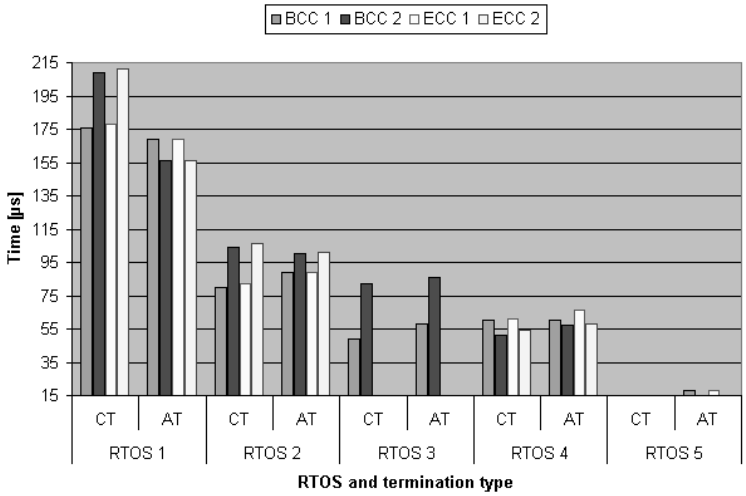


Bild 137: Vergleich mit allen Conformance Classes (10 Tasks)

Wenn nur eine Task in der Ready-Warteschlange ist, erhöhen sich die Werte um 5-25 %. Mit RTOS 1 ist die Umschaltung zwischen Tasks 22 % schneller, wenn sie dieselbe Priorität besitzen und in der Ready-Warteschlange sind, und das Umschalten von Tasks mit `Activate/TerminateTask()` ist 20 % langsamer, wenn sie unterschiedliche Prioritäten besitzen und in der Ready-Warteschlange sind – verglichen mit dem Fall, wenn sie nicht in der Ready-Warteschlange sind. Die Verwendung mit FP-Variablen oder Wind Rivers Real Time Analysis Suite (RTA) erzeugt keinen Overhead der Latenzzeit[65] beim Umschalten von Tasks im Gegensatz zu WindView, bei dem der Overhead ca. 30 % beträgt. Bei RTOS 2 ist die Task-Umschaltzeit mit FP etwa 11 % schneller als ohne FP, und bei RTOS 3 ist bei Verwendung von BCC1 die Task-Umschaltung 70-90 % langsamer mit FP als ohne.

5.1.3.2 Unterbrechungs-Latenzzeiten

Interrupt Latency ist das Zeitintervall zwischen der letzten Instruktion, die in der unterbrochenen Task ausgeführt wurde, und der ersten Instruktion, die im Unterbrechungs-Handler ausgeführt wurde, d. h. die Zeit, die das BS benötigt, um die laufende Task zu unterbrechen und die Ausführung des Unterbrechungs-Handlers zu starten. Zweck ist, festzustellen wie lange das System benötigt, um auf externe Ereignisse zu antworten. Die Interrupt Latency wird gemessen, wenn eine einzelne Unterbrechung auftritt, während eine Task einen Benutzeraufruf ausführt. Unterbrechungen werden periodisch generiert. Interrupt Dispatch Latency ist das Zeitintervall zwischen der letzten im Unterbrechungs-Handler ausgeführten Zeile und der ersten im geplanten Task ausgeführten Zeile, d. h. die Zeit, die das BS benötigt, um vom Unterbrechungs-Kontext zum Kontext der unterbrochenen Task oder der nächsten Task aus der Ready-Warteschlange, umzuschalten (Bild 138).

Das System wird gleichzeitig mit zwei Unterbrechungs-Anforderungen (Interrupt Requests, IRQs) beaufschlagt. Die Zeit, die benötigt wird, um beide Unterbrechungen zu bedienen, wird gemessen. Die Behandlung von Unterbrechungen ist immer noch verbreitet und effizienter als eine zyklische Abfrage, wenn die benötigte Antwortzeit auf das externe Ereignis beträchtlich kleiner als der minimale Zeitraum des Einfallsabstands (minimal interarrival period) ist [Zahi99].

Erläuterungen zum Unterbrechungs-System des MPC555 sind in [DuFM01] zu finden. Die schnellste Unterbrechungsbearbeitungsroutine (Interrupt Service Routine, ISR) ist eine Kategorie-1-Unterbrechung, die keine BS-API-Aufrufe benötigt. Da diese ISR nicht mit dem BS interagieren, besitzen sie den kleinsten Overhead von allen ISR-Kategorien. ISR-Kategorie-2-Unterbrechungen werden API-Dienste genannt. Diese Unterbrechungen benötigen typischerweise einen Zähler, der inkrementiert

[65] Die Latenzzeit einer Nachricht ist als das Intervall definiert, das zwischen dem Zeitpunkt der Erzeugung des Datenelementes beim Sender und dem frühesten Zeitpunkt des vollständigen Empfanges liegt.

wird, eine Task, die aktiviert wird, ein Ereignis, das gesetzt wird, oder eine Nachricht, die gesendet wird [Lemi01]. Da das BS gemessen werden soll, wird die Unterbrechungs-Kategorie 2 ausgewählt.

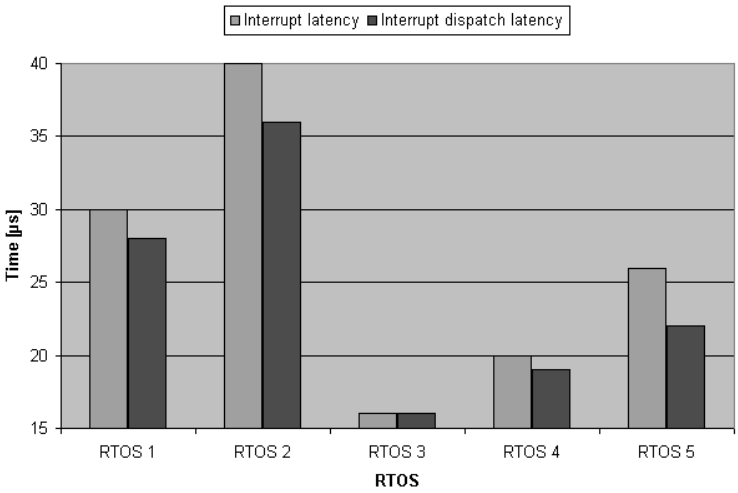


Bild 138: Interrupt (Dispatch) Latency

Der Wert hängt von der Position in der `While`-Schleife ab, wenn die Unterbrechung auftritt. Deshalb wird der kleinste positive Wert als das Ergebnis ausgewählt. Die Messung ist nur dann korrekt, wenn die Ausführung der Applikation bei Punkt 1 unterbrochen wird. Wenn sie bei Punkt 2 unterbrochen wird, ist der Wert von `c` negativ (Bild 139).

```
        ...
        1.     while(1){
        2.     b = readTBL();
               c = b - a;
        }
        TerminateTask();
}

ISR(isr1){
        USIU.SIPEND.B.IRQ5 = 1; //The bit
in the SIPEND must be cleared when a
falling edge interrupt occurs.
        b = 0;
        resetTB();
        a = readTBL();
}
```

Bild 139: Messzeitpunkt

5.1.3.3 Zeit zum Sperren von Ressourcen

Sperren von Ressourcen werden verwendet, wenn ein kritischer Abschnitt des Codes nicht von einer anderen Task unterbrochen werden darf. Verschachtelter Zugriff auf eine Ressource ist verboten. Weiterhin kann die Applikation nicht `TerminateTask()`, `ChainTask()`, oder `WaitEvent()` aufrufen, während eine Ressource gesperrt ist. Kritische Abschnitte in der Applikation sollten so knapp wie möglich gehalten werden, damit die Latenzzeit von Tasks, die eine Priorität zwischen der Priorität der Task, die die Ressource sperrt, und der höchsten Priorität der Ressource haben, begrenzt ist [Lemi01]. Die Zeit, eine OSEK-Ressource zu sperren (`get`) und wieder freizugeben (`release`) wird gemessen (Bild 140).

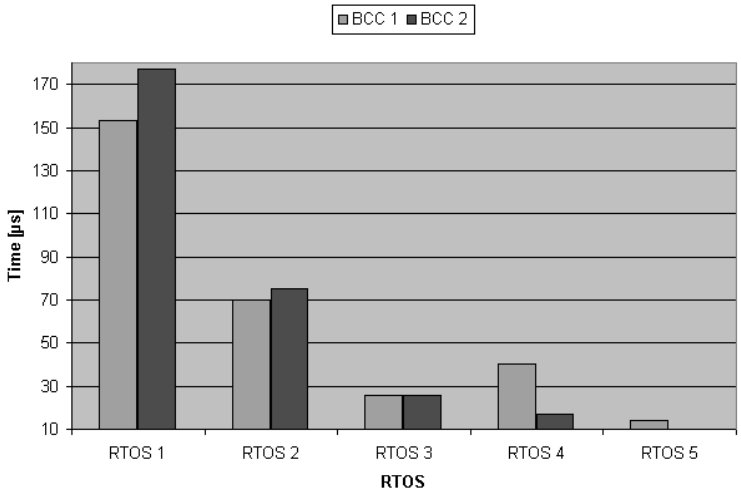


Bild 140: Zeit, um eine Ressource zu sperren (freizugeben)

5.1.3.4 Bearbeitungszeit für gleichzeitige und verschachtelte Interrupts

Es werden zwei annähernd simultane Unterbrechungen erzeugt. Zuerst wird die niederpriore Unterbrechung generiert, unmittelbar gefolgt von der höherprioren. Das Zeitintervall zwischen beiden aufeinander folgenden Unterbrechungen wird variiert, muss aber kleiner sein als die Zeit, die benötigt wird, um die Bearbeitung der ersten Unterbrechung abzuschließen. Zweck ist, herauszufinden wie lange das System benötigt, um auf zwei gleichzeitig aufgetretene Unterbrechungen zu antworten, ob die Unterbrechungsbehandlung priorisiert ist oder nicht und ob Unterbrechungen verschachtelt sein können (Bild 141).

Simultane Unterbrechungen (BCC1): Während einer spezifizierten Zeit (`delay(200)`) werden zwei Unterbrechungen manuell und simultan ausgelöst. Die Zeit zur Ausführung von `delay(200)` beträgt 2101330 Zähleinheiten. Die resultierende Zeit für `delay()` und die Unterbrechungen `INT 5` und `INT 7` beträgt 2101419 Zähleinheiten. Die Gesamtzeit für beide Unterbrechungen beträgt deshalb 89 Zähleinheiten.

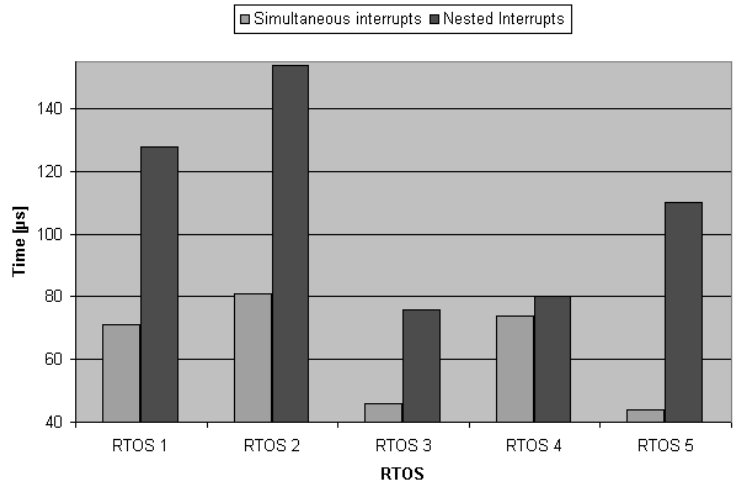


Bild 141: Simultane (verschachtelte) Unterbrechungen

Die ISR 5 (`Int_Hdl5()`) und 7 (`Int_Hdl7()`) beinhalten Code, um die Unterbrechungen 5 bzw. 7 zurückzusetzen (`Reset`), sowie eine Zählvariable, um sicherzustellen, dass die Routine nur einmal aufgerufen wird (`x++; SIU.SIPEND.B. IRQ5=1;`) (Bild 142).

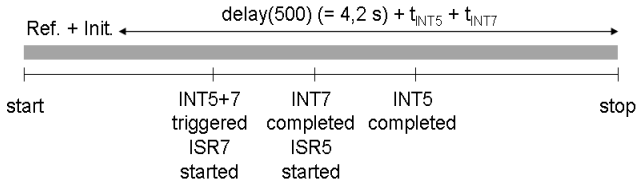


Bild 142: Messung simultaner Interrupts

Verschachtelte Unterbrechungen (BCC2): Die bekannte Zeit für `delay(500)` beträgt 5253306 Zähleinheiten, die Zeit für zwei `delay(500)`, einschließlich INT 7 und INT 5 beträgt 10506772 Zähleinheiten. Deshalb ergibt sich die Zeit für beide Unterbrechungen zu 160 Zähleinheiten. Jede Unterbrechung muss manuell innerhalb eines Zeitraums von ca. 3 s ausgelöst werden (Bild 143).

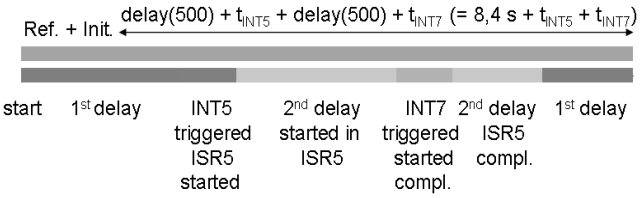


Bild 143: Messung verschachtelter Unterbrechungen

5.1.3.5 Größte dauerhafte Unterbrechungs-Frequenz

Die größte dauerhafte Unterbrechungs-Frequenz (maximum sustainable interrupt frequency) (Bild 144) legt die maximale periodische Unterbrechungs-Frequenz fest, die das System handhaben kann, ohne einen einzigen IRQ zu verlieren, d. h. jeder einzelne IRQ muss bedient werden.

Ein Frequenzzähler (im Zählermodus) ist mit dem Funktionsgenerator und mit dem IRQ-Anschluss des EVB verbunden. Wenn der Mikrocontroller die Gesamtanzahl an Unterbrechungen verarbeiten kann, ist der Zählerwert gleich dem Wert der Variablen, die bei der Flanke inkrementiert wird. „Lightweight“ und „Heavyweight“ Termination (RTOS 3) der Tasks ergibt im Ergebnis keinen Unterschied.

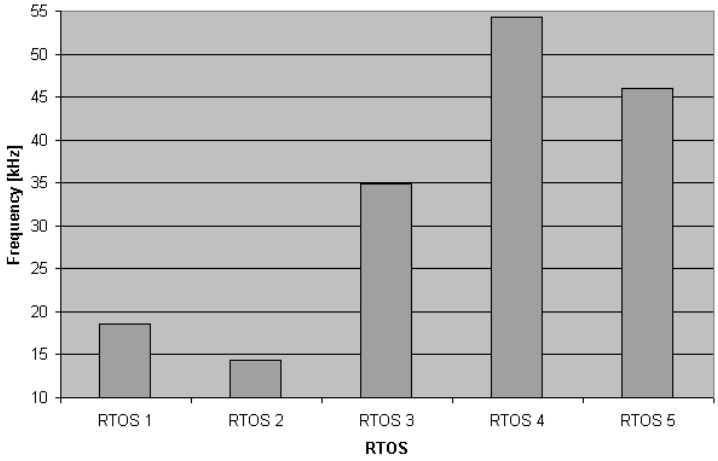


Bild 144: Maximale stabile Unterbrechungs-Frequenz

5.1.4 Speicherbedarf

Die Informationen aus der Speichertabellen-Datei (Map-Datei) werden verwendet, um den Speicherverbrauch zu berechnen. Der Bedarf an ROM-Code wird aus der Größe der Abschnitte `.text`, `.init`, `.fini` und `.eini` berechnet, die ROM-Konstanten von `.sdata2` und `.rodata`, der RAM-BSS-Verbrauch (nicht initialisierte Daten) aus `.sbss` und `.bss`, und der RAM-DATA-Bedarf (initialisierte Daten) aus `.data` und `.sdata`. Die individuellen Speicherbereiche setzen sich

aus Code, Konstanten, Stack und Variablen zusammen. Die Messungen werden durchgeführt mit `Activate-/TerminateTask()` und ohne den Overhead, der sich durch die Zeitmessungen ergeben hat (die Dateien `timer.c` und `timer.h`). Um die Messergebnisse der Messungen mit einer Task miteinander vergleichen zu können, werden alle Messungen mit einem Zeitgeber (Timer) durchgeführt, der in der Applikation eingebunden ist. Debug-Daten werden bei den Berechnungen ignoriert. Z. B. sind in Bild 145 die Speicheranforderungen für eine Task jedes RTOS dargestellt.

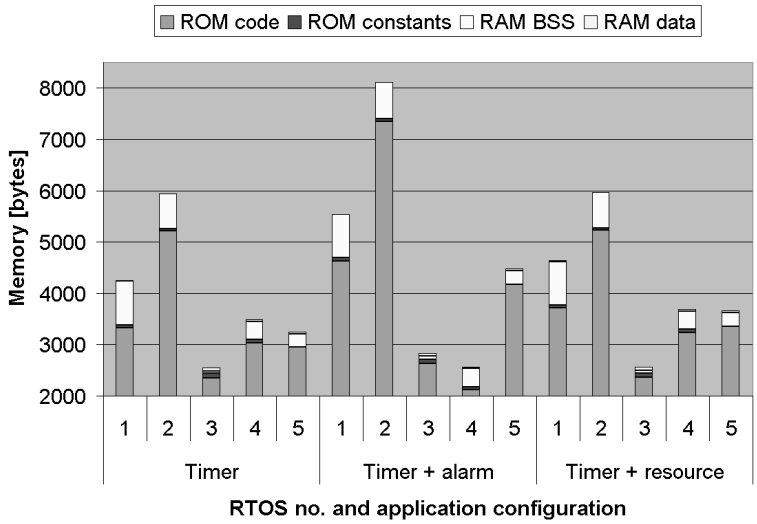


Bild 145: Speicheranforderungen für eine Task

Die Applikation besteht aus einem Timer, einem Timer und einem OSEK-Alarm sowie einem Timer und einer OSEK-Ressource. Bild 146 gibt die Speicheranforderungen an, wenn mehrere Tasks verwendet werden, wobei für beide Abbildungen die Conformance Class BCC1 zu Grunde gelegt wird.

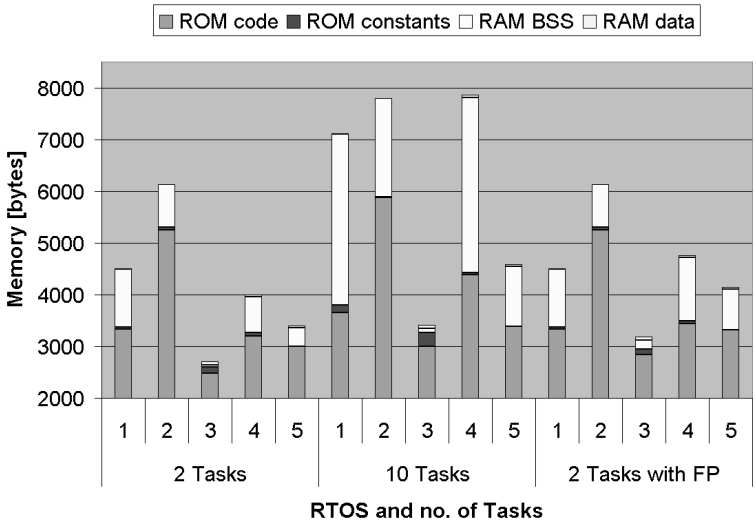


Bild 146: Speicheranforderungen für mehrere Tasks

Die Speicheranforderungen für RTOS 2 sind in Bild 147 dargestellt, wobei der wachsende Speicherbedarf für eine wachsende Anzahl an Tasks für alle vier OSEK-OS-Conformance Classes dargestellt wird.

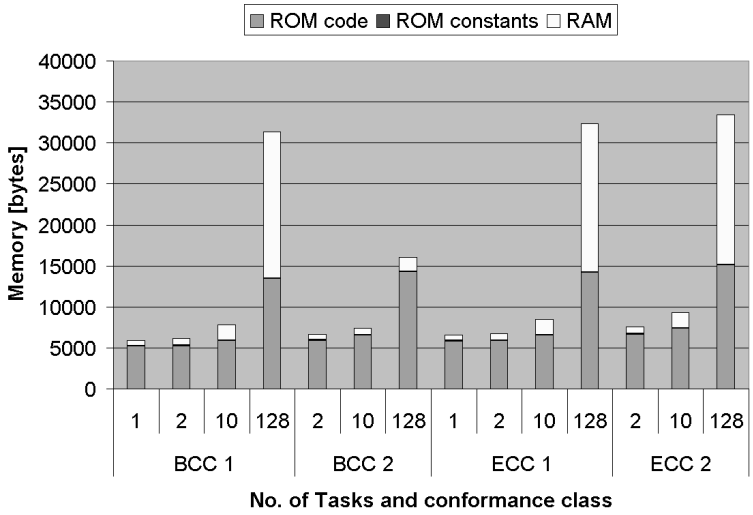


Bild 147: Speicheranforderungen für RTOS 2

Die Messergebnisse von RTOS 1 zeigen, dass die Task-Umschaltzeit von Anwendungen, die auf den Conformance Classes xCC2 basieren, 19-42 % größer ist, als die von Applikationen, die auf xCC1 (BCC1 und ECC1) basieren. Bei RTOS 3 beträgt dieser Wert 50-70 %. Bei RTOS 1 ist `ChainTask()` 5 % langsamer als die Kombination von `Activate-` und `TerminateTask()`, und die Zeit, um eine Ressource zu sperren und freizugeben ist 16 % größer, wenn eine xCC2 Class eingesetzt wird. RTOS 2 benötigt mehr Zeit zum Sperren und Freigeben von Ressourcen bei der BCC2-Klasse. Hinsichtlich Speicheranforderungen benötigt RTOS 1 mindestens 3,3 KB ROM für den Code und die Konstanten. Eine xCC2 Class benötigt 14 % mehr ROM als eine xCC1 Class, wobei sich dieser Wert auf 5 % reduziert, wenn viele Tasks verwendet werden. Bei RTOS 2 ist der Mindestbedarf an ROM 5,2 KB, aber es benötigt weniger RAM als RTOS 1. Die Summe aus RAM und ROM ist kleiner bei RTOS 1, wenn wenige Tasks verwendet werden, aber größer bei mehreren Tasks. RTOS 3 benötigt 2,5 KB Speicher für eine Task.

Tornado vermittelt den Eindruck einer weitgehend stabilen Entwicklungsplattform. Jedoch sind noch einige kleinere Fehler vorhanden, und einige Arbeitsflüsse, wie die Generierung von Make-Dateien, können automatisiert werden. Wind Rivers RTA und WindView sind nützliche Werkzeuge zur Ergänzung der IDE. RTA beinhaltet z. B. einen Profiler, und WindView zeigt die Prozesse auf OSEK-Ebene graphisch an. RTA erzeugt einen geringen Speicher-Overhead (1 % Code, wenige Konstanten und einen geringen Stackverbrauch). Wenn jedoch die Anwendung für die Unterstützung von WindView compiliert und gelinkt wird, steigt der Code-Overhead auf 200 %. Hinsichtlich der Task-Umschaltzeit erzeugt WindView 30 % Overhead.

Die evaluierten Komponenten haben ihre Vor- und Nachteile. Die Ergebnisse zeigen, dass es offensichtliche Unterschiede hinsichtlich Performance und Speicherverbrauch gibt, die nicht nur von der Marke abhängen, sondern auch von den individuellen Parametern und der Anwendungs-Konfiguration. Einige RTOS werden mit einer IDE geliefert, andere nicht. Einige sind mehrere Jahre auf dem Markt etabliert, umfangreich getestet und mit verschiedenen Werkzeugen ausgestattet, während andere innovative oder sehr nützliche Zusätze besitzen, oder ihre Hersteller unterbreiten attraktive finanzielle Angebote. Welche Eigenschaften notwendig sind, welche nützlich und welche nicht benötigt werden, hängt von der jeweiligen Applikation ab.

5.2 Code-Analyse kommerzieller Produktionscode-Generatoren

5.2.1 Kriterien

Über 80 % aller ECUs befinden sich in Systemen, die sicherheitsrelevante Aufgaben wahrnehmen. Der Beseitigungsaufwand steigt exponentiell an, je später der Fehler entdeckt wird. Typischerweise

kostet die Identifizierung und Behebung eines Fehlers, der erst in der Produktionsphase sichtbar wird, 16-mal mehr, als wenn der Fehler bereits während der Codierung entdeckt wird (Bild 3). Qualitätssicherungssysteme greifen daher schon vor der Codierung ein und unterstützen bei der Auswahl und Anpassung von Regeln für die Programmierung, die in einem konkreten Projekt einzuhalten sind. Ein konsistentes, zielorientiertes Regelwerk deckt die folgenden Ziele ab: Leichte Lesbarkeit des Codes (z. B. „Jede global definierte Variable soll mit einem Großbuchstaben beginnen“); bessere Verständlichkeit (z. B. „Die `THEN`- und `ELSE`-Blöcke eines `IF`-Statements dürfen nicht leer sein“); Erleichterung von Wartung und Weiterentwicklung (z. B. „Deklaration von Variablen auf Blockebene ist nicht zulässig“); Minimierung möglicher Fehlerquellen (z. B. „Der Zähler eines `FOR`-Statements darf innerhalb der Schleife nicht verändert werden“, oder „Die Verwendung von `GOTO`-Statements ist verboten“); Kompatibilität und Erleichterung der Portabilität (z. B. „Es dürfen keine Sonderzeichen in Namen verwendet werden“); Generelle Sinnhaftigkeit von Deklarationen (z. B. „Jede deklarierte Variable oder Funktion muss benutzt werden“). Nach [Somm87] hat ein gutes Programm ein „sauberes“ Layout, verwendet sinnvolle Namen, ist ausführlich kommentiert und verwendet Konstrukte der Sprache derart, dass maximale Robustheit und Lesbarkeit des Programms erreicht werden. Sauberes Layout heißt z. B. Verfügung über Tabulatoren, Leerzeilen und keine überlangen Zeilen. Mit sinnvollen Namen ist gemeint, dass die Bezeichner auf Semantik hinweisen. Die Code-Merkmale sind nach [Hitz99] und [Soly00] ein Maß für das Code-Layout und wie die Modellinformationen im Code bzgl. der Attribute, der Methoden und dem Modellverhalten umgesetzt werden. Die Bewertung des Code-Layouts gibt einen Überblick, wie sich der generierte Code lesen und nachvollziehen lässt. Anhand dieser Bewertung kann der Entwickler erkennen, wann und an welcher Stelle der generierte Code nachbearbeitet werden sollte. Diese Nacharbeit versichert dann, dass die Modellinformationen im Code genau umgesetzt werden.

Eine häufig anzutreffende Klasse von Fehlern ist durch die Komplexität der SW bedingt. Metriken sind ein geeignetes Hilfsmittel, besonders komplexe und damit fehleranfällige Module zu identifizieren. Der Einsatz professioneller Qualitätssicherungsmaßnahmen für Automotive-SW-Systeme verkürzt die Projektlaufzeit, weil die Maßnahmen einen schwer beherrschbaren Aufgabenbereich greifbarer und besser planbar machen. Vorteile sind: Markteinführungszeit für neue Fahrzeuggenerationen oder -varianten wird verkürzt, da die SW schneller den Zustand der Serienreife erreicht; Unterstützung zum Erlangen von Zertifizierungen nach den Standards ISO/IEC 9126, 9001 und DO-178B; Erfüllung der M.I.S.R.A.-Richtlinien[66]; geringere Entwicklungskosten durch kürzere Entwicklungszeiten; Qualität der SW wird messbar und die Messergebnisse sind reproduzierbar und vergleichbar. Die wichtigsten mittelbaren Vorteile sind die Einsparungen durch weniger Reklamationen und Reduzierung des Risikos von Rückrufaktionen [Stüc01].

Weiterhin ist relevant, ob der automatisch erzeugte Code nur für bestimmte BS und Plattformen verwendbar ist, oder ob es sich um Code nach ANSI handelt. Oft sind die Generatoren daher parametrisierbar. Weiterhin stellt sich die Frage nach der Performance. Für die vorliegende Arbeit ist C als Zielsprache besonders relevant, da die evaluierten zum Einsatz kommenden OSEK-Derivate z. T. ausschließlich die Sprache C unterstützen.

5.2.2 Code-Generatoren

5.2.2.1 TargetLink (dSPACE)

TL wurde im Rahmen dieser Arbeit dazu verwendet, aus Simulink- und Stateflow-Modellen C-Code zu generieren. TL ist ein SW-System, das sich in die MATLAB/Simulink/Stateflow-Entwicklungs-

[66] Die Motor Industry SW Reliability Association (M.I.S.R.A.) wurde im April 1998 ins Leben gerufen. Ziel der Vereinigung war die Festlegung von Standards in der SW-Entwicklung für Automotive Applikationen, und somit das Erreichen einer gleichbleibend hohen Qualität der SW. Die Programmiersprache C wird in zunehmendem Maße für Automotive Anwendungen verwendet. Die M.I.S.R.A. Richtlinien dienen der Entwicklung und der Anwendung von sicherer und zuverlässiger SW innerhalb der Automobilbranche [AuEl01].

umgebung integriert. Aus Simulink-Blockschaltbildern oder Stateflow-Zustandsdiagrammen kann mit TL C-Code nach dem ISO/IEC 9899 Standard erzeugt werden. Das TL-Blockset `tllib` enthält für jeden unterstützten Simulink-Block einen TL-Block, der die Funktionalität des jeweiligen Simulink-Blocks für die Code-Erzeugung erweitert. Eine der Stärken von TL ist die Möglichkeit einer hardwarenahen Code-Generierung, wobei C-Code gezielt für bestimmte Compiler/Prozessor-Kombinationen erzeugt werden kann. Wie SystemGenerator und DSP Builder verfügt auch das TL-Blockset über E/A-Ports zur Spezifikation einer Schnittstelle zwischen Simulink und TL. In TL 2.x sind BS-Objekte wie Alarme und Tasks auf Blockdiagrammebene verfügbar. Für weitere Informationen zu TL 2.x siehe [TL2d02].

5.2.2.2 RTW Embedded Coder

Die von The MathWorks verwendete Werkzeugkette besteht aus den Komponenten MATLAB, Simulink, Fixed Point Blockset (bei Festkomma-Arithmetik), Real-Time Workshop (The MathWorks, RTW), Embedded Coder (The MathWorks, EC). Für die Beschreibung von Zustandsautomaten kommen noch Stateflow und Stateflow Coder hinzu. Der EC ist ein Ergänzungsprodukt zum RTW [RTEC02] und dient zur automatischen Code-Generierung von C-Code für eingebettete Systeme. Ein Embedded Target zeichnet sich dadurch aus, dass es eine Simulink-Bibliothek mit HW-spezifischen Treiberblöcken (A/D-Wandler, pulsweitenmoduliertes Ansteuersignal PWM, CAN, etc.) zur Verfügung stellt [Elia02]. Ein Embedded Target ist jeweils auf einen bestimmten µC-Typ zugeschnitten. Neben der Unterstützung automatischer Code-Generierung und Cross-Entwicklungswerkzeugen bietet es Simulink-Blöcke für Peripherie-HW des Controllers wie z. B. A/D-Wandler und PWM-Modul. Verfügbar sind z. B. Embedded Targets für Infineon C166 und Motorola MPC555 [Math03]. Der EC unterstützt die automatische Code-Generierung aus S-Funktionen; allerdings muss im Gegensatz zu RP für Code-Generierung mit RTW eine korrespondierende TLC-Datei existieren. Das zu übersetzende MATLAB-Modell bzw. Subsystem darf keine kontinuierlichen Zustände enthalten. Beim von EC generierten Modellcode fehlen Kontrollstrukturen, eine Simulationszeitverwaltung und Kommunikationsmodule für den Datenaustausch mit einer Multiprozessor-Umgebung oder den E/A-Modulen. Für diese Aufgaben entwickelte dSPACE das Modul „Real Time Interface“ (RTI). Dieses Modul erzeugt alle für eine Simulation notwendigen Strukturen und führt alle Schritte bis zur Ausführung des Codes auf den einzelnen dSPACE-Komponenten durch (siehe 3.5.2.3).

5.2.2.3 ASCET ECCO und Rhapsody in MicroC

Der Code-Generator ECCO (Embedded Code Creator and Optimizer) erlaubt die Erzeugung von echtzeitfähigem Code zur Ausführung auf mehreren Zielplattformen. Das Target Integration Package (TIP) ermöglicht die Generierung einer lauffähigen Applikation für das Target (Steuergerät) aus ASCET. Das TIP generiert targetspezifischen C-Code für unterschiedliche Zielplattformen, bindet das RTOS ERCOSEK[67] ein, ruft den targetspezifischen Compiler und Linker auf und erzeugt somit eine lauffähige Applikation [TIPE00]. Der ASCET-Code-Generator ist für die Entwicklung sicherheitskritischer SW-Anwendungen nach IEC 61508 zertifiziert [JuGl04].

Eine exemplarische Durchführung von modellbasierter Entwicklung von Kfz-Steuergeräten und automatischer Integration des BS OSEK mit Hilfe des Werkzeugs Rhapsody in MicroC (i-Logix, Inc.) am Beispiel eines Kfz-Rücklichtmoduls wird in [Sche00] dargestellt.

5.2.3 Lesbarkeit des generierten Codes

Die Lesbarkeit des generierten Codes von TL, Embedded Coder, ECCO und Rhapsody in MicroC wurde untersucht, mit folgenden Ergebnissen:

[67] ERCOSEK ist ein OSEK-kompatibles RTOS von ETAS für den Einsatz in eingebetteten Systemen, das zusammen mit ASCET ausgeliefert wird und dort integriert ist. ERCOSEK unterstützt den OSEK-Standard, u. a. statisches und dynamisches Scheduling sowie Multitasking [ERCO00].

- TL verwendet Klein- und Großbuchstaben für Konstanten, Variablen und Strukturen. Kommentare werden in den Dateikopf, in Deklarationen und in die Implementierung (auch von Statecharts) eingefügt. Die Einstellungen des TL Main Dialogs werden in den Code eingefügt. Die von Benutzer hinzugefügten Hinweise werden in den Code eingefügt. Die Hierarchie bezieht sich auf die Programmstruktur. Der generierte Code enthält keine kryptischen Bezeichner. Die Länge eines Bezeichners lässt sich im TL Main Dialog begrenzen [Hora02].
- EC verwendet sowohl Klein- als auch Großbuchstaben für Konstanten, Variablen und Strukturen. Zugehörige Kommentare werden in den Dateikopf, in Deklarationen und in die Implementierung eingefügt. Implementierungen von Statecharts werden nicht kommentiert. Einstellungen des RTW bzgl. des Code-Generators werden nicht im Code dargestellt. Vom Benutzer hinzugefügte Hinweise werden in den Code eingefügt. Die Hierarchie hängt von der Programmstruktur ab. Der generierte Code enthält keine kryptischen Bezeichner, und die Bezeichner können u. U. länger als 20 Zeichen sein.
- ECCO verwendet Klein- und Großbuchstaben für Konstanten und Variablen. Die Namen der Modellblöcke, die die Prozess- und Ereignisnamen beeinflussen, enthalten nur Großbuchstaben. Zugehörige Kommentare werden in Dateikopf, in Deklarationen und in die Implementierung eingefügt. Einstellungen für ASCET bzgl. ECCO werden nicht im Code dargestellt. Die von Benutzer hinzugefügten Hinweise werden in den Code nicht eingefügt. Die Hierarchie hängt von den ineinander geschachtelten Strukturen ab. Der generierte Code enthält nur dann keine kryptischen Bezeichner, wenn der Code innerhalb eines Projekts generiert wird und wenn die Code-Generator-Option „Object Based Controller Implementation“ gewählt ist. Die Länge eines Bezeichners lässt sich bei ASCET nicht begrenzen.

5.2.4 Speicherplatzbedarf für die Zielplattformen

Im ROM-Programm-Code werden feste Datenstrukturen wie konstante Variablen (z. B. `const int x=12`) und konstante Textarrays (z. B. `const int Text[]=„Text“`) gespeichert. Im RAM-Speicher werden dynamische Programmdaten gespeichert, z. B. globale Variablen, `static` oder `extern` lokale Variablen, Zeichenketten (`char Text[] = „Text“`), Zuweisung an Zeichenketten (`#define Text „text“`), Arrays (Daten-RAM), Zeiger und Ereignisse. Im User-Stack werden z. B. Zwischenspeichern-Variablen, Funktionsparameter und Rücksprungsadressen abgelegt.

- Der generierte Code von EC hat im Vergleich mit TL- und ECCO-Code einen großen Speicherbedarf. EC kann Code nur aus diskreten Blöcken generieren. Code-Generierung von kontinuierlichen Blöcken wird nicht unterstützt. EC kann Quellcode anderer Sprachen wie C++, FORTRAN und Ada in Form von S-Funktionen in das Modell integrieren.
- Der generierte Code von TL verfügt über einen geringen Speicherbedarf. Das resultiert aus der eingesetzten Interblock-Optimierung, die die Funktionalität von mehreren Blöcken zu einem optimierten C-Ausdruck in einer Zeile zusammenfasst. TL kann nur C-Code ins Modell einbinden. Quellcode anderer Sprachen wird nicht unterstützt.
- Der generierte Code von ECCO hat nur bei Modellen, die keine Zustandsautomaten enthalten, einen geringen Speicherbedarf. Das liegt an der Zusammenfassung der Funktionalitäten mehrerer Blöcke zu einem optimierten C-Ausdruck in einer Zeile. Der Entwurf von Zustandsautomaten kann mit großem Aufwand verbunden sein, da nicht direkt nebenläufige Zustände modelliert werden können, sondern nur indirekt in Form von nebenläufigen Prozessen. Werden keine nebenläufigen Prozesse verwendet, können Zustandsautomaten in umfangreichem Code resultieren.

5.3 Verifikation der Funktionalität des INTERACT-Frameworks

5.3.1 Verifikation mit Logikschaltungen

Die Korrektheit des Partitionierungsverfahrens kann durch Vergleich der Ergebnisse der Standalone-Simulation und der verteilten Simulation verifiziert werden. Für dasselbe Modell sollen die beiden verschiedenen Arten von Simulationen bei gleichen Eingaben gleiche Ausgaben liefern. Die Schnittstelle zwischen Teilmodellen kann auch durch eine graphische Darstellung von Verbindungen zwischen Teilmodellen in Simulink geprüft werden.

Zur Verifikation der Partitionierung von Logikschaltungen werden zwei Modelle verwendet. Das erste in Bild 148 dargestellte Modell enthält verschiedene Typen von Logikgattern, ohne dass Rückkopplungen zwischen ihnen existieren.

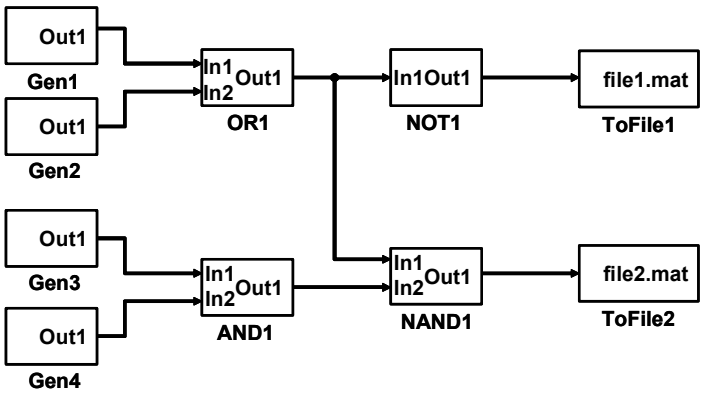


Bild 148: Modell einer Logikschaltung ohne Rückkopplung

In Bild 149 sind die Teilmodelle des Modells von Bild 148 nach Bipartitionierung dargestellt. Zwei Blöcke `client1Out1` und `client2In1` werden jeweils als `Outport` und `Inport` hinzugefügt. Das Teilmodell links enthält auch die Schnittstelleninformation:

```
OUT_INFO:client1Out1:client2/client2In1
```

Der virtuelle Rechnername `client2` wird durch die Konfigurationsdatei in einen realen Rechnernamen umgesetzt. Das rechte Teilmodell wird auf diesem Rechner simuliert.

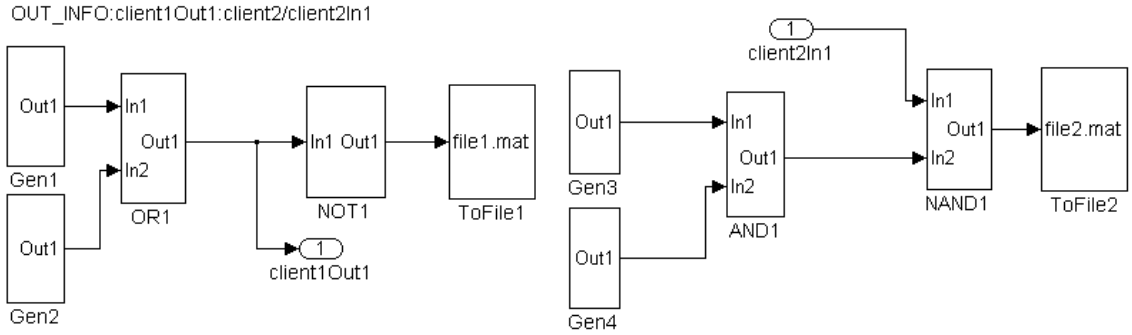


Bild 149: Teilmodelle des Modells von Bild 144

Das zweite Modell enthält eine Rückkopplung zwischen den Logikgattern `NOR1` und `NOR2` (Bild 150). Durch dieses Modell wird die azyklische Partitionierung von Logikschaltungen getestet. Weil die vorhandene Version von JLogSim Logikschaltungen mit Rückkopplung noch nicht simulieren kann, kann das Ergebnis der Partitionierung nur in MATLAB geprüft werden.

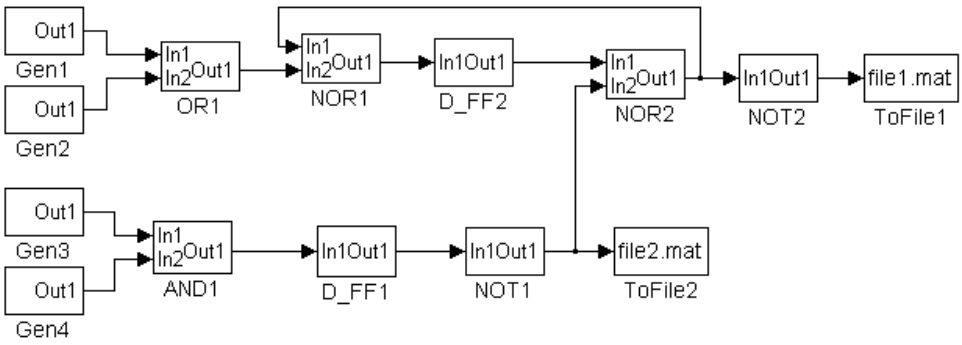


Bild 150: Modell einer Logikschaltung mit Rückkopplung

5.3.2 Verifikation mit zustandsbasierten, kontinuierlichen und hybriden System-Modellen

5.3.2.1 Verifikation von JVHDLGen

Zur Verifikation des Werkzeugs JVHDLGen wird der Simulator ModelSim verwendet. Für dasselbe Statechart müssen die Ergebnisse bei ModelSim mit denen bei Simulink übereinstimmen. Um vergleichbare Ergebnisse eines Modells mit Simulink zu erzielen, muss eine passende Testbench für ein Modell bei der Simulation aufgebaut werden. Die Ausführung eines Charts bei Simulink kann durch ein „Input from Simulink"-Ereignis ausgelöst werden. Der Prozess des generierten VHDL-Codes wird durch ein Taktsignal stimuliert.

Die nebenläufigen Zustände eines Modells mit Statecharts können bezüglich einer Variablen oder eines Ereignisses eine Abhängigkeit haben. Die Abhängigkeiten zwischen nebenläufigen Zuständen können auch einen Zyklus bilden. Jedes Modell wird in drei Partitionen partitioniert und verteilt simuliert. Die Ergebnisse der verteilten Simulation werden mit denen der Standalone-Simulation verglichen.

Tabelle 9 enthält die Modellnamen und Charakteristiken der entwickelten Testmodelle:

Modellname	Abhängigkeit bezüglich	mit/ohne Zyklus
part-test_ta_c	Transitionsaktion und Transitionsbedingung	ohne
part-test_ta_ta	Transitionsaktion und Transitionsaktion	ohne
part-test_evt	Ereignises	ohne
part-test_cycle	Zustandsaktion und Transitionsaktion	mit

Tabelle 9: Testmodelle für die Korrektheit des Partitionierungsverfahrens

In Bild 151 ist das Chart des Testmodells `part-test_ta_ta` abgebildet.

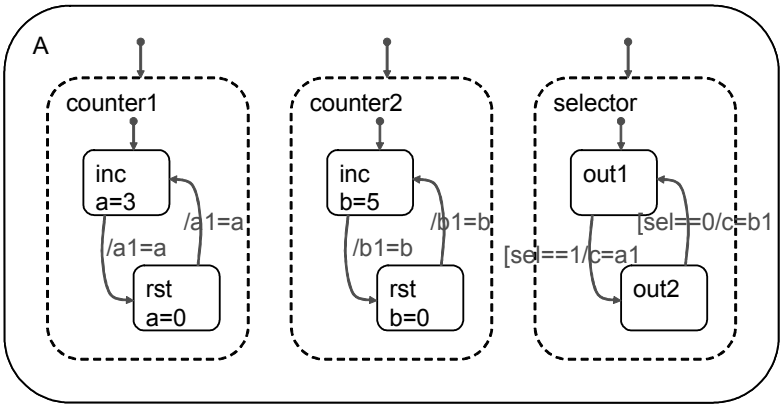


Bild 151: Chart des Modells part-test_ta_ta

Die Transitionsaktionen des Zustands `selector` lesen die Variable `a1` aus dem Zustand `counter1` und die Variable `b1` aus dem Zustand `counter2`. Die Variablen `a1` und `b1` werden als „local“, `c` als „output to Simulink“ und `sel` als „Input from Simulink” deklariert.

Nach der Partitionierung werden die Zustände `counter1`, `counter2` und `selector` einem eigenen Chart `chart0`, `chart1` bzw. `chart2` zugewiesen (Bild 152). Von `chart0` sowie `chart1` gibt es jeweils eine Verbindung zum `chart2` für die Variablen `a1` und `b1`.

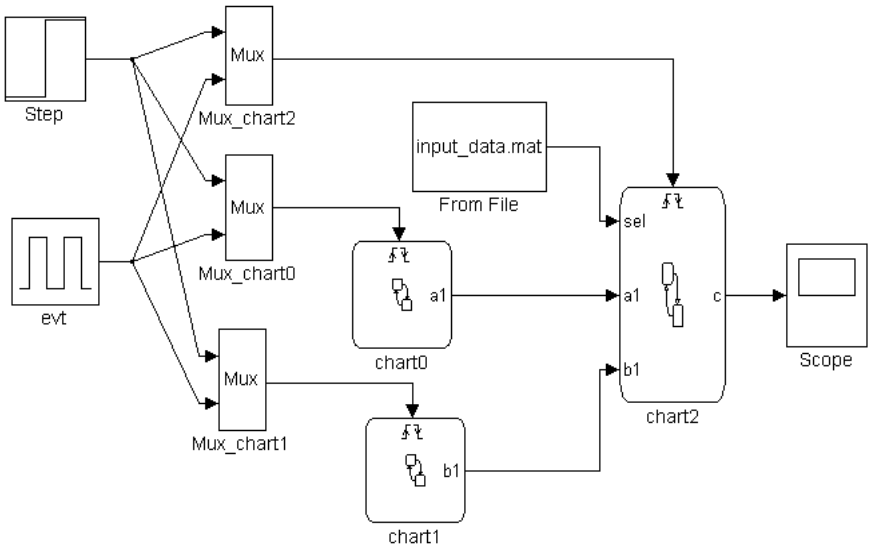


Bild 152: Charts des Modells part-test_ta_ta nach Partitionierung

Die Ereignisse `step` und `evt` im ursprünglichen Chart werden von diesen drei Charts verwendet. Die Variablen `a1` und `b1` werden als „Output to Simulink“ in den Charts `counter_sel0` bzw. `couter_sel1` deklariert. Sie werden gleichzeitig als „Input from Simulink” im Chart `counter_sel2` deklariert. Deshalb existiert jeweils eine Verbindung von dem Chartblock `counter_sel0` und `couter_sel1` zum Chartblock `counter_sel2`. Diese drei Charts können auf drei Rechnern simuliert werden. Aus den Simulationsergebnissen des Charts `chart2` wird der Wert der Variablen `c` zu jedem Zeitschritt mit dem Wert zum gleichen Zeitschritt bei der Standalone-Simulation verglichen.

Das Modell `part-test_cycle` (Bild 153) hat die gleiche Struktur wie `part-test_ta_ta`. Die Abhängigkeit zwischen den Zuständen `counter1` und `selector` bildet jedoch einen Zyklus, weil die Zustandsaktion `a = c` die Variable `c` vom Zustand `selector` liest. Umgekehrt liest die Transitionsaktion `c = a1` die Variable `a1` vom Zustand `counter1`.

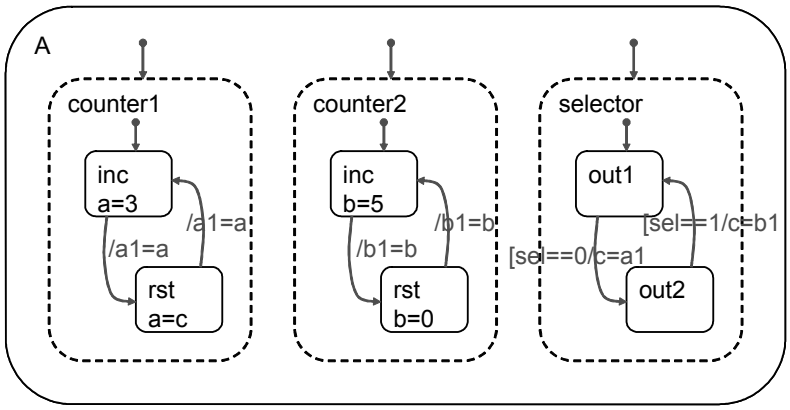


Bild 153: Chart des Modells part-test_cycle

Dieses Chart kann nur in zwei Partitionen partitioniert werden. Die Zustände `counter1` und `selector` werden `chart1` zugewiesen. Der Zustand `counter2` wird `chart0` zugewiesen (Bild 154).

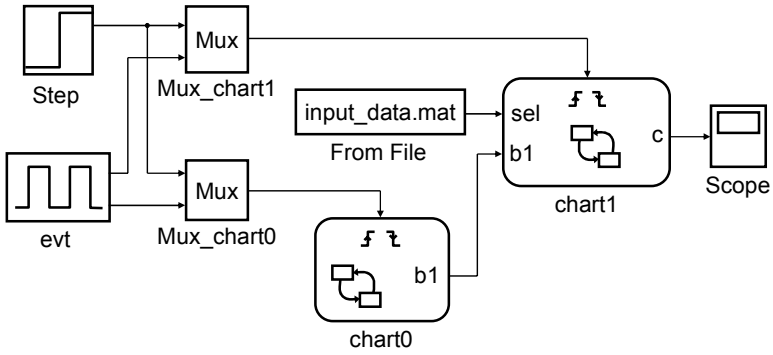


Bild 154: Chart des Modells part-test_cycle nach Partitionierung

Für die Prüfung der Korrektheit des Werkzeugs JVHDLGen werden drei verschiedene Modelle getestet. Das erste Modell ist ein Chart mit ausschließlich OR-Dekompositionen. Die anderen beiden Modelle sind Charts mit AND-Dekompositionen.

Durch das erste Modell `counter` (Bild 155) wird die von SF2VHD importierte Funktionalität getestet. Um vergleichbare Ergebnisse wie bei ModelSim zu erzielen, werden ein `reset`-Ereignis und ein `clk`-Ereignis in diesem Modell verwendet. Diese zwei Ereignisse funktionieren wie das Signal `clk` und `reset` im generierten VHDL-Code. Das Chart wird von der aufsteigenden Flanke des Ereignisses `clk` angestoßen. Die Eingangsvariable `sat_en` benutzt einen anderen Pulsgenerator. Um die Auswirkung eines Eingangssignals zu entdecken, muss sich das Taktsignal `clk` in der Testbench schneller als der Wert eines Eingangssignals ändern. Der Takt der Eingangsvariable `sat_en` wird in Simulink zehn Mal so lange wie das Ereignis `clk` gesetzt. Die Variable `count` wird als Ausgabe definiert. Um die Ergebnisse in Simulink mit denen in ModelSim anschaulich zu vergleichen, werden die Kurven von `clk`, `sat_en` und `count` zu einem `Scope` ausgegeben.

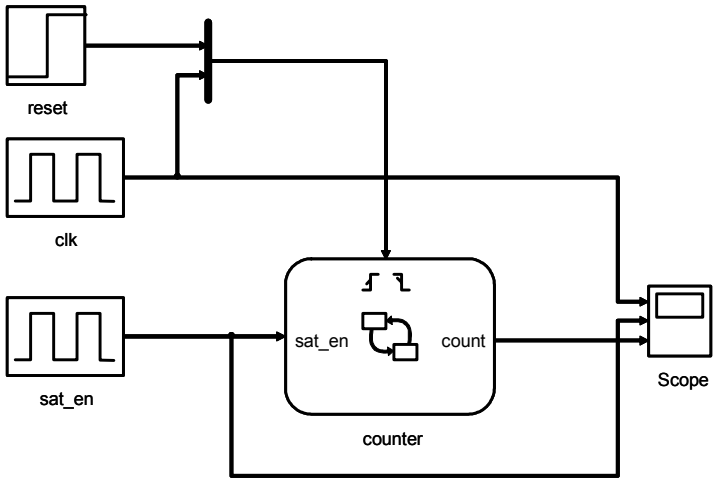


Bild 155: Modell counter

Im Chart des Modells `counter` sind verschiedene Beschreibungsmittel wie z. B. Zustandsaktionen, Transitionsbedingungen, Bedingungsaktionen und eine Bedingungs-Junction enthalten (Bild 156).

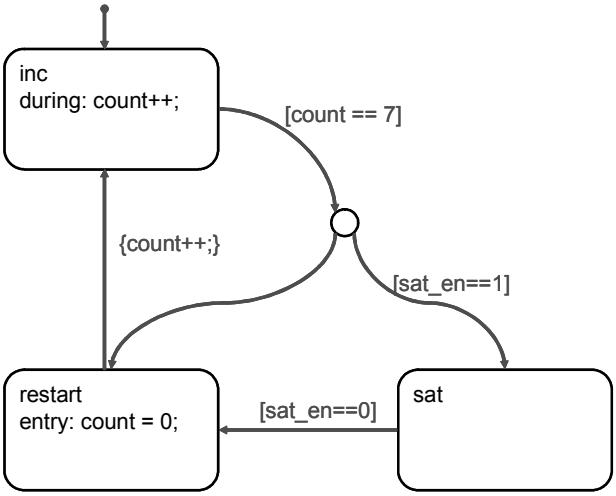


Bild 156: Chart des Modells counter

Bild 157 zeigt die Testbench für das Modell `counter` bei ModelSim. Sie besteht aus zwei Komponenten `counter` und `stimuli`.

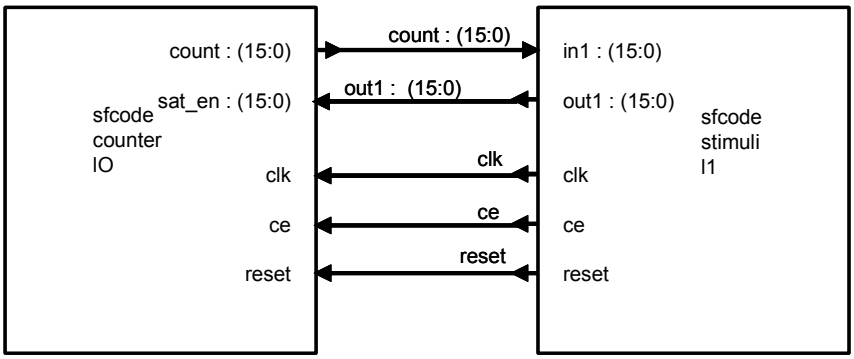


Bild 157: Testbench für das Modell counter

Das Taktsignal `clk` generiert mit der Periode 60 ns die Stimuli. Das Signal `ce` wird immer auf 1 gesetzt. Das Signal `reset` wird zum Zeitpunkt 100 ns auf 1 gesetzt, dann wird es zum Zeitpunkt 360 ns auf 0 gesetzt. Die Periode des Ausgangssignals `out1` ist 600 ns. Die Simulation bei ModelSim dauert 10 µs.

Bild 158 zeigt die Kurve des Modells `counter`. Das Fenster oben mit dem Titel `wave` zeigt die Ergebnisse von ModelSim. Das Fenster unten mit dem Titel `Scope` zeigt die Kurven von `clk`, `sat_en` und `count`. Die beiden Simulationsergebnisse des Modells `counter` von ModelSim und Simulink sind zu jedem Simulationsschritt übereinstimmend.

Durch die anderen beiden Modelle wird die erweiterte Funktionalität des Werkzeugs JVHDLGen getestet. In diesen beiden Modellen werden auch die Ereignisse `reset` und `clk` bei Simulink wie im ersten Modell `counter` verwendet.

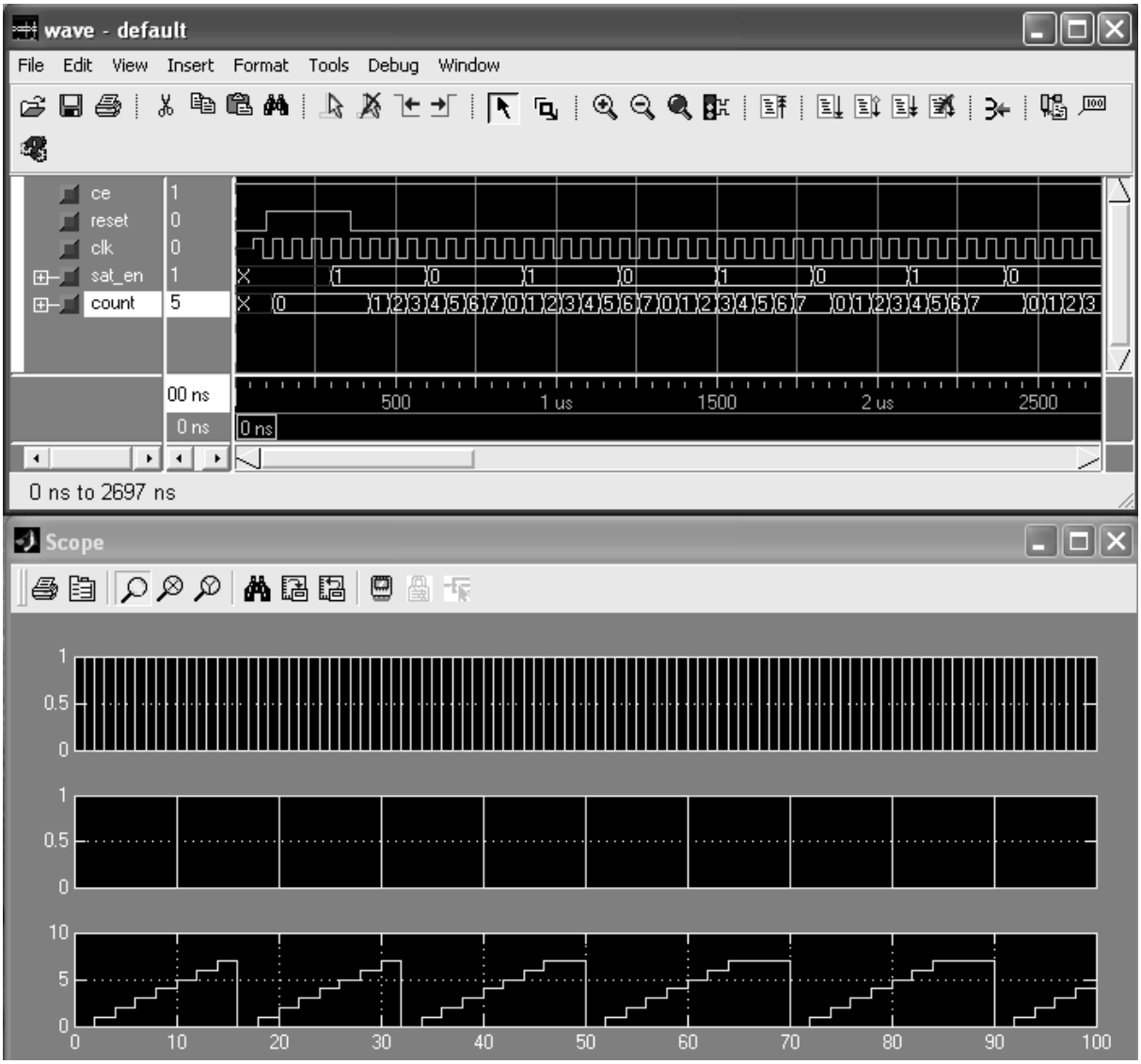


Bild 158: Signale aus der Simulation des Modells counter

Das in Bild 159 dargestellte zweite Modell `counter_sel` enthält drei nebenläufige Zustände `counter1`, `counter2` und `selector`. Im generierten VHDL-Code werden diese drei Zustände nach ihrer geometrischen Reihenfolge im Chart bearbeitet. Die Variable `sel` wird als Eingang und die Variable `c` als Ausgang definiert.

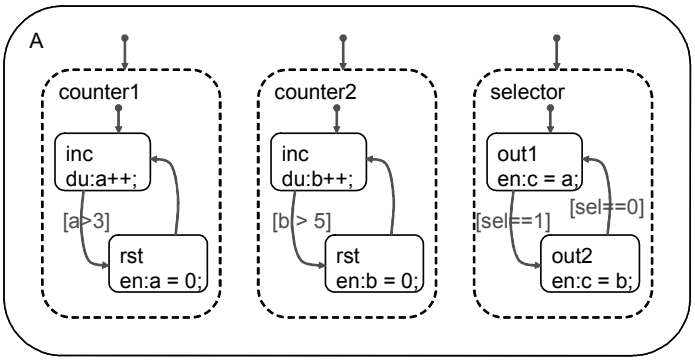


Bild 159: Chart des Modells counter_sel

Auch hier wurden übereinstimmende Ergebnisse erzielt (Bild 160). In Simulink sind die Signale `clk`, `sel` und `c` dargestellt.

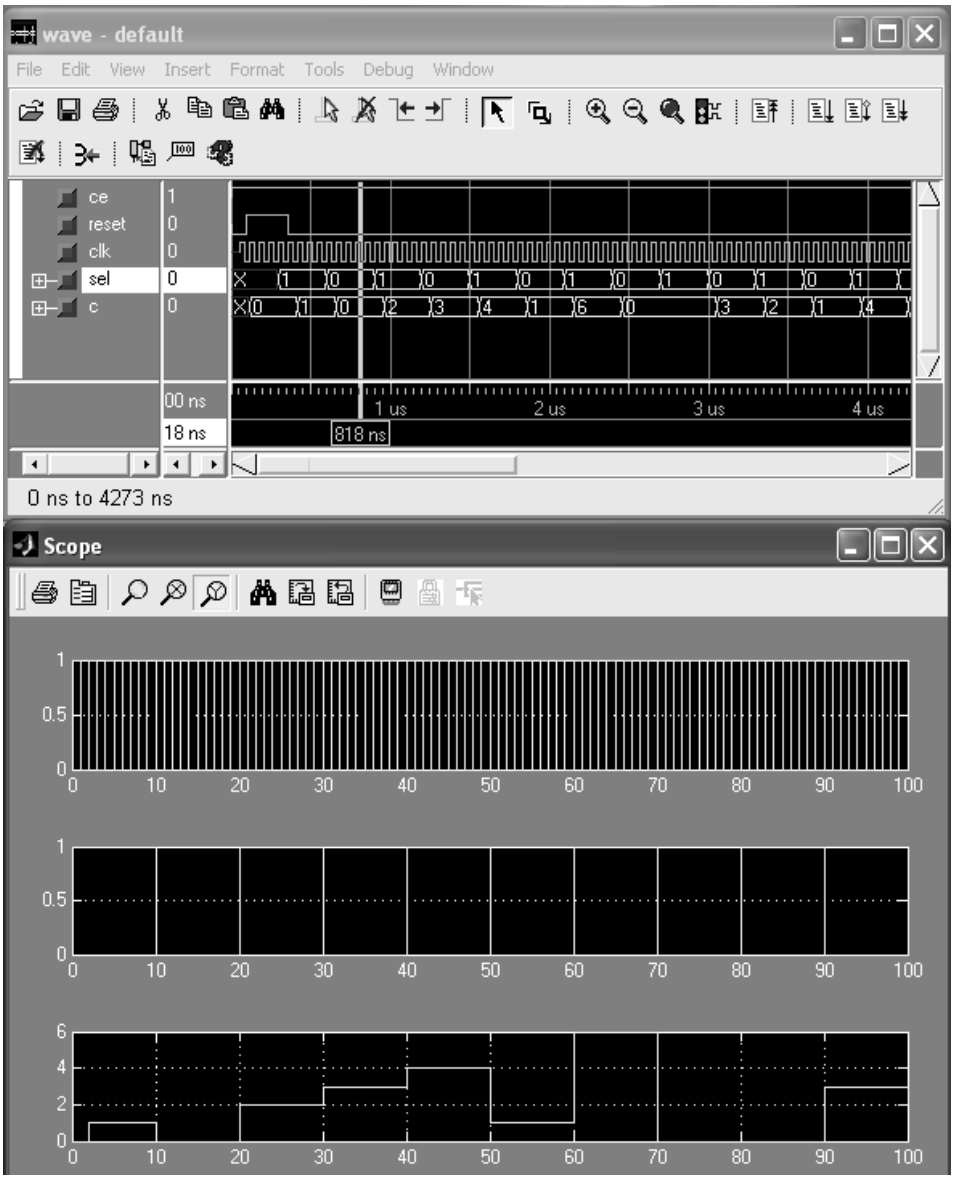


Bild 160: Signale aus der Simulation des Modells counter_sel

Das dritte Modell `s10_p2_h2` enthält verschachtelte nebenläufige Zustände (Bild 161).

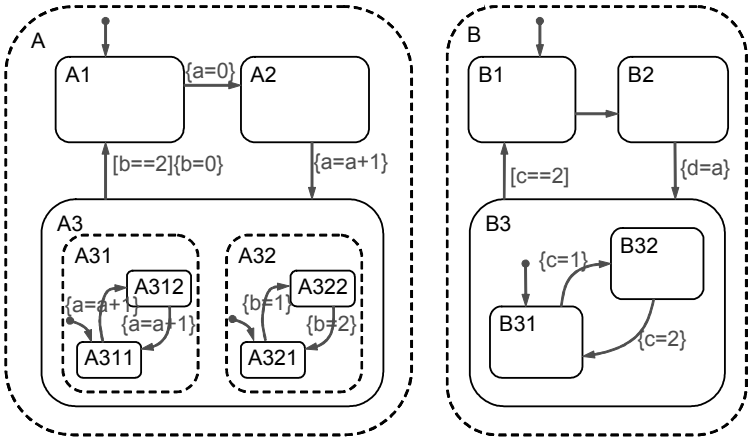


Bild 161: Chart des Modells s10_p2_h2

Das Chart enthält zwei nebenläufige Zustände `A` und `B`. Der Unterzustand `A3` enthält wieder zwei nebenläufige Zustände `A31` und `A32`.

Bild 162 zeigt die Kurven des Taktsignals `clk` und des Ausgangs `d` an.

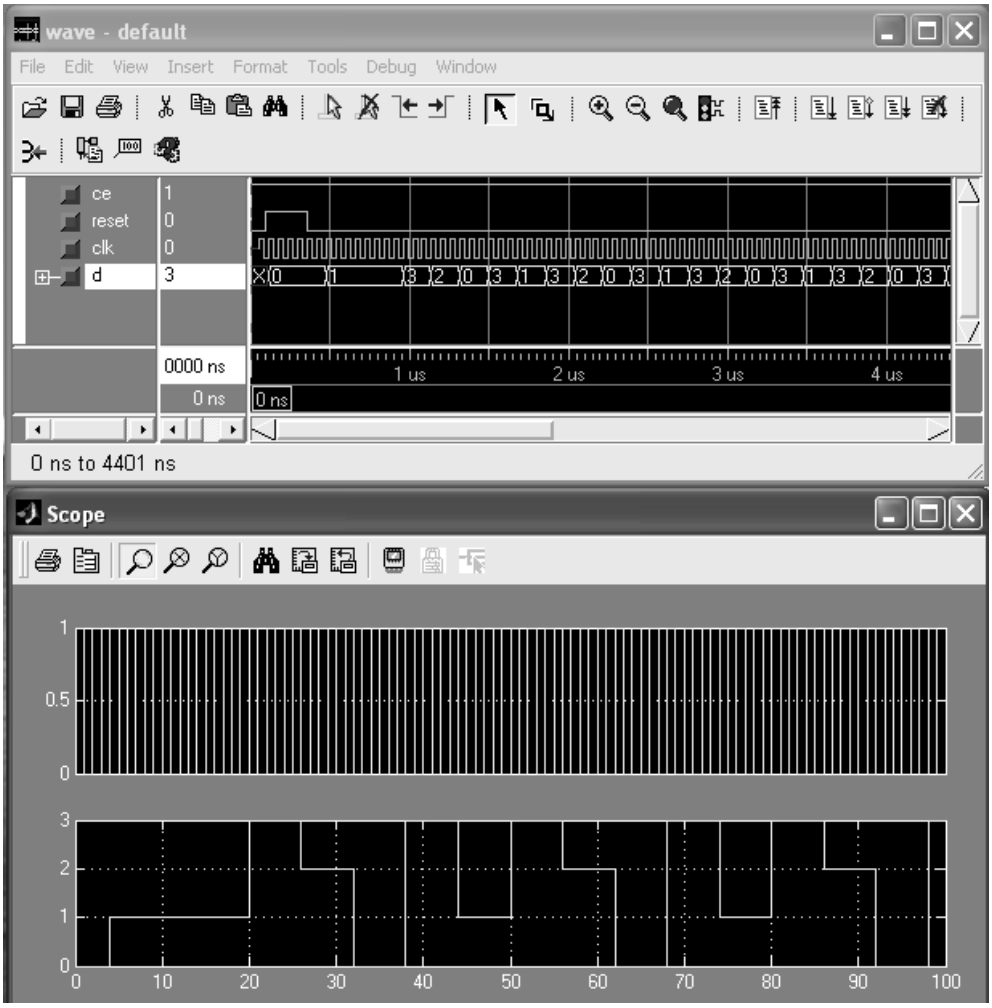


Bild 162: Wellenform der Simulation des Modells s10_p2_h2

5.3.2.2 PI-Controller

Dieses Modell stellt eine Anwendung dar, die nicht ausschließlich zu Messzwecken für diese Arbeit entworfen wurde, sondern auch in einer realen Umgebung seiner Funktion entsprechend zum Einsatz kommen könnte. Die ursprüngliche Version des Modells ist ein Beispiel von TL und implementiert einen PIPT1-Regler mit Strecke (Bild 163).

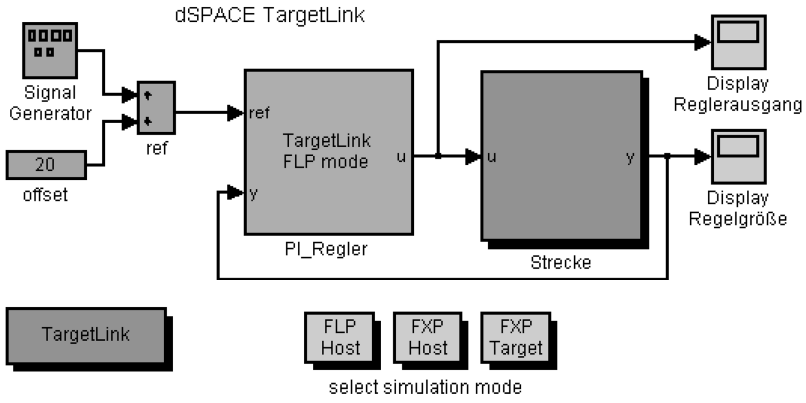


Bild 163: Ursprungsmodell von PIPT1-Regler mit Strecke

Dieses Modell wurde so modifiziert, dass der Regler auf dem MPC555 ausgeführt wird (Bild 164)[68]. Es enthält drei Subsysteme, deren Funktionalität bei der Code-Generierung mit TL jeweils in einer eigenen Task realisiert wird. Die Schedule-Task (`Tsk_schedule_tasks`) startet einen Alarm, der nach Ablauf die PI-Controller-Task (`Tsk_sub2`) aktiviert. Diese empfängt die von der CAN-ISR gesendeten Nachrichten `IN1` und `IN2`, die die vom Simulink-Modellteil gesendeten Werte der Führungsgröße (CAN-ID `0x1`) und Rückführgröße (CAN-ID `0x2`) enthalten. Dann wird die Reglerausgangsgröße (`Msg`) berechnet und gemeinsam mit der Störgröße (`Noise`) an die Kommunikations-Task (`Tsk_Kommunikation`) gesendet, die die zwei Werte als CAN-Nachrichten mit ID `0x5` und `0x6` an den Simulink-Modellteil weiterleitet. Die Kommunikations-Task wird innerhalb der PI-Controller-Task aktiviert. Nicht mit Simulink modelliert ist die Task niedrigster Priorität namens `dummy_tasks`, in dem der A/D-Wandler ausgelesen wird und der Wert der daran angeschlossenen Störgröße (`Noise`) an die Kommunikations-Task gesendet wird.

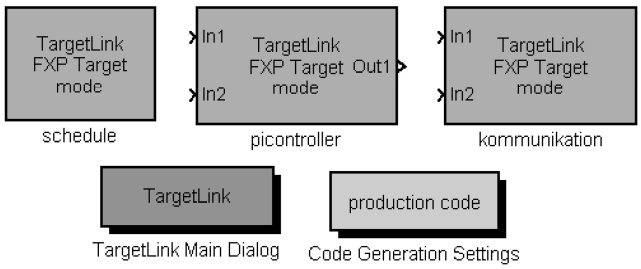


Bild 164: PIPT1-Reglermodell für MPC555

Führungsgröße, Strecke und Darstellung der Resultate bleibt in Simulink (Bild 165). Die S-Funktion `Manager` dient dazu, die Simulation so lange anzuhalten, bis eine Nachricht mit neuem Wert der Reglerausgangsgröße vorliegt. Bei Simulationsstart wird das Modell durch die S-Funktion `WaitOneStep` so lange in einem Wartezustand gehalten, bis vom MPC555 eine Startnachricht empfangen wird.

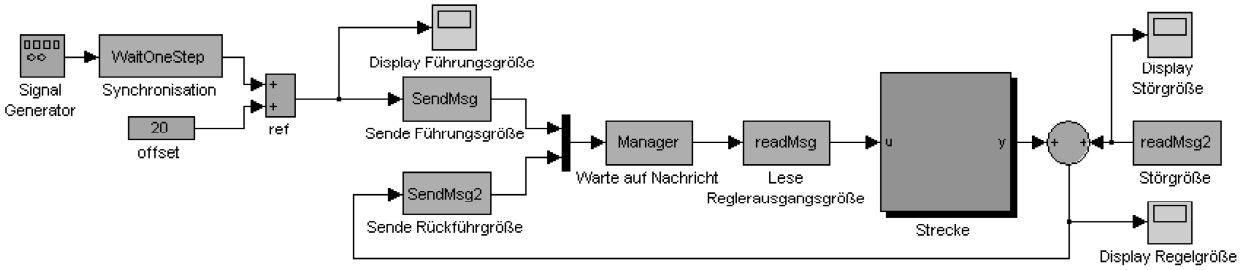


Bild 165: Strecke mit Sende- und Empfangsblöcken in Simulink

In den S-Funktionen `SendMsg` und `SendMsg2` werden die Werte der Führungsgröße (CAN-ID `0x1`) und Rückführgröße (CAN-ID `0x2`) an den MPC555 gesendet, mit den S-Funktionen `readMsg` und `readMsg2` werden Reglerausgangsgröße (CAN-ID `0x5`) und Störgröße (CAN-ID `0x6`) empfangen (Bild 166).

Bei diesem Modell wird nach Bestimmung von Buslast und Stromaufnahme das Zeitverhalten genauer analysiert. Da die Hauptfunktionalität in einer Task konzentriert und das Modell durch einen Alarm langsam getriggert ist, macht eine Verteilung des Modells auf mehrere Boards keinen Sinn. Es würde zusätzlicher Kommunikationsaufwand entstehen, ohne Erkenntnisse bezüglich Performancegewinn zu bringen.

[68] Verbindungslinien zwischen den Blöcken werden nicht benötigt.

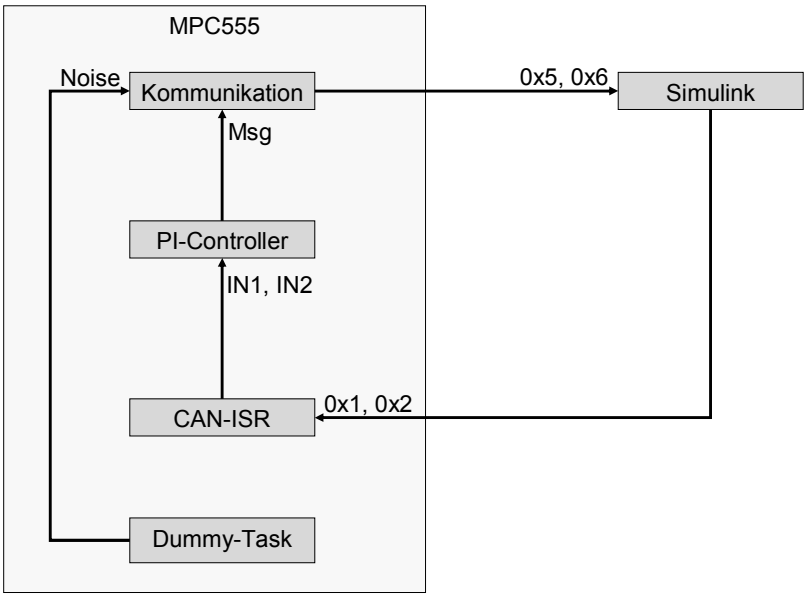


Bild 166: Nachrichtenfluss im PIPT1-Reglermodell

5.3.2.3 Modell eines ABS-Systems

Anhand eines komplexeren Modells werden die oben beschriebenen Methoden in einer realen Systemumgebung überprüft. Als Testsystem wird ein Modell [Lutz03] eines Antiblockiersystems (ABS) mit Schlupfregelung verwendet, wobei von einem Personenkraftfahrzeug mit Heckantrieb ausgegangen wurde (siehe auch [KrTa03]). Bei den Untersuchungen werden neben funktionalen auch nicht funktionale Randbedingungen berücksichtigt: Neben den Voraussetzungen zur Ausführung von Modellen auf unterschiedlichen Targets wird überprüft, welchen Einfluss die Modellstruktur (z. B. die Anzahl der Tasks) und unterschiedliche Partitionierungen der kontinuierlichen und zustandsbasierten Anteile der Testmodelle auf die PC-, MPC555- und FPGA-Knoten hinsichtlich Speicherverbrauch, Performance, Zeitverhalten, Buslast und Energieverbrauch haben. Zur Untersuchung des Zeitverhaltens wird die Ausführungsdauer pro Task, deren Aufrufreihenfolge, die ISR-Aufrufreihenfolge sowie weitere relevante Modellvariablen an Simulink übertragen.

Während eines Bremsvorgangs können sowohl Haft- (Räder drehen sich) als auch Gleitreibung (Räder blockieren) zwischen Reifen und Fahrbahn entstehen. Ist ein Fahrzeug mit einem Antiblockiersystem ausgestattet, so tritt beim Bremsen hauptsächlich Haftreibung auf, was eine Verkürzung des Bremswegs zur Folge haben kann. Oberstes Ziel ist es also den Bremsweg zu minimieren, indem ein Blockieren der Räder vermieden wird, so dass stets Haftreibung auftritt. Desweiteren ist das Fahrzeug während des Bremsvorgangs dann lenkbar und in einem stabilen Zustand.

Fahrphysikalische Grundlagen: Die durch Reibung verursachte Verzögerungskraft ist über die Normalkraft vom Reibungskoeffizienten abhängig. Es werden drei Arten von Reibung unterschieden: Haft-, Gleit- und Rollreibung. Jede besitzt ihren charakteristischen Reibungskoeffizienten. Mittels des Reibungskoeffizienten μ_R und der Normalkraft F_N lässt sich die maximal erreichbare Bremskraft (Reibkraft) F_R berechnen:

$$F_R = \mu_R \cdot F_N$$

Der Bremsweg S_B eines Fahrzeuges auf ebener Straße hängt von seiner Ausgangsgeschwindigkeit v_0 und dem Reibungskoeffizienten μ_R ab:

$$s_B = 0{,}5 \cdot \frac{v_0^{\,2}}{g \cdot \mu}$$

Beim Abbremsen eines Rades entsteht eine Differenzgeschwindigkeit zwischen Rad und Fahrzeug, das Rad dreht sich langsamer als es sich bei der eigentlichen Fahrzeuggeschwindigkeit drehen müsste. Die Definition des Bremsschlupfes λ in Abhängigkeit von der Fahrzeuggeschwindigkeit v_F und der Radumfangsgeschwindigkeit v_R zeigt folgende Formel. Ein Schlupf von 100 % bedeutet, dass das Rad stillsteht bzw. blockiert. Bei einem Schlupf von 0 % dreht sich das Rad entsprechend der Fahrzeuggeschwindigkeit, d. h. es läuft frei.

$$\lambda = \frac{v_F - v_R}{v_F} \cdot 100\%$$

Die Radumfangsgeschwindigkeit v_R errechnet sich aus dem Radius r_R und der Drehfrequenz f_R:

$$v_R = \omega \cdot r_R$$

Die am Reifen auftretenden Kräfte Normalkraft F_N, Peripherkraft F_U und Seitenführungskraft F_S stehen jeweils orthogonal zueinander. Je nach Vorzeichen wirkt die Peripherkraft als Brems- bzw. Beschleunigungskraft des Fahrzeuges. Man erkennt, dass sich die aus der Normalkraft resultierende Reibkraft auf die beiden Kraftvektoren F_U und F_S verteilt. Je nach Einlenkwinkel variieren die Beträge von F_U und F_S, und somit auch Bremsweg und Fahrstabilität. Bei der Kurvenfahrt stellt sich also stets eine Kompromisslösung zwischen Lenkbarkeit, Fahrstabilität und Bremsweg ein.

Simulink-Modell: Die Modellierung des Antiblockiersystems in Simulink beschränkt sich auf eine reine Schlupfregelung, d. h. es werden keine Beschleunigungen über Beschleunigungssensoren im Regelalgorithmus berücksichtigt. Weiterhin wurde in [Lutz03] von einem Personenkraftfahrzeug mit Heckantrieb ausgegangen, da hierbei die Ermittlung der Referenzgeschwindigkeit mit geringem Aufwand bestimmt werden kann. Hierfür sind insgesamt drei Drehzahlfühler notwendig, die sich vorne links und rechts sowie einmal am Heck des Fahrzeugs befinden.

Partitionierung und Allokation der Funktionalitäten: Aufgrund der besonderen Eigenschaften des Xilinx SystemGenerator, des Altera DSP Builders, des Stateflow- und des TL Blocksets sind sinnvolle Partitionierungen teilweise festgelegt. Die Modellierungsmöglichkeiten eines Zustandsautomaten mit dem SystemGenerator bzw. dem DSP Builder sind gegenüber Stateflow so gering, dass für ein komplexeres Steuerwerk lediglich TL entsprechende Voraussetzungen liefert, da es in vollem Umfang die Mächtigkeit des Stateflow-Blocksets unterstützt. Da SystemGenerator und DSP Builder-Blockset vollständig in HW umgesetzt werden, und dadurch die Möglichkeit des parallelen Abarbeitens von Algorithmen gegeben sind, macht es Sinn, die Signalaufbereitung und Verarbeitung mit dem SystemGenerator bzw. DSP Builder Blockset zu implementieren.

Eine Anforderung an die Steuereinheit ist, dass bereits nach kurzer Zeit möglichst genaue Informationen bezüglich Drehzahl oder Beschleunigung vorliegen müssen, falls die Regelung effektiv arbeiten soll. Laut [Burc93] wurde die Erfahrung gemacht, dass zur Erzielung einer einwandfreien Reglergüte ein Fehler von maximal 1 % anzustreben ist. Dies gelingt nur, falls während einer Messperiode mindestens 60 Impulse generiert werden. Nach [Burc93] muss zur Gewährleistung einer ausreichend guten Reaktion bereits nach spätestens 10 ms ein entsprechender Messwert vorliegen und falls nötig die Regelung eingreifen.

5.3.2.4 Verifikation der Integration von RP.2002

Zum einen muss an jedem Punkt im Entwurfsfluss sichergestellt sein, dass die Funktionalität des Modells nicht verändert wurde, zum anderen soll die anschließende Simulation gleiche Ergebnisse wie Simulink und JStateSim liefern.

Ein Modell wird zunächst als Statechart erstellt und anschließend in Simulink simuliert. Für die Verifikation des Entwurfsflusses wurde ein spezielles Modell verwendet, das bei jedem Zustandswechsel eine Zählervariable erhöht und diese ausgibt. Somit kann das Einhalten von Konditionen und der

sichere Übergang von Zuständen überwacht werden. Nach der VHDL-Code-Generierung wird das Modell in FPGA-Advantage importiert und kann wieder graphisch dargestellt werden. Die beiden Darstellungen (Bild 167) sollten keine Unterschiede aufweisen.

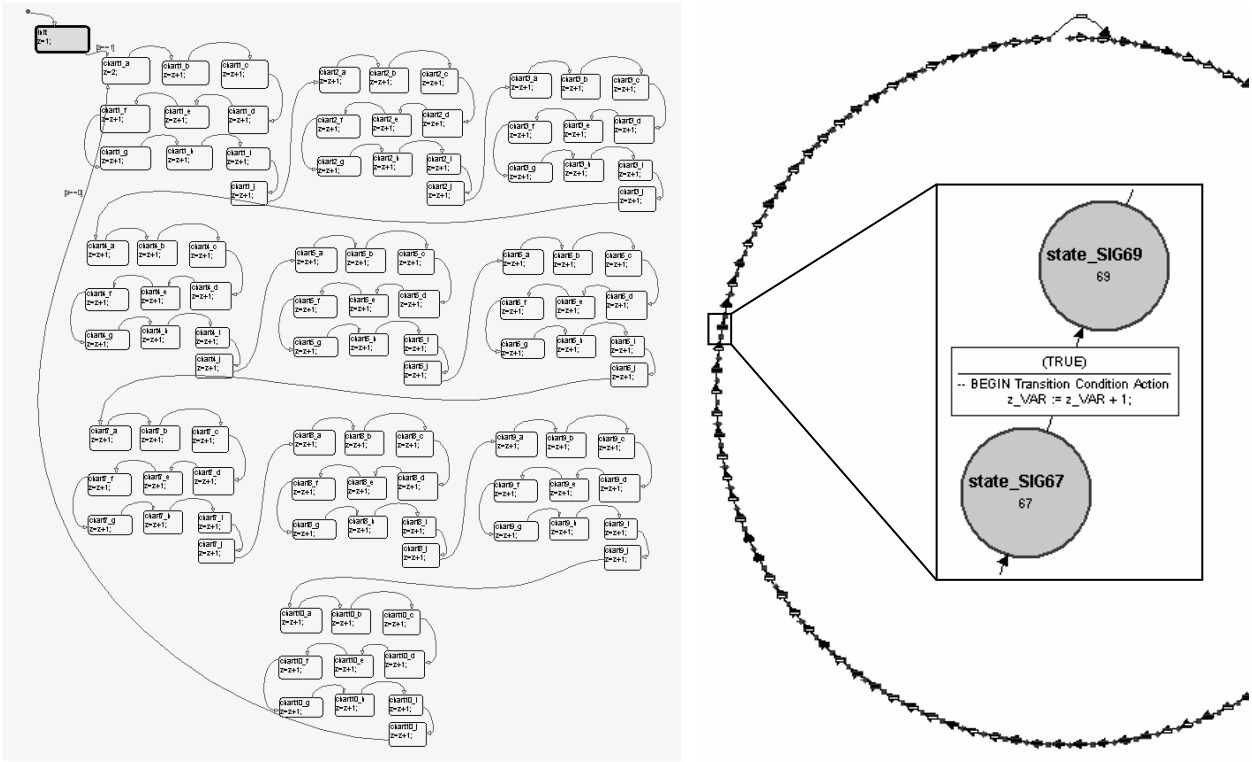


Bild 167: Modell in Stateflow und FPGA-Advantage

Nachdem das Modell in den HW-Teil des Simulators integriert ist, zeigt eine Logiksimulation an einer Testbench die Funktionsfähigkeit des Modells. Die Testbench simuliert dabei die SW-gesteuerten Zugriffe auf das FPGA. Nach der Synthese und anschließender Konfiguration des FPGAs testet ein C-Programm das korrekte Arbeiten des Modells.

Zum Testen der korrekten verteilten Simulation wird ein Modell einer Simulation in Simulink unterzogen. Die erhaltenen Werte werden in zwei Dateien geschrieben, die die angestrebte Partitionierung des Modells repräsentieren (Bild 168). Nach erfolgter Teilung wird das Modell mit JStateSim simuliert, die Werte in Simulink eingelesen und in gleicher Weise wieder in Dateien geschrieben. Durch den Vergleich der vier Dateien wird sichergestellt, dass JStateSim das Modell auf gleiche Art wie Simulink bearbeitet.

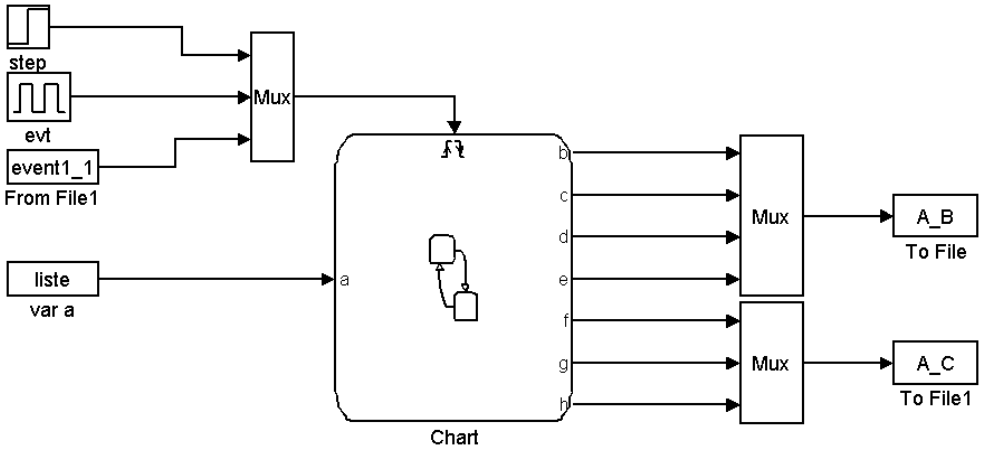


Bild 168: Verifikation der Simulation

Das obige Modell hat die folgenden Eigenschaften:

- Es enthält sowohl nebenläufige Zustände als auch Hierarchie.
- Nach jeder Änderung im Modell wird eine Variable geändert.
- Die Variablen werden in eine Datei geschrieben. Jede Zeile entspricht einem Zeitschritt. Die erste Spalte entspricht dem Zeitstempel, die anderen Spalten sind die Werte der Variablen.

Das obige Modell wird durch ein Partitionierungsprogramm in zwei Teilmodelle partitioniert.

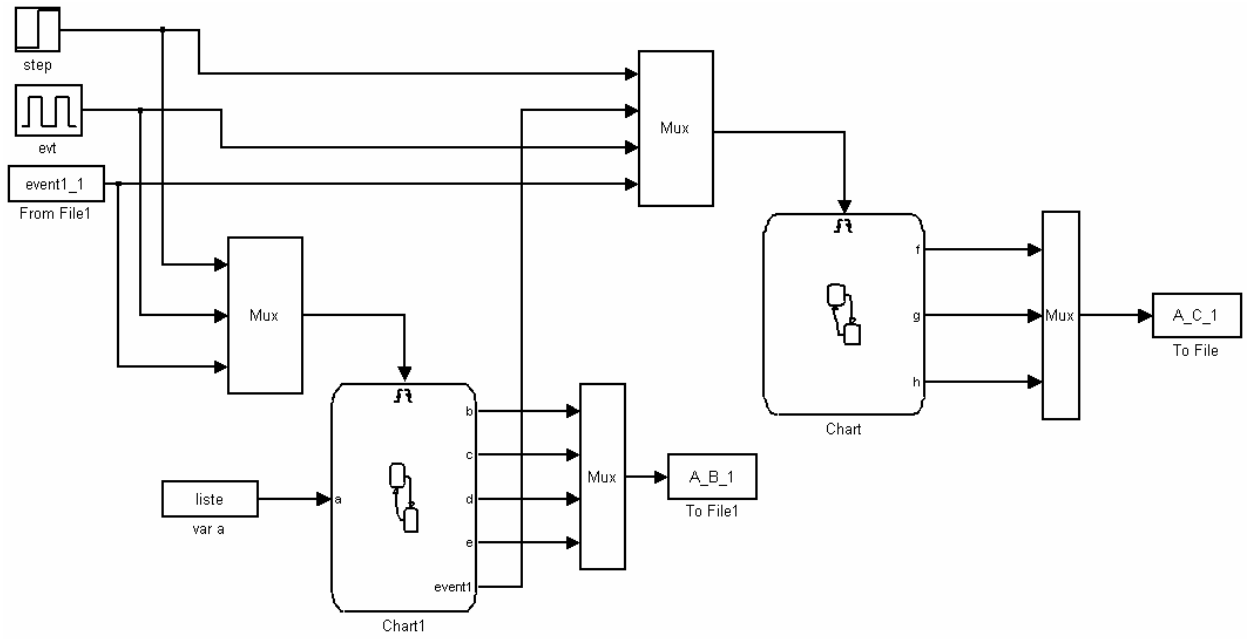


Bild 169: Verifikation der verteilten Simulation

Nach der Partitionierung wurden die beiden Teilmodelle auf zwei Rechner verteilt simuliert und die Ergebnisse in zwei Dateien gespeichert. Die beiden Dateien wurden dann durch ein Programm mit den originalen Dateien (von der eigenständigen Simulation) verglichen. Die verteilte Simulation wurde einmal nur mit JStateSim, einmal nur mit Stateflow und schließlich mit beiden durchgeführt. Beim Einsatz von Simulink muss eine S-Funktion[69] verwendet werden.

Wichtig für die Bewertung des Gesamtsystems zur Simulationsbeschleunigung ist neben der Überprüfung des richtigen Arbeitens der Umgang mit den verwendeten Ressourcen. Im Folgenden soll untersucht werden, welcher Ressourcenbedarf bei bestimmten Modellen an den FPGA gestellt wird. Bild 170 zeigt beispielhaft die Belegung der Elemente (Slices), in denen die Logik realisiert ist, und des Block-RAMs des im RP-System verwendeten FPGAs.

[69] Eine S-Funktion stellt eine in einer Programmiersprache formulierte Block-Beschreibung dar, die es ermöglicht, Simulink-Blöcke mit anwenderspezifischer Funktionalität zu entwerfen. S-Funktionen können als MATLAB-M-Skript, in C, C++, Ada und Fortran implementiert werden und werden mit Hilfe des S-Funktion-Blocks in ein Simulink-Modell eingebunden. Abgesehen von als MATLAB-M-Skript geschriebenen S-Funktionen, muss der Code vor der Ausführung mit dem mex-Compiler als mex-Datei compiliert werden, der in Windows als Objektcode in Form einer DLL ausgegeben wird [AIEM04].

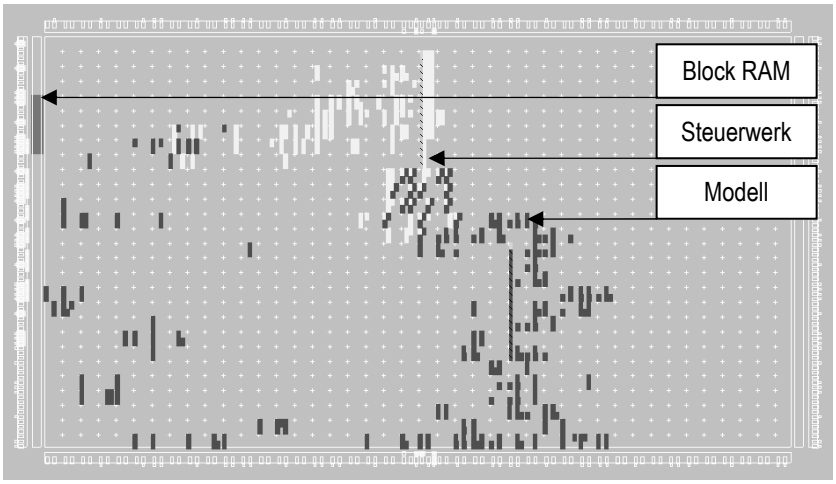


Bild 170: Belegung des FPGAs

Das Steuerwerk belegt als konstante Komponente 2 % der im verwendeten FPGA vorhandenen Slices. Von diesem stammt auch die Belegung eines der 14 RAM-Blöcke (Block-RAMs). Die variable Belegung in Abhängigkeit von der Größe des Modells zeigt Bild 171. Dabei wird ein ungefährer linearer Zusammenhang zwischen beiden Größen sichtbar. Der Anstieg des Ressourcenbedarfs auf über 100 % bedeutet, dass dieser Entwurf auf dem verwendeten FPGA nicht platziert werden kann, sehr wohl aber auf einem größeren.

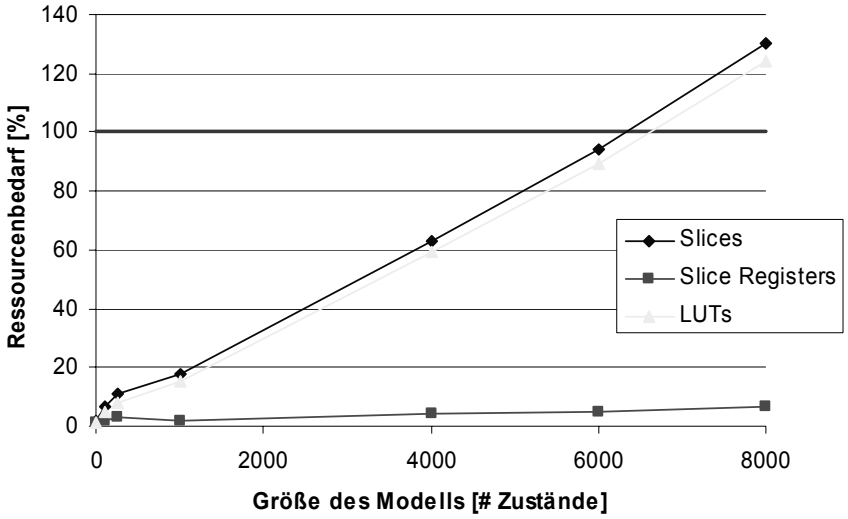


Bild 171: Ressourcenbedarf auf dem FPGA

In Bild 172 ist die Struktur der verwendeten Testmodelle abgebildet. Bei jedem Übergang zum nächsten Zustand wird eine Zählervariable erhöht. Aus dieser Anordnung wird ein endlos laufendes Statechart der gewünschten Länge erstellt. Hierzu werden Zustände so miteinander verbunden, dass sie einen Kreis bilden (anschaulich in Bild 167), nach dessen Durchlauf der Zählerstand ausgegeben sowie auf eine Bedingung gewartet wird, die den nächsten Durchlauf startet. Die Bedingung wird aus einer Liste, die in einer Datei gespeichert ist, ausgelesen.

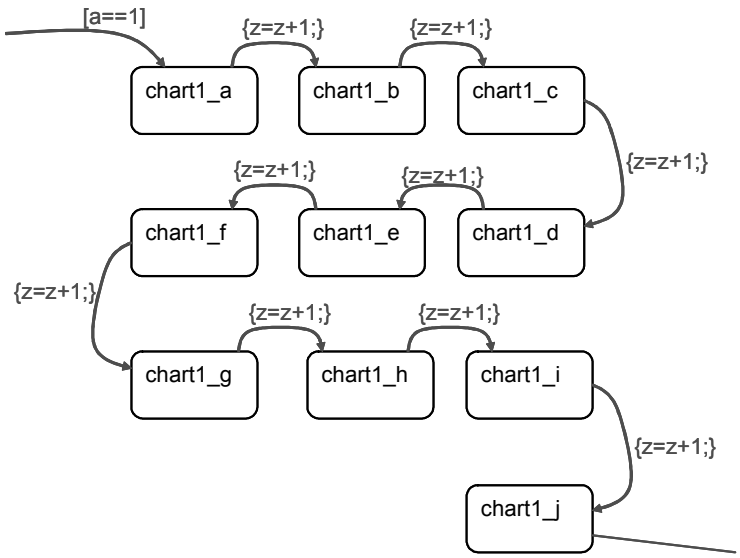


Bild 172: Testmodell für Ressourcenverbrauch

In einem weiteren Test wurden die Auswirkungen unterschiedlicher Modelleigenschaften auf den Ressourcenverbrauch im FPGA untersucht. Dazu wurde ein endlos laufendes Statechart mit 300 Zuständen verwendet und mit unterschiedlichen Eigenschaften versehen. In Bild 173 sind dazu verschiedene Ausschnitte der verwendeten Modelle abgebildet.

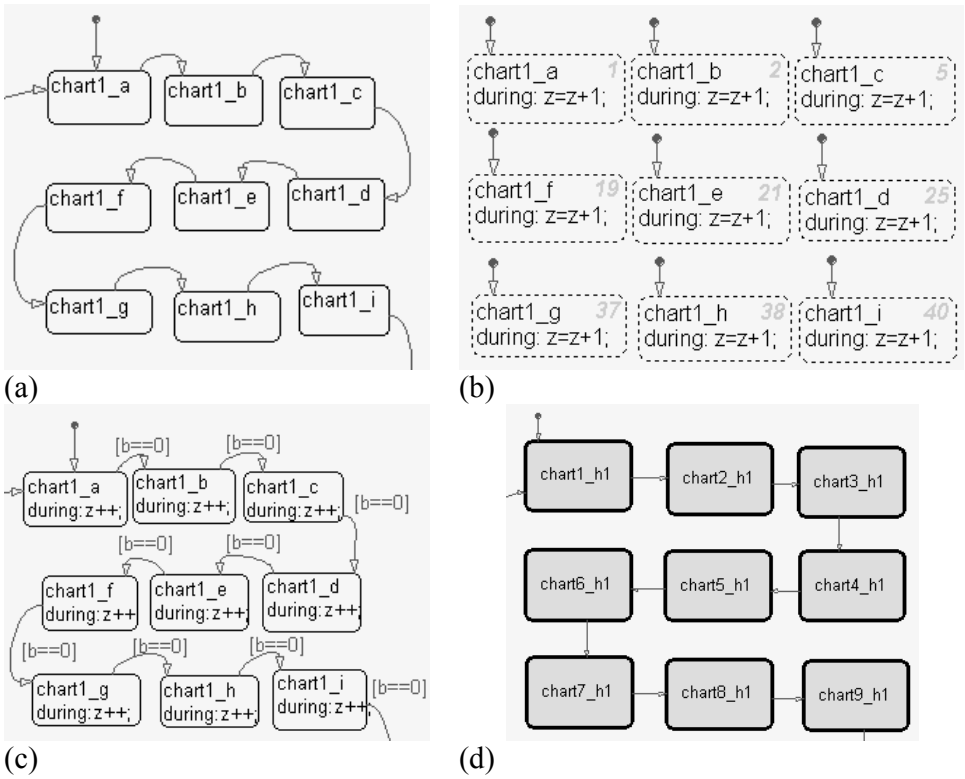


Bild 173: Verschiedene Modelle zum Test des Ressourcenverbrauchs

Die Ergebnisse der einzelnen Platzierungs- und Verdrahtungsläufe werden nach der Anzahl der verwendeten Slices verglichen, die einen Hinweis auf die tatsächliche Auslastung des FPGAs geben (beispielsweise könnten auch die Verdrahtungskanäle zum Engpass des Modells werden). Die Anzahl der verwendeten Register verläuft bei allen betrachteten Messungen parallel zu den belegten Slices.

Eine Interpretation der Ergebnisse aus Tabelle 10 ist schwierig, da die Werte dicht beieinander liegen und sich somit kein Trend zugunsten einer Methode erkennen lässt.

	Slices	Prozent
Steuerwerk alleine	63	2%
Keine Beschriftungen auf den Transitionen: Bild 173 (a)	262	11%
Beschriftungen: Bedingungen: ähnlich Bild 173 (c)	231	9%
Beschriftungen: Aktionen: ähnlich Bild 173 (c)	256	10%
Nebenläufigkeit: Bild 173 (b)	122	5%
2 Hierarchiestufen: Bild 173 (d)	146	6%
10 Hierarchiestufen: Bild 173 (d)	145	6%
Entry in jedem Zustand: ähnlich Bild 173 (c)	259	11%
During in jedem Zustand: ähnlich Bild 173 (a)	259	11%
During und Bedingung: Bild 173 (a)	229	9%

Tabelle 10: Ressourcenbedarf auf dem FPGA

Neben der Abschätzung des Ressourcenbedarfs soll noch die erwartete Simulationsgeschwindigkeit abgeschätzt werden. Die Randbedingungen für diese Abschätzung sind sowohl die Taktfrequenz von 33 MHz, die am FPGA anliegt und als Taktsignal für das Steuerwerk dient, als auch die Anzahl der Zustände des Steuerwerkes. Wird als maximale Simulationsdauer ein Zeitschritt vorgegeben, müssen dafür 12 Zustände durchlaufen werden. Bei einer Periodendauer von 30 ns ergibt sich eine Dauer d von:

$$d = 12 \times 30\ ns \approx 360\ ns\ .$$

Wird im Gegensatz dazu eine höhere Anzahl z an Simulationsschritten zugelassen, sind für jeden einzelnen Schritt nur 2 Zustände zu durchlaufen, womit sich die Dauer d_s pro Simulationsschritt ergibt zu:

$$d_s = \lim_{z \to \infty} = 60\ ns$$

Es ist somit sinnvoll, immer eine möglichst hohe Simulationsdauer für den FPGA zu ermöglichen. Eine unendlich große Zahl kann dem FPGA nicht übergeben werden. Die Implementierung des DP-RAMs beschränkt die Größe der übergebenen Werte auf 16 Bit, also Zahlen kleiner 65535. Es besteht jedoch aus jetziger Sicht kein Hinderungsgrund, diesen Wert auf 32 oder 64 Bit zu erhöhen.

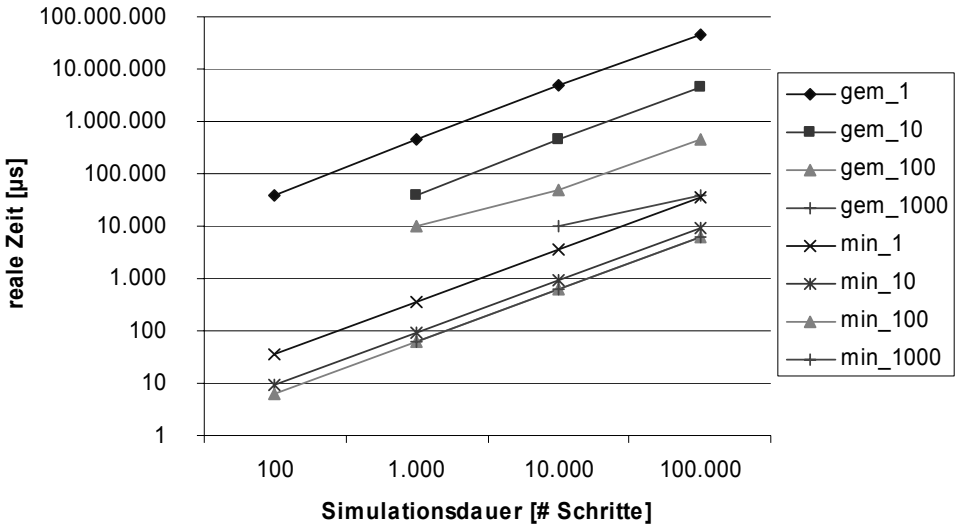


Bild 174: Gemessene und erwartete Werte

Die Diskrepanz zwischen der mindestens benötigten Zeit (Suffix: `min_`) nach der oben geführten Abschätzung und der tatsächlich gemessenen Zeit (Suffix: `gem_`) zeigt Bild 174. Die Zahlen 1, 10, 100 und 1000 in der Legende geben dabei die maximal zulässige Simulationsdauer, die das FPGA in einem Schritt rechnen darf, an. Wie erwartet, besteht auch bei den gemessenen Zeiten ein linearer Zusammenhang zwischen der Simulationsdauer und der benötigten Zeit. Auch die Geschwindigkeitszunahme bei möglichst langer eigenständiger Rechnung ist deutlich zu sehen. Nicht weiter untersucht wurde die scheinbare Grenze von 10 ms, unterhalb derer das Messverfahren (Bild 175) keine sinnvollen Ergebnisse mehr liefert [KeRi90].

```
#include <time.h>
clock_t start, stop;
int main(int argc, char** argv) {
   start= clock();
     // zu messender Programmcode
   stop=clock();
   printf("Zeit: %.4f\n",(float)(stop-start)/(float)CLOCKS_PER_SEC);
}
```

Bild 175: Zeitmessung im C-Code

Die erheblichen Abweichungen von zwei bis drei Größenordungen zwischen den beiden Gruppen lassen sich auf die Kommunikation über Kernelaufrufe zurückführen.

5.4 Performance-Messungen

5.4.1 Netzwerk- und Bus-Kommunikation

5.4.1.1 Java-RMI und Java-/C-Sockets

Für die statistischen Messungen an einem Simulink-Modell wurden bei den zu untersuchenden Modellen jeweils am Beginn und am Ende der Simulation so genannte Callback-Funktionen `StartZeitSpeichern` und `StopZeitSpeichern` definiert, die zusammen die für die Simulation benötigte Zeit ausgeben (Bild 176).

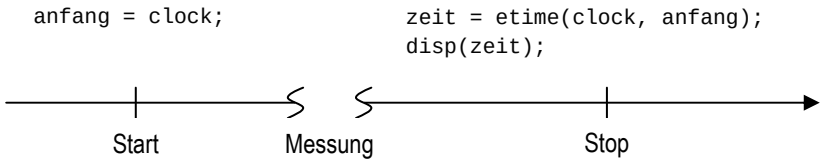


Bild 176: Callback-Funktionen zur Zeitmessung

Die Funktion `clock()` speichert in der Variablen `anfang` einen 6-elementigen Datumsvektor. Von diesem bildet `etime()` die Differenz zum gegenwärtigen Zeitpunkt, die mit `disp()` ausgegeben wird.

Dieselbe Aufgabe wurde in Java mittels RMI implementiert sowie in C mittels Sockets und TCP [Stev98]. Die schnellere Variante UDP wurde nicht verwendet, da mit dem Hintergrund einer verteilten Simulation eine sichere Verbindung gebraucht wird, Paketverluste also nicht hinnehmbar sind. Der Test besteht darin, dass ein Server auf Anfrage eines Clients den Wert einer Zählervariablen zurückschickt und diese danach erhöht. Der Server lief in beiden Fällen auf dem RP.2002-System. Bild 177 zeigt den konstanten Unterschied von mehr als einer Größenordnung zwischen beiden Methoden.

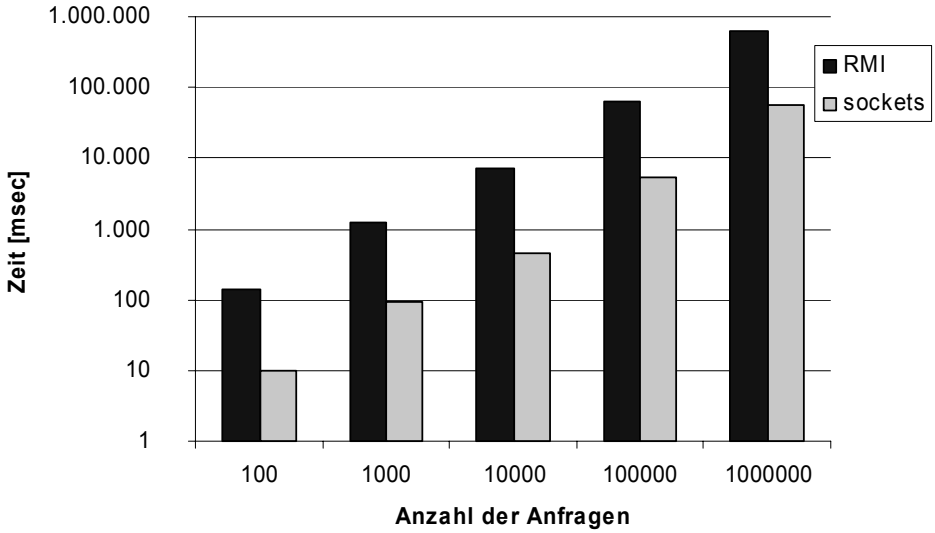


Bild 177: Unterschied zwischen RMI und TCP/IP

Obwohl der Geschwindigkeitsvorteil deutlich sichtbar ist, kann auf Java bei der Implementierung nicht ohne weiteres verzichtet werden. Der Aufwand, eine ähnlich komplexe Kommunikationsstruktur in C aufzubauen, ist erheblich.

TCP benötigt unter Java ca. 200 ms, um eine Verbindung herzustellen. Bei RMI wird die Verbindung innerhalb der JVM nicht untergebrochen, so dass wenn das Programm versucht, die Verbindung neu herzustellen, nur eine Referenz (Remote-Objekt) zurückgegeben wird. Bei kleineren Datenblöcken zeigt RMI eine höhere Datenrate als TCP.

5.4.1.2 Auslastung des CAN-Busses

Zwei Eigenschaften haben direkten Einfluss auf die Busauslastung: die Anzahl der Nachrichten pro Simulationsschritt und die Simulationsdauer eines Simulationsschritts [Grau04]. Bei gleich bleibender Ausführungsgeschwindigkeit mit steigender Nachrichtenzahl pro Simulationsschritt steigt die Buslast entsprechend; ebenso resultiert aus einer wachsenden Simulationsgeschwindigkeit bei konstanter Nachrichtenzahl pro Simulationsschritt eine steigende Busauslastung. Im Fall einer Busüberlastung kann die Kommunikation auf mehrere Busse verteilt werden. Es entsteht dabei kein zusätzlicher Rechenaufwand, wenn zum Buswechsel auf eine Komponente zurückgegriffen wird, die ohnehin im System als Funktionseinheit verwendet wird und die auf empfangene CAN-Nachrichten mit entsprechenden ausgehenden CAN-Nachrichten reagiert. S-Funktionen in Simulink eignen sich nicht als Einheiten für CAN-Buswechsel, da die Simulink-intern geregelte Ausführungsreihenfolge der Blöcke zu einem falschen Systemverhalten führt.

Senderate: Um die maximale Frequenz des `SystemTimer` zu ermitteln, bei der noch jede durch einen Alarm ausgelöste Task-Aktivierung[70] vollständig abgearbeitet wird, wurde in einer Messreihe die Frequenz des `SystemTimer` kontinuierlich gesteigert und die Senderate beobachtet. Wie Bild 178 zeigt, kann ab 2146 Hz nicht mehr jede Task vollständig ausgeführt werden, und die Senderate sinkt. Bei 7700 Hz wird keine Task mehr vollständig ausgeführt.

Wird nur bei jeder zehnten Task-Aktivierung eine CAN-Nachricht versendet, kann der `SystemTimer` um 830 Hz höher getaktet werden, ohne dass es zu Kommunikationsfehlern kommt. Es ist also nicht möglich, durch eine Skalierung der Senderate die Frequenz des `SystemTimer` ohne Ausfälle bei der Task-Aktivierung deutlich zu erhöhen.

[70] Die Aktivierung einer Task ist einem bestimmten Ereignis zugeordnet (Aktivierungsereignis, Activation Event). Diese Ereignisse sind z. B. Unterbrechungen oder das Eintreten von Zeitpunkten, zu denen nach dem Ablaufplan die Task aktiviert werden muss.

Frequency (Hz)	100	1000	2000	2125	2140	2144	2145	2146	2150	2165	2250	2500	3000	7000	7500	7600	7700
CAN messages / s	100	1000	2000	2132	2147	2150	2151	2121	2120	1930	1690	1670	1300	800	552	1	0

Bild 178: Senderate hängt von der Frequenz des SystemTimers ab

Als immer wiederkehrende Schwierigkeit hat sich die Ausführungsreihenfolge von Blöcken bei der Simulation in Simulink gezeigt. Es ist beim Systementwurf nicht immer möglich, die Eigenheiten des Simulationsverhaltens von Simulink vorauszusehen, so dass es bei der Ausführung aufgrund der „eigenwilligen“ Ausführungsreihenfolge zu nicht vorhersehbaren Komplikationen kommen kann. Empfohlen wird deshalb, möglichst wenige S-Funktionen innerhalb eines Modells zu verwenden und stattdessen die selbst programmierte Funktionalität in eine S-Funktion zu integrieren.

Optimierung durch Multiplexen: Das Problem der starken Verlangsamung bei Simulink lässt sich durch Multiplexen der zu sendenden Daten lösen, so dass alle Daten in einer einzigen Nachricht verschickt werden können. Eine Beschleunigung beim MPC555 wird durch ähnliches Vorgehen wie zuvor bei Simulink erreicht: das Multiplexen der Daten reduziert die Nachrichtenanzahl auf 1, also 12,5 % des vorherigen Werts. Das Senden beim MPC555 dauert nur noch 36 µs. Dies entspricht einer Beschleunigung um das 750-fache und führt zu einer Simulationsbeschleunigung um beinahe das 9-fache. Auch der Multiplexer benötigt Rechenzeit, allerdings fällt dieser Wert mit 19 bis 20 Zähleinheiten (16 µs) angesichts des Performance-Gewinns kaum ins Gewicht.

Kommunikationskosten: Bild 179 zeigt die Ausführungsdauer der ISR zur Behandlung von eingehenden CAN-Nachrichten. In der CAN-ISR wird zunächst die eingehende Nachricht ausgewertet, dann die Modell-Funktion ausgeführt und die Ergebnisse über CAN versendet.

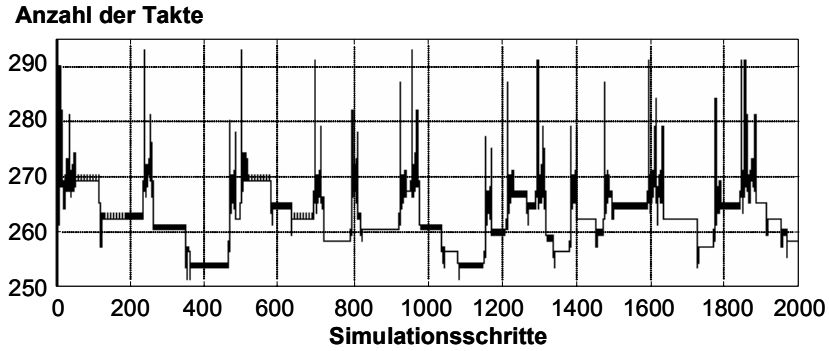


Bild 179: Simulationsdauer der CAN-ISR

Aus Bild 180 geht hervor, wie groß dabei der Anteil der Modellausführung ist. Werden diese Werte in Relation zu den Daten aus Bild 179 gesetzt, zeigt sich, dass der MPC555 nun zu ca. 80 % mit der Modellausführung beschäftigt ist und der Kommunikationsaufwand mit den verbleibenden 20 % nur

noch einen geringen Einfluss auf die Ausführungsdauer hat. Die Summe der Rechenzeiten für den kontinuierlichen und diskreten Modellteil bei verteilter Simulation liegt nur wenige µs über der in Bild 180 gezeigten Daten bei Simulation auf einem MPC555. Somit ist keine Beeinträchtigung der Performance durch die Modellverteilung festzustellen.

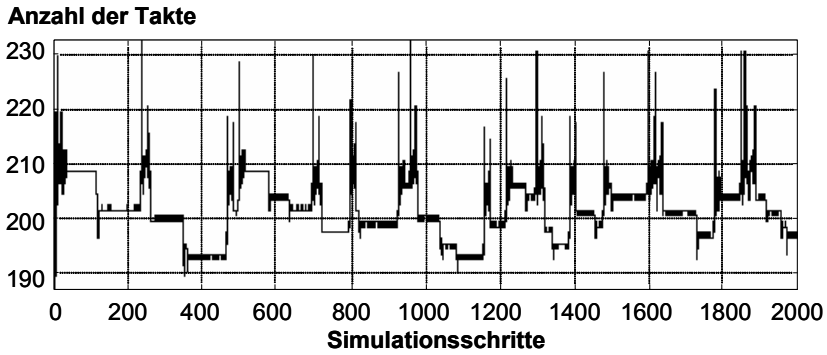


Bild 180: Simulationsdauer des Modells auf einem MPC555

Im Gegensatz dazu ist durch Verteilung allerdings auch keine Beschleunigung zu erreichen, da die Modellteile mit sequenziellem Simulationsverfahren ausgeführt werden und gegenseitig von den Ausgangsdaten der anderen Modellteile abhängig sind. Dadurch entsteht eine seriell-zyklische Ausführungsreihenfolge ausgehend von der Stimulierzeugung in Simulink über die auf HW ausgeführten Teilmodelle zurück zu Simulink, wo eine Darstellung der Ergebnisse und Rückkopplung einiger Daten auf die Systemeingänge für den nächsten Simulationsschritt erfolgt. Bei der Version mit Verwendung des FPGAs wurde eine ebenfalls schnelle Simulation erreicht. Die Berechnung des kontinuierlichen Teils unter Simulink braucht auf einem PC ungefähr die 5-fache Zeit wie der auf dem FPGA ausgeführte Stateflowchart-Teil.

5.4.2 Logikschaltungen und Stateflowchart-Modelle

Es wurden umfangreiche Beispiele aus der ISCAS'89-Benchmark Suite [BrBK89] verwendet, deren Schaltungsnetzlisten bei Vergleichsmessungen dieser Art innerhalb von Logiksimulationen eingesetzt werden.

5.4.2.1 Compiliertes Modell im Vergleich zu interpretiertem Modell

Durch Konvertierung eines Stateflow-Modells in eine Java-Klasse erhöht sich die Geschwindigkeit der Simulation.

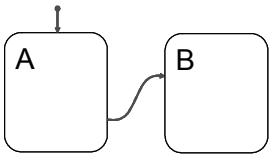


Bild 181: Statechart mit 2 Zuständen

Änderungen des Modellzustandes bei Zustand B des Charts (Modell Nr. 1) in Bild 181 werden weiter vom Simulator abgefragt. Es ergeben sich für das compilierte und das interpretierte einfache Modell Nr. 1 folgende Ergebnisse (Tabelle 11):

Zeitschritte	Compiliertes Modell [ms]	Interpretiertes Modell [ms]
100	735	813
1000	1359	1703
10000	5672	9094

Tabelle 11: Performance-Gewinn bei kleinem Modell

Zur Gewinnung eines komplexeren Modells werden Mehrfachkopien des einfachen Modells als nebenläufige Zustände auf derselben Ebene erzeugt. Das neue, komplexere Modell umfasst 755 Zustände und 766 Transitionen (Tabelle 12).

Zeitschritte	Compiliertes Modell [ms]	Interpretiertes Modell [ms]
100	844	1578
1000	1469	6046
10000	5889	49287

Tabelle 12: Performance-Gewinn bei großem Modell

Die Performance der Simulation wird verglichen, wenn compilierte und interpretierte Modelle verwendet werden. Modell 1 beinhaltet zwei Zustände und eine Transition. Wenn der zweite Zustand während der Simulation erreicht wird, treten keine weiteren Modifikationen ein, aber der Modellzustand wird weiterhin vom Simulator abgefragt, und die Simulation wird nicht beendet. Eine Mehrfachkopie der Zustände von Modell 1 als nebenläufige Zustände auf derselben Ebene resultiert in Modell 3, das 755 Zustände und 766 Transitionen enthält. Ein etwas komplexeres Modell (Modell 2), dessen Eigenschaften von JStateSim unterstützt werden, ist in Bild 182 dargestellt.

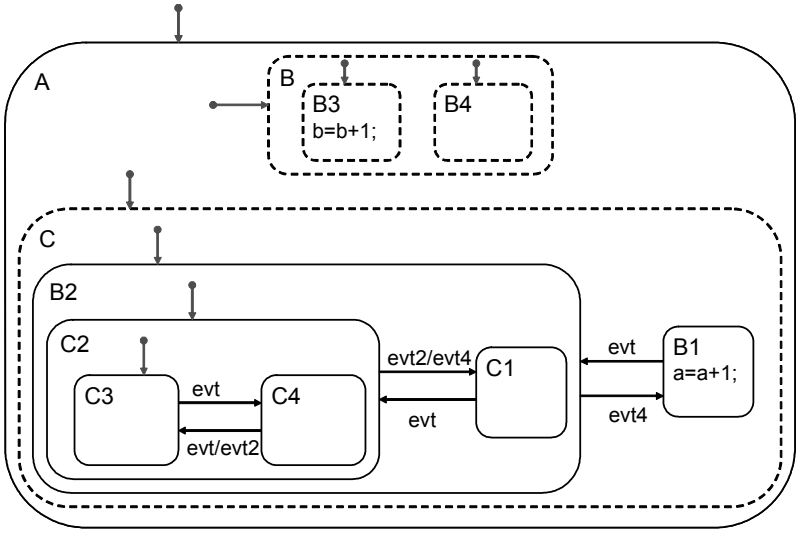


Bild 182: Performance-Gewinn bei sehr großem Modell (Nr. 4)

Eine Mehrfachkopie der Zustände von Modell 2 zu nebenläufigen Zuständen auf derselben Ebene resultiert in Modell 4, das 829 Zustände und 1054 Transitionen enthält. Weiterhin wurde ein reales komplexes Modell eines Fensterhebers untersucht, das bei FZI/ESM entworfen wurde (Modell Nr. 5). Die Messergebnisse sind in Tabelle 13 dargestellt.

Modell	Zeitschritte	Compiliertes Modell [ms]	Interpretiertes Modell [ms]
1	100	735	813
	1000	1359	1703
	10000	5672	9094
3	100	844	1578
	1000	1469	6046
	10000	5889	49287
4	100	766	3235
	1000	1532	23671
	10000	6171	248195
5	10000	5848	9433

Tabelle 13: Messergebnisse

5.4.2.2 Modelle mit einer unterschiedlichen Anzahl von Partitionen

Es wird untersucht, wie sich die Simulationszeit für ein Modell mit unterschiedlicher Anzahl von Partitionen und mit unterschiedlicher Abhängigkeit von den Partitionen verändert. Die folgenden Performance-Parameter werden verglichen[71]:

- *Verteilte Simulationszeit*: Die Teilmodelle nach der Partitionierung werden mit dem gleichen Simulationszeitschritt wie bei Standalone-Simulation verteilt simuliert. Das Teilmodell mit der längsten Simulationsdauer ist als Simulationszeit der verteilten Simulation definiert.
- *Beschleunigung*: Die Simulationszeit wird mit der Standalone-Simulationszeit verglichen.

Die beiden Modelle von Tabelle 14 werden so erzeugt, dass sie die gleichen Beschreibungsmittel benutzen, aber eine unterschiedliche Abhängigkeit zwischen Teilmodellen haben. Die Beschleunigung im Vergleich zur Standalone-Simulation bei unterschiedlicher Anzahl von Partitionen wird verglichen.

Modellname	Anzahl der Partitionen					
	1	2	3	4	5	6
	Simulationszeit [ms]					
s100_p10_h0	422477	236669	129539	126408	126995	91819
s100_p10_h0_9d	421975	165060	135465	113812	142687	147485

Tabelle 14: Testmodelle mit unterschiedlicher Abhängigkeit zw. Teilmodellen

Zwischen den nebenläufigen Zuständen des Modells `s100_p10_h0` gibt es keine Abhängigkeit. Also existieren keine Schnitte zwischen Partitionen dieses Modells. Zwischen den nebenläufigen Zuständen des Modells `s100_p10_h0_9d` gibt es insgesamt neun Abhängigkeiten. Ein nebenläufiger Zustand liest eine Variable eines anderen nebenläufigen Zustands, so dass die Abhängigkeiten zwischen den Partitionen eine Warteschlange bilden können. In Bild 183 und Bild 184 ist jeweils das Modell `s100_p10_h0` und das Modell `s100_p10_h0_9d` nach der Partitionierung in drei Teilmodelle abgebildet.

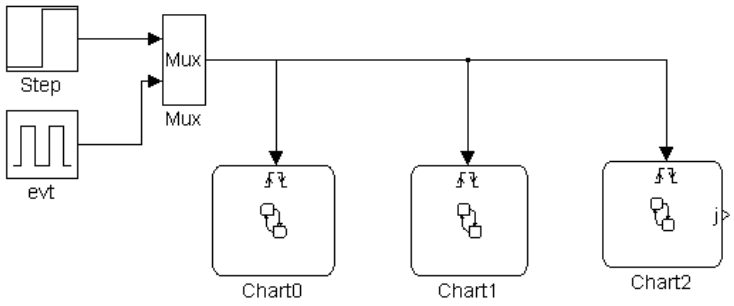


Bild 183: Modell s100_p10_h0 der nach Partitionierung

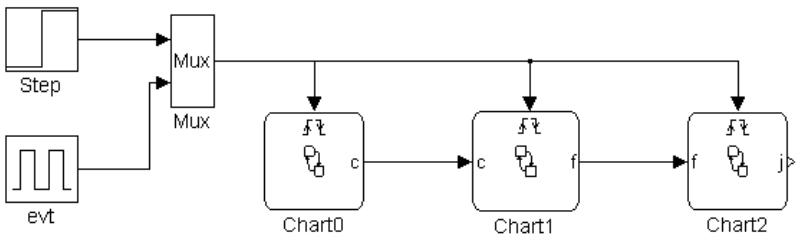


Bild 184: Modell s100_p10_h0_10d nach der Partitionierung

[71] Die Simulation wird unter Windows XP oder Windows 2000 durchgeführt. Alle benutzten Rechner sind durch 100 MBit/s-LAN verbunden und verfügen über eine 2-GHz-P4-CPU und 512 MB RAM.

Bild 185 zeigt die Simulationszeit dieser zwei Modelle bei unterschiedlicher Anzahl von Partitionen. Die Simulationszeit des Modells `s100_p10_h0` nimmt tendenziell mit zunehmender Anzahl von Partitionen ab. Im Vergleich dazu nimmt die Simulationszeit des Modells `s100_p10_h0_9d` ab fünf Partitionen nicht mehr weiter ab. Weil die Teilmodelle aufeinander warten müssen, wird die gesamte Simulationszeit bei langer Warteschlange verzögert.

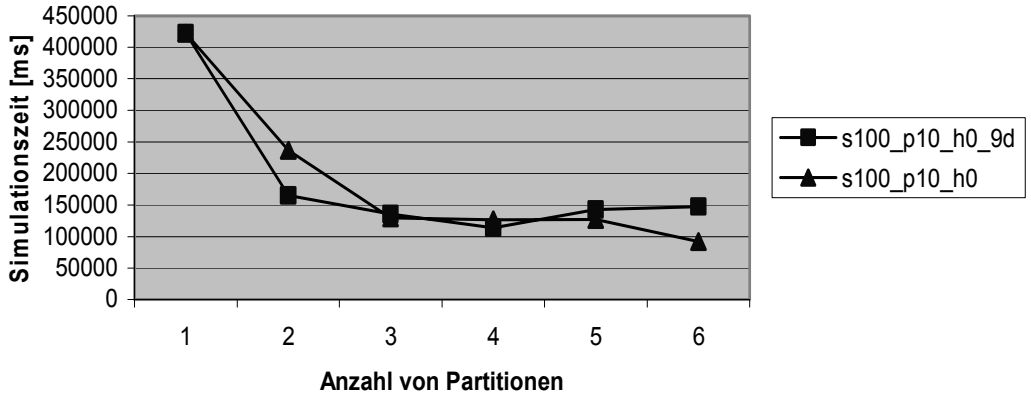


Bild 185: Simulationszeit der Modelle von Tabelle 14

Modellname	Anzahl der Partitionen					
	1	2	3	4	5	6
	Simulationszeit (ms)					
s100_p10_h0_9d	1	2,55	3,12	3,70	2,96	2,86
s100_p10_h0	1	1,78	3,26	3,34	3,32	4,60

Tabelle 15: Beschleunigung mit unterschiedlicher Anzahl von Partitionen

5.4.2.3 Modelle mit unterschiedlicher Größe

Um die Veränderung der Simulationszeit und des Speicherbedarfs von Modellen, deren Größe zugenommen hat, zu bestimmen, werden große Modelle standalone simuliert. Die folgenden Performance-Parameter werden verglichen:

- Standalone Simulationszeit: die Simulationsdauer eines Modells bei der Standalone-Simulation in bestimmten Simulationszeitschritten,
- Speicherbedarf: der Speicherbedarf eines Modells.

Für den Performance-Vergleich werden Charts mit 5.000 bis 300.000 Zuständen verwendet. Es werden 1000 Simulationsschritte ausgeführt. In Tabelle 16 werden die Simulationszeit, die Größe der Auslagerungsdatei sowie die Größe des verfügbaren Speichers aufgelistet[72].

Modellname	Simulations-zeit [ms]	Auslagerungs-datei [MB]	Verfügbarer Speicher [MB]
s5000_p10_h0	56840	191	274
s10000_p20_h0	90171	193	272
s20000_p40_h0	196426	200	268
s50000_p100_h0	694748	220	255
s100000_p200_h0	2429497	251	231
s200000_p400_h0	8945626	314	189
s300000_p600_h0	24255537	318	143

Tabelle 16: Testmodelle mit unterschiedlicher Größe

[72] Die letzten beiden Parameter werden im Task-Manager von Windows XP abgelesen.

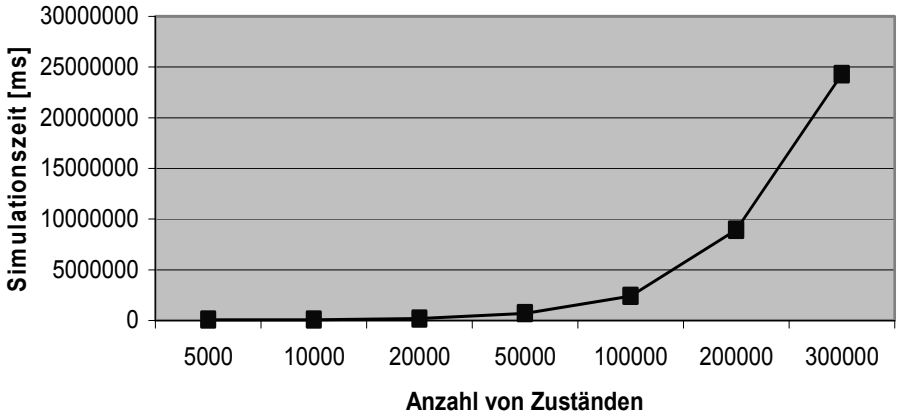


Bild 186: Simulationszeit für große Modelle

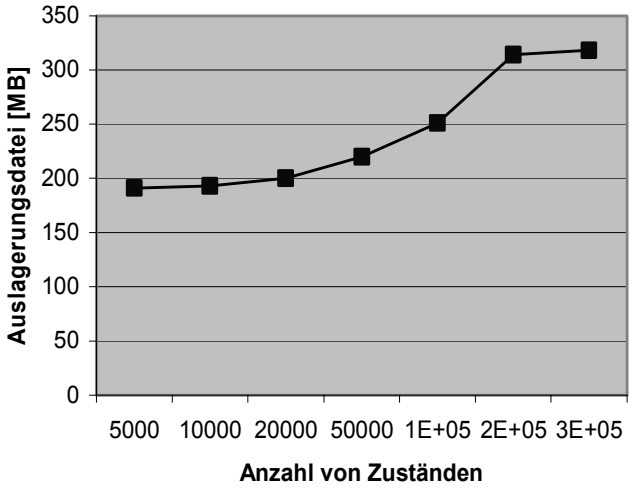


Bild 187: Auslagerungsdatei der Modelle von Tabelle 16

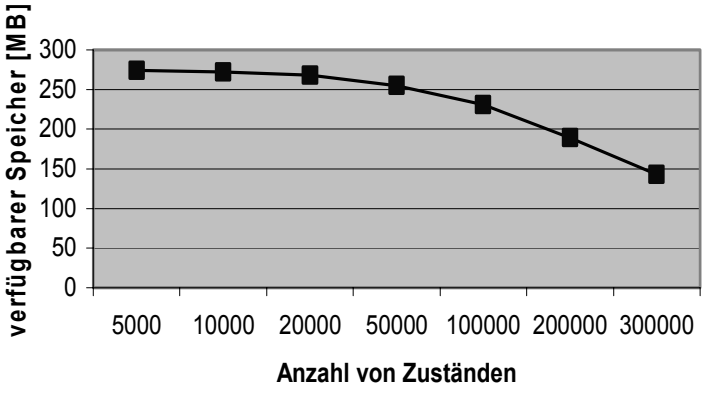


Bild 188: Verfügbarer Speicher der Modelle von Tabelle 16

Bild 186 zeigt die Simulationszeit für große Modelle mit 5.000 bis 300.000 Zuständen. Wenn die Anzahl der Zustände kleiner als 50.000 ist, steigt die Simulationszeit linear. Ab 50.000 Zuständen nimmt die Simulationszeit schneller als die Größe der Modelle zu, da mit zunehmender Anzahl der Zustände die Auslagerungsdatei größer geworden und der Speicherbedarf gestiegen ist. Der Speicherbedarf für jeweils 10.000 Zustände ist etwa 4 MB (Bild 187 und Bild 188).

Diese Ergebnisse zeigen, dass der Einsatz verteilter Simulation ab einer bestimmten Modellgröße eine exponentielle Zunahme der Simulationszeit verhindert. Die kritische Größe beginnt bei der ein-

gesetzten Rechnerkonfiguration und den obigen Modellen bei etwa 5.000 Zuständen. Die Standalone-Simulationszeit nimmt ab diesem Wert nicht mehr linear mit der Modellgröße zu.

5.4.2.4 Kommunikationsaufwand zwischen Teilmodellen

Um den Kommunikationswand zwischen Teilmodellen zu vergleichen, werden verschiedene Modelle von Statecharts nach der Partitionierung verteilt simuliert. Diese Modelle wurden so entworfen, dass sie die gleiche Zustandshierarchie und die gleiche Anzahl sowie die gleichen Arten unterschiedlicher Beschreibungsmittel enthalten und somit den gleichen Rechenaufwand besitzen. Um einen unterschiedlichen Kommunikationsaufwand zwischen den Teilmodellen zu erhalten, variieren die Operationen von Transitionsaktionen so, dass verschiedene Abhängigkeitsgrade von Variablen zwischen nebenläufigen Zuständen existieren. Dadurch können diese Modelle nach der Partitionierung eine unterschiedliche Anzahl von Schnitten und einen unterschiedlichen Kommunikationsaufwand besitzen. Auch die Anzahl von Nullnachrichten bei konservativer Synchronisation wird verglichen.

Für den Kommunikationsaufwand werden die Testmodelle von Tabelle 17 so erzeugt, dass die Anzahl von Schnitten nach der Bipartitionierung von 0 bis 40 skaliert. Alle Modelle werden in 5.000 Simulationsschritten auf zwei Rechnern verteilt simuliert. Weil diese Modelle die gleiche Struktur und die gleiche Anzahl von Transitionsaktionen haben, besitzen sie auch den gleichen Rechenaufwand. Bild 189 zeigt einen Ausschnitt des Charts von `s100_p10_h0_2ta_5d`.

Modellname	Simulationszeit [ms]	Anzahl von Ereignissen	Anzahl von Nullnachrichten
s100_p10_h0_2ta	186084	0	0
s100_p10_h0_2ta_5d	248090	2500	22505
s100_p10_h0_2ta_10d	365225	5000	45010
s100_p10_h0_2ta_20d	696408	10000	90020
s100_p10_h0_2ta_40d	1275300	200000	180040

Tabelle 17: Testmodelle für den Kommunikationsaufwand

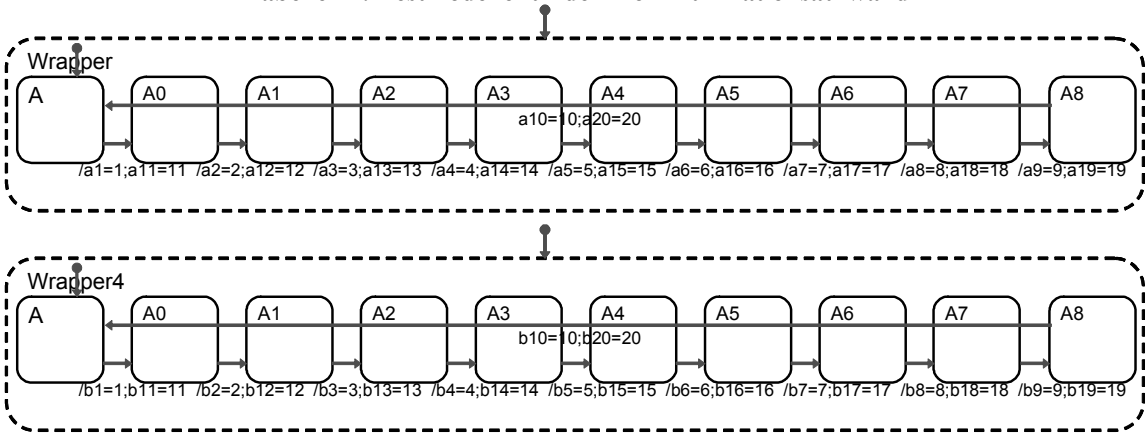


Bild 189: Ausschnitt des Charts von s100_p10_h0_2ta_5d

Die nebenläufigen Zustände `Wrapper` und `Wrapper4` werden nach der Bipartitionierung zwei Charts `Chart0` und `Chart1` zugewiesen. Zwischen beiden Chart-Blocks gibt es fünf Verbindungen (Bild 190).

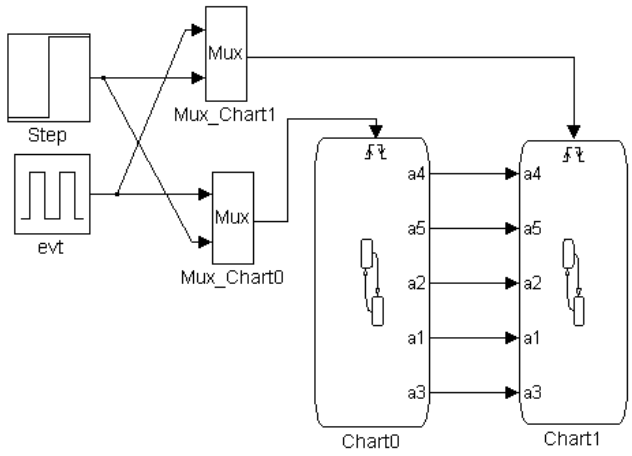


Bild 190: Modell s100_p10_h0_2ta_5d nach Bipartitionierung

Bild 191 zeigt, dass die verteilte Simulationszeit schneller als die Anzahl von Schnitten (Cuts) zunimmt. Bei Standalone-Simulation dauert die Simulation des Modells `s100_p10_h0_2ta` 415781 ms. Die Standalone-Simulationszeit anderer Modelle hat ungefähr den gleichen Wert. Für die Berechnung der Beschleunigung wird dieser Wert auch für andere Modelle benutzt.

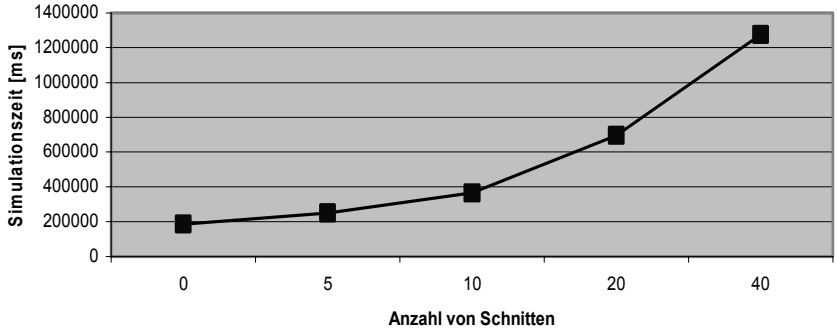


Bild 191: Simulationszeit der Modelle von Tabelle 12

Bild 192 zeigt die Beschleunigung bei verschiedener Anzahl von Schnitten. Wenn keine Schnitte zwischen zwei Partitionen existieren, dann ist die verteilte Simulation mehr als doppelt so schnell wie die Standalone-Simulation. Bei 10 Schnitten hat sich die verteilte Simulation noch beschleunigt. Weil jeder nebenläufige Zustand 10 Transitionen enthält, enthält das Chart mit 10 nebenläufigen Zuständen insgesamt 100 Transitionen. Diese Anzahl von Schnitten entspricht 10 Prozent der gesamten Anzahl von Transitionen. Ab 20 Schnitten hat die verteilte Simulation keine Beschleunigung mehr.

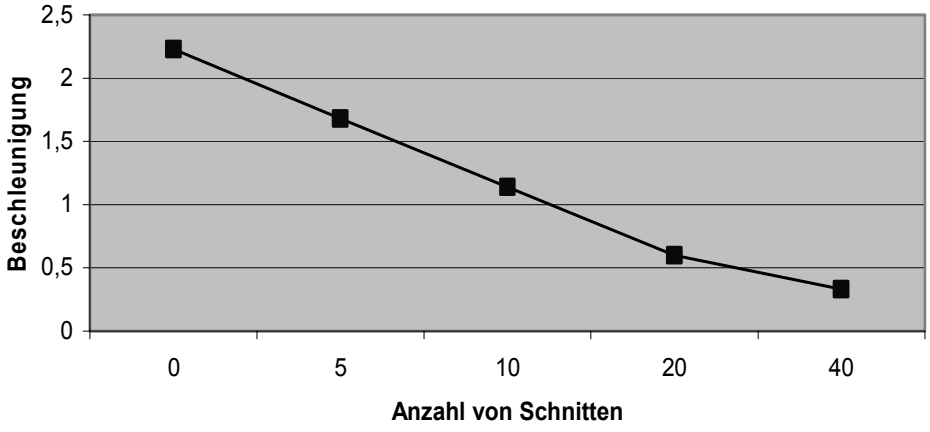


Bild 192: Beschleunigung nach Bipartitionierung der Modelle von Tabelle 12

Aus diesen Ergebnissen kann man ersehen, dass die Performance verteilter Simulation deutlich vom Kommunikationsaufwand beeinträchtigt wird, wenn die Anzahl von Kommunikationskanälen eine bestimmte Anzahl überschreitet. Beim Modell `s100_p10_h0_2ta_10d` ist die Anzahl von Partitionen 10 Prozent der gesamten Anzahl von Transitionen. Um eine Beschleunigung durch verteilte Simulation zu erhalten, muss das Partitionierungsverfahren geändert werden. Zwei nebenläufige Zustände müssen in eine Partition zusammengefasst werden, wenn die Anzahl von Schnitten zwischen ihnen 10 Prozent der gesamten Anzahl von Transitionen überschreitet.

In Bild 193 ist auch die Anzahl von Ereignissen und Nullnachrichten, die zwischen zwei Partitionen gesendet werden, dargestellt. Auf jedem Kanal werden 500 Ereignisse in 5000 Simulationszeitschritten gesendet. Zusätzlich werden 4501 Nullnachrichten auf jedem Kanal gesendet. In jedem Simulationsschritt wird entweder ein Ereignis oder eine Nullnachricht gesendet. Der Grund ist, dass eine Variable im Abstand von 10 Simulationsschritten eine Wertveränderung hat (siehe Bild 189). Wenn diese Variable in einem Simulationsschritt geändert wird, dann wird ein Ereignis zum Zielhost gesendet, sonst wird eine Nullnachricht gesendet. Deshalb werden viel mehr Nullnachrichten als Ereignisse bei diesen Modellen gesendet.

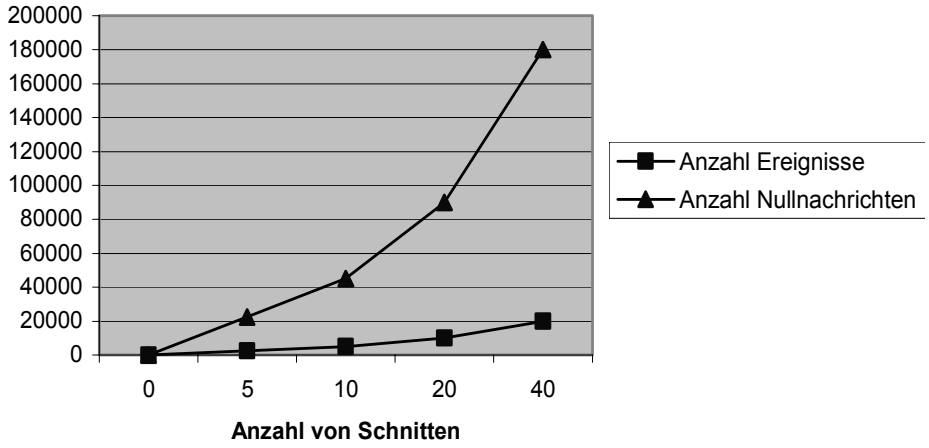


Bild 193: Anzahl von Ereignissen zwischen zwei Partitionen

5.4.2.5 Rechenaufwand unterschiedlicher Beschreibungsmittel

Um den Rechenaufwand von Statecharts festzustellen, wird in diesem Kapitel durch Simulation untersucht, wie die verschiedenen Beschreibungsmittel Zustandsaktionen, Transitionsaktionen und Transitionsbedingungen die Simulationszeit beeinflussen. Durch Vergleich der Simulationszeit wird der relative Rechenaufwand unterschiedlicher Beschreibungsmittel ermittelt.

Für die Messung des Rechenaufwands werden Modelle mit 100 Zuständen standalone simuliert. Jedes Modell wird in 5000 Simulationsschritten simuliert. In den folgenden Tabellen ist die relative Zeit das Verhältnis der Simulationszeit eines Modells zu der Simulationszeit des Modells `s100_p10_h0`.

Rechenaufwand von Transitionsbedingungen: Durch die Modelle in Tabelle 18 wird der Rechenaufwand von Transitionsbedingungen gemessen.

Modellname	Simulationszeit [ms]	relative Zeit
s100_p10_h0	223466	1
s100_p10_h0_c	265169	1,19

Tabelle 18: Testmodelle für den Rechenaufwand von Transitionsbedingungen

Die relative Simulationszeit des Modells mit Transitionsbedingungen ist 265169/223466 ≈ 1,19. Wenn der Rechenaufwand für Transitionen ohne Beschriftung als 100 definiert ist, dann ist der Rechenaufwand für Transitionsbedingungen:

100·0,19 = 19.

Rechenaufwand von Transitionsaktionen: Die Modelle von Tabelle 19 enthalten eine unterschiedliche Anzahl von Transitionsaktionen in einer Transition. Dadurch wird der Rechenaufwand von Transitionsaktionen gemessen.

Modellname	Simulationszeit [ms]	relative Zeit
s100_p10_h0	223466	1
s100_p10_h0_1ta	404463	1,81
s100_p10_h0_2ta	555487	2,49
s100_p10_h0_3ta	702020	3,14

Tabelle 19: Testmodelle für den Rechenaufwand von Transitionsaktionen

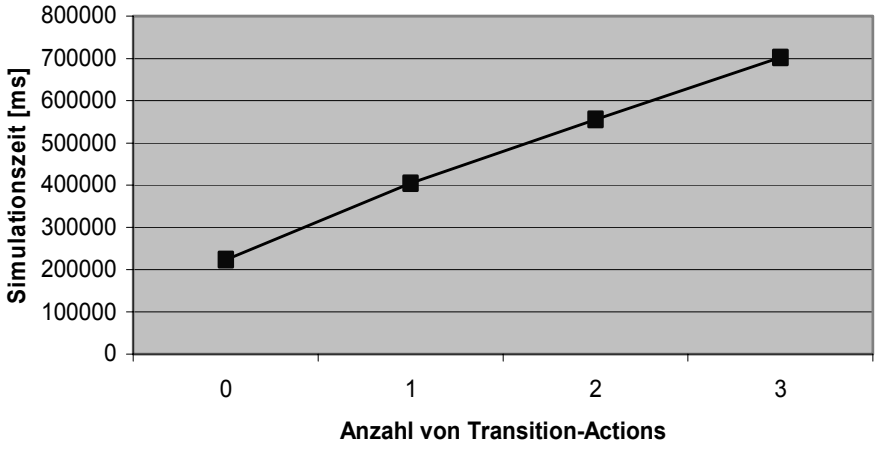


Bild 194: Simulationszeit der Modelle aus Tabelle 13

Bild 194 zeigt, dass die Simulationszeit mit zunehmender Anzahl von Transitionsaktionen ungefähr linear zunimmt. Wenn der Rechenaufwand für Transitionen ohne Beschriftung als 100 definiert ist, dann ist der durchschnittliche Rechenaufwand für eine Transitionsaktion:

100·(0,81 + 1,49 / 2 + 2,14 / 3) / 3 ≈ 76

Rechenaufwand von Zustandsaktionen: Die Modelle von Tabelle 20 enthalten eine unterschiedliche Anzahl von Zustandsaktionen in einem Zustand. Dadurch wird der Rechenaufwand von Zustandsaktionen gemessen.

Modellname	Simulationszeit [ms]	relative Zeit
s100_p10_h0	223466	1
s100_p10_h0_1sa	246468	1,10
s100_p10_h0_2sa	263457	1,18
s100_p10_h0_3sa	271436	1,21

Tabelle 20: Testmodelle für den Rechenaufwand von Zustandsaktionen

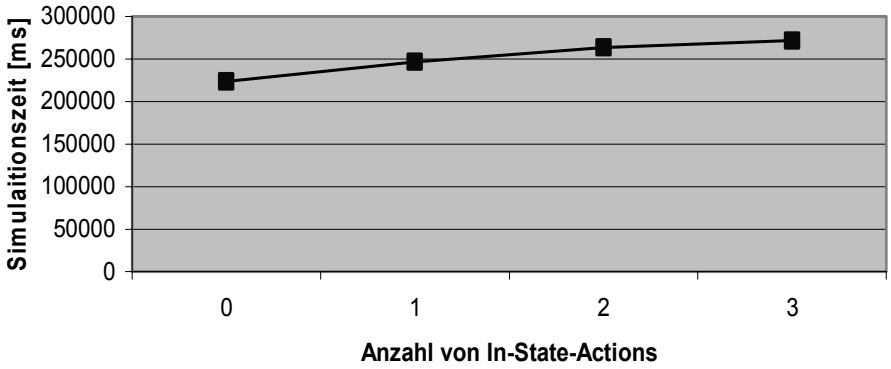


Bild 195: Simulationszeit der Modelle aus Tabelle 14

Aus Bild 195 kann man ersehen, dass die Simulationszeit nicht so deutlich von der Anzahl von Zustandsaktionen wie von Transitionsaktion beeinflusst wird. Wenn der Rechenaufwand für Transition zwischen Zuständen ohne Zustandsaktionen als 100 definiert ist, dann ist der durchschnittliche Rechenaufwand für eine Zustandsaktion:

$100 \cdot (0{,}10 + 0{,}18 / 2 + 0{,}21 / 3) / 3 \approx 8$

Rechenaufwand von Zustandshierarchie: Durch die Modelle von Tabelle 21 wird der Rechenaufwand von Modellen mit unterschiedlicher Hierarchiestufe gemessen. Die Zustandshierarchie skaliert von 0 bis 10. In dieser Arbeit werden nur Charts mit Transitionen zwischen Zuständen auf der gleichen Hierarchiestufe getestet. Je höher die Hierarchiestufe eines Charts ist, desto mehr Transitionen werden in einem Simulationsschritt ausgeführt, um einen Zielzustand zu erreichen. Deshalb nimmt die Simulationszeit mit zunehmender Hierarchiestufe auch zu.

Modellname	Simulationszeit [ms]	relative Zeit
s100_p10_h0	223466	1
s100_p10_h2	405624	1,82
s100_p10_h4	559852	2,51
s100_p10_h6	699964	3,13
s100_p10_h8	773818	3,46
s100_p10_h10	1027730	4,60

Tabelle 21: Testmodelle mit unterschiedlicher Zustandshierarchie

Bild 196 zeigt, dass die Simulationszeit mit zunehmender Hierarchiestufe auch ungefähr linear zunimmt. Wenn der Rechenaufwand für Transitionen ohne Beschriftung als 100 definiert ist, dann ist der durchschnittliche Rechenaufwand für eine Hierarchiestufe:

$100 \cdot (0{,}82/2 + 1{,}51/4 + 2{,}13/6 + 2{,}46/8 + 3{,}60/10) / 5 \approx 36$

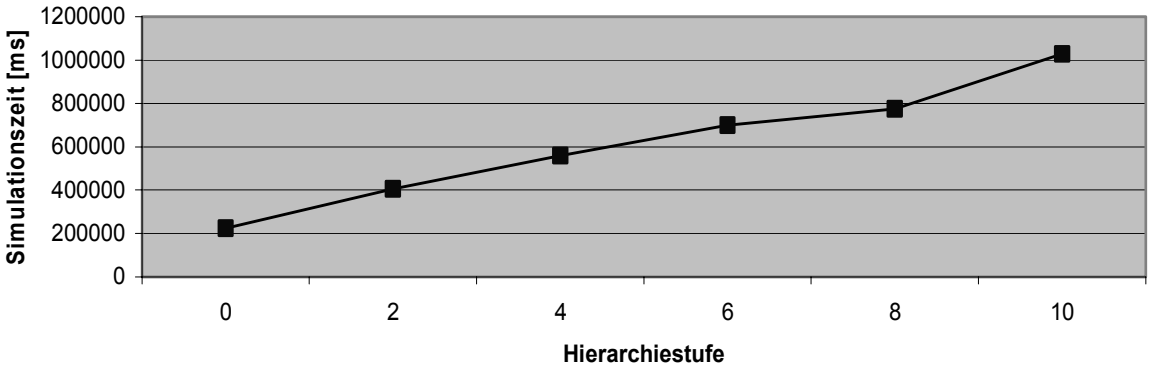


Bild 196: Simulationszeit der Modelle von Tabelle 15

Rechenaufwand von Nebenläufigkeit: Die Modelle von Tabelle 22 enthalten die gleiche Anzahl von Zuständen, aber unterschiedliche Nebenläufigkeit. Je höher die Nebenläufigkeit eines Charts ist, desto mehr Transitionen können in der Simulationszeit ausgeführt werden. Deshalb nimmt die Simulationszeit mit zunehmender Nebenläufigkeit auch zu.

Modellname	Simulationszeit [ms]	relative Zeit
s100_p1_h0	74337	1
s100_p2_h0	94383	1,27
s100_p4_h0	130522	1,75
s100_p5_h0	153958	2,07
s100_p10_h0	223466	3,00

Tabelle 22: Testmodelle mit unterschiedlicher Nebenläufigkeit

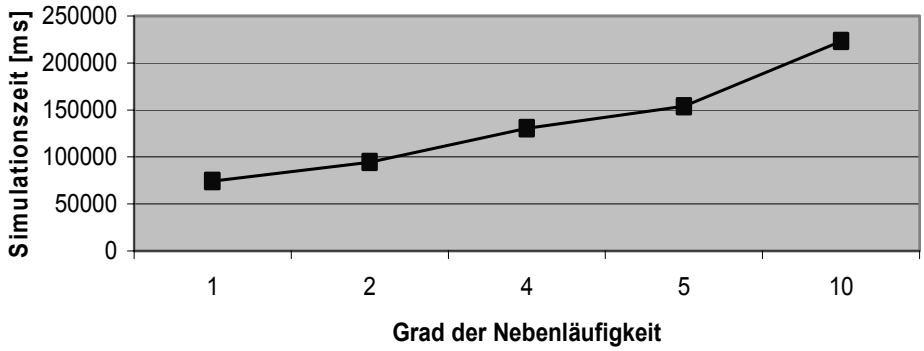


Bild 197: Simulationszeit der Modelle von Tabelle 16

Bild 197 zeigt, dass die Simulationszeit mit zunehmender Nebenläufigkeit auch ungefähr linear zunimmt. Wenn der Rechenaufwand für Transitionen ohne Beschriftung als 100 definiert ist, dann ist der durchschnittliche Rechenaufwand für eine Nebenläufigkeit:

$100 \cdot (0{,}27/2 + 0{,}75/4 + 1{,}07/5 + 2{,}00/10) / 4 \approx 18$

5.4.3 Synchronisationsverfahren

Eine Voraussetzung zur Durchführung einer optimistischen Simulation ist die Fähigkeit der beteiligten Simulatoren zur Speicherung kompletter Systemzustände und der Rücksetzung auf einen bestimmten Zeitpunkt. Z. B. lassen sich bei MATLAB die Systemvariablen und die aktuelle Zustandsposition zwar bei der Simulation von $t = t_0$ bis $t = t_2$ mit Hilfe eines Skriptes in einer Matrix zeitabhängig festhalten, jedoch ergibt eine erneute Simulation ab dem Zeitpunkt $t = t_1$, $t_0 < t_1 < t_2$, bei variabler Schrittweite, die für eine asynchrone Ausführung notwendig ist, verschiedene sowie auch nicht reproduzierbare Ergebnisse, abhängig vom Wiedereinsprungpunkt.

5.4.3.1 Optimistischer und konservativer Algorithmus

Es werden der Rechenaufwand von Statecharts sowie der Einfluss des Kommunikationsaufwands bei der optimistischen Synchronisation gemessen. Für die Messung des Rechenaufwands von Statecharts werden dieselben Testmodelle wie in Abschnitt 5.4.2.5 verwendet. Der Rechenaufwand verschiedener Beschreibungsmittel ist in Tabelle 23 dargestellt.

Beschreibungsmittel	Rechenaufwand
Bedingung	4
Zustandsaktion	2
Transitionsaktion	37

Tabelle 23: Rechenaufwand von Statecharts

Um den Einfluss des Kommunikationsaufwands festzulegen, werden die Testmodelle von Abschnitt 5.4.2.4 nach der Bipartitionierung simuliert. Ab fünf Schnitten hat die verteilte Simulation wegen der Rücksetzung keinen Geschwindigkeitsvorteil mehr (Tabelle 24).

Modellname	Simulationszeit [ms]	Anzahl von Rücksetzungen	Beschleunigung
s100_p10_h0_2ta	271006	0	1,63
s100_p10_h0_2ta_5d	549734	1	0,8
s100_p10_h0_2ta_10d	591902	1	0,74
s100_p10_h0_2ta_20d	621113	1	0,71
s100_p10_h0_2ta_40d	992668	2	0,45

Tabelle 24: Ergebnisse für Kommunikationsaufwand

Es werden zwei Modelle verglichen, die unterschiedliche Abhängigkeiten zwischen den Teilmodellen haben (Tabelle 25). Nach der Bipartitionierung des Modells `s100_p10_h0_2ta_5d` existieren fünf Schnitte zwischen den Teilmodellen. Für das Modell `s100_p10_h0_9d` ist die Anzahl der Schnitte zwischen den Teilmodellen eins.

Modellname	Beschleunigung	Anzahl von Rücksetzungen
s100_p10_h0_2ta_5d	0,8	1
s100_p10_h0_9d	0,57	1

Tabelle 25: Modelle mit unterschiedlichen Abhängigkeiten zwischen nebenläufigen Zuständen

Weil die Abhängigkeiten zwischen nebenläufigen Zuständen des Modells `s100_p10_h0_9d` eine Schlange bilden, kann dies zu einem kaskadierenden Zurücksetzen führen. Dies beeinträchtigt die Performance der verteilten Simulation bei der Verwendung der optimistischen Synchronisation.

Man erkennt, dass die gesamte Simulationszeit um die Hälfte gesenkt wurde (Bild 198). Das optimistische Verfahren (OPT) zeigt bei der Simulation auf drei und sechs Rechnern eine bessere Performance als das konservative (KON). Die Simulation mit dem optimistischen Verfahren auf zwei Rechnern zeigt aufgrund der Zustandsspeicherung eine schlechtere Performance als die konservative Simulation auf zwei Rechnern. Beim größeren Modell braucht der Simulator viel Zeit, um die Zustandsspeicherung durchzuführen, genauso beim Löschen der nicht mehr gebrauchten Daten nach Erhalt der GVT. Bild 199 zeigt, wie die Performance nicht linear steigt, da die Anzahl der gesendeten Nachrichten mit steigender Anzahl der beteiligten Rechner steigt.

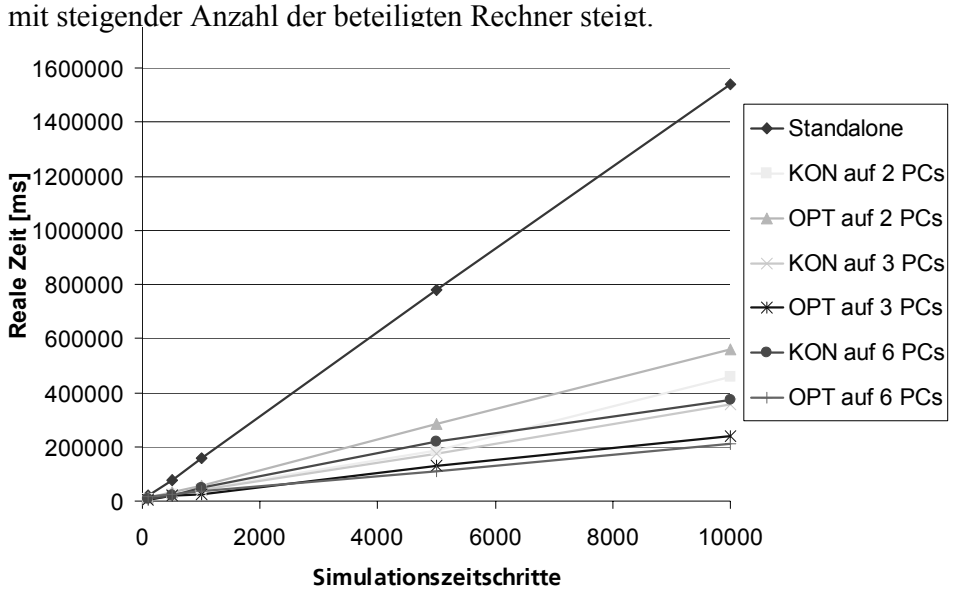


Bild 198: Reale Zeit als Funktion der Simulationszeitschritte

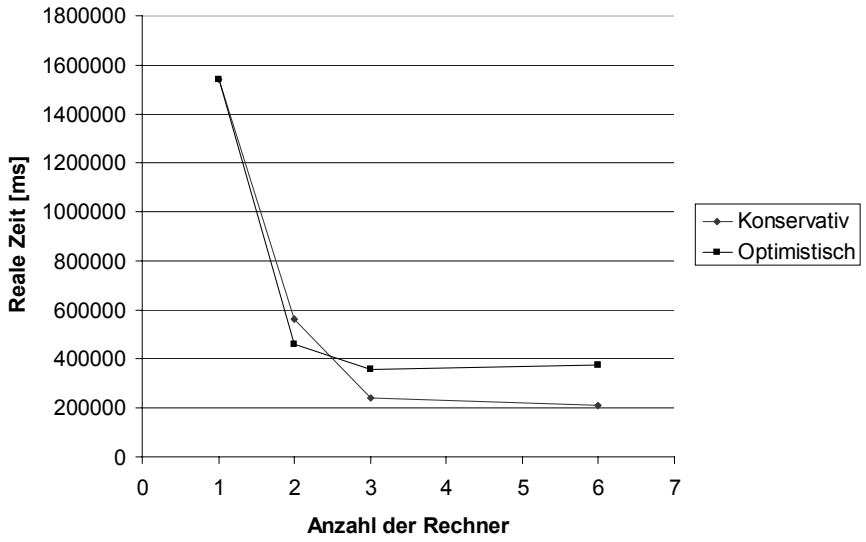


Bild 199: Reale Zeit als Funktion der Rechneranzahl

Da das Modell in mehrere gleichverteilte Teilmodelle (gleiche Anzahl von Zuständen) partitioniert wurde, werden in Bild 200 die Speichermessungsergebnisse eines Rechners angegeben. Aus Bild 200 erkennt man, dass am Anfang der Simulation mehr Speicher gebraucht wird. Die JVM konvertiert während der Ausführung alle Klassen in Maschinencode, so dass die Ausführung des Programms beschleunigt wird. Deshalb braucht das Programm viel Speicher am Anfang. Man erkennt auch, dass bei optimistischer Simulation auf drei Rechnern mehr Speicher benötigt wird und die Simulation schneller als auf zwei Rechnern ist. Es wurden mehr Zustände gespeichert als bei der Simulation auf zwei Rechnern, was zu mehr Speicherbedarf führt. Das ist allerdings nicht der Fall für die Simulation auf sechs Rechnern. Der Grund ist, dass die Simulation auf sechs Rechnern nicht viel schneller läuft als die Simulation auf drei Rechnern. Deshalb ist die Anzahl der durchgeführten Zustandsspeicherungen ungefähr gleich, aber die Anzahl der Zustände nicht gleich. Deshalb benötigt die Simulation auf sechs Rechnern weniger Speicher als die auf drei.

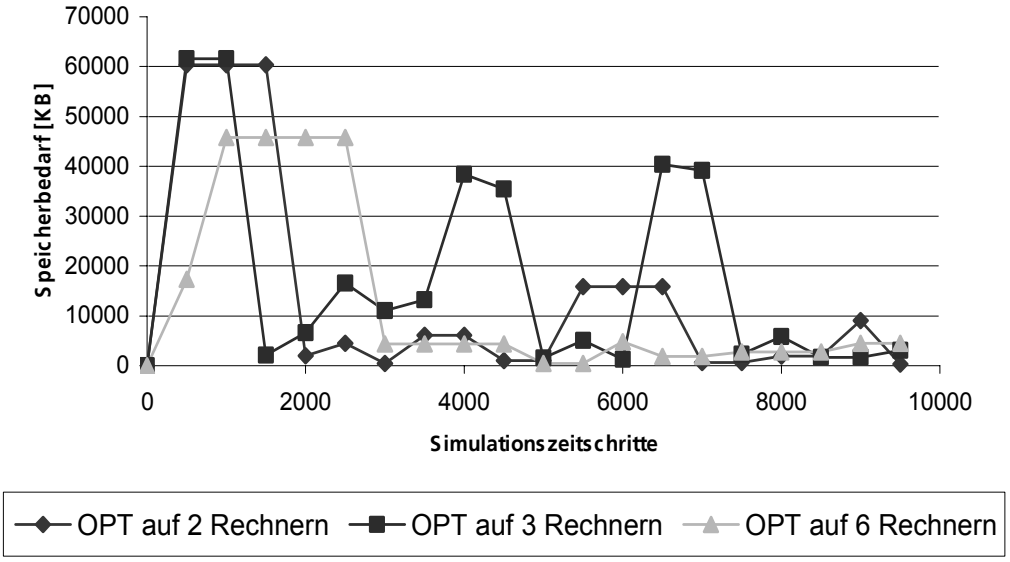


Bild 200: Speicherbedarf für das optimistische Verfahren

5.4.4 Verteilte Simulation hybrider Modelle

5.4.4.1 Performance-Modelle

Für die durchgeführten Worst- und Best-Case-Messungen wurden die in Bild 201 dargestellten Modelle verwendet. Diese sind von minimaler Komplexität; der Simulationsfortschritt konnte daher gut nachvollzogen und verifiziert werden, um korrekte Messergebnisse zu gewährleisten.

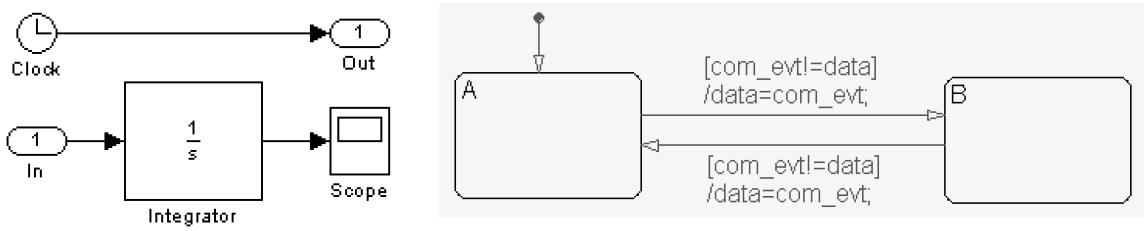


Bild 201: Simulink- und Stateflow-Modell zur Performance-Messung

Simuliert wurden 250 gültige Simulationsschritte, wobei die Schritte als gültig gelten, wenn die GVT die Simulationszeit 250 überschritten hat.

5.4.4.2 Worst-Case-Betrachtung

Zunächst wurden zwei Messungen mit unterschiedlicher Hardware-Konfiguration bei Worst-Case-Betrachtung durchgeführt. Durch eine bewusst unpassend gewählte Hypothese (lineare Funktion, Steigung 1, Offset 1, Toleranz 0,25) wurde der Eingang des DES immer mit falschen Hypothesewerten belegt. Somit wird jeder Hypothesewert durch einen Korrekturwert des CS annulliert und der jeweilige Simulationsschritt des DES muss nach einer Rücksetzung mit dem Korrekturwert erneut ausgeführt werden.

Bei der ersten Messung wurde JStateSim als DES auf einer Intel Pentium-II-CPU mit einer Taktfrequenz von 350 MHz (Workstation 1, WS 1), RTW als CS auf einer AMD Athlon-CPU mit einer Taktfrequenz von 1 GHz (Workstation 2, WS 2) ausgeführt. Dabei wurde die Triggerung des DES variiert: im ersten Fall wurde zu jeder Simulationssekunde getriggert (T 1), im zweiten Fall wurde JStateSim nur nach jeder zehnten Simulationssekunde (T 10) durch ein Ereignis getriggert. Um die Auswirkung einer verzögerten Aktivierung des CS auf die Simulationsgeschwindigkeit zu prüfen, wurde unterschiedlich lang mit der Aktivierung gewartet: aktiviert wurde direkt nach Triggerung des DES (TW 1), zehn Simulationssekunden nach DES-Triggerung (TW 10) und 50 Simulationssekunden nach DES-Triggerung (TW 50). Die verzögerte Aktivierung (TW 10 und TW 50) entspricht nicht den Synchronisationsregeln der SES, allerdings sollte mit der Messung getestet werden, ob eine entsprechende Variation Vorteile bringt.

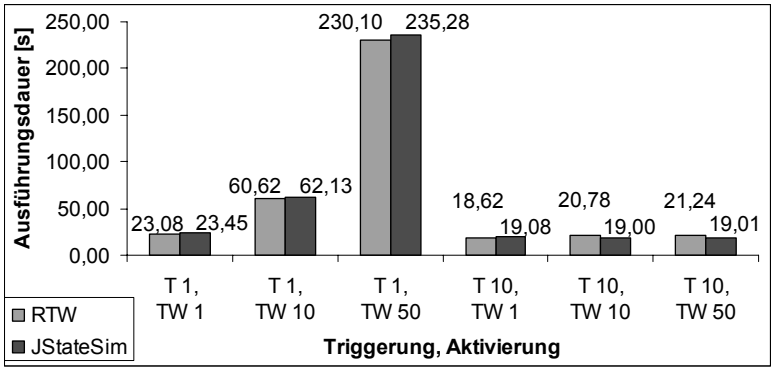


Bild 202: Worst-Case mit JStateSim auf WS 1

Bild 202 zeigt deutlich die jeweils ähnlichen Ausführungszeiten für RTW und JStateSim. Dies lässt sich durch die starke Bindung von CS und DES beim Simulationsfortschritt erklären: der CS sendet für jeden Simulationsschritt einen Korrekturwert und verhindert so einen unabhängigen, parallelen Simulationsfortschritt.

Bezüglich der Triggerrate gibt es nur geringe Unterschiede bei T 1 und T 10 mit jeweiliger Aktivierungsrate TW 1. Dies liegt daran, dass auch bei der niedrigeren Triggerrate T 10 für jeden Simulationsschritt des CS ein Korrekturwert an den DES gesendet wird, der dort bei jeder Simulationssekunde zu einer Rücksetzung führt.

Den größten Einfluss auf die Simulationsdauer hat die Variation der CS-Aktivierungsrate. Eine frühe Aktivierung führt zu frühere Erkennung der falschen Hypothese. Spätere Aktivierung bedeutet, dass der DES zunächst optimistisch mit den – in diesem Fall falschen – Hypothesewerten simuliert, ehe die Ungültigkeit der Hypothese festgestellt werden kann. Somit führt der DES viele unnötige, weil falsche Simulationsschritte aus, die die Simulation deutlich verlangsamen. So ist bei einer Triggerrate T 1 eine Verlangsamung um den Faktor 10 festzustellen, wenn die Aktivierungsrate von TW 1 auf TW 50 gesenkt wird.

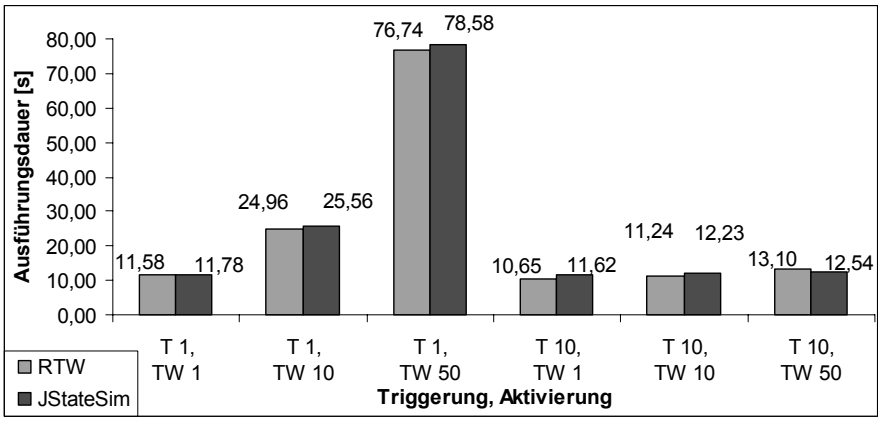


Bild 203: Worst-Case mit RTW auf WS 1

Die Ergebnisse der zweiten Worst-Case-Messung sind in Bild 203 dargestellt. Hier wurde JStateSim als DES auf WS 2 und RTW als CS auf WS 1 ausgeführt. Die prinzipiellen Zusammenhänge bezüglich Triggerrate und Aktivierungsrate gelten auch hier. Allerdings konnte im Vergleich zur ersten Messung die Dauer der Ausführung bis um den Faktor 3 gesenkt werden. Dies lässt erkennen, dass JStateSim für die Simulation seines Modellteils höhere Performance-Anforderungen hat als RTW für seinen Modellteil. Es zeigt sich, dass eine günstige Zuordnung der Simulatoren auf die vorhandenen Hardware-Komponenten deutliche Performance-Gewinne bewirken kann.

5.4.4.3 Best-Case-Betrachtung

Die nächsten beiden Messungen sind Best-Case-Betrachtungen, bei denen die Hypothesefunktion ideal an den tatsächlichen Signalverlauf angepasst wurde (lineare Funktion, Steigung 1, Offset 0, Toleranz 0,25). Dadurch kann der DES komplett auf die Hypothesewerte zurückgreifen, was eine deutliche Beschleunigung erwarten lässt.

Bei der ersten Messung wurde JStateSim als DES wiederum auf Workstation (WS) 1, RTW als CS auf WS 2 ausgeführt. Bild 204 gibt die erzielten Resultate wieder. Im Gegensatz zu den Diagrammen der Worst-Case-Betrachtung sind sie jeweiligen Dauer der Ausführungen von CS und DES zum Teil stark abweichend. Dies zeigt, dass CS und DES im besten Fall vollkommen unabhängig parallel simulieren können. Die Dauer der Gesamtsimulation wird von der langsamsten Simulator/HW-Kombination bestimmt.

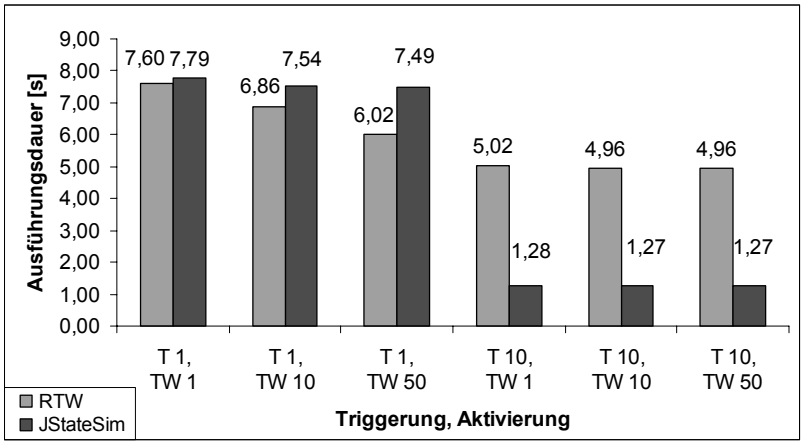


Bild 204: Best-Case mit JStateSim auf WS 1

Wie Bild 204 zeigt, hat die CS-Aktivierungsrate beim Best-Case geringeren Einfluss auf die Simulationsdauer, da der CS die Hypothesewerte lediglich verifizieren, aber nicht korrigieren muss. Durch Senken der Aktivierungsrate reduziert sich der Kommunikationsaufwand, was zu einer geringfügigen Beschleunigung führt. Allerdings kann je nach Simulationsgeschwindigkeit der anderen Simulatoren die Dauer der Gesamtsimulation beeinträchtigt werden: in Bild 204 benötigt der CS bei einer Triggerrate T 10 deutlich mehr Zeit als der DES. In diesem Fall kann eine weitere Verzögerung der CS-Aktivierung zu einem Anstieg der Gesamtsimulationsdauer führen, da verfügbare Rechenzeit, in der der CS auf seine Aktivierung wartet, ungenutzt verstreicht.

Aus der von T 1 auf T 10 gesenkten Triggerrate resultiert insbesondere beim DES eine deutliche Simulationsbeschleunigung: ein plausibles Verhalten, da JStateSim deutlich weniger Schritte simulieren muss. Während der CS bei T 1 wegen des langsameren DES noch auf die Aktivierung warten muss, erreicht RTW bei einer DES-Triggerrate von T 10 seine momentan maximale Ausführungsgeschwindigkeit. In der momentanen Version des optimistischen RTW-Simulations-Frameworks wird zur Thread-Synchronisation und zur Sicherung der Datenkonsistenz an einigen Stellen die `sleep`-Funktion verwendet, die die Simulation abbremst. Durch Wechsel des zur Übertragung der Befehle und Ereignisse verwendeten Protokollformats von der ASCII-Ebene auf binäre Ebene kann Rechenaufwand für das Parsen eingespart werden. Diese beiden Punkte verhindern momentan eine noch schnellere Ausführung des RTW-Modells.

Am besten wurde das Leistungspotential bei der Ausführung von JStateSim als DES auf WS 2 und RTW als CS auf WS 1 ausgenutzt (Bild 205).

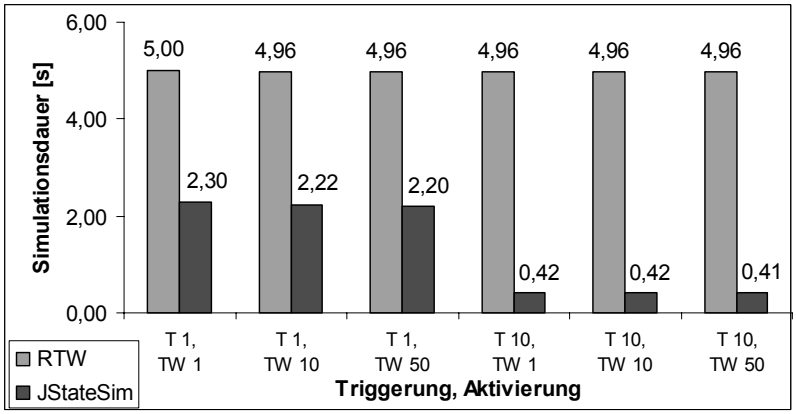


Bild 205: Best-Case mit RTW auf WS 2

Hier wird bereits bei T 1 die maximale Ausführungsgeschwindigkeit des RTW-Modells erreicht, eine weitere Beschleunigung von JStateSim bei T 10 führt im Moment zu keiner weiteren Steigerung der

Gesamtsimulationsdauer. Allerdings zeigt sich hier weiteres Beschleunigungspotential durch Optimierung des RTW-Simulations-Frameworks, das dann am besten ausgenutzt ist, wenn die Ausführungsdauern von CS und DES ähnlich sind, ohne dass der CS auf die Aktivierung vom DES warten muss.

5.4.4.4 Auswertung von Best- und Worst-Case

Bild 206 zeigt, dass der Performance-Gewinn im Best-Case nur gering ausfällt, während die Verlangsamung im Worst-Case sehr stark ausfällt. Daher stellt das Aktivierungskonzept der SES, die wenig Optimierungsspielraum bietet, eine gute Lösung dar; bei einer verzögerten Aktivierung kann keine deutliche Beschleunigung erreicht werden.

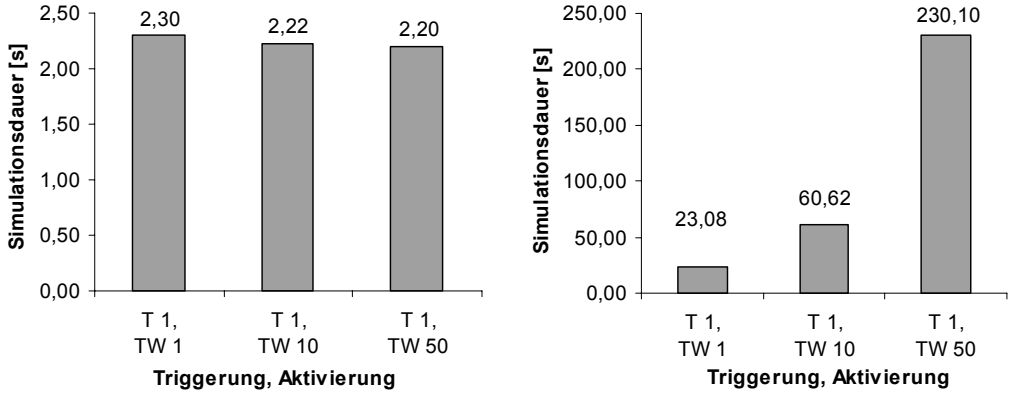


Bild 206: Einfluss der Aktivierungsrate bei Best-Case (links) und Worst-Case (rechts)

Beschleunigungspotential liegt hingegen bei der Hypothesegüte, der Triggerrate und einer den HW-Anforderungen von Modell und Simulator angepassten Prozessorzuteilung. Hier konnte bei JStateSim eine Beschleunigung um den Faktor 57, bei RTW um den Faktor 4,6 erreicht werden (Bild 207), wobei eine optimierte Version des RTW-Frameworks eine vergleichbare Beschleunigung wie JStateSim bringen kann.

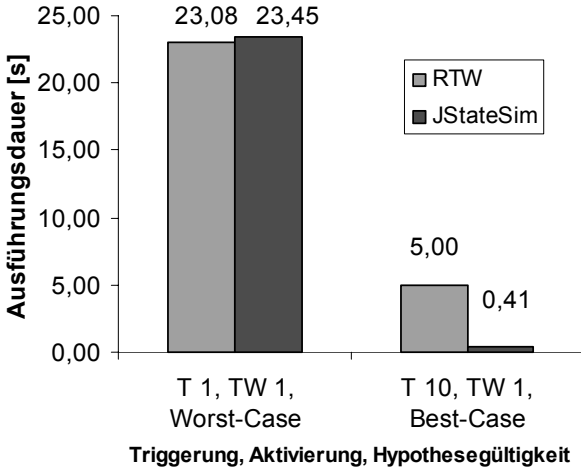


Bild 207: Beschleunigungspotential mit den aktuellen Simulatorversionen

Die Simulation hybrider Systeme kann also durch Verteilung und den Einsatz optimistischer Synchronisationsmethoden deutlich beschleunigt werden. Voraussetzung sind möglichst gut angepasste Hypothesefunktionen mit geringem Performance-Bedarf. Idealerweise sollten die ereignisdiskreten Modellteile eine niedrige Triggerrate aufweisen, den Vorteil einer ereignisgesteuerten Ausführung verglichen mit einer zeitgesteuerten Simulationsweise also nutzen können. Durch eine entsprechende Zuordnung von Simulator und Modell auf Prozessoren mit passender Rechenleistung kann die Gesamtsimulationsdauer weiter reduziert werden.

5.4.4.5 Verteilung auf weitere Rechner

Zwei Konstellationen wurden untersucht, um zu prüfen, welchen Einfluss die Verteilung auf eine größere Anzahl Rechner auf die Simulationsgeschwindigkeit hat. Zunächst wurden die RTW-Teilmodelle von einer JStateSim-Instanz aktiviert (`3rtw_1jss`, Bild 208), wofür ingesamt fünf Rechner zum Einsatz kamen: drei für RTW, einer für JStateSim und einer für JSimControl.

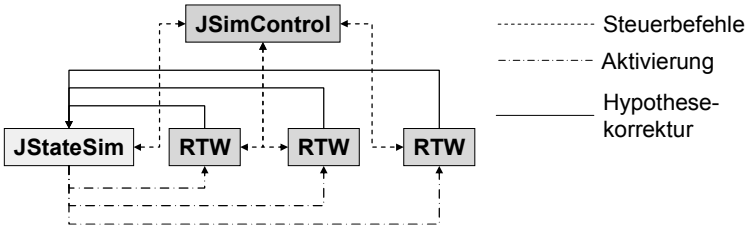


Bild 208: Verteiltes Modell mit 5 eingesetzten Rechnern

Bei der zweiten Verteilungskonstellation `3rtw_3jss` wurden sieben Rechner eingesetzt: einer für JSimControl, drei für RTW und drei für JStateSim, wobei jede JStateSim-Instanz ein RTW-Teilmodell aktiviert (Bild 209).

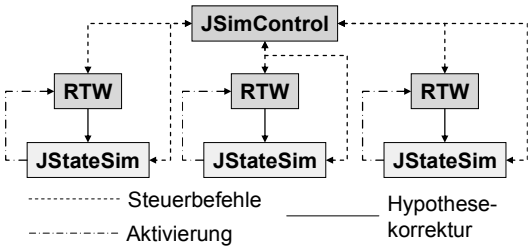


Bild 209: Verteiltes Modell mit 7 eingesetzten Rechnern

Die Ausführungsgeschwindigkeit beider Verteilungen wurde jeweils unter drei Voraussetzungen gemessen: bei der ersten Messung lagen alle Simulationsergebnisse im Toleranzband (Best Case), bei der zweiten Messung war eine Hypothese immer falsch, bei der dritten Messung waren die Hypothesen von zwei RTW immer falsch. Es wurde gemessen, wie weit die GVT innerhalb von 60 s voranschreitet; dies entspricht der Simulationsgeschwindigkeit des langsamsten Simulators innerhalb der verteilten Simulation. Bild 210 gibt die Ergebnisse wieder.

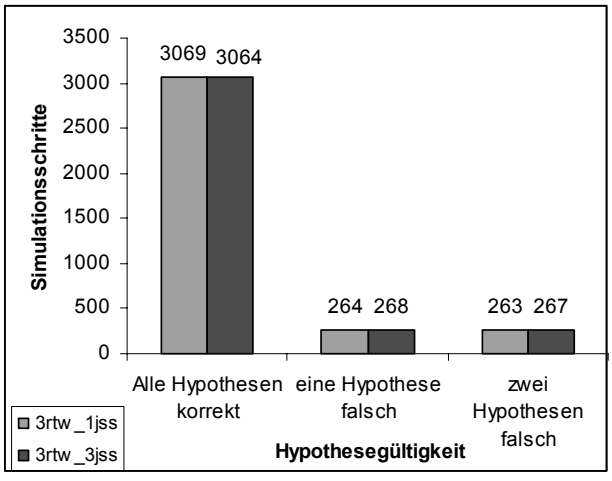


Bild 210: Ausführungsgeschwindigkeit bei 3rtw_1jss und 3rtw_3_jss

Erwartungsgemäß liegt die Ausführungsgeschwindigkeit beim Best-Case um mehr als das zehnfache über den Werten, die bei ungültiger Hypothese erreicht werden. Hierbei ist es unerheblich, wie viele Hypothesen falsch sind: sobald ein RTW Korrekturwerte senden muss, bremst er die Gesamtsimulation relativ zum Best-Case ab. Bei den Messungen hat sich das Ergebnis von 5.4.4.4 bestätigt, wo-

nach der RTW in seiner aktuellen Version langsamer läuft als JStateSim: im besten Fall können die Simulatoren unabhängig voneinander simulieren, hier berechnete JStateSim bis zu 20910 Simulationsschritte innerhalb von 60 s. D. h. JStateSim wurde durch RTW auf ein Drittel seines Leistungspotentials abgebremst. Die Performance-Unterschiede bei den beiden Verteilungskonstellationen sind vernachlässigbar. Leistungseinbrüche durch Netzwerküberlastung traten nicht auf, für ein solches Phänomen müsste RTW schneller simulieren oder es müssten noch mehr Rechner verwendet werden.

5.4.5 Plattformen

5.4.5.1 FPGA, Mikrocontroller, PC

Es wird untersucht wie sich die Simulation paralleler Zustände in Stateflowcharts auf die Ausführungsgeschwindigkeit bei MPC555 und FPGA auswirkt. Um vergleichbare Resultate beim MPC555 und dem FPGA zu erhalten, wurden alle Messungen mit einer konstanten CAN-Busrate und mit einer konstanten Anzahl von Simulationsschritten durchgeführt. Der MPC555 zeigt mit wachsender Zahl paralleler Zustände eine steigende Ausführungsdauer (Bild 211).

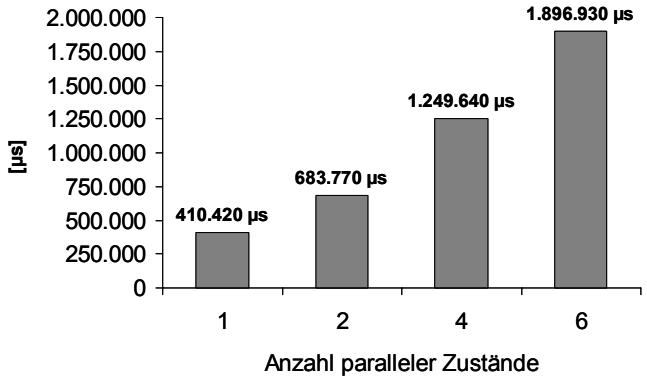


Bild 211: Simulationsdauer, wenn auf einem MPC555 ausgeführt

Da das FPGA entsprechend seiner HW-Eigenschaften Parallelausführung unterstützt, werden alle vier Modellvarianten innerhalb derselben Ausführungsdauer von 17.470 µs berechnet. Somit ist das FPGA bereits bei der Modellvariante mit nur einfach verwendeter Basisfunktionalität bereits um das 23,5-fache schneller als der MPC555. Diese Überlegenheit hinsichtlich der Performance steigert sich auf das 108,5-fache bei 6-facher Parallelität. Bei Stateflowcharts mit vielen parallelen Zuständen sollte also ein FPGA zur Performance-Steigerung verwendet werden. Der Ausgang des kontinuierlichen Teilsystems liefert in der FPGA-Realisierung Ergebnisse mit einer Breite von 18 Bit. Da Simulink/Stateflow aber nur Variablen der Größe 1, 2 und 4 Byte zur Verfügung stellt, muss der 4 Byte große Datentyp genutzt werden; bei Verwendung eines 16-Bit-Datentyps wäre der Speicher für große Werte zu klein. Der Einsatz von 32 Bit großer Datentypen (`UInt32`) führt bei der Synthese für das FPGA zu großen Signalleitungen und entsprechend viel benötigten Logikelementen. Es hat sich gezeigt, dass bei der Abbildung des Stateflowcharts auf das FPGA nach der Synthese und der anschließenden Platzierung und Verdrahtung Optimierungen durchgeführt werden, die zu einer fehlerhaften Simulationsausführung führen können. Die Modellierung muss daher z. B. durch Eliminierung von Konstanten so lange variiert werden, bis der Optimierungsvorgang kein verändertes Systemverhalten mehr verursacht.

Die Signalverzögerungszeiten sind in hohem Maße von der physikalischen Distanz zwischen Logikelementen und der Art der Routing-Ressourcen abhängig, die für die Herstellung der Verbindung genutzt werden. Die Verzögerungen, die man dem Bereich Interconnect (Verbindungen zwischen den einzelnen Metallisierungsebenen) zuordnen kann, dominieren das Gesamt-Timing in den heute üblichen programmierbaren Bausteinen. Nimmt ein kritischer Pfad innerhalb eines Designs z. B. eine

lange Route quer über ein großes FPGA hinweg, dann ist es recht wahrscheinlich, dass dieser Pfad nur geringe Timing-Vorgaben erfüllen kann. Veränderungen der Timing-Vorgaben oder Back-Annotation der Verzögerungszeit nach dem Place&Route-Vorgang können bis zu einem gewissen Grad helfen, allerdings in der Regel nur nach vielen Iterationen. Die physikalische Synthese stellt einen direkten Weg zur Verfügung, um strukturelle Optimierungen durchzuführen, die jenseits dessen liegen, was die reine Logiksynthese tun kann. Darüber hinaus ist die physikalische Synthese in der Lage, zusätzlich die Platzierung von Logik entlang kritischer Pfade eines Designs zu steuern. Obwohl diese Methode hochgradig automatisiert ist, handelt es sich nicht um einen Designfluss auf Knopfdruck, weil der Designer die entsprechenden Leitlinien in Form von physikalischen Constraints als RTL-Floorplan vorgeben muss, um die besten Ergebnisse zu bekommen [Garr01]. Die Stromaufnahme für den kontinuierlichen und den diskreten Teil ist ungefähr gleich. Werden beide zusammen auf einem MPC555 ausgeführt, steigt der Wert gering an, aber die Grundlast des zweiten MPC555 entfällt. Auffällig ist die geringe Stromaufnahme des FPGA, das außerdem mit einer geringeren Versorgungsspannung betrieben wird. Somit ist das FPGA hinsichtlich des Energieverbrauchs um mehr als 70 % effektiver als der MPC555.

Insbesondere bei Charts mit parallelen Zuständen lässt sich die Ausführungsgeschwindigkeit aufgrund der HW-Architektur zur parallelen Verarbeitung auf dem FPGA erheblich steigern. Das FPGA zeigte sich als sehr energiesparend und ist auch wegen seiner hohen Rechenleistung eine gute Alternative zum MPC555. Nachteile sind die begrenzte Anzahl und Datenbreite der Ein- und Ausgänge und der Kostenfaktor. Ein µC stellt hingegen bereits jetzt eine flexible, aber auch deutlich langsamere Lösung dar. Eine Alternative zu einem FPGA, um die Simulation von Stateflowcharts zu beschleunigen, ist der Einsatz eines dedizierten Simulators wie JStateSim, der auch effizientere Synchronisationsprotokolle unterstützt, die eine parallele Modellausführung erlauben, anstatt einer ausschließlich seriellen.

Ein Performance-Vergleich zwischen HW (FPGA) und SW (JStateSim, ausgeführt auf PC) liefert folgende Ergebnisse. Einen ersten Überblick über die erreichte Leistung gibt die Versuchsanordnung in Bild 212. Dabei wird das Modell noch nicht verteilt simuliert, sondern einzeln auf unterschiedlichen Rechnern bearbeitet. Das Modell `Zähler` ist ein Statechart mit 100 Zuständen, die zu einem Ring zusammengeschlossen sind. An jedem Übergang wird eine Zählervariable erhöht und nach einem Durchlauf ausgegeben. Danach wird auf eine Bedingung am Eingang `a` gewartet, bevor der Zyklus neu beginnt. In einem weiteren Versuch wurde die Anzahl der Zustände auf 252 erhöht.

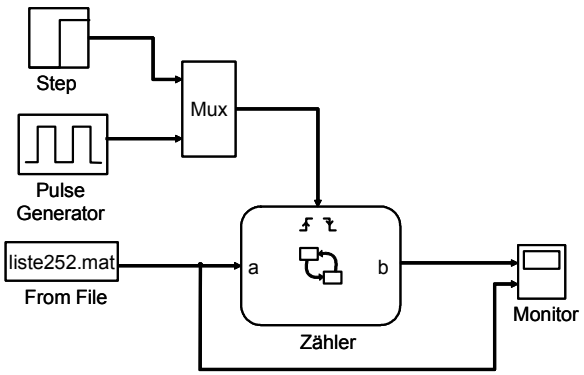


Bild 212: Autonomes Modell

Die Ergebnisse in Bild 213 zeigen eine deutliche Simulationsbeschleunigung von JStateSim mit FPGA-Unterstützung gegenüber den beiden anderen Simulationen. Die Prozessoren der Rechner, auf denen MATLAB und JStateSim laufen, sind dabei mehr als doppelt so schnell (ca. 1,7 GHz) wie der des RP-Systems (0,7 GHz). Dieser Unterschied gewinnt bei zunehmendem Kommunikationsaufkommen und großer Simulationsdauer an Bedeutung. Bei 100 Simulationsschritten liefert das FPGA das Ergebnis praktisch sofort zurück. Dieser Vorteil nimmt mit Zunahme der Simulationsdauer ab, da in diesem Fall die Kommunikation zwischen FPGA und dem Simulator in den Vordergrund tritt.

Sichtbar bleibt er jedoch im doppelt so schnellen Abarbeiten des größeren Modells. Die herkömmliche Simulation benötigt für größere Modelle auch mehr Zeit. Das schlechte Abschneiden von JStateSim ist auf seine Implementierung zurückzuführen, die für die verteilte Simulation optimiert ist. Weitere Untersuchungen zur Simulation mit JStateSim werden in [Bukh03] durchgeführt.

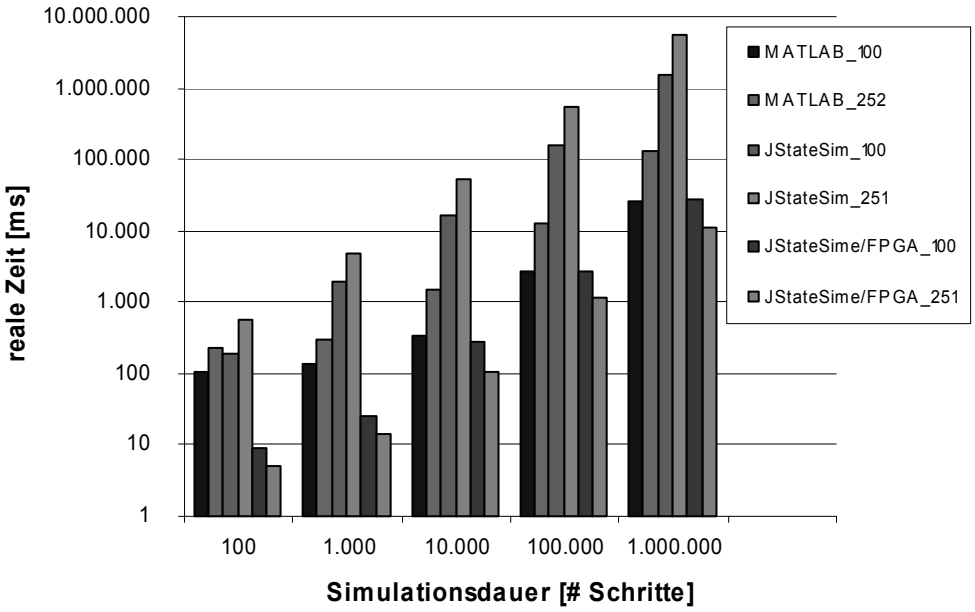


Bild 213: Messung am Stand-alone-Modell

Um zu verdeutlichen, wie groß der Aufwand für die Kommunikation beim Einsatz des JNI ist, wurde eine Vergleichsmessung zu einem in C geschriebenen Simulator erstellt (Bild 214). Die ersten beiden Säulenpaare sind dabei dieselben Messungen aus Bild 213. Die beiden letzten stellen die gleichen Versuche dar, wurden jedoch ohne JNI, nur mittels eines C-Programms simuliert. Der Geschwindigkeitszuwachs beträgt Faktor 3 bis 3,5.

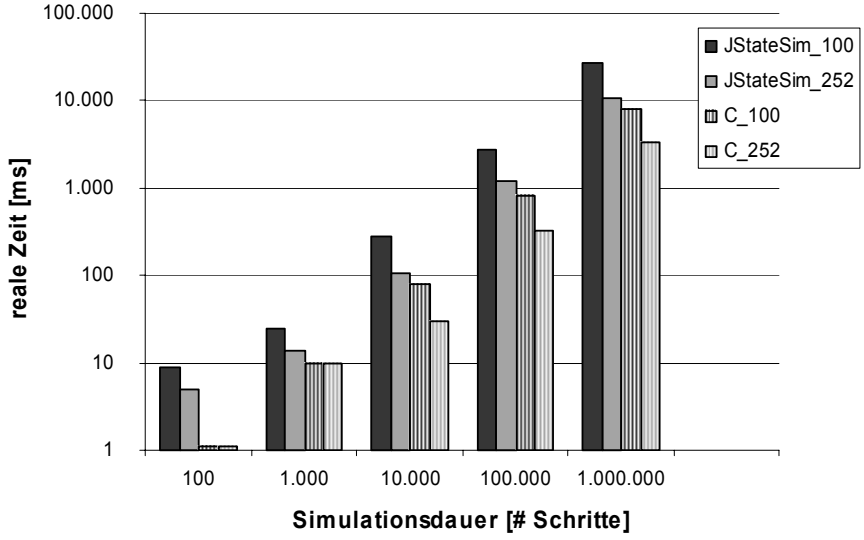


Bild 214: Vergleich zwischen JNI und C

Für die verteilte Simulation wurde folgendes Modell verwendet. Das Simulink-Modell entspricht dem aus Bild 212. Das Statechart (Bild 215) besteht jedoch aus zwei parallelen Zuständen (`B`, `C`). Diese bestehen wiederum aus vier Unterzuständen zu je 1000 Zuständen.

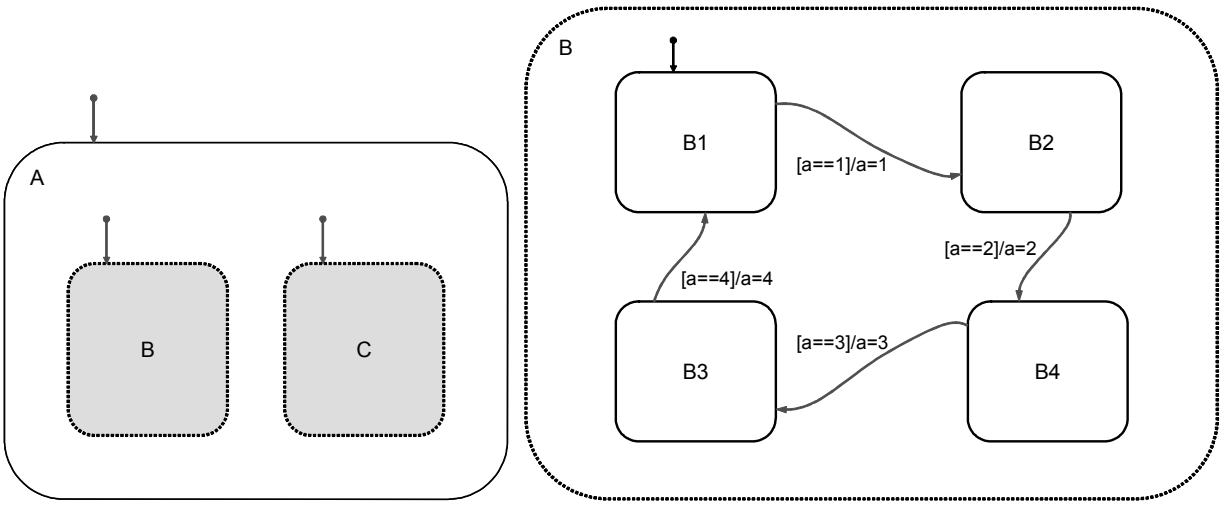


Bild 215: Modell zur verteilten Simulation

Eine Verteilung auf zwei Rechner (Bild 216: `JStateSim_verteilt`) erbringt hierbei bei langer Laufzeit eine doppelt so schnelle Ausführungszeit wie die entsprechende Simulation auf einem Rechner (Bild 216: `JStateSim_autonom`).

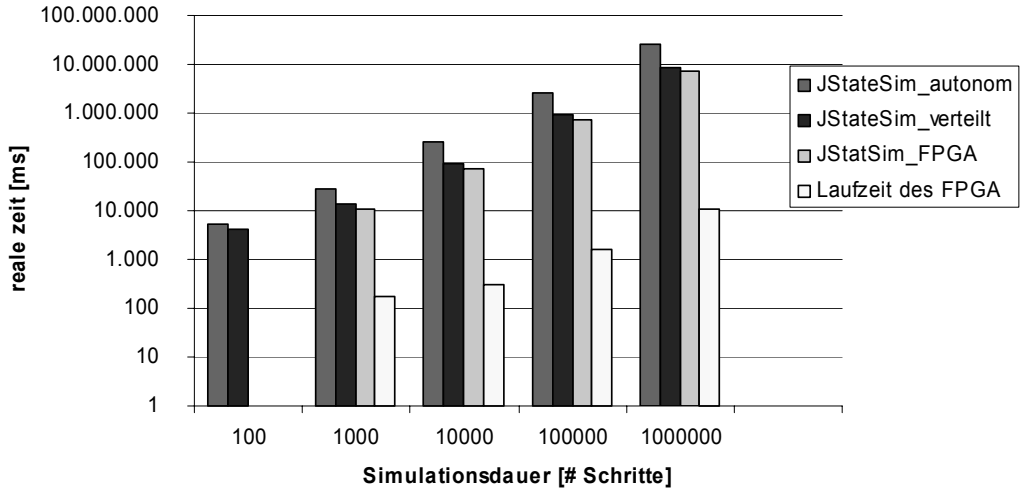


Bild 216: Messung bei verteilter Simulation

Keinen großen Gewinn bringt hingegen der Einsatz des FPGA (Bild 216: `JStateSim_FPGA`) in der verteilten Simulation, da die beiden Teilmodelle ungefähr die gleiche Komplexität aufweisen. Die Beschleunigung des FPGA von beinahe drei Größenordungen bei langer Laufzeit führen nur zu einer um Faktor 1,2 kürzeren Gesamtsimulationsdauer. Hier wird die Bedeutung einer geeigneten Partitionierung auf Modellebene sichtbar.

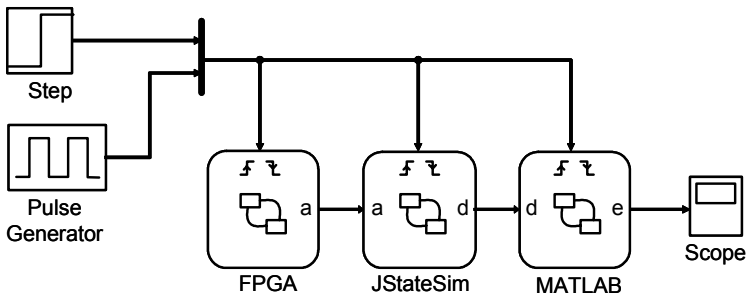


Bild 217: Verteilte Simulation mit MATLAB

Beim in Bild 217 dargestellten Versuch werden alle Simulatoren in eine verteilte Simulation einbezogen. Ausgehend von den vorangegangenen Versuchen enthält das Teilmodell `FPGA` 4000 und `JStateSim` 50 Zustände sowie `MATLAB` einen Zustand. Die Teilmodelle wurden entsprechend ihren Namen auf den verschiedenen Simulatoren ausgeführt. Die entsprechende Schnittstelle ist in [Bukh-03] beschrieben. In ihr wird ebenfalls JNI verwendet, um auf Java-Klassen zur Kommunikation mit JStateSim zuzugreifen. Bild 218 zeigt das Modell nach der Partitionierung in MATLAB. Im vorliegenden Fall dient es lediglich der Aufbereitung und Darstellung der Simulationsergebnisse.

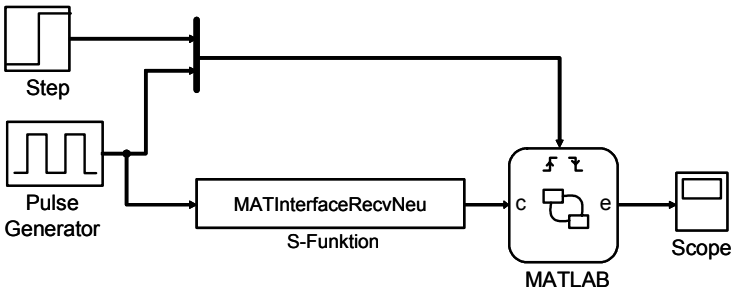


Bild 218: Teil des Modells in MATLAB

In Bild 219 werden zwei Messreihen dieses Modells (Bild 215) gegenübergestellt. In der Reihe `FPGA-JS-MAT` wird das Modell, wie oben beschrieben, auf dem FPGA, in JStateSim und MATLAB simuliert. Zum Vergleich zeigt `JS-JS-MAT` die gleiche Simulation ohne die Unterstützung des FPGA. Dessen Teilmodell wird, da MATLAB Modelle dieser Größe nicht mehr simuliert, von JStateSim übernommen. Die Geschwindigkeitsdifferenz beträgt bis zu Faktor 50.

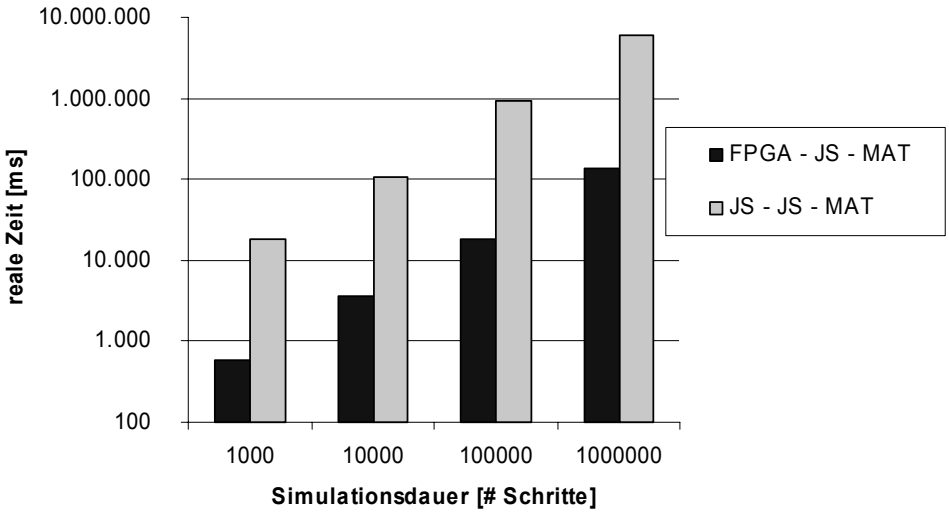


Bild 219: Verteilte Simulation

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Ereignisgesteuerte diskrete Simulation ist eine große und wichtige Klasse von Anwendungen. In den letzten Jahren wurde nach Alternativen gesucht, solche Simulationen schneller durchführen zu können. Durch die Parallelisierung, bei der jedes Einzelexperiment auf einem anderen Prozessor abläuft, wird für viele Bereiche eine hohe Beschleunigung erreicht. Dies gilt umso mehr, als Prozessoren zunehmend billiger, Zeit jedoch immer kostbarer wird. Dieser Ansatz hilft jedoch wenig, wenn die Experimente voneinander abhängig sind oder ein einzelnes Experiment möglichst schnell durchgeführt werden soll. Die verteilte und parallele Simulation versucht, die vorhandene Parallelität durch Parallelisierung der Ereignisausführungen auszunutzen. Die Partitionierung der Menge aller auszuführenden Ereignisse vereinfacht das Erkennen kausal unabhängiger Ereignisse erheblich. Die verbleibenden Abhängigkeiten lassen sich durch eine Vielfalt konservativer, optimistischer oder hybrider Simulationsverfahren synchronisieren. Bei verteilter und paralleler Simulation stellt sich die Frage nach einer möglichen Beschleunigung gegenüber sequenzieller Simulation. Da sich diese Frage aufgrund der Komplexität realistischer Simulationsmodelle und deren inhärent asynchronen Ausführung i. A. einer mathematisch-analytischen Untersuchung entzieht, stehen hier vor allem empirische Untersuchungen solcher Verfahren im Vordergrund. Einen großen Einfluss auf die Performance hat der vergleichsweise hohe Synchronisationsaufwand, der in verteilten Systemen durch den Austausch von Nachrichten entsteht.

Im Rahmen der Arbeit wurde das plattformunabhängige Co-Simulations-Framework INTERACT entwickelt, das heterogene Ausführungsumgebungen integriert und bei dem die Kommunikation über Ethernet erfolgt. Mit INTERACT können Systemmodifikationen in Richtung des späteren gewünschten Verhaltens sowie Fehlerkorrekturen in einer sehr frühen Entwicklungsphase angewandt werden. Es wurde gezeigt, dass Wege zur Steigerung der Simulationsgeschwindigkeit der Einsatz dedizierter Simulatoren wie JLogSim und JStateSim für die Logik- bzw. für die zustandsbasierte Steuergeräte-Simulation sind sowie vorcompilierte Modelle anstatt Modelle, die bei ihrer Ausführung interpretiert werden. JLogSim und JStateSim unterstützen effiziente Synchronisationsprotokolle und erlauben eine parallele Modellausführung alternativ zu einer seriellen Ausführungsreihenfolge. Weiterhin wurde eine Target-Host-Kommunikations-Infrastruktur entwickelt, die eine Prozessor-in-the-Loop-Simulation mit dem späteren Produktionscode, dem Betriebssystem und der Bus-Schnittstelle ermöglicht sowie eine Lösung zur verteilten Simulation hybrider elektronischer Systeme auf PC (Simulink und Stateflow), Mikrocontrollern und FPGAs mit Kommunikation über CAN. Komplexe, automatisch generierte Modelle und Modelle realer Systeme dienten zur Validierung des Verteilungs- und Kommunikationskonzeptes. Zur Generierung von VHDL-Code für Statecharts wurde das Werkzeug JVHDLGen entwickelt. Basierend auf der Semantik von Stateflow werden nebenläufige Zustände sequenziell in einem Prozess bearbeitet. Die Bearbeitungsreihenfolge im VHDL-Code stimmt mit der Priorität in den Stateflowcharts überein. Mit ModelSim wurde generierter VHDL-Code simuliert, und durch den Vergleich der Ergebnisse mit denen von MATLAB wurde JVHDLGen verifiziert.

Für einen durchgängigen Entwurfsfluss wird automatisch generierter Code von komplexen Statecharts benötigt, der für FPGAs z. B. mit Hilfe von JVHDLGen generiert werden kann. Da ein FPGA ein System emuliert, anstatt es zu simulieren, besteht dadurch ein großes Potential, die Ausführung um mehrere Größenordnungen zu beschleunigen. Z. B. kann die Ausführungsgeschwindigkeit von Statecharts durch Parallelverarbeitung auf dem FPGA aufgrund seiner Hardware-Architektur wesentlich gesteigert werden, besonders von Charts mit nebenläufigen Zuständen. Ein Nachteil von FPGAs ist zurzeit noch der Kostenfaktor.

Die Integration verschiedener Simulatoren in ein Co-Simulations-Framework verbessert die Performance und ermöglicht die Kopplung von Simulatoren zur Simulation unterschiedlicher Kategorien von Systemen. Um diese beiden Ziele zu erreichen, wurde ein Kontrollprozess entwickelt, der die

gesamte Ausführung steuert. Schließlich wurde eine hardwarebasierte Ausführungsumgebung (FPGA-Emulator) in das Co-Simulations-Framework integriert. Der FPGA-Emulator besitzt die gleiche Kommunikationsschicht, die von JStateSim und JLogSim verwendet wird.

Die Kommunikation bei JSimControl und JStateSim wurde von RMI nach TCP/IP umgestellt, um eine flexible Ankopplung weiterer Simulationskomponenten zu ermöglichen. Hierfür wurde die Programmablaufstruktur von JStateSim an die zusätzlich notwendigen Threads für die Kommunikation und die Implementierung von Protokollen zum Austausch von Steuer-, Status- und Simulationsdaten angepasst. Durch die Einbindung von Simulink in das Simulations-Framework über eine S-Funktion mit TCP/IP-Funktionalität können während der Simulationslaufzeit generierte Stimuli gesendet und empfangene Simulationsergebnisse dargestellt werden. Das erstellte System wurde hinsichtlich seiner Leistung überprüft. Beantwortet wurden hierbei Fragen nach dem Ressourcenbedarf auf der einen und der Leistungssteigerung auf der anderen Seite. Dazu wurden spezielle Testmodelle und -methoden entworfen und zahlreiche Messungen durchgeführt und bewertet. Die erreichte Simulationsbeschleunigung der Gesamtsimulation liegt im erwarteten Bereich zwischen 10 und 100. Die Beschleunigung des FPGA gegenüber der Ausführung auf einem PC konnte mit bis zu drei Größenordnungen angegeben werden. Es zeigte sich somit, dass die Partitionierung eines Modells und die damit entstehende Kommunikation einen entscheidenden Einfluss auf die Leistung des Gesamtsystems hat, da die damit verbundene Kommunikation über das Netzwerk die Simulation bremst.

Durch Verteilung der Simulation auf mehrere Rechner steigt die Anzahl der gesendeten Nachrichten. In Abhängigkeit vom zu simulierenden Modell und von der Anzahl der Nachrichten kann die Performance schlechter werden. Der Speicherbedarf für das optimistische Verfahren wurde gemessen. Zu Beginn der Simulation wird mehr Speicher gebraucht, da die JVM während der Ausführung alle Klassen konvertiert, so dass die Ausführung des Programms danach beschleunigt wird. Bei optimistischer Simulation auf drei Rechnern wird mehr Speicher gebraucht als bei der Simulation auf zwei Rechnern, da die Simulation schneller als die auf zwei Rechnern ist. Daraus folgt, dass mehr Zustandsspeicherungen als auf zwei Rechnern durchgeführt werden. Im Gegensatz dazu wurde bei Simulation auf sechs Rechnern weniger Speicher gebraucht; da die Simulation auf sechs Rechnern nicht viel schneller als diejenige auf drei Rechnern läuft, ist die Anzahl der durchgeführten Zustandsspeicherungen ungefähr gleich, aber die Anzahl der Zustände nicht. Deshalb benötigt die Simulation auf sechs Rechnern weniger Speicher als die auf drei Rechnern.

Eine effiziente Partitionierung individueller Statecharts ist eine Voraussetzung für einen guten statischen Lastausgleich. Eine Anforderung für die Ausführung einer Co-Simulation sind deshalb passend partitionierte Modelle, so dass jeder Teilnehmer der Simulation bzw. Emulation so effizient wie möglich ist und der aus der Verteilung des Modells resultierende Zusatzaufwand für die Kommunikation so gering wie möglich bleibt. Als Basis für eine Co-Simulation wurde deshalb eine effiziente Partitionierungsstrategie für Statecharts vorgeschlagen, die für Logikschaltungen bewährte Partitionierungsalgorithmen integriert. Ein Modell einer Logikschaltung kann direkt in einen Hypergraph umgesetzt und partitioniert werden. Ein Modell aus Statecharts wird anhand seiner Nebenläufigkeit und der Abhängigkeiten zwischen nebenläufigen Zuständen in einen Hypergraph umgesetzt. Anders als bei einer Logikschaltung ist der Rechenaufwand jedes Knotens in einem Hypergraph für Statecharts unterschiedlich, weil jeder nebenläufige Zustand ein anderes Verhalten haben kann. Durch Performance-Messungen wurde der Rechenaufwand verschiedener Beschreibungsmittel bei der Umsetzung eines Modells von Statecharts in einen Hypergraphen berechnet, um das Gewicht eines Zustands festzulegen. Weil der Kommunikationsaufwand die Performance der verteilten Simulation beeinträchtigt, wurde der Partitionierungsalgorithmus entsprechend so angepasst, dass zwei Partitionen zusammengefasst werden, wenn die Anzahl von Schnitten zwischen ihnen nach der Partitionierung eine bestimmte Anzahl überschreitet. Wie man aus den Messergebnissen entnehmen kann, kann die verteilte Simulation durch Partitionierung eines Modells in sechs Partitionen mehr als vier Mal so schnell wie die Standalone-Simulation sein. Die Performance der verteilten Simulation hängt aber auch von der Struktur der Abhängigkeitsgraphen der Partitionen ab. Bei einem langen Pfad im Ab-

hängigkeitsgraph kann sich die verteilte Simulation wegen einer langen Wartezeit auf Ereignisse verlangsamen. Der längste Pfad wird mit Hilfe des Partitionierungsalgorithmus ermittelt.

Mit Hilfe von Performance-Messungen wurden auch die Simulationszeit und der Speicherbedarf bei einer Zunahme der Modellgröße untersucht. Ab einem Bereich von 50000 Zuständen nimmt die Effektivität einer verteilten Simulation überproportional zu, da dann die Standalone-Simulationszeit wesentlich schneller als die Modellgröße zunimmt. Auch der mit der Modellgröße zunehmende Speicherbedarf ist ein Grund für die verteilte Simulation. Bei der optimistischen Synchronisation wird die Performance verteilter Simulation nicht nur von der Anzahl der Schnitte zwischen Partitionen stark beeinflusst, sondern es spielen die Abhängigkeiten zwischen nebenläufigen Zuständen innerhalb einer Partition eine große Rolle.

Es wurde weiterhin eine Lösung zur verteilten Simulation hybrider elektronischer Systeme auf PC (Simulink/Stateflow), Mikrocontrollern (MPC555) und FPGAs mit Kommunikation über CAN entwickelt. Die Realisierung basiert auf einem Konzept zur Kopplung von Modellteilen zwischen Simulink/Stateflow und einem Mikrocontroller mit Hilfe von S-Funktionen zur CAN-Kommunikation in Simulink. Zur Verifikation der Funktionalität wurden sowohl Modelle geringer als auch hoher Komplexität entworfen, mit denen zum einen die Anwendbarkeit der Verteilung und des Kommunikationskonzepts bei realitätsnahen Systemen überprüft wurde und bei denen zum anderen eine im Rahmen dieser Arbeit entwickelte Methode zur Analyse des Zeitverhaltens angewandt wurde, die die graphische Darstellung von Ausführungsdauer und -zeitpunkten der Tasks und die Erkennung von Leerlaufzeiten ermöglicht und somit eine Optimierung des Task-Schedulings unterstützt. Bei den Modellen für Mikrocontroller kommt zum Teil OSEK zum Einsatz, so dass die in OSEK realisierte externe Kommunikation hinsichtlich Zweckmäßigkeit, Performance und Speicherbedarf analysiert werden konnte. Zusätzlich wurden OSEK-Alarms zur Task-Aktivierung im Hinblick auf ihre Anwendbarkeit bei performance-optimierten verteilten Systemen mit CAN-Kommunikation untersucht. Für das FPGA wurde mit JVHDLGen aus Stateflowcharts VHDL-Code generiert und dieser in ein FPGA-CAN-Framework integriert. Bezüglich der Sendeleistung des MPC555 lässt sich feststellen, dass der direkte Zugriff auf die Treiberfunktionen einen wesentlich höheren Datendurchsatz ermöglicht, als dies bei Nutzung der externen Kommunikation von OSEK der Fall ist.

Die evaluierten Echtzeit-Betriebssysteme haben ihre jeweiligen Vor- und Nachteile. Die Ergebnisse zeigen, dass offensichtliche Unterschiede hinsichtlich der Performance und des Speicherverbrauchs existieren, die u. a. von den jeweiligen Parametern und der Konfiguration der Applikation abhängen. Einige RTOS werden mit integrierter Entwicklungsumgebung geliefert, andere nicht. Einige sind mehrere Jahre auf dem Markt etabliert, extensiv getestet und mit zahlreichen Werkzeugen ausgestattet, während andere innovative oder sehr nützliche Erweiterungen bieten. Welche Eigenschaften essentiell sind, welche nützlich sind und welche nicht benötigt werden, hängt von der jeweiligen Applikation ab.

Ein bereits etabliertes, für Mixed-Mode-Simulation bewährtes Protokoll wurde erweitert und an das INTERACT-Framework angepasst. Statecharts und kontinuierliche Modelle können durch eine effiziente Art und Weise gekoppelt werden, da die Simulationsgeschwindigkeit durch den Einsatz eines optimistischen Mechanismus anstatt der Ausführung im Lock-Step-Modus signifikant gesteigert wurde. Ein kontinuierlicher Simulator wurde in eine entsprechend erweiterte optimistische Co-Simulationsumgebung integriert. Dieses Framework zur optimistischen Co-Simulation hybrider Systeme wurde durch Umsetzung der Second Event Synchronization und der konzipierten synchronen Hypothese realisiert. Dadurch können Modelle hybrider Systeme auf mehrere Rechner verteilt mit reduziertem Kommunikationsaufwand optimistisch simuliert werden. Hierfür wird mit RTW aus den in Simulink entworfenen kontinuierlichen Teilmodellen C-Code generiert und daraus selbständig ausführbare Modelle erstellt. Die hierfür benötigte RTW-Laufzeitumgebung wurde den Anforderungen entsprechend modifiziert, indem TCP/IP-Kommunikation integriert und notwendige Funktionalitäten für die optimistische Simulation (Zustandsspeicherung, Rücksetzung, Negativnachrichten, GVT, Speicherverwaltung) implementiert wurden. Bislang stehen vier parametrisierbare Hypothese-

funktionen zur Verfügung; die Modulbibliothek ist durch eine definierte Schnittstelle und Klassenvererbung für Erweiterungen vorbereitet. Dabei ist beim Entwurf der Hypothesefunktionen auf einen möglichst geringen Performance-Bedarf zu achten, da andernfalls die Leistungsgewinne aufgrund einer höheren Hypothesegüte durch die hierfür benötigte Rechenzeit verloren gehen können.

Die Funktionalität von JSimControl, JStateSim, Simulink und der optimistischen RTW-Laufzeitumgebung wurde zunächst einzeln und im Anschluss bei verteilter Ausführung mit entsprechend entworfenen Testmodellen verifiziert. Die Ergebnisse der Performance-Messungen zeigen, dass das Aktivierungskonzept der SES eine gute Lösung darstellt. Das Potential zur Simulationsbeschleunigung durch optimistische Verfahren konnte für hybride Modelle nachgewiesen werden. So liegt die Ausführungsgeschwindigkeit von JStateSim im Best-Case bis zu 57 mal höher als beim Worst-Case, da der kontinuierliche Simulator aufgrund ideal angepasster Hypothesefunktionen keine Korrekturwerte sendet und JStateSim unter optimaler Ausnutzung des Optimismus ohne Rücksetzung simulieren kann. In diesem Fall sind die Voraussetzungen für eine Simulationsbeschleunigung durch parallele Ausführung vollständig erfüllt. Neben der Hypothesegüte sind die Triggerrate und eine den HW-Anforderungen von Modell und Simulator angepasste Prozessorzuteilung entscheidende Parameter für die Simulationsgeschwindigkeit.

6.2 Ausblick

Die Konzepte sind nicht an ein bestimmtes eingebettetes System gebunden, sondern können auch auf andere Systeme angewendet werden. Aufbauend auf den Ergebnissen der vorliegenden Arbeit könnten in zukünftigen Untersuchungen andere Komponenten und Konzepte eingebunden werden. So lassen sich evtl. durch den Einsatz von konservativen und optimistischen Ausführungsverfahren bei FPGAs Beschleunigungen im Ausführungsablauf erreichen. Weiterhin können Bussysteme wie LIN, FlexRay und MOST in die Untersuchungen mit eingebunden werden. Nach einer detaillierten Analyse ausgewählter Anwendungsbeispiele mit charakteristischen Eigenschaften, evtl. unter Verwendung weiterer Mikrocontroller- und FPGA-Typen, könnten die in einem Richtlinienkatalog zusammengefassten Ergebnisse den Entwickler bei der Wahl einer passenden Partitionierung unterstützen. Weitere CASE-Werkzeuge wie iLogix STATEMATE können ins Framework eingebunden werden. Dabei ist allerdings das Zeitverhalten von STATEMATE und Stateflow unterschiedlich, so dass bei einer Erweiterung des Frameworks durch STATEMATE dies berücksichtigt werden muss. JStateSim und JVHDLGen können weiterentwickelt werden. Die Übertragung optimistischer Methoden in den Kontext automatischer Code-Generierung für Hardware wie FPGAs scheint aufgrund ihrer inhärenten Performance erfolgversprechend zu sein.

Als nächster Schritt zur Verbesserung des Entwurfsflusses ist die Automatisierung der einzelnen Schritte zu nennen. Bisher wird das Modell von Hand konvertiert und in den Simulator eingehängt. Sinnvoll erscheint ein automatischer Aufruf von JVHDLGen nach der Partitionierung des Modells mittels JStateSim. Hat man das Modell in VHDL vorliegen, wäre außerdem eine automatische Einbindung des Modells in den bestehenden Simulator wünschenswert. Dazu ist eine Erweiterung bzw. eine parametrisierbare Instanziierung des Steuerwerks notwendig, so dass die Anzahl der Ein- und Ausgangsvariablen beliebig skalierbar ist. JVHDLGen kann z. B. hinsichtlich der Typkonvertierung von Operanden erweitert werden. Die FPGA-CAN-Infrastruktur kann flexibel hinsichtlich der Anzahl und Datenbreite von Ein- und Ausgangssignalen erweitert werden.

Bei der Implementierung des Partitionierungsalgorithmus wurde angenommen, dass alle Ereignisse zwischen verschiedenen Partitionen mit gleicher Häufigkeit gesendet werden. Wenn die mögliche Häufigkeit eines Ereignisses ermittelt werden kann, kann der Partitionierungsalgorithmus den Schnitt mit dem größten Gewinn beim Verschieben von Knoten auswählen.

Eine optimierte Version der angepassten optimistischen RTW-Laufzeitumgebung kann zu einer weiteren deutlichen Beschleunigung der Gesamtsimulation führen. Aus implementierungstechnischer Sicht steckt insbesondere in der Thread-Synchronisation noch Optimierungspotential. Hinsichtlich

der Hypothesemodule gibt es ein großes Spektrum möglicher zusätzlicher Realisierungsvarianten, die je nach Modell zu einer erhöhten Trefferquote und damit einer schnelleren Simulation führen können. Jedoch sollte bei der Implementierung eines neuen Moduls daran gedacht werden, dass komplexe Hypothesefunktionen einen größeren Bedarf an Rechenzeit und Speicher haben, der beim Übersteigen eines bestimmten Grades an Komplexität, die Simulationsgeschwindigkeit wesentlich beeinflusst. Der Einsatz von statistischen Methoden und Interpolationsverfahren bietet sich an und könnte interessante Ergebnisse liefern. Ein Kriterienkatalog bezüglich der Eignung unterschiedlicher Modelltypen kann deutliche Zeitersparnis bringen, da der Benutzer bereits im Vorfeld einer verteilten Simulation einordnen kann, unter welchen Bedingungen und mit welchen Parametern das jeweilige Modell am günstigsten simuliert wird.

Auch auf Hardwareseite sind Optimierungsansätze denkbar. So kann z. B. die Ausführung eines oder mehrerer DES-CS-Paare auf einem Mehrprozessorsystem die Netzwerklast reduzieren und dadurch die Simulation weiter beschleunigen. Ein Toleranzintervall kann für die Eingangssignale des DES eingeführt werden, wobei der aktuelle Wert mit dem vorhergehenden verglichen wird. Das Signal wird als unverändert betrachtet, wenn die Abweichung innerhalb des Toleranzintervalls liegt. Somit kann die Sensibilität des Systems und seine Dynamik reduziert werden, so dass der DES das Stateflowchart seltener auswerten muss. Die Voraussetzung der zyklenfreien Verteilungsstruktur bezüglich der Signalpfade beim verwendeten Time-Warp-Algorithmus stellt konzeptionell die größte Einschränkung dar, die sich bei der Simulation hybrider Systeme noch stärker als bei rein zustandsbasierten Modellen auswirkt, da Signalschleifen, die sowohl kontinuierliche als auch ereignisdiskrete Modellteile einschließen, nicht durch Integration in ein gemeinsames Teilmodell eliminiert werden können, wie dies bei homogenen Systemen der Fall ist. Um in einem solchen Fall optimistisch simulieren zu können, müssen die Signalschleifen über hybride Modellteile durch Modellmodifikationen aufgebrochen werden, was nicht immer möglich ist. In diesem Kontext können zukünftige Arbeiten zu einer weiteren Verbreiterung des Spektrums verteilt simulierbarer Modelle beitragen.

Anhang

A Literaturverzeichnis

A.1 Referenzierte Literaturstellen

[AaBr89] Y. Aahlad, J. C. Browne. Balanced Sequencing Protocols; Proceedings of the SCS Multiconference on Distributed Simulation (Tampa, Florida, 1989), 58-63

[Abra88] M. Abrams. „The Object Library for Parallel Simulation"; Proceedings of the Winter Simulation Conference; 1988

[AgTi91] J. Agre, P. Tinker. „Useful Extensions to a Time Warp Simulation System"; S. 78-85; Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation; 1991; Anahein/California

[AhQu00] S. P. Ahuja, R. Quintao. Performance Evaluation of Java RMI: A Distributed Object Architecture for Internet Based Applications. IEEE Proc. of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000.

[AlDD98] W. v. Almsick, T. Drabe, W. Daehn et al. A Central Control Engine for an Open and Hybrid Simulation Environment. IEEE Second International Workshop on Distributed Interactive Simulation and Real Time Applications (DIS-RT 1998), Montreal/Canada, 1998.

[Amda67] G. Amdahl. Validity of the single-processor approach to archieving large-scale computing capabilities. Proc. AFIPS Conference 30 (1967) 483

[Anso03] Ansoft. Mit Co-Simulation zu alternativen Antrieben, Auto & Elektronik, 2-3/2003, S.35, Mai 2003

[ArSm92] D. Arvind, C. Smart. „Hierarchical Parallel Discrete Event Simulation in Composite Elsa"; S. 147-156; Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92); 1992; Newport Beach/California

[ASCE00] ETAS GmbH & Co.KG. ASCET V4.0 Benutzerhandbuch; Stuttgart, 2000

[AuEl00] Auto & Elektronik, Sonderheft D 51994, Ausgabe 2/2000, Lässt tief blicken, S.98, Hüthig-Verlag

[AuEl01] Auto & Elektronik, D 51994, Ausgabe 2/2001, QA-C unterstützt M.I.S.R.A. Standard, S. 10, Hüthig-Verlag

[AuEl02] Auto & Elektronik, D 51994, Ausgabe 6/2002, Flexibles, Entwicklungstool für komplexe Netzwerke, S.59, Hüthig-Verlag

[AuEl99] Auto & Elektronik, Sonderheft D 19067, Ausgabe 1/1999, S.73 und S.113, Hüthig-Verlag

[AvTr95] H. Avril, C. Tropper. Clustered Time Warp and Logic Simulation. In: Proceedings of the 9th Workshop on Parallel and Distributed Simulation: 112-119, 1995.

[Bach00] R. Bachmann. HLA und CORBA: Partner oder Konkurrenten?. 14. Symposium SCS/Asim Simulationstechnik, Hamburg, 9/2000

[Bagr97] R. Bagrodia. PARSEC User Manual, Release 1.0. UCLA Parallel Computing Lab, 1997.

[Balz96] H. Balzert. Lehrbuch der Softwaretechnik: Software-Entwicklung. Spektrum, Akademischer Verlag, Heidelberg 1996.

[BaPL01] A. Bakshi, V. K. Prasanna, A. Ledeczi. MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems. ACM LCTES 2001, Snowbird, Utah, USA, S.82-87, 2001.

[Bark96] E. Barke: Bei Electronic Design Automation bleibt der Weg das Ziel. In: F & M, Volume 104, Heft 6, Seiten 422 bis 424, 1996.

[BaRR01] V. Balakrishnan, R. Radhakrishnan, D. M. Rao et al. A performance and scalability analysis framework for parallel discrete event simulators. Elsevier Simulation Practice and Theory Vol. 8, S.529-553, 2001.

[Barr02] M. Barr, (2002). Special Report: Choosing an RTOS. Embedded Systems Programming.

[Barr99] M. Barr. „Programming Embedded Systems in C and C++“, O'Reilly & Associates, Inc., Beijing, Cambridge, Köln, 1999

[BaSc88] W. Bain, D. Scott. „An Algorithm for Time Synchronization in Distributed Discrete Event Simulation"; S. 30-33; Proceedings of the Multiconference on Distributed Simulation; 1988; San Diego/California

[Baue02] Automotive Electronics + Systems, Ausgabe 1/2002, Jahrgang 1, F. Bauer, Mit vereinten Kräften, S.30, Hanser-Verlag

[BeHa01] T. Beierlein, O. Hagenbruch. Taschenbuch Mikroprozessortechnik, 2. Auflage, Fachbuchverlag Leipzig, 2001

[BeJe85] O. Berry, D. Jefferson. „Critical Path Analysis of Distributed Simulation"; S. 57-60; Proceedings of the Distributed Simulation Conference; 1985; San Diego

[Bell93] S. Bellenot. „Performance of a Riskfree Time Warp Operating System"; S. 155-158; Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS93); 1993; San Diego/California

[Berg02] A. S. Berger. Embedded Systems Design, CMP Books, 2002

[Bhas99] J. Bhasker. A Verilog HDL Primer, Second Edition. Star Galaxy Publishing, 1999.

[BjKo02] Elektronik Automotive, April 2002, M. Björkander, C. Kobryn. „UML 2.0 – der nächste Schritt“, S.30-33, WEKA Fachzeitschriften-Verlag GmbH

[Blon01] Auto & Elektronik, D 51994, Ausgabe 1/2001, D. Blondin, Vollständige OSEK-Entwicklungsplattform, S.62-65, Hüthig-Verlag

[BMFT94] Bundesministerium für Forschung und Technologie (1994), Initiative zur Förderung der Software-Technologie; Wirtschaft, Wissenschaft und Technik, Bonn

[Boll00] G. Bollella. The Real-Time Specification for Java, Addison-Wesley, 2000

[Bort94] J. Bortolazzi. Untersuchungen zur rechnergestützten Erfassung, Verwaltung und Prüfung von Anforderungsspezifikationen und Einsatzbedingungen elektronischer Steuerungs- und Regelungssysteme. Dissertation, Universität Erlangen-Nürnberg, 1994.

[Bosc05] Robert Bosch GmbH. Die automatische Code-Generierung feiert Jubiläum, S.30, Elektronik Automotive, Ausgabe 2/2005

[BoSe94] G. Bolch, M. M. Seidel. Prozeßautomatisierung, Teubner 1993

[BrBK89] F. Brglez, D. Bryan, K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits. In: Proceedings of the IEEE International Symposium on Circuits and Systems: 1929-1934, S.75-81, 1989.

[BrFS87] P. Bratley, B. L. Fox, L. E. Schrage. A Guide to Simulation; Springer-Verlag (New York, 1987)

[BrFu85] F. Brglez, H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran. Proc. of the IEEE International Symposium on Circuits and Systems, 1985, S.104-111.

[Brie01] B. Briel. Analyse eingebetteter Systeme mittels verteilter Simulation. Dissertation, Oktober 2001, Universität Oldenburg, Abteilung für Betriebssysteme und Verteilte Systeme, Berichte aus dem Fachbereich Informatik

[BrMS99] J. Bruns, C. Müller-Schloer. An Integrated System-Level Modelling and Simulation Environment for Embedded Control Systems. ESM 1999, S. 247-251, 1999.

[Broc01] Embedded Engineering, Ausgabe 2, Juni 2001, H. Brock, Monitor auf Rädern, S.30-32, Hanser-Verlag

[Broc99] Auto & Elektronik, Sonderheft D 19067, Ausgabe 1/1999, Standards schaffen mit OSEK, H. Brock, S.101-103, Hüthig-Verlag

[Broo99] D. Brook, (1999). Real Time Embedded Operating Systems. Embedded Intelligence '99, Design & Elektronik. Hartmut Rogge.

[Brya77] R. E. Bryant. Simulations of Packet Communication Architecture Computer Systems. Master of Science Thesis, Massachusetts Institute of Technology, 1977.

[Brya79] R. E. Bryant. Simulation on a Distributed System. In: Proceedings of the Conference on Distributed Computing Systems: 544-552, 1979.

[BSPW02] Wind River, OSEKWorks for PowerPC Board Support Package Guide 5.0 (Wind River, 2002).

[Büch00] Auto & Elektronik, Sonderheft D 51994, Ausgabe 1/2000, F. Büchner, Neue Ansätze für das Testen von OSEK/VDX-Applikationen, S.106-109, Hüthig-Verlag

[Büch01] Embedded Engineering, Ausgabe 1 2001, Februar 2001, F. Büchner, OSEK/VDX-Debugging auf dem Weg zum Standard, S.45-49, Hanser-Verlag

[Buhl02] Auto & Elektronik, D 51994, Ausgabe 6/2002, M. Buhlmann. „TTA (zeitgesteuerter Daten-Bus), S.16-19, Hüthig-Verlag

[BuJa98] A. Buss, L. Jackson. Distributed Simulation Modeling: A Comparison of HLA, CORBA, and RMI. Proc. of the Winter Simulation Conference 1998. S.819-825, 1998.

[Burc93] M. Burckhardt, Fahrwerktechnik: Radschlupf-Regelsysteme. Vogel, 1993

[C16697] Siemens AG. User's Manual SAB 80C166/83C166, 1997

[CaBP00] C. D. Carothers, D. Bauer, S. Pearce. ROSS: A High-Performance, Low Memory, Modular Time Warp System, Proc. of IEEE PADS 2000, 2000

[Calv93] J. P. Calvez. „Embedded Real-Time Systems“, Willy series in software engineering practice, John Wiley & Sons, Chichester, New York, Brisbane, 1993

[Came01] K. Camera. SF2VHD: A Stateflow to VHDL Translator, Master Thesis, Department of Electrical Engineering and Computer Science, R. Brodersen, University of California, Berkeley, California, USA, 2001.

[CAN94] ISO: ISO 11898: Austausch digitaler Informationen; Controller Area Network (CAN) für schnellen Datenaustausch, 1994.

[CANB91] Robert Bosch GmbH, CAN Specification 2.0 (Robert Bosch GmbH, 1991)

[CANL03] Vector Informatik GmbH, CAN Driver Library, User Interface Description, Version 3.2 (Vector Informatik GmbH, 2003)

[ChGo93] D. Cheriton, H. Goosen et al. „Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: The Value of Distributed Synchronization"; 159-162; Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS93); 1993; San Diego/California

[ChHM79] K. M. Chandy, V. Holmes, J. Misra. Distributed Simulation of Networks; Computer Networks, Vol. 3, No. 1 (1979), 105-113

[ChJB97] Y. Chen, V. Jha, R. Bagrodia. 1997. "A Multidimensional Study on the Feasibility of Parallel Switch-Level Circuit Simulation". 11th Workshop on Parallel and Distributed Simulation PADS '97.

[ChKR03] C. Chang, K. Kuusilinna, B. Richards et al. Rapid Design and Analysis of Communiation Systems Using the BEE Hardware Emulation Environment. Proc. of the IEEE Rapid Prototyping Workshop, Juni 2003

[ChMi78] K. Chandy, J. Misra. „A Non-Trivial Example of Concurrent Processing: Distributed Simulation"; S. 822-826; Proceedings of COMPSAC; 1978

[ChMi79] K. Chandy, J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. In: IEEE Transactions on Software Engineering, 5(5): 440-452, 1979.

[ChMi81] K. M. Chandy, J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations; Communications of the ACM, Vol. 24, No. 11 (1981), 198-206

[ChSh89] K. M. Chandy, R. Sherman. Space-Time and Simulation; Proc. of the SCS Multiconference on Distributed Simulation, (Tampa, Florida, 1989), 53-57

[Comf84] J. Comfort. „The Simulation of a Master-Slave Event Set Processor"; S. 117-124; Simulation 42; 1984

[Coom01] Design & Elektronik, Heft 2, Februar 2001, A. Coombes, Blick ins OSEK, S.60-61, WEKA Fachzeitschriften-Verlag GmbH

[CoZB94] J. Cong, L. Zheng, R. Bagrodia. 1994. "Acyclic Multi-Way Partitioning of Boolean Networks". Design Automation Conference 1994, 670-675.

[DaBe01] Embedded Engineering, Ausgabe 4 2001, Nov. 2001, C. Dallmayr, G. Bengs. Quo vadis Steuergerätevernetzung?, S.50-51, Hanser-Verlag

[DaFP94] S. Das, R. M. Fujimoto, K. Panesar et al. GTW: A Time Warp System for Shared Memory Multiprocessors. Proc. of 1994 Winter Simulation Conference, S.1332-1339, Dez. 1994.

[DaKW98] J. S. Dahmann, F. Kuhl, R. Weatherly. Standards for Simulation: As Simple As Possible But Not Simpler, The High Level Architecture For Simulation. Simulation 1998, 71:6,7, S. 378-387.

[DaPa02] D. Davis, M. Parashar. Latency Performance of SOAP Implementations. IEEE Cluster Computing and the Grid, 2002.

[DaWT95] E. M. Dale, P. A. Wilsey, J. M. Timothy. WARPED Simulation Kernel Documentation Version 0.5, Sept. 1995.

[DeE01a] Design & Elektronik, Heft 2, Februar 2001, LIN macht mobil, S.83-84, WEKA Fachzeitschriften-Verlag GmbH

[DeE01b] Design & Elektronik, Heft 4, April 2001, Mentor Graphics, Veni, Vidi, Leonardo, S.56, WEKA Fachzeitschriften-Verlag GmbH

[DeE02a] Design & Elektronik, Heft 3, März 2002, Universität Magdeburg, Was ist ein OSEK-OS?, Betriebssystem-Spezifikationen, S.26-27, WEKA Fachzeitschriften-Verlag GmbH

[DeE02b] Design & Elektronik, Heft 3, März 2002, FPGA macht mobil, Flexible Hardware-Plattform, S.14-16, WEKA Fachzeitschriften-Verlag GmbH

[DeE02c] Design & Elektronik, Heft 3, März 2002, Zweikampf, Alteras Stratix und Xilinx' Virtex-II Pro, S.6, WEKA Fachzeitschriften-Verlag GmbH

[DeSE01] Dedicated Systems Experts (2001). What make a good RTOS and Evaluation Report Definition. Belgium.

[Diec96] M. Dieckmann. Entwurf und Implementierung einer CORBA-basierten Simulation Backplane unter Berücksichtigung verschiedener Kommunikationsmechanismen. Diplomarbeit, Universität Karlsruhe (TH), 1996.

[DiRe90] P. Dickens, P. Reynolds. „SRADS with Local Rollback"; S. 161-164; SCS Multi-Simulation Conference; 1990; Washington D.C.

[DuFM01] J. Dunlop, J. Fuchs, S. Mihalik, (2001). User manual MPC555 and application note: AN2109/D interrupts rev. 0. Motorola, Inc.

[DuIn01] Duden Informatik, 3. Auflage, Dudenverlag, 2001

[EbDi89] M. Ebling, M. Diloreto et al. „An Ant Foraging Model Implemented on the Time Warp Operating System"; S. 21-26; Proceedings of the SCS Multiconference on Distributed Simulation; 1989

[ElA01a] Elektronik Automotive, Mai 2001, FlexRay für verteilte Anwendungen im Fahrzeug, S.40-43, WEKA Fachzeitschriften-Verlag GmbH

[ElA01b] Elektronik Automotive, Sept. 2001, VHDL-AMS, S.88-95, WEKA Fachzeitschriften-Verlag GmbH

[ElA02a] Elektronik Automotive, April 2002, Durchbruch revolutioniert Automobilindustrie, X-by-Wire-Technologie, S.6, WEKA Fachzeitschriften-Verlag GmbH

[ElA02b] Elektronik Automotive, Dezember 2002, S.59, WEKA Fachzeitschriften-Verlag GmbH

[ElAu00] Elektronik Automotive, Juni 2000, S.18-20, WEKA Fachzeitschriften-Verlag GmbH

[ElAu03] Autoindustrie etabliert offenen Standard, S.10, Elektronik Automotive, Ausgabe 5/2003

[Elia02] Design & Elektronik, Heft 10, Oktober 2002, C. Elias. Code auf Knopfdruck, S.56-57, WEKA Fachzeitschriften-Verlag GmbH

[ERCO00] ETAS GmbH & Co.KG. ERCOSEK V4.0 Benutzerhandbuch; Stuttgart, 2000

[ErST94] J. Ernst, S. Schmerler, Y. Tanurhan. Studie zu Spezifikation und Aufbau einer Cosimulationsumgebung unter Berücksichtigung des Simulator-Backplane-Standards der CAD Framework Initiative. FZI/ESM, Karlsruhe, 1994.

[ETHa05] ETAS. Optimierter Code mit ASCET V5.1. Hanser Automotive, Ausgabe 5-6 2005, S.44.

[EXCI03] H.-M. Schulz. EXITE Release 1.3 Facts Sheet. Extessy AG, 2003.

[EXDS02] H.-M. Schulz. Description of ExITE and Distributed Simulation Toolbox. Extessy AG, Dez. 2002.

[Fers95] A. Ferscha. Probabilistic Adaptive Direct Optimism Control in Time Warp. In: Proceedings of the 9th Workshop on Parallel and Distributed Simulation: 120-129, 1995.

[FiMa82] C. Fiduccia, R. Mattheyses. 1982. "A Linear Time Heuristic for Improving Network Partitions". Proc. of 19th ACM/IEEE Design Automation Conference DAC 1982, 175-181.

[Fish78] G. S. Fishman. Principles of Discrete Event Simulation; Wiley & Sons (New York, 1978)

[Fish95] P. A. Fishwick. Simulation Model Design and Execution. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[FlKe89] H. P. Flatt, K. Kennedy. Performance of Parallel Processors, Parallel Computing 12:1-20, 1989

[Föll86] O. Föllinger. Lineare Abtastsysteme, 3. Auflage, Oldenbourg, 1986

[Forr72] J. W. Forrester. Grundzüge einer Systemtheorie; Th. Gabler (Wiesbaden, 1972)

[FoSc97] M. Fowler, K. Scott. UML Distilled, Applying the Standard Object Modeling Language. Addison-Wesley, 1997.

[FrCW97] P. Frey, H. W. Carter, P. A. Wilsey. Parallel Synchronization of Continuous Time Discrete Event Simulators. Proc. of the IEEE 1997 International Conference on Parallel Processing ICPP'97, 1997.

[FrCW98] P. Frey, H. W. Carter, P. A. Wilsey. Optimistic Synchronization of Mixed-Mode Simulators. Proc. of the 12th International Parallel Processing Symposium, März 1998.

[FrRa00] P. Frey, R. Radhakrishnan, H. W. Carter et al. Parallel mixed-technology simulation. Proc. of the 14th IEEE Workshop on PADS, Bologna, Italy, S.7-14, Mai 2000.

[Fuj88a] R. Fujimoto. „Lookahead in Parallel Discrete Event Simulation"; S. 24-41; Proceedings of the International Conference on Parallel Processing; 1988

[Fuj88b] R. Fujimoto. „Performance Measurements of Distributed Simulation Strategies"; S. 14-20; Proceedings of the Distributed Simulation Conference; 1988; San Diego/California

[Fuji00] R. M. Fujimoto. Georgia Institute of Technology: „Parallel and Distributed Simulation Systems" (John Wiley & Sons, Inc., 2000).

[Fuji01] R. M. Fujimoto. Parallel and distributed simulation systems. Proc. of the 2001 Winter Simulation Conference, S.147-157, 2001

[Fuji89] R. M. Fujimoto. „Time Warp on a Shared Memory Multiprocessor"; S. 242-249; Proceedings of the International Conference on Parallel Processing; 1989

[Fuji90] R. M. Fujimoto. Parallel Discrete Event Simulation. In: Communications of the ACM, 33(10): 31-53, 1990.

[GaDW92] D. D. Gajski, N. D. Dutt, C. A. Wu. High-Level Synthesis: Introduction to Chip and System Design, Kluwer academic publishers, 1992.

[Gafn85] A. Gafni. Space Management and Cancelling Mechanisms for Time Warp. In: Technical Report TR-85-31, University of Southern California, 1985.

[GaKu83] D. Gajski, R. Kuhn. Guest Editor's Introduction: New VLSI Tools. IEEE Computer, 6(12): 11-14, 12 1983

[Garr01] Design & Elektronik, Heft 4, April 2001, J. Garrison, Das richtige Timing, S.62-63, WEKA Fachzeitschriften-Verlag GmbH

[GaSp01] Auto & Elektronik, D 51994, Ausgabe 2/2001, T. Galla, M. Sprachmann et al., Entwurf und Konfiguration von X-by-Wire-Netzwerken, S.50-53, Hüthig-Verlag

[GhPa03] B. Ghahramani, M. A. Pauley. Java in High-Performance Environments, p.109, IEEE Computer, Sept. 2003

[GoFU95] F. Gomes, S. Franks, B. Unger, et al. SimKit: A High Performance Logical Process Simulation Class Library in C++. Proc. of the 1995 Winter Simulation Conference, S. 706-713, Arlington, VA. Dez. 1995.

[Golz96] U. Golze. VLSI Chip Design with the Hardware Description Language VERILOG, Springer-Verlag, 1996

[Görz00] S. Görzig. Und nun alle zusammen. c't 2000, Heft 22, Heise-Verlag.

[Gose91] K. Goser. Großintegrationstechnik. Teil 2: Von der Grundschaltung zum VLSI-System. Heidelberg: Hüthig Buch Verlag, 1991, erschienen in der Reihe ELTEX Studientexte Elektrotechnik

[Gray99] D. Gray. (1999) Introduction to the Formal Design of Real Time Systems; Springer Verlag

[Gril02] Design & Elektronik, Heft 3, März 2002, J. Grillmayer, CAN wird aktiv, S.40-43, WEKA Fachzeitschriften-Verlag GmbH

[Gros95] B. Groselj. CPSim: A Tool for Creating Scalable Discrete-Event Simulations. Proc. of 1995 Winter Simulation Conference, S. 579-583, Arlington, VA, Dez. 1995.

[Grüt02] R. Grützner. Simulation im Wandel der Zeit – Mathematische Konzepte: Entwicklung, Anwendung, Probleme. ASIM Nachrichten, Nov. 2002. S.4-9

[GTW94] Georgia Tech Time Warp GTW Version 3.1 Programmer's Manual. Georg Tech Research Corporation, Atlanta, Georgia, USA. 1994.

[GTWP96] K. S. Perumalla, R. M. Fujimoto. User Manual: GTW++ – An Object-oriented Interface in C++ to the Georgia Tech Time Warp System. Sept. 1996.

[HaDo88] T. Hartrum, B. Donlan. „Distributed Battle-Management Simulation on a Hypercube”; S. 3-7; Proceeding of the Distributed Simulation Conference; 1988; San Diego/California

[Häfn98] T. Häfner. Diplomarbeit: Entwurf und Implementierung einer Ankopplung der Echtzeit-Rapid-Prototyping Plattform dSPACE an die CoSimulationsumgebung SimBa, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 11. Dez. 1998

[HaGe97] D. Harel, E. Gery. Executable Object Modeling with Statecharts. IEEE Computer, 1997.

[Hare87] D. Harel, A Visual Formalism for Complex Systems, Science of Computer Programming, Heft 8, Seiten 231 bis 274, 1987.

[HeAr04] H. Herold, J. Arndt. C Programmierung unter LINUX, UNIX, WINDOWS, SuSE Press, April 2004

[Henk96] J. Henkel. Automatisierte Hardware/Software-Partitionierung im Entwurf integrierter Echtzeitsysteme. Dissertation, Technische Universität Braunschweig, Shaker Verlag, 1996.

[HeSc02] Elektronik Automotive, Sept. 2002, H. Heinecke, A. Schedl et al. FlexRay – ein Kommunikatonssystem für das Automobil der Zukunft, S.36-45, WEKA Fachzeitschriften-Verlag GmbH

[Hoar85] T. Hoare. Communicating Sequential Processes, Prentice-Hall (New Jersey, 1985)

[HoBe89] P. Hontalas, B. Beckman et al. „Performance of the Colliding Pucks Simulation on the Time Warp Operating Systems (Part 1: Asynchronous Behvior & Sectoring)"; S. 3-7; Proceedings of the SCS Multiconference on Distributed Simulation; 1989

[Hoff99] J. Hoffmann: MATLAB- und SIMULINK in Signalverarbeitung und Kommunikationstechnik, Addison-Wesley Verlag, 1999

[Hofm00] Elektronik Automotive, Juni 2000, P. Hofmann u. a., Automotive UML – eine neue objektorientierte Entwicklungstechnik, S.88-95, WEKA Fachzeitschriften-Verlag GmbH

[Hofm55] J. Hofmeister. Wörterbuch der philosophischen Begriffe, 2. Aufl., Hamburg 1955

[Hora02] S. Horabi. Diplomarbeit: Evaluierung der Qualität und Anwendbarkeit der automatischen Code-Generierung beim durchgängigen Entwurf eingebetteter elektronischer Systeme, Nov. 2002

[Hupf00] S. Hupfer. The Nuts and Bolts of Compiling and Running JavaSpaces Programs. Java Developer Connection, Sun Microsystems, Inc., 2000.

[IEC91] ISO IEC 9126 (1991); Information Technology; Software Product Evaluation, Quality, Characteristics and Guidelines for their use

[IEEE96] The IEEE Standard Dictionary of Electrical and Electronics Terms 6th Edition, IEEE Press, 1996

[IRef99] IEEE Standard VHDL Language Reference Manual (Integrated with VHDL-AMS changes), IEEE Standard 1076.1, Juli 1999

[Jazd98] N. Jazdi. „Komponentenbasierte Entwicklung Eingebetteter Systeme (KEES)", Arbeitsbericht von Juli 1997 bis Juni 1998, Universität Stuttgart, Institut für Automatisierungsund Softwaretechnik, 1998

[JeBe87] D. Jefferson, B. Beckmann et al. „Distributed Simulation and the Time Warp Operating System"; S. 77-93; Proceedings of the 11th ACM Symposium on Operating Systems Principles; 1987

[Jef85a] D. R. Jefferson. Virtual Time; ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3 (Juli 1985), S.404-425

[Jef85b] D. R. Jefferson. Implementation of Time Warp on the Caltech Hypercube, Proc. of the SCS Distributed Simulation Conference (San Diego, California, 1985)

[JeRe91] D. R. Jefferson, P. Reiher. „Supercritical Speedup"; S. 159-168; Proceedings of the 24th Annual Simulation Symposium; 1991; New Orleans/Louisiana

[JeSo82] D. R. Jefferson, H. Sowizral. Fast Concurrent Simulation using the Time Warp Mechanism, Part I: Local Control, Rand Corporation (Santa Monica, California, 1982)

[JeSo83] D. R. Jefferson, H. Sowizral. Fast Concurrent Simulation using the Time Warp Mechanism, Part II: Global Control, Rand Corporation (Santa Monica, California,

1983)

[JeSo85] D. R. Jefferson, H. Sowizral. Fast Concurrent Simulation using the Time Warp Mechanism; Proc. of the SCS Conference on Distributed Simulation (San Diego, California, 1985), 63-69

[Joha93] R. Johannis. Handbuch des 80C166. Berlin: Siemens AG, 1993.

[JoKR63] Johnson, Kast, Rosenzweig. „The Theory and Management of Systems“, 1963

[JuGl04] F. Junker, G. Glöe. Gewährleistung der Produktsicherheit im Sinne der IEC 61508, RealTimes, ETAS S.28

[Jütt02] P. Jüttner. Siemens Product Structures: Airbag and Electromechanical Brake. Siemens-Veröffentlichungen 2002, 2002

[KeLi70] B. W. Kernighan, S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. Bell Syst. Tech. J., Vol. 49, No. 2, Feb. 1970, S. 291-307

[Kell88] H. B. Keller. Parallele Simulationsmethoden zur Ausführung komplexer Modelle; Informatik-Fachberichte Bd. 187, Springer-Vorlag, (New York, 1988), 270-278

[Kepp94] T. Keppler. Entwurf, Realisierung und Bewertung von Algorithmen und Strategien zur verteilten ereignisgesteuerten Simulation auf Mehrrechner-Systemen. Fortschrittberichte VDI, Reihe 10, Nr.285, 1994

[KeRi90] Kernighan, Ritchie. Programmieren in C, Carl Hanser, 1990

[Kien97] U. Kiencke. Ereignisdiskrete Systeme, Oldenbourg, 1997

[KiJä02] U. Kiencke, H. Jäkel. Signale und Systeme, 2. Auflage, Oldenbourg, 2002

[Kope97] H. Kopetz. (1997); Real-Time Systems, Kluwer Academic Publishers

[KöTS01] L. Köster, T. Thomsen, R. Stracke. Connecting Simulink to OSEK: Automatic Code Generation for Real-Time Operating Systems with TargetLink. Embedded Intelligence Nürnberg, Feb. 2001.

[KrBa96] V. Krishnaswany, P. Banerjee. Actor Based Parallel VHDL Simulation Using Time Warp. ACM SIGSIM Simulation Digest, Proc. of the 10th Workshop on PADS, V. 26, Is. 1, S. 135-142, July 1996.

[KrBE01] H. Krisp, Jochen Bruns, Stefan Eilers et al. Multi-Domain Simulation for the Incremental Design of Heterogeneous Systems. SCS 15th European Simulation Multiconference ESM'2001, Prag, Tschechische Republik, Juni 2001, S.381-386.

[Kris84] B. Krishnamurthy. An Improved Min-Cut Algorithm for Partitioning VLSI Networks. IEEE Transaction on Computers, Vol. 33, No. 5, Mai 1984, S.438-446

[KrTa03] Kraftfahrtechnisches Taschenbuch, 25. Auflage, Robert Bosch GmbH, 2003, Friedr. Vieweg & Sohn Verlag

[Krüg03] G. Krüger. Handbuch der Java-Programmierung, 3. Auflage, Addison-Wesley, 2003

[Krüg75] S. Krüger. Simulation – Grundlagen, Techniken, Anwendungen Walter de Gruyter Verlag, 1975

[KuZO95] F. Kuhn, N. Zalila, C. Otto. Neue Entwurfsmethode in der Übertragungstechnik. In: telcom report 18, Heft 6, Seiten 314 und 315, 1995.

[LaBa92] P. Lanches, U. G. Baitinger. A Parallel Evaluation Environment for Distributed Logic Simulation. Modelling and Simulation, European Simulation Multiconference, S.465-469, 1992.

[LaMu83] S. Lavenberg, R. Muntz. Performance Analysis of a Rollback Method for Distributed Simulation; Performance '83, North-Holland Publ. Comp., (Amsterdam, 1983), 117-132

[Lapr92] J.C. Laprie. Dependability: Basic Concepts and Terminology. Springer-Verlag, Wien 1992

[Lasc02] Automotive Electronics + Systems, Ausgabe 5/2002, Jahrgang 1, R. Laschewski. LIN – Günstige Vernetzung im Fahrzeug, S.22-24, Hanser-Verlag

[LeCl89] E. Leung, J. Cleary et al. „The Effects of Feedback on the Performance of Conservative Algorithms"; S. 44-49; Proceedings of the SCS Multiconference on Distributed Simulation; 1989

[Ledi01] J. Ledin. Simulation Engineering, CMP Books, 2001

[Leim01] Design & Elektronik, Heft 4, April 2001, W. Leimbach, Rhapsody in UML, S.24-S.26, WEKA Fachzeitschriften-Verlag GmbH

[Lemi00] J. Lemieux, (2000). Embedded Systems Programming, Vol. 13, No. 3 (pp. 90-108). California, San Francisco.

[Lemi01] J. Lemieux, Programming in the OSEK/VDX Environment. CMP Books, 2001.

[Leng90] T. Lengauer, 1990. „Combinational Algorithms for Integrated Circuit Layout". John Wiley & Sons, Inc.

[LePa99] T. Lechler, B. Page. DESMO-J: An Object-Oriented Discrete Simulation Framework in Java. Proc. of the European Simulation Symposium '99, 1999.

[LeWS94] G. Lehmann, B. Wunder, M. Selz, Schaltungsdesign mit VHDL (Franzis, 1994)

[Li03] Q. Li. Real-Time Concepts for Embedded Systems, CMP Books, 2003

[LiMo04] The MathWorks. Link for ModelSim. Select, Ausgabe 1/2004, S.43-45

[Lin92] Y. Lin. „Parallelism Analyzers for Parallel Discrete Event Simulation"; S. 239-264; ACM Transactions on Modeling and Computer Simulation 2:3; 1992

[Lipp00] H. M. Lipp. Grundlagen der Digitaltechnik, 3. Auflage, Oldenbourg, 2000

[LiPr93] Y. Lin, B. Preiss. „Selecting the Checkpoint Interval in Time Warp Simulation"; S. 3-10; Proceedings of the 7th Workshop on Parallel and Distributed Simulation; 1993; San Diego/California

[LoLC99] Y.-H. Low, C.-C. Lim, W. Cai et al. Survey of Languages and Runtime Libraries for Parallel Discrete-Event Simulation. SCS Simulation, Special Issue: Parallel and Distributed Simulation, Vol.72, No.3, S.170-186, 1999.

[LuSh89] B. Lubachevsky, A. Shwartz. „Rollback Sometimes Works ... If Filtered"; S.630-639; Proceedings of the Winter Simulation Conference; 1989

[Lutz03] B. Lutz, Diplomarbeit: Hardware/software co-design with abstract design tools on production code level, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, Sept. 2003.

[MaHa92] V. Madisetti, D. Hardaker. „The MIMDIX Operating System for Parallel Simulation"; S. 65-75; Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92); 1992; Newport Beach/California

[MaKR95] E. Mascarenhas, F. Knop, V. Rego. ParaSol: A Multithreaded System for Parallel Simulation Based on Mobile Threads. Proc. of 1995 Winter Simulation Conference,

S. 690-697, Arlington, VA, Dez. 1995.

[MaLo93] N. Manjikian, W. Loucks. „High Performance Parallel Logic Simulation on a Network of Workstations"; S. 76-84; Proceedings of the 7th Workshop on Parallel and Distributed Simulation; 1993; San Diego/California

[MaMe89] F. Mattern, H. Mehl. „Diskrete Simulation – Prinzipien und Probleme der Effizienzsteigerung durch Parallelisierung"; S. 198-210; Informatik-Spektrum 12:4; 1989

[Math03] Embedded Targets, MATLAB Select 1/03, S.21, S.42, The MathWorks, 2003

[MaWa88] V. Madisetti, J. Walrand. „A Rollback Algorithm for Optimistic Distributed Simulation Systems"; S. 296-305; Proceedings of the Winter Simulation Conference; 1988

[MaWa89] V. Madisetti, J. Walrand. „Efficient Distributed Simulation"; S. 5-21; Annual Simulation Symposium; 1989

[MaWa91] V. Madisetti, J. Walrand et al. „Asynchronous Algorithms for the Parallel Simulation of Event-Driven Dynamical Systems"; S. 244-274; ACM Transaction on Modeling and Computer Simulation, Vol. 1, Issue 3, July 1991

[McCl89] C. McClure. CASE is Software Automation. Prentice Hall, Englewood Cliffs, NJ, USA, 1989.

[Meh94b] H. Mehl. „Methoden verteilter Simulation"; Vieweg Verlag; 1994

[Mehl91] H. Mehl. „Speed-Up of Conservative Distributed Discrete Event Simulation Methods by Speculative Computing"; S. 163-166; Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation; 1991; Anaheim/California

[Mehl94] H. Mehl, 1994. „Methoden verteilter Simulation, Vieweg-Verlag". Braunschweig, Germany

[Ment02] VIKING CSM, Datasheet, Mentor Graphics, 2002

[MGBS99] K. D. Müller-Glaser, A. Burst, S. Schmerler. Rapid Prototyping von Informationssystemen für Kraftfahrzeuge; it+ti 05/1999, Oldenbourg Verlag, 1999

[MGFS04] K. D. Müller-Glaser, G. Frick, E. Sax, et al. Multiparadigm Modeling in Embedded Systems Design. IEEE Transactions on Control Systems Technology, Vol. 12, Nr. 2, März 2004, S.279-292

[MiLD92] P. Michel, U. Lauther, P. Duzy. The Synthesis Approach to Digital System Design. Kluwer Academic Publishers, Boston, MA, USA, 1992.

[Misr86] J. Misra. Distributed Discrete Event Simulation; ACM Computing Surveys, Vol. 18, No. 1 (März 1986), 39-65

[Mont99] S. Montenegro. 1991; German National Research Center for Information Technology, Institute for Computer Architecture and Software Technology; Einfache Sicherheit – Komplexe Gefahr; Zeitschrift Elektronik, Mai 1999, Heft Nummer 9

[Moor65] G. Moore. Moore's Law: Electronics, 35th Anniversary Issue, April 1965

[MPC500] MPC555 / MPC556 User's Manual, Rev. 15., Motorola, Oct. 2000

[MüBS00] K. D. Müller-Glaser, A. Burst, B. Spitzer et al. Rapid Prototyping von eingebetteten elektronischen Systemen. it +ti Informationstechnik und Technische Informatik, Jahrgang 42, Heft 2, 2000, Oldenbourg Verlag, S.8-15

[MüGl00] K. D. Müller-Glaser. Entwurf elektronischer Systeme I, Vorlesungsunterlagen, ITIV Universität Karlsruhe, 2000

[Müll01] Auto & Elektronik, D 51994, Ausgabe 2/2001, B. Müller, Die Zeit ist reif!, S.44-47, Hüthig-Verlag

[Neel87] F. Neelamkavil. „Computer Simulation and Modelling"; John Wiley & Sons; 1987

[Nico88] D. Nicol. „Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks"; S. 124-137; SIGPLAN 23:9; 1988

[Nied96] R. Niederhagen. Logiksimulation heute. In: Elektronik, Heft 9, Seiten 64 bis 70, 1996.

[NiJo97] D. M. Nicol, M. M. Johnson. The IDES Framework: A case study in development of a parallel discrete-event simulation system. Proc. of the 1997 Winter Simulation Conference, 1997, S.93-99.

[NoCh95] B. L. Noble, R. D. Chamberlain. Predicting the Future: Resource Requirements and Predictive Optimism. In: Proceedings of the 9th Workshop on Parallel and Distributed Simulation: 157-164, 1995.

[OSEK03] OSEK/VDX syndicate (2003). OSEK/VDX operating system specification and ORTI specification.

[Oswa03] M. Oswald. Zuverlässigkeit mit Brief und Siegel, S.50, Elektronik Automotive, 2003

[Page91] B. Page. „Diskrete Simulation – Eine Einführung mit Modula-2"; Springer-Verlag; New York, 1991

[PaLe00] B. Page, T. Lechler, S. Claassen: Objektorientierte Simulation in Java, Libri Books on Demand, 1. Auflage, 2000

[PaWi93] A. Palaniswamy, P. Wilsey. „An Analytical Comparison of Periodic Checkpointing and Incremental State Saving"; S. 127-134; Proceedings of the 7th Workshop on Parallel and Distributed Simulation; 1993; San Diego/California

[Perr91] D. L. Perry. VHDL, McGraw-Hill 1991

[Pete90] L. R. Peter. Parallel Simulation Using the Time Warp Operating System. Proc. of 1990 Winter Simulation Conference, Dez. 1990.

[Petr62] C. A. Petri. Kommunikation mit Automaten, Dissertation, Schriften des Rheinisch-Westfälischen Instituts für Instrumentelle Mathematik, 1962

[PrEb89] M. Presley, M. Ebling et al. „Benchmarking the Time Warp Operating System with a Computer Network Simulation"; S. 8-13; Proceedings of the SCS Multiconference on Distributed Simulation; 1989

[Prei90] B. Preiss. „Performance of Discrete Event Simulation on a Multiprocessor Using Optimistic and Conservative Synchronization"; S. 218-222: International Conference on Parallel Processing; 1990

[PreY90] B. R. Preiss. Yaddes – Yet Another Distributed Discrete-Event Simulator: User Manual. Technical Report, Dept. of Electrical and Computer Engineering, University of Waterloo, Canada, 1990.

[Pril93] G. Prillwitz. Simulatoren: Vielseitigkeit durch Vielfalt Zeitschriftenaufsatz, Elektronik 18/93

[PrLo91] B. Preiss, W. Loucks et al. „Null Message Cancellation in Conservative Distributed Simulation"; S. 33-38; Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation; 1991; Anaheim/California

[PrRa88] A. Prakash, C. Ramamoorthy. „Hierarchical Distributed Simulation"; Proceedings of

the 8th International Conference on Distributed Computing ICDCS, San Jose, S. 341-348, 1988.

[RaAy93] H. Rajaei, R. Ayani et al. „The Local Time Warp Approach to Parallel Simulation"; S. 199-126; Proceedings of the 7th Workshop on Parallel and Distributed Simulation; 1993; San Diego/California

[RaDe05] O. Ragonesi, F. Dengler. Rasch simulieren, sicher testen, Elektronik Automotive 3/2005, S.47-49

[Ramm89] F. J. Rammig. Systematischer Entwurf digitaler Systeme, Stuttgart: B. G. Teubner, 1989, S. 14 ff.

[Rao01] Embedded Engineering, Ausgabe 2, Juni 2001, M. Rao, UML für Echtzeitsysteme, S.42-45, Hanser-Verlag

[RaPr02] Rapid-Prototyping System RP.2002, entwickelt am Institut für Technik der Informationsverarbeitung, Universität Karlsruhe, Karlsruhe, 2002.

[Raus02] Elektronik Automotive, Dezember 2002, M. Rausch: Optimierte Mechanismen und Algorithmen in FlexRay, S.36-40, WEKA Fachzeitschriften-Verlag GmbH

[Reed83] D. A. Reed. A Simulation Study of Multimicrocomputer Networks; Proc. of the Int. Conf. on Parallel Processing (1983), 161-163

[ReFu87] D. A. Reed, R. M. Fujimoto. Multicomputer Networks: Message Based Parallel Processing; M.I.T. Press, (Cambridge, 1987)

[Reis86] W. Reisig. Petrinetze: Eine Einführung, Springer-Verlag, 1986

[ReM88a] D. A. Reed, A. D. Malony. Parallel Discrete Event Simulation: The Chandy Misra Approach; Proc. of the Distributed Simulation Conference, (San Diego, California, 1988), 8-13

[RePa93] P. F. Reynolds, C. M. Pancerella et al. „Design and Performance Analysis of Hardware Support for Parallel Simulations"; S. 435-453; Journal of Parallel and Distributed Computing 18; 1993

[RePo99] P. Rechenberg, G. Pomberger. Informatik-Handbuch, 2. Auflage, Carl Hanser Verlag, 1999

[Reyn88] P. Reynolds. „A Spectrum of Options for Parallel Simulation"; S. 325-332; Proceedings of the Winter Simulation Conference; 1988

[Reyn91] P. F. Reynolds. „An Efficient Framework for Parallel Simulations"; S. 167-174; Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation; 1991; Anaheim/California

[RoLA96] R. Ronngren, M. Liljenstam, R. Ayani et al. A Comparative Study of State Saving Mechanism for Time Warp Synchronized Parallel Discrete-Event Simulation. IEEE Proc. of Simulation, 1996.

[RTWU02] The MathWorks, Inc. Real-Time Workshop: For Use with Simulink, User's Guide V. 5, 2002

[SaHS05] E. Sax, N. Hartmann, S. Schmerler. HIL-Testing. Hanser Automotive, Ausgabe 5-6 2005, S.51-55.

[SaLi04] S. Sadeghipour, M. Lim. Konvertierungs-Tool unterstützt Software-Entwicklung, S.54, Elektronik Automotive, Ausgabe 1/2004

[Sama85] B. Samadi. Distributed Simulation, Algorithms and Performance Analysis; Ph. D. Dissertation, Dept. of Computer Science, University of California, (Los Angeles,

1985)

[Sanc89] L. A. Sanchis. Multiple-Way Network Partitioning. IEEE Transaction on Computers, Vol. 38, No. 1, Jan.1989, S.62-81

[SaZe00] H. S. Sarjoughian, B. P. Zeigler. DEVS and HLA: Complementary Paradigms for Modeling and Simulation, SCS Transactions of the Society for Modeling and Simulation International, Vol.17, No.4, Dez. 2000, S.187-197.

[Sche00] Elektronik Automotive, Juni 2000, P. Schedl, Nahtlos von der Spezifikation zum Code, S.83-87, WEKA Fachzeitschriften-Verlag GmbH

[Schm00] Auto & Elektronik, Sonderheft D 51994, Ausgabe 2/2000, Das LIN-Protokoll in einem RKE-System, T. Schmidt, S.20-24, Hüthig-Verlag

[Schm81] B. Schmidt. „Kombinierte Simulationsmodelle"; S. 126-127; Informatik Spektrum 4; 1981

[Schn97] H.-J. Schneider. Lexikon der Informatik, 4. Auflage, R. Oldenbourg Verlag, 1997

[Scho01] Embedded Engineering, Ausgabe 1 2001, Februar 2001, J. Schoof, Bremsen in Echtzeit, S.40-44, Hanser-Verlag

[Scho03] J. Schoof. OSEK trifft Middleware, S.28, Elektronik Automotive, Ausgabe 6/2003

[Scho99] Auto & Elektronik, Sonderheft D 19067, Ausgabe 1/1999, Ereignisse statt Zeitscheiben, J. Schoof, S.104-107, Hüthig-Verlag

[Schu02] Elektronik Automotive, Juni 2002, M. Schumacher: OSEKtime – ein zeitgesteuertes Standard-Betriebssystem, S.36-41, WEKA Fachzeitschriften-Verlag GmbH

[Schu03] Hans-Martin Schulz: Durchgängige Methodik und Techniken für verteilte Simulation, MATLAB Select 1/03, S.21, The MathWorks, 2003

[Schu78] H. H. Schulze. rororo Lexikon der Datenverarbeitung, Hamburg, 1978

[Schw02] R. Schwebel: Echtzeit unter Linux mit RTAI, Elektronik, Juli 2002

[ScMS99] S. Scherber, C. Müller-Schloer: An Efficient and Flexible Methodology for Modelling and Simulation of Heterogeneous Mechatronic Systems. DATE '99: Design, Automation and Test in Europe. München, März 1999

[ScMu97] H. Schwetman, A. Mulpur. The Vissim/Discrete Event Modeling Environment. Proc. of the 1997 Winter Simulation Conference, S.693-697, 1997.

[ScNe90] F. Schönthaler, T. Németh. Software-Entwicklungswerkzeuge: Methodische Grundlagen. Teubner, Stuttgart, 1990.

[ScSc03] R. Schmid, M. Schläfer. Testautomatisierung durch Adaption von xPC Target an TARA, MATLAB Select 2/03, S.16, The MathWorks, 2003

[ScTM95] S. Schmerler, Y. Tanurhan, K. D. Müller-Glaser. A Backplane Approach for Cosimulation in High-Level System Specification Environments, Proc. of the European Design Automation Conference with EURO-VHDL 95, Brighton, UK, 1995.

[ScTM96] S. Schmerler, Y. Tanurhan, K. D. Müller-Glaser. Towards Real-Time System Specification and Design. In: Proceedings of IPC '96.

[ScTM97] S. Schmerler, Y. Tanurhan, K. D. Müller-Glaser. Predictive Optimism in Logic Simulation, IASTED International Conference on Applied Modelling and Simulation AMS'97, Banff, Canada, 1997.

[ScWe04] U. Schneider, D. Werner. Taschenbuch der Informatik, 5. Auflage, Fachbuchverlag

Leipzig, 2004

[ScZu03] J. Schäuffele, T. Zurawka. Automotive Software Engineering, Vieweg & Sohn Verlag, 2003

[Seet78] M. Seethalakshmi. Performance Analysis of Distributed Simulation; M. S. Thesis, Dept. of Computer Science, University of Texas, (Austin, 1978)

[Seri94] ISO: ISO 11519: Straßenfahrzeuge – Serielle Datenübertragung mit niedriger Übertragungsrate, 1994.

[Seru02] Design & Elektronik, Heft 3, März 2002, M. Serughetti, Im Netz des Autos, Kfz-Kommunikationsstandards, S.18-21, WEKA Fachzeitschriften-Verlag GmbH

[SeTh01] Auto & Elektronik, D 51994, Ausgabe 1/2001, M. Seibt, M. Thanner, Software-In-The-Loop, S.50-57, Hüthig-Verlag

[SiBa97] S. Schmerler. Statemate Simulation Backplane V1.5 Integrator's Guide V1.0, Specification V0.2 und User's Guide V1.4, Projektdokument FZI/ESM, 1998

[Sieb03] F. Siebert. Java in Echtzeit, S.66, Elektronik, August 2003

[Siem01] C. Siemers. Hardwaremodellierung, Einführung in Simulation und Synthese von Hardware, Carl Hanser Verlag, 2001.

[Siem99] C. Siemers. Prozessorbau: Eine konstruktive Einführung in das Hardware-Software-Interface, Carl Hanser Verlag, 1999
Petri-Netze

[SiR01a] Embedded Engineering, Ausgabe 3, Sept. 2001, S. Siemers, T. Rapf, C. Siemers: Der Staat im Staate, S.9-12, Echtzeit-Betriebssysteme, Teil 2: Konkrete Auswahlkriterien, Hanser-Verlag

[SiR01b] Embedded Engineering, Ausgabe 2, Juni 2001, S. Siemers, T. Rapf, C. Siemers: Der Staat im Staate, S.7-10, Echtzeit-Betriebssysteme, Teil 1: Technische Grundlagen, Hanser-Verlag

[Simu04] The MathWorks, Inc., Simulink User's Guide, V. 6, 2004

[Stat04] The MathWorks, Inc., Stateflow User's Guide, V. 6, 2004

[SiSi03] C. Siemers, A. Sikora. Taschenbuch Digitaltechnik, Fachbuchverlag Leipzig, 2003

[SoBW88] L. M. Sokol, D. P. Briscoe, A. P. Wieland. MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution. In: Proceedings of the SCS Multiconference on Distributed Simulation: 34-42, 1988.

[SoGu91] L. Soulé, A. Gupta. „An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation"; 308-347; ACM Transactions on Modeling and Computer Simulation 1:4; 1991

[Soly00] A. Solymosi, I. Schmiedecke. „Programmieren mit Java", Vorlesung and der Universität Berlin, 2000

[Somm87] J. Sommerville. „Software Engineering", Addison-Wesley Verlag, 1987

[SoSt89] L. Sokol, B. Stucky et al. „MTW: A Control Mechanism for Parallel Discrete Simulation"; S. 250-254; International Conference on Parallel Processing; 1989

[Spie82] P. P. Spies. Grundlagen stochastischer Modelle; Carl Hanser Verlag, (München, 1982)

[Spir90] H. Spiro. Simulation integrierter Schaltungen durch universelle Rechenprogramme;

R. Oldenburg Verlag, (München, 1990)

[Spor94] C. Sporrer. „Verfahren zur Schaltungspartitionierung für die parallele Logiksimulation“; Shaker Verlag; 1994

[SpSS99] R. Spolon, M. J. Santana, R. H. C. Santana. Distributed Simulation, Time Warp and its Variations: Taxonomy and Performance Evaluation Issues. ESM 1999, S. 220-227, 1999.

[STAi95] STATEMATE User Reference Manual und Analyzer Reference Manual, Version 6.1. i-Logix Inc., Burlington, MA, USA, 1995.

[STAT95] Einführung in das CASE-Tool STATEMATE, Vorlesungsunterlagen, Institut für Technik der Informationsverarbeitung, Universität Karlsruhe (TH), 1995

[Ste92a] J. S. Steinman. SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation. International Journal in Computer Simulation, Vol. 2, No. 3, s. 251-286, 1992.

[Ste92b] J. Steinman. „SPEEDES: A Unified Approach to Parallel Simulation”; S.75-84; Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92); 1992; Newport Beach/California

[Stei91] J. Steinman SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. Proc. of the SCS Western Multiconference on Advances in Parallel and Distributed Simulation. Vol. 23, No. 1, 1991, Pages 95-103; Anaheim/California

[Stei93] J. S. Steinman. Breathing Time Warp. Proc. 7th Workshop on Parallel and Distributed Simulation, San Diego, California, Mai 1993, S. 109-118.

[Stev92] W. R. Stevens. Advanced Programming in the UNIX Environment, Addison-Wesley, 1992

[Stev98] W. R. Stevens. Network Programming Volume 2: Interprocess Communications, Prentice-Hall, 1998

[Stra99] S. Straßburger. On the HLA-Based Coupling of Simulation Tools, ESM 1999, 1999, S.45-51.

[Stre96] K. Strehl. Studienarbeit: Entwurf und Implementierung der Ankopplung des CASE-Werkzeugs STATEMATE an die Cosimulationsumgebung SimBa, Juli 1996

[Stre97] K. Strehl. Diplomarbeit: Entwurf und Implementierung eines prädiktiven, optimistischen Synchronisationskernels für die Cosimulationsumgebung SimBa, März 1997

[StSc01] S. Straßburger, T. Schulze. Verteilte- und Web-basierte Simulation: Gemeinsamkeiten und Unterschiede. SCS Frontiers in Simulation, Asim Fortschritte in der Simulationstechnik 15. Symposium in Paderborn, Hrsg. K. Panreck, F. Dörrscheidt. S.169-174, 9/2001.

[Stüc01] Design & Elektronik, Heft 2, Februar 2001, R. Stücka, Mit Methode zu gutem Code, S.57-59, WEKA Fachzeitschriften-Verlag GmbH

[SuSe89] W.-K. Su, C. Seitz. „Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm”; S. 38-43; Proceedings of the Multiconference on Distributed Simulation; 1989; Tampa/Florida

[Suth00] S. Sutherland. The IEEE Verilog 1364-2001 Standard – What's New, and Why You Need It. 9th Annual International HDL Conference and Exhibition, 2000

[SySC00] Synopsys. CoWare, Frontier Design. SystemC Version 1.0 User's Guide, 2000.

[TaBD03] C. Tapia, G. Bauer, M. Dornseiff. Effizienzsteigerung durch automatisierte Softwaretests, S.16, Automotive Electronics, Sonderausgabe von ATZ und MTZ und Automotive Engineering Partners, Sept. 2003

[TaSM95] Y. Tanurhan, S. Schmerler, K. D. Müller-Glaser. A Backplane Approach for Co-simulation in High-Level System Specification Environments. In: Proceedings of EURO-DAC ’95.

[Teic97] J. Teich, Digitale Hardware Software Systeme – Synthese und Optimierung. Springer, 1997.

[Tele03] Design & Elektronik, Telelogic: UML simulieren, S.69, 01/2003

[TeNg02] Y. M. Teo, Y. K. Ng. SPaDES/Java: Object-Oriented Parallel Discrete-Event Simulation. IEEE Proc. of the 35th Annual Simulation Symposium SS'02, 2002.

[TeTK96] Y. M. Teo, S. C. Tay, S. T. Kong. SPaDES: An Environment for Structured Parallel Simulation. Technical Report, TR20/96, Dep. of Information Systems and Computer Science, National University of Singapore, Okt. 1996.

[ThTh03] T. Thomsen. MISRA C – Normgerechtes oder effizientes C, S.84, Elektronik, Dez. 2003

[TiAg89] P. Tinker, R. Agre. „Object Creation, Messaging and State Manipulation in an Object Oriented Time Warp System”; S. 79-84; Proceedings of the Multiconference on Distributed Simulation; 1989

[Tind00] Auto & Elektronik, Sonderheft D 51994, Ausgabe 2/2000, OSEK auf der Überholspur, K. Tindell, S.78-82, Hüthig-Verlag

[Tind99] Auto & Elektronik, Sonderheft D 19067, Ausgabe 1/1999, K. Tindell, 99,99% reichen nicht!, S.121-123, Hüthig-Verlag

[TIPE00] ETAS GmbH & Co.KG, „ASCET Target Integration Package V4.0: C16x“, Benutzerhandbuch, Stuttgart, 2000

[TL2d02] dSPACE. More Than Just a Code Generator: TargetLink 2.0. dSPACE News Ausgabe 2, 2002, S.13-15.

[TLCR02] The MathWorks, Inc. Target Language Compiler: For Use with Real-Time Workshop, Reference Guide, V. 5, 2002

[TLdS01] dSPACE GmbH. TargetLink Production Code Generation Guide, for TargetLink 1.3; Paderborn, 2001

[TLdS02] dSPACE Inc., TargetLink von dSPACE – die Software-Lösung für die Erzeugung von hocheffizientem C-Code für Seriensteuergeräte, 2002

[Trau02] Elektronik Automotive, Dezember 2002, G. Trautzl: Nichts dem Zufall überlassen, S.60-62, WEKA Fachzeitschriften-Verlag GmbH

[TTCA02] ISO: ISO 11898-4: Time Triggered CAN. 2002

[Ulri78] H. Ulrich. „Unternehmungspolitik“, Band VI, Bern und Stuttgart, 1978

[VCad94] Verilog-XL Reference Manual, Cadence Design Systems Inc., 1994

[VDA99] Verbund der Automobilindustrie e.V., Auto 1999 Jahresbericht

[VDI03] VDI: Fluch oder Segen, S.6, Elektronik Automotive, Ausgabe 5/2003

[VDIS00] Verein Deutscher Ingenieure. Düsseldorf: VDI-Richtlinie 3633, Blatt 1: Simulation von Logistik-, Materialfluss- und Produktionssystemen, Grundlagen. VDI-Handbuch

Materialfluss und Fördertechnik, Band 8, Gründruck, Beuth 2000.

[Vöhr00] Auto & Elektronik, Sonderheft D 51994, Ausgabe 1/2000, K. D. Vöhringer, X-by-Wire – das Feeling bleibt!, S.122-126, Hüthig-Verlag

[VoWo88] F. Vogt, B. Wolfinger. Spezifikationssprachen und Modellierung für Verteilte Systeme; Tutorium, 18. Jahrestagung der Gessellschaft für Informatik, Universität Hamburg, Fachbereich Informatik, (Hamburg, 1988)

[VRef93] VHDL Language Reference Manual, ANSI/IEEE Standard 1076-1993, Juni 1993

[WaLa88] D. Wagner, E. Lazowska et al. „Techniques for Efficient Shared Memory Parallel Simulation"; Technical Report 88-04-05, Department of Computer Science, University of Washington/Seattle; 1988

[Walt01] Design & Elektronik, Heft 1, Januar 2001, K.-D. Walter, Abkürzung ins System, S.123-125, WEKA Fachzeitschriften-Verlag GmbH

[WARP99] D. E. Martin, T. J. McBrayer, R. Radhakrishnan et al. WARPED Version 1.02 Documentation. The University of Cincinnati, Computer Architecture Design Laboratory, Dept. of ECECS, 1999.

[WaTh85] R. Walker, D. Thomas. A Model o Design Representation and Synthesis. 22nd Design Automation Conference, Las Vegas, 1985

[WeFr01] Elektronik Automotive, Mai 2001, M. Wernicke, U. Freund: Verteilte Systeme im Automobil, S.79-83, WEKA Fachzeitschriften-Verlag GmbH

[WeFü01] Elektronik Automotive, Sept. 2001, H. Weiler, T. Führer et al. CAN wird erweitert zu TTCAN, S.74-77, WEKA Fachzeitschriften-Verlag GmbH

[Weis96] M. Weisser. Spezifikationen nehmen Gestalt an. In: Elektronik, Heft 13, Seiten 82 bis 87, 1996.

[WeMu88] J. West, A. Mullarney. ModSim: A Language for Distributed Simulation. Proc. of SCS MultiConference on Distributed Simulation, S. 155-159, 1988.

[Wenz02] S. Wenzel. Modellbildung in der ereignisdiskreten Simulation. ASIM Nachrichten, Nov. 2002. S.10-15.

[Wern01] Embedded Engineering, Ausgabe 1 2001, Februar 2001, M. Wernicke, Teile und herrsche, S.50-54, Hanser-Verlag

[WiHa89] F. Wieland, L. Hayley et al. „The Performance of a Distributed Combat Simulation with the Time Warp Operating System"; S. 35-50; Concurrency: Practice and Experience 1:1; 1989

[WiVi01] WindView for OSEKWorks User Guide, Version 2.0, Wind River System, Inc., 2001

[WoBr96] P. Wonnacott, D. Bruce. The APOSTLE Simulation Language: Granularity Control and Performance Data. Proc. of 10th IEEE/ACM/SCS Workshop on Parallel and Distributed Simulation (PADS'96), S. 114-123, Mai 1996.

[WoMB01] A. Wohnhaas, R. Moser, P. Brangs. Standardblockbibliothek für Steuergerätesoftwareentwicklung. SCS Frontiers in Simulation, Asim Fortschritte in der Simulationstechnik, Hrsg. Dr. Ingrid Bausch-Gall, S.55-79, 2001.

[Wüst63] K. D. Wüsteneck. Zur philosophischen Verallgemeinerung und Bestimmung des Modellbegriffes. Deutsche Zeitschrift für Philosophie, Heft 12, 1963.

[WyYo83] D. Wyatt, R. Young et al. „An Experiment in Microprocessor-based Distributed Digital Simulation"; S. 271-277; Proceedings of the Winter Simulation Conference;

1983

[xPCT02] The MathWorks. xPC TargetBox: PC-basiertes Hardware-System für das Rapid Prototyping. MATLAB Select Ausgabe 2, 2002, S.61.

[Zahi99] A. Zahir, (1999). Real-Time Magazine (pp. 25-32).

[Zeid02] B. Zeidman. Designing with FPGAs & CPLDs, CMP Books, 2002

[Zeig00] B. P. Zeigler, H. Praehofer, T. G. Kim. Theory of Modeling and Simulation, Second Edition. Academic Press, 2000.

[Zimm02] Elektronik Automotive, Sept. 2002, A. Zimmermann. Wandelbare Hardware, Rekonfigurierbarkeit von Hardwarefunktionen in Automotive-Systemen, S.78-83, WEKA Fachzeitschriften-Verlag GmbH

[Zöll02] M. Zöller: Entwicklung einer FPGA-basierten Ein-/Ausgabeschnittstelle mit PCI-Interface für Rapid-Prototyping-Anwendungen, Diplomarbeit, FZI/ESM, Karlsruhe, März 2002

[Zünd02] Design & Elektronik, Heft 10, Oktober 2002, D. Zündorf. Zusammen sind wir flexibler, S.32-34, WEKA Fachzeitschriften-Verlag GmbH

A.2 Referenzen im World Wide Web (WWW)

[Acce05] AccelChip, Inc. AccelChip DSP Design Using a True Top-Down Design Methodology, White Paper. http://www.accelchip.com/.

[AEE99] Architecture Electronique Embarquée. Projekt mehrerer Firmen zur Definition von Architekturen für Entwicklungsprozesse im Bereich der Automobilelektronik, 1999, http://aee.inria.fr/de/introduction.html

[AgHH99] R. Agreiter, M. Haberl, J. Hacker. Leistungsreduktion in Digitalschaltungen, März 1999,

http://www.morawek.at/Arbeiten/Leistungsreduktion/Leistungsreduktion.html#_Toc5

[AIEM04] Institut für Angewandte Informatik an der TU Dresden: Einführung in MATLAB, http://www.iai.inf.tu-dresden.de/tis/lehre/fachsoi/komplexpraktikum/matlab_beginner.pdf, 2004

[Alte05] Altera. http://www.altera.com

[Barr02] M. Barr. Special Report: Choosing an RTOS. Dez. 2002, Embedded.com, http://www.embedded.com/

[BEE03] BEE. http://bwrc.eecs.berkeley.edu/Research/BEE

[Bohn00] J. Bohn. „Software Development in Automotive Business“, Teil 2, 2000 http://ca.informatik.uni-oldenburg.de/~fraenzle/SES00/VL-SES00.html

[CAIn03] CANoe MATLAB Interface User Guide, Version 1.1, Vector Informatik GmbH, 2003, http://www.vector-informatik.de

[Came00] K. Camera. SF2VHD: A Stateflow to VHDL Translator, Master Thesis. University of California, Berkeley, Dept. of Electrical Engineering and Computer Sciene, Okt. 2000, http://bwrc.eecs.berkeley.edu/People/kcamera/sf2vhd/masters_thesis.pdf

[DASA00] DaimlerChrysler Aerospace (2000). Flugsysteme, Ausfallsichere Rechnersysteme, Fly-By-Wire; http://www.dasa.com/dasa/g/milair/fcs/sf_01.htm

[DESM05] DESMO-J. http://asi-www.informatik.uni-hamburg.de/desmoj/

[DMSO05] Defense Modeling and Simulation Office. http://www.dmso.mil

[dSPA05] dSPACE GmbH. http://www.dspace.de/ww/en/pub/home.htm/

[ETAS05] ETAS GmbH. http://en.etasgroup.com/

[FESM05] Forschungsbereich Elektronische Systeme und Mikrosysteme am FZI Forschungszentrum Informatik in Karlsruhe. 2001 http://www.fzi.de/esm/

[FlRa05] FlexRay. http://www.flexray.com, http://www.flexray-group.com

[Geiß00] R. Geißler. Anleitung zu Synthese und Optimierung, September 2000, http://mikro.e-technik.uni-ulm.de/vhdl/anl-deut.syn/html/

[GNU05] GNU, Free Software Foundation, http://www.gnu.org/, 2005

[GTTW05] Georgia Tech Time Warp, http://www.cc.gatech.edu/computing/pads/tech-parallel-gtw.html

[Hitz99] M. Hitz. „Kriterienkatalog“, Universität Wien, 1999 http://www.ifs.uni-linz.ac.at/ifs/teaching/offer/ss99/umlKriterienkatalog.html

[HLAM05] High Level Architecture. http://www.dmso.mil

[IIG99] Institut für Informatik und Gesellschaft (1999). Universität Freiburg; http://modell.iig.uni-freiburg.de/forschung/swengineering/prj-zuverlaessig.html

[ILog05] I-Logix, Inc. http://www.ilogix.com/

[Infi05] Infineon. http://www.infineon.com

[ISCA89] ISCAS'89 Suite zur digitalen Schaltungssimulation. CAD Benchmarking Lab, NCSU, ISCAS'89 Benchmark Information, http://www.cbl.ncsu.edu/www/CBL_Docs/iscas89.html

[Jama05] aicas GmbH. JamaicaVM, http://www.aicas.com

[JaSp98] JavaSpaces Specification. http://java.sun.com/products/javaspaces

[Java04] WebGain, Inc. http://www.webgain.com/products/java_cc/

[Jone01] D. W. Jones. Iowa Logic Simulator User's Manual Version 10. University of Iowa, Department of Computer Science, Feb. 2001, ftp://ftp.cs.uiowa.edu/pub/jones/ logic-sim/manual.ps

[LIN05] LIN Local Interconnect Network. http://www.lin-subbus.de, http://www.LIN-sub-bus.org

[Math05] The MathWorks, Inc. http://www.mathworks.com/

[MeBa99] R. A. Meyer, R. Bagrodia. PARSEC User Manual Release 1.1. Sept. 1999, UCLA Parallel Computing Laboratory, http://pcl.cs.ucla.edu/projects/parsec/

[Ment05] Mentor Graphics. http://www.mentor.com/germany

[MISR05] MISRA. http://www.misra.co.uk

[MoTe05] Model Technology. http://www.model.com/

[MOST] MOST Media Oriented System Transport. http://www.mostcooperation.com, http://www.mostnet.de

[Moto05] Motorola. http://www.motorola.com/General/index.html

[MPI05] MPI. http://www.mpi-forum.org/

[OMG05] OMG. http://www.omg.org/

[OPCO05] OPENCORES. http://www.opencores.org/

[PDEC05] ProDesign Electronic & CAD-Layout GmbH, 2005, http://www.prodesigncad.com/index.htm

[Ptol03] C. Hylands, E. A. Lee, J. Liu, et. al. Ptolemy II Heterogeneous Concurrent Modeling and Design in Java, Document Version 3.0, Vol. 1-3. Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Juli 2003, http://ptolemy.eecs.berkeley.edu/ptolemyII/

[PVM05] PVM. http://www.epm.ornl.gov/pvm/pvm_home.html

[RTAI00] DIAPM RTAI Programming Guide 1.0, Sep. 2000, http://www.aero.polimi.it/~rtai/documentation/reference/rtai_prog_guide.pdf

[RTAI05] DIAPM RTAI, Dipartimento di Ingegneria Aerospaziale – Politecnico di Milano, Real Time Application Interface, http://www.aero.polimi.it/~rtai/, 2005

[RTEC02] The MathWorks, Inc. „Real-Time Workshop Embedded Coder, Task Management“, 2002 http://www.mathworks.co.uk/access/helpdesk/help/toolbox/ecoder/ecode10c.shtml

[RTSJ05] Real-Time Specification for Java. http://www.rtj.org

[Simk05] Simkit. http://diana.gl.nps.navy.mil/Simkit/

[SPaD01] SPaDES/Java Simulation Library User Manual Version 0.1. Dept. of Computer Science, National University of Singapore, Feb. 2001, http://www.comp.nus.edu.sg/~pasta/spades-java/spadesJava.html

[SPEE03] SPEEDES User's Guide, Document Number S024, Rev. Number 4, April 2003, Metron, Inc., http://www.speedes.com/docs/SpUG.pdf

[Syst05] SystemC. http://www.systemc.org

[TaWi04] K. Taubert; W. Wiedl: Differentialgleichungen mit Simulink, http://www.rrz.uni-hamburg.de/RRZ/W.Wiedl/Skripte/Simulink/index.html

[Tele05] Telelogic. http://www.telelogic.com/

[TTP05] TTP Time Triggered Protocol. http://www.tttech.com

[VAMS05] VHDL-AMS. http://www.vhdl.org/analog/

[Vdeu97] Deutsches Vorgehensmodell. 1997 http://www.v-modell.iabg.de/

[Vect05] Vector Informatik GmbH. http://www.vector-informatik.de

[Veri05] Verilog. http://www.verilog.com

[VHDLAM] VHDL-AMS. http://www.vhdl-ams.com

[VHDLor] VHDL International Home page. http://vhdl.org

[VMod97] V-Modell – Entwicklungsstandard für IT-Systeme des Bundes. Vorgehensmodell Kurzbeschreibung. 1997. http://www.v-modell.iabg.de/vm97.htm

[VMXT05] V-Modell XT des Bundes, http://www.kbst.bund.de/-,279/V-Modell.htm

[Xili05] Xilinx. http://www.xilinx.com

[XiSG03] Xilinx System Generator for DSP version 2.2, User Guide, Xilinx Blockset Reference Guide, 2003, http://www.xilinx.com/

[XiSp01] Spartan-II 2.5V FPGA Family: Functional Description, Xilinx, 2001, http://www.xilinx.com/

[XiSp02] Spartan-II 2.5V FPGA Family: DC and Switching Characteristics, Xilinx, 2002, http://www.xilinx.com/

[XiVi01] Xilinx Virtex-II Platform FPGA Handbook, v1.3, Dez. 2001, http://www.xilinx.com/

[Zimm03] W. Zimmermann: Hardware Input / Output Library for Matlab / Simulink, http://www.it.fht-esslingen.de/~zimmerma/software/IOlib.htm

A.3 Eigene Beiträge

[DrBS97] R. Dreier, O. Barheine, T. Stingl et al. Experiences with a Java/C-based MPEG-4 prototype (M2868); ISO/IEC JTC1/SC29/WG11, Fribourg, Schweiz, Okt. 1997.

[DrDZ03] R. Dreier, G. Dummer, G. Zhang et al. Partitioning and FPGA-based co-simulation of Statecharts; Proc. 15th SCS European Simulation Symposium and Exhibition (ESS), Niederlande, Delft, Okt. 2003, S. 500-506.

[Drei00] R. Dreier. Implikation zwischen System-Partitionierung und Prozess-Synchronisation bei der verteilten Simulation; Seminarvortrag am Institut für Technik der Informationsverarbeitung, Karlsruhe, Juli 2000.

[Drei01] R. Dreier. Performanceaspekte bei verteilter, ereignisgesteuerter Simulation; Seminarvortrag am Institut für Technik der Informationsverarbeitung, Karlsruhe, Juli 2001.

[Drei02] R. Dreier. Analyse ereignisdiskreter elektronischer Systeme auf verschiedenen Abstraktionsebenen mittels verteilter Simulation unter Berücksichtigung von Partitionierungs- und Performanceaspekten; Seminarvortrag am Institut für Technik der Informationsverarbeitung, Karlsruhe, Juli 2002.

[Drei03] R. Dreier. PC-, FPGA- und Mikrocontroller-basierte Co-Simulation elektronischer Systeme; Seminarvortrag am Institut für Technik der Informationsverarbeitung, Karlsruhe, Juli 2003.

[Drei04] R. Dreier. Ein Framework zur verteilten Ausführung eingebetteter elektronischer Systeme; Seminarvortrag am Institut für Technik der Informationsverarbeitung, Karlsruhe, Juni 2004.

[Drei99] R. Dreier. Architektur einer multimedialen Applikation am Beispiel eines Videotelefoniesystems; Seminarvortrag am Institut für Technik der Informationsverarbeitung, Karlsruhe, April 1999.

[DrMG03] R. Dreier, K. D. Müller-Glaser. Integration of heterogeneous execution environments for co-simulation; Proc. 12th IASTED International Conference on Applied Simulation and Modelling (ASM), Spanien, Marbella, Sept. 2003, S. 125-129.

[DrMG04] R. Dreier, K. D. Müller-Glaser. Requirements for real time operating systems and features of operating systems implementing the OSEK/VDX standard API; 12th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Poster Session, Volendam, Niederlande, Okt. 2004, S.32-35.

[DrSc00] R. Dreier, S. Schmerler. Evaluating study on the I-Logix tool chain with regard to code generation; 8. Deutsches STATEMATE Anwenderforum, Aschheim, Mai 2000.

[DrSM01] R. Dreier, E. Sax, K. D. Müller-Glaser. Requirements and state of the art of automated software development for embedded systems based on CASE tools; Proc. IEEE Design, Automation and Test in Europe, Designers' Forum (DATE), München, März 2001, S. 44-48.

[DrSW98] R. Dreier, T. Stingl, G. Wendt et al. Structure and interaction of two IM1 related demo applications (M3853); ISO/IEC JTC1/SC29/WG11, Dublin, Irland, Juli 1998.

[DrWM04] R. Dreier, G. Wendt, K. D. Müller-Glaser. A hybrid, CAN based distributed execution environment for electronic systems; Proc. 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), Cambridge, MA, USA, Nov. 2004, Nr. 439-032.

[DrWM05] R. Dreier, G. Wendt, K. D. Müller-Glaser. Optimistic distributed simulation of hybrid system models; Proc. 16th IASTED International Conference on Modelling and Simulation (MS), Cancun, Mexiko, Mai 2005, S. 268-276.

[StBD97] T. Stingl, O. Barheine, R. Dreier et al. Description of a prototypical MPEG-4 infotainment application (M2861); ISO/IEC JTC1/SC29/WG11, Fribourg, Schweiz, Okt. 1997.

[StDB97] T. Stingl, R. Dreier, K. Bahr et al. MPEG-4 event model and scene update formats (M2448); ISO/IEC JTC1/SC29/WG11, Stockholm, Schweden, Juli 1997.

[StDB98] T. Stingl, R. Dreier, O. Barheine. Experiences with the development of an MPEG-4 oriented PC multimedia application; Proc. SPIE Multimedia Systems and Applications Vol. 3528, Boston, MA, USA, Nov. 1998, S. 50-59.

[StDr99] T. Stingl, R. Dreier. Modellierung und Simulation mit Zustandsautomaten – Schwerpunkt BetterState; ASIM Arbeitsgemeinschaft Simulation, Fachtreffen, Aachen, März 1999.

A.4 Betreute wissenschaftliche Arbeiten

[Bukh03] A. Bukhari. Bachelorarbeit: Integration ereignisgesteuerter Simulatoren in ein Co-Simulations-Framework, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 25. Aug. 2003.

[Dumm03] G. Dummer, Diplomarbeit: Entwicklung einer FPGA-basierten Simulationsbeschleunigung für ereignisdiskrete Systeme; Universität Karlsruhe, Institut für Technik der Informationsverarbeitung und Institut für Rechnerentwurf und Fehlertoleranz, 8. Aug. 2003.

[Grau04] X. Grau. Masterarbeit: Model-based rapid prototyping of a CAN-connected automotive ECU, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 28. Apr. 2004.

[Para04] E. Paranthaman. Studienarbeit: Enhancement of the VHDL code generator JVHDLGen regarding support of Stateflow, hybrid systems and distributed simulations, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 23. Mai. 2004.

[Ritt99] C. Ritter. Diplomarbeit: Konzeption und Implementierung der Server-Software für ein ISDN-Videokonferenzsystem, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 22. Dez. 1999.

[Waib99] M. Waibel. Diplomarbeit: Konzeption und Implementierung von Codec- und LAN-Softwarekomponenten für ein Videotelefoniesystem, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 3. Dez. 1999.

[Wen04a] G. Wendt. Studienarbeit: Implementierung und Analyse einer CAN-basierten verteilten Ausführungsumgebung für hybride elektronische Systeme, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 23. Apr. 2004.

[Wen04b] G. Wendt. Diplomarbeit: Entwurf und Realisierung eines Verfahrens zur optimistischen Co-Simulation hybrider elektronischer Systeme, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 27. Nov. 2004.

[Zhan03] G. Zhang. Diplomarbeit: Partitionierung von Statecharts zur effizienten verteilten Simulation, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung und Institut für Rechnerentwurf und Fehlertoleranz, 16. Sept. 2003.

B Abbildungsverzeichnis

C Tabellenverzeichnis

D Implementierung

Anhang D enthält Details zu wesentlichen Implementierungsaspekten des INTERACT-Frameworks.

D.1 Mdl-Parser und -Generator

Der Parser wandelt eine Modelldatei in entsprechende Java-Objekte um, und der Generator schreibt die Modell-Objekte wieder in eine Modelldatei zurück. Nachdem eine Modelldatei in interne Java-Objekte umgesetzt wurde, können diese für die weitere Bearbeitung verwendet werden. Z.B. wird ein `State`-Objekt mit JStateGen mehrfach kopiert.

Eine Modelldatei enthält Schlüsselworte und Parameter-Wert-Paare, um ein Modell zu beschreiben. Die Modelldatei beschreibt die Modellkomponente anhand der Hierarchie. Die Struktur einer Modelldatei ist wie folgt:

```
Model {
    <Model Parameter Name> <Model Parameter Value>
    …
    BlockParameterDefaults {
           Block {
                  <Block Parameter Name> <Block Parameter Value>
                  …
           }
    }
    AnnotationDefaults {
           <Annotation Parameter Name> <Annotation Parameter Value>
    …
    }
    System {
           <System Parameter Name> <System Parameter Value>
           …
           Block {
                  <Block Parameter Name> <Block Parameter Value>
                  …
           }
           Line {
                  <Line Parameter Name> <Line Parameter Value>
                  …
                  Branch {
                         <Branch Parameter Name> <Branch Parameter Value>
                         …
                  }
           }
           Annotation {
                  <Annotation Parameter Name> <Annotation Parameter Value>
                  …
           }
    }
}
```

Der `Model`-Teil definiert Parameter eines Models. Der `BlockParameterDefaults`-Teil enthält Defaultwerte für Blöcke des Modells. Der `AnnotationDefaults`-Teil enthält die Layoutinformationen über Bemerkungen zum Modell. Der `System`-Teil enthält Parameter, die das Modell beschreiben. Es besteht aus mehreren `Block`-, `Line`- und `Annotation`-Teilen im System. Wenn eine `Line` eine Verzweigung hat, enthält sie zusätzlich einen `Branch`-Teil. Für das Modell einer Logikschaltung werden ein Logikgatter als ein `Block`-Teil, und seine Verbindungen zu anderen Gattern als mehrere `Line`-Teile gespeichert. Ein Statechart wird in Simulink als ein `Stateflow`-Block repräsentiert. Die Informationen der Stateflow-Machine werden in einem eigenen Teil der Modelldatei im folgenden Format gespeichert:

```
Stateflow {
    machine {
           ...
    }
    chart {
           ...
    }
    state {
           ...
    }
    junction{
           ...
    }
    transition{
           ...
           src {
                  ...
           }
           dst {
                  ...
           }
           ...
    }
    event {
           ...
    }
    data {
           ...
    }
}
```

Jeder Teil besteht aus Paaren eines Parameters und eines Werts. Der Parameter `id` tritt immer an der ersten Zeile jedes Teils auf. Jeder Teil besitzt seine eindeutige Identifikation innerhalb des Modells und ist in der ersten Linie jedes Teils enthalten. Dadurch kann ein Teil als Referenz von anderen Teilen benutzt werden. Ein `transition`-Teil enthält z. B. einen `src`-Teil und einen `dst`-Teil, in denen jeweils die Identifikationen des Quellzustands und Zielzustands enthalten sind. Die mit einem Zustand verbundenen Aktionen sind im Parameter `labelString` des `state`-Teils gespeichert.

Klassendiagramm des mdl-Parsers und Generators: Bild 220 zeigt das Klassendiagramm des mdl-Parsers und -Generators mit den für die Konvertierung zwischen einer Modelldatei und ihrem `Model`-Objekt relevanten Klassen.

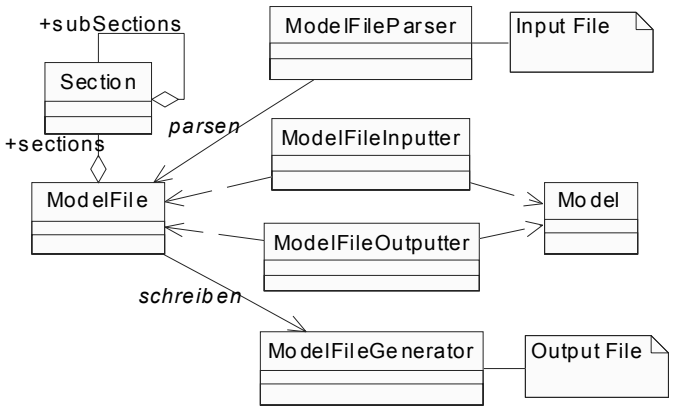


Bild 220: Parsen und Schreiben von Modelldateien

Die eigentliche Struktur eines `Model`-Objekts ist in Bild 222 abgebildet, und der Aufbau der Erweiterung von JStateGen zur skriptbasierten, flexiblen Generierung von Statecharts in Bild 221[73].

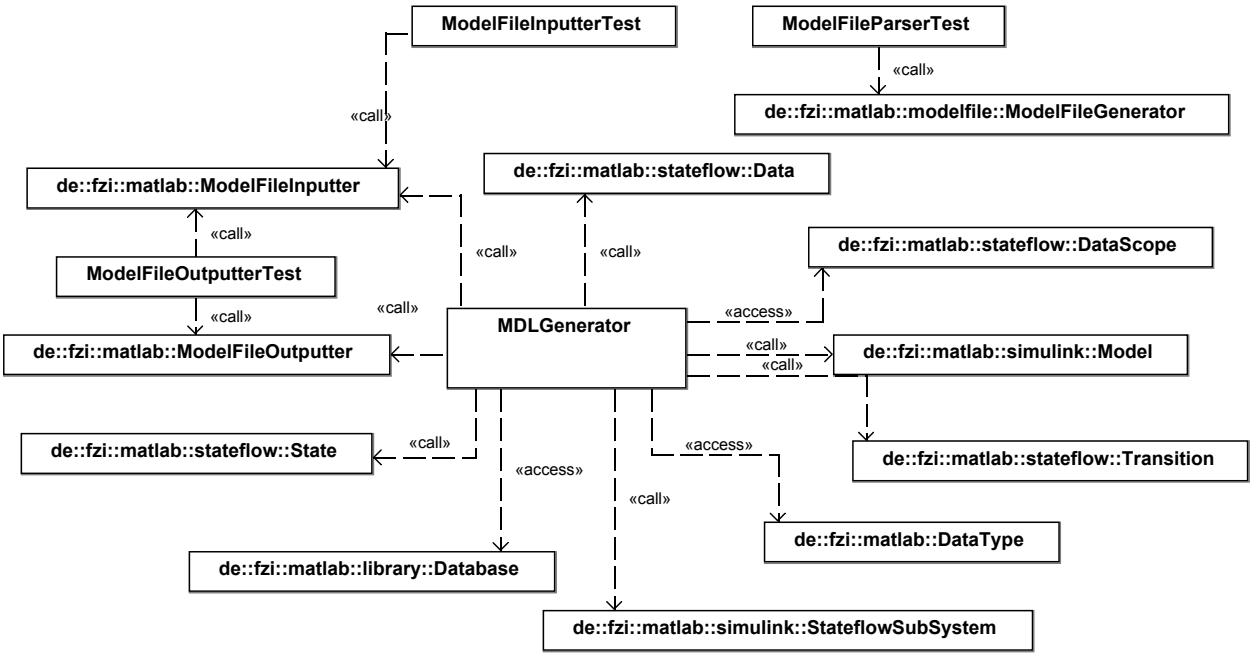


Bild 221: Klassendiagramm des Werkzeugs JStateGen

Eine Modelldatei besteht aus mehreren Teilen, die durch die Klasse `ModelFileParser` in die entsprechenden Klassen `ModelFile` und `Section` geparst werden. Ein Teil kann wieder mehrere Subteile haben. Jeder Typ der jeweiligen Teile hat einen Namen und enthält bestimmte Parameter. Ein Parameter hat einen Namen und Datentyp wie z. B. `integer`, `float`, `logic`, `string`, `list` und `enumeration`. Diese Informationen werden in der XML-Datei `library.xml` gespeichert. Die Klassen `ModelFileInputter` und `ModelFileOutputter` benutzen diese Informationen, um ein `ModelFile`-Objekt in ein `Model`-Objekt oder umgekehrt umzuwandeln.

Ein `ModelFile`-Objekt kann durch die Klasse `ModelFileGenerator` wieder in einer Modelldatei zurückgeschrieben werden. Weil ein Zustand Aktionen beinhalten kann, und eine Transition Ereignisse, Bedingungen, Bedingungsaktionen und Transitionsaktionen enthalten kann, wird ein spezieller Parser `ActionLanguageParser` für sie erzeugt. Sowohl der `ModelFileParser` als auch der `ActionLanguageParser` werden mit Hilfe eines von Webgain [Java04] angebotenen kostenlosen Parser-Generators `JavaCC` erstellt. Die Syntax sowie die für eine bestimmte Syntax entsprechende Java-Methode werden zuerst in einer Datei mit dem Suffix `jj` definiert. Diese Datei wird als Eingabe des Befehls `javacc` benutzt, und der Parser wird als Ausgabe erzeugt. Wenn neue Syntax in dieser Datei hinzugefügt wird, muss die entsprechende Klasse für den Parser erneut erstellt werden.

Bild 222 zeigt das Klassendiagramm eines Modells. Ein Modell-Objekt enthält ein System-Objekt. Dieses besteht aus mehreren `Block`- und `Line`-Objekten. Ein `Block`-Objekt repräsentiert einen Funktionsblock in Simulink. Ein `Line`-Objekt verbindet einen `Source`-Block und einen `Destination`-Block. Weil eine Verbindung mehrere Zielblöcke haben kann, wird der einzelne Zielblock in der Modelldatei als ein Subblock des `Line`-Blocks gespeichert. In diesem Fall werden mehrere `Line`-Objekte für einen `Line`-Block erzeugt. Stateflow wird als einen speziellen Block behandelt. Ein `StateflowSubSystem`-Objekt enthält ein `State`-Objekt. Ein `Transition`-Objekt hat zwei

[73] In den folgenden Klassendiagrammen bezeichnet „call“ Methoden- und „access“ Datenzugriffe; „implementation“ bezeichnet die Implementierung von Schnittstellen (interfaces).

Endpunkte von `Connectable`-Objekten. Die `Connectable`-Klasse wird durch die Klasse `State` und die Klasse `Junctor` implementiert. Die Klasse `State` ist zusätzlich eine Unterklasse der Klasse `Scope`. Ein `Scope`-Objekt enthält mehrere `Connectable-`, `Transition-`, `Data-` und `Event`-Objekte. Die Beziehungen zwischen den Klassen `State`, `Scope` und `Connectable` entspricht der Hierarchie von Zuständen. Ein `State`-Objekt kann zusätzlich mit ihm verbundene Zustandsaktionen haben.

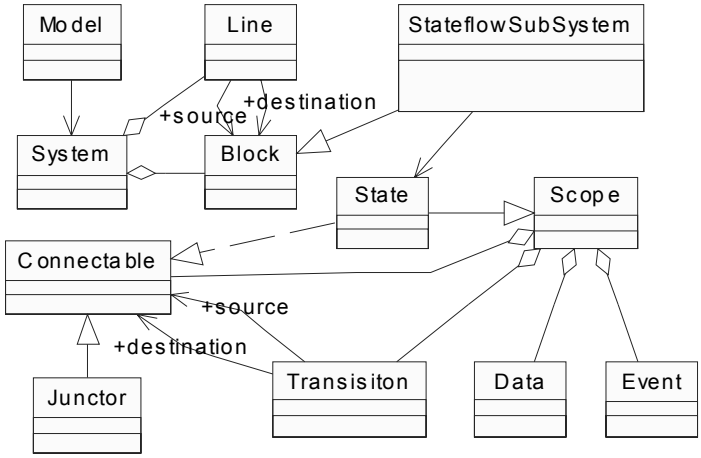


Bild 222: Klassendiagramm eines Modells

Für die Partitionierung von Logikschaltungen und Statecharts wird der Parser wie folgt erweitert:

- Parameter für Logikblöcke: Um ein Modell mit Logikblöcken zu parsen, müssen die Informationen über einen Logikblock in die Datei `library.xml` hinzugefügt werden. Ein Logikgatter wird in der Modelldatei als ein Block mit dem Typ `Logic` beschrieben. Der Parameter Operator eines Logic-Blocks hat den Typ `enumeration` mit dem Wert OR, NOT, XOR, NAND oder NOR.
- Annotation-Teil: Weil der `Annotation`-Teil im vorhandenen Parser nicht betrachtet wird, muss eine neue Klasse `Annotation` hinzugefügt werden. In der Klasse `ModelFileInputter` und `ModelFileOutputter` muss eine zusätzliche Methode für das Lesen und Schreiben von `Annotation`-Teilen hinzugefügt werden (Bild 223). Der `Annotation`-Teil wird im folgenden Abschnitt für die Informationen über die Schnittstelle zwischen Partitionen benutzt, weil man nur in diesem Teil selbst benötigte Informationen definieren kann und die Syntax der anderen Teile einer mdl-Datei vorgeschrieben ist.
- Parsen der impliziten Ereignisse: In die Datei `ActionLanguage.jj` wird die Grammatik zur Definition der impliziten Ereignisse eingefügt. Zwei Klassen `ImplicitEvent` und `ImplicitEventSpecifier` werden zusätzlich hinzugefügt.
- Defaultwert eines Parameters: Für den Teil `BlockParameterDefaults` in einer mdl-Datei wird eine Klasse `BlockParameterDefaults` hinzugefügt, um den Defaultwert eines Parameters für einen Block wie z. B. die Verzögerung eines `Transport-Delay`-Blocks zu setzen.
- Vervielfachung eines Charts mit AND-Dekomposition: Wenn der oberste Zustand eine AND-Dekomposition hat, werden nicht nur der erste Unterzustand, sondern alle Unterzustände dieses Zustands kopiert.
- Zusammenfassung von Charts: Wenn mehrere Charts die gleiche Modellstruktur haben, können sie als nebenläufige Zustände in ein Chart zusammengefasst werden. Variable und Ereignisse im einzelnen Chart werden in diesem Chart hinzugefügt.

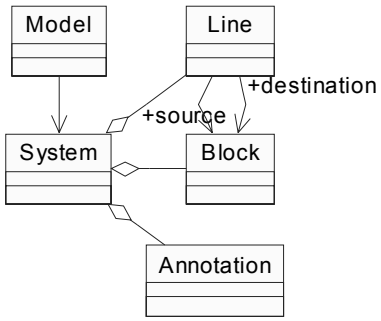


Bild 223: Erweiterung der Annotation-Klasse

D.2 K-Way Partitionierung

Weil der K-FM-Algorithmus von Sanchis für beliebige Partitionen und Knoten eines Graphen mit verschiedenen Gewichten benutzt werden kann, wird dieser Algorithmus für die Partitionierung von Logikschaltungen implementiert (Bild 224).

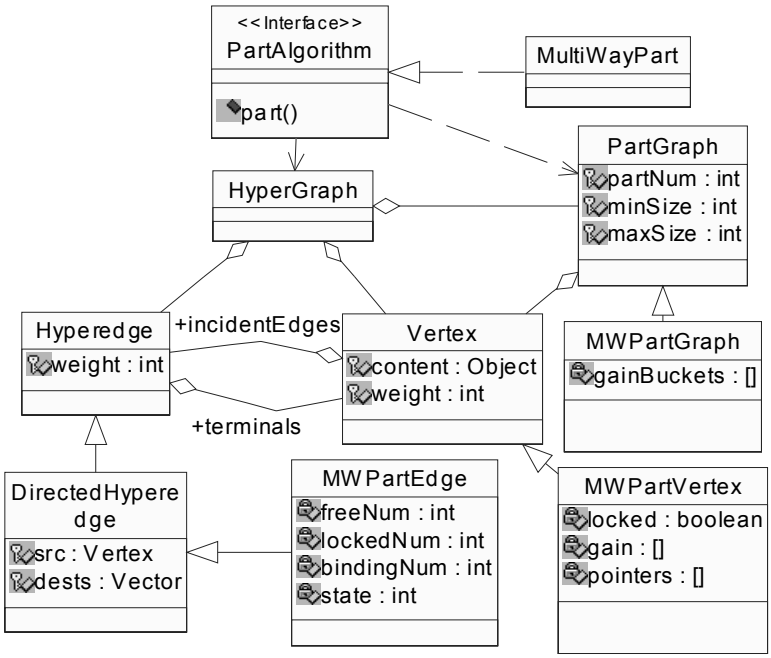


Bild 224: Klassendiagramm des Partitionierungsalgorithmus

Das Attribut `minSize` und `maxSize` eines `PartGraph`-Objekts gibt die Nebenbedingung der Balance an. Die Klasse `MultiWayPart` implementiert die Methode `part()` der Schnittstelle `PartAlgorithm`. Diese Methode hat zwei Eingaben: ein `Hypergraph`-Objekt und ein `Properties`-Objekt. Das `Hypergraph`-Objekt ist der Hypergraph zur Partitionierung. Das `Properties`-Objekt gibt die für den Partitionierungsalgorithmus benötigten Parameter an. Für den K-FM-Algorithmus müssen die Parameter `PartCount` (Anzahl der Partitionen), `GainLevel` (Anzahl der Gain-Levels), `BalanceVariation` (Variation der Balance), `Seed` (zum Erzeugen einer zufälligen Anzahl für die initiale Partitionierung) und `Acyclic` (zur Auswahl einer der Algorithmen K-FM und K-FM azyklisch (K-AFM)) gegeben werden. Die Klassen `MWPartGraph`, `MWPartEdge` und `MWPartVertex` enthalten die Datenstrukturen für den K-FM-Algorithmus.

Außerhalb der Methode `part()` werden andere Methoden in der Klasse `MultiWayPart` wie folgt definiert:

- `part()`: führt den nächsten Durchlauf durch Aufruf der Methode `doPass()` aus, wenn die Anzahl von Schnitten im letzten Durchlauf abgenommen hat. Sonst wird dieser Vorgang unterbrochen, und die Schnitte zurückgegeben.
- `getStartPart()`: eine initiale Partitionierung wird berechnet. Die Knoten werden zufällig in verschiedene Partitionen eingeteilt. Die Größe einer Partition muss die gegebene Bedingung der Balance erfüllen.
- `initPart()`: weist jeden Knoten anhand der aktuellen Partitionierung der entsprechenden Partition zu.
- `updateGain()` und `reverseUpdateGain()`: aktualisiert die Gain-Struktur nach dem Verschieben eines Knotens.
- `doPass()`: bestimmt die Verschiebereihenfolge von Knoten mit dem besten Gewinn in einem Durchgang, und gibt die nächste Partitionierung zurück. Bei der Auswahl des nächsten zu verschiebenden Knotens wird die Bedingung der Balance geprüft.
- `makeMove()`: verschiebt einen Knoten von einer Partition in eine andere.

Der K-AFM-Algorithmus benutzt die gleichen Methoden wie der K-FM-Algorithmus. Es wird nur eine neue Methode in der Klasse `MultiWayPart` hinzugefügt, um eine azyklische Anfangspartitionierung zu gewinnen. In der Methode `doPass()`werden einige Veränderungen gemacht:

- `getAcyclicStartPart()`: erzeugt eine azyklische initiale Partitionierung. Die Knoten werden zuerst topologisch sortiert, dann werden sie nach ihren Reihenfolgen in einige Partitionen eingeteilt. Dieses Verfahren kann eine azyklische Partitionierung garantieren. Bei der topologischen Sortierung von Knoten wird der Tiefensuche-Algorithmus verwendet[74].
- `doPass()`: bei der Bestimmung der Verschiebereihenfolge von Knoten wird eine zusätzliche Bedingung (Zyklusfreiheit) zwischen Partitionen geprüft. Die Klasse `GraphBuilder` erzeugt einen Abhängigkeitsgraph von Partitionen. Bei der Überprüfung der Zyklusfreiheit des Abhängigkeitsgraphen wird ebenfalls der Tiefensuche-Algorithmus verwendet.

D.3 Modell-Partitionierung im ISCAS-Format

Ein Modell einer Logikschaltung wird zuerst in einen Hypergraph umgesetzt. Dann kann der implementierte K-FM-Algorithmus auf diesem Hypergraph verwendet werden. Nach der Partitionierung werden Teilmodelle mit Informationen über die Schnittstelle zwischen diesen erstellt. Um ein Modell im ISCAS-Format zu partitionieren, muss es zuerst ins mdl-Format konvertiert werden.

Erstellung eines Hypergraphen für eine Logikschaltung: In Bild 225 ist das Klassendiagramm für die Partitionierung von Logikschaltungen und Statecharts dargestellt. Weil der K-FM-Algorithmus auf dem mdl-Parser basiert, muss ein Modell im mdl-Format vorliegen. Die Methode der Klasse `GraphBuilder` wandelt ein `Model`-Objekt für eine Logikschaltung in ein `Hypergraph`-Objekt um. Umgekehrt konvertiert die Klasse `ModelBuilder` die `PartGraph`-Objekte nach der Partitionierung eines Hypergraphs in `Model`-Objekte. Das `Block`-Objekt für ein logisches Gatter wird als Inhalt in einem `Vertex`-Objekt gespeichert. Ein `Hyperedge`-Objekt wird anhand der `Line`-Objekte mit dem gleichen Quellblock erzeugt. Für eine gerichtete Hyperkante werden der Quellknoten und die Zielknoten festgelegt. Ein Hypergraph kann gerichtet oder ungerichtet sein, je nachdem ob seine Kanten `Hyperedge`-Objekte oder `DirectedHyperedge`-Objekte sind. Sowohl `Hyper-`

[74] Depth first search (DFS) ist ein Verfahren zum Durchlaufen eines Graphen, bei dem jeder Knoten genau einmal besucht wird. Der Name Tiefensuche erklärt sich daher, dass das Verfahren zunächst in die Tiefe und erst dann in die Breite des Graphen führt. Das heißt: Man geht von einem Knoten, der gerade besucht wird, zu einem noch nicht besuchten Nachbarknoten und setzt dort den Algorithmus rekursiv fort. Sowohl der Algorithmus zum Berechnen von stark zusammenhängenden Komponenten als auch der Algorithmus zur topologischen Sortierung kann durch DFS-Verfahren implementiert werden.

edge-Objekt als auch ein Vertex-Objekt hat ein Attribut Gewicht. Das Gewicht eines Logikgatters wird auf eins gesetzt.

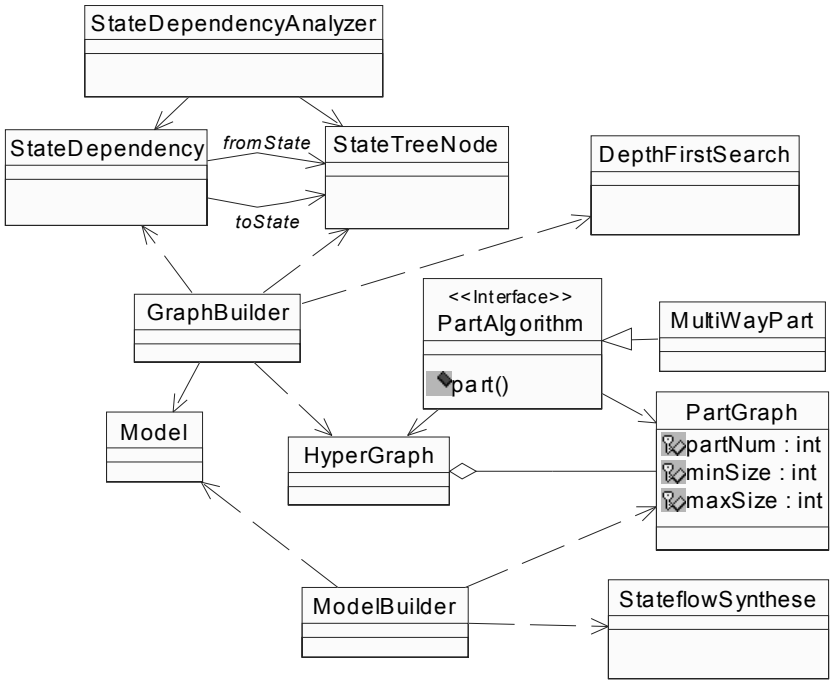


Bild 225: Klassendiagramm des Partitionierungsverfahrens

Die Schnittstelle zwischen Teilmodellen: Nach der Partitionierung werden Teilmodelle als mdl-Datei gespeichert. Um die entsprechenden INPUT_NET- und OUTPUT-Blöcke von ISCAS aus Teilmodellen im mdl-Format zu erzeugen, wird ein Paar, bestehend aus Inport- und Outport-Block, für jeden Schnitt in Teilmodellen hinzugefügt. Bild 226 stellt diese Veränderung der Schaltung dar, nachdem die Verbindung zwischen dem Block Gen2 und dem Block Nor durch einen Schnitt getrennt wird. Die zwei Blöcke Out1 und In1 werden hinzugefügt.

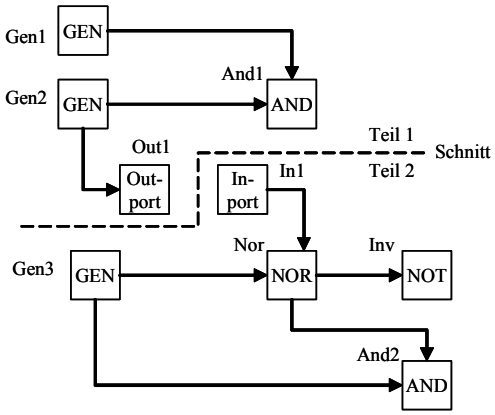


Bild 226: Schaltung nach Partitionierung

Die Schnittstelle zwischen Teilmodellen wird durch die Netzangabe des OUTPUT_NET-Blocks in JLogSim definiert. Um die Netzangabe eines OUTPUT_NET-Blocks später aus Teilmodellen zu gewinnen, wird diese bei der Partitionierung als Kommentar in der mdl-Datei gespeichert.

Die Netzangabe eines OUTPUT_NET-Blocks ist wie folgt definiert:

```
Annotation {
    Name „OUT_INFO:Output Block Name:Submodel Name/Input Block Name“
    Position [x,y]
}
```

Der Inhalt eines Kommentars wird mit dem Parameter `NAME` gespeichert. `OUT_INFO` ist das Schlüsselwort für die Netzangabe. Die Netzinfo besteht aus drei Teilen, dem Namen des `OUTPUT`-Blocks, dem Namen des Submodells, in dem der entsprechende `INPUT`-Block enthalten ist, und dem Namen des `INPUT`-Blocks. Für die Partitionierung muss die folgende Netzangabe im Teilmodell `Teil1` hinzugefügt werden:

```
„OUT_INFO:Out1:Teil2/In1“
```

Umsetzung einer mdl-Datei in Objekte von JLogSim: Um ein Modell im mdl-Format mit JLogSim zu simulieren, muss die Modelldatei zuerst in Objekte von JLogSim umgesetzt werden (Bild 228 und Bild 229). JLogSim führt eine Verzögerung (Delay) für jeden Block ein. Allerdings sind in MATLAB keine primitiven Logikblöcke mit dem Parameter Delay vorhanden. Ein Logikblock mit Delay wird durch einen `Subsystem`-Block von MATLAB beschrieben, der sich aus einem `Logik`-Block, einem `Transport-Delay`-Block und einigen `Inport-` und `Outport`-Blöcken zusammensetzt (Bild 227).

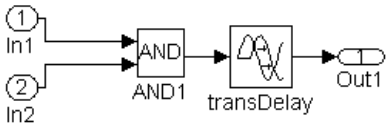


Bild 227: Subsystem-Block mit Verzögerung

Die Klasse `ISCASBlock` wird zur Konvertierung eines `Subsystem`-Blocks in ein `JLBlock`-Objekt hinzugefügt. Ein `ISCASBlock`-Objekt enthält ein `Subsystem`-Object von MATLAB. Um die Attributwerte eines `JLBlock`-Objekts von JLogSim zu setzen, stellt die Klasse `ISCASBlock` einige Methoden zur Verfügung.

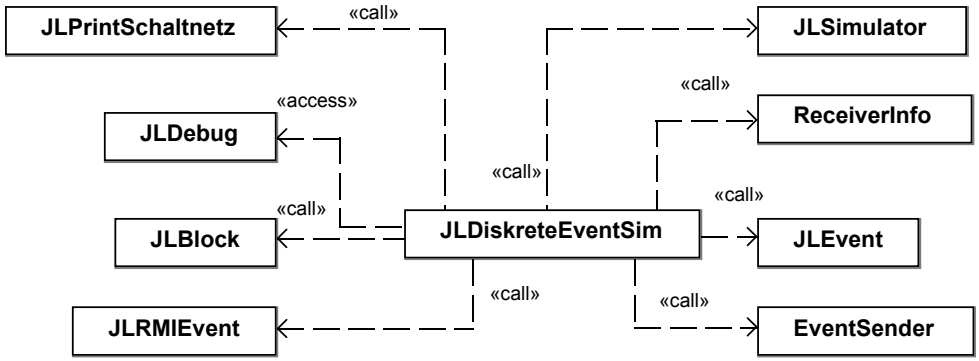


Bild 228: Klassendiagramm des Werkzeugs JLogSim (1)

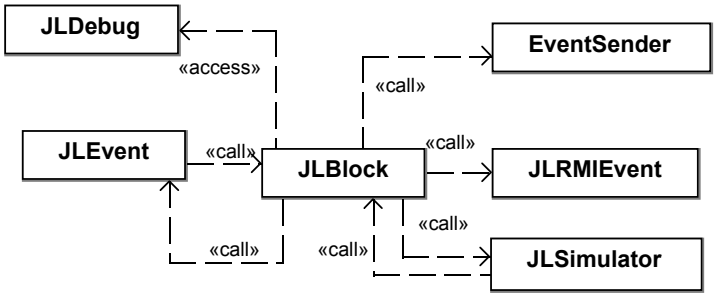


Bild 229: Klassendiagramm des Werkzeugs JLogSim (2)

Die Methode `getType()`gibt den Typ des `Logik`-Blocks zurück. Die Methode `getDelay()` gibt den Delay des `Delay`-Blocks zurück. Die Liste von Ein- und Ausgangsblöcken eines `JLBlock`-Objekts wird anhand der `Line`-Objekte von mdl erstellt. Die Klasse `ModelConverter` konvertiert alle `Block`-Objekte eines Modells in `JLBlock`-Objekte von JLogSim. Die Verbindungen zwischen

ihnen werden mit Hilfe von `Line`-Objekten erstellt. Die beiden Klassen werden in JLogSim integriert, damit JLogSim Modelle entweder in ISCAS-Format oder in mdl-Format simulieren kann.

D.4 Partitionierungsverfahren für Statecharts

Für ein Modell mit Statecharts wird zuerst eine Baumstruktur erstellt, um den Rechenaufwand eines Zustands zu modellieren. Nach der Analyse der Abhängigkeit zwischen Zuständen wird ein Modell mit Statecharts in einen Hypergraph umgesetzt. Dann kann der implementierte K-AFM-Algorithmus auf diesem Hypergraph angewendet werden. Nach der Partitionierung wird ein neues Modell mit Teilcharts erstellt.

Erstellung der Baumstruktur von Zuständen: In Bild 225 ist auch das Klassendiagramm des Partitionierungsverfahrens für Statecharts dargestellt. Um die Analyse eines Modells mit Statecharts leicht zu behandeln, wird es zuerst durch die Klasse `StateDependencyAnalyzer` in eine Baumstruktur konvertiert. Das Chart oder der einzige Unterzustand mit AND-Dekomposition des Charts ist die Wurzel dieses Baums. Ein Knoten wird durch die Klasse `StateTreeNode` repräsentiert. Während der Erstellung der Baumstruktur wird das Attribut `weight` jedes `StateTreeNode`-Objekts zuerst auf den lokalen Aufwand des entsprechenden Zustands gesetzt. Der globale Aufwand jedes Zustands wird durch die Methode `reviseWeight()` erneut berechnet. In einem `StateTreeNode`-Objekt werden alle Variablen, auf die von diesem Zustand und von den Unterzuständen zugegriffen wird, gespeichert. Die Klasse `StateDependencyAnalyzer` analysiert zusätzlich die Abhängigkeit zwischen nebenläufigen Zuständen unter dem Wurzelknoten, und erzeugt `StateDependency`-Objekte. Ein `StateDependency`-Objekt hat folgende Attribute:

- `fromState`: Zustand, der eine Variable schreibt oder ein Ereignis sendet,
- `toState`: Zustand, der eine Variable liest, schreibt oder ein Ereignis empfängt,
- `dependantObject`: ein `Data`- oder `Ereignis`-Objekt.

Erstellung des Hypergraphs für ein Statechart-Modell: Die Klasse `GraphBuilder` erzeugt mit `StateTreeNode`-Objekten und `StateDependency`-Objekten einen Hypergraph. Das `StateTreeNode`-Objekt für einen Zustand wird als Inhalt in einem `Vertex`-Objekt gespeichert. Ein `Hyperedge`-Objekt wird anhand der `StateDependency`-Objekte mit dem gleichen Quellzustand erzeugt. Der Attribut `dependantObject` des `StateDependency`-Objekts wird auch als Inhalt eines `Hyperedge`-Objekts gespeichert. Wenn ein Zyklus in diesem Hypergraph existiert, dann wird ein azyklischer Graph als Eingabe der Partitionierung verwendet. Wenn die mögliche Anzahl von Partitionen kleiner als der Wert des Parameters `PartCount` ist, dann wird das Modell in die mögliche Anzahl von Partitionen partitioniert. Sonst wird es in die angegebene Anzahl von Partitionen partitioniert.

Wenn die Eingabeparameter `HostNames` bei der Partitionierung nicht angegeben wird oder die vorhandene Anzahl von Rechnern kleiner als der Wert des Parameters `PartCount` ist, werden die Namen von Zielteilcharts statt Zielrechnernamen in der Datei für Schnittstelleninformationen gespeichert. Für die Simulation muss der Name von Zielteilcharts in entsprechende Namen von Rechnern geändert werden, dem dieses Zielteilchart zugewiesen ist. Die Teilcharts eines Charts nach der Partitionierung werden in einer Modelldatei gespeichert.

Wiederherstellung eines neuen Modells nach der Partitionierung: Nach der Partitionierung erstellt die Klasse `ModelBuilder` mit `Hyperedge`-Objekten für Schnitte und `PartGraph`-Objekten das neue Modell. Anhand der in einem Teilgraph enthaltenen `StateTreeNode`-Objekte wird ein `State`-Objekt für ein Teilchart erzeugt. Alle in diesen `StateTreeNode`-Objekten gespeicherten Variablen werden diesem `State`-Objekt hinzugefügt. Die Variable von Schnitten werden im Quellzustand als Ausgang und im Zielzustand als Eingang hinzugefügt. Ein `StateflowSubSystem`-Objekt für einen Chart-Block wird mit Hilfe der Klasse `StateflowSynthese` erstellt. Die Verbin-

dungen zwischen Chart-Blöcken werden anhand der Schnitte erzeugt. Gleichzeitig werden die Informationen für die Schnittstellen zwischen Teilcharts in einer Datei gespeichert.

D.5 JVHDLGen

SF2VHD ist ein MATLAB-Skript, das das MATLAB-API nutzt und Statecharts von einer graphischen Repräsentation in VHDL übersetzt. Da der vorhandene Konverter SF2VHD mit Hilfe von MATLAB-M-Skripten implementiert ist, kann die Konvertierung nur in der Umgebung von MATLAB durchgeführt werden. Weil sich die Codestruktur nach der Erweiterung von SF2VHD grundlegend ändert, wird der Konverter in Java unter dem Namen JVHDLGen neu implementiert.

Die Struktur des mit SF2VHD generierten VHDL-Codes: In VHDL wird die Schnittstelle einer Komponente als `port` im Teil `entity` definiert. SF2VHD generiert für jedes Stateflowchart eine Entity. In der Entity-Deklaration wird die Schnittstelle eines Charts mit Simulink beschrieben. Alle Daten in Stateflow mit dem Scope „input from Simulink“ oder „output to Simulink“ werden jeweils als ein `in`- oder `out`-Port in `entity` deklariert. Weil keine Daten in Stateflow als Mischung von Input und Output definiert werden können, sind auch keine Ports mit dem Typ `inout` in `entity` deklariert. Drei zusätzliche Ports `ce`, `clk` und `reset` werden im VHDL `entity` hinzugefügt. Die Ausführung des Zustandsautomaten wird von dem Signal `clk` ausgelöst. Das Signal `reset` setzt den Zustandsautomat auf den initialen Zustand. Das Signal `ce` wird als Enable-Terminal benutzt, um die im letzten Zyklus berechneten Ergebnisse in diesem Zyklus auszugeben. In `entity` werden zusätzliche Funktionen zur Typkonversion deklariert.

Zu einer kompletten VHDL-Beschreibung eines Moduls gehört neben einer Schnittstellenbeschreibung auch noch mindestens eine Modellbeschreibung. Die Modellbeschreibung ist eine Entwurfseinheit, welche die inneren Eigenschaften des Moduls beschreibt. Dazu gehört einerseits der strukturelle Aufbau des Moduls, der mittels der Strukturmodellierung beschrieben werden kann, andererseits aber auch dessen innere Funktionalität, welche durch die Verhaltensmodellierung abgebildet werden kann. In SF2VHD wird die Verhaltensmodellierung verwendet. Im Deklarationsteil in `architecture` werden zuerst alle benötigten Signale deklariert. SF2VHD verwendet Signale, um die Daten in zwei verschiedenen Phasen darzustellen. Signale mit dem Suffix `_SIG` repräsentieren die Eingänge des Zustandsautomaten. Sie sind Input-Ports, im letzten Zyklus berechnete Werte oder Werte des aktuellen Zustands. Signale mit dem Suffix `_NEXT` repräsentieren aktuelle Ausgänge des Zustandsautomaten. Sie werden durch kombinatorische Funktionen der Eingänge berechnet. Für jedes Signal wird eine Variable mit dem Suffix `_VAR` deklariert, um die Daten während der Berechnung in einem Zyklus darzustellen (Tabelle 26).

VHDL-Typ	Suffix	Stateflow Scope	Verwendung
contant	_SIG	Constant	Konstant
signal	_SIG	Input, Output, Local	Inputs, letzte Outputs, momentane Zustände
signal	_NEXT	Output, Local	momentane Outputs, nächste Zustände
Variable	_VAR	Alle	Für alle Operationen

Tabelle 26: Deklaration von Signalen und Variablen

Im Anweisungsteil in `architecture` werden die Input-Ports entsprechende Signale, und Signale in entsprechende Output-Ports abgebildet. Es folgen zwei Prozesse `logic` und `synch`. Der erste Prozess entspricht der Logik des Zustandsautomaten. In der Sensitivity-Liste dieses Prozesses sind Signale mit dem Suffix `_SIG` enthalten. Im Deklarationsteil dieses Prozesses werden Variablen mit dem Suffix `_VAR` für jedes Signal in der Sensitivity-Liste deklariert. Im Anweisungsteil dieses Prozess werden diese Variablen zuerst mit dem entsprechenden Signal in der Sensitivity-Liste initialisiert. Es folgt eine `case`-Anweisung. Anhand des aktuellen Wertes des Signals für einen Zustand werden entsprechende Operationen ausgewählt. Jede Auswahl der `case`-Anweisung entspricht einem Basis-

zustand. Am Ende dieses Prozesses wird der Wert dieser Variablen dem entsprechenden Signal mit dem Suffix `_NEXT` zugewiesen.

Der zweite Prozess enthält nur ein Signal `clk` in seiner Sensitivity-Liste. Bei einer steigende Flanke werden entweder Daten und der Zustand initialisiert, oder die Signale mit dem Suffix `_NEXT` dem entsprechenden Signal mit dem Suffix `_SIG` zugewiesen, je nachdem, ob das Signal `reset` oder `ce` gleich 1 ist. Dadurch werden die aktuell berechneten Ergebnisse als Eingänge des Prozesses `logic` weitergegeben. Bild 230 zeigt die Beziehung zwischen zwei Prozessen und die Verwendung von Signalen und Daten in [Came01].

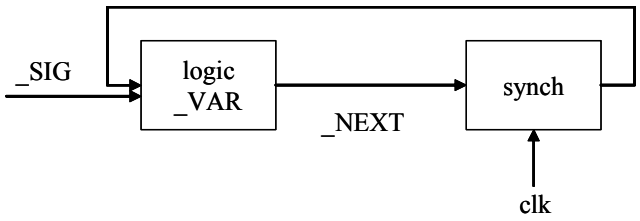


Bild 230: Verwendung von Signalen und Variablen in SF2VHD

Die vorhandene Implementierung mit M-Skripten: Die vorhandene Version ist mit zwei MATLAB-M-Skripten `sf2vhd.m` und `ParseALCodeAST.m` implementiert und benutzen die von MATLAB gelieferte Stateflow-API. Der VHDL-Code wird als Zeichenkette in den einzelnen Funktionen des ersten M-Skriptes erzeugt. Das zweite M-Skript dient zur Konvertierung der Syntax des Action-Labels in entsprechende VHDL-Operationen. Es konvertiert auch den Typ von Operanden, wenn sie in einer Operation verschiedene Typen haben. Hier werden einige wichtige Funktionen des Skriptes `sf2vhd.m` aufgelistet.

- `WriteEntity`: Code-Generierung für `entity`.
- `WriteArchitecture`: Code-Generierung für `architecture`. Die Funktion `WriteStateCode` wird aufgerufen, um Operationen für jeden Basiszustand zu generieren.
- `WriteStateCode`*:* Generierung einer `case`-Anweisung. Der Zustand mit dem Wert 0 ist ein virtueller Zustand. Die Operation des virtuellen Zustands entspricht der Default-Transition an der obersten Ebene. Der Code von Transitionen wird durch Aufruf der Funktion `WriteTransitionCode` generiert. Für andere Basiszustände wird die Funktion `WriteStateCodeHelper` aufgerufen.
- `WriteStateCodeHelper`: Für jeden Basiszustand wird die Funktion `WriteExecuteStatesCode` aufgerufen. Für einen Zustand mit Unterzuständen wird diese Funktion rekursiv aufgerufen.
- `WriteExecuteStatesCode`: Code-Generierung für einen gegebenen Zustand. Die Bedingungen hinter `if` oder `elsif`-Schlüsselwörtern entsprechen Transitionsbedingungen der Outer-Transitionen des gegebenen Zustands, deren Zielzustand anders als der Quellzustand ist. Als letztes wird `during-State`-Action in der `if`-Anweisung ausgeführt. Dann wird der Code für Inner-Transitionen, deren Zielzustand gleichzeitig auch der Quellzustand ist, generiert. Wenn der gegebene Zustand Unterzustände hat, dann wird diese Funktion rekursiv aufgerufen.
- `WriteTransitionsCode`: Code-Generierung einer `if`-Anweisung für gegebene Transitionen. Für jede Transition wird die Funktion `WriteTransitionCode` aufgerufen.
- `WriteTransitionCode`: Code-Generierung für die Ausführung einer Transition. Wenn das Ziel ein Zustand ist, dann wird der Code für die Transitionsaktionen und die `entry`-Zustandsaktionen generiert. Wenn das Ziel eine Connective-Junction ist, wird die Funktion `WriteTransitionsCode` rekursiv aufgerufen.

In Bild 231 ist das Klassendiagramm des Konverters JVHDLGen dargestellt. Ein Modell wird zuerst mit Hilfe des mdl-Parsers in ein `Model`-Objekt umgesetzt. Das Objekt wird von der Klasse `VHDLCodeGenerator` in ein `VHDLCode`-Objekt konvertiert.

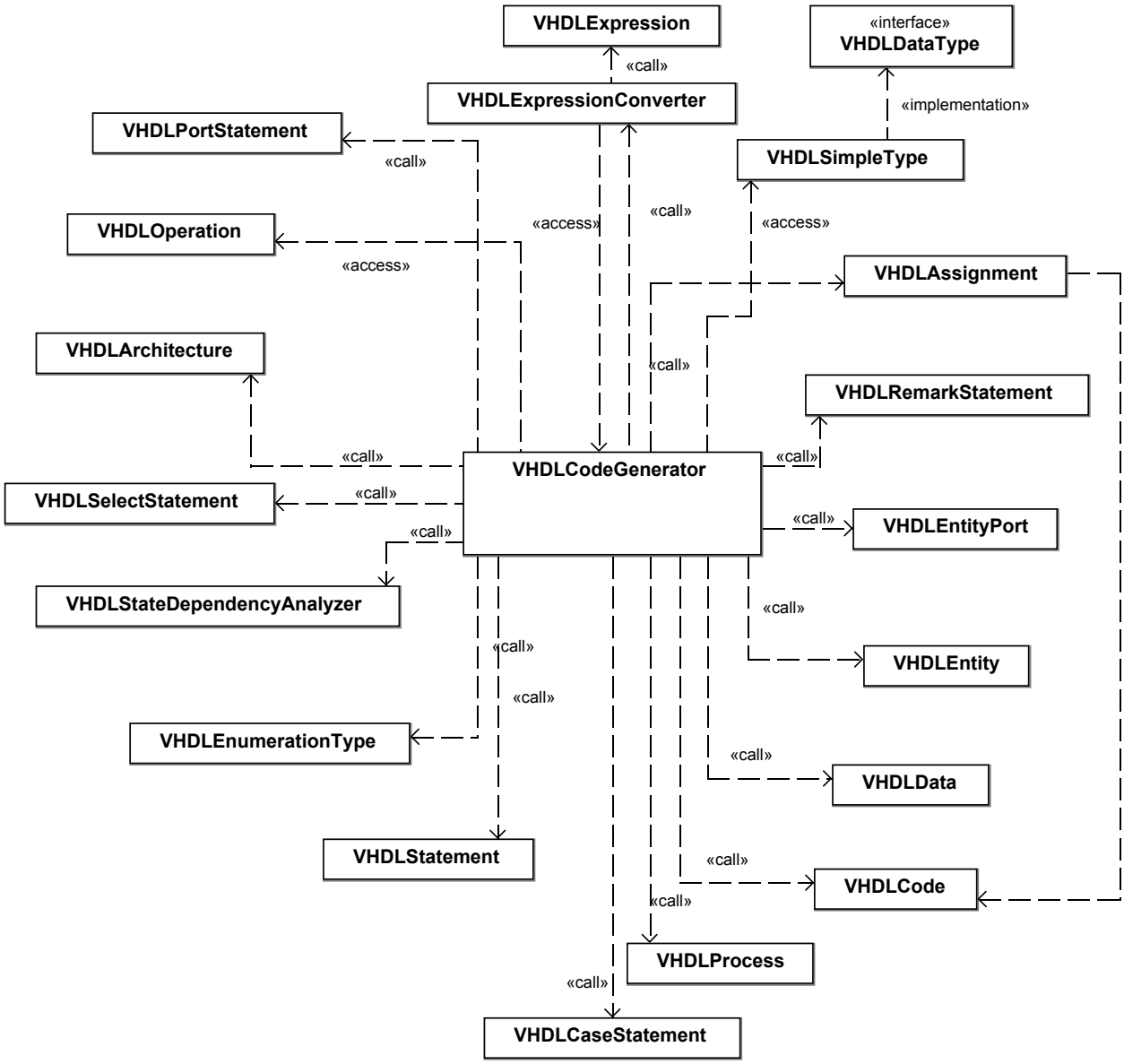


Bild 231: Klassendiagramm des Werkzeugs JVHDLGen

Die Klasse `VHDLEntity` und die Klasse `VHDLArchitecture` entsprechen dem Teil `entity` und `architecture` des VHDL-Codes. In den Methoden der Klasse `VHDLCodeGenerator` werden verschiedene `VHDLStatement`-Objekte erzeugt. Die Klasse `VHDLData` repräsentiert die Deklaration von Variablen. Ein `VHDLProzess`-Objekt enthält eine Reihenfolge von `VHDLStatement`-Objekten. Die Logikstruktur für die Bearbeitung eines Charts wird durch ein `VHDLCaseStatement`-Objekt repräsentiert. Die Bearbeitung der Transitionen aus einem Zustand wird durch ein `VHDLSelectStatement`-Objekt repräsentiert. In der Klasse `VHDLStatement` und seinen Unterklassen wird die Methode `toString()` implementiert, um verschiedene VHDL-Zuweisungen auszugeben.

Wenn nebenläufige Zustände in mehrere Prozesse umgesetzt werden, und auf die gleiche Variable zugreifen, müssen die Prozesse synchronisiert werden. Die Klasse `StateDependancyAnalyzer` analysiert vor der Konvertierung die Abhängigkeit zwischen Zuständen. Die Klasse `VHDLExpres-`

sionConverter setzt Ausdrücke des Aktions-Labels in entsprechende VHDL-Operationen um. Die Funktionalität der Typkonvertierung von Operanden wird nicht von dem Skript ParseALCodeAST.m importiert.

In der Klasse VHDLCodeGenerator werden die meisten Methoden von den Funktionen des Skriptes sf2vhd.m importiert. Einige Methoden, die für nebenläufige Zustände Änderungen haben, und neue hinzugefügte Methoden werden hier vorgestellt:

- generateStateCode: wenn der übergebene Zustand ein AND-Zustand und kein Basiszustand ist, wird diese Methode für jeden Unterzustand aufgerufen. Sonst, wenn der übergebene Zustand ein OR-Zustand und kein Basiszustand ist, wird Code für diesen Zustand erzeugt. Für jeden Unterzustand wird die Methode generateExecutStateCode aufgerufen.
- generateExecuteStateCode: erzeugt Code für alle Outer-Transitionen und für during-Zustandsaktionen dieses Zustands. Wenn der gegebene Zustand ein AND-Zustand und kein Basiszustand ist, wird die Methode generateStateCode für jeden Unterzustand aufgerufen.
- generateEnterStateCode: wenn der Quellzustand der übergebenen Transition ein AND-Zustand und kein Basiszustand ist, werden alle Unterzustände auf den virtuellen Zustand gesetzt. Dann wird der übergebene Zustand auf den Zielzustand gesetzt. Wenn der Zielzustand der übergebenen Transition ein AND-Zustand und kein Basiszustand ist, wird die Methode generateStateCode für jeden Unterzustand aufgerufen.
- getStateTypes: gibt Zustände zurück, für die ein enumeration-Typ deklariert wird. Solche Zustände sind OR-Zustände aber keine Basiszustände, deren Oberzustand ein AND-Zustand oder das Chart ist.

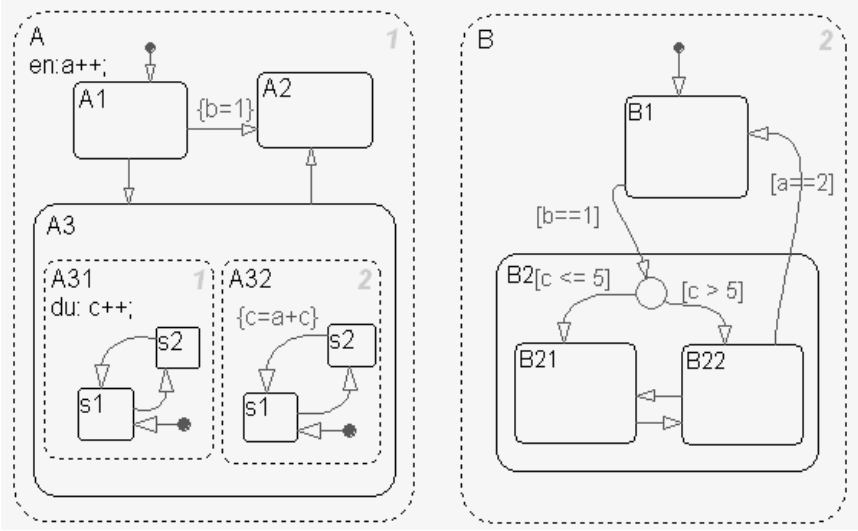


Bild 232: Chart mit verschachtelter AND-Dekomposition

Definition von enumeration-Typen für Zustände: Für jeden nebenläufigen Zustand wird ein enumeration-Typ deklariert. Um den Zustandsnamen eindeutig zu identifizieren, werden die Namen aller Oberzustände als Präfix hinzugefügt. Für das in Bild 232 dargestellte Chart werden vier enumeration-Typen mit dem Suffix _TYPE wie folgt deklariert:

```
type Chart_A_TYPE is (virtual, Chart_A_A1, Chart_A_A2, Chart_A_A3);
type Chart_A_A3_A31_TYPE is (virtual, Chart_A_A3_A31_s2, Chart_A_A3_A31_s1);
type Chart_A_A3_A32_TYPE is (virtual, Chart_A_A3_A32_s2, Chart_A_A3_A32_s1);
type Chart_B_TYPE is (virtual, Chart_B_B1, Chart_B_B2_B21, Chart_B_B2_B22);
```

Für jeden `enumeration`-Typ wird zusätzlich ein Signal mit dem Suffix `_SIG` deklariert. Es wird als Ausdruck in einer `case`-Anweisung ausgewertet, um den momentan aktiven Zustand festzulegen:

```
signal Chart_A_SIG            : Chart_A_TYPE;
signal Chart_A_A3_A31_SIG     : Chart_A_A3_A31_TYPE;
signal Chart_A_A3_A32_SIG     : Chart_A_A3_A32_TYPE;
signal Chart_B_SIG            : Chart_B_TYPE;
```

Synchronisation nebenläufiger Zustände: In Bild 232 greifen zwei Zustände `A31` und `A32` auf die gleiche Variable `c` zu. Weil Zustand `A31` rechts von Zustand `A32` liegt, muss die entsprechende VHDL-Anweisung auch höhere Priorität haben. Der Prozess für den Zustand `A32` muss vor der Ausführung `c := a + c` warten, bis Prozess `A31` am Ende das Signal `c_flag` auf `true` gesetzt hat (Bild 235). Die Variable `c` wird als `shared variable` deklariert, damit beide Prozesse auf sie zugreifen können (Bild 233).

```
signal flag_c: boolean;
shared variable a, c: unsigned(15 downto 0);

A31: process                    A32: process
…                               …
c := c + 1;                     wait until c_flag;
…                               c := a + c;
c_flag <= true;                 …
end process;                    c_flag <= false;
                                end process;
```

Bild 233: Synchronisation nebenläufiger Zustände

Wenn mehr als zwei nebenläufige Zustände von einer einzigen Variable abhängen, kann eine Warte-Schlange nach der Priorität erstellt werden. Angenommen, es greifen drei nebenläufige Zustände `A`, `B` und `C` auf eine gemeinsame Variable `a` zu. Der Zustand `A` hat die höchste Priorität, es folgt `B`, `C` hat die niedrigste Priorität. Die entsprechende Warteschlange wird in Bild 234 gezeigt. Ein Zustand wartet nur auf den direkten Vorgängerzustand. Der Zustand `B` wartet auf das Kennzeichen `a_flag_1` aus `A`, und der Zustand `C` wartet auf das Kennzeichen `a_flag_2` aus `B`.

Wenn die Abhängigkeit eines Zustands von seinen Unterzuständen mit AND-Dekomposition verursacht ist, dann wird das Kennzeichen für diese Abhängigkeit vom Oberzustand am Ende der Ausführung auf `true` gesetzt. Z. B. hängen zwei Zustände `A` und `B` in Bild 232 von der Variable `c` ab, und auf `c` wird nur von den Unterzuständen `A31` und `A32` des Zustands `A` zugegriffen. Das Kennzeichen für `c` wird von dem Prozess des Zustands `A` gesetzt.

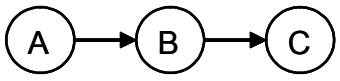


Bild 234: Warteschlange von drei nebenläufigen Zuständen

Synchronisation eines AND-Zustands und seiner Unterzustände: Bild 236 zeigt die Struktur des Codes für das in Bild 232 dargestellte Chart, wenn jeder nebenläufige Zustand in einen Prozess umgesetzt wird. Der Prozess `Chart` wartet zum Beginn auf das `clk` Signal. Nachdem das Signal `Chart_ACT` auf `true` gesetzt worden ist, können die beiden Prozesse `Chart_A` und `Chart_B` parallel ausgeführt werden. Der Prozess `Chart` wird nur dann wieder vom Anfang an ausgeführt, bis die Signale `Chart_A_END` und `Chart_B_END` von dem entsprechenden Prozess `Chart_A` und `Chart_B` auf `true` gesetzt wurden. Das Signal wird am Ende des Prozesses `Chart` auf `false` gesetzt. Die Synchronisation zwischen dem Zustand `A3` und seinen Unterzuständen `A31` sowie `A32` wird auf die gleiche Weise durchgeführt. Allerdings wird das Kennzeichen für die Synchronisation im Prozess `Chart_A` auf `true` gesetzt.

```
Chart : process

begin
wait until (clk'EVENT and clk = '1') or reset = '1';
if (clk'EVENT and clk = '1') then
   Chart_ACT <= true;
   wait until Chart_A_END and Chart_B_END;
   Chart_ACT <= false;
elsif (reset = '1') then
   …
end if;
end process;
```

Bild 235: Prozess für das Chart mit AND-Dekomposition

```
Chart_A : process

begin
      wait until Chart_ACT;
      case Chart_A_SIG is
      when virtual =>
            …
      when Chart_A_A2 =>
            …
      when Chart_A_A1 =>
            …
      when Chart_A_A3 =>
            …
      when others =>
            NULL;
      end case;
      Chart_A_END <= true;
      wait until Chart_ACT =false;
      Chart_A_END <= false;
end process;
```

```
Chart_B : process

begin
      wait until Chart_ACT;
      case Chart_B_SIG is
      when virtual =>
            …
      when Chart_B_B1 =>
            …
      when Chart_B_B2_B21 =>
            …
      when Chart_B_B2_B22 =>
            …
      when others =>
            NULL;
      end case;
      Chart_B_END <= true;
      wait until Chart_ACT =false;
      Chart_B_END <= false;
      end process;
```

Bild 236: Prozesse für die nebenläufigen Zustände A und B

Nebenläufige Zustände in einen Prozess: Bild 237 zeigt die Struktur des VHDL-Codes für das in Bild 232 dargestellte Chart, wenn alle nebenläufigen Zustände in einen Prozess umgesetzt werden. Weil der Zustand `A` höhere Priorität als der Zustand `B` hat, wird das Signal `chart_A_SIG` für den Zustand `A` vor dem Signal `chart_B_SIG` für den Zustand `B` bearbeitet. Wenn ein Zustandsübergang vom Zustand `A1` in den Zustand `A3` auftritt, wird der Unterzustand `A31` vor dem Zustand `A32` ausgeführt. Wenn der Zustand `A3` verlassen ist, werden die Unterzustände `A31` und `A32` auf den virtuellen Zustand gesetzt.

```
case Chart_A_SIG is
       when virtual =>
              …
       when Chart_A_A2 =>
              …
       when Chart_A_A1 =>
              if (…) then
                     …
              elsif (…) then
                     Chart_A_VAR := Chart_A_A3;
                     case Chart_A_A3_A31_SIG is
                            …
                     end case;
                     case Chart_A_A3_A32_SIG is
                            …
                     end case;
              …
       when Chart_A_A3 =>
              if (…) then
                     Chart_A_A3_A31_VAR := virtual;
                     Chart_A_A3_A32_VAR := virtual;
                     Chart_A_VAR := Chart_A_A2;
              else
                     Chart_A_VAR := Chart_A_A3;
              end if;
       when others =>
              NULL;
end case;
case Chart_B_SIG is
   when virtual =>
              …
   when Chart_B_B1 =>
              …
   when Chart_B_B2_B21 =>
              …
   when Chart_B_B2_B22 =>
              …
   when others =>
              NULL;
end case;
```

Bild 237: VHDL-Code des Charts von Bild 232 in einem Prozess

D.6 µC-Integration unter Einsatz von CAN und OSEK/COM

Die auf OSEK basierten Modelle sind unter Verwendung der OSEK-Realisierung OSEKWorks von Wind River implementiert. Dabei kommen folgende Softwaremodule zum Einsatz:

- Anwendungscode: Der vom Anwendungsentwickler geschriebene Programmcode, in dem Funktionalitäten von OSEKWorks verwendet werden.
- System Library: Eine Bibliothek, die alle von OSEKWorks bereit gestellten Funktionen enthält.
- Board Support Package (BSP) Library: Eine Bibliothek, die Funktionalität zur Unterstützung der Zielhardware bereitstellt, z. B. Initialisierungsroutinen und Gerätetreiber.
- OIL: Datenobjekte und Präprozessor-Makros, die vom OIL-Reader aus einer OIL-Datei erstellt und zur Konfiguration des OSEKWorks-Systems verwendet wird.

Externe Kommunikation mit in OSEKWorks integrierter CAN-Funktionalität: Über die API-Funktionen des CAN-Treibers von Wind River kann innerhalb einer Anwendung direkt auf die vom CAN-Controller zum Senden und Empfangen von CAN-Nachrichten bereitgestellten CAN-Nachrichtenpuffer zugegriffen werden. Um eine Übertragung über CAN zu initiieren, wird der CAN-Nachrichtenpuffer mit der CAN-ID und den Nachrichtendaten gefüllt und sein Zustand auf „trans-

mit“ gesetzt. Dann übernimmt die Hardware die Kontrolle und überträgt die Daten über das Netzwerk. Mit einem Interrupt kann signalisiert werden, dass die Nachricht übermittelt wurde. Um eine CAN-Nachricht empfangen zu können, muss ein Nachrichtenpuffer zum Empfang der Nachricht mit entsprechender CAN-ID konfiguriert sein. Dies geschieht über die CAN-Treiber-API-Funktionen. Wenn eine Nachricht mit passender CAN-ID vom Controller empfangen wird, löst die Hardware ein Interrupt aus.

Beim Einsatz von OSEKWorks/COM für externe Kommunikation über ein CAN-Netzwerk kommen zusätzlich zu `MESSAGE`- und `COM`-Objekten folgende drei OIL-Objekttypen zum Einsatz:

- Das `CANOBJECT`-Objekt repräsentiert den Nachrichtenpuffer des CAN-Controllers. Für das `CANOBJECT`-Objekt wird spezifiziert, ob es zum Senden oder Empfangen eingesetzt wird und im zweiten Fall welche CAN-ID empfangen werden soll.
- Das `NETWORK`-Objekt repräsentiert den CAN-Controller. Es spezifiziert unterschiedliche COM-Protokollparameter und enthält eine Anzahl von `CANOBJECTs`. Wenn die CAN-Hardware mehrere Controller unterstützt, wird für das `NETWORK`-Objekt auch konfiguriert, welchen Controller es repräsentiert.
- Das `CANADDRESS`-Objekt verknüpft ein `MESSAGE`-Objekt mit einer CAN-ID und dem Netzwerk, über das die Nachricht gesendet oder empfangen werden soll.

In Bild 238 ist der Zusammenhang der einzelnen Objekte dargestellt. Dabei bedeutet ein Pfeil von Objekt 1 zu Objekt 2, dass Objekt 1 Objekt 2 enthält.

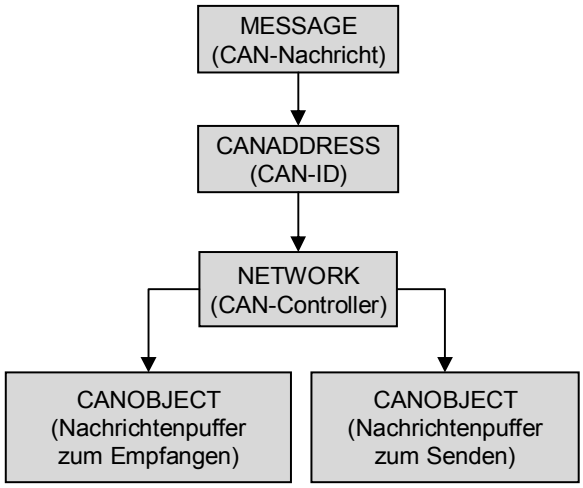


Bild 238: OIL-Objekte zur externen Kommunikation über CAN

Der TouCAN ist eine externe IMB-Komponente (intermodule bus), was eine Vielzahl an möglichen Konfigurationen der Interrupt-Ebenen bedeutet. Entsprechend der jeweiligen Interrupt-Konfiguration muss ein `ISR`-OIL-Objekt als Repräsentant für das externe Interrupt verwendet werden, über das die Interrupt-Ebene in OSEKWorks definiert wird. Bild 239 zeigt ein Anwendungsbeispiel eines solchen `ISR`-OIL-Objekts.

```
ISR Ext500DispatcherISR {
       CATEGORY = 2;
       ENABLE_AT_STARTUP = FALSE;
       VECTOR = EXTERNAL_IO {
              SOURCE = INT_IMBIRQ3 {
                     HANDLER_FUNCTION = "COM_ISR_Module0";
              };
       };
};
```

Bild 239: Interrupt-Konfiguration in OIL

Ist das OSEK-System entsprechend konfiguriert, kann auf die Funktionalität des CAN-Kommunikationssystems im C-Code über die in der API von OSEK COM spezifizierten Funktionen zugegriffen werden. In der Funktion `StartupHook()` wird OSEK COM über den Aufruf `StartCOM()` initialisiert. Gesendet wird eine Nachricht mit `SendMessage(send_msg, &send_value)`, wobei `msg` in der OIL-Datei als `MESSAGE`-Objekt aufgeführt sein muss und `send_value` eine Variable repräsentiert. Empfangen wird eine Nachricht mit `ReceiveMessage(rcv_msg, &rcv_value)`, wobei wiederum `msg` in der OIL-Datei als `MESSAGE`-Objekt aufgeführt sein muss und `send_value` eine Variable repräsentiert.

Die jeweils verwendeten Variablen müssen in der OIL-Datei als Parameter des `TASK`-Objekts der entsprechenden aufrufenden Task konfiguriert sein (Bild 240).

```
TASK send_task {
       SCHEDULE = FULL;
       PRIORITY = 5;
       …
       ACCESSOR = SENT {
              ACCESSNAME = "send_value";
              WITHOUTCOPY = TRUE;
              MESSAGE = send_msg;
       };
};
TASK rcv_task {
       SCHEDULE = FULL;
       PRIORITY = 10;
       …
       ACCESSOR = RECEIVED {
              ACCESSNAME = "rcv_value";
              WITHOUTCOPY = FALSE;
              MESSAGE = rcv_msg;
       };
};
```

Bild 240: Variablen-Konfiguration

Beim Herunterfahren des Systems wird OSEK COM in der Funktion `ShutdownHook()` über den Aufruf `StopCOM()` beendet.

Externe Kommunikation mit direktem Zugriff auf Funktionen des CAN-Treibers: Alternativ dazu wurde eine andere Lösung dafür realisiert, CAN-Nachrichten mit dem MPC555 zu senden und zu empfangen. Hierzu wurde der in OSEKWorks integrierte CAN-Treiber isoliert und in eine OSEK-unabhängige Fassung gebracht. Somit lässt er sich auch bei Systemen nutzen, die kein OSEK verwenden. Die OIL-Objekte `MESSAGE`, `CANOBJECT`, `CANADDRESS` und `NETWORK` können nun nicht mehr zum Einsatz kommen, sondern es wird bei dieser Vorgehensweise direkt auf Funktionen und Datentypen des CAN-Treibers von Wind River zugegriffen, die in [BSPW02] beschrieben sind.

In Bild 241 wird anhand der Funktion `can_init()` zur CAN-Initialisierung die Verwendung der Funktionen des CAN-Treibers verdeutlicht: Zunächst wird mit `OW_CAN_Init()`der CAN-Controller in einen definierten Ausgangszustand versetzt. `OW_CAN_InstallISR()` macht die ISR bekannt und nach entsprechender Konfiguration mit `OW_CAN_Make_Msg()` wird der Puffer zum Empfang von Nachrichten in das System eingebunden. Nun kann die Arbitrierung des CAN-Busses mit der Funktion `OW_CAN_Start()`eingeleitet werden.

```
void can_init() {
      CAN_MsgType std_msg_rx;

      OW_CAN_Init(Channel);
      OW_CAN_InstallISR(test_isr, Channel);

      // Konfiguration eines Receive-Buffers
      std_msg_rx.MSG_NUM = 0x1;
      std_msg_rx.MSG_IDE = CAN_STANDARD;
      std_msg_rx.MSG_TYPE = CAN_RX_MSG;
      std_msg_rx.MSG_INT = CAN_ENABLE;
      std_msg_rx.ID = 0x1;

      OW_CAN_Make_Msg(&std_msg_rx,Channel);

      OW_CAN_Start(Channel);
}
```

Bild 241: CAN-Initialisierung in der Funktion StartupHook()

Nach Bestimmung der relevanten CAN-Register wird der Inhalt des Interrupt Flag Register `can_IFLAG` ausgelesen und lokal gespeichert. Nun werden in der `while`-Schleife die einzelnen Bits dieses Registerinhalts betrachtet. Über `OW_CAN_Get_Msg()` kann die im Nachrichtenpuffer vorliegende Nachricht ausgelesen werden. Im vorliegenden Beispiel wird auf den Empfang einer Nachricht mit ID 1 mit dem Senden einer anderen Nachricht mit der ID 3 reagiert. Hierzu müssen zunächst die Parameter der zu sendenden Nachricht konfiguriert werden, danach wird sie mit der Funktion `OW_CAN_Make_Msg()` dem Nachrichtenpuffer übergeben, von wo aus sie über den CAN-Bus übertragen wird. Im Rahmen dieser Arbeit durchgeführte Messungen haben ergeben, dass mit dieser Methode eine deutlich höhere Datenrate erzielt werden kann. Allerdings ist die Programmierung aufwändiger. Zu beachten ist, dass Nachrichten verloren gehen können, wenn im Quelltext der Nachrichtenversand mehr als zweimal direkt aufeinander folgend aufgerufen wird. Dieses Problem wird gelöst, wenn man nach jedem Versand die Funktion `delay(1)` (siehe folgenden Abschnitt) aufruft, so dass dem CAN-Controller genügend Zeit zwischen dem Versenden der Nachrichten bleibt. Wenn innerhalb eines Projekts sowohl über OSEK COM als auch unter direktem Zugriff auf den CAN-Treiber CAN-Nachrichten versendet werden, kann es zu Inkompatibilitäten kommen, die zu Kommunikationsfehlern führen.

CAN-Kommunikation mit Simulink und CANcardX: Für eine komfortable Verarbeitung der von der externen Hardware via CAN-Nachrichten gelieferten Ergebnisse bietet sich ein graphisches Frontend wie Simulink an. Außerdem können mit diesem Werkzeug Stimuli erzeugt werden, die als CAN-Nachrichten an den Controller gesendet und von diesem verarbeitet werden. Wichtigste Komponenten sind dabei zwei Simulink S-Funktionen `SendMessage` und `ReadMessage`, in denen ein Großteil der notwendigen Funktionalität realisiert ist und die als `S-Function`-Blöcke in unterschiedlichen Modellen nach entsprechender Anpassung wieder verwendet werden können (Bild 242).

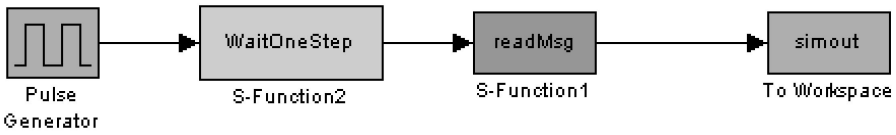


Bild 242: Simulinkmodell zum Auslesen von CAN-Nachrichten

Es gilt zu beachten, dass bei einer Simulation in Simulink jeder Block zur Initialisierung vor dem eigentlichen Simulationsstart einmal berechnet bzw. ausgeführt wird. Da allerdings im vorliegenden Fall bei der Initialisierung CAN-Nachrichten weder versendet noch empfangen werden sollen, wird dies in der S-Funktion mit Hilfe der als Flag verwendeten Variablen `firststep` bzw. `firstcall` umgangen. Um die als CAN-Nachrichten vorliegenden Daten auf Simulink-Seite empfangen zu können, muss zunächst die CAN-Funktionalität des PCs initialisiert werden. In vorliegendem Fall muss

der Treiber für die PC-CAN-Hardware CANcardX geladen werden (API hierzu in [CANL03]). Die `InitDriver`-Funktion wird in `mdlInitializeSizes` aufgerufen. Dort wird außerdem mit `CreateEvent` ein Ereignis der Windows32-API erzeugt und mit `ncdSetNotification` mit dem Porthandle des CAN-Anschlusses gebunden. So kann die Funktion `mdlOutputs` der S-Funktion mit `WaitForSingleObject` auf Signalisierung ankommender CAN-Nachrichten warten und arbeitet erst nach Auftreten eines Ereignisses (oder Erreichen eines Timeouts) weiter. Die CAN-Nachricht wird mit `ncdReceive1` empfangen, der Zugriff auf die Daten geschieht über den Zeiger auf das Datenfeld `tagData.msg.data` der Nachrichtenvariablen vom Typ `Vevent`. Es handelt sich hierbei um ein Byte-Array, dessen Größe abhängig von der Datenlänge der empfangenen CAN-Nachricht ist. Beim Versenden wird zunächst mit `ssGetInputPortRealSignalPtrs(S, 0)` ein Zeiger auf den Block-Eingangswer erzeugt. Der darüber referenzierbare Variablenwert wird in das Datenfeld der zu erstellenden CAN-Nachricht `pEvent` kopiert und deren Parameter konfiguriert. Mit `ncdTransmit(gPortHandle, gChannelMask, &pEvent)` wird die Nachricht über die CANcardX übertragen.

D.7 Messung der Ausführungsdauer von Tasks

Eine Möglichkeit, die Ausführungsdauer von Tasks zu bestimmen, ist die Zeitmessung mit OSEK-Alarms. Diese sind gemäß der OSEK OS-Spezifikation an einen OSEK-Counter gebunden. Ein Alarm wird ausgelöst, wenn der Counter einen vorgegebenen Wert erreicht hat. Bei Auslösen des Alarms können unterschiedliche Aktionen durchgeführt werden: Aktivierung einer Task, Setzen eines Ereignisses oder Flags oder Aufruf einer Callback-Funktion. Konfiguriert werden Counter und Alarm in der OIL-Beschreibung (Bild 243).

```
COUNTER SystemTimer {
       MAXALLOWEDVALUE = 1000;
       TICKSPERBASE = 1000;
       MINCYCLE = 1;
};

ALARM timeAlarm {
       COUNTER = SystemTimer;
       ACTION = ACTIVATETASK{
              TASK = incr_timecycle;
       };
};
```

Bild 243: Konfiguration von Counter und Alarm

Da der Counter zyklisches Verhalten aufweist, ist es nicht möglich, allein über die Differenz der Counter-Werte zu zwei Zeitpunkten die Dauer zwischen diesen zwei Zeitpunkten zu bestimmen. Vielmehr muss zusätzlich die Anzahl $n = n_1\text{-}n_0$ der zwischenzeitlich abgelaufenen Zyklen in die Berechnung mit eingehen. Die Zeitspanne zwischen zwei Zeitpunkten t_0, n_0 und t_1, n_1 berechnet sich demnach wie folgt:

$$t = (t_1 - t_0) + (n_1 - n_0) \cdot T$$

mit

t_0: Zählerwert zum Beginn der Messung
t_1: Zählerwert zum Ende der Messung
n_0: Anzahl der bereits durchlaufenen Counter-Zyklen zum Beginn der Messung
n_1: Anzahl der durchlaufenen Counter-Zyklen zum Ende der Messung
T: Zeitdauer eines vollständigen Counter-Zyklus

Anhand von Bild 244 wird das Vorgehen veranschaulicht: Das Erreichen des Maximalwerts des Counters wird durch einen Alarm signalisiert, der in der `StartupHook()`-Funktion über `SetAbsAlarm()` initialisiert wird. Dieser aktiviert den Task `incr_timecycle`, in dem eine Variable `n` zum

Zählen der durchlaufenen Zyklen jeweils um eins erhöht wird. In dem Task `send_task`, dessen Ausführungsdauer gemessen werden soll, wird bei Eintritt der aktuelle Counterwert t_0 und die Anzahl der bereits durchlaufenen Counterzyklen n_0 festgehalten (Variablen `time_taskBegin` und `cycle_taskBegin`). Zur Bestimmung des aktuellen Counterwerts dient dabei die Funktion `GetCounterValue()`. Nach Abarbeitung der eigentlichen Task-Funktionalität wird die Ausführungsdauer mit Hilfe erneut bestimmter Werte für Counter und Zyklenanzahl berechnet. Das Ergebnis wird als Nachricht versendet und somit dem Nutzer bereitgestellt.

```
void StartupHook () {
	SetAbsAlarm(timeAlarm, 1000, 1);
	(void)StartCOM();
}

TASK(send_task) {
	TickType time_taskBegin = 0;
	long task_duration = -1, cycle_taskBegin = -1;
	long i = -1;
	int x = 0;
	// Bestimme Anfangswerte der Zeitmessung
	cycle_taskBegin = timeCycles;
	GetCounterValue(SystemTimer, &time_taskBegin);
	// Aktionsschleife: hier ist die Task-Funktionalität implementiert
	for (i = 0; i < 30000; i++) {
		x = i;
	}
	// Bestimme Endwerte der Zeitmessung
	GetCounterValue(SystemTimer, &time_taskEnd);
	// Formel: (t1 - t0) + (n1 -n0)*T
	task_duration = ((long)(time_taskEnd)- (long)(time_taskBegin))
			+ ((timeCycles - cycle_taskBegin) * maxAllowedValue);
	// Sende Ergebnis der Zeitmessung
	SendMessage(msg1to2, &task_duration);
	TerminateTask();
}

TASK(incr_timecycle) {
	timeCycles++;
	TerminateTask();
}
```

Bild 244: Zeitmessung bei Tasks

Im Rahmen dieser Arbeit durchgeführte Messungen haben gezeigt, dass diese Methodik nur bei Tasks mit langer Ausführungsdauer anwendbar ist, da zum einen die Zeitauflösung des verwendeten `SystemTimers` zu gering ist und es zum anderen eine Grenzfrequenz gibt, bei der der vom Alarm gestartete Task nicht mehr vollständig ausgeführt wird. Aus diesem Grund wurde bei den Messungen eine zweite Methode gewählt: Unter Umgehung von OSEK wird direkt über Assembler-Befehle auf das Timebase-Register des MPC555 zugegriffen. Die resultierenden Werte werden in Zähleinheiten (Ticks) gemessen, die jeweils 0,8 µs entsprechen. Dieser Wert wurde in einer Referenzmessung ermittelt, bei der die Ausführung des Befehls `delay(30000)` 253 s benötigte, die mit 315197637 Zähleinheiten gemessen wurden.

Durch Einführen einer weiteren CAN-Nachricht kann man die Aktiv-/Passiv-Zeiten der Tasks in Simulink graphisch darstellen (siehe 4.7.2). Bei Start der Task-Ausführung wird zusätzlich zur CAN-Nachricht mit der Startzeit eine weitere Nachricht mit einer Eins als Datum versendet, die den Start der Ausführung anzeigt. Bei Beendigung werden wiederum zwei Nachrichten geschickt: die Endzeit und eine Null zum Kenntlichmachen des Ausführungsendes. Die Nachrichten haben unterschiedliche IDs und werden in Simulink von zwei zugehörigen S-Funktionen empfangen. Die Werte werden an ein XY-Scope weitergegeben, wobei die Zeitangaben auf der X-Achse und die Aktiv-/Passiv-Signale auf der Y-Achse aufgetragen werden.

Um die Methodik der Zeitmessung möglichst weit zu automatisieren, wurde die dort aufgeführte Funktionalität in zwei `Custom Code`-Blöcke von TL integriert, wobei der erste für die Initialisierung und den Start der Messung und der zweite für deren Beendigung und die Ergebnisübertragung zuständig ist. Der zu messende Modellteil kann somit zwischen diese beiden Blöcke geschaltet werden, was die Handhabung der Zeitmessung erleichtert. Will man innerhalb eines Modells mehrere `Custom Code`-Blöcke mit identischer Funktionalität nutzen, so darf man nicht dieselbe Quelltextdatei verwenden, sondern muss für jeden Block eine eigene Datei mit eigenen einmaligen Variablennamen erzeugen. Da TL bei der Code-Generierung die Quelltexte der `Custom Code`-Blöcke direkt in den generierten Code einbindet, kommt es andernfalls zu unerlaubten Mehrfachdeklarationen. Zur Einbindung eigenen Quellcodes in `Custom Code`-Blöcke wird von TL ein Template zur Verfügung gestellt, in dem das spezielle Format für Quelltexte von `Custom Code`-Blöcken bereits berücksichtigt ist, so dass der Benutzer die entsprechenden Implementierungsabschnitte nur an den entsprechenden Textstellen einfügen muss, die durch spezielle Kommentare gekennzeichnet sind. Im vorliegenden Beispiel (Bild 245) werden die globalen Variablen zwischen den Kommentarkennungen `/* fxp_decl_begin */` und `/* fxp_decl_end */` deklariert, der Quellcode des Funktionsrumpfs erscheint zwischen `/* fxp_output_begin */` und `/* fxp_output_end */`. Dieses Vorgehen ist notwendig, um bei der Code-Generierung durch TL die korrekte Einbindung des eigenen Quellcodes in den generierten Quelltext sicherzustellen.

```
# Deklaration von globalen Variablen
/* fxp_decl_begin */
extern long timeCycles, maxAllowedValue;
TickType time_taskBegin = 0;
long cycle_taskBegin = -1;
/* fxp_decl_end */

# Funktionsrumpf mit u als Input und y als Output
/* fxp_output_begin */
       // Variablen zur Zeitmessung
       long task_duration = -1;
       long i = -1;
       int x = 0;
       // Beginne Zeitmessung
       cycle_taskBegin = timeCycles;
       getActualCounter(&time_taskBegin);
       // Leite Eingangswert weiter
       y = u;
/* fxp_output_end */
```

Bild 245: Initialisierung und Start der Zeitmessung im Custom Code-Block

```
# Funktionsrumpf mit u als Input und y als Output
/* fxp_output_begin */
       // Variablen zur Zeitmessung
       TickType time_taskEnd = 0;
       long task_duration = -1;
       // Beende Zeitmessung
       GetCounterValue(SystemTimer, &time_taskEnd);
       // Formel: (t1 - t0) + (n1 -n0)*T
       task_duration = ((long)(time_taskEnd)- (long)(time_taskBegin))
                        + ((timeCycles - cycle_taskBegin) * maxAllowedValue);
       // Sende Ergebnis der Zeitmessung
       SendMessage(OutDurationMsg, &task_duration);
       // Leite Eingangswert weiter
       y = u;
/* fxp_output_end */
```

Bild 246: Beendigung der Zeitmessung und Ergebnisübermittlung im Custom Code-Block

Zwei weitere Sonderheiten zeichnen den Code eines `Custom Code`-Blocks aus: Zum einen kann man im Quellcode eines solchen Blocks durch definierte Ein- und Ausgabevariablen mit der Modellum-

gebung Informationen austauschen, wie z. B. durch `u` und `y` in Bild 246. Außerdem ist es möglich, auf Objekte der Modellumgebung zuzugreifen, wie z. B. auf `OutDurationMsg`.

D.8 Integration eines ABS-Systems

In Simulink/TL wurde ein komplexes Modell eines ABS-Systems realisiert [Lutz03], aus dem mit TL Code für den MPC555 erzeugt werden kann. Hierzu wird für jede Partition ein TL-Subsystem erzeugt, das jeweils den für diese Partition vorgesehenen Systemteil enthält und auf einem MPC555 ausgeführt werden soll. Für die Simulink Modelle Nr. 1-3 (`in1_5send`, `in1_1send` und `1_1muxsend`) ist das Modell komplett in ein Subsystem integriert. Bild 247 zeigt die Aufteilung in zwei Subsysteme, wobei das eine Subsystem den kontinuierlichen, das andere den diskreten Modellteil enthält (relevant für Simulink-Modelle Nr. 4: `in2_1muxsend`, Nr. 5: `in2_1muxsend verteilt`, Nr. 6: `in2_1muxsendreceive` und Nr. 7: `in2_1muxsendreceive verteilt`). Zur Kommunikation der Subsysteme untereinander und mit Simulink wird in den C-Code manuell CAN-Funktionalität integriert. Bei Auslösung eines Interrupts durch eine eingehende CAN-Nachricht wird innerhalb der CAN-ISR nach Auswertung der empfangenen CAN-Nachricht die Funktionalität des Subsystems realisierenden Funktion aufgerufen und somit die neuen Ausgangswerte des Subsystems bestimmt, die dann via CAN übermittelt werden.

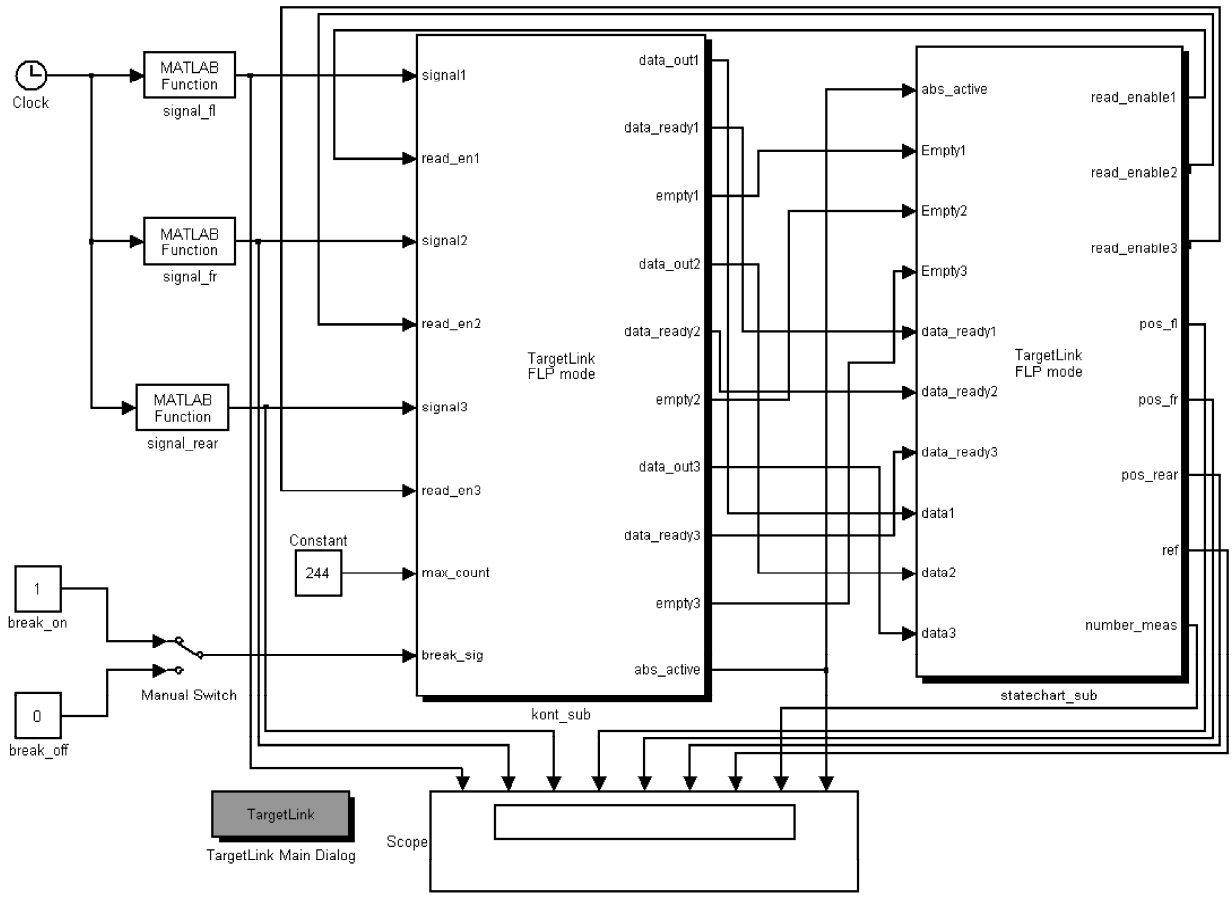


Bild 247: In zwei TL-Subsysteme integriertes ABS-Modell zur Code-Generierung

Bei Verwendung eines Multiplexer-Blocks (Bild 248, Modell Nr. 1: `in1_5send`) besteht der Nachteil, dass die Ausführungsreihenfolge der S-Funktion-Blöcke aufgrund der Parallelschaltung am Multiplexereingang Simulink-intern festgelegt wird und daraus eine von außen willkürlich erscheinende Sendereihenfolge resultiert.

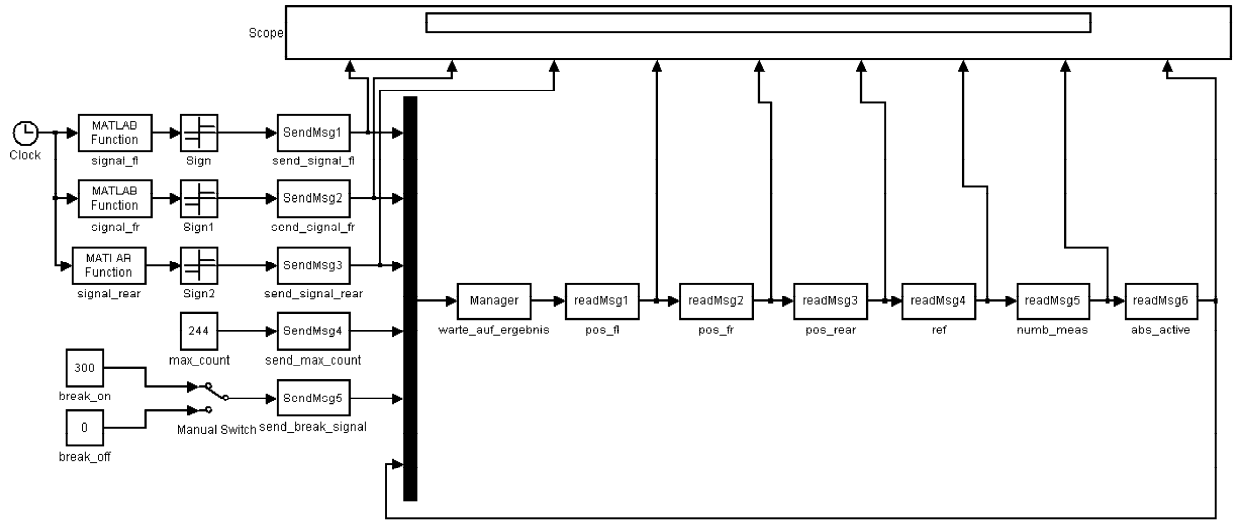


Bild 248: Senden von CAN-Nachrichten mit fünf S-Funktion-Blöcken

Will man die Sendereihenfolge vorgeben, empfiehlt es sich, nur eine S-Funktion zu nutzen, die alle Daten in der vorgesehenen Ordnung versendet (Bild 249, Modell Nr. 2: `in1_1send`).

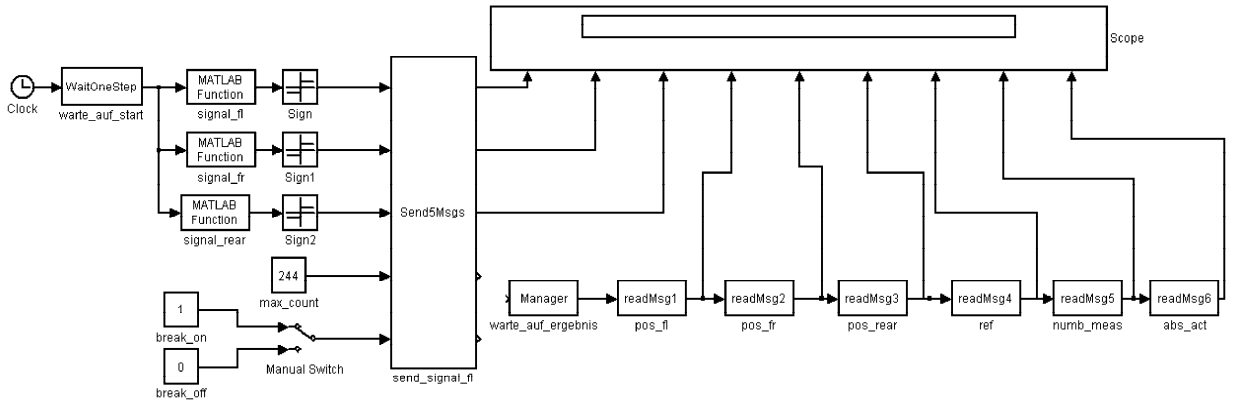


Bild 249: Senden von CAN-Nachrichten mit einem S-Funktion-Block

D.9 FPGA-Integration

In einer anderen Modellvariante (Modell Nr. 8: `fpga`) wird aus dem Stateflowchart des ABS-Modells mit Hilfe des Werkzeugs JVHDLGen VHDL-Code erzeugt, der auf dem FPGA ausgeführt werden kann. Der kontinuierliche Teil wird in Simulink simuliert. Der Versuch, den kontinuierlichen und den diskreten Teil gemeinsam auf dem FPGA auszuführen, scheitert an der begrenzten Zahl von Logikelementen auf dem FPGA. Dies liegt hauptsächlich an der Verwendung 32 Bit großer Datentypen (`UInt32`) in Stateflow. Der Ausgang des kontinuierlichen Teilsystems liefert in der FPGA-Realisierung Ergebnisse mit einer Breite von 18 Bit. Da Stateflow aber nur Variablen der Größe 1, 2 und 4 Byte zur Verfügung stellt, ist man im vorliegenden Fall gezwungen, den 4 Byte großen Datentyp zu nutzen, um die Werte am 18-Bit-Ausgang zu übernehmen. Der Einsatz von `UInt32` führt bei der Synthese für das FPGA zu großen Signalleitungen und entsprechend großen Bedarf an Logikelementen.

Tabelle 27 fasst die realisierten Partitionierungs- und Kommunikationsvarianten des ABS-Modells in einer Übersicht zusammen.

Nr.	Subsysteme	MPCs	Sende-S-Funktio-nen	Nachrich-ten von Simulink	Nachrichten von MPC1	Nachrichten von MPC2	Nachrichten vom FPGA	Empfangs-S-Funktio-nen	Nach-richten gesamt
1	1	1	5	5	6	-	-	6	11
2	1	1	1	5	6	-	-	6	11
3	1	1	1	1	6	-	-	6	7
4	2	1	1	1	8	-	-	8	9
5	2	2	1	1	1	8	-	8	10
6	2	1	1	1	1	-	-	1	2
7	2	2	1	1	1	1	-	1	3
8	-	-	1	1	-	-	1	1	2

Tabelle 27: Implementierungsvarianten des ABS-Modells

D.10 Integration von TCP/IP-Funktionalität in JStateSim und JSimControl

Zur Anbindung des vom RTW generierten Code an das INTERACT-Framework muss eine TCP/IP-Kommunikationsstruktur eingebunden werden, die den Datenaustausch mit den anderen INTERACT-Komponenten ermöglicht. Zur Steuerung der Modellausführung ist eine Kontrollstruktur notwendig, die auf eingehende Steuerbefehle entsprechend reagiert. Da das Gesamtmodell mit optimistischen Methoden simuliert werden soll, sind Mechanismen wie Zustandsspeicherung, Rücksetzung und Negativnachrichten zu realisieren.

Die Integration von TCP/IP macht eine neue Server-Client-Struktur notwendig, da bei einem RMI-Programm durch die Verwendung der durch das RMI-Framework zur Verfügung gestellten Funktionalitäten das Kommunikationsverhalten auf einer höheren Abstraktionsebene implementiert werden kann, als dies bei Socket-Programmierung mit TCP/IP möglich ist. Bild 250 zeigt die Threads, die bei der RMI-Version von JSimControl und JStateSim verwendet wurden, wobei jede Simulatorinstanz nur einen Kommunikationsthread hat. JSimControl hat aus Synchronisationsgründen für jede angekoppelte JStateSim-Instanz einen eigenen Kommunikationsthread. Die Klassendiagramme zu JSimControl und JStateSim sind in Bild 257 bzw. in Bild 258 bis Bild 262 dargestellt.

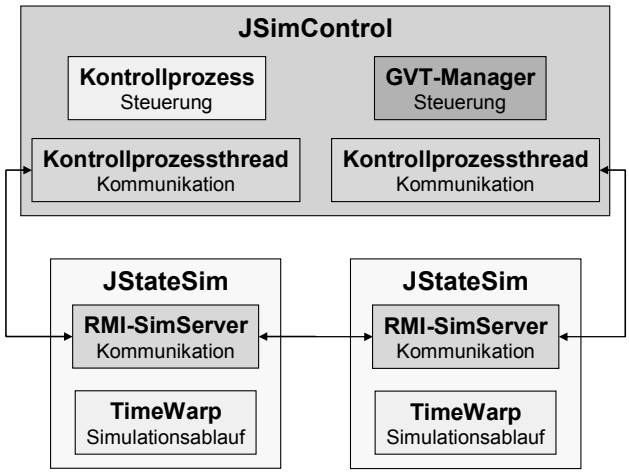


Bild 250: Threads bei der RMI-Version von JSimControl und JStateSim

Bei einer TCP/IP-Realisierung sind die Kommunikationsthreads wesentlich enger an den jeweiligen Kommunikationskanal gebunden, wenn eine ständige Verbindung zwischen zwei Kommunikationsteilnehmern aufrechterhalten werden soll, um so jederzeit Datenaustausch zu ermöglichen. Da die Empfangsfunktionen bei TCP/IP für eine möglichst schnelle Datenverarbeitung auf eingehende Informationen warten und somit blockierende Wirkung haben, müssen sie jeweils in einem eigenen Thread realisiert sein, um nicht den gesamten Programmfortschritt zu stoppen. Bild 251 zeigt die neue Kommunikationsstruktur in der TCP/IP-Version von JSimControl und JStateSim, bei der deutlich mehr Threads nötig sind, um eine verteilte Simulation zu realisieren. Bei JSimControl ist aus

kommunikationstechnischer Sicht für jeden angekoppelten Simulator ein eigener Thread notwendig, da jeder Thread eine eigene Socket-Verbindung zu seinem entsprechenden Simulator verwaltet. Um beim Verbindungsaufbau keine Verklemmung zu erzeugen, werden für unterschiedliche Aufgaben Verbindungen über unterschiedliche Ports aufgebaut. Dies erhöht die Zahl an Kommunikationsthreads weiter, ist aber für eine reibungslose Initialisierung notwendig.

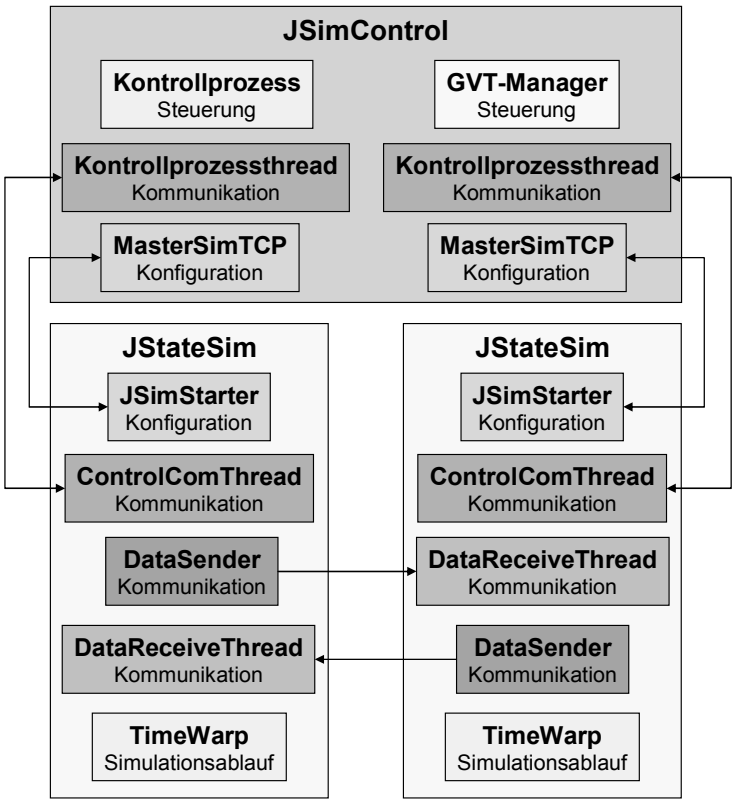


Bild 251: Threads bei der TCP/IP-Version von JSimControl und JStateSim

Während bei RMI serialisierte Objekte übertragen werden können, muss bei TCP/IP ein Protokoll verwendet werden, dass das Format der zu übertragenden Information festlegt und auf Empfängerseite eine gültige Interpretation ermöglicht.

Verwendete TCP/IP-Ports: Die `ServerSockets` öffnen festgelegte TCP/IP-Ports, um auf Verbindungsanfragen antworten zu können. Da feste Portnummern verwendet werden, darf pro Rechner genau ein Simulator ausgeführt werden. `JSimStarterTCP` öffnet Port 3000 zur Initialisierung (Übertragen von Modell und Simulations-Parameter), `SimManagerTCP` verwendet Port 3010 für den Austausch von Steuer- und Statusinformation während der Simulation. Auf Port 3020 werden vom `EventReceiverThread` während der Simulation die Ereignisse empfangen.

Verbindungsaufbau: Da jeder Simulator hinsichtlich der Kommunikation zugleich Server und Client ist, also Verbindungen sowohl über `ServerSocket` als auch über `Socket` aufbaut, besteht bei einer seriellen Verbindungsherstellung die Gefahr einer Verklemmung: warten beispielsweise zwei Simulatoren auf dem `ServerSocket` auf eine Verbindungsanfrage des jeweils anderen Simulators, bevor sie selbst Verbindungsanfragen über `Socket` stellen, kommt es zu einer Blockade. Dieses Problem kann durch parallelen Verbindungsaufbau gelöst werden: sobald ein Simulator die Adresse des Remote-Hosts kennt, versucht er, in einem Thread mit `while`-Schleife eine Verbindung zum Kommunikationspartner herzustellen (Bild 252). Dies geschieht solange, bis eine Verbindung hergestellt ist. So wird zusätzlich erreicht, dass der Simulator die Verbindungsanfrage so oft wiederholt, bis die Gegenseite den entsprechenden `ServerSocket` initialisiert hat, was andernfalls einen komplexen Synchronisationsmechanismus erfordert hätte. Zwischen den Verbindungsanfragen wird

250 ms gewartet, was die Prozessorlast und das Datenaufkommen stark reduziert und dennoch einen zügigen Verbindungsaufbau ermöglicht.

```
while (connection == null) {
  try {
    connection = new Socket(hostname,ComProtokol.CONTROL_PORT);
  }
  catch (Exception e) {
    ...
    Thread.sleep(250);
    ...
    System.out.println(
     "KontrollProzessThreadTCP: Retrying to connect with " + hostname
     + ":" + ComProtokol.CONTROL_PORT);
  }
}
```

Bild 252: Verbindungsaufbau mit while-Schleife

Auf Seiten von JSimControl darf die Simulation erst gestartet werden, wenn alle Kommunikationsverbindungen hergestellt wurden. Dies bezieht sich zum einen auf die Verbindung zwischen JSimControl und den Simulatoren, zum anderen aber auch auf die Verbindungen zwischen den einzelnen Simulatoren. Den Simulatoren sind durch die E/A-Konfigurationsdatei (Bild 253) die entsprechenden ein- und abgehenden Verbindungen bekannt.

```
RECEIVERNAME: linux
BUFFERSIZE: 512
TIMEOUT: 15
OUTEVENT: outevt1 1
OUTEVENT: outevt2 1

SENDERNAME: linux
BUFFERSIZE: 2048
INEVENT: inevt1 1
INEVENT: inevt2 1
```

Bild 253: Beispiel einer E/A-Konfigurationsdatei

Dadurch lässt sich feststellen, ob für einen Simulator bereits alle Verbindungen hergestellt wurden. Ist dies der Fall, sendet der Simulator ein `ALL_CONNECTED`-Signal an JSimControl, das mit einem Zähler protokolliert, ob die Verbindungen alle Simulatoren hergestellt sind (Bild 254). Erst dann wird der START-Button der graphischen Oberfläche aktiviert und der Benutzer kann die Simulation starten.

```
if (rcv.compareTo("ALL_CONNECTED") == 0) {
  System.out.println("KontrollProzessThreadTCP: " + hostname
    + " established all connections.");
  ControlProtocol.decrementRemainingSims();
}
```

Bild 254: Verarbeitung des ALL_CONNECTED-Signals bei JSimControl

Kommunikation zwischen den Simulatoren: Zwischen den Simulatoren werden Daten- und Aktionsereignisse ausgetauscht. Das entsprechende Format wird in JStateSim durch die `toString`-Methode der Klasse `ComEvent` erzeugt (Bild 255).

```
public String toString() {
  String result = "";

  result += name + "\n";
  result += type + "\n";
  result += doubleToComString(value) + "\n";
  result += timestamp + "\n";
  result += antM + "\n";
  result += isToRun;

  return result;
}
```

Bild 255: toString-Methode von ComEvent

Wird ein solches Ereignis empfangen, muss es wieder in ein `ComEvent`-Objekt konvertiert werden. Hierfür kommt ein spezieller Konstruktor mit Parser zum Einsatz, der die empfangene Zeichenkette als Parameter übernimmt und die Zeilen auswertet (Bild 256).

```
public ComEvent(String str) {
  String[] subStrings;
  int index;
  subStrings = str.split("\n");
  String doubleComString = "";
  index = 0;

  name = subStrings[index++];
  type = Integer.parseInt(subStrings[index++]);
  while (index < subStrings.length - 3) {
    doubleComString += subStrings[index++] + "\n";
  }
  value = comStringToDouble(doubleComString);
  timestamp = Long.parseLong(subStrings[index++]);
  antM = Boolean.getBoolean(subStrings[index++]);
  isToRun = Boolean.getBoolean(subStrings[index++]);
}
```

Bild 256: Parser für empfangene Ereignisse

Kommunikation zwischen Simulator und JSimControl: Folgende Befehle sind als Steuerprotokoll implementiert:

- `START`: nachdem alle Simulatoren initialisiert sind, kann die Simulation mit `START` aktiviert werden.
- `PAUSE`: um die Simulation vorübergehend anzuhalten wird `PAUSE` verwendet.
- `CONTINUE`: mit `CONTINUE` wird eine vorübergehend angehaltene Simulation wieder aktiviert.
- `STOP`: nach dem `STOP`-Befehl kann die Simulation nicht mehr aktiviert werden
- `QUIT`: Abbauen der Datenverbindungen und Beenden der Threads geschieht bei Empfang von `QUIT`.
- `CALC_LOCALTIME`: da die Berechnung der lokalen Simulationszeit aus Gründen der Thread-Synchronisation vorbereitet werden muss, wird mit `CALC_LOCALTIME` signalisiert, dass die lokale Simulationszeit in Kürze abgefragt wird.
- `GET_LOCALTIME`: mit `GET_LOCALTIME` wird die nach Empfang von `CALC_LOCALTIME` berechnete lokale Simulationszeit abgefragt.
- `LOCALTIME`: der Simulator sendet angehängt an das Schlüsselwort `LOCALTIME` seine lokale Simulationszeit.
- `NEWGVT`: JSimControl sendet angehängt an das Schlüsselwort `NEWGVT` die neue GVT.

- `GET_STATISTIK`: der Befehl `GET_STATISTIK` wird verwendet, um nach `STOP` Informationen über den Simulationsablauf von JStateSim abzurufen.

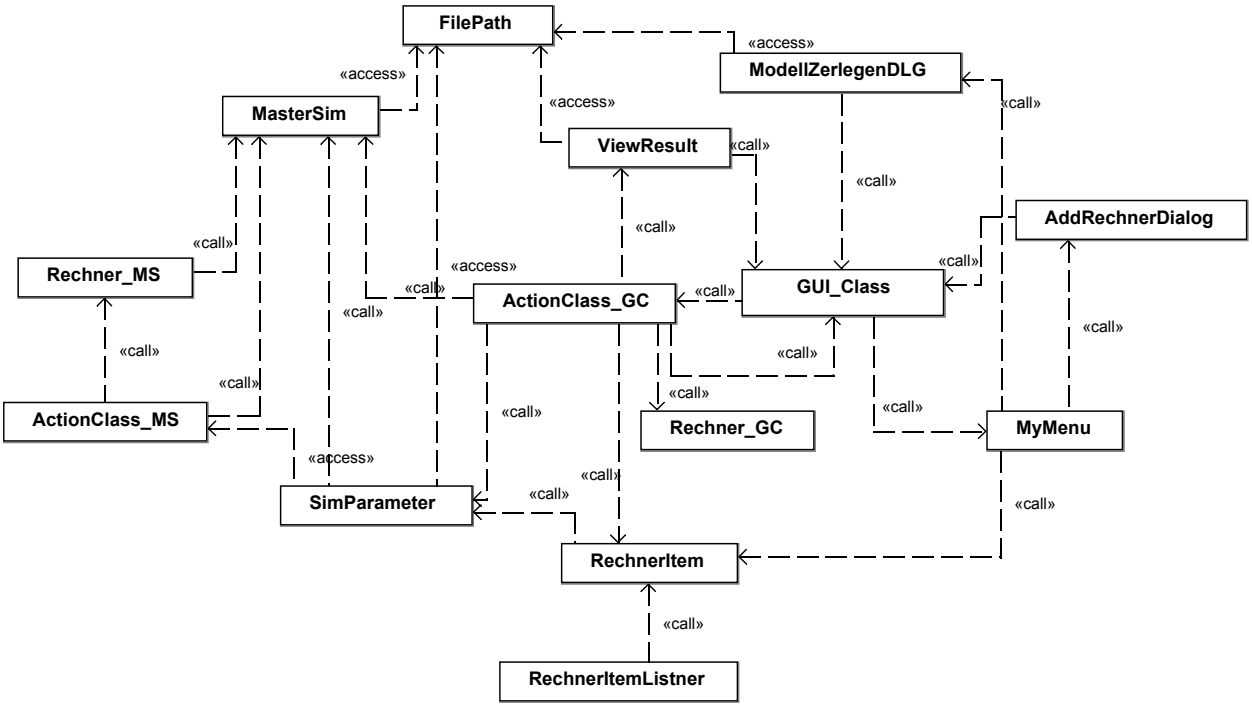


Bild 257: JSimControl-Klassendiagramm

Klassendiagramme des Werkzeugs JStateSim:

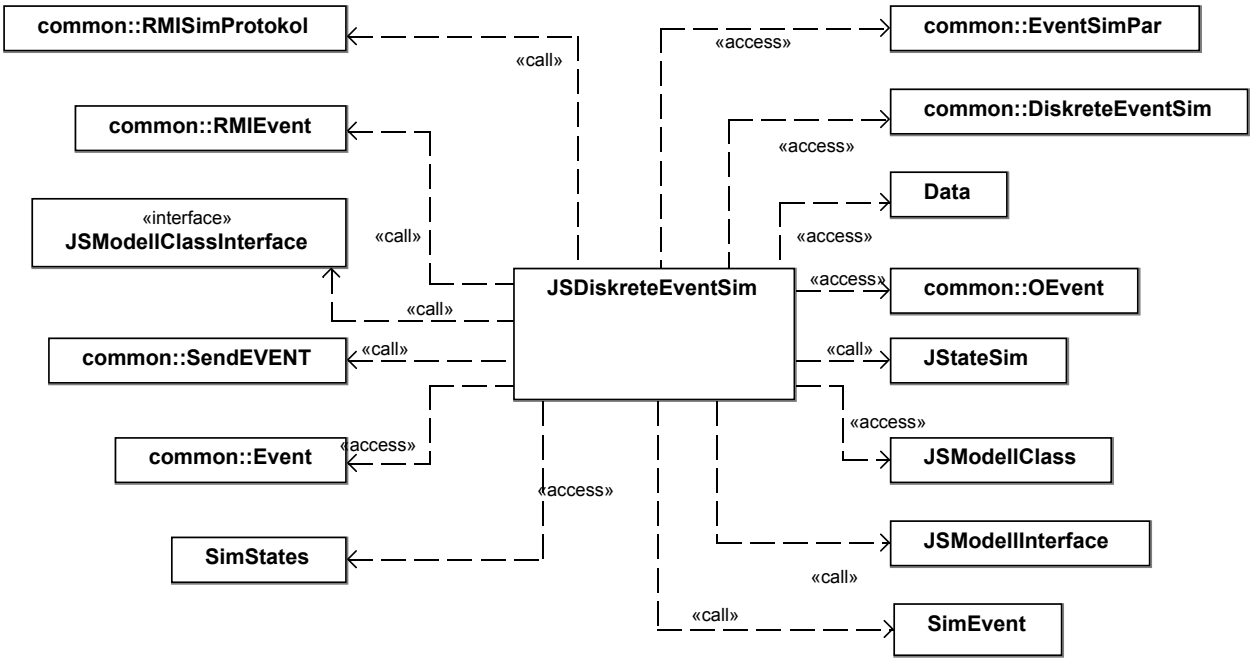


Bild 258: JStateSim-Klassendiagramm (1)

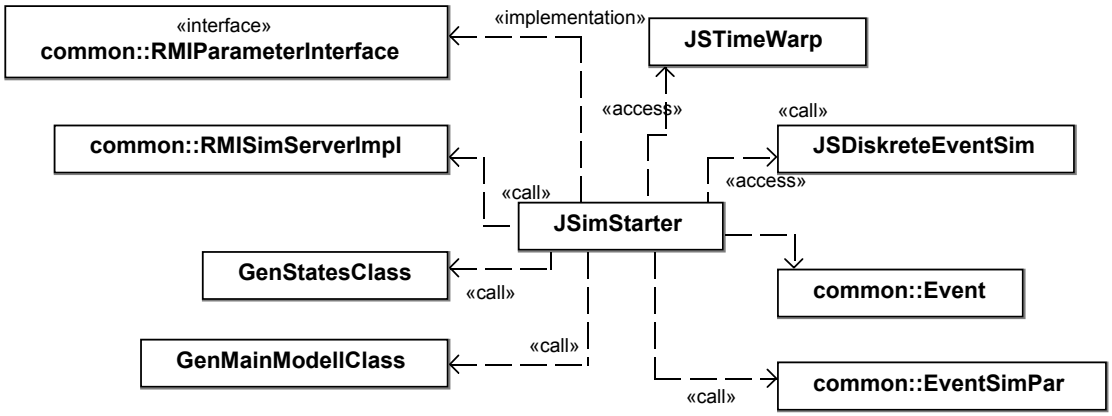


Bild 259: JStateSim-Klassendiagramm (2)

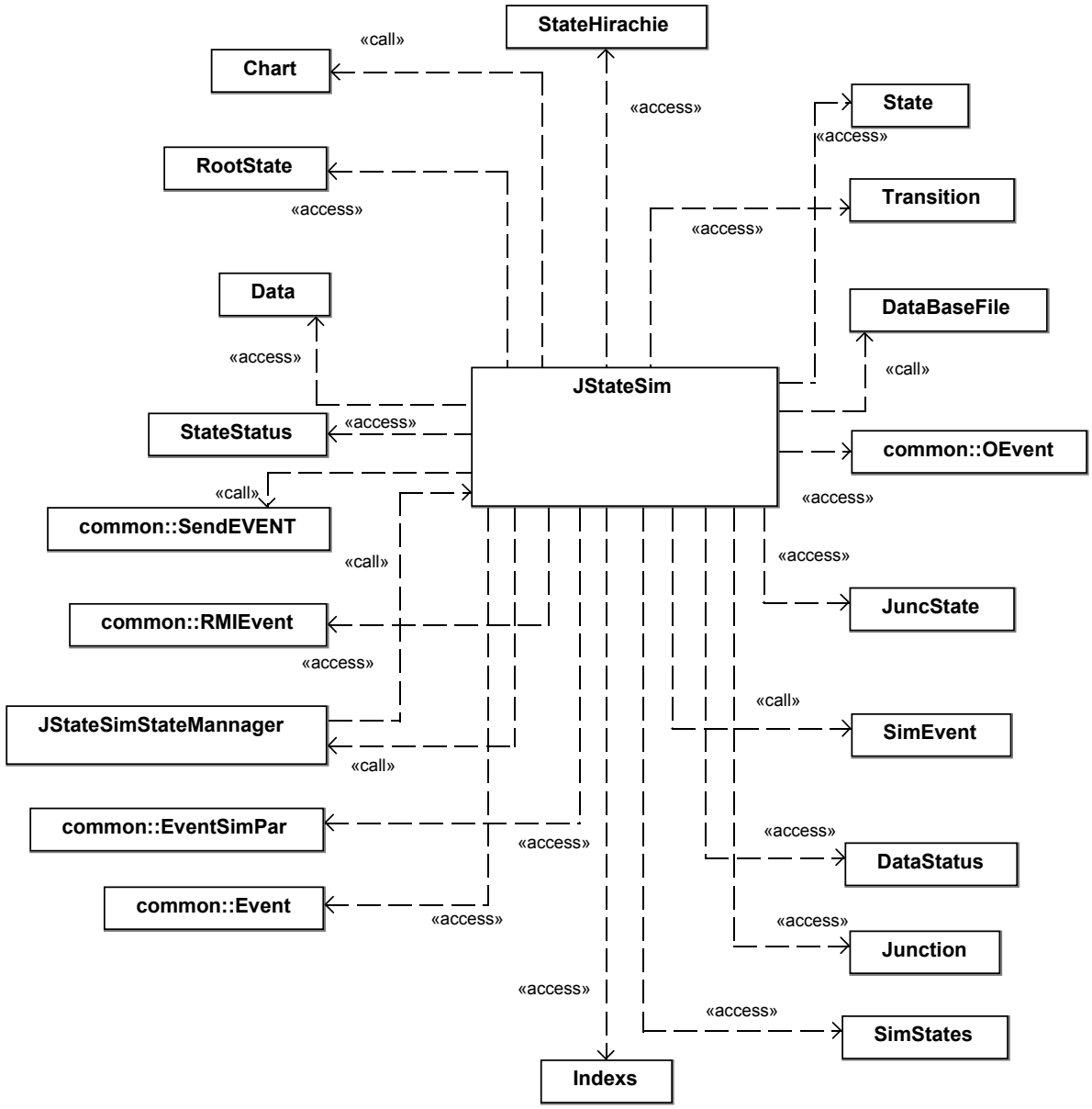


Bild 260: JStateSim-Klassendiagramm (3)

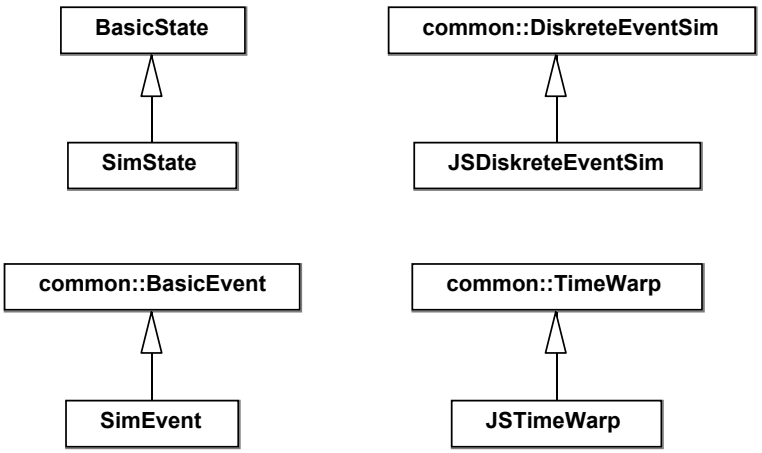


Bild 261: JStateSim-Klassendiagramm (4)

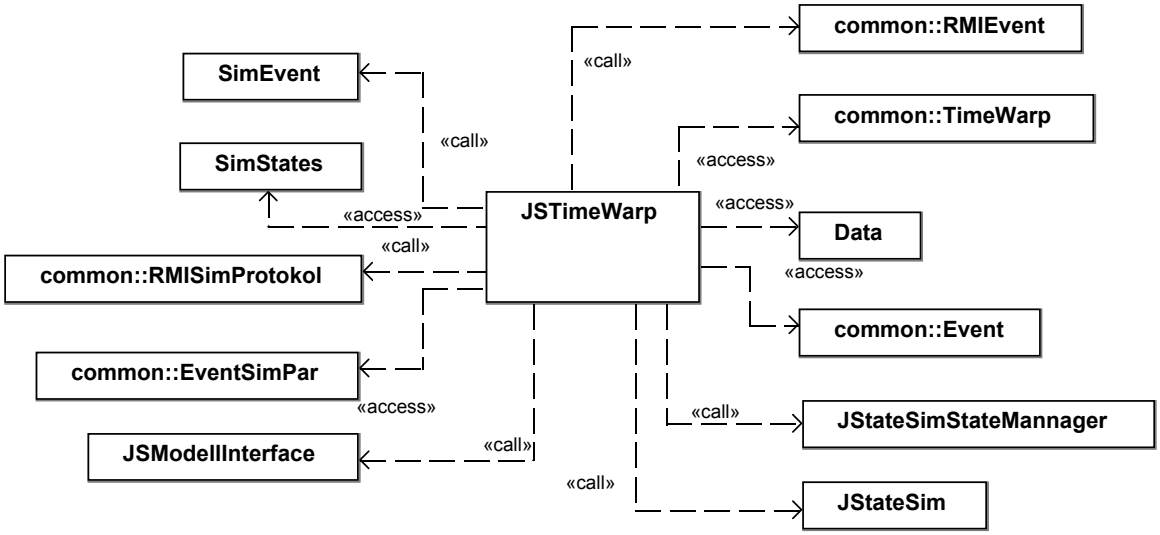


Bild 262: JStateSim-Klassendiagramm (5)

D.11 Simulink-Anbindung an das Co-Simulations-Framework mittels TCP/IP

Für die Möglichkeit einer interaktiven Beeinflussung der Simulation und Darstellung von Ergebnissen in Simulink wurde TCP/IP-Funktionalität in Simulink integriert. Der im Internet frei erhältliche Simulink-Block Remote-IO [Zimm03] ermöglicht den Datenaustausch von Simulink-Modellen mit anderen TCP/IP-Kommunikationsteilnehmern. Hierzu wird dieser Block in ein Modell eingefügt und mit den Signalen verknüpft, die über das Netzwerk gesendet oder empfangen werden sollen. Der Block fungiert dabei als TCP/IP-Server, der auf eingehende Verbindungsanfragen auf dem in der Blockparameter-Maske konfigurierten Port wartet und nach Verbindungsaufbau Signale, die mit den Blockeingängen verbunden sind, an den Remote-Client sendet. Die vom Remote-IO-Block über TCP/IP empfangenen Daten werden an die Blockausgänge gelegt und somit den daran angeschlossenen Signalen zur Verfügung gestellt. Anzahl der Ein- und Ausgänge müssen in den Block-Parametern eingestellt werden, wobei maximal 16 Ausgangssignale angeschlossen werden können.

Es kann ein zweiter Remote-IO-Block in einem anderen Modell verwendet werden, das auf einer zweiten MATLAB-Engine läuft und über TCP/IP Daten mit dem ersten Modell austauscht. In diesem Fall arbeitet ein Block als Client, bei dem die Host-Adresse des Rechners, auf dem das Modell mit dem Server-Block ausgeführt wird, konfiguriert werden muss. Laufen beide Modelle auf demselben Rechner, wird `'localhost'` als Host-Adresse verwendet. Im Server-Betrieb wird im Feld `Host address` eine leere Zeichenkette eingetragen, also `''`. Das Format der ausgetauschten Daten ist `"t=d_value\nch1=d_value\nch2=d_value\n ...\n"`, wobei `d_value` für einen als Zeichenkette übertragenen Fließkommawert steht. Zusätzlich stehen die Befehle `"cmd=P\n"` zum Pausieren

der Simulink Simulation und "cmd=C\n" zur Fortsetzung zur Verfügung. Gesendet wird mit der konfigurierten Abtastrate. Empfangene Daten werden beim auf den Empfang folgenden Simulationsschritt an den Ausgang gelegt [Zimm03]. Da zu jeder Zeit immer nur ein Client verbunden sein kann und auch die Ausgangsanzahl auf 16 begrenzt ist, konnte dieser Block die in dieser Arbeit gestellten Anforderungen nicht erfüllen. Daher war es notwendig, einen eigenen Funktionsblock zu implementieren.

Konfiguration von Sendern und Empfängern: Auf dem Empfangskanal werden alle Daten eines Simulators empfangen. Entsprechend gibt es einen Ausgangskanal, auf dem Simulink alle für einen Simulator vorgesehenen Daten sendet. Da die ein- und ausgehenden Daten über Ein- und Ausgangsports des S-Funktion-Blocks eingelesen, bzw. ausgegeben werden, sind Konfigurationsangaben nötig, über die die Daten den Ports zugeordnet werden können. Dies geschieht über entsprechende Konfigurationsdateien, die zu Beginn der Simulation vom Kontrollprozess an Simulink übertragen werden. Da beim Empfangen von Daten auf eingehende Nachrichten gewartet wird, kommt die Anwendung an dieser Stelle in einen Wartezustand, der erst beim Eingang neuer Daten verlassen wird. Es ist daher eine Auslagerung des Datenempfangs in einen weiteren, vom Simulations-Primärtask unabhängigen Empfangstask notwendig, so dass auf eingehende Daten gewartet werden kann, ohne den Primärtask zu blockieren. Da mehrere Simulatoren Daten empfangen können sollen, und für jeden Simulator ein eigener Empfangskanal existiert, muss jeder Empfangskanal in einer eigenen Empfangstask abgearbeitet werden. Da die Primärtask Daten darstellt, auf denen auch die Empfangstask operiert, muss bei der Implementierung ein Mechanismus zur Sicherung der Datenkonsistenz integriert werden. Für die vorliegende Realisierung kamen Mutexe zum Einsatz.

Filterung gültiger Daten: Die Bedingung, dass nur gültige Daten dargestellt werden sollen, lässt sich grundsätzlich auf zwei Arten lösen, wobei unterschieden wird, ob die Prüfung auf Datengültigkeit auf Senderseite beim Simulator oder auf Empfängerseite in Simulink durchgeführt wird. Im Folgenden werden die relevanten Aspekte beider Varianten erörtert und erklärt, wieso für eine Datengültigkeitsprüfung auf Senderseite entschieden wurde. Werden die Daten auf Empfängerseite, also in Simulink gefiltert, müssen zunächst alle Daten – auch die ungültigen – übertragen werden, was einen erhöhten Kommunikationsaufwand bewirkt. Zusätzlich ist in Simulink die Implementierung eines Mechanismus notwendig, der die Gültigkeit der Daten prüft. Dies ließe sich z. B. durch Empfang und Verarbeitung von Negativnachrichten realisieren, stellt aber einen vermeidbaren Implementierungsaufwand dar. Durch Filterung auf Simulatorseite lässt sich das Senden ungültiger Daten vermeiden, da bei den Simulatoren Mechanismen zur Datenvalidierung ohnehin im Rahmen des optimistischen Simulationsalgorithmus implementiert sind. So ist auf Simulatorseite bei Erreichen der GVT die Gültigkeit von Daten gewährleistet, deren Zeitstempel kleiner oder gleich der GVT ist und die nicht durch Negativnachrichten oder Rücksetzungen gelöscht wurden. Ein Sammeln dieser Daten lässt sich mit geringem Aufwand realisieren, und auch die Implementierung der notwendigen Aktualisierungen der Datensammlung durch Negativnachrichten und Rücksetzungen ist eine überschaubare Aufgabe. Es ist ein Identifikator für die Daten notwendig, die an Simulink gesendet werden sollen, da dem sendenden Simulator kenntlich gemacht werden muss, welche Daten an Simulink gesendet werden sollen. Diese Funktionen sind an entsprechender Stelle auf Simulatorseite zu realisieren.

Zeitdomänen bei der Simulation: Da im selben Modell unsynchronisiert Ausgangsdaten gesendet und Eingangsdaten dargestellt werden sollen, muss in unterschiedlichen Zeitdomänen operiert werden. Hinzu kommt, dass die Modellausführung in Simulink kontinuierlich fortlaufen muss, was bedeutet, dass die interne Simulationszeit von Simulink unabhängig vom Simulationszustand (INIT, START, STOP, ...) ständig voranschreitet und nicht als Referenzzeit verwendet werden kann. Für die zu sendenden Daten wird eine eigene Zeitvariable eingeführt, die bei jedem Simulationsschritt, in dem Daten gesendet werden, um eins erhöht wird. Die empfangenen Daten enthalten einen Zeitstempel, so dass dieser direkt übernommen werden kann. Da die Simulationszeit von Simulink nicht identisch mit der Zeitdomäne der empfangenen Daten ist, können keine `Scope` Blöcke von Simulink zur Darstellung der empfangenen Daten verwendet werden, weil dort die X-Achse die Simulations-

zeit von Simulink darstellt. Es kommen daher XY-Graph Blöcke zum Einsatz, wobei auf der X-Achse die Zeitstempel und auf der Y-Achse die Werte der empfangenen Daten aufgetragen werden (Bild 263).

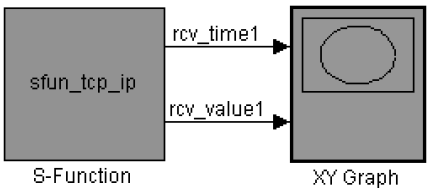


Bild 263: Darstellen der empfangenen Daten mit XY Graph

Die gesamte Funktionalität wird in einer S-Funktion realisiert. Da die Anzahl der Ein- und Ausgänge des S-Funktions-Blocks statisch im Code festgelegt wird, müssen diese Schnittstellenangaben bereits zum Compilierzeitpunkt der S-Funktion feststehen. Diese Werte sind in der Header-Datei `com_config.h` manuell einzustellen[75].

Kommunikationsablauf: Den grundsätzlichen Kommunikationsablauf einer INTERACT-Simulation in Hinblick auf ein eingebundenes Simulink-Modell zeigt Bild 264.

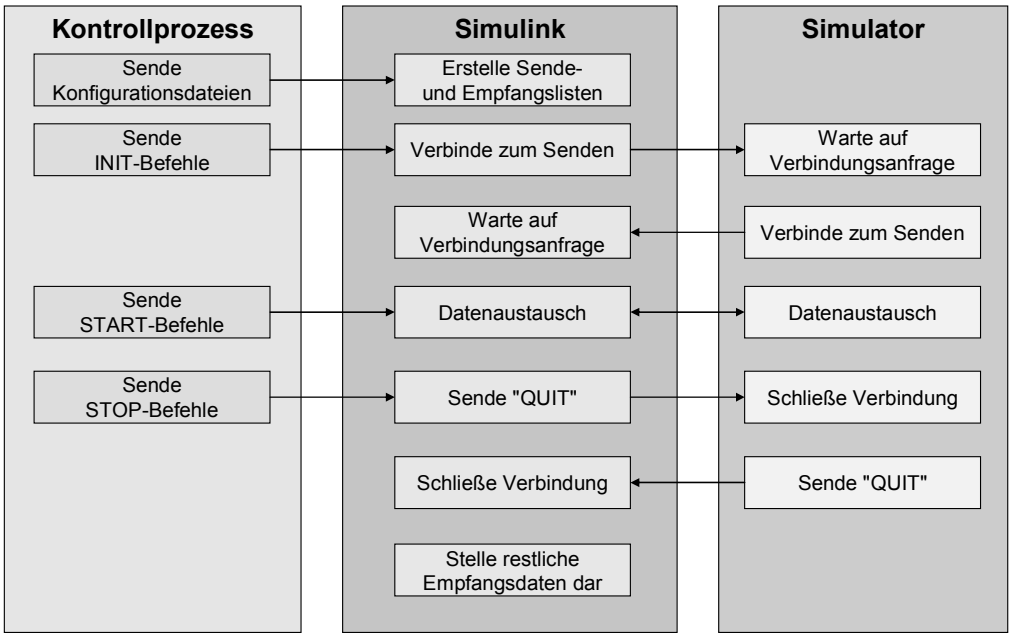


Bild 264: Kommunikation zwischen Simulink, Kontrollprozess und Simulator

Das Simulink-Modell muss – so wie die Simulatoren – gestartet sein, bevor JSimControl eine Verbindung auf Port 3000 anfordern kann. Wird in JSimControl `INIT` aktiviert, werden zunächst die Konfigurationsdateien auf Port 3010 an Simulink übertragen. Dann fordern die als Send-Host angegebenen Simulatoren Verbindungen auf Port 3020 an. Im Anschluss fordert Simulink Verbindungen mit den als Receive-Host konfigurierten Simulatoren auf Port 3020 an. Nun können Stimuli gesendet und Ergebnisse empfangen werden.

Anpassungen bei JSimControl: Simulink wird in JSimControl wie ein Simulator ohne optimistisches Simulationsverfahren betrachtet. In der graphischen Benutzeroberfläche ist Simulink als zusätzlicher Simulatortyp einzubinden. Wenn Simulink als Stimuli-Sender agiert, muss die Senderate

[75] Alternativ hätte ein S-Funktions-Block mit je einem Ein- und Ausgang verwendet werden können, bei dem die Eingangsdaten durch einen vorgeschalteten Multiplexer-Block am Eingang gebündelt und die Ausgangsdaten mit einem mit dem Ausgang verbundenen Demultiplexer-Block getrennt werden. Allerdings ist der Zugriff auf gemultiplexte Daten innerhalb der S-Funktion aufwendiger und die Handhabung fehleranfälliger.

kontrolliert werden, um nicht zu viele Stimuli zu erzeugen. Hierfür wird bei GVT-Aktualisierung auch die Simulationszeit von Simulink abgefragt. Liegt diese zu weit vor der aktuellen GVT, wird ein `PAUSE`-Befehl gesendet und Simulink durch `CONTINUE` erst reaktiviert, wenn die Differenz zwischen GVT und der Simulationszeit von Simulink wieder einen konfigurierbaren Wert erreicht hat. Wird Simulink nur als Empfänger eingesetzt, darf die Simulationszeit von JSimControl nicht beachtet werden. Um dies zu signalisieren, wird von Simulink bei GVT-Aktualisierung ein spezieller Wert gesendet, den JSimControl entsprechend auswertet.

Ablaufsteuerung: Der Simulationsablauf wird in der Callback-Methode `mdlOutputs` der Datei `sfun_tcp_ip.c` kontrolliert. Beim erstmaligen Aufruf werden grundlegende Initialisierungsvorgänge durchgeführt: zunächst wird die WinSock-DLL (Dynamic-Link Library) geladen, um die TCP/IP-Funktionalität von Windows nutzen zu können. Nun wird der Socket zum Empfang der Konfigurationsdaten erzeugt und anschließend diese Daten empfangen. Es folgt die Erzeugung des Threads zum Empfang der Steuerbefehle. Danach werden die TCP/IP-Verbindungen zum Senden der Stimuli und Empfangen der Simulationsergebnisse hergestellt. Sind alle Verbindungen etabliert, wird dies mit dem `ALL_CONNECTED`-Signal JSimControl mitgeteilt. Zuletzt wird noch der Speicher für einige Datenelemente allokiert. Bei allen weiteren Aufrufen von `mdlOutputs`, werden zunächst pro Simulationsschritt einmal die Werte an den Eingängen des S-Funktions-Blocks eingelesen. Dann wird überprüft, ob der Thread zum Empfang der Steuerbefehle noch aktiv ist. Falls ja, wird geprüft, ob ein neuer, gültiger Befehl empfangen wurde. Bild 265 zeigt die erlaubten Zustandsübergänge, die durch die entsprechenden Befehle ausgelöst werden.

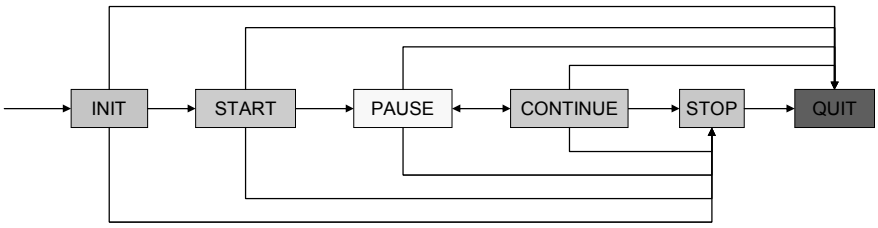


Bild 265: Zustandsübergangsdiagramm der TCP/IP S-Funktion

Bei Empfang des `START`-Befehls kann das Senden der Stimuli und die Darstellung der empfangenen Daten beginnen. Mit `PAUSE` wird das Senden unterbrochen und mit `CONTINUE` wieder fortgesetzt. `STOP` beendet das Senden endgültig, während der Empfangsthread weiter aktiv sein kann. Erst wenn alle Verbindungen zum Empfangen von Ergebnisdaten seitens der sendenden Simulatoren abgebaut wurden oder der `QUIT`-Befehl empfangen wurde, wird die Ausführung vollständig beendet und das Modell in den Ausgangszustand zurückgesetzt. Der Ablauf innerhalb eines Simulationsschritts ist immer gleich: zunächst werden Daten am Blockeingang übernommen, dann wird auf einen neu eingegangenen Befehl überprüft. Ist der aktuelle Befehl `START` oder `CONTINUE`, werden die am Blockeingang anliegenden Werte an die Funktion `dataSend()` übergeben, die für das Senden der Stimuli zuständig ist. Zum Schluss werden die eingegangenen Daten mit kleinstem Zeitstempel an die Blockausgänge gelegt.

Empfang der Konfigurationsdaten: Die Zuordnung der Ein- und Ausgangsdaten zu den entsprechenden Ein- und Ausgangsports des S-Funktion-Blocks geschieht mit der E/A-Konfigurationsdatei (Bild 253). Hier wird angegeben, welche Daten an welche Rechner gesendet und von welchen Rechnern empfangen werden. `BUFFERSIZE` gibt an, wie groß der Puffer beim Senden bzw. Empfangen sein soll. Mit `TIMEOUT` wird festgelegt, nach wievielen Simulationsschritten die Stimuli spätestens gesendet werden. Der Inhalt dieser Konfigurationsdateien wird durch die in der Datei `configRcv.c` implementierten Funktion `rcvConfig()` empfangen und interpretiert. Es resultiert eine Liste mit Datenelementen vom Typ `HostItem` (Bild 266). Darin sind alle relevanten Informationen zusammengefasst, die zum Datenaustausch mit je einem konfigurierten Sender oder Empfänger benötigt werden, z. B. der Hostname, ein Zeiger auf den Verbindungssocket, die Puffergröße und der

eigentliche Datenpuffer. Ein wichtiges Element ist die `EventDefList`-Liste, die für jedes konfigurierte Ereignis einen Eintrag vom Typ `EventDefItem` enthält (Bild 267).

```
struct HostItem {
      struct HostItem           *next;
      char                      *name;
      char                      *addr_str;
      SOCKET              *hostS;
      int                       bufferSize;
      int                       timeout;
      struct EventDefList *evtDefList;
      char                      *comText;
};
```

Bild 266: Datenstruktur HostItem

`EventDefItem` speichert die Eigenschaften der in der Konfiguration definierten Ereignisse: den Ereignisnamen, den Typ und die Nummer des entsprechenden Eingangs- oder Ausgangports des S-Funktions-Blocks.

```
struct EventDefItem {
  struct EventDefItem           *next;
  char                    *name;
  char                    *typ;
  int                           portIndex;
  struct EventDataList    *evtDataList;
};
```

Bild 267: Datenstruktur EventDefItem

Bild 268 verdeutlicht nochmals graphisch die Zusammenhänge der einzelnen Datenstrukturen.

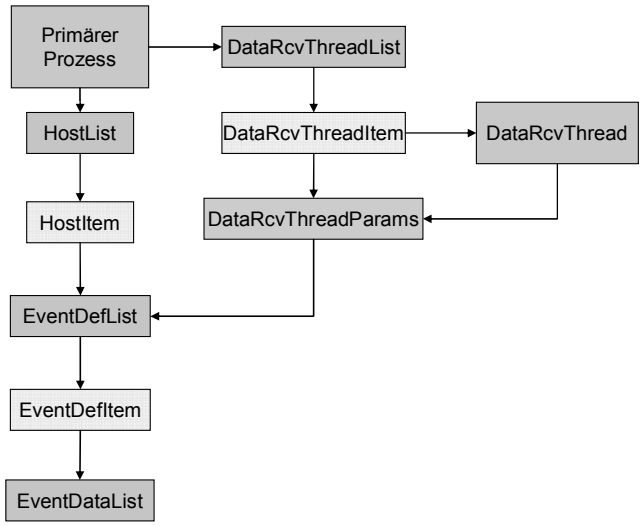


Bild 268: Abhängigkeiten und Verwendung der Datenstrukturen

Die empfangenen Ereignisse werden in der `EventDataList`-Liste als Elemente vom Typ `EventDataItem` abgelegt (Bild 269). Hier werden Wert und Zeitstempel des empfangenen Ereignisses gespeichert.

```
struct EventDataItem {
      struct EventDataItem *next;
      double value;
      int          timeStamp;
};
```

Bild 269: Datenstruktur EventDataItem

Empfang der Kontrolldaten: Das Empfangen der vom Kontrollprozess gesendeten Steuerbefehle soll zu jeder Zeit der Simulation möglich sein. Daher wird die hierfür notwendige Funktionalität in einem eigenen Thread ausgeführt, der in `commandRcvThread.c` implementiert ist. Um sicherzustellen, dass der primäre Prozess nicht zeitgleich mit dem Thread auf die Befehlsvariable zugreift, wird diese mit einem Mutex gesichert. In der Win32-API wird ein Mutex mit `CreateMutex()` erzeugt, das Sperren erfolgt mit `WaitForSingleObject()`, das Befreien mit `ReleaseMutex()` (Bild 270).

```
...
HANDLE hCmdMutex = CreateMutex(NULL, FALSE, "CMDMUTEX");
...
WaitForSingleObject(hCmdMutex, INFINITE);     // Lock cmd with Mutex
thread_param->cmdP = cmd;                     // Read cmd
ReleaseMutex(hCmdMutex);                      // Unlock cmd
...
```

Bild 270: Verwendung von Mutex der Win32-API

Senden der Stimuli: Bevor die Stimuli an die Simulatoren gesendet werden können, werden mit der Funktion `initDataSendSockets()` die Kommunikationsverbindungen aufgebaut. Diese Funktion kehrt erst zurück, wenn alle konfigurierten Empfänger verbunden sind.

Pro Simulationsschritt werden die an den Blockeingängen anliegenden Daten an die Funktion übergeben, in der zunächst geprüft wird, ob sich der Wert eines Eingangsports im Vergleich zum Simulationsschritt davor geändert hat. Ist dies der Fall, wird das `HostItem` des für den Empfang dieses Werts konfigurierten Hosts geladen und ein Ereignis in textueller Darstellung erzeugt. Überschreitet die Größe des Sendepuffers einen konfigurierten Wert oder liegt der letzte Sendevorgang länger als ein ebenfalls festgelegtes Zeitintervall zurück, wird der Inhalt des Sendepuffers übertragen.

```
char * doubleToComString(double dbl, char *dblText) {
	int i;

	// Copy double into first 8 Byte of char[]
	memcpy (dblText, &dbl, sizeof(double));

	// Negative value and 0 value not allowed
	// -> make all Bytes positive
	for (i = 0; i < 8; i++) {
		if (dblText[i] == -128) {
			dblText[i] = 1;
			dblText[i+8] = 3; // Mark -128 value
		}
		else if (dblText[i] < 0) {
			dblText[i] += (char) 128;
			dblText[i+8] = 1; // Mark negative value
		}
		else if (dblText[i] == 0) {
			dblText[i] = 1;
			dblText[i+8] = 2; // Mark zero value
		}
		else {
			dblText[i+8] = 4; // No modification
		}
	}
	return dblText;
}
```

Bild 271: Modifikationen am IEEE 754-Format

Da das Umwandeln der Fließkomma-Zahl in eine ASCII-Darstellung zu Ungenauigkeiten führen kann, werden Werte in Fließkomma-Darstellung nicht als ASCII-Text übertragen, sondern in einer

modifizierten Variante des IEEE 754-Formats gesendet: da im eigentlichen IEEE 754-Format z. B. ein Bytewert erlaubt ist, der in C als Textendezeichen (`'\0'`) interpretiert wird, muss dieser Bytewert modifiziert werden, um innerhalb eines ASCII-Strings versendet werden zu können. Andernfalls würde diese Stelle als Textende ausgewertet und das Parsen abgebrochen werden. Die Modifikationen werden in acht Bytes markiert, die den acht Bytes folgen, die die eigentliche Fließkommazahl darstellen (Bild 271). Auf Empfängerseite müssen die Modifikationen wieder rückgängig gemacht werden.

Empfang der Simulationsergebnisse: Für jeden Host, der als Sender konfiguriert ist, gibt es einen Thread, der für den Datenempfang von diesem Host verantwortlich ist. Die Implementierung findet sich in `dataRcvThread.c`. Da alle Simulatoren denselben Port kontaktieren, um eine Verbindung mit Simulink aufzubauen, können die Verbindungsanfragen nur nacheinander bearbeitet werden. Um den Programmfortschritt nicht zu blockieren, bis alle Verbindungsanfragen eingegangen sind, wird die Funktionalität in einen eigenen Thread `initDataRcvThreads` ausgelagert (Bild 272).

```
DWORD WINAPI initDataRcvThreads(LPVOID lpParam) {
  ...
  for (i=0; i<nrHosts; i++) {
    drtSocket = (SOCKET *) calloc(1, sizeof(SOCKET));
    ...
    *drtSocket = accept(sListen, (struct sockaddr *) &client,
                                 &iAddrSize);
    ...
    params->sockP = drtSocket;
    ...
    hThread = CreateThread(NULL, 0, DataRcvThread,  (LPVOID)params, 0,
                                 &threadid);
    ...
}
```

Bild 272: Thread zur Verarbeitung von Verbindungsanfragen

Dort wird für jede eingehende Verbindungsanfrage ein eigener Thread `DataRcvThread` erzeugt und der Verbindungssocket als Parameter übergeben. Die empfangenen Ereignisse werden als Elemente vom Typ `EventDataItem` in der `EventDataList`-Liste gespeichert. Bei jedem Simulationsschritt wird `getRcvData()` ausgeführt: darin wird über alle `EventDefItem`-Elemente iteriert und, falls vorhanden, das erste Element der jeweiligen `EventDataList` an den entsprechenden Ausgangsport gelegt. Auch diese Datenelemente müssen durch Mutexe vor Mehrfachzugriff geschützt werden.

Anpassung, Erstellung und Verwendung der S-Funktion: Durch Angaben im Quellcode wird die Zahl der Ein- und Ausgänge des S-Funktions-Blocks statisch festgelegt. Die Anpassung dieser Werte muss vor der Compilierung in der Datei `com_config.h` erfolgen (Bild 273).

```
// Anzahl der Eingänge des S-Function-Blocks
#define NR_SEND 1

// Anzahl der Ausgänge des S-Function-Blocks
#define NR_RCV 2
```

Bild 273: Konfiguration der Ein- und Ausgänge in com_config.h

D.12 Modifikation des RTW-Codes zur optimistischen Co-Simulation

Eingriff in die Steuerung des Simulationsablaufs: RTW stellt eine `main`-Funktion in der Datei `grt_main.c` bereit, die für jedes Modell identisch ist und daher bei der Code-Generierung nicht jeweils neu erzeugt werden muss, sondern aus dem Installationsverzeichnis von RTW übernommen, compiliert und zu den anderen Binärdateien gelinkt wird. Für Änderungen in dieser Datei wurde sie zunächst kopiert und im RTW-Framework-Verzeichnis und `main.cpp` gespeichert. Nun konnten in der `main`-Funktion alle notwendigen Anpassungen der Ablaufsteuerung durchgeführt werden. Zu-

sätzlich zu der von RTW vorgegebenen Initialisierung wird in einer ersten zusätzlichen Initialisierungsphase eine TCP/IP-Verbindung zum Kontrollprozess aufgebaut und die Konfigurationsdaten empfangen. Dann wird ein Mutex zum geschützten Auslesen der eingegangenen Steuerbefehle erzeugt und der Thread zum Empfang dieser Befehle initialisiert. Zuletzt wird der Speicher für einige Datenstrukturen allokiert (Bild 274).

```
if (initConfigRcvSocket(&sConfig) == 0) {
      printf("\nFailed to initialize ConfigRcvSocket\n");
      exit(1);
}
if (rcvConfig(&sConfig, &sendList, &rcvList) == 0) {
      printf("\nFailed to receive + parse Config-Files\n");
      exit(1);
}

pthread_mutex_init( &cmdMutex, NULL );
initCommandRcvThread(&rcParams, &cmdMutex, &rcThread);

initRTW_Arrays();
dataToSend = (double*) calloc(NR_SEND, sizeof(double));
oldDataSend = (double*) calloc(NR_SEND, sizeof(double));
for (i=0; i<NR_SEND; i++) {
      oldDataSend[i] = -1.1;
}
dataRcv = (real_T*) calloc(NR_RCV, sizeof(real_T));
```

Bild 274: Erste zusätzliche Initialisierungsphase bei RTW

Nun kann die Simulation beginnen. Zu Beginn eines Simulationsschritts wird geprüft, ob der Befehlsempfänger-Thread noch aktiv ist. Dies wird in POSIX[76] durch die in Bild 275 gezeigte Befehlsfolge realisiert.

```
exitCode = pthread_kill (rcThread, 0);
if (exitCode != ESRCH) { // rcThread is still alive
...
}
```

Bild 275: Prüfen auf Aktivität eines POSIX-Threads

Es folgen die fünf grundsätzlichen Aktionen innerhalb eines Simulationsschritts:

- Prüfen auf neu eingegangenen Befehl
- Übernahme der empfangenen Werte an den Eingangsports
- Berechnung der Werte der Ausgangsports
- Übermittlung der neuen Werte der Ausgangsports an die `send`-Funktion
- Zustandsspeicherung für die aktuelle Simulationszeit

Wie bei der TCP/IP-S-Funktion gibt es auch hier sechs definierte Befehle. Der jeweils aktuelle Befehl wird vom `CommandRcvThread` Mutex-geschützt ausgelesen (Bild 276).

```
pthread_mutex_lock (&cmdMutex); // Lock cmd with Mutex
cmd = rcParams.cmdP; // Copy cmd
pthread_mutex_unlock (&cmdMutex);
```

Bild 276: Durch einen POSIX-Mutex geschützter Variablenzugriff

[76] Für UNIX/Linux-Systeme stellt IEEE POSIX 1003.1c von 1995 eine standardisierte Programmierschnittstelle dar. Implementationen, die diesem Standard entsprechen, werden POSIX-Threads oder Pthreads genannt. Sie definieren ein Set von Datentypen und Prozeduren in C, implementiert durch die Header-Datei `pthread.h` und eine Thread-Library.

Bei Empfang von INIT wird eine zweite Initialisierungsphase durchlaufen, in der die Datenverbindungen zu den anderen Simulatoren etabliert werden. Dann kann durch START die Simulation begonnen werden. PAUSE unterbricht die Ausführung, CONTINUE setzt sie wieder fort. STOP beendet die Simulation endgültig, hält die Anwendung aber noch aktiv. Erst mit QUIT wird der Prozess beendet.

Datenstrukturen zum Datenaustausch: Ein- und ausgehende Daten werden in speziellen Datenstrukturen gespeichert, die als Elemente in STL[77]-Container eingefügt sind (Bild 277). Daher kommen in C++ implementierte Klassen zum Einsatz.

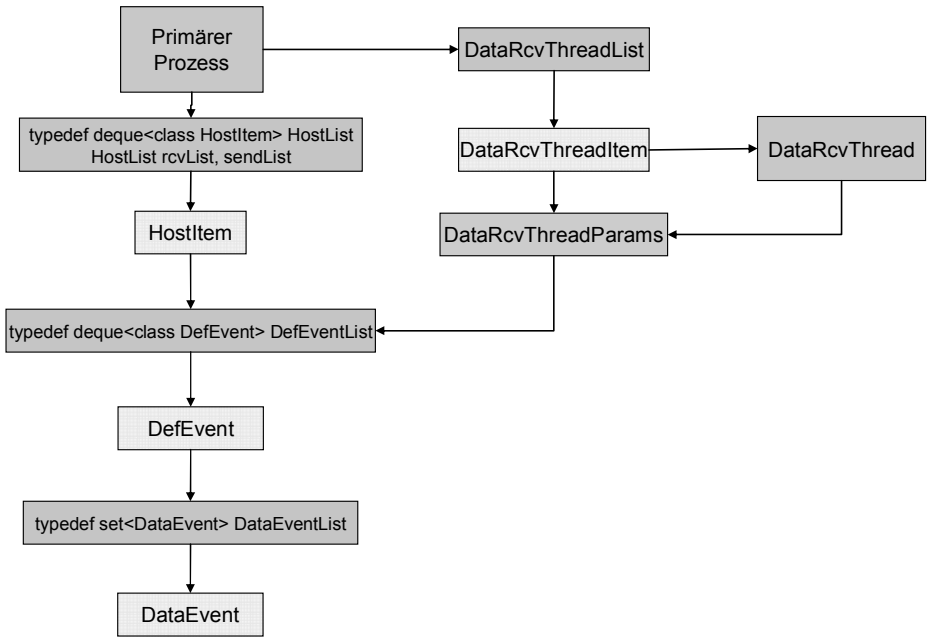


Bild 277: Datenstrukturen zum Datenaustausch mit RTW

Die Klasse `HostItem` repräsentiert die in `rtw_rcv_config.txt` und `rtw_send_config.txt` konfigurierten Kommunikationspartner. Objekte vom Typ `DefEvent` stehen für die definierten Ereignisse, während `DataEvent` ein tatsächlich ausgetauschtes Ereignis mit Wert und Zeitstempel enthält. Objekte vom Typ `HostItem` und `DefEvent` sind in einfach verketteten Listen abgelegt, realisiert als STL-Container `deque`. Der STL-Container `set`, ein Binärbaum, kommt bei `DataEvents` zum Einsatz, da alle Objekte dieses Typs schnell auffindbar sein müssen. STL speichert nicht das Original-Objekt, sondern immer einen Klon. Hierfür muss für die in einen STL-Container abzulegende Klasse ein Copy-Konstruktor implementiert sein, da STL diesen zum Klonen der Objekte verwendet. Ein Beispiel für einen solchen Copy-Konstruktor zeigt Bild 278.

```
DataEvent::DataEvent(const DataEvent &evt) {
       value              = evt.value;
       timeStamp          = evt.timeStamp;
       isAntiMessage= evt.isAntiMessage;
}
```

Bild 278: Copy-Konstruktor für DataEvent

[77] Für die Verwaltung der beim kontinuierlichen Simulator eingehenden Events wird die bei Hewlett-Packard entwickelte C++-Bibliothek Standard Template Library (STL) verwendet, die vom ANSI/ ISO-Komitee als Teil des C++-Standards akzeptiert worden ist. Ihr Schwerpunkt liegt auf Datenstrukturen für Behälter (englisch Container) und Algorithmen, die damit arbeiten, wobei sie den Template-Mechanismus zur Parametrisierung von Komponenten nutzt. Die einheitliche Art der Schnittstellen erlaubt eine flexible Zusammenarbeit der Komponenten und auch die Konstruktion neuer Komponenten im STL-Stil.

Bei Verwendung eines STL-Containers, in dem die Elemente einer Ordnung folgend abgelegt sind, muss diese Ordnung durch einen Vergleichsoperator definiert sein (Bild 279).

```
friend bool operator<(const DataEvent& evt1, const DataEvent& evt2) {
      return (evt1.timeStamp < evt2.timeStamp);
}
```

Bild 279: Vergleichsoperator für DataEvent

Negativnachrichten: Eine empfangene Negativnachricht wird durch Setzen der Membervariable `isAntiMessage` von `DataEvent` auf `true` kenntlich gemacht. Um für die gesendeten Ereignisse bei Bedarf Negativnachrichten generieren zu können, werden alle gesendeten Ereignisse dem Generierungszeitpunkt nach geordnet in einem Binärbaum gespeichert, durch den eine schnelle Suche möglich wird.

Zustandsspeicherung: Für die Zustandsspeicherung muss auf die modell- und simulationsinternen Variablen zugegriffen werden können, so z. B. auf die Ausgangswerte der Modellblöcke und die Simulationszeit. Drei Variablen repräsentieren in RTW die Blockzustände: `rtU`, `rtB` und `rtY`. Anzahl und Namen der Komponenten dieser Datenstrukturen sind allerdings von Modell zu Modell verschieden und ein komponentenweiser Zugriff daher für ein modellunabhängige Systematik ungeeignet. Aus diesem Grund wurde eine Java-Anwendung implementiert, die die modellspezifisch generierte Datei `model.h` analysiert und die Dateien `rtw_data_arrays.cpp` und `rtw_data_arrays.h` erzeugt, in denen drei Arrays `rtU_Array`, `rtB_Array` und `rtY_Array` realisiert sind, die als Elemente Zeiger auf die Komponenten der oben genannten Variablen `rtU`, `rtB` und `rtY` enthalten (Bild 280). Nun kann mit einer Index-Variablen auf die Komponenten zugegriffen werden, was eine generalisierte Handhabung ermöglicht.

```
void initRTW_Arrays() {

      // Block signals
      rtB_Array[0] = &(rtB.Gain);
      rtB_Array[1] = &(rtB.Gain1);
      rtB_Array[2] = &(rtB.Integrator);

      // External inputs
      rtU_Array[0] = &(rtU.In1);
      rtU_Array[1] = &(rtU.In2);

      // External outputs
      rtY_Array[0] = &(rtY.Out1);
      rtY_Array[1] = &(rtY.Out2);
}
```

Bild 280: Zugriff auf interne Modellzustandswerte über Arrays

So lassen sich die modellinternen Zustandswerte leicht speichern, indem über die Array-Elemente iteriert und der referenzierte Wert in ein zweites Array (`rtU_Copy`, `rtB_Copy` bzw. `rtY_Copy`) kopiert wird. Zur Zustandsspeicherung wird weiterhin der Zeitstempel `timeStamp` der aktuellen lokalen Simulationszeit benötigt. Um zwischen einem vollständig und inkrementell gespeicherten Zustand unterscheiden zu können, wird eine Kennungsvariable `typ` eingeführt.

```
static int          nrHosts;
static long         prevTotalState;
long                timeStamp;
short               typ;
long                *lastSendTimes;
real_T*             rtB_Copy;
real_T*             rtU_Copy;
real_T*             rtY_Copy;
```

Bild 281: Membervariablen der Klasse State

Um einen inkrementell gespeicherten Zustand wiederherstellen zu können, muss der Zeitpunkt der letzten vollständigen Zustandsspeicherung bekannt sein, dieser wird `prevTotalState` in gespeichert. Gekapselt werden diese Daten in der Klasse `State` (Bild 281).

Rücksetzung: Beim Eintreffen einer Nachricht mit einem Zeitstempel, der kleiner als die aktuelle Simulationszeit ist, wird eine Rücksetzung (Rollback) ausgelöst. Um den Ablauf einer Rücksetzung nachvollziehen zu können, muss zunächst die Verwaltung der eingegangenen Nachrichten erklärt werden. Bild 282 zeigt exemplarisch eine Momentanaufnahme der Organisationsstruktur der eingegangenen Ereignisse einer Simulation. Dabei empfangen zwei Empfangsthreads von zwei Sendern Daten: Sender 1 sendet Ereignisse an zwei Eingangsports des RTW-Modells (`DefEvent` 1 und 2), von Sender 2 wird ein Eingangsport mit Daten versorgt (`DefEvent` 3). Die eigentlichen Ereignisse werden nach Parsen der eingegangenen Nachrichten als Objekte vom Typ `DataEvent` in den `DefEvent`-Objekten zugeordneten Ereignisliste gespeichert, in Bild 282 jeweils rechts vom zugehörigen `DefEvent` dargestellt. Bei jedem Simulationsschritt wird überprüft, ob diese Listen Ereignisse enthalten, deren Zeitstempel gleich der aktuellen Simulationszeit ist. Ist dies der Fall, wird der Wert des Ereignisses an den entsprechenden Eingangsport gelegt und das Ereignis in eine andere Liste verschoben, in der die abgearbeiteten Ereignisse eines jeweiligen `DefEvents` gespeichert werden. In Bild 282 sind diese Listen rechts von den `DefEvents` dargestellt, wobei sich die Simulation vor Ausführung des Simulationsschritts mit Simulationszeit 7 befindet.

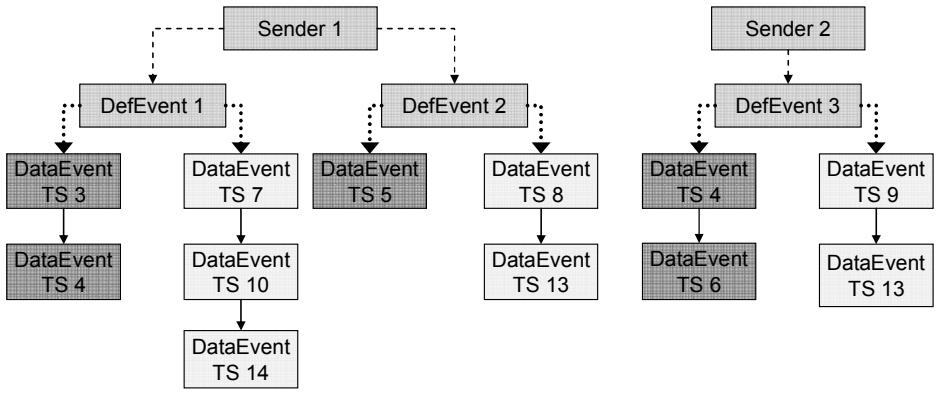


Bild 282: Organisation der empfangenen Ereignisse

Vor einer Rücksetzung tritt nun folgende Situation ein: ein Empfangsthread empfängt eine Nachricht und fügt die geparsten Ereignisse nach Zeitstempel sortiert in die Listen der entsprechenden `DefEvents` ein. Ist ein Ereignis darunter, dessen Zeitstempel kleiner als die aktuelle Simulationszeit ist, wird es aufgrund der linearen Ordnung innerhalb der Liste an den Anfang der Liste gesetzt. Beim nächsten Simulationsschritt wird der Zeitstempel dieses Ereignisses überprüft und signalisiert, dass eine Rücksetzung zu der Simulationszeit ausgeführt werden muss, die mit dem Zeitstempel des Ereignisses übereinstimmt.

Die eigentliche Rücksetzung läuft wie folgt ab:

- Zunächst wird die Simulationszeit zurückgesetzt. In RTW sind dafür zwei Schritte notwendig: zum einen wird mit der Funktion `rtmSetT` der Zeitwert des aktuellen Simulationsschritts manipuliert, zum anderen die Variable `clockTick`, aus der die Simulationszeit für den nächsten Simulationsschritt berechnet wird (Bild 283).

```
for (i=0; i<NUMST; i++) {
      td_statesaving.clockTick[i] = TimeStamp;
}
rtmSetT(S, TimeStamp);
```

Bild 283: Rücksetzen der Simulationszeit beim Rollback

- Anschließend wird der Zustand in der Liste der gespeicherten Zustände gesucht, auf den zurückgesprungen werden soll. Eine Rücksetzung zurück zu einem Zeitpunkt *t* bedeutet, dass der Zustand nach Ausführen von Zeitschritt *t-1* wieder hergestellt werden muss. Demnach wird Zustand mit Zeitstempel *t-1* wiederhergestellt. Hierfür wird ein Vergleichsobjekt mit gleichem Zeitstempel erzeugt und damit in der Liste gesucht (Bild 284).

```
findState = new State(TimeStamp -1);
stateIter = lower_bound(savedStates.begin(),
                                savedStates.end(), *findState);
actualState = (State *) &(*stateIter);
```

Bild 284: Suchen des wiederherzustellenden Zustands

- Nun werden die Zustandswerte des Modells mit denen aus dem `State`-Objekt überschrieben. Bild 285 zeigt dies exemplarisch anhand der Werte der Eingangsports.

```
rt_Copy = actualState->get_rtU_Copy();
for (i=0; i < rtU_Size; i++) {
      *(rtU_Array[i]) = rt_Copy[i];
      dataRcv[i] = rt_Copy[i];
}
```

Bild 285: Überschreiben der Modelleingangswerte

- Als nächstes werden die Listen der empfangenen Ereignisse zurückgesetzt. Die Ereignisse sind, wie weiter oben erläutert, nicht im gespeicherten Zustand enthalten, sondern werden für jedes `DefEvent` in zwei Listen verwaltet: eine für die Ereignisse mit Zeitstempel in der Zukunft, eine zweite für die bereits abgearbeiteten Ereignisse. Bei der Rücksetzung werden die Ereignisse mit Zeitstempel größer oder gleich der Rücksetzungszeit von der Liste der abgearbeiteten Ereignisse in die Liste der zukünftigen Ereignisse verschoben (Bild 286).

```
while (procEvtIter != procEvtList->end()) {
      dataEvt = (DataEvent *) &(*procEvtIter);
      success = dataEvtList->insert(*dataEvt);
      procEvtList->erase(procEvtIter);
      procEvtIter = lower_bound(procEvtList->begin(),
                          procEvtList->end(), *findDataEvt);
}
```

Bild 286: Verschieben der Ereignisse

- Zuletzt werden für die Ereignisse, die nach der Zeit erzeugt wurden, zu der bei der Rücksetzung zurückgesprungen wird, Negativnachrichten gesendet. An dieser Stelle macht sich die Tatsache positiv bemerkbar, das RTW zeitgesteuert simuliert und der Zeitstempel eines Ereignisses immer der Simulationszeit entspricht, an der das Ereignis erzeugt wurde. Somit lässt sich anhand des Zeitstempels des Ereignisses sein Erzeugungszeitpunkt erkennen, und bei der Rücksetzung wird für alle gesendeten Ereignisse mit Zeitstempel größer oder gleich der Rücksetzungszeit eine Negativnachricht erzeugt und gesendet. Bild 287 zeigt, wie durch alle Listen der gesendeten Ereignisse iteriert wird, um für die relevanten Ereignisse Negativnachrichten zu senden.

```
// Iterate all receiving hosts
for (i = 0; i < sendList->size(); i++) {
       receiver = &((*sendList)[i]);
       lastSend = receiver->getLastSendTime();
       defEvtList = receiver->getDefEvtList();

       // Iterate all DefEvents
       for (j = 0; j < defEvtList->size(); j++) {
              defEvt = &((*defEvtList)[j]);
              dataEvtList = defEvt->getDataEvtList();
              findDataEvt = new DataEvent(0, rollbackTime);
              dataEvtIter = lower_bound(dataEvtList->begin(),
                                        dataEvtList->end(), *findDataEvt);

              // Iterate DataEvents with TS >= Rollback-Time
              while (dataEvtIter != dataEvtList->end()) {
                     // Create Anti-Message
                     ...
              }
       }
}
```

Bild 287: Iteration durch Ereignisliste zur Erzeugung von Negativnachrichten

D.13 Implementierung der SES und der synchronen Hypothese

Das realisierte Framework zur optimistischen Co-Simulation hybrider Systeme verknüpft zur Synchronisation und Kommunikation zwei grundlegende Konzepte: die in 3.6.3 beschriebene SES und die im Rahmen dieser Arbeit entwickelte synchrone Hypothese (siehe 4.6.6). Im Folgenden werden implementierungsspezifische Aspekte bezüglich der Umsetzung dieser beiden Mechanismen dargestellt.

Aktivierung gemäß SES: Jeder CS wird von einem DES nach den Regeln der SES aktiviert und berechnet dann das ihm vorgegebene Simulationsintervall.

Senden des Aktivierungsereignisses in JStateSim: Nach Ausführung eines Simulationsschritts in JStateSim wird in der Methode `TimeWarpTCP.runCycle()` über `JSTimeWarpTCP.updateSimulator()` die Methode `SimManagerTCP.updateActEvts()` aufgerufen. Hier wird über die Liste der zu aktivierenden RTW iteriert und für jeden Eintrag `ActivationEvent.sendActivation()` ausgeführt. Dort wird zunächst überprüft, ob ein Rücksetz-Ereignis gesendet werden muss (Bild 288).

```
for (int i=0; i<hypos.size(); i++) {
  hypo = (Hypothese) hypos.elementAt(i);
  sendReset |= hypo.isAddedNewCorrectedValue();
}
if (sendReset) {
  comEvt = new ComEvent(defEvt.getName(), DefEvent.RTW_ACTION,
                                  ComEvent.JSS_ROLLBACK, 1.0);
  simManager.sendOutputEvent(comEvt, defEvt, true);
}
```

Bild 288: Erzeugen des Rücksetz-Ereignisses

Dies ist notwendig, wenn der RTW zuvor eine Schwellwertüberschreitung festgestellt und einen Korrekturwert gesendet hat. Er unterbricht daraufhin die Simulation vorübergehend und ignoriert die eintreffenden Aktivierungsereignisse solange, bis JStateSim das Rücksetz-Ereignis gesendet und damit signalisiert hat, dass er den Korrekturwert empfangen und verarbeitet hat und nun mit aktualisiertem Wert weiter simuliert. Das Rücksetzereignis wird durch die Kennung `JSS_ROLLBACK` identifiziert.

Als nächstes wird überprüft, ob ein Aktivierungsereignis gesendet werden muss (Bild 289). Dies ist notwendig, wenn sich die Simulationszeit bei JStateSim geändert hat. Bei einer Rücksetzung wird zwar nicht direkt ein Aktivierungsereignis gesendet, allerdings muss die Variable, die den Zeitpunkt der letzten Aktivierung speichert, auf die Rücksetzungszeit gesetzt werden, um so eine Aktivierung bei der nächsten Änderung der Simulationszeit durchführen zu können.

```
if (didRollback) {
  didRollback = false; // Reset flag
  firstActivationTime = rolledBackTo + 1;
  lastActivationTime = firstActivationTime;
}
else {
  if (simTime > lastActivationTime) {
    sendAct = true;
    lastActivationTime = simTime;
  }
}
```

Bild 289: Bedingungen für Erzeugung des Aktivierungsereignisses

Auswerten des Aktivierungsereignisses in RTW: Die Aktivierungsereignisse werden als Aktionsereignisse mit Ereignistyp-ID `RTW_ACTION` von JStateSim an RTW gesendet und dort in einer Liste gesammelt. Hat der RTW sein aktuelles Simulationsintervall vollständig berechnet, oder ist eine Schwellwertüberschreitung aufgetreten, hält er die Simulation an und wartet auf eine neue Aktivierung. Die entsprechende Funktionalität ist in der Funktion `manageActivationEvents()` in `dataRcvThread.cpp` implementiert (Bild 290).

```
if (defEvt->getDataEvtList()->size() > 0) {
  if (opCode == GET_NEXT_STOPTIME) {
    dataEvtIter = defEvt->getDataEvtList()->begin();
    dataEvt = (DataEvent *) &(*dataEvtIter); // get first DataEvent
    returnCode = dataEvt->getTimeStamp();
    defEvt->getDataEvtList()->erase(dataEvtIter);
  }
  else if (opCode == DEL_ACT_EVENTS) {
    dataEvtIter = defEvt->getDataEvtList()->begin();
    dataEvt = (DataEvent *) &(*dataEvtIter); // get first DataEvent
    if(dataEvt->getTimeStamp() != JSS_ROLLBACK) {
      defEvt->getDataEvtList()->erase(
             defEvt->getDataEvtList()->begin(),
             defEvt->getDataEvtList()->end());
    }
    returnCode = NO_NEW_STOPTIME;
  }
}
else {
  returnCode = NO_NEW_STOPTIME;
}
```

Bild 290: Verwaltung der Aktivierungsereignisse bei RTW

Die Funktion gibt bei vorhandenem Aktivierungsereignis die Endzeit des neuen Simulationsintervalls zurück, andernfalls die Kennung `NO_NEW_STOPTIME`.

Synchrone Hypothese: Hypothesefunktion bei RTW: Auf RTW-Seite werden die Ergebnisse eines Simulationsschritts mit den Werten der entsprechenden Hypothesefunktion verglichen und bei Schwellwertüberschreitung ein Korrekturwert an JStateSim gesendet. Dabei wird zunächst die dem jeweiligen Ausgangsport zugeordnete Hypothesefunktion bestimmt, über die dann der Hypothesewert mit `calcHypoValue()` berechnet wird. Bild 291 zeigt die Implementierung dieser Funktion für die Sinus-Hypothese.

```
double HypoFcnSinus::calcHypoValue(long time) {
       double value;
       value = amp*sin(time*frq + phase) + y_offset;
       return value;
}
```

Bild 291: Funktion calcHypoValue() der Sinus-Hypothesefunktion bei RTW

Bild 292 enthält den zur Verarbeitung der Hypothese relevanten Ausschnitt der Funktion `dataSend()` von `dataSend.cpp`.

```
hypoFcn = defEvt->getHypoFcn();
hypoID = hypoFcn->getID();
if (hypoID == LINEAR_ID) {
  hypoFcnLinear = (HypoFcnLinear *) hypoFcn;
  hypoValue = hypoFcnLinear->calcHypoValue(timeStamp);
}
else if (hypoID == LVV_ID) {
  hypoFcnLVV = (HypoFcnLVV *) hypoFcn;
  hypoValue = hypoFcnLVV->calcHypoValue(timeStamp);
}
else if (hypoID == SINUS_ID) {
  hypoFcnSinus = (HypoFcnSinus *) hypoFcn;
  hypoValue = hypoFcnSinus->calcHypoValue(timeStamp);
}
else if (hypoID == FROM_FILE_ID) {
  hypoFcnFromFile = (HypoFcnFromFile *) hypoFcn;
  hypoValue = hypoFcnFromFile->calcHypoValue(timeStamp);
}
hypoTrue = hypoFcn->validateHypothese(dataOut[k], hypoValue);
```

Bild 292: Aufruf der Hypothesefunktion bei RTW

Die Schwellwertüberprüfung ist in `validateHypothese()` von `HypoFcn.cpp` realisiert (Bild 293).

```
if (upRel) {
  upTreshold = hyp + hyp*upTol;
}
else {
  upTreshold = hyp + upTol;
}
if (lowRel) {
  lowTreshold = hyp - hyp*lowTol;
}
else {
  lowTreshold = hyp - lowTol;
}
if ( (res > upTreshold) || (res < lowTreshold) ) {
  return false;
}
else {
  return true;
}
```

Bild 293: Schwellwertprüfung bei RTW

Hypothesefunktion in JStateSim: Die synchrone Hypothese wird durch eine virtuelle Datenverbindung zwischen RTW und JStateSim etabliert. Daher werden die von der Hypothesefunktion generierten Ereignisse als externe Ereignisse interpretiert, da sie stellvertretend für die Signalwerte des kontinuierlichen Simulators verwendet werden. Bei JStateSim werden mit `ComProtokoll.getEvent()` vor jedem Simulationsschritt die Modelleingänge auf neu anliegende Werte überprüft (Bild 294). Dabei wird die Methode `addHypotheseValues()` der Klasse `SimManagerTCP` ausgeführt, die für die Verwaltung der Hypothesefunktionen zuständig ist.

```
if (simManager.getArg().hasPulsGen) {
  hypoTime = tw.GetSimTime() + 1;
  simManager.addHypotheseValues(hypoTime);
}
else { // external Trigger
  nextEvent = getComEvent();
  if ( (nextEvent != null) &&
       (nextEvent.getType() == DefEvent.JSS_ACTION) ) {
    hypoTime = nextEvent.getTimestamp();
    simManager.addHypotheseValues(hypoTime);
  }
}
```

Bild 294: Belegen der Modelleingänge mit Hypothesewerten

Die Hypothesefunktionen selbst sind Unterklassen von `Hypothese`, die grundlegende Methoden implementiert, die von allen Hypothesefunktionen verwendet werden.

Mit `getComEvent()` wird ein neuer Hypothesewert erzeugt, dabei wird die von jeder Unterklasse überschriebene Methode `calcValue()` aufgerufen, wo die charakteristische Funktionsweise der jeweiligen Hypothesefunktion implementiert ist. Bild 295 zeigt die entsprechende Implementierung für die Sinus-Hypothese.

```
protected double calcValue(long time) {
  return ampl * Math.sin(frq*time + phase) + y_offset;
}
```

Bild 295: Methode calcValue() der Sinus-Hypothesefunktion bei JStateSim

D.14 Performance-Messungen mit EXITE

Um vergleichbare Ergebnisse zu erreichen wurden alle Messungen in einem Netzwerk mit einer Übertragungsrate von 100 MBit/s durchgeführt. Um die Funktionalität und den Einfluss der Verteilung auf die Simulationsgeschwindigkeit zu messen, wurden vier Testmodelle entworfen, die in unterschiedlicher Konfiguration ausgeführt wurden. Die Messungen wurden alle für 10000 Simulationsschritte mit fester Schrittweite 1 ausgeführt.

Beim ersten Testmodell (Bild 296 oben) wird in einem Stateflowchart (Bild 296 unten) bei jedem Simulationsschritt die Ausgabevariable `out_data1` verändert, wodurch sich der Ausgangswert des Stateflow-Blocks bei jedem Simulationsschritt ändert.

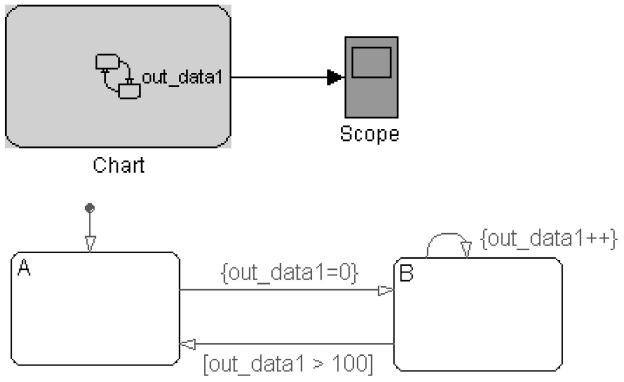


Bild 296: EXITE-Modell 1

Bei der verteilten Version dieses Modells (Bild 297) muss daher bei jedem Simulationsschritt der neue Ausgangswert übertragen werden. Hier wird im ersten Teilmodell der Stateflow-Block ausgeführt und dessen Ausgangswerte über den EXITE-Master-Block an das zweite Teilmodell übertra-

gen. Dort wird das empfangene Signal über den Scope-Block graphisch dargestellt und zurück an das erste Teilmodell gesendet.

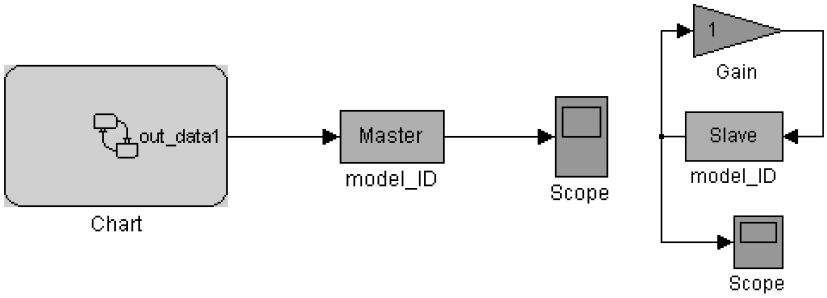


Bild 297: Verteiltes EXITE-Modell 1

Die Messergebnisse in Bild 298 zeigen, dass die verteilte Ausführung in jedem Fall langsamer ist als die Simulation des Modells ohne Verteilung. Dabei benötigt die Ausführung auf zwei Rechnern noch etwas mehr Zeit als wenn die beiden Teilmodelle auf zwei MATLAB-Instanzen desselben PCs simuliert werden. Auffällig ist, dass der Full-Duplex-Modus am langsamsten abläuft.

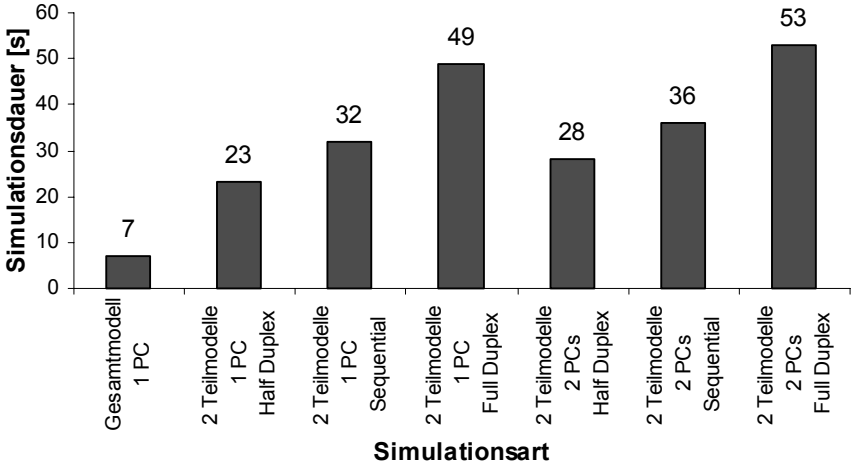


Bild 298: Messergebnisse von EXITE-Modell 1

Das zweite Testmodell unterscheidet sich vom ersten durch die Rate, mit der sich der Wert am Ausgang des Stateflow-Blocks ändert (Bild 299). Hier wird eine interne Variable `local_data` pro Simulationsschritt verändert, die Ausgabevariable `out_data1` nur bei jedem hundertsten Durchlauf. Dadurch reduziert sich im Vergleich zu Modell 1 die Häufigkeit, mit der der Ausgangswert übertragen werden muss.

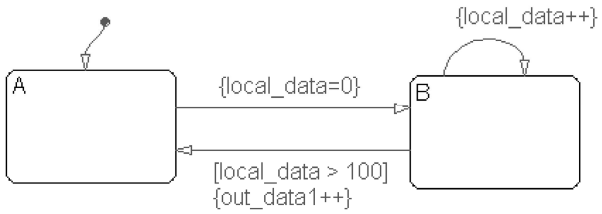


Bild 299: Stateflowchart von EXITE-Modell 2

Ein Geschwindigkeitszuwachs im Vergleich zu Modell 1 lässt sich bei der Ausführung auf zwei Rechnern feststellten (Bild 300). Diese Simulationsvariante ist sogar schneller, als die Simulation der beiden Teilmodelle auf zwei MATLAB-Instanzen desselben PCs. Das Testmodell 3 (Bild 301) enthält zwei Stateflow-Blöcke mit vergleichbarer Funktionalität wie in Modell 1. In der verteilten Variante wurden diese Stateflowcharts in zwei in Reihe geschalteten Teilmodellen ausgeführt und das Endergebnis an ein drittes Teilmodell weitergeleitet, das auch die Stimuli erzeugt.

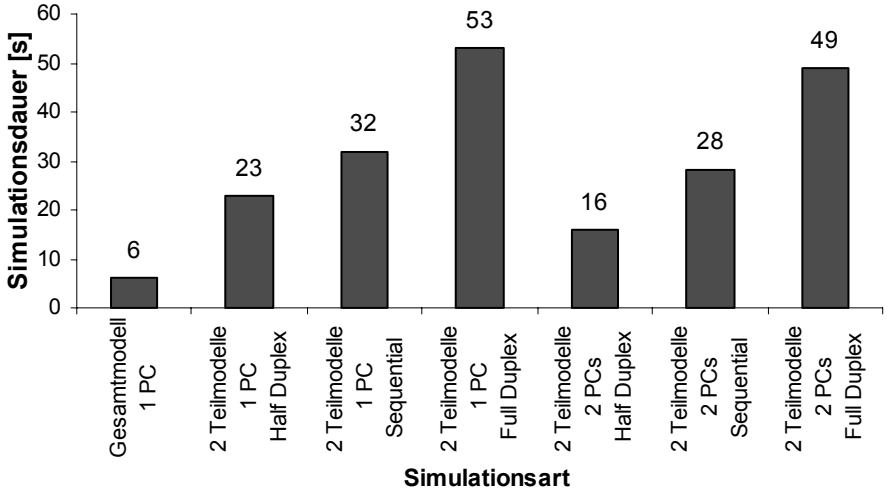


Bild 300: Messergebnisse von EXITE-Modell 2

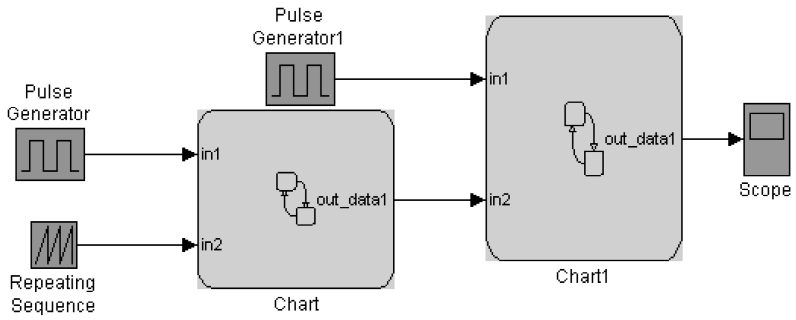


Bild 301: EXITE-Modell 3

Wie aus Bild 302 und Bild 303 zu entnehmen ist, wurden die kürzesten Simulationszeiten bei der Verteilung auf zwei Rechner erreicht, wobei auf einem Rechner zwei Teilmodelle in zwei MATLAB-Instanzen ausgeführt wurden.

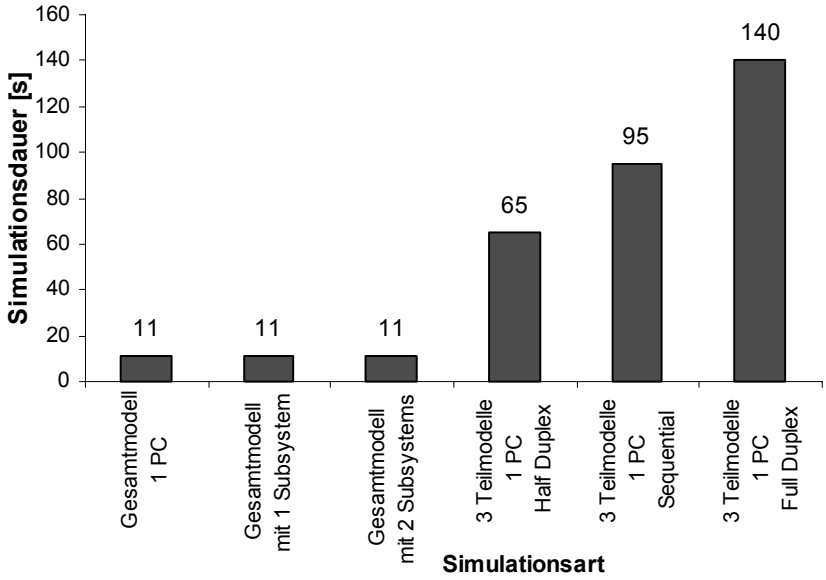


Bild 302: Messergebnisse von EXITE-Modell 3 (Teil 1)

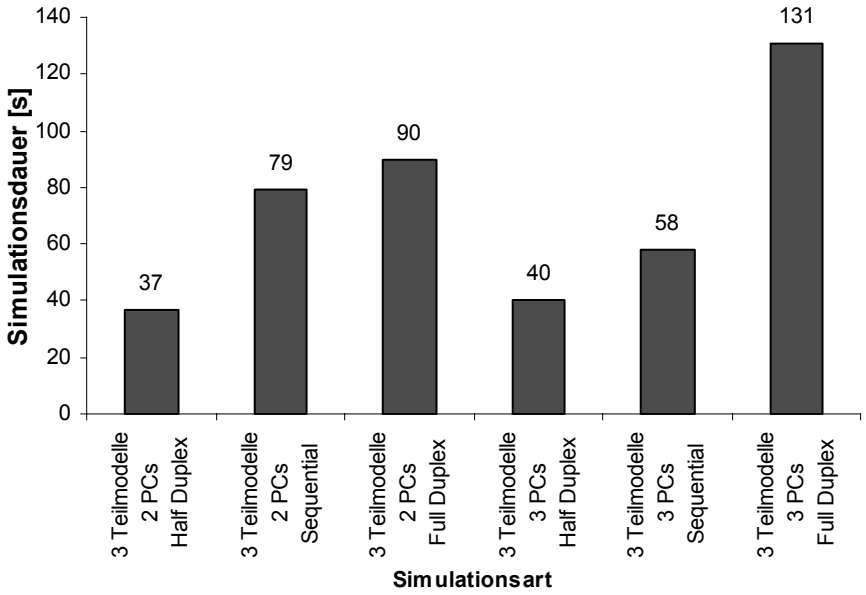


Bild 303: Messergebnisse von EXITE-Modell 3 (Teil 2)

Das vierte Testmodell enthält wiederum zwei Stateflowcharts, die allerdings parallel ausgeführt werden können, da ihre Ein- und Ausgangswerte unabhängig voneinander sind (Bild 304).

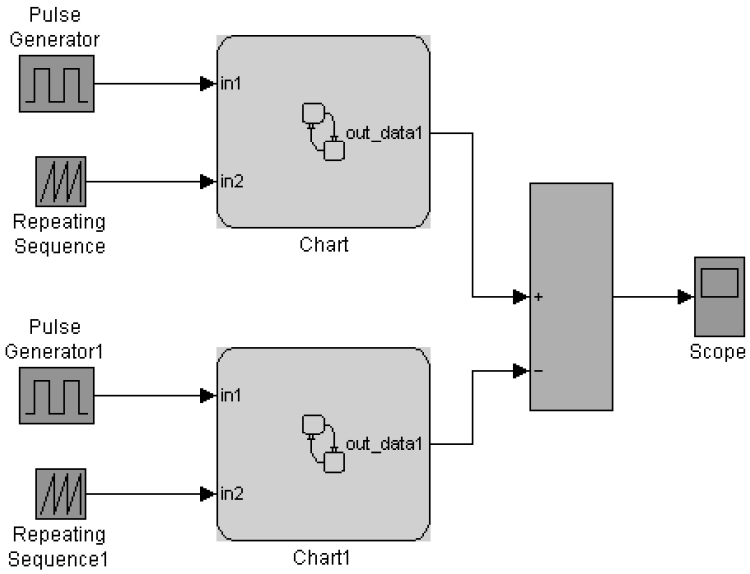


Bild 304: EXITE-Modell 4

Die parallele Ausführung macht sich in der Ausführungsgeschwindigkeit deutlich bemerkbar (Bild 305 und Bild 306). So lässt sich beispielsweise im Halbduplex-Modus bei Simulation auf zwei Rechnern die gleiche Simulationsdauer wie von Modell 1 in vergleichbarer Konfiguration erreichen (28 s für 10000 Simulationsschritte). Dies bedeutet, dass das zusätzlich simulierte Teilmodell aufgrund der parallelen Ausführung die Gesamtsimulation nicht verlangsamt.

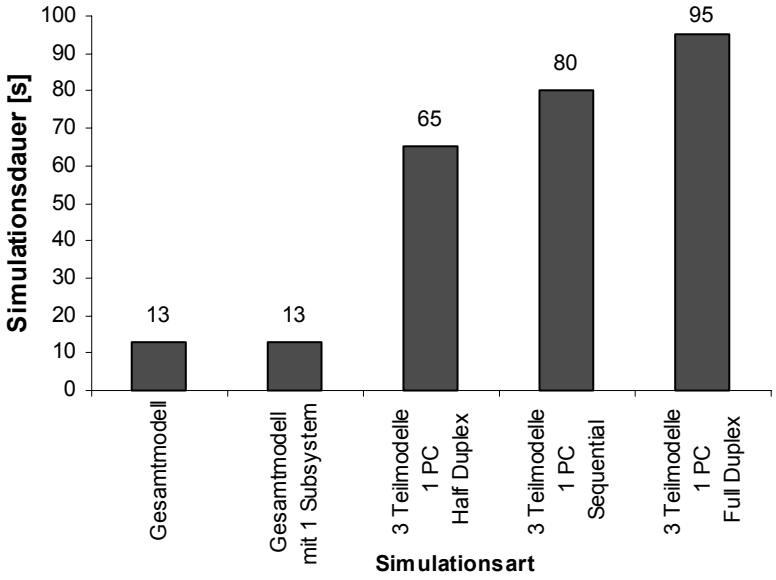


Bild 305: Messergebnisse von EXITE-Modell 4 (Teil 1)

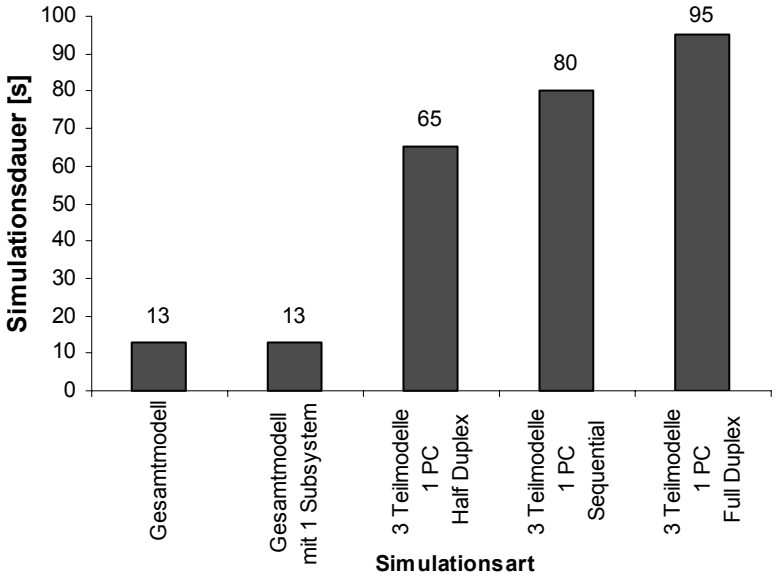


Bild 306: Messergebnisse von EXITE-Modell 4 (Teil 2)

D.15 Komplexität

In der folgenden Tabelle ist die Anzahl der Code-Zeilen der entwickelten SW-Komponenten aufgelistet[78].

Software-Komponente	Zählweise	Code-Zeilen
Konverter zwischen ASCII- und mdl-Format		190
JStateSim für RP.2002	ohne Parser	372
Konverter zwischen ISCAS- und mdl-Format für Logikschaltungen		1040
JLogSim		3013
JSimControl	Die bei anderen Klassen wiederverwendeten Zeilen werden nur bei JSimControl gezählt: • `common java package`: 2938 Zeilen; enthalten auch bei JLogSim, JStateSim und JStateSim für TCP/IP und RTW • `RMISimServerImpl`, `SendEvent` und `BasicEvent`: 205 Zeilen; enthalten auch bei JStateSim für RP.2002	5806
JStateSim	ohne Parser	6209
JStateSim-Erweiterung zum Compilieren von Java-Klassen		1692
JStateSim-Erweiterung für TCP/IP und RTW	Klassen mit Endung „TCP“	5881
JStateGen	einschließlich mdl-Generator	640
JStimGen		317
JVHDLGen	ohne Parser	2558
Partitionierung Stateflow Charts		3324
HIL-Co-Simulation zwischen Simulink und einem µC-Knoten	ohne Erweiterungen für mehrere µC-Knoten und ohne FPGA-Integration	565
Parser für mdl-Format, allgemeingültige Version		11819
Parser für mdl-Format, Version für spezielle Modelle, als Grundlage für allgemeingültige Version		1078
		44504

[78] ohne Kommentar- und Leerzeilen

E Stichwortverzeichnis

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

F Lebenslauf

Name:	Rico Dreier
Geboren:	14. März 1972 in Sinsheim
Schulausbildung:	1991 Erwerb der allgemeinen Hochschulreife an der Friedrich-Hecker-Schule (berufliches Gymnasium, technische Richtung) in Sinsheim
Studium:	WS 91/92 - WS 97/98 Studium der Elektrotechnik an der Universität Karlsruhe, Schwerpunkt Technik der Informationsverarbeitung
Beruflicher Werdegang:	Januar 1998 - Juni 2005 wissenschaftlicher Mitarbeiter am FZI Forschungszentrum Informatik an der Universität Karlsruhe, Abteilung Elektronische Systeme und Mikrosysteme Seit Juli 2005 Mitarbeiter bei der MB-technology GmbH in Sindelfingen