IAS-Forschungsberichte

Forschungsbericht
Institut für Automatisierungs- und Softwaretechnik

Hrsg.: Prof. Dr.-Ing. Dr. h. c. P. Göhner

Thomas Ringler

Entwicklung und Analyse zeitgesteuerter Systeme

Band 2/2002

Universität Stuttgart

Entwicklung und Analyse zeitgesteuerter Systeme

Von der Fakultät Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde eines
Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

Vorgelegt von
Thomas Karl Ringler
aus Abtsgmünd

Hauptberichter:	Prof. Dr.-Ing. Dr. h. c. Peter Göhner
Mitberichter:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Tag der Einreichung:	19.12.2001
Tag der mündlichen Prüfung:	25.07.2002

Institut für Automatisierungs- und Softwaretechnik
der Universität Stuttgart

2002

IAS-Forschungsberichte

Band 2/2002

Thomas Karl Ringler

Entwicklung und Analyse zeitgesteuerter Systeme

D 93 (Diss. Universität Stuttgart)

Shaker Verlag
Aachen 2002

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Ringler, Thomas Karl:
Entwicklung und Analyse zeitgesteuerter Systeme / Thomas Karl Ringler.
Aachen : Shaker, 2002
(IAS-Forschungsberichte ; Bd. 2002,2)
Zugl.: Stuttgart, Univ., Diss., 2002
ISBN 3-8322-0782-1

ISBN 3-8322-0782-1
ISSN 1610-4781

Shaker Verlag GmbH • Postfach 101818 • 52018 Aachen
Telefon: 02407 / 95 96 - 0 • Telefax: 02407 / 95 96 - 9
Internet: www.shaker.de • eMail: info@shaker.de

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Automatisierungs- und Softwaretechnik (IAS) der Universität Stuttgart.

Mein besonderer Dank gilt

Herrn Prof. Dr.-Ing. Dr. h. c. P. Göhner für die Betreuung dieser Arbeit sowie für den Freiraum zur Vollendung dieser Arbeit,

Herrn Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder für die Übernahme des 1. Mitberichts,

allen Kolleginnen und Kollegen am Institut für die gute Zusammenarbeit und hilfsbereite Unterstützung,

den Studenten, die im Rahmen von Studien- und Diplomarbeiten zum Gelingen dieser Arbeit beigetragen haben, damit die vorgestellten Konzepte nicht nur Ideen blieben, sondern auch verwirklicht werden konnten,

meinen Eltern für die langjährige Unterstützung während meiner gesamten Ausbildungszeit

und Ulla für ihr Verständnis, ihre Geduld und ihren Ansporn während der Entstehung dieser Arbeit.

Stuttgart, im Juli 2002

Thomas Ringler

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ABS	**A**nti-**B**lockier**s**ystem
ASCET-SD	**A**dvanced **S**imulation and **C**ontrol **E**ngineering **T**ool - **S**oftware **D**evelopment
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
ASR	**A**nti-**S**chlupf**r**egelung
BA	**B**rems**a**ssistent
BCET	**B**est-**C**ase **E**xecution **T**ime
COM	**C**omponent **O**bject **M**odel
DMA	**D**irect **M**emory **A**ccess
DV-Teil	**D**aten**v**erarbeitungs**teil**
ESP	**E**lectronic **S**tability **P**rogram
FTDMA	**F**lexible **T**ime **D**ivision **M**ultiple **A**ccess
GKB	**G**lobale **K**ommunikations**b**eziehung
GNU	**G**NU's **N**ot **U**nix!
IDL	**I**nformation **D**escription **L**anguage
IPE	**I**mplicit **P**ath **E**numeration
ISR	**I**nterrupt **S**ervice **R**outine
LKB	**L**okale **K**ommunikations**b**eziehung
MARS	**M**aintainable **R**eal-**T**ime **S**ystem
MATLAB	**Ma**trix **Lab**oratory
MFC	**M**icrosoft **F**oundation **C**lasses
MINT	**M**inimum **I**nter-**A**rrival **T**ime
OC	**O**bject **C**ode
OIL	**O**SEK/VDX **I**mplementation **L**anguage
OSEK/VDX	**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug / **V**ehicle **D**istributed **E**xecutive

RAM	**R**andom **A**ccess **M**emory
ROM	**R**ead-**O**nly **M**emory
SAE	**S**ociety of **A**utomotive **E**ngineering
SETTA	**S**ystem **E**ngineering Environment for **T**ime-**T**riggered **A**rchitectures
SSC	**S**orted **C**ircuit **C**ode
TDMA	**T**ime **D**ivision **M**ultiple **A**ccess
TLC	**T**arget **L**anguage **C**ompiler
TTA	**T**ime-**T**riggered **A**rchitecture
TTP/C	**T**ime-**T**riggered **P**rotocol / SAE Class **C**
ViETTA	**Vi**sual **E**nvironment for **T**ime-**T**riggered **A**rchitectures
ViPER	**Vi**sual **P**rogramming **E**nvironment for **E**mbedded **R**eal-Time Systems
WCET	**W**orst-**C**ase **E**xecution **T**ime

Begriffsverzeichnis

Aktor: Einheit zur Umsetzung von Stellinformation tragenden Signalen geringer Leistung in leistungsbehaftete Signale einer zur Prozessbeeinflussung notwendigen Energieform

Berechnete minimale Ausführungszeit ($BCET_C$): Rechnerisch ermittelte untere Schranke für die minimale Ausführungszeit eines gegebenen Codestücks

Blockschaltbild: Hilfsmittel zur Beschreibung von kontinuierlichen (bzw. dynamischen) Systemen

Deadline: Spätester Zeitpunkt, zu dem eine Aktion (z. B. Task oder Kommunikationsbeziehung) abgearbeitet sein muss

Dispatcher: Teil eines Betriebssystems, der Tasks zur Ausführung bringt

Echtzeitsoftware: Software, deren Korrektheit nicht nur von logischen Ergebnissen abhängt, sondern auch von dem Zeitpunkt, zu dem die Ergebnisse berechnet werden

Fail-operational Systeme: Systeme, die selbst im Falle eines Fehlers, der sonst schwerwiegende Folgen hätte, die vorgeschriebene Funktionalität in jedem Fall erfüllen müssen

Fehlertoleranz: Fähigkeit eines Systems, auch mit einer begrenzten Anzahl fehlerhafter Subsysteme seine Spezifikation zu erfüllen

Globale Kommunikationsbeziehung (GKB): Zeitlich beschränkte Kette von lokalen Kommunikationsbeziehungen und netzwerkweiten Kommunikationsaktivitäten, die von einem verteilten System bearbeitet werden

Grundblock (engl. Basic Block): Die maximale Anzahl nacheinander folgender Anweisungen im Kontrollflussgraf ohne Verzweigung

Knoten: Mikrorechner mit eigener Hardware (Prozessor, Speicher, Kommunikationsschnittstelle, Schnittstelle zum technischen Prozess) und Software (Anwendungssoftware, Systemsoftware), der eine Menge von definierten Funktionen in einem verteilten Rechnersystem wahrnimmt

Kontrollflussgraf: Gerichteter Graf zur Beschreibung des möglichen Programmflusses in einem Programm bzw. Codestück

Lokale Kommunikationsbeziehung (LKB): Zeitlich beschränkte Kette von Tasks und knotenlokaler Kommunikation über Signale, die von einem Knoten bearbeitet werden

Pfad: Eine Folge von Knoten und Kanten im Kontrollflussgraf, die mit dem Startknoten beginnt und mit einem Endknoten endet

Redundanz: Vorhandensein von mehr als für die Ausführung der vorgesehenen Aufgaben an sich notwendigen Mittel

Sensor: Einheit zur Umsetzung von physikalischen oder chemischen Größen in eine elektronische Größe

Steuergerät: Mikrorechner, der neben der Zentraleinheit zur Bearbeitung arithmetischer Operationen und logischer Verknüpfungen spezielle Funktionsmodule zur Erfassung externer Signale und zur Erzeugung von Ansteuersignalen für externe Aktoren besitzt

Task: Ein durch ein Betriebssystem verwaltetes Programmteil

Tatsächliche minimale Ausführungszeit eines Codestücks ($BCET_A$): Zeit, die der Prozessor des Rechnersystems bei gegebenem Eingaberaum minimal mit der Ausführung eines gegebenen Codestückes zubringt

Tatsächliche maximale Ausführungszeit ($WCET_A$): Zeit, die der Prozessor des Rechnersystems bei gegebenem Eingaberaum maximal mit der Ausführung eines gegebenen Codestückes zubringt

Verfügbarkeit: Wahrscheinlichkeit, ein System zu einem beliebigen Zeitpunkt fehlerfrei anzutreffen

Verteilte Anwendung: Die gesamte Anwendung, die auf den einzelnen Knoten eines verteilten Systems ausgeführt wird

Zeitanalyse: Analyse, die das zeitliche Verhalten von Echtzeitsoftware mit dem Ziel untersucht, die zeitliche Korrektheit des Systems nachzuweisen

Zustandsautomat (engl. Finite State Machine): Hilfsmittel zur Beschreibung von ereignisdiskreten (bzw. reaktiven oder zustandsbasierten) Systemen. Ein Zustandsautomat wird durch eine endliche Menge von Zuständen und Zustandsübergängen beschrieben

Zuverlässigkeit: Fähigkeit eines Systems, während einer vorgegebenen Zeitdauer bei zulässigen Betriebsbedingungen die spezifizierte Funktion zu erbringen

Zweig: Gerichtete Kante im Kontrollflussgraf

Schedulability-Analyse: Analyse, ob eine Menge durch ein Betriebssystem verwalteter Programmteile unter den gegebenen Randbedingungen ausführbar ist

Zusammenfassung

Zeitgesteuerte Architekturen versprechen aufgrund ihrer deterministischen Eigenschaften die Anforderungen zukünftiger By-Wire-Systeme in Automobilen zu erfüllen. Im Vergleich zu herkömmlichen ereignisgesteuerten Architekturen benötigen sie jedoch einen erheblich höheren Entwicklungsaufwand und erfordern deshalb neue Verfahren und Werkzeugkonzepte.

In der vorliegenden Arbeit wird ein Verfahren zur Entwicklung und Analyse zeitgesteuerter Systeme erarbeitet. Aktivitäten zur Entwicklung zeitgesteuerter Systeme werden identifiziert und in den etablierten Entwicklungsprozess der Automobilindustrie eingegliedert. In den frühen Phasen des Entwicklungsprozesses erfolgt die Planung des zeitlichen Ablaufs im verteilten System. In den späten Phasen wird durch die Zeitanalyse nachgewiesen, dass die implementierten Programme geplante Zeitbedingungen stets einhalten. Die Bestimmung der maximalen Ausführungszeit von Programmen durch die Worst-Case-Execution-Time-Analyse (WCET-Analyse) ist hierfür die Grundlage und bildet den Schwerpunkt der Arbeit. Dem aktuellen Trend in der Automobilindustrie Software modellbasiert zu erstellen wird Rechnung getragen, indem ein Konzept der modellbasierten WCET-Analyse entwickelt wird. Die Analyse der erstellten Modelle wird dadurch ohne die Angabe zusätzlicher Informationen durch den Anwender möglich. Dazu werden aus den Modellen heraus Informationen über das Ausführungsverhalten des Codes gewonnen. Die Übertragbarkeit des Konzepts wird aufgezeigt, indem es bei den zwei unterschiedlichen Software-Entwicklungswerkzeugen ViPER und MATLAB/Simulink/Stateflow angewandt wird.

Zum praktischen Nachweis des erstellten Konzepts wird eine Werkzeugumgebung entwickelt. Anhand eines Steer-by-Wire-Fallbeispiels wird die Anwendbarkeit des Konzepts aufgezeigt. Durch die grafische Spezifikation des zeitlichen Ablaufs wird die Entwicklung verteilter regelungstechnischer Anwendungen unterstützt. Die integrierte WCET-Analyse für den in der Automobilindustrie weit verbreiteten Siemens 80C167 Mikrocontroller erfolgt voll automatisiert und bietet dem Anwender wichtige Informationen für den modellbasierten Softwareentwurf.

Abstract

Time-triggered architectures promise to fulfil the increasing requirements of future by-wire-systems in automobiles, because of their deterministic properties. However, compared to conventional event-triggered architectures they need a substantially higher development effort, and therefore new methods and tool concepts are required.

In this thesis a technique for the development and analysis of time-triggered systems is evolved. Supplementary activities for the development of time-triggered systems are identified and integrated into the established software development process of the automotive industry. In the early phases of the development process the planning of the required chronological sequence in the distributed system takes place. Conducting a timing analysis in the later phases proves that the planned timing requirements are fulfilled by the implemented programs under all possible circumstances. Being the basis for the timing analysis, the main emphasis of this thesis is the worst-case execution time analysis (WCET-analysis) of programs. The current trend of model-based software development in the automotive industry is considered by designing a concept for model-based WCET-analysis. The analysis of the designed models is done without declaration of additional information by the user. Necessary information about the dynamic execution behaviour of the code is generated from the models. The validity of the concept is demonstrated by applying it to two different software-development tools ViPER and MATLAB/ Simulink/Stateflow.

To prove the concept practically a tool environment has been developed. The validity of the concept is demonstrated by means of a steer-by-wire case study. The development of distributed control applications is supported by the graphical specification of the sequence of actions. The integrated WCET-analysis for the Siemens 80C167 microcontroller, which is widespread in the automotive industry, is performed automatically and offers the user important information for the model-based software development.

1 Einleitung

1.1 Zeitgesteuerte Architekturen für By-Wire-Systeme

Der Anteil der Elektronik in Kraftfahrzeugen hat in den letzten Jahren deutlich zugenommen. Laut Angaben eines führenden deutschen Automobilherstellers hat sich der Anteil der Elektronik an den Gesamtfahrzeugkosten in den letzten 10 Jahren von 20% auf 30% erhöht, obwohl auf dem Elektroniksektor Preissenkungen von bis zu 50% pro Jahr zu verzeichnen waren [Daim98]. Die Motivation für den massiven Einsatz von Elektronik ist vielfältig. Sie reicht vom Wunsch nach mehr Komfort, über die Verbrauchsreduzierung und die Berücksichtigung von Gesetzesvorschriften hinsichtlich aktiver und passiver Sicherheit, bis hin zur Reduzierung der Abgasemissionen sowie länderspezifischen Anforderungen [WiKr98].

Die Elektroniksysteme dringen dabei zunehmend in sicherheitsrelevante Bereiche vor. Durch den Einsatz von Software können Fahrzeugfunktionen realisiert werden, die durch rein mechanisch-hydraulische Systeme nicht oder nicht im selben Maße erreicht werden können. Beispiele hierfür sind Fahrerassistenzfunktionen wie Anti-Blockiersystem (ABS), Anti-Schlupfregelung (ASR), Electronic Stability Program (ESP) oder der Bremsassistent (BA). Sie unterstützen den Fahrer aktiv in kritischen Fahrsituationen und tragen somit zur Fahrsicherheit bei. Ihnen ist die Verbindung von Sensoren, Elektronikkomponenten, sog. *Steuergeräten*, und Regelungssoftware mit dem hydraulischen Bremssystem gemein. Bei der Fehlfunktion eines Steuergerätes greifen Sicherheitsmaßnahmen ein, die das Steuergerät kontrolliert abschalten, um gefährliche Auswirkungen auf das Fahrverhalten zu vermeiden. Durch das Abschalten des betroffenen Systems erfolgt ein automatisches Umschalten auf die hydraulische Rückfallebene, die den direkten Durchgriff auf die Bremsen sicherstellt.

Eine neue Generation von intelligenten Fahrerassistenzfunktionen soll den Fahrer zukünftig von Routineaufgaben entlasten, ihn in kritischen Fahrsituationen aktiv unterstützen und damit insgesamt die Fahrsicherheit erhöhen [DFM+97]. Die Zukunftsvision ist autonomes Fahren. Nächste zu realisierende Schritte hin zu dieser Vision sind die Umsetzung von Fahrerassistenzfunktionen wie automatisches Einparken, die dynamische Anpassung des Lenkverhältnisses und automatisches Halten der Spur [dSpa01]. Zur Umsetzung dieser Fahrerassistenzfunktionen durch den Einsatz von Software wird eine elektronische Schnittstelle zum Fahrwerk (Bremsen, Lenken, Federn) benötigt. Ein großes Potenzial steckt in der Kombination von Systemen für Bremsen (Brake-by-Wire) und Lenkung (Steer-by-Wire). Diese Systeme können modular aufgebaut werden; lokal begrenzte Aufgaben können vor Ort erledigt werden, übergeordnete Aufgaben werden durch die Vernetzung der einzelnen Steuergeräte mit Hilfe eines Kommunikationssystems realisiert [BHH+98].

Die Vorteile von By-Wire-Systemen sind vielfältig: Der Wegfall von Bauteilen wie Lenkstange oder Bremskraftverstärker bietet neue konstruktive Freiheiten und ermöglicht die Erhöhung der passiven Sicherheit in der Fahrgastzelle. Die stärkere Modularisierung trägt zu einer Vereinfachung der Montage bei der Produktion bei. Durch die individuelle Anpassung des Fahrwerks an die Wünsche des Fahrers (sportlich oder komfortabel) kann eine stärkere Personalisierung des Fahrzeugs erreicht werden.

By-Wire-Systeme werden ohne mechanische Rückfallebenen ausgelegt und gehören somit zu der Klasse der *fail-operational* Systeme [HeSe98], die auch im Falle eines auftretenden Fehlers noch einwandfrei funktionieren müssen. Sie stellen deshalb höchste Anforderungen bezüglich der Zuverlässigkeit und der Verfügbarkeit an die Elektronikarchitektur. Die zukünftigen Fahrerassistenzfunktionen sind verteilte Regelungssysteme, die über das Kommunikationssystem geschlossen werden und sehr hohe Echtzeitanforderungen an die Softwarearchitektur stellen [BBB+00]. Eine wesentliche Voraussetzung für die Umsetzung von By-Wire-Systemen ist eine Softwarearchitektur, die gleichzeitig den Anforderungen nach Zuverlässigkeit und Verfügbarkeit [BHH+98] und nach Unterstützung verteilter regelungstechnischer Anwendungen gerecht wird [BBB+00].

Zeitgesteuerte Architekturen erfüllen diese Anforderungen [BBB+00]. Ihre Kerneigenschaft ist die statische, zeitlich festgelegte Aktivierung von Aktionen im verteilten Softwaresystem. Das Prinzip der Zeitsteuerung ermöglicht die Realisierung von darauf aufbauenden Konzepten wie Fehlertoleranz durch Kopplung von Redundanzen und zeitgleiches Aktivieren von Aktionen, die den gestellten Anforderungen gerecht werden. Zeitgesteuerte Architekturen unterscheiden sich dadurch wesentlich von den bisher eingesetzten ereignisgesteuerten Architekturen.

1.2 Problemstellung bei der Entwicklung zeitgesteuerter Systeme

Bei zeitgesteuerten Systemen muss der Programmablauf des verteilten Softwaresystems bereits vor der Laufzeit, in der Entwicklungsphase, verbindlich festgelegt werden. Die Entscheidungen über den zeitlichen Ablauf von Programmteilen müssen bereits zur Entwicklungszeit getroffen werden, während sich bei herkömmlichen ereignisgesteuerten Systemen der zeitliche Ablauf aufgrund von dynamischen Entscheidungen zur Laufzeit mehr oder weniger zufällig ergibt. Das Prinzip der Zeitsteuerung hat somit einen erheblich höheren Entwicklungsaufwand zur Folge. Die daraus resultierenden Problemstellungen sind nachfolgend aufgeführt.

Festlegen des zeitlichen Ablaufs
In einem zeitgesteuerten System müssen Startzeitpunkte und Zeitpunkte, zu denen Programmteile abgearbeitet sein müssen, statisch eingeplant werden und es muss nachgewiesen werden, dass der geplante Programmablauf von der realen Implementierung in jedem Fall eingehalten

wird. Das Prinzip der Zeitsteuerung setzt somit voraus, dass die maximale Ausführungszeit der Programmteile bekannt ist. In Anbetracht der Tatsache, dass Software-Entwicklungswerkzeuge zunehmend Einzug in der automobilen Softwareentwicklung finden [BSW00], ist darüber hinaus die Kenntnis der maximalen Ausführungszeit von modellbasierter erstellter Software notwendig.

Planung verteilter regelungstechnischer Anwendungen
In einem zeitgesteuerten System lassen sich Programmteile einer regelungstechnischen Anwendung auf mehrere Steuergeräte verteilt anordnen, die aufgrund der netzwerkweiten Zeitsteuerung in eine direkte zeitliche Abhängigkeitsbeziehung gesetzt werden können. Die lokalen Abläufe auf den einzelnen Steuergeräten hängen somit über den gemeinsamen Ablauf der Kommunikation voneinander ab. Um eine möglichst optimale Auslegung der verteilten regelungstechnischen Anwendungen zu erreichen (geringe Latenzzeiten von Signalen und Zykluszeiten der Regelschleife), müssen alle Abläufe aufeinander abgestimmt werden.

Verteilte Entwicklung
Automobile werden in einer engen Kooperation zwischen dem Automobilhersteller und mehreren Zulieferern über mehrere Jahre entwickelt. Das notwendige Abstimmen der zeitlichen Abläufe in einem verteilten zeitgesteuerten System stellt somit eine neue zusätzliche Managementaufgabe dar, die bisher so nicht notwendig war. Ein definiertes Vorgehen ist erforderlich, um diese Aufgabe bewältigen zu können.

1.3 Ziel der Unterstützung der Entwicklung zeitgesteuerter Systeme

Aus der Problemstellung bei der Entwicklung zeitgesteuerter Systeme leitet sich die Zielsetzung der vorliegenden Arbeit ab: Durch die Bereitstellung geeigneter Verfahren und Werkzeugkonzepte soll die Entwicklung und Analyse zeitgesteuerter Systeme unterstützt und somit der Entwicklungsaufwand minimiert werden. Auf diese Weise soll die Grundlage für den Einsatz zeitgesteuerter Systeme in By-Wire-Systemen geschaffen werden. Hierbei haben die nachfolgenden Aspekte eine besondere Bedeutung.

Modellbasierte Worst-Case-Execution-Time-Analyse
Während der Entwicklung eines zeitgesteuerten Systems muss durch die Zeitanalyse der Nachweis erbracht werden, dass es sich in jedem Falle zeitlich korrekt verhält. Die Worst-Case-Execution-Time-Analyse (WCET-Analyse) bestimmt die maximale Ausführungszeit von Programmen und nimmt dabei eine Schlüsselrolle ein. Unter Beachtung der modellbasierten, werkzeugunterstützten Softwareentwicklung muss eine modellbasierte WCET-Analyse durchgeführt werden, um eine durchgängige Werkzeugkette zu erreichen. Dazu müssen die notwendigen Konzepte entwickelt und auf konkrete Werkzeuge übertragen werden.

Werkzeugkonzept für zeitgesteuerte Systeme
Ein Werkzeug für die Entwicklung zeitgesteuerter Systeme muss die Planung des zeitlichen Ablaufs im verteilten System, die Zeitanalyse sowie die Entwicklung verteilter regelungstechnischer Anwendungen unterstützen. Bei der Planung des zeitlichen Ablaufs ist darauf zu achten, dass die konkreten Vorstellungen der Entwickler über den zeitlichen Ablauf im verteilten System auf einfache intuitive Weise spezifiziert werden können, um somit den Entwurf der verteilten Regelkreise ausreichend zu unterstützen. Das Werkzeugkonzept sollte dabei in die bestehende Werkzeuglandschaft eingegliedert werden, um das Bestreben nach einer durchgängigen Werkzeugkette durch ein automatisches Ineinandergreifen einer Vielzahl spezialisierter Werkzeuge zu ermöglichen.

Entwicklungsprozess für zeitgesteuerte Systeme
Für die zusätzlich notwendigen Planungs- und Analyseaktivitäten zur Entwicklung zeitgesteuerter Systeme ist ein geeigneter Entwicklungsprozess zu definieren. Dabei sind die in der Automobilindustrie vorherrschende Vorgehensweise zur Softwareentwicklung, der üblicherweise große Zeithorizont bei der Entwicklung von Automobilen sowie die verteilte Entwicklung bei Automobilhersteller und Zulieferern zu berücksichtigen.

1.4 Übersicht über die Arbeit

Kapitel 2 gibt zunächst eine Einführung in zeitgesteuerte Architekturen und stellt die Grundlagen der Zeitanalyse und speziell der WCET-Analyse vor. Kapitel 3 untersucht den Stand der Technik auf den Gebieten der Entwicklung zeitgesteuerter Systeme und der WCET-Analyse. Der Bedeutung der Worst-Case-Execution-Time-Analyse bei der Entwicklung zeitgesteuerter Systeme entsprechend, wird im vierten Kapitel zunächst ein Konzept der modellbasierten WCET-Analyse erarbeitet. Dabei werden Kriterien herausgestellt, anhand derer Software-Entwicklungswerkzeuge bezüglich der Übertragbarkeit des erarbeiteten Konzepts bewertet werden können. Die drei darauf folgenden Kapitel gehen konkret auf die Entwicklung und Analyse zeitgesteuerter Systeme ein. Kapitel 5 stellt den neu definierten Entwicklungsprozess für zeitgesteuerte Systeme vor. Darauf aufbauend folgt in Kapitel 6 die Definition eines Werkzeugkonzepts für die Entwicklung zeitgesteuerter Systeme, in dem die modellbasierte WCET-Analyse eine zentrale Rolle einnimmt. Die Übertragbarkeit des Konzepts der modellbasierten WCET-Analyse wird dazu in Kapitel 7 anhand zweier Software-Entwicklungswerkzeuge beispielhaft aufgezeigt. Bei den untersuchten Software-Entwicklungswerkzeugen handelt es sich um das am IAS entwickelte Werkzeug *ViPER/ESTEREL* und das kommerzielle Werkzeug *MATLAB/Simulink/ Stateflow*. In Kapitel 8 werden die im Rahmen der vorliegenden Arbeit erstellten Softwarewerkzeuge vorgestellt. Anhand eines Beispiels wird in Kapitel 9 die Entwicklung eines zeitgesteuerten Systems aufgezeigt und die modellbasierte WCET-Analyse durchgeführt. Abschließend werden in Kapitel 10 die Ergebnisse der vorliegenden Arbeit zusammengefasst und ein Ausblick auf weiterführende Arbeiten gegeben.

2 Grundlagen

In diesem Kapitel werden die zum Verständnis der Arbeit notwendigen Grundlagen bereitgestellt. Im ersten Abschnitt werden die Grundlagen zeitgesteuerter Architekturen beschrieben. Dazu werden zunächst grundlegende Eigenschaften von Echtzeitsystemen aufgeführt und zugehörige Begriffe definiert. Anschließend werden die Unterschiede von ereignisgesteuerten und zeitgesteuerten Echtzeitsystemen diskutiert. Im Anschluss werden Grundprinzipen der zeitgesteuerten Architektur vorgestellt und deren Eignung für By-Wire-Systeme aufgezeigt. Im zweiten Abschnitt werden die Grundlagen der Zeitanalyse von Echtzeitsoftwaresystemen vorgestellt. Dazu werden zunächst die verschiedenen Teilgebiete der Zeitanalyse unterschieden. Im Anschluss an die Definition der Anforderungen an die WCET-Analyse werden wesentliche Begriffe der WCET-Analyse und Begriffe zur Beschreibung des Kontrollflusses in Programmen definiert. Anschließend werden grundsätzliche Vorgehensweisen der WCET-Analyse diskutiert.

2.1 Grundlagen der zeitgesteuerten Architekturen

2.1.1 Eigenschaften von Echtzeitsystemen

Als Echtzeitsysteme werden Rechnersysteme bezeichnet, die mit externen technischen Systemen in einer Wechselwirkung stehen und die technische Prozesse steuern oder regeln. Der Echtzeitbetrieb unterscheidet sich von der allgemeinen Datenverarbeitung durch das explizite Hinzutreten der Dimension *Zeit* [LaGö99]. So muss in einem Echtzeitsystem die Verarbeitung der Programme zeitlich mit den im technischen System ablaufenden Vorgängen Schritt halten. Die Menge aller Programme, die zur Erfüllung der gestellten Aufgabe erforderlich sind, werden hierbei als Echtzeitsoftwaresystem bezeichnet. Die wesentlichen Anforderungen, die im Allg. an Echtzeitsysteme gestellt werden, sind:

- Rechtzeitigkeit
- Gleichzeitigkeit
- Zuverlässigkeit
- Verfügbarkeit
- Vorhersehbarkeit

Die Forderung nach Rechtzeitigkeit bedeutet, dass Erfassung, Verarbeitung und Ausgabe von Daten sowie daraus abgeleitete Reaktionen pünktlich ausgeführt werden müssen. Nicht Schnelligkeit, sondern Reaktion innerhalb von vorgegebenen und vorhersagbaren Zeitschranken ist hierbei entscheidend, denn die Korrektheit des Systems hängt nicht nur vom Ergebnis der Berechnung, sondern auch davon ab, wann dieses Ergebnis produziert wird. Die Forderung nach

Gleichzeitigkeit besagt, dass mehrere Teilvorgänge gleichzeitig von einem Rechnersystem zu kontrollieren sind. Dieser Anforderung muss das Rechnersystem durch nebenläufige Ausführung von Programmen gerecht werden. Die Forderung nach Zuverlässigkeit ist von Bedeutung bei Echtzeitsystemen, bei denen ein Ausfall des Rechner- und Kommunikationssystems zu einer Gefährdung von großen Sachwerten oder gar von Menschenleben führen würde. Die Forderung nach Verfügbarkeit bedeutet, dass ein System zu einem vorgegebenen Zeitpunkt mit einer bestimmten Wahrscheinlichkeit in einem funktionsfähigen Zustand anzutreffen sein muss. Die Forderung nach Vorhersagbarkeit besagt, dass alle durch das Rechnersystem auszuführenden Aktionen vor der Laufzeit bekannt und deterministisch sein müssen. Dies gilt insbesondere bei Überlast und Fehlersituationen.

Verteilte Echtzeitsysteme bestehen aus einer Menge von *Rechner-Knoten* (Steuergeräte), die über ein *Echtzeit-Kommunikationssystem* Daten austauschen. Durch eine verteilte Architektur kann dem etablierten Entwurfsprinzip gefolgt werden, die physikalische Form des technischen Systems auf das Echtzeitsystem abzubilden. Ein Knoten kann so als verstehbare Abstraktion angesehen werden, der essentielle Funktionen und zeitliche Eigenschaften eines Knoten von Irrelevantem trennt [Kope97a]. Darüber hinaus haben verteilte Architekturen im Vergleich zu zentralisierten Architekturen den Vorteil, dass Fehler von Subsystemen besser gekapselt und Redundanzkonzepte einfacher realisiert werden können.

2.1.2 Zeitgesteuerte versus ereignisgesteuerte Systeme

Nach Art des Programmierverfahrens lassen sich zwei grundsätzlich verschiedene Arten von Echtzeitsystemen unterscheiden [LaGö99]:

- Zeitgesteuerte Echtzeitsysteme
- Ereignisgesteuerte Echtzeitsysteme

Bei zeitgesteuerten Echtzeitsystemen werden alle Aktivitäten (Ablauf der Programme, Senden von Nachrichten) zu vorher bestimmen Zeitpunkten angestoßen [LaGö99]. Die Zustandsgrößen des technischen Prozesses werden zu festgelegten Zeitpunkten erfasst, alle Teilprogramme werden periodisch ausgeführt. Die zeitlichen Signale für die Aktivierung von Aktionen werden vom Fortschreiten der Zeit abgeleitet. Wenn immer die Echtzeituhr eines Knoten einen zuvor bestimmten Zeitpunkt erreicht, wird ein Aktivierungssignal erzeugt [Kope97a]. Bei zeitgesteuerten Echtzeitsystemen kann es deshalb zu keinen Überlastsituationen durch eine Vielzahl nacheinander eintreffender Ereignisse kommen. Sie sind zeitlich deterministisch und somit einfacher zu analysieren und zu testen als ereignisgesteuerte Echtzeitsysteme. Im Gegenzug sind sie inflexibel und eignen sich deshalb vornehmlich für zyklische wiederkehrende Vorgänge.

Bei ereignisgesteuerten Echtzeitsystemen werden Aktivitäten durch Ereignisse ausgelöst. Die Organisation der Abläufe erfolgt zur Laufzeit. Die zeitlichen Signale für die Aktivierung von

Aktionen werden aus einer signifikanten Zustandsänderung abgeleitet. Solche Ereignisse können vom Rechnersystem selbst oder vom technischen Prozess stammen [Kope97a]. Beispiele hierfür sind Eintreffen einer Nachricht an einem Knoten oder das Beenden einer Task. Ereignisgesteuerte Systeme ermöglichen die schnelle Reaktion auf asynchrone Vorgänge, sind jedoch nicht deterministisch.

2.1.3 Eigenschaften einer zeitgesteuerten Architektur

Die wesentlichen Elemente einer zeitgesteuerten Architektur sind in Abbildung 2.1 dargestellt. Das *verteilte System*, auch *Cluster* genannt, besteht aus einer Menge von *Knoten* (Steuergeräte), die über ein *Broadcast*[1]*-Kommunikationssystem* bzw. *Bussystem* kommunizieren. Zwei oder mehrere Knoten können zu redundanten Einheiten zusammengefasst werden, die in statischer Redundanz dieselbe Aufgabe wahrnehmen.

Eine zeitgesteuerte Architektur wird im Wesentlichen durch zwei Subsysteme realisiert, durch ein *zeitgesteuertes Kommunikationssystem*, das auf allen Kommunikationscontrollern der Knoten verteilt ausgeführt wird und durch *zeitgesteuerte Betriebssysteme*, die auf den *Host*-Rechnern lokal ausgeführt werden. Die zeitgesteuerten Betriebssysteme bringen zeitgesteuert Anwendungssoftware zur Ausführung. Die Anwendungssoftware wird in abgeschlossenen Einheiten (*Tasks*) ausgeführt. Die gesamte Anwendung, die auf den einzelnen Knoten des Systems ausgeführt wird, wird zusammen als *verteilte Anwendung* bezeichnet.

Das Kommunikationssystem kann redundante Kommunikationskanäle besitzen. Dadurch wird Fehlertoleranz auf der Ebene des Kommunikationssystems erreicht.

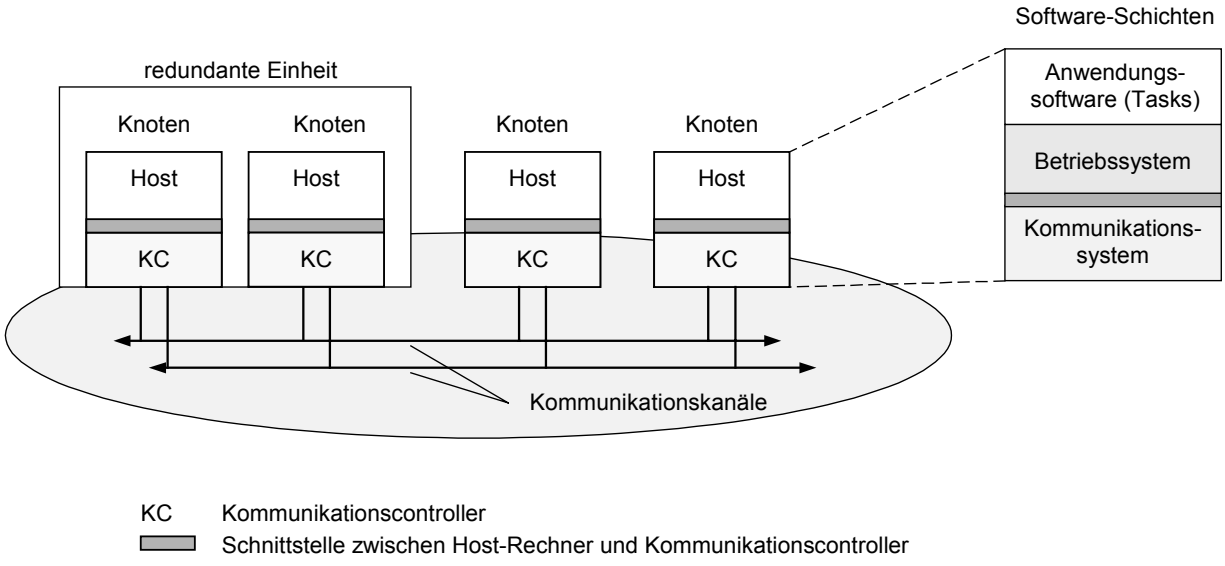


Abbildung 2.1: Zeitgesteuerte Architektur

[1] Jeder Knoten empfängt die Nachrichten aller übrigen Knoten.

Eigenschaften
Die wesentlichen Eigenschaften einer zeitgesteuerten Architektur sind nachfolgend zusammengestellt [KoGr92, KoGr94]:

- Eine zeitgesteuerte Architektur verfügt über eine global synchronisierte Uhrzeit, auf die sich alle Knoten beziehen. Aktionen werden durch das Fortschreiten der Zeit ausgelöst (Prinzip der Zeitsteuerung). Die Aktivierung der Aktionen erfolgt bezüglich der global synchronisierten Uhrzeit.

- Die Zeitpunkte für die Aktivierung der Aktionen sind statisch festgelegt. Das System ist somit weitgehend konstant ausgelastet. Die Aktionen werden zyklisch abgearbeitet.

- Die Sicht auf die Systemumgebung erfolgt als Momentaufnahme. Alle Knoten haben somit eine konsistente Sicht.

- Die Semantik der Nachrichten ist zustandsbasiert. Es kann somit zu keiner Überlastung des Systems durch schnelle aufeinander folgende Ereignisse kommen.

Subsysteme
Das zeitgesteuerte Kommunikationssystem ist für das zeitgesteuerte Senden und Empfangen von Nachrichten zuständig. Der Zugriff auf das Kommunikationsmedium erfolgt kollisions- und arbitrierungsfrei durch ein *Time Division Multiple Access* (TDMA) Verfahren zu bestimmten Zeitschlitzen. Die Information, wann welcher Teilnehmer eine Nachricht sendet, ist in einem sog. *Nachrichten-Schedule* gespeichert. Die Abarbeitung des Nachrichten-Schedules erfolgt zyklisch und wird Kommunikationszyklus genannt.

Die Hauptaufgabe eines zeitgesteuerten Betriebssystems ist das *Task Management,* d. h. die zeitgesteuerte Aktivierung von Tasks und die Überwachung derer Deadlines. Die Task-Aktivierung erfolgt statisch bezüglich der globalen Uhrzeit, die vom Kommunikationssystem zur Verfügung gestellt wird. Die Aktivierungszeitpunkte sind in einem *Task Schedule* gespeichert, der zyklisch abgearbeitet wird. Dessen Wiederholung wird *Anwendungszyklus* genannt. Ein zeitgesteuertes Betriebssystem muss sich vorhersagbar verhalten und es muss bzgl. seiner zeitlichen Eigenschaften statisch analysierbar sein [Kope97a]. Es darf deshalb keine dynamischen, unberechenbaren Mechanismen aufweisen, wie beispielsweise dynamische Task-Aktivierungen.

2.1.4 Software-Architektur für By-Wire-Systeme

By-Wire-Systeme besitzen gegenüber den konventionellen mechanischen oder hydraulischen Fahrwerksystemen im Automobil neue Eigenschaften, die spezifische Anforderungen an das Echtzeitsystem stellen. By-Wire-Systeme zeichnen sich durch eine fehlende mechanische, kräfteübertragende Verbindung zwischen Fahrer und Fahrwerk als technischem System aus. Stattdessen wird die Kopplung über ein verteiltes Elektroniksystem hergestellt. Steuergeräte nehmen

über Sensoren den Wunsch des Fahrers und weitere physikalische Größen auf (Sensorsignale) und stellen sie dem verteilten Elektroniksystem über ein Kommunikationssystem zur Verfügung. Weitere Steuergeräte verarbeiten die Sensorsignale und berechnen Stellsignale, die über Aktoren das Fahrverhalten beeinflussen.

Die Berechnung der Stellsignale erfolgt durch eine verteilte regelungstechnische Anwendungssoftware, deren Regelkreis über das verteilte Elektroniksystem geschlossen wird. Daraus resultieren sehr hohe Anforderungen bezüglich Rechtzeitigkeit und Gleichzeitigkeit an die Softwarearchitektur des verteilten Elektroniksystems. Die Reglergüte der verteilten Regelungssysteme hängt sehr stark von der äquidistanten Erfassung von Sensorsignalen, Ausführung der verteilten regelungstechnischen Anwendungssoftware und Ansteuerung der Aktoren auf den Steuergeräten ab. Verteilte Regelkreise erfordern deshalb die Einhaltung einer festgelegten zeitlichen Wirkungskette [BBB+00]. Sowohl eine äquidistante Aktivierung der verantwortlichen Programmteile auf den Steuergeräten als auch die äquidistante Übertragung von Nachrichten über das Kommunikationssystem ist notwendig. Die Nachrichtenübertragung darf nur eine sehr geringe zeitliche Abweichung (engl. *Jitter*) aufweisen. Steer- oder Brake-by-Wire-Systeme erfordern sehr kurze Zykluszeiten der verteilten Regelungssysteme, in der Größenordnung von wenigen Millisekunden. Die kurzen Zykluszeiten verlangen sehr hohe Übertragungsgeschwindigkeiten von kleinen Datenrahmen (engl. *Frames*) mit sehr kurzer Verzögerungszeit (*Latenzzeit*) zwischen dem Senden und Empfangen.

By-Wire-Systeme besitzen keine mechanische bzw. hydraulische Rückfallebene, auf die beim Ausfall des Elektroniksystems zurückgegriffen werden könnte. Der Ausfall des Elektroniksystems kann schwerwiegende Folgen haben. By-Wire-Systeme stellen deshalb neue Anforderungen bezüglich Zuverlässigkeit und Verfügbarkeit an die Softwarearchitektur. Bei Automobilen wird im Allg. davon ausgegangen, dass jeder beliebige Einfachfehler toleriert werden muss [BBB+00]. Ein Mittel, um Fehlertoleranz zu erreichen ist die mehrfache sog. redundante Auslegung von Einheiten. In der Literatur [Echt90] sind zahlreiche Redundanzkonzepte bekannt. Bei By-Wire-Systemen kommt aufgrund der sehr hohen Echtzeit-Anforderungen nur das Prinzip der *statischen Redundanz* in Frage, bei der alle redundanten Einheiten während des gesamten Einsatzzeitraumes aktiv arbeiten. Dynamische Redundanz, bei der eine redundante Einheit erst nach Auftreten eines Fehlers aktiviert wird, würde zu viel Zeit in Anspruch nehmen. Einheiten in statischer Redundanz müssen zeitgleich die gleichen Aktionen ausführen, um in einer sich schnell ändernden Umwelt dieselben Ergebnisse zu berechnen. Redundante Einheiten müssen sich deshalb zeitlich vorhersagbar (deterministisch) verhalten. Hierzu wird eine Softwarearchitektur benötigt, die es ermöglicht, Aktionen auf verteilten Steuergeräten synchron zu aktivieren.

Die Entwicklungsphase von Automobilen erstreckt sich über mehrere Jahre und Automobile werden über Jahrzehnte genutzt. Daraus ergeben sich Anforderungen an die Softwarearchitektur bezüglich Flexibilität. Während der Produktion von Fahrzeugen einer Baureihe können Änderungen und Nachbesserungen erforderlich werden. Die Softwarearchitektur muss deshalb eine

flexible Systemerweiterung beispielsweise in Form von Aufnahme weiterer Steuergeräte ermöglichen. Des Weiteren ergibt sich aus der langen Nutzungsphase eines Automobils die Anforderung, dass die Softwarearchitektur für den Werkstattbesuch Fehlerdiagnose und Software-Aktualisierungen unterstützen muss.

Sowohl die Anforderungen bezüglich Rechtzeitigkeit und Gleichzeitigkeit der verteilten Regelungssysteme als auch die Anforderungen bezüglich Zuverlässigkeit und Verfügbarkeit an die Softwarearchitektur führen zu dem Prinzip der Zeitsteuerung. Die reine Zeitsteuerung widerspricht jedoch der Anforderung bezüglich Flexibilität. Denn eine permanente Reservierung von nur sehr selten genutzter Rechenzeit und Kommunikationsbandbreite ist teuer und somit nicht realisierbar. Es wird deshalb neben der Zeitsteuerung ein gewisses Maß an asynchronen, ereignisgesteuerten Anteilen benötigt. Um den Anforderungen bezüglich Rechtzeitigkeit und Gleichzeitigkeit zu genügen, dürfen sie jedoch das Prinzip der Zeitsteuerung nicht verletzen, sondern müssen sich ihm unterordnen.

Der Stand der Technik auf dem Gebiet der Elektronikarchitektur für By-Wire-Systeme und deren Entwicklung wird im Kapitel 3 untersucht. Im Anschluss wird auf die Grundlagen der Zeitanalyse von Echtzeitsoftwaresystemen eingegangen.

2.2 Grundlagen der Zeitanalyse von Echtzeitsoftwaresystemen

2.2.1 Teilgebiete der Zeitanalyse von Echtzeitsoftwaresystemen

Die Zeitanalyse von Echtzeitsoftwaresystemen untersucht das zeitliche Verhalten von Echtzeitsoftware mit dem Ziel, die zeitliche Korrektheit des Systems nachzuweisen [PuVr96]. Sie besteht aus den Teilgebieten Schedulability-Analyse, Worst-Case-Execution-Time-Analyse und Analyse der Hardwareaktivitäten (siehe Abbildung 2.2). Durch die Schedulability-Analyse wird überprüft, ob eine Menge der durch ein Betriebssystem verwalteten Programmteile, sog. *Tasks,* unter den gegebenen Randbedingungen ausführbar ist. Eine gegebene Menge von Tasks ist dann ausführbar, wenn jede Task den spätesten Zeitpunkt, zu dem die Task beendet sein muss (*Deadline*), einhält [Klei93]. Randbedingungen sind beispielsweise Reihenfolgebeziehungen der Tasks untereinander sowie zeitliche Anforderungen in Form von bestimmten Zyklen, in denen die Tasks ausgeführt werden müssen sowie globale Abhängigkeiten, hervorgerufen durch Ankopplung an ein Kommunikationssystem. Die Schedulability-Analyse bedient sich der Informationen der WCET-Analyse und der Analyse der Hardwareaktivitäten. Im Rahmen der Analyse der Hardwareaktivitäten werden Aktionen der Hardware analysiert, die zusätzlich, parallel zur Ausführung der Software, ausgeführt werden. Dies sind beispielsweise Refresh-Zyklen von dynamischen Speicherbausteinen, Direct-Memory-Access-Aktivitäten (DMA) und Unterbrechungen durch Hardware-Interrupts. Die WCET-Analyse bestimmt die maximale Ausfüh-

rungszeit eines Codestücks, das frei von Unterbrechungen ist und stellt somit die grundlegenden Ausführungszeiten der Codestücke, bzw. Tasks zur Verfügung. Die gestellten Anforderungen an sie werden im Folgenden analysiert.

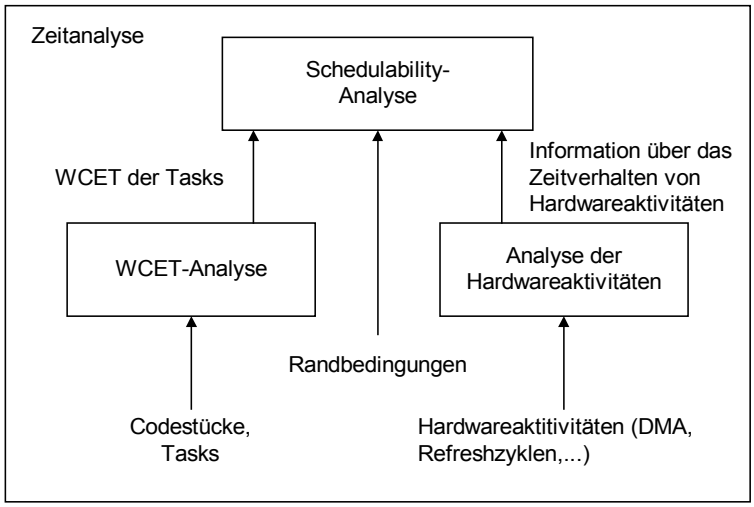


Abbildung 2.2: Teilgebiete der Zeitanalyse

2.2.2 Anforderungen an die WCET-Analyse

Das Prinzip der Zeitsteuerung der zeitgesteuerten Architektur setzt voraus, dass eine obere Schranke der maximalen Ausführungszeit bekannt ist. Da die zeitgesteuerten Systeme in dem Massenprodukt Automobil ihren Einsatz finden, führt der Kostendruck dazu, möglichst langsame und somit kostengünstige Prozessoren der Zielhardware zu verwenden, die den Programmcode ausführen, ohne die gestellten Echtzeit-Anforderungen zu verletzen. Dies impliziert eine möglichst genaue Bestimmung der maximalen Ausführungszeit. Die Anforderungen an die WCET-Analyse lassen sich somit folgendermaßen formulieren:

> *Die ermittelten maximalen Ausführungszeiten müssen obere und enge Schranken der tatsächlichen maximalen Ausführungszeit des Programmcodes sein.*

2.2.3 Begriffe zur WCET-Analyse

Aus der Definition der Anforderungen an die WCET-Analyse lassen sich Definitionen grundlegender Begriffe ableiten. In Anlehnung an [Pusc93] werden in der vorliegenden Arbeit folgende Definitionen für Ausführungszeiten von Codestücken verwendet (siehe Abbildung 2.3).

Ausführungszeit T eines Codestücks: Die Ausführungszeit T eines Codestücks auf einem Rechnersystem ist die Zeit, die der Prozessor des Rechnersystems mit der Ausführung dieses Codestücks zubringt.

Tatsächliche maximale Ausführungszeit eines Codestücks: Die tatsächliche maximale Ausführungszeit (Actual Worst-Case Execution Time – $WCET_A$) eines Codestücks auf einem Rechnersystem ist die Zeit, die der Prozessor des Rechnersystems bei gegebenem Eingaberaum maximal mit der Ausführung dieses Codestücks zubringt.

Berechnete maximale Ausführungszeit eines Codestücks: Die berechnete maximale Ausführungszeit (Computed Worst-Case Execution Time – $WCET_C$) eines Codestücks ist eine rechnerisch ermittelte obere Schranke für die maximale Ausführungszeit dieses Codestücks.

Tatsächliche minimale Ausführungszeit eines Codestücks: Die tatsächliche minimale Ausführungszeit (Actual Best-Case Execution Time – $BCET_A$) eines Codestücks auf einem Rechnersystem ist die Zeit, die der Prozessor des Rechnersystems bei gegebenem Eingaberaum minimal mit der Ausführung dieses Codestücks zubringt.

Berechnete minimale Ausführungszeit: Die berechnete minimale Ausführungszeit (Computed Best-Case Execution Time – $BCET_C$) eines Codestücks ist eine rechnerisch ermittelte untere Schranke für die minimale Ausführungszeit dieses Codestücks.

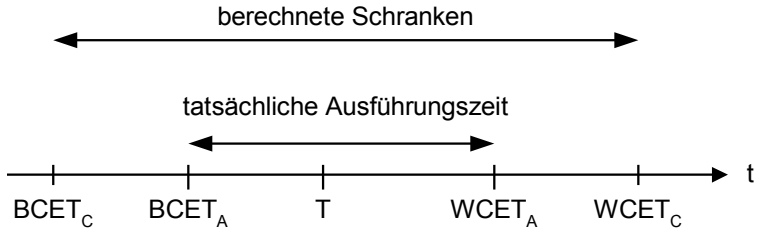


Abbildung 2.3: Definitionen der Ausführungszeiten von Codestücken

2.2.4 Begriffe zur Beschreibung des Kontrollflusses in Programmen

Zur Bestimmung der Ausführungszeit wird eine Beschreibung des Kontrollflusses in Programmen in Form eines Kontrollflussgrafen benötigt. Ein Kontrollflussgraf ist ein gerichteter Graf, der aus einer endlichen Menge von Knoten besteht [Balz96], die jeweils eine ausführbare Anweisung darstellen. Die gerichteten Kanten zwischen zwei Knoten beschreiben einen möglichen Kontrollfluss zwischen zwei Knoten und werden als *Zweige* bezeichnet (siehe Abbildung 2.4). Die maximale Anzahl von Anweisungen ohne Verzweigung wird als *Grundblock* (engl. *Basic Block*) bezeichnet [ASU88]. Ein Kontrollflussgraf besitzt einen Startknoten und einen Endknoten. Eine Folge von Knoten und Kanten, die mit dem Startknoten beginnt und mit einem Endknoten endet, wird als *Pfad* bezeichnet.

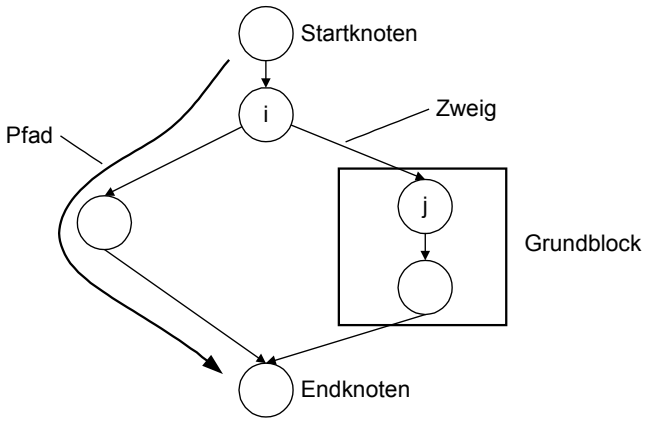


Abbildung 2.4: Elemente eines Kontrollflussgrafs

2.2.5 Grundsätzliche Vorgehensweisen zur WCET-Analyse

2.2.5.1 Dynamische WCET-Analyse

Die Ausführungszeit eines Codestücks kann grundsätzlich entweder durch eine dynamische oder eine statische Analyse bestimmt werden. Die dynamische WCET-Analyse ermittelt Informationen während des Programmablaufs [KoPl96]. Sie basiert auf den Messungen der Ausführungszeit auf der Zielhardware. Das Ziel der dynamischen Analyse ist es, die Eingangsparameter und den internen Zustand eines Programms zu finden, die zu der Ausführung des Worst-Case-Pfades (Pfad mit der WCET) führen. Sie reicht vom unkoordinierten Messen der Ausführungszeit während der Ausführung des Programmcodes bis hin zur gezielten Veränderung der Eingangsparameter durch die Anwendung von Heuristiken, beispielsweise genetischer Algorithmen, die die Eingangsparameter in Richtung größere Ausführungszeit verändern [AHP99].

Bei der dynamischen Analyse besteht prinzipiell das Problem, dass die gemessene Ausführungszeit sich „von unten“ an die maximale Ausführungszeit nähert. Es besteht keine Garantie, die maximale Ausführungszeit gemessen zu haben. Darüber hinaus fehlt die Information darüber, wie stark die gemessene Ausführungszeit von der maximalen Ausführungszeit abweicht. Die dynamische Analyse widerspricht somit den oben aufgeführten Anforderungen und kann deshalb bei den sicherheitsrelevanten By-Wire-Systemen nicht angewandt werden. Aus diesem Grund wird sie in der vorliegenden Arbeit nicht weiter verfolgt.

2.2.5.2 Statische WCET-Analyse

Bei der statischen WCET-Analyse wird die maximale Ausführungszeit des Programmcodes während oder nach dem Compilierungsprozess analytisch bestimmt, ohne ihn auf der Zielhardware auszuführen. Die Schwierigkeit besteht darin, dass der Worst-Case-Pfad äußerst selten bekannt ist. Ist der Worst-Case-Pfad bekannt, so wird das Problem trivial. Durch Erzwingung dieses Pfades kann die zugehörige Ausführungszeit durch eine einfache Messung bestimmt werden [LiMa99]. Bei einem gegebenen Codestück hängt die Ausführungszeit von zwei Hauptfaktoren

ab, einerseits von den möglichen Ausführungspfaden des Codestücks und andererseits von den Ausführungszeiten jeder Instruktion dieses Pfades. Dementsprechend werden sowohl die ausführbaren Pfade des Codestücks (Pfadmodellierung) als auch die Zielhardware (Mikroarchitekturmodellierung) modelliert.

Pfadmodellierung

Die Pfadmodellierung hat die Aufgabe, die ausführbaren Pfade eines Programmcodes auf möglichst effiziente Art und Weise mathematisch darzustellen. Die Bestimmung der maximalen Ausführungszeit für ein beliebiges Codestück ist im Allg. unmöglich. Das würde bedeuten, das unentscheidbare Halteproblem[2] zu lösen [StAl97, PuVr96]. Es muss deshalb sichergestellt werden, dass das Codestück in jedem Fall terminiert und dass die möglichen Pfade statisch bestimmt werden können. Dafür müssen bestimmte Einschränkungen im Sprachumfang der Programmiersprache in Form von Codierungsrichtlinien (nur abgeschrankte Schleifen, keine dynamischen Konstrukte etc.) getroffen werden, die nur zeitlich deterministische Codestrukturen erlauben. Eventuell sind zusätzliche Angaben über mögliche Pfade notwendig. Die statische Analyse des Programmcodes alleine reicht in der Regel nicht aus, um eine enge Schranke der $WCET_A$ zu bestimmen. Nicht alle statisch möglichen Pfade sind auch ausführbar; aufgrund von Abhängigkeiten zwischen Zweigen im Programmcode sind manche Pfade nicht möglich. In Abbildung 2.5 ist beispielsweise der gekennzeichnete Pfad nicht ausführbar, denn der Parameter *a* kann nicht gleichzeitig 1 und 0 sein, sofern er im dazwischenliegenden Codestück nicht verändert wird. Um möglichst enge Schranken zu erhalten, werden deshalb zusätzlich semantische Informationen über das dynamische Ausführungsverhalten des Programms, sog. Pfadinformationen, benötigt.

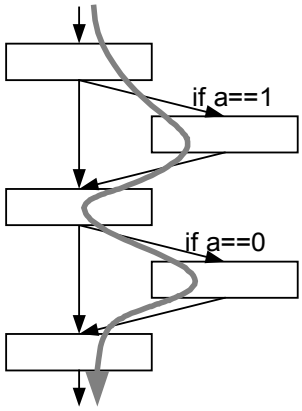


Abbildung 2.5: Beispiel für einen nicht ausführbaren Pfad

[2] Das allgemeine Halteproblem für die (programmierbare) Turing-Maschine ist wie folgt: Gibt es einen Algorithmus, der zu einem beliebigen Turingprogramm P und beliebigem Argument x entscheidet, ob die Turing-Maschine bei Eingabe von x mit dem Programm P nach endlich vielen Schritten stoppt oder nicht? Das allgemeine Halteproblem ist unentscheidbar.

Mikroarchitekturmodellierung

Die Mikroarchitekturmodellierung hat die Aufgabe, ein hinreichend genaues Abbild der Zielhardware zu erstellen, um die Ausführungszeiten von Grundblöcken ermitteln zu können. Die Ausführungszeiten eines Grundblocks setzen sich hierbei aus den elementaren Ausführungszeiten der Instruktionen auf Assemblercode-Ebene zusammen. Der Aufwand der Mikroarchitekturmodellierung hängt entscheidend von der Komplexität der Zielhardware ab, insbesondere davon, ob die Architektur *Pipelines* und *Caches* enthält. Pipelines führen mehrere Befehle, die sich in verschiedenen Phasen der Programmausführung befinden, gleichzeitig aus. Die Ausführung einer Instruktion ist somit abhängig von den anderen Instruktionen, die sich im Prozessor befinden. Caches sind schnelle Pufferspeicher, die zwischen Hauptspeicher und Prozessor geschaltet sind. Sie besitzen relativ kleine Speicherkapazitäten und können deshalb jeweils nur einen bestimmten Speicherbereich zwischenspeichern. Der Aufwand der Mikroarchitekturmodellierung kann somit in folgende drei Fälle unterschieden werden:

- *Im Falle einer einfachen* Prozessorarchitektur *ohne Pipelines und Caches* ist die Ausführungszeit einer Instruktion unabhängig von anderen Instruktionen. Die Ausführungszeit eines Grundblocks ist die Summe der Ausführungszeiten aller Instruktionen.

- *Im Falle einer* Prozessorarchitektur *mit Pipelines ohne Caches* hängt die Ausführungszeit einer Instruktion ab von Instruktionen, die gleichzeitig ausgeführt werden. Dabei handelt es sich um ein lokal begrenztes Problem; die Ausführungszeit kann durch Simulation der Sequenz der Instruktionen des Grundblocks ermittelt werden. Dabei müssen die Überlappungen von Instruktionen in der Pipeline an Grundblock-Grenzen getrennt betrachtet werden.

- *Im Falle von Caches* hängt die Ausführungszeit einer Instruktion auch von den Instruktionen und Daten ab, die in der Vergangenheit adressiert wurden. Das ist ein globales Problem, da die Instruktionen und Daten, auf die zuvor zugegriffen wurde, von verschiedenen Funktionen oder Modulen stammen können. Bei Instruktions-Caches sind die möglichen Instruktionsadressen nach dem Compilierungsprozess bekannt. Bei Daten-Caches ist dies im Allg. nicht bekannt. Bei Daten-Caches müssen für alle Datenreferenzen unter Berücksichtigung der Zugriffsreihenfolge die Adressen berücksichtigt werden, auf die zuvor zugegriffen werden konnte.

In diesem Kapitel wurde die Basis zum Verständnis der vorliegenden Arbeit geschaffen. Die wesentlichen Prinzipen der zeitgesteuerten Architektur wurden vorgestellt und deren Eignung für By-Wire-Systeme aufgezeigt. Anschließend wurden die Grundlagen der Zeitanalyse von Echtzeitsoftwaresystemen vorgestellt und detailliert auf WCET-Analyse eingegangen, die die Basis für die Entwicklung zeitgesteuerter Systeme darstellt. Aufbauend auf den vorgestellten Grundlagen wird im anschließenden Kapitel der Stand der Technik auf dem Gebiet der Entwicklung zeitgesteuerter Systeme und dem Gebiet WCET-Analyse analysiert und bewertet.

3 Stand der Technik

Das Ziel der vorliegenden Arbeit ist es, die Entwicklung zeitgesteuerter Systeme durch die Bereitstellung von Verfahren und Werkzeugkonzepten zu unterstützen. In diesem Kapitel wird dazu der derzeitige Stand der Technik untersucht. Zunächst wird im folgenden Abschnitt der Stand der Technik auf dem Gebiet der Entwicklung zeitgesteuerter Systeme analysiert und bewertet. Wie in Kapitel 1 festgestellt, hat die WCET-Analyse bei der Entwicklung zeitgesteuerter Systeme eine besondere Bedeutung. Im darauf folgenden Abschnitt wird dazu der derzeitige Wissensstand untersucht. Abschließend werden die wesentlichen Erkenntnisse zusammengefasst und resultierende Anforderungen an die vorliegende Arbeit festgelegt.

3.1 Stand der Technik bei der Entwicklung zeitgesteuerter Systeme

3.1.1 Übersicht über die Entwicklung zeitgesteuerter Systeme

Bei der Entwicklung zeitgesteuerter Systeme spielen Standards für zeitgesteuerte Kommunikations- und Betriebssysteme eine wichtige Rolle. Im Folgenden werden zwei erfolgversprechende Vertreter, *FlexRay* und *OSEKtime,* vorgestellt. Anschließend werden in der Automobilindustrie etablierte Software-Entwicklungswerkzeuge bezüglich ihrer Unterstützung der Entwicklung zeitgesteuerter Systeme untersucht. Im Anschluss daran werden die an der TU Wien entwickelte Entwicklungsmethodik, die sog. Schnittstellen-fokussierte Methodik, sowie die im Rahmen des EU-Projekts *Time-Triggered Architecture – TTA* [SHS+97] definierte Werkzeugumgebung *SETTA* analysiert.

Mittlerweile sind von den Firmen *TTTech* [TTTe] und *DeComSys* [DCS] erste Werkzeuge für die Konfiguration der zeitgesteuerten Kommunikationssysteme TTP/C bzw. FlexRay verfügbar. Diese Werkzeuge werden anschließend vorgestellt und bewertet.

3.1.2 Standards für zeitgesteuerte Systeme

3.1.2.1 Zeitgesteuertes Kommunikationssystem FlexRay

In den letzten zehn Jahren wurden mehrere neue zeitgesteuerte Kommunikationssysteme wie beispielsweise *Arinc 629* [Arin90], *TTP/C* [KoGr94, Kope97b, TTTe99b] und *Byteflight* [Byte, PBG99, BPG00] entwickelt. Daneben existieren weitere Ansätze [ISO99, FMD+01, HMFH01], das bestehende ereignisgesteuerte *CAN* Kommunikationssystem [Bosc91] um eine Zeitsteuerung zu erweitern. Das Kommunikationssystem FlexRay [BBB+00, BES+01, Flex] stellt

die jüngste Entwicklung dar. Es wurde von den Automobilherstellern DaimlerChrysler und BMW auf die Anforderungen der By-Wire-Systeme zugeschnitten und hat gute Aussichten zukünftig bei By-Wire-Systemen eingesetzt zu werden [Hans01].

Das FlexRay Protokoll erfüllt sowohl die Anforderung nach Vorhersehbarkeit als auch Flexibilität durch die Kombination einer skalierbaren statischen und dynamischen Nachrichtenübertragung. Es stellt somit eine Kombination der Vorteile der bestehenden Protokolle TTP/C und Byteflight dar. Das Protokoll unterstützt fehlertolerante Uhrensynchronisation über eine globale Uhrzeit, kollisions- und arbitrierungsfreien Zugriff auf die Kommunikationskanäle, garantierte Nachrichten-Latenzzeit, nachrichtenorientierte Adressierung über Nachrichten-Identifier und skalierbare Fehlertoleranz-Eigenschaften durch einfache oder redundante Kommunikationskanäle. Es sind optische oder elektrische Übertragungsmedien möglich, sowohl in einer Bus- als auch in einer Stern-Topologie.

Die Kommunikation erfolgt im Rahmen eines Kommunikationszyklus, der aus einem statischen und einem dynamischen Teil besteht (siehe Abbildung 3.1). Die Grenze zwischen dem statischen und dem dynamischen Teil kann flexibel vor der Laufzeit festgelegt werden. Ferner sind sowohl dynamische als auch rein statische Konfigurationen möglich [Flex00].

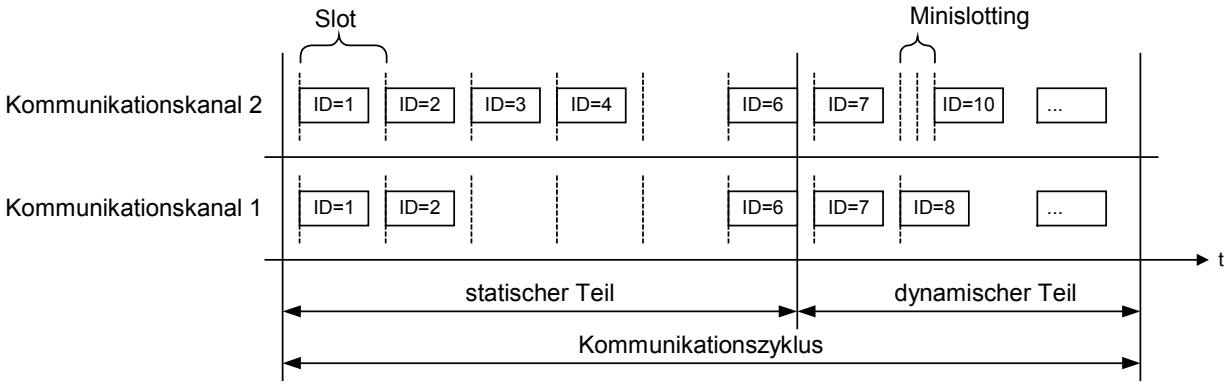


Abbildung 3.1: Statischer und dynamischer Teil eines Kommunikationszyklus von FlexRay

Statischer Teil: Der statische Teil enthält eine frei konfigurierbare Anzahl von Sendezeitschlitzen sog. *Slots*. Der Buszugriff erfolgt nach dem Time Division Multiple Access (TDMA) Verfahren. Dieses Verfahren wird auch bei TTP/C eingesetzt. Im Unterschied zu TTP/C ist bei FlexRay die Datenübermittlung nachrichtenorientiert. Ein Knoten kann eine oder mehrere Nachrichten pro Runde versenden. Wird ein Slot nicht genutzt, bleibt der Slot leer. Somit sind im statischen Teil konstante Latenzzeit und Jitter garantiert. Der Zugriff auf die Kommunikationsmedien wird durch einen unabhängigen Baustein, den sog. *Bus Guardian*, überwacht. Er lässt den Zugriff nur zu den zuvor definierten Slots zu. Auf diese Weise kann eine Monopolisierung durch einen fehlerhaften Knoten, ein sog. *Babbling Idiot*, verhindert werden.

Dynamischer Teil: Im dynamischen Teil erfolgt die Übertragung gemäß der Byteflight Spezifikation [Byte] nach einem Flexible Time Division Multiple Access (FTDMA) Buszugriffsverfahren, auch *Minislotting* genannt. Liegt keine Sendeanforderung eines Knotens vor, kann nach einer kurzen Wartezeit die nächste Nachricht gesendet werden. Auch hier erfolgt der Zugriff auf das Kommunikationsmedium zeitgesteuert und somit kollisionsfrei. Latenzzeiten und Jitter der Nachrichten können jedoch nicht mehr garantiert werden. Unter Umständen kann sich das Senden der Nachrichten um mehrere Runden verzögern. Der dynamische Teil kann somit nicht für die Übertragung von Signalen verteilter regelungstechnischer Anwendungen verwendet werden.

3.1.2.2 Zeitgesteuerter Betriebssystem-Standard OSEKtime

Die Arbeitsgruppe OSEKtime[3] der *OSEK/VDX*[4] Initiative definiert derzeit einen Standard eines zeitgesteuerten Betriebssystems für By-Wire-Anwendungen im Automobil [PGT+00, KDH+00, HRK+00]. OSEKtime ist auf die Anforderungen im Automobil zugeschnitten und berücksichtigt sowohl die Anforderung nach zeitlicher Vorhersehbarkeit als auch die Anforderung nach Flexibilität, wobei die erste Anforderung stets Vorrang hat. Die Aktivierung der Tasks erfolgt statisch. Nicht vorhersehbare dynamische Konzepte, wie dynamisches Aktivieren von Tasks, sind nicht gestattet. Die Reaktion auf dynamische Ereignisse erfolgt nur über Interrupt Service Routinen (ISR). Die Verwendung von Interrupts ist jedoch stark eingeschränkt. Sie ist nur erlaubt, sofern gewisse Zeitdauern (MINT – *Minimum Inter-Arrival Time*) garantiert werden können, zu denen mit Sicherheit keine weiteren Interrupts eintreffen können.

Nachfolgend werden die für die vorliegende Arbeit relevanten Konzepte von OSEKtime vorgestellt und diskutiert.

Prioritätsebenen

Die Prioritätsebenen des Betriebssystems sind in Abbildung 3.2 dargestellt. Das Betriebssystem ist oberhalb der Tasks und Interrupts angeordnet und kann stets Kontrolle über sie ausüben. Die Prioritäten von Interrupts und Tasks können sich überschneiden. Oberhalb des Betriebssystems sind nichtmaskierbare Interrupts angeordnet, die für die Mitteilung von bedrohlichen Fehlerzuständen reserviert sind. Unterhalb des OSEKtime Teils kann ein konventionelles OSEK/VDX-Betriebssystem angeordnet sein, das dynamische Konzepte aufweist [OSEK00a]. Nicht benötigte Rechenzeit kann so von dem OSEK/VDX-Teil für die Bearbeitung dynamischer Vorgänge genutzt werden. Durch die höhere Priorität des OSEKtime-Teils bleibt der Determinismus der OSEKtime-Aktivitäten stets gewahrt. Alternativ zu dem OSEK/VDX-Teil kann auch eine einfache *Idle Task* stehen.

---

[3] Die Gruppe wurde 1998 von der DaimlerChrysler Forschung ins Leben gerufen. Gründungsmitglieder dieser Gruppe waren: DaimlerChrysler, Bosch, BMW, Continental Teves, Siemens, Motorola, TTTech und das IAS.

[4] OSEK: Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
VDX: Vehicle Distributed Executive

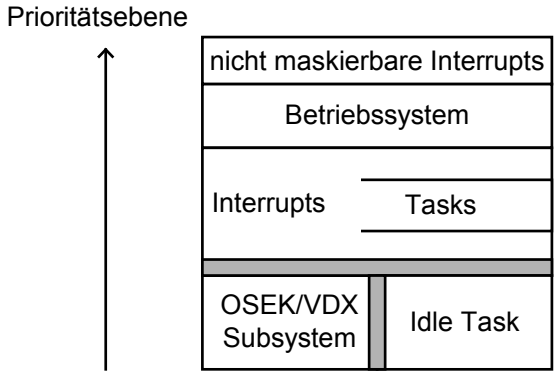


Abbildung 3.2: Prioritätsebenen bei OSEKtime

Scheduling-Strategie

Die OSEKtime-Scheduling-Strategie basiert auf einem *statischen* Scheduling [Kope97a], das vor der Laufzeit des Systems durchgeführt wird. Die Aktivierungszeitpunkte der Tasks werden in einem Task Schedule festgeschrieben. Zur Laufzeit erfolgt ein statisches Aktivieren der Tasks, durch den sog. *Dispatcher* (Teil des Betriebssystems), zu den zuvor geplanten Zeitpunkten (siehe Abbildung 3.3 linke Seite). Die Aktivierung der Tasks erfolgt bezüglich der lokalen Uhrzeit des Knotens. Die lokale Uhrzeit wird fortwährend mit der globalen Uhrzeit, die vom Kommunikationssystem zur Verfügung gestellt wird, synchronisiert. Somit können Tasks auf verschiedenen Knoten bezüglich derselben globalen Uhrzeit aktiviert werden.

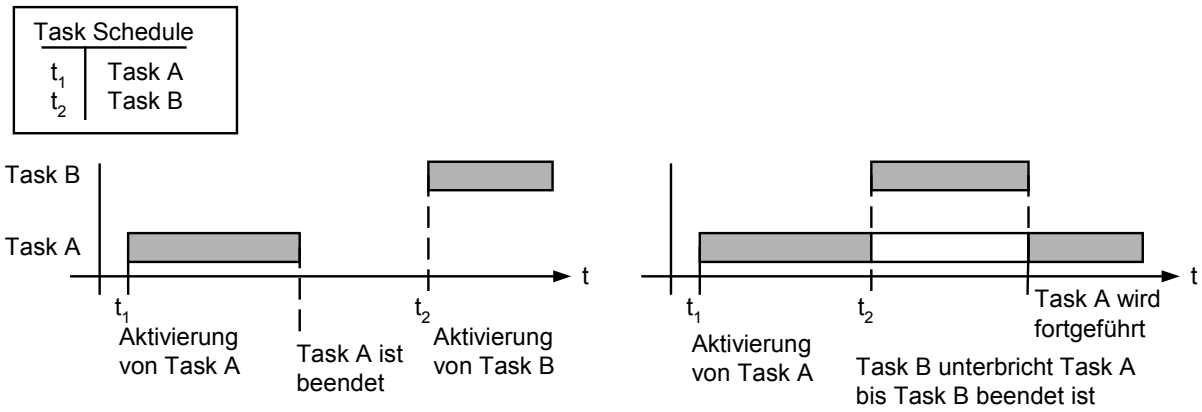


Abbildung 3.3: Stackbased-Scheduling-Strategie bei OSEKtime

Eine bessere Ausnutzung der Rechenzeit wird durch die so genannte *Stackbased-Scheduling-Strategie* erreicht. Der Dispatcher aktiviert eine Task beim Eintreffen ihres Aktivierungszeitpunkts ungeachtet dessen, ob zu diesem Zeitpunkt bereits eine andere Task ausgeführt wird (siehe Abbildung 3.3 rechte Seite). Wird eine Task B zum Zeitpunkt t_2 aktiviert während der Ausführung von Task A, wird die Task A unterbrochen und bleibt solange unterbrochen bis die Task B beendet ist. Beim Stackbased-Scheduling-Verfahren wird somit garantiert, dass die Aus-

führung einer Task zu dem eingeplanten Startzeitpunkt auch wirklich begonnen wird, vorausgesetzt, es trifft nicht zeitgleich ein höherpriorer Interrupt ein.

Stackbased Scheduling ermöglicht beispielsweise, dass eine Task mit einer langen Ausführungszeit durch weitere Tasks mit kurzen Ausführungszeiten unterbrochen wird. Andernfalls müsste die Task mit der langen Ausführungszeit in viele kleine Teile aufgeteilt werden, wobei zwangsläufig Rechenzeit ungenutzt bleibt. Zur Reduzierung der Dispatcher-Aktivitäten können mehrere, unmittelbar nacheinander folgende Tasks zu einer sog. Task Chain zusammengefasst werden.

Mechanismen zum Ressourcen-Mangement während der Laufzeit sind bei OSEKtime nicht gestattet. Aufgrund der Zeitsteuerung kann ein eventueller gemeinsamer Zugriff auf eine Ressource bereits vor der Laufzeit erkannt werden. So kann beispielsweise ein gemeinsamer Zugriff von Tasks auf Daten vor der Laufzeit analysiert werden. Gegebenenfalls müssen lokale Kopien von Daten verwendet werden um Dateninkonsistenzen zu vermeiden.

Deadline-Überwachung
Das OSEKtime Betriebssystem besitzt einen Fehlererkennungs-Mechanismus (*Deadline-Überwachung*), der das Überschreiten der Deadline einer Task erkennt. Um ein deterministisches Systemverhalten garantieren zu können, muss der Task Schedule so erstellt werden, dass jede Task ihre Deadline einhalten kann, inklusive aller möglichen Unterbrechungen durch andere Tasks oder Interrupts. Überschreitet eine Task dennoch ihre Deadline, muss zwangsläufig ein ernsthafter Fehler aufgetreten sein, der eine Fehlerbehandlung erfordert. In diesem Fall wird die Bearbeitung der Task durch das Betriebssystem gestoppt.

Planung des Task Schedules
Die OSEKtime Konzepte Offline Scheduling, Stackbased-Scheduling-Strategie, Interrupt-Verarbeitung und Deadline-Überwachung setzen die genaue Kenntnis der WCET bei der Planung des Task Schedules voraus. Wird die WCET einer Task als zu klein angenommen und folglich die Deadline als zu klein bestimmt, kann dies zu einer ungewollten Deadline-Verletzung führen. Eine zu klein bestimmte WCET kann bei der Stackbased-Scheduling-Strategie zu ungewollten Verzögerungen von Tasks führen. Wird beispielsweise von einer zu klein bestimmten WCET einer Task A ausgegangen und im Anschluss an die Task A eine Task B eingeplant, kann die Task B die Task A ungewollt unterbrechen. Die Verzögerung der Task A kann wiederum eine Deadline-Verletzung der Task A nach sich ziehen. Ist die zweite Task von den Ergebnissen der ersten Task abhängig, kann dies zu ernsthaften Beeinträchtigungen führen, da die Ergebnisse nicht vorliegen. Für die Bestimmung des Offline Schedules ist folglich eine obere Schranke der WCET erforderlich. Wird allerdings die WCET zu groß angenommen, wird unnötigerweise Rechenzeit vergeudet; denn die zu große WCET muss wiederum bei allen unterbrochenen Tasks berücksichtigt werden. Die Konzepte erfordern folglich enge obere Schranken der realen WCET von Programmen zur Planung der Task Schedules.

3.1.3 Software-Entwicklungswerkzeuge in der Automobilindustrie

Softwaresysteme im Automobil sind komplexe verteilte Systeme, zu deren Entwicklung spezialisierte Entwicklungswerkzeuge eingesetzt werden. Zur Entwicklung der Steuergerätesoftware werden Software-Entwicklungswerkzeuge wie ASCET-SD [ETAS00a] der Firma ETAS [ETAS] oder MATLAB (MATrix LABoratory) [Math98a] der Firma MathWorks [Math] eingesetzt, zur Konfiguration der Kommunikation zwischen Steuergeräten finden Werkzeuge wie CANdb++ der Firma Vector [Vect] ihren Einsatz.

Das Software-Entwicklungswerkzeug ASCET-SD der Firma ETAS ist auf die Entwicklung von Steuergerätesoftware im Automobil zugeschnitten. ASCET-SD bietet eine durchgehende Werkzeugunterstützung in allen Phasen der Entwicklung herkömmlicher Steuergerätesoftware, von der Modellierung über Prototyping und Implementierung bis hin zur Feinabstimmung. Es ermöglicht die Spezifikation funktionaler Modelle unabhängig vom Ziel-Steuergerät. Mit Hilfe des integrierten Codegenerierungswerkzeuges kann C-Code für verschiedene Serien-Steuergeräte generiert werden. Mit ASCET-SD kann das Betriebssystem ERCOS^{EK} [ETAS00b] konfiguriert werden, in dessen Laufzeitumgebung der generierte Code ausgeführt wird. Den Tasks können bestimmte Prioritäten sowie Aktivierungseigenschaften (periodisch, ereignisgesteuert) zugeordnet werden.

Die Werkzeugumgebung MATLAB (MATrix LABoratory) [Math] ist ein umfangreiches Software-Entwicklungswerkzeug zur Entwicklung dynamischer und ereignisdiskreter Systeme. Die Stärken von MATLAB liegen in der Unterstützung der Entwicklung von Algorithmen für die Regelungstechnik und Signalverarbeitung. Es wird in der Automobilindustrie zum Entwurf und zur Simulation von dynamischen Systemen eingesetzt. MATLAB ermöglicht auch die Integration von Werkzeugen fremder Hersteller wie beispielsweise dSpace [dSpa]. Die Werkzeuge von dSpace unterstützen die Entwicklung von Anwendungssoftware in ähnlicher Weise wie ASCET-SD, von der Modellierung in MATLAB über Prototyping und Implementierung bis hin zur Feinabstimmung.

Das Werkzeug CANdb++ der Firma Vector [Vect] ermöglicht die Definition und Verwaltung von Kommunikationsdaten für das ereignisgesteuerte Kommunikationssystem CAN [Bosc91] sowie für weitere Kommunikationssysteme, die im Automobil Einsatz finden, wie LIN [LIN99] oder MOST [MOST]. In CANdb++ werden Nachrichten und die darin übertragenen Signale exakt beschrieben. Das Datenmodell von CANdb++ sieht die Kommunikationsobjekte Nachrichten, Signale, Knoten sowie Beziehungen zwischen den Objekten wie z. B. Sendeknoten von Nachrichten, oder Empfangssignale von Knoten vor. Es werden verschiedene Sichten (z. B. Netzwerk, Steuergerät, Signal) zur Darstellung und Bearbeitung der Daten unterstützt. Eine Sicht zur Planung des zeitlichen Ablaufs von Aktionen im verteilten System ist jedoch nicht vorgesehen.

Bewertung
Die Software-Entwicklungswerkzeuge MATLAB und ASCET-SD sind bewährte Werkzeuge, die sehr gut auf die Entwicklung von Anwendungssoftware zugeschnitten sind. Über die Entwicklung der Anwendungssoftware hinaus unterstützen sie jedoch die Entwicklung und Analyse zeitgesteuerter Systeme nicht. So ist weder eine statische noch eine dynamische WCET-Analyse der erstellten Anwendungssoftware möglich. Eine Schedulability-Analyse zum Nachweis, ob eine Menge erstellter Tasks ausführbar ist, wird ebenfalls nicht unterstützt. Tasks können für jeweils nur einen Knoten eingeplant werden; eine Planung zeitlicher Abläufe im verteilten System ist nicht möglich.

Das Werkzeug CANdb++ bietet keine Unterstützung zur Planung des zeitlichen Ablaufs der Kommunikation im verteilten System. Die direkte Spezifikation zeitlicher Abläufe im verteilten System, die zur Planung zeitgesteuerter Systeme notwendig ist, ist somit mit heute etablierten Werkzeugen nicht möglich. Für die bisher eingesetzten ereignisgesteuerten Kommunikationssysteme, die lediglich die Realisierung lose gekoppelter Systeme zulassen, war dies nicht notwendig. Zur Unterstützung der Entwicklung zeitgesteuerter Systeme wird deshalb ein übergeordnetes Werkzeug benötigt, das sowohl die Planung zeitlicher Abläufe von Tasks als auch von Nachrichten zusammen ermöglicht.

3.1.4 Schnittstellen-fokussierte Methodik

Der Grundgedanke der Schnittstellen-fokussierten Methodik [KoNo95, Noss97, KFMN99] ist, den Nachrichten-Schedule, der eine wohldefinierte Schnittstelle im Werte- und Zeitbereich zwischen den Knoten darstellt, als Entwicklungsschnittstelle zwischen dem Automobilhersteller und den Zulieferern zu nutzen. Die Methodik unterscheidet zwischen zwei Schritten, dem *globalen Entwurf* und dem *lokalen Entwurf*.

Das Kommunikationssystem verbindet alle Knoten miteinander. Die zeitlichen Eigenschaften des Kommunikationssystems haben somit globale Auswirkungen auf alle Knoten. Die Aktivitäten rund um die Definition des Nachrichten-Schedules sind dem globalen Entwurf zugeordnet, der zunächst vom Automobilhersteller durchzuführen ist, bevor die Zulieferer mit dem lokalen Entwurf beginnen können. Im lokalen Entwurf werden die Anwendungen auf den Knoten und die Task Schedules entwickelt, unter Beachtung der Schnittstelle Nachrichten-Schedule.

Der Schwerpunkt der Arbeiten an der TU Wien liegt auf der automatischen zweistufigen Schedule-Generierung. In einer ersten Stufe wird das Nachrichten-Scheduling durchgeführt, in einer zweiten das Task Scheduling. Das Scheduling von zeitgesteuerten Systemen ist im Allg. NP-

vollständig[5] [NoGa97]. Zum Scheduling wurden deshalb genetische Algorithmen[6] eingesetzt [KoNo95, Noss97, NoGa97].

Bewertung
Mit der Schnittstellen-fokussierten Methodik wurden die theoretischen Grundlagen für die Entwicklung zeitgesteuerter Systeme geschaffen. Die Verwendung des Nachrichten-Schedules als Entwicklungsschnittstelle und Trennung in globalen und lokalen Entwurf ist sinnvoll. Diese Trennung spiegelt die organisatorischen Gegebenheiten in der Entwicklung von Automobilen wider und unterstützt sie.

Die Methodik zielt vornehmlich auf die Generierung von Schedules ab. Es wurde nicht ausreichend beachtet, dass die Entwicklung von Automobilen in einem sich über mehrere Jahre erstreckenden iterativen Prozess erfolgt. Die Methodik sieht jedoch keine Iterationen vor. So wird die WCET von Tasks schon zu Beginn des Entwurfs als gegeben angesehen, obwohl zum Zeitpunkt des Entwurfs noch keine Implementierungen vorhanden sind.

Weitere wichtige Aspekte bei der Entwicklung zeitgesteuerter Anwendungen, insbesondere die Berücksichtigung der Auswirkungen der Zeiteigenschaften des zeitgesteuerten Systems auf den Entwurf verteilter regelungstechnischer Anwendungen, werden nicht berücksichtigt. Die automatische Schedule-Generierung erfordert zunächst eine aufwändige Spezifikation aller Eingangsdaten durch den Entwickler des zeitgesteuerten Systems. Die genetischen Algorithmen führen zu einer möglichen, jedoch nicht zwangsweise zu der für die regelungstechnische Anwendung optimalen Lösung.

3.1.5 System-Entwicklungsumgebung SETTA

In dem TTA Projekt [TTA] wurde eine generische zeitgesteuerte Architektur für fehlertolerante verteilte Echtzeitsysteme definiert.[7] Ein Teilprojekt war die Definition einer verständlichen, benutzerfreundlichen Entwicklungsumgebung [SHS+97, TTA98] für den Entwurf zeitgesteuerter Systeme, die Entwicklung der Anwendungssoftware und einer Sicherheitsanalyse. Im Rahmen des Projekts wurde das „System Engineering Environment for Time-Triggered Architectures – SETTA" konzipiert und prototypisch umgesetzt [SFB98].

Die Entwicklungsumgebung SETTA unterstützt den grafischen Entwurf zeitgesteuerter Systeme. Die Methode basiert im Wesentlichen auf der Schnittstellen-fokussierten Methodik mit der Trennung in globalen und lokalen Entwurf. Die Tätigkeiten während der zwei Entwurfs

---

[5] NP vollständig bedeutet: Nicht polynominal in endlicher Zeit lösbar.
[6] Algorithmen, die Strategien aus der Evolutionstheorie verwenden, um zu einem Problem eine möglichst gute Lösung zu finden.
[7] Aus dem TTA Projekt ist eine erste Prototypimplementierung des TTP/C Kommunikationssystems entstanden.

schritte wurden präzisiert. Neu vorgesehen wurde beispielsweise das manuelle Editieren von automatisch erzeugten Schedules, da die Ergebnisse des automatischen Schedulings keine zufriedenstellenden Ergebnisse lieferten.

Die Entwicklungsumgebung SETTA stellt eine Reihe von unterschiedlichen Sichten auf das zeitgesteuerte System bereit. In einer logischen Topologie können Beziehungen zwischen Subsystemen ohne Redundanzen festgelegt werden. Anschließend wird die physikalische Topologie festgelegt, indem die Knoten im System definiert und die Subsysteme bestimmten Knoten zugeordnet werden. In einem nächsten Schritt werden die Kommunikationsabhängigkeiten zwischen den Subsystemen definiert. Anschließend wird die Task-Topologie festgelegt. Weitere Sichten sind Nachrichten-Scheduling und Task Scheduling. Die Umsetzung erfolgt auf Basis des bestehenden Werkzeugs *Trapper*, das ursprünglich für den Entwurf von Parallelrechnern entwickelt wurde. Bis zur Anwendbarkeit hin wurde nur das Werkzeug zur Generierung des Nachrichten-Schedules realisiert. Die Definition des Nachrichten-Schedules erfolgte manuell mittels Eintragungen in eine Datenbank.

Bewertung
Die Entwicklungsumgebung SETTA sollte neben der Entwicklung des zeitgesteuerten Systems die Entwicklung der Anwendungssoftware unterstützen. Zu diesem Zweck wurde die Unterstützung der Entwicklung von fehlertoleranten Anwendungen durch eine Fehlerbaum-Analyse vorgesehen. Eine Unterstützung der Entwicklung verteilter regelungstechnischer Anwendungen wurde jedoch nicht berücksichtigt. Die WCET-Analyse wurde lediglich konzeptionell vorgesehen. Eine WCET-Analyse von modellbasiert erstellter Software wurde nicht angedacht.

Die Entwicklungsumgebung SETTA bietet eine Vielzahl grafischer Sichten zur schrittweisen Spezifikation des zeitgesteuerten Systems. Die grafische Spezifikation gestaltet sich jedoch sehr aufwändig und langwierig. Trotz der vielen grafischen Sichten fehlt eine entscheidende Sicht zur Spezifikation und Darstellung von zeitlichen Abläufen im verteilten System, die eine direkte Spezifikation der konkreten Vorstellungen des Entwicklers ermöglicht.

3.1.6 Werkzeugumgebungen für die Konfiguration zeitgesteuerter Kommunikationssysteme

Die Firmen TTTech [TTTe] und DeComSys [DCS] bieten erste kommerzielle Werkzeugumgebungen für die Konfiguration zeitgesteuerter Kommunikationssysteme an. Das TTP/C Software Development Environment der Firma TTTech ermöglicht den Entwurf von TTP/C-Systemen. Mit der Werkzeugumgebung wird die zweigeteilte Methodik mit globalem und lokalem Entwurf verfolgt. Es werden dazu zwei getrennte Werkzeuge, *TTPplan* und *TTPbuilt*, [TTTe99a] angewandt.

Auf Basis eines Objektmodells werden die Eigenschaften der Objekte des zeitgesteuerten Systems und Relationen zwischen den Objekten[8] definiert. Ein automatischer Scheduler erzeugt aus dieser Spezifikation einen Nachrichten-Schedule, der anschließend manuell verändert werden kann. Als Ausgangspunkte des globalen Entwurfs gelten Signale, die von Knoten ausgetauscht werden, und deren Zykluszeiten. In der Werkzeugumgebung von DeComSys wird der Entwurf des zeitgesteuerten Systems mit dem Werkzeug *xDesigner* [DCS00] vorgenommen. Es kann sowohl ein Nachrichten-Schedule für FlexRay als auch für TTP/C-Systeme entwickelt werden. Im xDesigner wird direkt die Spezifikation von Nachrichten und den Signalen, die sie übertragen, vorgenommen.

Bewertung
Vorrangiges Ziel bei der Entwicklung beider verfügbarer Werkzeugumgebungen war es, schnellstmöglich Werkzeuge zu schaffen, mit deren Hilfe die Kommunikationssysteme in Betrieb genommen werden können. Im Gegensatz zur Werkzeugumgebung SETTA bieten sie nur Minimalfunktionalitäten.

Beide Werkzeugumgebungen bieten keine Unterstützung bei der Planung zeitlicher Abläufe im verteilten System und bei der Entwicklung der zeitgesteuerten Anwendungssoftware. Die Planung muss der Benutzer zuvor selbst auf dem Papier ohne Werkzeug vornehmen und die Ergebnisse dieser Überlegungen ins Werkzeug eingeben.

Bei dem Werkzeug TTPplan ist die Spezifikation des Objektmodells ein sehr langwieriger Vorgang, der dann erst das automatische Erstellen des Nachrichten-Schedules ermöglicht. Im Gegensatz hierzu ist bei dem Werkzeug xDesigner keine umständliche Spezifikation von Objekteigenschaften notwendig. Die Erstellung eines Nachrichten-Schedules erfolgt sehr schnell.

Die automatische Schedule-Generierung bei TTPplan hat sich als wenig hilfreich herausgestellt. Der Automatismus führt überwiegend zu suboptimalen Nachrichten-Schedules, die nicht der Vorstellung des Benutzers entsprechen. Die Umgehung des Automatismus durch Variation der Eingangsparameter gestaltet sich schwierig.

3.2 Stand der Technik auf dem Gebiet der WCET-Analyse

3.2.1 Übersicht über die Ansätze der WCET-Analyse

Die statische Worst-Case-Execution-Time-Analyse hat sich in den letzten zehn bis fünfzehn Jahren zu einem weiten Forschungsgebiet entwickelt. Forschungsarbeiten reichen von der Pfadmodellierung [Shaw89, PuKo89, Mok89, Park92] bis zu kombinierten Ansätzen der Pfad

[8] Objekte sind beispielsweise Knoten, Signale, Nachrichten etc. Eine Relation gibt beispielsweise an, welcher Knoten welches Signal sendet.

modellierung und Modellierung komplexer Mikroarchitekturen [StaAl97, LiMa99, FeWi99]. In den nachfolgenden Abschnitten werden für die vorliegende Arbeit relevante Ansätze in weitgehend chronologischer Reihenfolge vorgestellt. Für die modellbasierte WCET-Analyse ist insbesondere die Modellierung der Pfade des generierten Codes von Bedeutung. Die Ansätze werden deshalb aus Blickpunkt der Pfadmodellierung betrachtet und abschließend bewertet. Die Ansätze lassen sich in zwei Gruppierungen aufteilen:

- Ansätze der Pfadmodellierung mit Modellierung einfacher Mikroarchitekturen
- Ansätze der Pfadmodellierung mit Modellierung komplexer Mikroarchitekturen

3.2.2 Pfadmodellierung mit Modellierung einfacher Mikroarchitekturen

Nachfolgend werden verschiedene Ansätze der Pfadmodellierung mit Modellierung einfacher Mikroarchitekturen bewertet.

Allan C. Shaw

Von Allan C. Shaw [Shaw89] stammt (nach seinen Angaben) die Idee, die WCET von Programmen zu ermitteln, die in einer höheren Programmiersprache geschrieben sind. Shaw entwickelte den sog. *Timing-Schema-Ansatz*. Für jedes Sprachkonstrukt wird ein formales Zeitschema definiert, nach dem die Zeitanalyse erfolgt. Die WCET wird in einer Bottom-Up-Vorgehensweise bestimmt: Für eine *if-else*-Anweisung wird die größere WCET der Anweisungen des *if*- oder des *else*-Zweiges verwendet. Für eine Schleife wird die WCET des Schleifenrumpfes bestimmt und mit der maximalen Anzahl Schleifeniterationen multipliziert. Chang Yun Park und Allan C. Shaw entwickelten dazu ein WCET-Analyse-Werkzeug für eine Untermenge der Programmiersprache C [KeRi88] und für eine einfache Hardwarearchitektur (Motorola MC68010). Die Programmiersprache enthält keine Rekursionen, keine dynamischen Konstrukte und es sind nur abgeschrankte Schleifen zugelassen. Die Informationen über die maximale Anzahl von Schleifeniterationen muss der Benutzer des Werkzeuges interaktiv angeben. Zurückgegeben werden die Ergebnisse der WCET-Analyse auf Quellcode-Ebene.

Peter Puschner und Christian Koza

Peter Puschner und Christian Koza [PuKo89] erarbeiteten ebenfalls ein Konzept für die WCET-Analyse von Programmen einer höheren Programmiersprache. Ergänzend zu dem Ansatz von Shaw berücksichtigten sie die Möglichkeit, Ausführungshäufigkeiten von bestimmten Anweisungen anzugeben. Sie erweiterten die Programmiersprache *MARS-C*[9] um zusätzliche Sprachkonstrukte *Markers, Scopes* und *Loop Sequences*. Durch diese Sprachkonstrukte kann der Pro-

[9] *MARS-C* ist eine Submenge der Programmiersprache C, die im Rahmen des MARS-Projekts an der TU Wien entwickelt wurde.

grammierer Informationen über das tatsächliche Verhalten der Algorithmen angeben, die nicht durch Standardsprachkonstrukte ausgedrückt werden können [Blie00]. Somit können beispielsweise Ausführungshäufigkeiten bei geschachtelten Schleifen angegeben werden, bei denen die Schleifengrenzen der inneren Schleife vom Schleifenindex der äußeren abhängt.[10]

Aloysius K. Mok

Der Ansatz von Aloysius K. Mok [Mok89] sieht ebenfalls die Angabe von Ausführungshäufigkeiten von Instruktionen wie bei dem Ansatz von Puschner und Koza [PuKo89] vor. Im Gegensatz dazu wird die Ausführungshäufigkeit nicht auf Quellcode-Ebene durch zusätzliche Sprachkonstrukte angegeben, sondern auf Assemblercode-Ebene. Ein Compiler erzeugt aus dem Assemblercode C-ähnliche Skripte für die Angabe von Ausführungshäufigkeiten durch den Programmierer. Da der Programmierer den Programmfluss des Assemblercodes verstehen muss, um korrekte Angaben vornehmen zu können, ist der Ansatz bei weitem nicht so benutzerfreundlich wie der von Puschner und Koza.

Chang Yun Park

Chang Yun Park [Park92] berücksichtigte, dass bei langen Programmen häufig bestimmte Teile des Programms voneinander abhängig sind, d. h. dass nicht alle statisch möglichen Pfade auch ausführbar sind. Diese Abhängigkeiten können nicht durch Ausführungshäufigkeiten ausgedrückt werden. Park löste das Problem, indem er dem Programmierer ermöglicht, stets gemeinsame ausgeführte bzw. sich gegenseitig ausschließende Programmzweige über die Skriptsprache IDL (Information Description Language)[11] anzugeben. Dadurch kann die Genauigkeit der berechneten WCET verbessert werden. Im Gegensatz zu den anderen Ansätzen wurde die Strukturanalyse auf Quellcodeebene durchgeführt. Dazu wurde der von dem GNU-Compiler generierte Assemblercode vorhergesagt. Code-Optimierungen, die den Code umorganisieren, waren deshalb nicht erlaubt [LiMa99].

Andreas Ermedahl und Jan Gustafsson

Andreas Ermedahl und Jan Gustafsson [ErGu97] stellten eine Methode vor, die es ermöglicht, automatisch Schleifenschranken und Informationen über nicht ausführbare Pfade aus dem Code mit Hilfe der Datenflussanalyse zu gewinnen. Die Datenflussanalyse stammt aus dem Bereich Compilerbau und wird für Code-Optimierungen eingesetzt. Sie berechnet die Definition und Verwendung von Variablen [KoPl96]. In Verbindung mit der WCET-Analyse wird sie dazu genutzt, Werte von am Kontrollfluss beteiligten Variablen an bestimmten Stellen im Programm zu bestimmen bzw. einzuschränken. Die Ausgangsbasis bilden zum Einen die zur Zeit des Compilierens bekannten Werte von Variablen, zum Anderen die Relationen zwischen Variablen. Auf

---

[10] Eine andere Einsatzmöglichkeit war, die Ausführungshäufigkeit von bedingten Anweisungen innerhalb von Schleifen angeben zu können.

[11] Nicht zu verwechseln mit der Interface Description Language für verteilte Anwendungen der Object Management Group (OMG).

diese Weise können nicht ausführbare Pfade erkannt werden. Der Umfang der Programmiersprache ist jedoch sehr stark eingeschränkt. Es sind keine Gleitkommaarithmethik, keine Felder und Strukturen, keine Funktionsaufrufe, keine dynamische Speicherallokierung und keine Zeiger zugelassen.

3.2.3 Pfadmodellierung mit Modellierung komplexer Mikroarchitekturen

Nachfolgend werden verschiedene Ansätze der Pfadmodellierung mit Modellierung komplexer Mikroarchitekturen bewertet.

Yau-Tsun Steven Li und Sharad Malik
Yau-Tsun Steven Li und Sharad Malik [LiMa95, LiMa99] vereinigten die zuvor weitgehend getrennt betrachteten Gebiete Mikroarchitekturmodellierung und Pfadmodellierung. Sie entwickelten eine effiziente Methode zur Beschreibung der Programmpfade, die auch im Falle komplexer Mikroarchitektur geeignet ist. Diese Methode wurde später auch von anderen Forschern aufgegriffen [PuSc97].

Sie identifizierten als Hauptproblem der Pfad-Modellierung die exponentiell ansteigende Anzahl der Pfade bei zunehmender Codegröße. Sie entwickelten die sog. *Implicit Path Enumeration (IPE)* Methode für die implizite Beschreibung der Programmpfade anstatt der expliziten Aufzählung aller möglichen Pfade. Die implizite Beschreibung wird als lineares Optimierungsproblem formuliert. Die Ausführungszeit eines Codestücks wird beschrieben durch die Zielfunktion T, die die Summe der Ausführungszeiten der Grundblöcke multipliziert mit deren Ausführungshäufigkeiten darstellt:

$$T = \max \sum_{i=1}^{N} c_i * x_i$$

Mit:
T = Ausführungszeit des Programms (Zielfunktion)
c_i = Ausführungszeit eines Grundblocks
x_i = Ausführungshäufigkeit eines Grundblocks
N = Anzahl der Grundblöcke

Unter Verwendung von algebraischen und/oder logischen Einschränkungen werden ausführbare Pfade implizit durch die Ausführungshäufigkeit der Grundblöcke und Abhängigkeiten zwischen Zweigen beschrieben. Dies sind einerseits strukturelle Einschränkungen, die sich aus der statischen Struktur des Programms ergeben, anderseits funktionale Einschränkungen.

Die statische Struktur wird durch die Ausführungshäufigkeit der Grundblöcke und die Ausführungshäufigkeit der Zweige (Zweigvariablen d_j) beschrieben. Die Ausführungshäufigkeit der

Grundblöcke ist gleich der Summe der Ausführungshäufigkeit der hineinfließenden Zweige, die wiederum gleich der Summe der Ausführungshäufigkeit der hinausfließenden Zweige ist (Knotenregel). In Abbildung 3.4 sind die strukturellen Einschränkungen beispielhaft an einer einfachen Funktion aufgestellt. Weitere Einschränkungen ergeben sich daraus, dass bei einem Funktionsaufruf die Zweige d_1 und d_8 genau einmal durchlaufen werden.

Schließen sich beispielsweise die Grundblöcke 2 und 4 gegenseitig aus, dann kann dies durch funktionale Einschränkungen angegeben werden. Schleifenschranken werden angegeben, indem der Zweigvariablen die maximale Iterationszahl zugewiesen wird.

Die Berechnung der WCET erfolgt durch die Maximierung der Zielfunktion T, unter Berücksichtigung aller Einschränkungen. Der Ansatz wurde in dem Werkzeug *Cinderella* [LiMa99] umgesetzt. Die funktionalen Einschränkungen werden vom Benutzer dieses Werkzeuges interaktiv eingegeben.

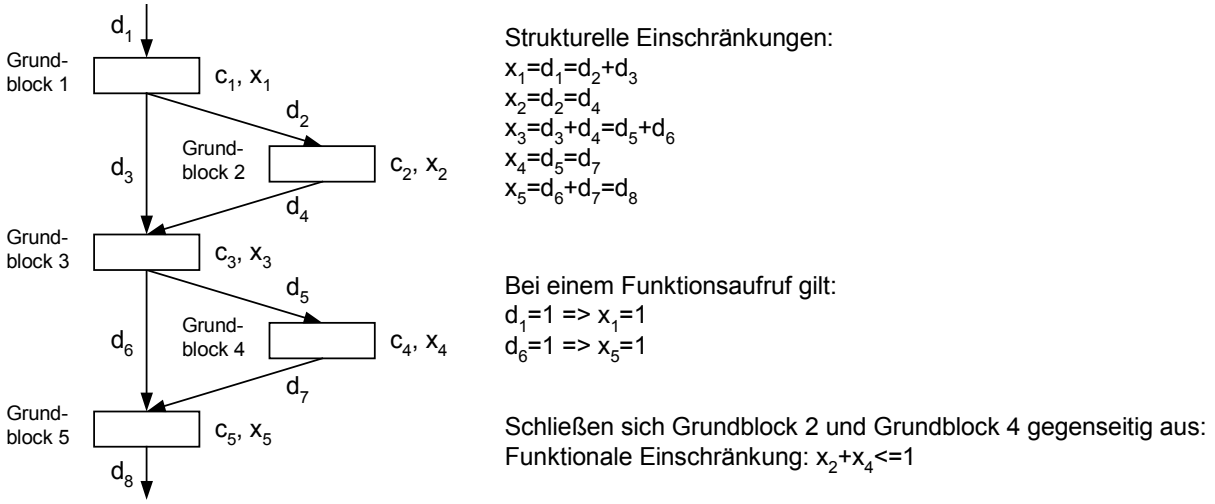


Abbildung 3.4: Beispiel für die implizite Beschreibung von Programmpfaden

Friedhelm Stappert und Peter Altenbernd

Friedhelm Stappert und Peter Altenbernd stellten ein vollständiges WCET-Analyse-Verfahren inklusive Instruktions- und Daten-Cache-Modellierung vor [StaAl97]. Der Schwerpunkt dieses Ansatzes liegt auf der Vollständigkeit des Verfahrens und der Verbindung von Mikroarchitekturmodellierung und Pfadmodellierung. Sie gehen von einer sehr einfachen Struktur des analysierten Programmcodes, ohne Schleifen, wie er ihren Angaben zufolge typischerweise aus Codegeneratoren stammt, aus.

Die Pfade werden im Gegensatz zum Ansatz von Li und Malik explizit modelliert (pfadbasierte Methode). Da Schleifen ausgeschlossen sind, bleibt die Anzahl der Pfade überschaubar. Die WCET des Programms wird bestimmt durch die Suche des längsten Pfades im Kontrollflussgra-

fen mit Hilfe von Standard-Grafen-Algorithmen. Anschließend wird untersucht, ob der so ermittelte Pfad ausführbar ist. Ist er nicht ausführbar, wird er vom Graf ausgeschlossen und die Suche wiederholt, bis der längste Pfad gefunden wurde.

Zur Ermittlung ausführbarer Pfade verwenden sie Techniken zur symbolischen Berechnung von Variablen. Zunächst unbekannte Werte von Variablen werden aufgrund von Abhängigkeitsbeziehungen zwischen Variablen infolge von Zuweisungen in mögliche Wertebereiche eingeschränkt.

3.2.4 Bewertung der Ansätze

Die Aufstellung in Tabelle 3.1 gibt einen Überblick über die verschiedenen Ansätze der Pfadmodellierung. Die modellbasierte WCET-Analyse wurde hingegen bislang in der Literatur nicht detailliert betrachtet. Der Ansatz von Stappert und Altenbernd geht zwar von generiertem Code von Software-Entwicklungswerkzeugen aus; jedoch stellt die Verwendung von generiertem Code vornehmlich eine vereinfachende Einschränkung dar, um die Cache-Analyse zu ermöglichen. Weitergehende Konzepte der modellbasierten WCET-Analyse wurden nicht erstellt.

Bei der Betrachtung der Gegenüberstellung der bestehenden Ansätze in Tabelle 3.1 wird deutlich, dass der Sprachumfang der verwendeten Programmiersprachen durchweg eingeschränkt wird, um die WCET-Analyse zu ermöglichen bzw. die Forschungsarbeiten auf ein wesentliches Teilgebiet zu fokussieren. Insbesondere werden Rekursionen und dynamische Konstrukte nicht erlaubt.

Alle Ansätze benötigen zusätzliche Pfadinformationen, um die WCET-Analyse zu ermöglichen. Typischerweise werden sowohl Schleifeniterationen als auch Abhängigkeiten zwischen Zweigen benötigt. Es kann zwischen zwei Arten unterschieden werden, die benötigte Pfadinformation zu erhalten: Entweder werden die Pfadinformationen vom Programmierer eingegeben oder durch eine Datenflussanalyse erzeugt. Die Art der Übergabe der Pfadinformationen durch den Programmierer unterscheidet sich bei den verschiedenen Ansätzen sehr. Sie reicht von einer interaktiven Eingabe über die Übergabe als zusätzliche Konstrukte im Quellcode bis hin zu einer getrennten Übergabe unter Verwendung einer speziellen Beschreibungssprache.

Darüber hinaus lassen sich zwei Arten von Pfadmodellierungen unterscheiden, einerseits die explizite Aufzählung von Pfaden, andererseits die implizite Pfadmodellierung. Die implizite Pfadmodellierung ist sehr effizient und wurde auch von anderen Forschern aufgegriffen. Darüber hinaus stellt der Ansatz ein umfassendes Gesamtkonzept der WCET-Analyse dar, das auch komplexe Hardware und komplexere Codestrukturen effizient modellieren kann. Der Ansatz von Li und Malik stellt somit eine sehr gute Ausgangsbasis für weitergehende Ansätze dar.

Tabelle 3.1: Gegenüberstellung der Merkmale verschiedener Ansätze der WCET-Analyse

Ansatz	Restriktionen	Art der Pfadinformation	Übergabe der Pfadinformation	Pfadmodellierung/ Berechnungs-verfahren	Hardware-modellierung
Shaw	keine Rekursionen, keine dynamischen Konstrukte, nur abgeschrankte Schleifen, keine Compiler Opti-mierungen	maximale Schleifeniterationen	interaktive Abfrage durch das WCET-Analyse-Werkzeug	explizit/pfadbasiert	einfache HW MC 68010
Puschner und Koza	keine Rekursionen, keine dynamischen Konstrukte, nur abgeschrankte Schleifen	maximale Schleifeniterationen, Ausführungshäufigkeit von Anweisungen	neue Programmiersprache MARS-C: Erweiterung der Programmiersprache C um zusätzliche Konstrukte Marker, Scope, Loop Sequence	explizit/pfadbasiert	einfache HW
Mok	dito	Ausführungshäufigkeit von Anweisungen	während der WCET-Analyse durch Editierung generierter C-ähnlicher Skripte auf Assembler-Ebene	explizit/pfadbasiert	einfache HW: MC68000, 8086/88,i286 Navy TC2
Park	dito	Abhängigkeiten zwischen Zweigen	mittels Information Description Language	explizit/pfadbasiert	einfache HW MC 68010
Ermedahl und Gustaffson	keine Gleitkomma-arithmethik, Felder und Strukturen, Funktionsauf-rufe, dynamische Speicher-allokierung und Zeiger	Schleifeniterationen und Abhängigkeiten zwischen Anweisungen	Gewinnung der Informationen aus dem Code	Beschränkung auf die Gewinnung von Pfadinformationen	Beschränkung auf die Gewinnung von Pfadinforma-tionen
Li und Malik	keine Rekursionen, keine dynamischen Konstrukte, nur abgeschrankte Schleifen	maximale Schleifeniterationen Ausführungshängigkeit von Anweisungen, Abhängigkeiten zwischen Zweigen	interaktive Abfrage durch das WCET-Analyse-Werkzeugs	implizit/ILP	Direct Mapped Instruktions- und Daten-Caches
Stappert und Altenbernd	keine Schleifen statische Daten	Abhängigkeiten zwischen Anweisungen	Gewinnung der Informationen aus dem Code	explizit/pfadbasiert	Piplining, Instruktions- und Daten-Cache mit statischen Daten

3.3 Zusammenfassende Bewertung und Festlegung der Anforderungen

3.3.1 Defizite bei der Entwicklung und Analyse zeitgesteuerter Systeme

Bei der Analyse des Stands der Technik sind bei den bestehenden Ansätzen für die Entwicklung und Analyse zeitgesteuerter Systeme Defizite zu erkennen, auf deren Grundlage anschließend Anforderungen an die vorliegende Arbeit abgeleitet werden.

Die WCET-Analyse findet bislang keine Beachtung bei der Softwareentwicklung in der Automobilindustrie, wenngleich sie auch bei der Entwicklung herkömmlicher ereignisgesteuerter Systeme hilfreich wäre. Obwohl in der Literatur eine Vielzahl von WCET-Analyse-Ansätzen zu finden ist, wurde die WCET-Analyse bisher nicht in die in der Automobilindustrie eingesetzten Werkzeugumgebungen integriert. Auch die Entwicklung notwendiger Konzepte der modellbasierten WCET-Analyse steht noch aus.

Zur Entwicklung zeitgesteuerter Systeme wurden in den letzten Jahren die Schnittstellen-fokussierte Methodik und verschiedene Werkzeugumgebungen konzipiert und erstellt, die jedoch noch erhebliche Schwächen aufweisen. Einerseits wird der Zeitanalyse kein ausreichender Stellenwert beigemessen, andererseits werden die in der Automobilindustrie etablierten Vorgehensweisen zur Softwareentwicklung nicht genügend berücksichtigt.

Die bereits erstellten Werkzeugumgebungen basieren auf der Schnittstellen-fokussierten Methodik und können demzufolge die Entwicklung von zeitgesteuerten Systemen nicht ausreichend unterstützen. Generell wird davon ausgegangen, dass der zeitliche Ablauf im zeitgesteuerten System das Resultat eines Spezifikations- und Generierungsprozesses ist. Es wird nicht berücksichtigt, dass ein Entwickler konkrete Vorstellungen über den zeitlichen Ablauf im zeitgesteuerten System hat, insbesondere über den zeitlichen Ablauf eines verteilten Regelkreises. Die bestehenden Werkzeugumgebungen bieten demzufolge keine ausreichenden Möglichkeiten zur zielgerichteten Planung von zeitlichen Abläufen im verteilten System. Darüber hinaus fehlt ihnen die Unterstützung der Entwicklung regelungstechnischer Anwendungen.

Aus obigen Feststellungen resultiert die Forderung nach der Definition geeigneter Verfahren und Werkzeugkonzepte zur Unterstützung der Entwicklung und Analyse zeitgesteuerter Systeme. Die modellbasierte WCET-Analyse spielt hierbei die zentrale Rolle. Im Hinblick auf eine Vielzahl eingeführter Software-Entwicklungswerkzeuge ist zunächst die Erarbeitung eines allgemeines Konzepts angebracht, das auf konkrete Werkzeuge übertragbar ist. Im Anschluss kann das Konzept in einen neu zu definierenden Entwicklungsprozess für zeitgesteuerte Systeme eingebettet werden. Darauf aufbauend kann ein Werkzeugkonzept entwickelt werden, das das definierte Vorgehen geeignet unterstützt.

3.3.2 Anforderungen an die modellbasierte WCET-Analyse

An die modellbasierte WCET-Analyse wird, wie an die herkömmliche WCET-Analyse auch, die Forderung nach der Bestimmung enger oberer Schranken der maximalen Ausführungszeit gestellt. Zur Realisierung einer durchgängigen Werkzeugkette muss die modellbasierte WCET-Analyse automatisch, ohne Eingreifen des Anwenders, erfolgen. Anpassungen am generierten Code durch den Anwender dürfen nicht mehr erfolgen, damit der Code statisch analysierbar ist. Auch zusätzliche Pfadinformationen darf der Anwender nicht mehr bereitstellen, da er keine Kenntnis über das dynamische Ausführungsverhalten des Codes, den die Software-Entwicklungswerkzeuge generieren, haben kann. Sowohl zur Anpassung des Codes als auch zur Bereitstellung notwendiger Pfadinformationen müssen geeignete Konzepte entwickelt werden, die entweder bei den Software-Entwicklungswerkzeugen oder bei den WCET-Analyse-Verfahren ansetzen.

Bei der Analyse des Stands der Technik hat sich gezeigt, dass umfangreiche WCET-Analyse-Verfahren bekannt sind. Sie sind jedoch nicht direkt anwendbar. Es sollte deshalb auf die vorhandenen WCET-Analyse-Verfahren zurückgegriffen und diese gegebenenfalls verändert bzw. erweitert werden. Die Mikroarchitekturmodellierung ist weit fortgeschritten, sodass sie zum Erreichen des geforderten Ziels der modellbasierten WCET-Analyse nicht mehr weiterentwickelt werden muss.

3.3.3 Anforderungen an einen Entwicklungsprozess für zeitgesteuerte Systeme

Bei der Softwareentwicklung im Automobilbereich ist die Entwicklung eines zeitgesteuerten Systems ein neuer Aspekt. Dieser darf den bestehenden Entwicklungsprozess nicht völlig umwerfen. Der Aufwand zur Einführung eines neuen Entwicklungsprozesses, die Schulung der Mitarbeiter und die Einführung neuer Werkzeuge wären nicht vertretbar. Idealerweise sollten zusätzliche Planungsaktivitäten, die bei zeitgesteuerten Architekturen notwendig werden, in den vorhandenen Entwicklungsprozess eingegliedert werden. Dieser ist in der Automobilindustrie typischerweise als V-Modell [BrDr95] definiert.

Die zusätzlichen Entwicklungsaktivitäten, die aufgrund der Zeitsteuerung notwendig werden, müssen an entsprechender Stelle im V-Modell berücksichtigt werden. Zu diesen zählen die Planung der zeitlichen Abfolge von Aktionen im verteilten System und der Nachweis der Einhaltung der geplanten Zeitanforderungen durch die Zeitanalyse. Des Weiteren ist die Unterstützung einer schrittweisen Verfeinerung der zeitlichen Abläufe zu beachten, die parallel zur Verfeinerung funktionaler Anforderungen erfolgen sollte.

Die Entwicklung von Automobilen ist ein langwieriger komplexer und iterativer Vorgang, der in einer Arbeitsteilung zwischen dem Automobilhersteller und den Zulieferern stattfindet. Es sind

deshalb geeignete Rückschritte im V-Modell zu berücksichtigen. Darüber hinaus sind präzise definierte Entwicklungsschnittstellen erforderlich. Neben den herkömmlichen Schnittstellen im Wertebereich kommt bei zeitgesteuerten Systemen eine Schnittstelle im Zeitbereich hinzu.

3.3.4 Anforderungen an ein Werkzeugkonzept für die Entwicklung zeitgesteuerter Systeme

Ein Werkzeugkonzept zur Entwicklung zeitgesteuerter Systeme sollte die Planung, die Zeitanalyse sowie die Entwicklung verteilter regelungstechnischer Anwendungen geeignet unterstützen. Diese Anforderungen werden im Folgenden näher erläutert.

Das Werkzeugkonzept sollte die Spezifikation der konkreten Vorstellung eines Entwicklers über den zeitlichen Ablauf von Aktionen im zeitgesteuerten System auf einfache und intuitive Weise, ohne langwierigen Spezifikationsaufwand, unterstützen. Dabei sollte eine schrittweise Verfeinerung des zeitlichen Ablaufs von Aktionen im zeitgesteuerten System möglich sein. Auch das Nachrichten- und Task Scheduling sollte grafisch möglich sein, um einen direkten visuellen Eindruck der entstehenden Auswirkungen zu erhalten.

Die Zeitanalyse muss in das Werkzeugkonzept integriert sein, um die Korrektheit der Planungsdaten zu überprüfen. Die WCET-Analyse hat dabei die maximale Ausführungszeit der modellbasiert entwickelten Software zu bestimmen. Nach erfolgter WCET-Analyse sollte auf einfache Weise nachprüfbar sein, ob die neu bestimmten WCET-Analyse-Ergebnisse alle Zeitanforderungen weiterhin erfüllen. Aus diesem Grund sollte in die Werkzeugumgebung eine Schedulability-Analyse integriert sein.

Die Werkzeugumgebung muss eine Latenzzeit-Analyse von Signalen durchführen und eine Schnittstelle zu den regelungstechnischen Software-Entwicklungswerkzeugen, die die Analyseergebnisse benötigen, besitzen. Um die Parametrisierung der Regler entsprechend der aktuellen Latenzzeit-Analyse-Ergebnisse zu garantieren, sollten die Ergebnisse direkt an das regelungstechnische Software-Entwicklungswerkzeug übergeben werden.

In diesem Kapitel wurde der Stand der Technik im Hinblick auf die Unterstützung der Entwicklung und Analyse zeitgesteuerter Systeme untersucht. Es wurde festgestellt, dass ein Konzept der modellbasierten WCET-Analyse benötigt wird, das die WCET-Analyse von modellbasiert erstellter Software ermöglicht. Dazu wird im nächsten Kapitel ein allgemeines Konzept erarbeitet. Darüber hinaus muss für die Entwicklung zeitgesteuerter Systeme eine praxistaugliche Methodik und eine geeignete Werkzeugunterstützung zur zielgerichteten Planung von zeitlichen Abläufen im verteilten System entwickelt werden. Darauf wird in den Kapiteln 5 bis 7 eingegangen.

4 Konzept der modellbasierten WCET-Analyse

Die WCET-Analyse hat bei der Entwicklung zeitgesteuerter Systeme eine herausragende Bedeutung. Zur Realisierung einer durchgängigen Werkzeugkette muss die WCET-Analyse die mit Software-Entwicklungswerkzeugen erstellten Modelle automatisch, ohne Eingreifen des Anwenders, analysieren können. Dazu bedarf es eines Konzepts der modellbasierten WCET-Analyse, das in diesem Kapitel erstellt wird.

Zunächst werden wesentliche Eigenschaften des modellbasierten Softwareentwurfs analysiert und daraus das Konzept der modellbasierten WCET-Analyse abgeleitet. Das Grundkonzept der modellbasierten WCET-Analyse wird vorgestellt, die Teilaspekte werden ausführlich diskutiert und verschiedene Realisierungen untersucht. Anschließend werden die notwendigen Erweiterungen bekannter WCET-Analyse-Verfahren für die modellbasierte Analyse dargelegt und der vollständige Ablauf der modellbasierten WCET-Analyse anhand eines einfachen Beispiels dargestellt. Abschließend werden Kriterien herausgestellt, anhand derer Software-Entwicklungswerkzeuge bezüglich der Übertragbarkeit des erarbeiteten Konzepts bewertet werden können.

4.1 Modellbasierter Softwareentwurf

4.1.1 Motivation für den modellbasierten Softwareentwurf

Das Ziel des modellbasierten Softwareentwurfs ist die Realisierung einer durchgängigen Werkzeugkette durch ein automatisiertes Ineinandergreifen aller am Entwicklungsprozess beteiligten Werkzeuge ohne manuelle Konvertierungsschritte. Beim modellbasierten Softwareentwurf wird das gewünschte Verhalten der Software durch abstrakte Spezifikation in einem ablauffähigen Modell dargestellt. Die Modellierung erfolgt meist unter Verwendung von grafischen Beschreibungstechniken. Durch Simulation kann das ablauffähige Modell validiert werden. In einigen Fällen ist der formale Nachweis von Eigenschaften möglich. Der Code für ein Steuergerät (sog. *Zielcode*) ist durch die automatische Codegenerierung direkt an das Modell gekoppelt. Dadurch entfällt die fehleranfällige und zeitaufwändige Implementierung. Der Entwickler kann sich auf konzeptionelles Denken konzentrieren und ist nicht mit implementierungsnahen Überlegungen belastet [GöFu98]. Durch die automatische Codegenerierung können kürzere Änderungszyklen erreicht und somit Entwicklungskosten eingespart werden. Außerdem kann eine höhere Qualität der realisierten Fahrzeugfunktion erreicht werden, da die Modelle besser an den jeweiligen technischen Prozess angepasst werden können [KWB98].

4.1.2 Beschreibungstechniken zur Modellierung

Beim modellbasierten Softwareentwurf erfolgt die Erstellung der Modelle meist durch Zusammensetzen von standardisierten Modellelementen. Die grafischen Beschreibungstechniken zur Modellierung der Software hängen dabei vom jeweiligen Anwendungsgebiet ab. By-Wire-Systeme sind hybride Systeme, die sowohl ereignisdiskrete als auch kontinuierliche Subsysteme besitzen. Ereignisdiskrete Systeme können sehr gut durch Zustandsautomaten, kontinuierliche Systeme durch Blockdiagramme beschrieben werden. Diese beiden Beschreibungstechniken werden im Folgenden kurz vorgestellt.

Zustandsautomaten

Bei ereignisdiskreten Systemen, auch reaktive oder zustandsbasierte Systeme genannt, erfolgt die Reaktion auf ein Ereignis, indem zwischen verschiedenen Verhaltensmodi gewechselt wird [KWB98]. Sie können deshalb sehr gut durch Zustandsautomaten beschrieben werden, die aus einer endlichen nichtleeren Menge von Zuständen bestehen, wobei stets nur ein Zustand aktiv sein kann. Ein Zustandsautomat ist durch die drei Größen Zustände, Eingangsgrößen und Ausgangsgrößen definiert (siehe Abbildung 4.1) [LaGö99]. Zwischen Zuständen existieren Zustandsübergänge, die in Abhängigkeit von den Eingangsgrößen (z. B. Ereignisse, anliegende Signale) ausgeführt werden [LaGö99]. Die Ausgangsgrößen werden durch Ausführen von Aktionen, beim Zustandsübergang (Mealy-Automat) oder in einem Zustand (Moore-Automat) ausgegeben [Balz96]. Harel [Hare87] führte mit den sog. *Statecharts* Konzepte wie Hierarchie und Parallelität zur Modellierung komplexer Zusammenhänge ein. Bei hierarchischen Zustandsautomaten besteht ein Zustand selbst wiederum aus einem Zustandsautomaten. Parallele Zustandsautomaten können mehrere aktive Zustände gleichzeitig haben. Zustandsautomaten können grafisch als Zustandsdiagramm dargestellt werden (siehe Abbildung 4.2), wobei Zustände durch abgerundete Rechtecke oder Kreise symbolisiert werden. Zustandsübergänge werden durch Pfeile zwischen Ausgangs- und Folgezustand dargestellt, mit angehefteten Übergangsbedingungen und auszuführender Aktion.

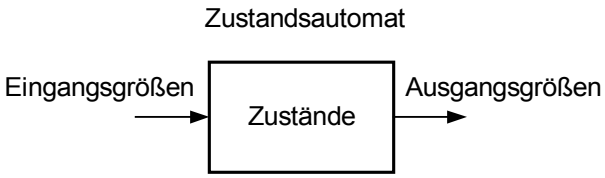


Abbildung 4.1: Definition eines Zustandsautomaten

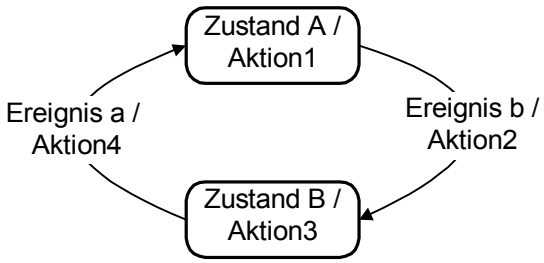


Abbildung 4.2: Darstellung eines Zustandsautomaten als Zustandsdiagramm

Blockschaltbilder

Bei kontinuierlichen Systemen, auch dynamische Systeme genannt, sind die Eingangsgrößen im Sinne einer mathematischen Funktion mit den Ausgangsgrößen verknüpft [LaGö99]. Sie werden in der Regel als Blockschaltbilder dargestellt, wie sie aus der Regelungstechnik [Föll78] bekannt sind. Blockschaltbilder, auch Strukturbilder genannt, unterstützen die Strukturierung komplexer Systeme [KWB98], indem sie die Aufteilung eines Systems in Subsysteme und Datenflüsse zwischen Subsystemen ermöglichen. Elemente der Blockschaltbilder sind (siehe Abbildung 4.3):

- Variablen, Parameter, Konstanten
- Eingangs- und Ausgangsgrößen
- arithmetische und logische Operationen, Zugriffsoperationen auf Datenstrukturen
- unterlagerte Subsysteme und
- Datenflusslinien.

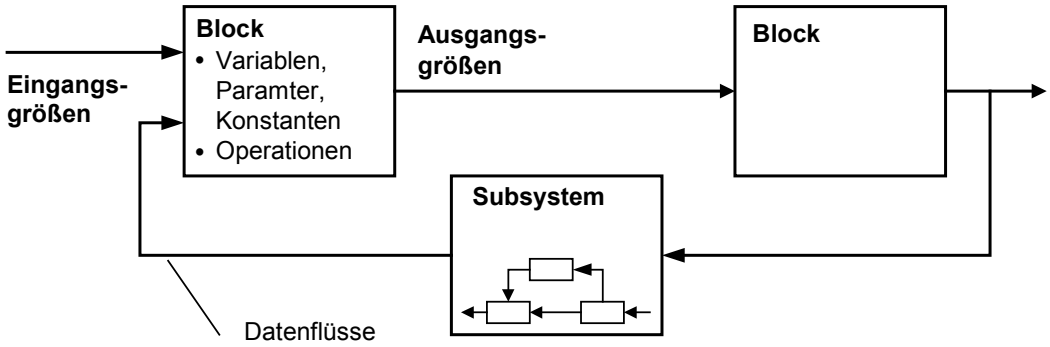


Abbildung 4.3: Elemente eines Blockschaltbilds

4.1.3 Codegenerierung

Die Generierung des Zielcodes auf Basis des erstellten Modells erfolgt typischerweise in mehreren Stufen (siehe Abbildung 4.4). Aus dem Modell wird zunächst Quellcode einer höheren Programmiersprache erzeugt. Codegenerierungsvorschriften steuern hierbei den Codegenerierungsprozess und ermöglichen somit die Beeinflussung der Codegenerierung und die Anpassung an bestimmte Zielhardware sowie die Optimierung hinsichtlich Speicherplatz- und Laufzeiteffizienz. Bestimmte feststehende, unveränderliche Bibliotheksfunktionen und Rahmenprogramme liegen oftmals als Quellcode vor. Neben dem Quellcode wird zusätzlich vom Codegenerator ein sog. *Makefile* erzeugt, das den Compilierungsprozess koordiniert. Als Zwischenformat wird hierbei vom Compiler Assemblercode erzeugt, bevor durch den *Assembler* und *Linker* ein ausführbares Programm erzeugt wird. Der Compiler greift auf sog. *Compiler-Bibliotheken* zu, in denen beispielsweise mathematische Funktionen gespeichert sind.

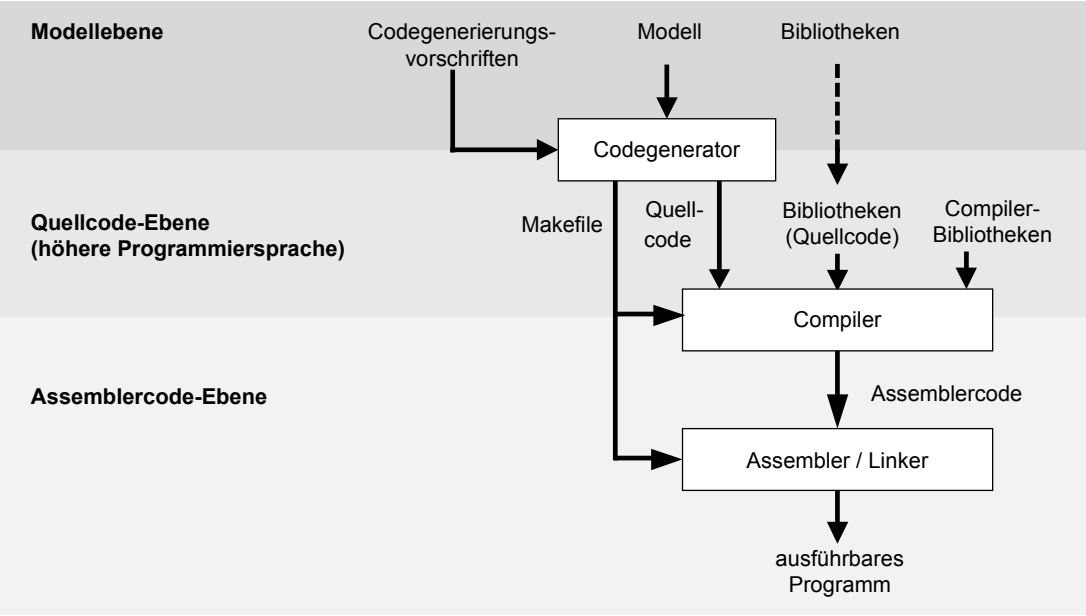


Abbildung 4.4: Typischer Codegenerierungsprozess beim modellbasierten Softwareentwurf

4.1.4 Abbildungsbeziehung zwischen Modellelementen und Codestruktur

Der modellbasierte Softwareentwurf erfolgt auf einer hohen Abstraktionsebene. Der Anwender des Software-Entwicklungswerkzeugs ist aus diesem Grund bei den WCET-Analyse-Ergebnissen nicht mehr an Detailinformationen wie Ausführungszeiten einzelner Anweisungen interessiert, die bisher von den bekannten WCET-Analyse-Ansätzen ausgegeben wurden. Vielmehr benötigt er Informationen, die der Abstraktionsebene entsprechen und ihn bei Auswahl, Verknüpfung und Parametrisierung von Modellelementen unterstützen. Die WCET-Analyse sollte deshalb idealerweise auf Basis von Modellelementen erfolgen.

Verknüpfungen der Modellelemente im Modell entscheiden über Anordnung und Art der verwendeten Anweisungen und die Struktur des generierten Quellcodes und somit über die Struktur des Assemblercodes, der vom Compiler erzeugt wird. Eine direkte Abbildungsbeziehung zwischen Modellelementen auf Modellebene und der Codestruktur auf Assemblercode-Ebene kann im Allg. nicht gefunden werden, sie hängt vom gesamten Codegenerierungsprozess ab. Folgende Beobachtungen können festgehalten werden:

- Ein Modellelement kann Beiträge an vielen Stellen des generierten Codes besitzen (Abbildung 4.5 Pfeile A und B).
- Ein Grundblock kann wiederum Beiträge von mehreren Modellelementen aufnehmen (Pfeile A und C).

- Je nach Art der Verknüpfung können sich Beiträge von Modellelementen addieren oder keine Auswirkung haben, da sie sich auf sich gegenseitig ausschließenden Pfaden befinden (siehe Abbildung 4.5 Pfeile B und D).

- Die Gesamtheit aller Modellelemente bestimmt über die Struktur des generierten Codes. Durch Hinzufügen eines Modellelements in ein bestehendes Modell kann die gesamte Struktur des generierten Codes verändert werden.

Entscheidend für die Ausführungszeit ist die Struktur des generierten Codes auf Assemblercode-Ebene. Die Struktur des Assemblercodes und die Assemblerinstruktionen liefern die atomaren Zeitbeiträge, aus denen sich die WCET zusammensetzt (siehe Abschnitt 3.2). Die WCET kann deshalb nicht durch die direkte Analyse des Modells ermittelt werden, sondern muss nach erfolgter Codegenerierung auf Basis des generierten Assemblercodes erfolgen.

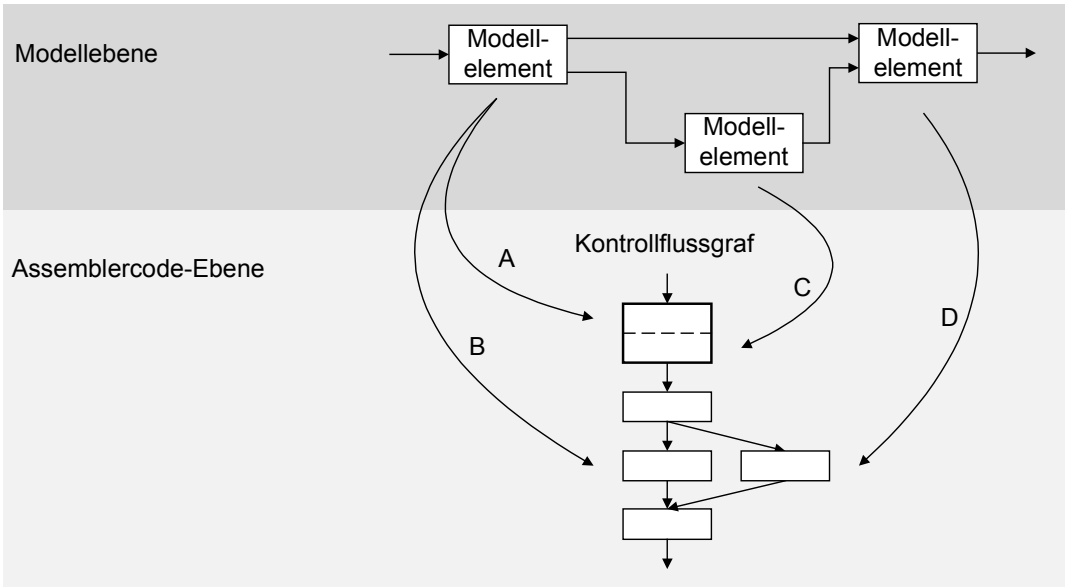


Abbildung 4.5: Abbildungsbeziehung zwischen Modellelementen und Codestruktur

4.2 Grundkonzept der modellbasierten WCET-Analyse

Für die WCET-Analyse der modellbasiert erstellten Software nach erfolgter Codegenerierung ist die Verwendung bekannter Ansätze aus Kapitel 3 naheliegend. Die Ansätze mit Datenflussanalyse verlangen große Einschränkungen beim Sprachumfang der verwendeten Programmiersprachen und sind deshalb im Allg. für generierten Code nicht einsetzbar. Die anwendbaren WCET-Analyse-Ansätze verlangen das manuelle Eingreifen des Programmierers. Sie erfordern das Einhalten von Codierungsrichtlinien und das Einbringen von Zusatzwissen über das dynamische Ausführungsverhalten des Codes. Zur Realisierung einer durchgängigen Werkzeugkette muss die modellbasierte WCET-Analyse jedoch automatisch, ohne Eingreifen des Anwenders, erfol-

gen. Denn es ist nicht tragbar, dass der Anwender eines Software-Entwicklungswerkzeugs nach jeder Codegenerierung den Code gemäß den geforderten Codierungsrichtlinien anpassen oder Informationen über das dynamische Ausführungsverhalten des Codes beisteuern muss.

Aus diesen Überlegungen resultiert die Grundidee des Konzepts (siehe Abbildung 4.6): Statt auf den tiefen Assembler- oder Quellcode-Ebenen anzusetzen, kann der hohe Abstraktionsgrad genutzt werden, um Code derart zu genieren, dass er leicht statisch analysierbar ist. Darüber hinaus sind im Modell meist bereits Informationen über das dynamische Ausführungsverhalten des Codes vorhanden, die im Quellcode nur sehr schwer und im Assemblercode nicht mehr zu erkennen sind. Die Informationen aus dem Modell müssen der WCET-Analyse als Pfadinformationen zur Verfügung gestellt werden. Beispielsweise sind im Modell bekannte Informationen wie Abhängigkeiten zwischen Pfaden in unterschiedlichen Funktionen eines Programms nur noch mit sehr hohem Aufwand im Quellcode zu identifizieren. Auf Assemblercode-Ebene sind selbst statische Schleifenschranken sehr schwer zu erkennen, da sie sich hinter schwer zuzuordnenden Lade-, Vergleichs- und Sprunginstruktionen verbergen.

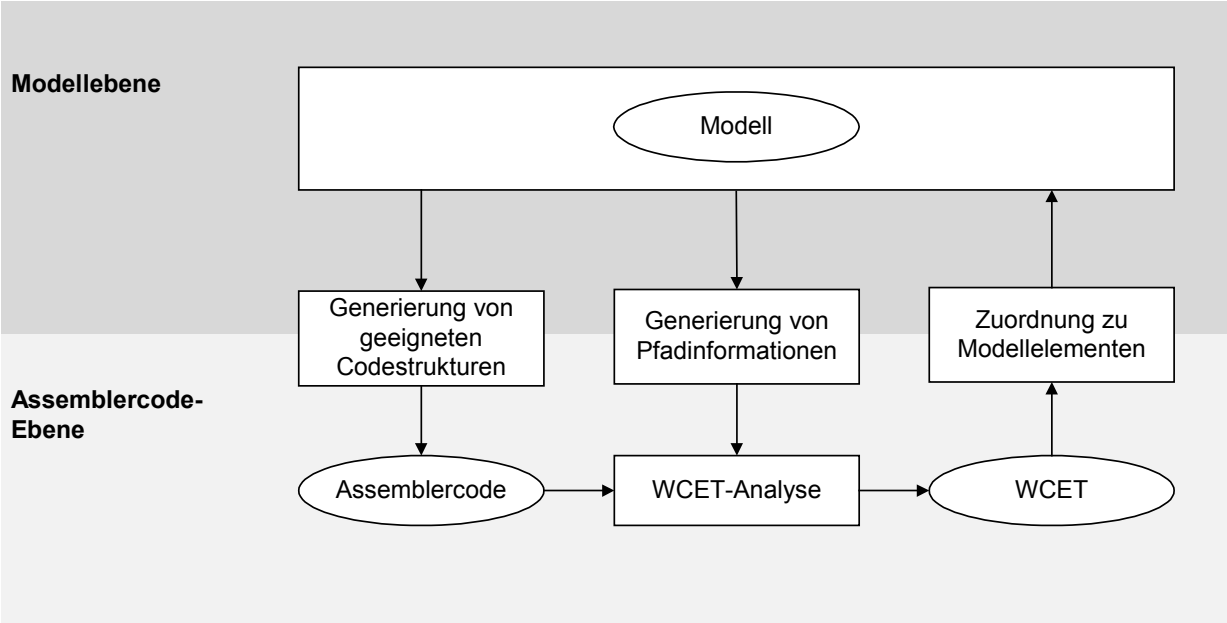


Abbildung 4.6: Konzept der modellbasierten WCET-Analyse

Zwischen geeigneter Codegenerierung und Generierung von Pfadinformationen muss abgewogen werden. Ob der Code so generiert werden kann, dass er leicht zu analysieren ist, oder ob Pfadinformationen erzeugt werden müssen, die den ansonsten nicht statisch analysierbaren Code verständlich machen, hängt von den Eigenschaften des Modells und dem Codegenerierungskonzept ab. Die Generierung von rekursiven Aufrufen, die ohne weitere Angaben nicht statisch analysierbar wären, können beispielsweise erlaubt werden, sofern die maximale Rekursionstiefe auf Modellebene bekannt ist.

Durch die gewollte Abstraktion geht der Bezug zum Code und dessen Ausführungszeit verloren. Hinter den Modellelementen kann sich komplexer Code verbergen, dessen Umfang den Modellelementen nicht angesehen werden kann. Um dem Anwender der Abstraktionsebene entsprechende Informationen zu übergeben, müssen die WCET-Analyse-Ergebnisse den Modellelementen zugeordnet werden. Dabei kann zwischen zwei Arten von Informationen unterschieden werden:

- Die Beiträge einzelner Modellelemente zur WCET unterstützen den Anwender bei der Auswahl, Verknüpfung und Parametrisierung von Modellelementen. Gegebenenfalls könnte er sich für ein Modellelement mit geringerer Ausführungszeit entscheiden, bzw. die Parametrisierung eines Modellelements entsprechend vornehmen.

- Integrale Informationen über die WCET des Codes zur Initialisierung, Terminierung und zyklischen Ausführung des Modells werden zur Planung des zeitgesteuerten Systems benötigt. Sie sind Informationen über die Ausführungszeit der Gesamtheit aller Modellelemente.

Die Umsetzung des Konzepts auf konkrete Software-Entwicklungswerkzeuge hängt von deren Eigenschaften ab. In den nächsten Abschnitten werden dazu verschiedene Möglichkeiten vorgestellt.

4.3 Generierung von geeigneten Codestrukturen

4.3.1 Problemstellung bei der Generierung von geeigneten Codestrukturen

Die Modellierung der Software auf einem abstrakten Niveau, entfernt von einer konkreten Implementierung, ermöglicht große Freiheiten bei der Codegenerierung. Für den Einsatz in Serien-Steuergeräten optimieren die Werkzeughersteller derzeit Codegeneratoren im Hinblick auf geringen Speicherplatzbedarf von RAM und ROM und Minimierung der durchschnittlich benötigten Ausführungszeit des generierten Codes. Zeitgesteuerte Anwendungen stellen neue Anforderungen an die Codegenerierung. Um die WCET-Analyse des Codes durchführen zu können, muss der Code statisch analysierbar sein. Des Weiteren sollte der Code im Hinblick auf eine minimale WCET optimiert werden:

- **Statische Analysierbarkeit:** Die ausführbaren Pfade des Codes müssen, unabhängig davon ob der Code von einem Programmierer oder vom Software-Entwicklungswerkzeug stammt, durch eine statische Analyse bestimmbar sein. Der generierte Code darf nur zeitlich deterministische Codestrukturen enthalten, deren Ausführungsverhalten statisch vorhersagbar ist. Dynamische Konstrukte, deren Ausführungsverhalten erst zur Laufzeit bekannt ist, dürfen nicht verwendet werden. Beispiele hierfür sind dynamisches Anfordern von Speicher zur

Laufzeit oder Zeiger auf Funktionen, bei denen erst zur Laufzeit feststeht, welche Funktion tatsächlich ausgeführt wird.

- **Minimale WCET:** Für herkömmliche ereignisgesteuerte Systeme sind die Code-Optimierungen so ausgelegt, dass im Durchschnitt eine kleine Ausführungszeit erreicht wird. Wird in seltenen Fällen eine längere Ausführungszeit benötigt, fällt dies nicht ins Gewicht. Bei zeitgesteuerten Systemen hingehen spielt eine durchschnittliche Ausführungszeit keine bedeutende Rolle. Es muss stets die maximale Ausführungszeit bei der Planung des zeitgesteuerten Systems berücksichtigt werden. Der Code muss deshalb bei zeitgesteuerten Systemen so generiert werden, dass die Ausführungszeit auch im schlechtesten Fall minimal ist.

Um die oben genannten Anforderungen auch beim ausführbaren Programm wiederzufinden, müssen während des gesamten Codegenerierungsprozesses geeignete Maßnahmen ergriffen werden. Es gilt zu verhindern, dass die Optimierung in einer Stufe durch eine spätere Stufe wieder zunichte gemacht wird. In Abbildung 4.7 sind die notwendigen Maßnahmen in den verschiedenen Stufen der Codegenerierung dargestellt, im Folgenden wird anhand von Beispielen näher auf sie eingegangen.

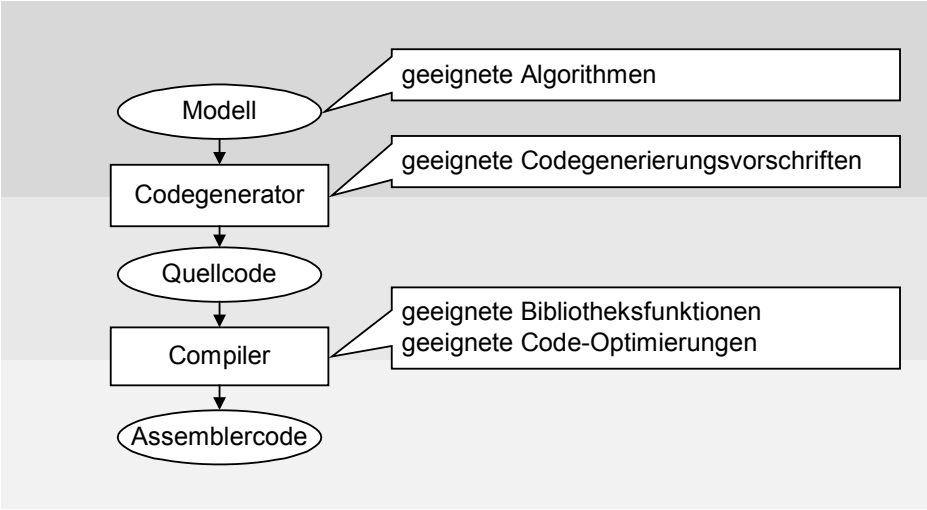


Abbildung 4.7: Ebenen der Optimierung der Codegenerierung für zeitgesteuerte Anwendungen

4.3.2 Geeignete Algorithmen

Ein Algorithmus ist ein Verfahren zur Lösung einer Klasse gleichartiger Probleme [Schn91]. Ein Unterscheidungskriterium von Algorithmen ist die Iterationszahl zur Erreichung einer vorgeschrieben Güte. Sie kann je nach Algorithmus datenabhängig variieren oder im Voraus datenunabhängig feststehen. Für die Lösung von numerischen Gleichungen kann beispielsweise entweder das Gauss- oder das Jacobi-Verfahren angewandt werden. Beim Gauss-Verfahren ist die Iterationszahl für eine feste Größe des Gleichungssystems bekannt, während beim Jacobi-Ver-

fahren die Iterationszahl sehr stark abhängig von den Werten der Variablen ist [Kaus89]. Im Gegenzug benötigt das Gauss-Verfahren im Vergleich zum Jacobi-Verfahren mehr Speicherplatz. Für zeitgesteuerte Anwendungen dürfen nur Algorithmen verwendet werden, die nach bekannter Iterationszahl terminieren, auch wenn dadurch ein größerer Speicherplatzbedarf in Kauf genommen werden muss.

4.3.3 Geeignete Codegenerierungsvorschriften

Die Codegenerierungsvorschriften legen fest, auf welche Art und Weise das abstrakte Modell in Code umgesetzt wird. Für die Implementierung von Zustandsautomaten beispielsweise sind in der Literatur [Berr98, BGP98] zahlreiche Varianten bekannt. Die Möglichkeiten reichen von der Implementierung der verschiedenen Zustände durch *switch*-Konstrukte, über die Implementierung als boole'sches Gleichungssystem, das zur Laufzeit gelöst wird, bis hin zur expliziten Repräsentation der Zustände und Übergänge in Datenstrukturen. Die Implementierung mit switch-Konstrukten ist einfach statisch mit den bekannten Verfahren zu analysieren, da statisch bekannt ist, dass sich verschiedene *case*-Zweige ausschließen. Wird jedoch der Zustandsautomat in Datenstrukturen gespeichert, die in einer einfachen Schleife durchsucht werden, reicht die Analyse der statischen Codestruktur nicht aus. Denn aus der statischen Codestruktur kann nicht auf die ausführbaren Pfade geschlossen werden. Durch alleiniges Betrachten der Schleife ist nicht bekannt, wie oft sie im Worst-Case iterieren muss, bis der Zustandsautomat einen neuen Zustand erreicht und welche Aktionen beim Übergang ausgeführt werden. Es ist vielmehr eine gesonderte Analyse der Inhalte der Datenstrukturen notwendig, um die WCET ermitteln zu können.

4.3.4 Compiler-Regeln und Code-Optimierung des Compilers

Compiler-Regeln legen fest, wie der Quellcode in Assembler-Code übersetzt wird. Code-Optimierungen zielen auf die Erzeugung von laufzeit- bzw. speicherplatzeffizientem Assemblercode [ASU88] ab.

Sequentieller Quellcode muss nicht zwangsläufig zu sequentiellem Assemblercode führen. Der Compiler kann Sprünge und Schleifen einfügen. Wird beispielsweise eine 32-Bit-Integer-Variable auf einem 16-Bit-Rechner verwendet, müssen zwangsläufig zwei 16-Bit-Register verwendet werden, um die Variable aufzunehmen. Eine Schiebeoperation lässt sich dann nicht mehr durch eine Instruktion realisieren, deren Ausführungszeit unabhängig von der Anzahl Stellen ist, um die verschoben wird. Statt dessen muss die Schiebeoperation durch eine Schleife realisiert werden, deren Ausführungszeit abhängig von der Anzahl Stellen ist, um die verschoben wird. Dieser Schleife auf Assembler-Code steht keine Schleife im Quellcode gegenüber und ist somit ohne Zusatzwissen nicht statisch analysierbar.

Herkömmliche Code-Optimierungen können zwar die durchschnittlichen Ausführungszeiten verkürzen; es ist jedoch zu beachten, dass sie zu einer Verschlechterung der Ausführungszeit im

Worst-Case führen können. Ein Beispiel hierfür ist die *Logical Expression Optimization*. Anstatt boole'sche Operationen durch boole'sche Assembler-Instruktionen zu realisieren, werden Vergleich- und Sprunginstruktionen verwendet. So wird aus linearem Quellcode stark verzweigter Assemblercode, der viele Sprünge enthält. Zwar verkürzt sich die Ausführungszeit in den meisten Fällen, die WCET vergrößert sich jedoch, da im Worst-Case erheblich mehr zeitaufwändige Sprunginstruktionen durchlaufen werden. Aus diesem Grund muss auf solche Code-Optimierungen verzichtet werden.

4.3.5 Compiler-Bibliotheksfunktionen

Compiler-Bibliotheksfunktionen werden während des Compilierungsprozesses vom Compiler eingefügt. Sie sind feststehende Funktionen und müssen deshalb nur einmal auf ihre statische Analysierbarkeit hin untersucht werden. Sind sie nicht statisch analysierbar, so müssen sie verändert bzw. durch andere ersetzt werden. Schwierigkeiten können entstehen, wenn sie nur als Objektcode und nicht als Quellcode verfügbar sind. Die statische Analyse fällt dann bedeutend schwieriger aus, da der Objektcode zunächst disassembliert werden muss und Schleifengrenzen auf Assemblercode-Ebene, ohne Bezug zum Quellcode, sehr viel schwerer zu erkennen sind.

Verschiedene Vertreter von Compiler-Bibliotheksfunktionen werden im Folgenden bezüglich ihrer Eignung bei zeitgesteuerten Systemen untersucht.

Mathematische Operationen
Mathematische Operationen zur Berechnung von Sinus, Kosinus oder Quadratwurzel sind typische Funktionen, die in Compiler-Bibliotheken abgelegt sind. Zur Berechnung der Quadratwurzel beispielsweise gibt es verschiedene Bibliotheken, die dasselbe Newton'sche Verfahren unterschiedlich implementieren. In der Funktion der *Tasking* Compiler-Bibliothek für den Siemens 80C167 Mikrocontroller [Task98] wird abhängig von der erreichten Genauigkeit des Ergebnisses abgebrochen, während in [Jams86] eine Implementierung beschrieben ist, die nach einer festgelegten Iterationszahl abbricht, ungeachtet der Genauigkeit des Ergebnisses. Für zeitgesteuerte Systeme muss die zweite Variante zum Einsatz kommen.

Laufzeitsysteme für Objektorientierte Programmiersprachen
Objektorientierte Programmiersprachen wie C++ oder Ada95 besitzen objektorientierte Konzepte wie Polymorphismus und Vererbung. Polymorphismus ermöglicht dem Programmierer, eine einzige Funktion zu erzeugen, die auf verschiedene Typen von Objekten angewandt werden kann. Vererbung ermöglicht dem Programmierer, neue Objekte zu definieren, die die Eigenschaften anderer Objekte erben können. Diese Konzepte erfordern jedoch das Prinzip der späten Bindung. Funktionsaufrufe können nicht zur Compilierzeit mit den Funktionen verbunden werden. Erst zur Laufzeit wird in Tabellen die aufzurufende Funktion gesucht. Die Ausführungszeit ist vom aktuellen Zustand abhängig (z. B. Anzahl erzeugter Objekte) und kann somit nicht vor

der Laufzeit determiniert werden. Für den Einsatz in harten Echtzeitsystemen muss deshalb von der späten Bindung Abstand genommen werden.

Automatisches Speichermanagement
Automatisches Speichermanagement, z. B. dynamisches Allokieren des Speichers und Beseitigung von blockiertem, nicht mehr genutztem Speicher (*Garbage Collection*), wird bei objektorientierten Programmiersprachen wie beispielsweise Java [GJSB00] eingesetzt [Plöd00]. Das Speichermanagement läuft meist im Hintergrund ab und kann nicht beeinflusst werden. Da es sich um einen dynamischen Vorgang handelt, kann weder der Zeitpunkt, zu dem er zuschlägt, noch die benötigte maximale Ausführungszeit im Voraus abgeschätzt werden. Automatisches Speichermanagement kann deshalb in zeitgesteuerten Systemen nicht eingesetzt werden.

4.4 Generierung von Pfadinformationen

4.4.1 Problemstellung bei der Generierung von Pfadinformationen

Die statische Analyse des Programmcodes reicht im Allg. nicht aus, um die WCET berechnen zu können (siehe Abschnitt 3.2). Die WCET-Analyse erfordert zusätzlich Informationen über die Ausführungshäufigkeit von Grundblöcken. Um möglichst enge Schranken der WCET berechnen zu können, benötigt die WCET-Analyse darüber hinaus Informationen über Abhängigkeiten zwischen Grundblöcken. Diese Informationen müssen aus dem Modell generiert werden. Auf Modellebene sind jedoch meist sehr abstrakte Informationen vorhanden, während die benötigten Pfadinformationen sehr konkrete Informationen auf Assemblercode-Ebene sind (siehe Abbildung 4.8). Die Lücke zwischen den Abstraktionsebenen muss auf geeignete Weise ge-

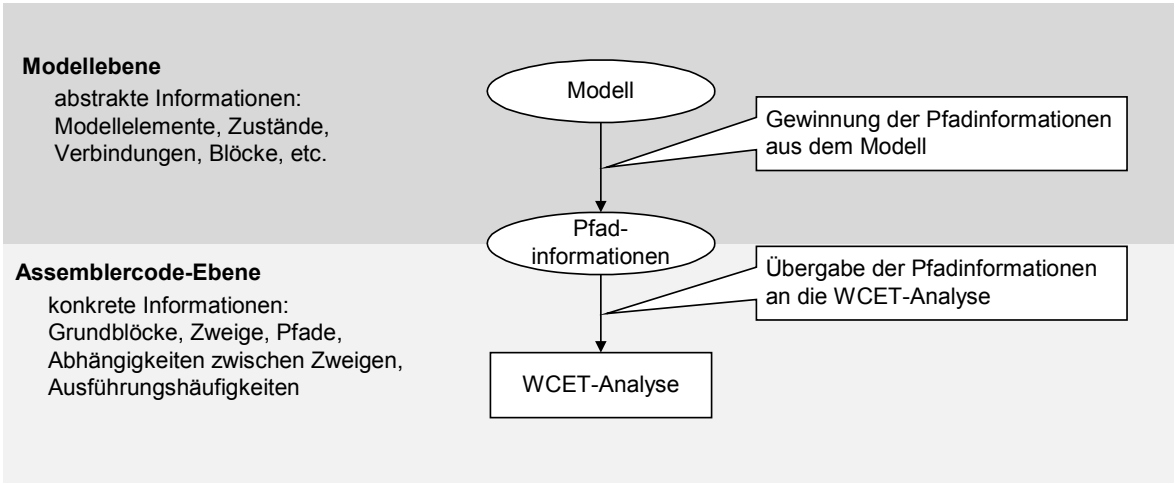


Abbildung 4.8: Problemstellungen bei der Generierung von Pfadinformationen

schlossen werden. Dazu müssen aus den abstrakten Informationen auf Modellebene die konkreten Informationen für die WCET-Analyse erzeugt werden. Dabei können zwei Problemstellungen identifiziert werden: die Gewinnung der Pfadinformationen aus dem Modell und die Übergabe der Pfadinformationen an die WCET-Analyse.

4.4.2 Gewinnung der Pfadinformation aus dem Modell

Die Gewinnung der Pfadinformation aus dem Modell kann auf zwei Arten geschehen. Einerseits kann die Analyse des Modells durch ein zusätzliches Werkzeug erfolgen, andererseits kann versucht werden, den bestehenden Codegenerator um die Ermittlung und Bereitstellung der benötigten Pfadinformationen zu erweitern. Beide Möglichkeiten werden nachfolgend diskutiert.

Gesonderte Analyse des Modells durch ein zusätzliches Werkzeug
Die Voraussetzung für die gesonderte Analyse des Modells ist eine offene Schnittstelle zum Modell. Problematisch wird die gesonderte Analyse des Modells, wenn teilweise nur grafische Beschreibungen in der Modell-Datei enthalten sind und keine semantischen Informationen, die direkt verwertet werden können. Die grafischen Beschreibungen beschränken sich oftmals auf die Position von Modellelementen und deren Verbindungen untereinander. Die eigentliche Semantik ist im Codegenerator, der die Modell-Datei analysiert, verborgen. Die Analyse dieser grafischen Beschreibung und eine korrekte Deutung der Semantik ist im Allg. sehr aufwändig und nur sehr schwer zu realisieren. Die gesonderte Analyse des Modells durch ein zusätzliches Werkzeug eignet sich somit nur, wenn die Semantik des Modells exakt definiert und auch bekannt ist.

Erweiterung des Codegenerators
Die Erweiterung des bestehenden Codegenerators und die Gewinnung der notwendigen Pfadinformationen direkt bei der Codegenerierung hat den Vorteil, dass das Modell nicht gesondert analysiert werden muss. Der Codegenerator liefert alle notwendigen Informationen, die dann nur noch für die neuen Zwecke erweitert werden müssen. Hierfür sind jedoch ein offenes veränderbares Codegenerierungskonzept und offen gelegte Codegenerierungsvorschriften Voraussetzung. Bei den Erweiterungen muss jedoch sehr genau darauf geachtet werden, dass die Veränderungen nicht zu unbeabsichtigten Seiteneffekten führen, die eine fehlerhafte Codegenerierung zur Folge haben.

4.4.3 Übergabe der Pfadinformation an die WCET-Analyse

Für die Übergabe der Pfadinformationen von der Modellebene zur WCET-Analyse bieten sich ebenfalls zwei Möglichkeiten an. Die Übergabe der Pfadinformationen kann zusammen mit dem generierten Code oder separat über eine zusätzliche Schnittstelle erfolgen. Welche Möglichkeit verwendet werden kann, hängt von den Eigenschaften des Software-Entwicklungswerkzeugs ab.

Übergabe zusammen mit dem generierten Code

Die Übergabe der Pfadinformationen zusammen mit dem generierten Code entspricht einer Übertragung des Ansatzes von Puschner und Koza [PuKo89], die Pfadinformationen direkt auf Quellcode-Ebene den entsprechenden Quellcode-Anweisungen zuzuordnen. Eine bestehende Programmiersprache um zusätzliche Konstrukte zu erweitern, wie in [PuKo89] vorgeschlagen, ist jedoch nicht praktikabel. Die gängigen Standard-Compiler lassen eine solche Erweiterung des Sprachumfanges nicht zu.

Eine mögliche Lösung ist es, die Pfadinformation in Kommentarzeilen an die korrespondierenden Quellcode-Anweisungen anzuheften. Das Anheften nach der Gewinnung der Pfadinformation aus dem Modell kann entweder durch den Codegenerator oder durch ein zusätzliches Werkzeug erfolgen. Die Quellcode-Kommentare werden von den gängigen Compilern für Mikrocontroller nicht verändert und sind in der Assemblercode-Datei wiederzufinden. Die Zuordnung der Pfadinformation vom Quellcode zum Assemblercode erfolgt somit durch den Compiler, der die Quellcode-Konstrukte den resultierenden Assembler-Instruktionen zuordnet (siehe Abbildung 4.9).

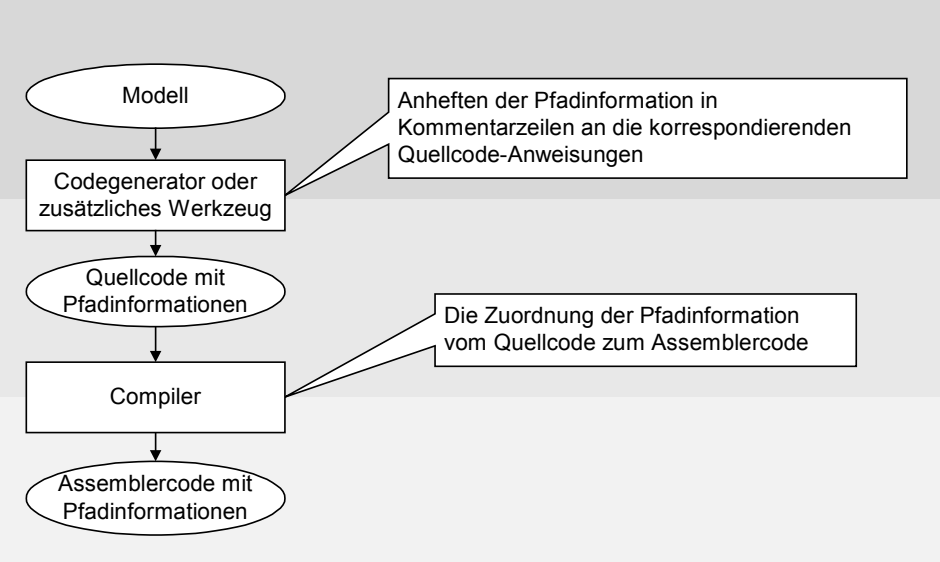


Abbildung 4.9: Zuordnung der Pfadinformationen zu den Assembler-Instruktionen

Die WCET-Analyse kann bei der Analyse des Assemblercodes die angehefteten Informationen mit aufnehmen und dem Assemblercode zuordnen. Dieser Ansatz hat den Vorteil, dass die aufwändige Zuordnung der Pfadinformation durch die WCET-Analyse entfällt, da die Pfadinformation schon durch den Compiler den entsprechenden Assembler-Instruktionen zugeordnet wird.

Direkte Übergabe über eine zusätzliche Schnittstelle
Die direkte Übergabe der Pfadinformation an die WCET-Analyse über eine zusätzliche Schnittstelle entspricht einer Übertragung des Ansatzes der Information Description Language – IDL von Park [Park92], mit deren Hilfe getrennt vom eigentlichen Code dessen Ausführungsverhalten spezifiziert werden konnte. Sie bietet sich an, wenn die Gewinnung der Pfadinformation über die gesonderte Analyse des Modells mit Hilfe eines zusätzlichen Werkzeugs erfolgt. In diesem Fall müsste ansonsten die Pfadinformation an entsprechender Stelle im Quellcode angeheftet werden. Um jedoch die Pfadinformationen direkt über die Schnittstelle zu übergeben, muss bereits auf Modellebene eine Möglichkeit bestehen, eine Zuordnung der Pfadinformationen zu den später generierten Zweigen oder Grundblöcken im Kontrollflussgraf durchzuführen. Dazu bedarf es erkennbarer und zuordenbarer Konstrukte oder Anweisungen, die auch auf Assemblercode-Ebene eindeutig identifizierbar sind. Bei Funktionsaufrufen ist dies beispielsweise der Fall, da sie auf Assemblercode-Ebene durch eine *call*-Instruktion mit eindeutiger Bezeichnung des Funktionsnamens gekennzeichnet sind.

4.5 Zuordnung der WCET-Analyse-Ergebnisse zu Modellelementen

Zur Unterstützung des modellbasierten Entwurfs müssen dem Anwender sowohl integrale Informationen über die WCET des Codes zur Initialisierung, Terminierung und zyklischen Ausführung des Modells als auch die Beiträge der Modellelemente zur Ausführungszeit mitgeteilt werden. Die integralen Informationen können relativ einfach ermittelt werden. Sie sind die Summe der Ausführungszeiten von Quellcode-Funktionen. Diese Ausführungszeiten werden bei den bekannten WCET-Analyse-Ansätzen bereitgestellt.

Die Analyse der Ausführungszeiten von Modellelementen stellt eine größere Schwierigkeit dar. Für die Modellelemente gibt es kein korrespondierendes Beschreibungsmittel im Kontrollflussgraf. Sie tragen im Allg. zu vielen Codeteilen in mehreren Quellcode-Funktionen und auch in mehreren Grundblöcken bei. Darüber hinaus können sich in einem Grundblock die Beiträge mehrerer Modellblöcke wiederfinden (siehe Abschnitt 4.1.4). Es muss deshalb eine Zuordnung der Ausführungszeiten von Grundblöcken und Teilen von Grundblöcken auf Assemblercode-Ebene zu den Modellelementen auf Modellebene erfolgen (siehe Abbildung 4.10). Dazu muss der WCET-Analyse bereits von der Modellebene mitgeteilt werden, welche Modellelemente für welche Codesequenzen verantwortlich sind, damit die WCET-Analyse den Modellblöcken Ausführungszeiten zuordnen kann.

Verschmelzen alle Modellelemente zu einer vollkommen neuen Programmstruktur, ist eine Zuordnung der Beiträge einzelner Modellelemente nicht mehr möglich. Die Ausführungszeit der resultierenden Programmstruktur muss als integrale Information dargestellt werden.

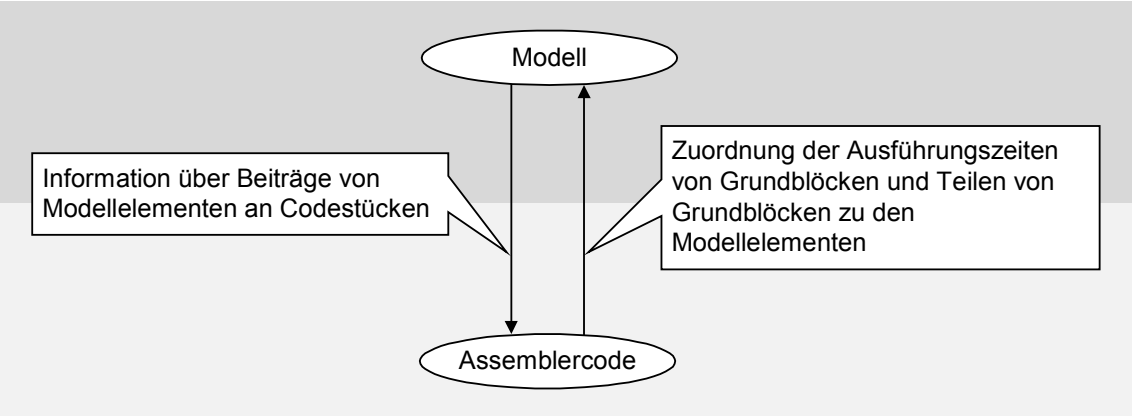


Abbildung 4.10: Zuordnung der Ausführungszeiten zu Modellelementen

4.6 Erweiterung bekannter WCET-Analyse-Verfahren für die modellbasierte Analyse

Für die modellbasierte WCET-Analyse von generiertem Code können, wie in Abschnitt 3.3.2 festgestellt, bestehende Ansätze herangezogen werden. Die modellbasierte WCET-Analyse hat jedoch Auswirkungen auf die eingesetzten WCET-Analyse-Verfahren. Abhängig von den Eigenschaften des Software-Entwicklungswerkzeugs ist ein bestimmter Ansatz erforderlich, bzw. es werden Anpassungen und Erweiterungen notwendig.

Anpassung der statischen WCET-Analyse an das Software-Entwicklungswerkzeug
Die bekannten WCET-Analyse-Ansätze erfordern unterschiedliche Restriktionen beim Sprachumfang des generierten Codes. Welche Konstrukte erlaubt werden können, hängt von den zusätzlichen Informationen über das Ausführungsverhalten des Codes ab, die im Modell verfügbar sind. Die statische Analyse muss deshalb individuell an das Software-Entwicklungswerkzeug angepasst werden.

Berechnungsverfahren zur Bestimmung des Worst-Case-Pfades
Die Auswahl des Berechnungsverfahrens hängt davon ab, welche Pfadinformationen vom Software-Entwicklungswerkzeug verfügbar sind:

- Sind explizite Pfadinformationen verfügbar, die einen ausführbaren Pfad exakt beschreiben, eignet sich die Suche im Kontrollflussgraf nach Stappert und Altenbernd [StAl97]. Die bekannten ausführbaren Pfade können nacheinander durchlaufen und der mit der größten Ausführungszeit ausgewählt werden.

- Sind nur implizite Informationen über Relationen von Zweigen bzw. Grundblöcken untereinander bekannt, ist die Implicit Path Enumeration nach Li und Malik [LiMa99] zweckmäßig. Die Pfade brauchen nicht explizit beschrieben zu werden, sondern implizit in Form eines linearen Optimierungsproblems, das durch ein Optimierungswerkzeug maximiert wird.

Verarbeitung der Beiträge von Modellelementen
Informationen über die Beiträge von Modellelementen an Codestücken werden von bisher bekannten WCET-Analyse-Verfahren nicht verarbeitet. Diese Funktionalität muss deshalb zusätzlich realisiert werden. Die Kommentare zur Kennzeichnung der Beiträge von Modellelementen an Codestücken müssen von der statischen Analyse erkannt und verarbeitet werden. Finden sich in einem Grundblock die Beiträge mehrerer Modellblöcke wieder, so müssen die Grundblöcke in Subblöcke unterteilt werden, denen genau ein Modellelement zugeordnet ist. Anschließend muss die WCET für jeden Subblock getrennt bestimmt werden. Der Aufwand ist sehr stark abhängig von der Komplexität der Hardwarearchitektur. Enthält die Hardwarearchitektur Pipelines, wie es bei dem im Rahmen der vorliegenden Arbeit beispielhaft herangezogenen Mikrocontroller der Fall ist, so muss die WCET jedes Grundblocks durch die Simulation der enthaltenen Instruktionen ermittelt werden. Bei der Simulation müssen die Überlappungen von nachfolgenden Grundblöcken in der Pipeline berücksichtigt werden. Die Subblöcke können dabei durchgehend simuliert werden, da sie stets zusammen ausgeführt werden. Nach erfolgter Bestimmung des Worst-Case-Pfades müssen die Beiträge der einzelnen Modellelemente an der WCET aufsummiert werden.

4.7 Ablauf der modellbasierten WCET-Analyse

Der vollständige Ablauf der modellbasierten WCET-Analyse ist in Abbildung 4.11 anhand eines einfachen Beispiels dargestellt. Im Software-Entwicklungswerkzeug wird im Modellierungseditor ein Modell erstellt, aus dem im Beispiel ein Ausschnitt, bestehend aus drei Modellelementen (ME1, ME2 und ME3), herausgegriffen ist. Jedes dieser Modellelemente leistet einen bestimmten Beitrag zum Code und somit zu dessen Ausführungszeit. Das Software-Entwicklungswerkzeug generiert Code aus dem Modell heraus und steuert Pfadinformationen und Informationen über Beiträge von Modellelementen an Codestücken bei. In diesem Beispiel sei auf Modellebene bekannt, dass sich der Zweig A und der Zweig B gegenseitig ausschließen. Dazu wird der generierte Code mit entsprechenden Zweigvariablen versehen.

Anschließend werden die drei Phasen der WCET-Analyse durchlaufen. Zunächst führt die Strukturanalyse eine lexikalische Analyse des Assemblercodes durch, teilt ihn in Grundblöcke und Zweige auf und erstellt einen Kontrollflussgraf. Zweigen werden gegebenenfalls die Zweigvariablen zugeordnet (wie beispielweise Zweig A und Zweig B). Aus der generierten Information über die Beiträge von Modellelementen an Codestücken werden den Grundblöcken die Beiträge der Modellelemente zugeordnet. Enthält ein Grundblock Beiträge mehrerer Modellelemente (wie beispielsweise der oberste Grundblock im Kontrollflussgraf), wird der Grundblock in Subblöcke aufgeteilt, die jeweils einem Modellelement zugeordnet werden. Die Grundblock-Analyse bestimmt anschließend die WCET eines jeden Grundblocks getrennt. Die jeweiligen Beiträge der Subblöcke zur Ausführungszeit des Grundblocks müssen dazu getrennt ermittelt und den Modellelementen zugeordnet werden. Als Ergebnis der Grundblock-Analyse

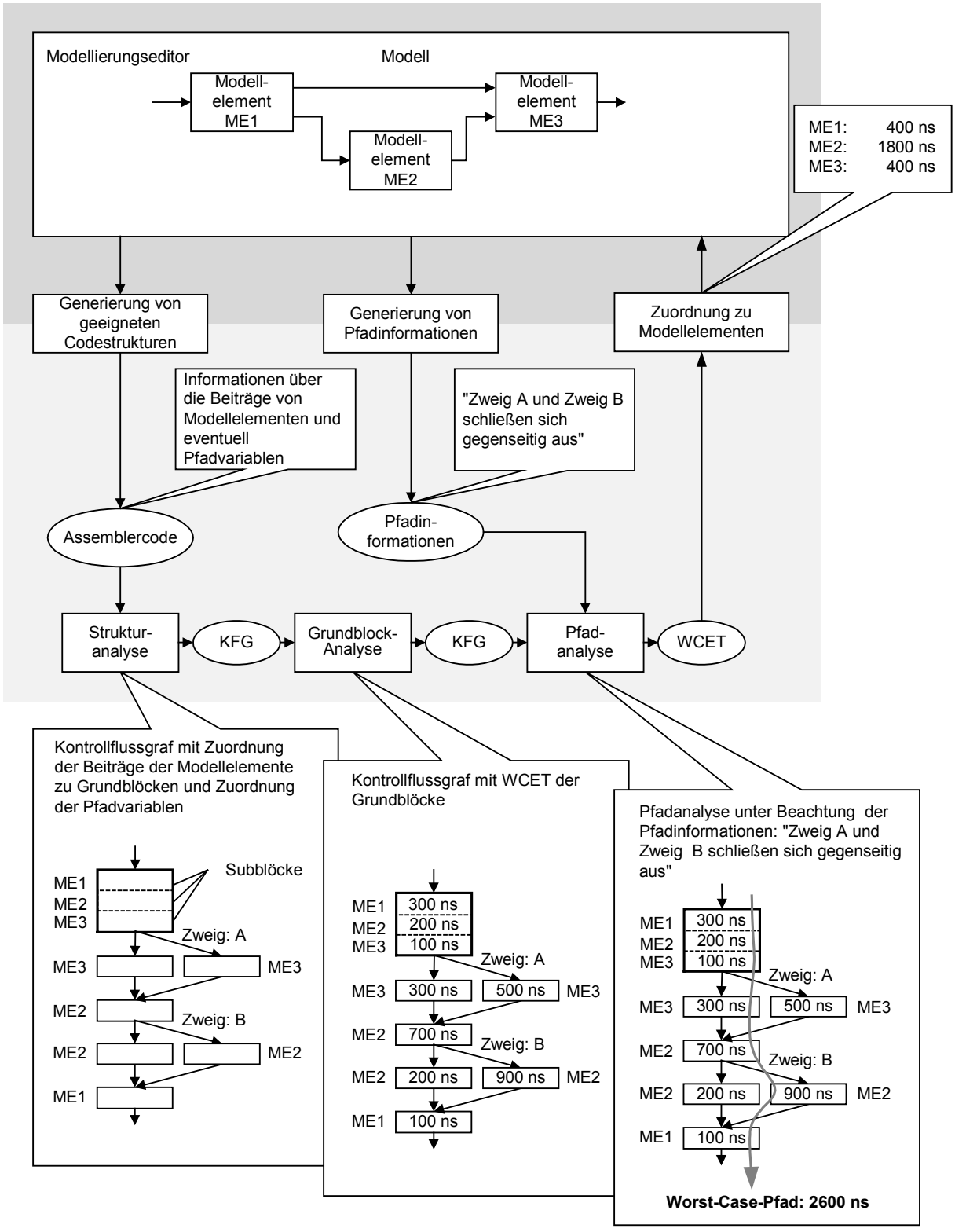


Abbildung 4.11: Veranschaulichung des Ablaufs der modellbasierten WCET-Analyse mit Beispielergebnissen

entsteht ein Kontrollflussgraf, der die WCET der Grundblöcke enthält (wie beispielsweise 300 ns für das Modellelement ME1 im obersten Grundblock).

Die Pfadanalyse bestimmt anschließend den längsten ausführbaren Pfad im Kontrollflussgraf. Die Ausführungszeit eines Pfades wird durch das Summieren der WCET der Grundblöcke des Kontrollflussgrafen bestimmt. Dabei werden die mitgelieferten Pfadinformationen dazu genutzt, nicht ausführbare Pfade auszuschließen. Im Beispiel sind Zweig A und Zweig B nicht zusammen ausführbar. Da die Ausführungszeit im Zweig B größer ist als im Zweig A, durchläuft der Worst-Case-Pfad den Zweig B. Er besitzt eine Ausführungszeit von 2600 ns.

Nachdem der Worst-Case-Pfad ermittelt wurde, können die Beiträge der einzelnen Modellelemente zu diesem Worst-Case-Pfad summiert und an die Modellebene des Software-Entwicklungswerkzeugs übergeben werden. Für das Modellelement ME2 ergibt sich beispielsweise eine Ausführungszeit von 1800 ns im Worst-Case Pfad, für ME3 400 ns.

4.8 Kriterien zur Bewertung der Übertragbarkeit des Konzepts

In diesem Abschnitt werden Kriterien definiert, anhand derer Software-Entwicklungswerkzeuge bezüglich der Übertragbarkeit des erstellten Konzepts bewertet werden können. Es kann zwischen zwei Arten von Kriterien unterschieden werden; einerseits Kriterien zur Bewertung der Software-Entwicklungswerkzeuge, andererseits Kriterien zur Bewertung des generierten Codes.

Eigenschaften des Software-Entwicklungswerkzeugs
Für die Umsetzung des Konzepts müssen die nachfolgenden Kriterien erfüllt sein. Die Generierung von geeigneten Codestrukturen, die Generierung von Pfadinformationen und die Generierung von Beiträgen der Modellelemente an Codestücken erfordern entweder eine offen gelegte Schnittstelle zum Modell oder einen offen gelegten Codegenerator. Daraus ergeben sich folgende Kriterien:

Kriterium 1: Ist eine offen gelegte Schnittstelle zum Modell vorhanden und enthält das Modell verwertbare semantische Informationen?
Kriterium 2: Ist ein offen gelegter Codegenerator vorhanden?

Die Bibliotheksfunktionen der Modellelemente müssen ebenfalls offen gelegt sein, damit sie verändert bzw. mit Pfadinformationen versehen werden können:

Kriterium 3: Sind offen gelegte Bibliotheksfunktionen der Modellelemente vorhanden?

Für die Darstellung der Ergebnisse auf der Modellebene ist eine offene Schnittstelle zur Oberfläche wünschenswert.

Kriterium 4: Ist eine offen gelegte Schnittstelle zur Oberfläche vorhanden?

Eigenschaften des Codes

Die Eigenschaften des Codes geben Aufschluss darüber, auf welche Art und Weise das Konzept übertragen werden kann. Die Kriterien 5 und 6 decken den Aspekt *geeignete Codegenerierung* ab:

Kriterium 5: Sind die möglichen Pfade des generierten Codes statisch analysierbar?
Kriterium 6: Sind Änderungen am generierten Code notwendig?

Die folgenden Kriterien gehen auf den Aspekt *Generierung von Pfadinformationen* ein:

Kriterium 7: Sind Pfadinformationen notwendig, um die Analyse zu ermöglichen?
Kriterium 8: Führen zusätzliche Pfadinformationen zu einer Verbesserung der oberen Schranke der WCET?

In diesem Kapitel wurde ein Konzept der modellbasierten WCET-Analyse erarbeitet, das die WCET-Analyse von modellbasiert erstellter Software ohne zusätzliches Eingreifen des Anwenders ermöglicht. Die WCET-Analyse der Modelle erfolgt hierbei auf einer hohen Abstraktionsebene. Die wesentliche Grundidee des Konzepts ist, die Codegenerierung so anzupassen und zu verändern, dass analysierbarer Code generiert wird und zusätzlich Informationen über das dynamische Ausführungsverhalten des Codes erzeugt werden. Die ermittelten WCET-Analyse-Ergebnisse werden wiederum den in Software-Entwicklungswerkzeugen verwendeten Modellelementen zugeordnet. Zur WCET-Analyse können hierbei aus der Literatur bekannte Ansätze herangezogen werden. Die Umsetzung auf konkrete Software-Entwicklungswerkzeuge ist abhängig von deren Eigenschaften. Dazu wurden verschiedene Realisierungsmöglichkeiten untersucht. Anhand der erarbeiteten Kriterien kann die Übertragbarkeit des erstellten Konzepts der modellbasierten WCET-Analyse auf Software-Entwicklungswerkzeuge bewertet werden. Das Konzept der modellbasierten WCET-Analyse stellt die Basis für die Entwicklung zeitgesteuerter Systeme für By-Wire-Systeme dar, wozu im nächsten Kapitel ein Entwicklungsprozess erarbeitet wird. Im Anschluss daran wird in Kapitel 6 ein zugehöriges praxistaugliches Werkzeugkonzept vorgestellt.

5 Entwicklungsprozess für zeitgesteuerte Systeme

In diesem Kapitel wird ein Entwicklungsprozess für zeitgesteuerte Systeme erarbeitet. Das beschriebene Vorgehen bildet die Basis für das anschließend vorgestellte Werkzeugkonzept. Der Entwicklungsprozess basiert auf dem in der Automobilindustrie etablierten Vorgehen nach dem V-Modell [BrDr95]. Dazu wird zunächst ein Beschreibungsmittel zur Planung zeitgesteuerter Systeme definiert und anschließend die generelle Vorgehensweise der schrittweisen Verfeinerung von Zeitanforderungen und deren Nachweis erarbeitet. Abschließend folgt die Darstellung der einzelnen Phasen des neuen Entwicklungsprozesses für zeitgesteuerte Systeme. Primär werden die zusätzlichen Aktivitäten zur Entwicklung von zeitgesteuerten Systemen beschrieben. Herkömmliche Entwicklungsschritte zur Entwicklung der Anwendungssoftware werden erwähnt, wenn es die Beschreibung der zeitlichen Aktivitäten erfordert. Für die Entwicklung von By-Wire-Systemen sind darüber hinaus Aktivitäten aus dem Bereich der Fehleranalyse, Zuverlässigkeitsanalyse, Fehlermodellierung und Fehlerinjektion notwendig, die hier jedoch nicht weiter betrachtet werden (siehe dazu [Fuch96, Schu98, Hede98, HeSe98]).

5.1 Beschreibungsmittel für den zeitlichen Ablauf von Aktionen

Zur Planung der verteilten Anwendung im zeitgesteuerten System wird ein geeignetes Beschreibungsmittel für den zeitlichen Ablauf von Aktionen – Berechnungsaktivitäten und Kommunikationsaktivitäten – benötigt. In der vorliegenden Arbeit wird der Ansatz der *Real-Time Transactions* [Noss97] aufgegriffen. Es wird zwischen lokalen und globalen Kommunikationsbeziehungen zur Beschreibung knotenlokaler und netzwerkweiter zeitlicher Abläufe von Aktionen unterschieden.

Globale Kommunikationsbeziehung (GKB): Eine globale Kommunikationsbeziehung ist eine zeitlich beschränkte Kette von lokalen Kommunikationsbeziehungen und netzwerkweiten Kommunikationsaktivitäten, die vom verteilten System bearbeitet werden. Sie beginnt mit einem Eingangssignal des technischen Prozesses und endet mit einem Ausgangssignal an den technischen Prozess.

Die zeitlichen Beschränkungen der globalen Kommunikationsbeziehung sind die Periode, in der sie ausgeführt wird, und die End-to-End Deadline. Die Periode ist das Intervall zwischen zwei aufeinander folgenden Eingangssignalen. Die End-to-End Deadline ist die maximale Zeit, die für die Berechnungs- und Kommunikationsaktivitäten benötigt werden darf. Sie wird ausschließlich durch die Dynamik des technischen Prozesses bestimmt. Die End-to-End Deadline

kann größer sein als die Periode. Somit ist möglich, dass eine neue Kommunikationsbeziehung beginnt, bevor die vorherige zu Ende ist. Die Berechnungs- und Kommunikationsaktivitäten besitzen selbst Deadlines, innerhalb derer die Aktivitäten beendet sein müssen. Bei der Aufteilung der globalen Kommunikationsbeziehung in lokale Kommunikationsbeziehungen und Kommunikationsaktivitäten wird die End-to-End Deadline aufgeteilt. Die Deadlines geben den Zeitpunkt an, zu dem die Aktivität beendet sein muss. Sie beziehen sich jeweils auf den Startzeitpunkt der entsprechenden Aktivität. In Abbildung 5.1 ist ein Beispiel mit drei lokalen Kommunikationsbeziehungen dargestellt. Die End-to-End Deadline ist aufgeteilt in die Deadlines der lokalen Kommunikationsbeziehungen D_{LKB1}, D_{LKB2}, D_{LKB3} und Deadlines der netzwerkweiten Kommunikationsaktivitäten D_{KA1} und D_{KA2}.

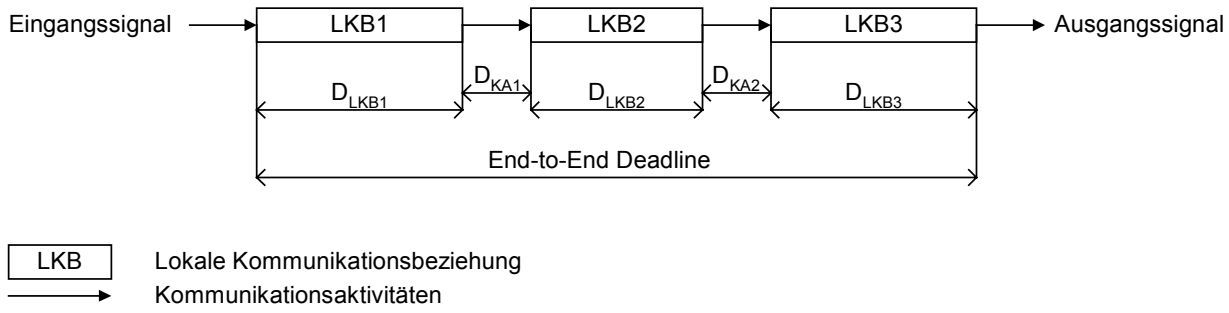


Abbildung 5.1: Definition einer globalen Kommunikationsbeziehung

Lokale Kommunikationsbeziehung (LKB): Eine lokale Kommunikationsbeziehung ist eine zeitlich beschränkte Kette von Tasks und knotenlokaler Kommunikation über Signale. Sie beginnt mit einem Eingangssignal und endet mit einem Ausgangssignal.

In Abbildung 5.2 ist unter Fortführung des Beispiels in Abbildung 5.1 eine lokale Kommunikationsbeziehung dargestellt. Sie entspricht einer Berechnungsaktivität einer globalen Kommunikationsbeziehung, die einem Knoten zugeordnet wurde (hier Knoten 2). Die Deadline der lokalen Kommunikationsbeziehung bestimmt die Deadlines aller Tasks. Eine globale Kommunikationsbeziehung schreibt somit der lokalen Kommunikationsbeziehung die Zeitanforderungen vor. Die lokale Kommunikationsbeziehung wiederum gibt den zeitlichen Rahmen vor, in dem Tasks ausgeführt werden müssen. Bei der Festlegung der Zeitanforderungen an Tasks muss beachtet werden, dass das Auftreten möglicher Interrupts zu Verzögerungen der Ausführung von Tasks führen kann.

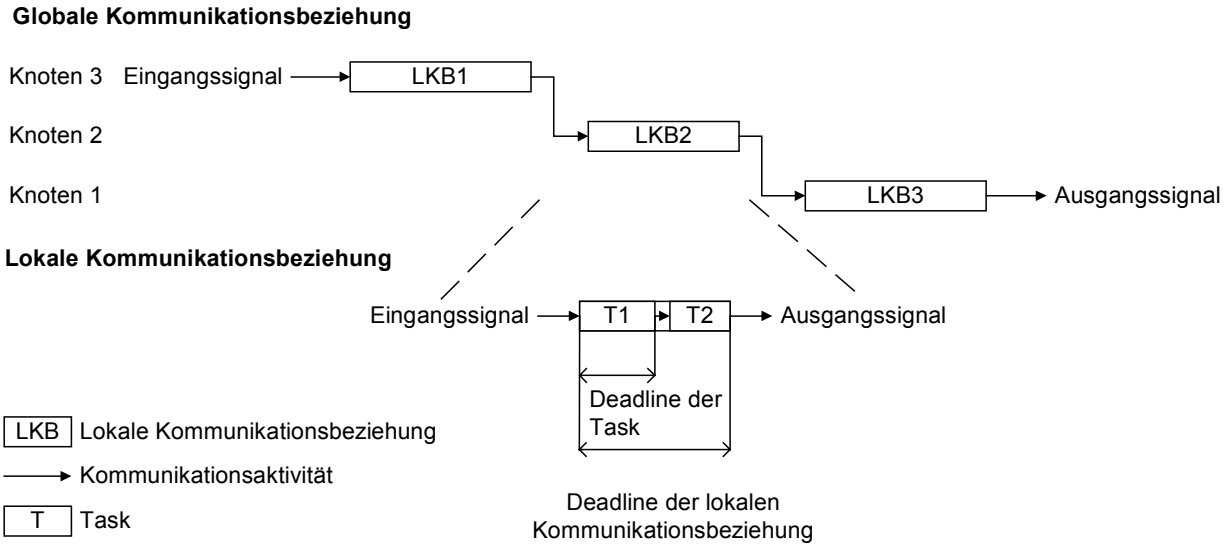


Abbildung 5.2: Definition einer lokalen Kommunikationsbeziehung

5.2 Einordnung zusätzlicher Aktivitäten in den bestehenden Entwicklungsprozess

Aus der Analyse der Anforderungen an den Entwicklungsprozess für zeitgesteuerte Systeme wird ersichtlich, dass die zusätzlichen Planungsschritte in den bestehenden Entwicklungsprozess eingebunden werden müssen. Die Planung des zeitgesteuerten Systems durch die Spezifikation und Verfeinerung der Abläufe von Aktionen muss parallel zu der herkömmlichen Verfeinerung der funktionalen Anforderungen an das System erfolgen, da für die Entwicklung von Teilsystemen durch den Zulieferer präzise definierte Schnittstellen erforderlich sind.

Der bestehende Entwicklungsprozess

Die Softwareentwicklung in der Automobilindustrie erfolgt in einer Arbeitsteilung zwischen Automobilhersteller und Zulieferern. Die Entwicklung der Software zur Serienreife erfolgt hierbei typischerweise in den drei Phasen: *Funktionsentwicklung*, *Funktionsintegration* und *Reifegradentwicklung* [Ruh00]. In der Funktionsentwicklung werden einzelne Fahrzeugfunktionen noch getrennt entwickelt und anschließend in der Funktionsintegration zusammengefügt. Am Ende der Reifegradentwicklung hat die Produktqualität schließlich den vorgegebenen Reifegrad erreicht. Der Grundbestandteil des Entwicklungsprozesses ist ein abgeleitetes V-Modell [BSW00]. Im Laufe der Entwicklung der Software wird das V-Modell mehrfach durchlaufen. Die Phasen des V-Modells [BrDr95] sind in Abbildung 5.3 dargestellt. In der ersten Phase *Analyse und Definition* werden die Anforderungen an das zu entwickelnde System analysiert und festgeschrieben. Dabei fließen die Erfahrungen ein, die bereits in der vorhergehenden Technologiephase gesammelt wurden. Der Fahrzeughersteller führt den *Grobentwurf* durch, wobei die Systemarchitektur festgelegt und das Gesamtsystem in Systemkomponenten unterteilt wird. Der sich anschließende *Feinentwurf* wird vorwiegend beim Zulieferer durchgeführt.

Auf Basis des *Feinentwurfs* erfolgt die *Implementierung* des Programmcodes. Anschließend werden die erstellten Komponenten schrittweise integriert. Zunächst wird die Integration von Programmteilen zu größeren Einheiten vorgenommen (*Komponenten-Integration*). Anschließend werden diese Einheiten in die Steuergeräte integriert. In der abschließenden Phase der *System-Integration*, die wieder beim Fahrzeughersteller erfolgt, werden die Steuergeräte in ein Netzwerk von Steuergeräten integriert. Dabei wird die Validierung des Systems durchgeführt. Während des Durchlaufens des V-Modells sind Rückschritte möglich. Die Pfeile in Abbildung 5.3 deuten an, dass sowohl zwischen zwei aufeinander folgenden als auch zwischen gegenüberliegenden Phasen Rückschritte möglich sind.

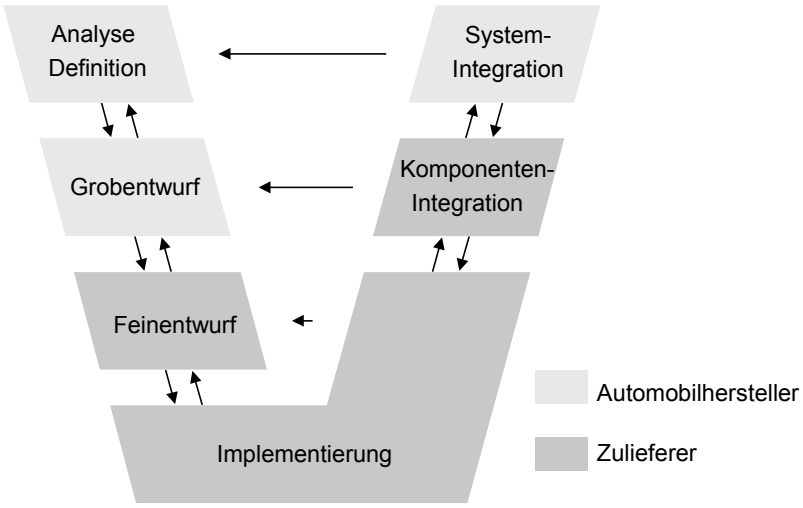


Abbildung 5.3: Vereinfachter Software-Entwicklungsprozess in der Automobilindustrie in Form eines V-Modells

Einordnung der zusätzlichen Aktivitäten

In Abbildung 5.4 sind die zusätzlichen Aktivitäten für die Entwicklung zeitgesteuerter Systeme als Erweiterung des bestehenden Entwicklungsprozesses den entsprechenden Phasen zugeordnet. Zunächst erfolgt in der ersten Phase (Analyse und Definition) die Spezifikation der Funktionen, die die verteilte Anwendung zu erfüllen hat, in Form von globalen Kommunikationsbeziehungen. Deren globale Zeitanforderungen werden im Laufe der folgenden Phasen (Grobentwurf und Feinentwurf) sukzessive verfeinert: von den Zeitanforderungen an die lokalen Kommunikationsbeziehungen nach erfolgtem Nachrichten-Scheduling (Grobentwurf) bis hin zu Zeitanforderungen an einzelne Tasks auf den Knoten, nach erfolgtem Task Scheduling (Feinentwurf).

Bei dieser Verfeinerung der Zeitanforderungen sind jedoch die Ausführungszeiten der Tasks noch nicht bekannt, da in diesen Phasen noch keine Implementierung der Tasks erfolgt ist. Verlässliche Ausführungszeiten können jedoch nur auf Grundlage des ausführbaren Codes gewonnen werden (siehe Abschnitt 3.2), der erst in der Implementierungsphase erstellt wird.

Die Abläufe im zeitgesteuerten System können somit nicht auf Basis realer Ausführungszeiten geplant werden. Es müssen vielmehr im Vorfeld bestimmte zeitliche Annahmen getroffen werden, die in einem späteren Entwicklungsstadium durch die Zeitanalyse nachgeprüft werden.

Die Planung durch die Definition von Zeitanforderungen wird in den frühen Phasen am linken Ast des V-Modells durchgeführt, während der Nachweis der Einhaltung der Anforderungen in den späteren Phasen am rechten Ast vorgenommen wird. Die Planung erfolgt in einer Top-Down-Vorgehensweise durch die Verfeinerung der Spezifikation von Zeitanforderungen. Demgegenüber erfolgt die Zeitanalyse in umgekehrter Reihenfolge in einer Bottom-Up-Vorgehensweise. Zunächst wird in der Implementierungsphase die WCET-Analyse für die Tasks durchgeführt und überprüft, ob die Tasks ihre Deadlines einhalten können. Im nächsten Schritt wird bei der Komponenten-Integration überprüft, ob der Knoten alle Deadlines der lokalen Kommunikationsbeziehungen einhalten kann. Können die geplanten Zeitanforderungen nicht eingehalten werden, so wird ein horizontaler Rückschritt durchgeführt (siehe Abbildung 5.4 horizontale Pfeile).

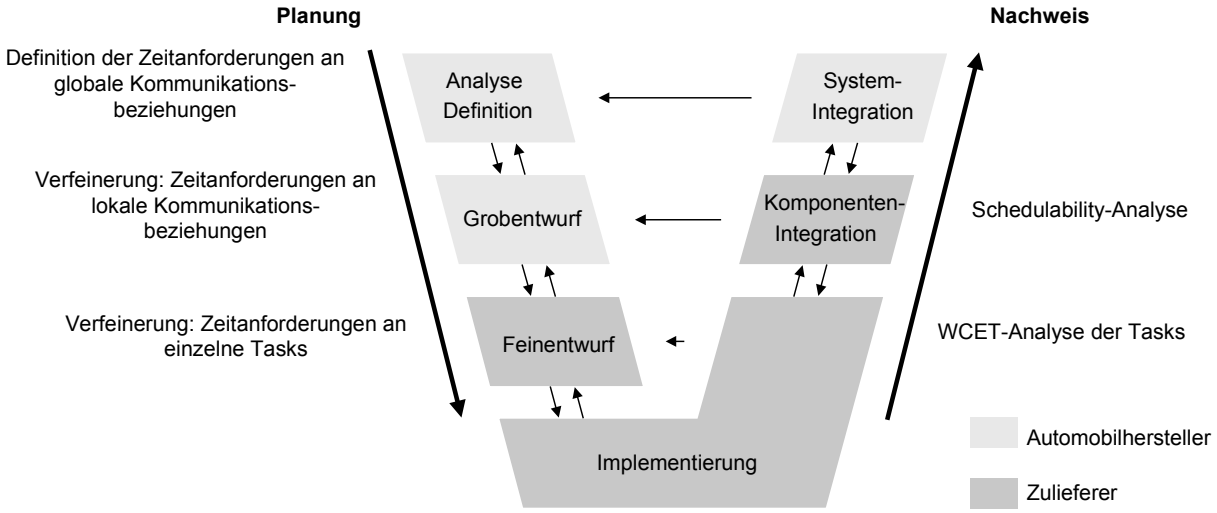


Abbildung 5.4: Zusätzliche Schritte für die Planung und Analyse zeitgesteuerter Systeme im V-Modell

5.3 Phasen des Entwicklungsprozesses für zeitgesteuerte Systeme

5.3.1 Analyse und Definition

Der Entwicklungsprozess beginnt mit der Analyse der Anforderungen und der Definition der verteilten Anwendung unter der Federführung des Automobilherstellers. Dieser legt fest, welche Funktionen (beispielsweise ABS oder ESP) durch das By-Wire-System realisiert werden sollen und welche Anforderungen diese Funktionen an das Elektroniksystem stellen. Anhaltspunkte hierfür liefern Erfahrungen aus früheren Projekten sowie vorhergehenden Technologiephasen.

Neben den funktionalen Anforderungen an das Elektroniksystem müssen die Zeitanforderungen an das Elektroniksystem analysiert werden. Die Zeitanforderungen stammen ausschließlich von den technischen Fahrwerksystemen (Bremssystem, Lenkungssystem, Federungssystem etc.) her, die mit Hilfe der verteilten Anwendung geregelt werden sollen.[12] Diese Anforderungen werden in Form von globalen Kommunikationsbeziehungen und deren Zeiteigenschaften, d. h. End-to-End Deadline und Periode spezifiziert.

5.3.2 Grobentwurf

Der Grobentwurf wird ebenfalls unter der Leitung des Automobilherstellers durchgeführt. Die Architektur des zu realisierenden verteilten Systems wird definiert und es werden Schnittstellen festgelegt, auf deren Basis die Zulieferer in der nächsten Phase mit dem Feinentwurf beginnen können.

Die Definition von Zeitanforderungen in Form von globalen Kommunikationsbeziehungen bildet den Ausgangspunkt für die weitere Spezifikation des zeitgesteuerten Systems. Um die Entwicklungsschnittstelle, den Nachrichten-Schedule (siehe Abschnitt 3.1.4), definieren zu können, müssen folgende Aktivitäten durchgeführt werden:

- Definition der Architektur des verteilten Systems
- Verteilung von Teilfunktionen auf die Knoten und Abschätzung der Bearbeitungszeiten
- Erstellung des Nachrichten-Schedules und Definition von Zeitanforderungen an Knoten

Nach der detaillierten Definition des Funktionsumfanges der verteilten Anwendung kann die Architektur des verteilten Systems definiert werden. Die Anzahl der Steuergeräte ist von vielen Faktoren abhängig. Einerseits ist die redundante Auslegung von Steuergeräten zu berücksichti-

[12] Eine elektronische Lenkung erfordert beispielsweise eine Zykluszeit für die verteilte Regelung von ca. 1 - 3 ms, eine Bremse von 5 - 10 ms.

gen, andererseits Aspekte wie Minimierung des Verkabelungsaufwandes, Bauraum und Verwendung bestehender Steuergeräte aus anderen Baureihen.

Sind die Funktionen der verteilten Anwendung und die Architektur des verteilten Systems bekannt, können die einzelnen lokalen Kommunikationsbeziehungen der globalen Kommunikationsbeziehungen auf die Steuergeräte des Clusters verteilt werden. Bei dieser Tätigkeit müssen die End-to-End Deadlines der globalen Kommunikationsbeziehungen in Deadlines der lokalen Kommunikationsbeziehungen aufgeteilt werden. Die Schwierigkeit in diesem Schritt liegt darin, dass zu diesem Zeitpunkt noch keine maximalen Ausführungszeiten der Implementierungen bekannt sind. Für die Verteilung ist also ein gewisses Maß an Erfahrungswissen notwendig. Können die Deadlines später nicht eingehalten werden, so wird ein weiterer Iterationsschritt im V-Modell eingeleitet.

Nach der Verteilung der lokalen Kommunikationsbeziehungen auf die Knoten kann der Nachrichten-Schedule erstellt werden. Mit der Definition des Nachrichten-Schedules entstehen zusätzlich Zeitanforderungen in Form von Deadlines der lokalen Kommunikationsbeziehungen, die die Steuergeräte einhalten müssen. Diese Schnittstellenspezifikation wird den Zulieferern übergeben.

5.3.3 Feinentwurf

Der lokale Entwurf wird hauptsächlich von den Zulieferern durchgeführt, die die einzelnen Funktionen der Steuergeräte detailliert spezifizieren. Der zeitliche Ablauf von Tasks auf dem Steuergerät wird als Task Schedule festlegt. Als Ausgangspunkt hierfür gelten die Zeitanforderungen an die lokalen Kommunikationsbeziehungen, die der Automobilhersteller dem Zulieferer als Schnittstellenspezifikation vorschreibt. Die Definition des Task Schedules wird schrittweise durchgeführt:

- Spezifikation der auf den Knoten ablaufenden Tasks
- Festlegung der Ausführungszeiten aller Tasks
- Erstellung des Task Schedules

Aus den abstrakten lokalen Kommunikationsbeziehungen werden zusammenhängende Einheiten in Form von konkreten Tasks gebildet, die durch ein Betriebssystem zur Ausführung gebracht werden sollen. Zusätzlich können weitere Tasks für die Bereitstellung von Hilfsfunktionen notwendig werden. Die Deadlines der lokalen Kommunikationsbeziehungen werden dann in Deadlines der Tasks aufgeteilt. Dazu müssen die Ausführungszeiten aller Tasks abgeschätzt werden. Dabei besteht ebenfalls die Schwierigkeit, die maximale Ausführungszeit der Tasks abzuschätzen, denn die konkreten Ausführungszeiten der Tasks stehen erst in der nächsten Phase zur Verfügung. Auf Basis der geschätzten Ausführungszeiten und Deadlines der Tasks wird der zeitliche Ablauf der Tasks geplant. Startzeitpunkte und Deadlines der Tasks werden in einem Task

Schedule festgeschrieben. Der Task Schedule stellt die Grundlage für die weiteren Entwurfsentscheidungen (Art der Programmiersprache, Leistungsfähigkeit der Zielhardware etc.) dar, die in der nächsten Phase getroffen werden müssen.

5.3.4 Implementierung

Die Implementierung der lokalen Anwendungen, die auf den Steuergeräten ausgeführt werden, wird hauptverantwortlich von den Zulieferern durchgeführt. Die lokalen Anwendungen bestehen bei By-Wire-Systemen zum Großteil aus Regelungs- und Steuerungssoftware, deren Entwicklung in zunehmendem Maße unter der Verwendung von Software-Entwicklungswerkzeugen erfolgt [BSW00]. Da die Software in einem zeitgesteuerten System ausgeführt wird, werden die Latenzzeit-Analyse von am Regler beteiligen Signalen und die WCET-Analyse der Tasks notwendig. Die Latenzzeit-Analyse muss zur Parametrisierung der Regler vor der Codegenerierung durchgeführt werden. Die WCET-Analyse kann erst nach der Codegenerierung auf Basis des endgültigen Codes für einen bestimmten Prozessor erfolgen. Sie sollte kontinuierlich während der Implementierung der Tasks durchgeführt werden, um zu überprüfen, ob die Tasks die geplanten Deadlines einhalten können.

Kann eine Task ihre Deadline nicht einhalten, dann muss der Task Schedule angepasst werden. Dazu ist ein Rückschritt in den Feinentwurf erforderlich. Kann trotz der Veränderung des Task Scheduling die Deadline nicht eingehalten werden, so muss eine effizientere Codierung, Auswahl eines schnelleren Prozessors etc. in Betracht gezogen werden. Ein neuer Grobentwurf würde sich auf alle Beteiligten, Automobilhersteller und Zulieferer, auswirken und sollte deshalb wenn möglich verhindert werden.

5.3.5 Komponenten-Integration

Die Komponenten-Integration erfolgt nach der Implementierung durch den Zulieferer. Die zuvor implementierten Tasks werden in ein Betriebssystem integriert und Konfigurationsdateien für das Betriebssystem erzeugt. Dabei ist mit der Schedulability-Analyse nachzuweisen, dass alle Tasks ihre Deadline einhalten können. In der vorhergehenden Phase wurden die Ausführungszeiten aller Tasks ohne Unterbrechungen ermittelt. In dieser Phase muss durch die Schedulability-Analyse nachgewiesen werden, dass das Steuergerät alle zeitlichen Anforderungen einhalten kann, die in Form von Zeitanforderungen an die lokalen Kommunikationsbeziehungen gestellt wurden. Alle Tasks müssen inklusive aller möglichen Unterbrechungen durch andere Tasks oder Interrupts ihre Deadlines einhalten. Werden nicht alle Zeitanforderungen erfüllt, muss eine Iteration im V-Modell erfolgen und müssen im Grobentwurf nochmals neue Planungen vorgenommen werden.

5.3.6 System-Integration

Bei der System-Integration werden abschließend unter Federführung des Automobilherstellers, zusammen mit den Zulieferern, die Steuergeräte im Cluster integriert und intensive Tests durchgeführt. Im Rahmen der Tests wird überprüft, ob die zu Grunde gelegten globalen Zeitanforderungen zu den gewünschten Ergebnissen führen. Mit den gewonnenen Erkenntnissen kann dann ein weiterer Durchlauf des Entwicklungsprozesses auf dem Weg zur Serienreife stattfinden.

In diesem Kapitel wurde ein Entwicklungsprozess für zeitgesteuerte Systeme vorgestellt. Die zusätzlich erforderlichen Aktivitäten zur Entwicklung zeitgesteuerter Systeme wurden in den bestehenden Entwicklungsprozess eingegliedert. Diese Aktivitäten sind in den frühen Phasen der Entwicklung die Definition von Zeitanforderungen an globale und lokale Kommunikationsbeziehungen sowie der Nachweis der Einhaltung dieser Zeitanforderungen durch die Implementierungen in den späteren Phasen. Die WCET-Analyse spielt hierbei eine zentrale Rolle. Nur wenn es gelingt, die WCET von Software-Entwicklungswerkzeugen generiertem Code zu bestimmen, kann sichergestellt werden, dass sich das entwickelte System in jedem Fall zeitlich korrekt verhält. Basierend auf dem vorgestellten Entwicklungsprozess wird im anschließenden Kapitel eine geeignete Werkzeugunterstützung für die verschiedenen Phasen der Entwicklung zeitgesteuerter Systeme vorgestellt.

6 Werkzeugkonzept für die Entwicklung zeitgesteuerter Systeme

In diesem Kapitel wird ein Werkzeugkonzept für die Entwicklung zeitgesteuerter Systeme vorgestellt. Es berücksichtigt die zusätzlichen Aktivitäten zur Entwicklung zeitgesteuerter Systeme, die im vorherigen Kapitel aufgezeigt wurden. Zunächst wird die Architektur des Werkzeugkonzepts aus den Eigenschaften der zeitgesteuerten Systeme und deren Anforderung an ein Werkzeugkonzept hergeleitet. Anschließend wird die Planung von Abläufen in zeitgesteuerten Systemen durch die Spezifikation von Kommunikationsbeziehungen aufgezeigt. Für den freien Datenaustausch wurde eine Spezifikationssprache entwickelt, deren wesentliche Elemente anschließend vorgestellt werden. Abschließend wird auf die Unterstützung der Zeitanalyse zeitgesteuerter Systeme durch das Werkzeugkonzept eingegangen.

6.1 Architektur des Werkzeugkonzepts für die Entwicklung zeitgesteuerter Systeme

Zeitgesteuerte Systeme besitzen durch das Prinzip der Zeitsteuerung Eigenschaften, die neue Anforderungen an die Entwicklungswerkzeuge stellen. Insbesondere können aufgrund der Zeitsteuerung regelungstechnische Anwendungen auf ein Netzwerk verteilt werden (siehe Abschnitt 2.1.1). Ein Werkzeugkonzept für die Entwicklung zeitgesteuerter Systeme muss daher die Planung von zeitlichen Abläufen, die Zeitanalyse sowie die Entwicklung verteilter regelungstechnischer Anwendungen geeignet unterstützen (siehe Abschnitt 1.3). Die Analyse des Stands der Technik hat gezeigt, dass die bestehenden Werkzeugkonzepte diese Anforderungen nicht ausreichend erfüllen. Als wesentliche Defizite wurden erkannt (siehe Abschnitt 3.3.1):

- Sie bieten keine einfache zielgerichtete Planung zeitlicher Abläufe im verteilten System.
- Sie sehen keine Zeitanalyse vor.
- Sie unterstützen die Entwicklung regelungstechnischer Anwendungen nicht ausreichend.

Das im Rahmen der vorliegenden Arbeit erstellte Werkzeugkonzept zur Entwicklung zeitgesteuerter Systeme ViETTA (**Vi**sual **E**nvironment for **T**ime-**T**riggered **A**rchitectures) wurde unter Beachtung der Anforderungen der zeitgesteuerten Systeme entwickelt und beseitigt diese Defizite. ViETTA unterstützt die zeitlichen Aspekte bei der Entwicklung zeitgesteuerter Systeme, bindet bestehende spezialisierte Werkzeuge ein und bietet Schnittstellen.

Die Planungsdaten des zeitgesteuerten Systems müssen während des Entwicklungszeitraums zwischen Automobilhersteller und Zulieferern ausgetauscht werden. Sie müssen somit unter Umständen von verschiedenen Werkzeugen lesbar sein. ViETTA besitzt deshalb eine offene Architektur mit einer definierten Datenbasis als zentrales Element, die als Schnittstelle zwischen Werkzeugmodulen fungiert (siehe Abbildung 6.1). Die Werkzeugmodule können somit auch von verschiedenen Herstellern stammen. Die ViETTA Datenbasis wurde in Anlehnung an bestehende Standards für ereignisgesteuerte Systeme definiert und um die Beschreibung verteilter zeitgesteuerter Systeme erweitert.

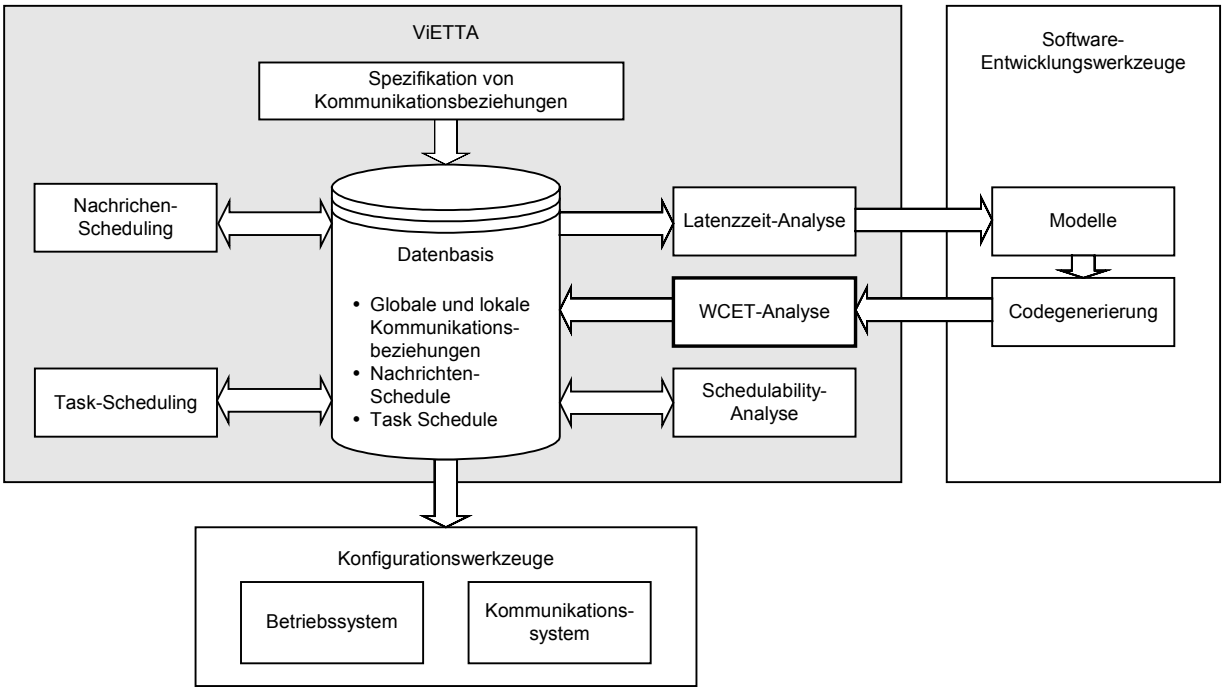


Abbildung 6.1: Architektur des Werkzeugs für die Entwicklung zeitgesteuerter Systeme

Das Vorgehen bei der Entwicklung zeitgesteuerter Systeme wurde in Abschnitt 5.2 folgendermaßen definiert:

- Spezifikation von Zeitanforderungen in Form von Kommunikationsbeziehungen,
- deren schrittweise Verfeinerung parallel zur Verfeinerung funktionaler Anforderungen
- sowie Nachweis der Einhaltung der Zeitanforderungen durch die Zeitanalyse.

ViETTA unterstützt dieses Vorgehen, indem es direkt die Planung des zeitlichen Ablaufs durch die grafische Spezifikation von Kommunikationsbeziehungen, die über der Zeitachse aufgetragen werden, ermöglicht. ViETTA unterscheidet sich hierbei wesentlich von bestehenden Werkzeugen bzw. Werkzeugkonzepten, bei denen eine Vielzahl langwieriger abstrakter Spezifikationsvorgänge notwendig sind, um ein zeitgesteuertes System zu entwerfen (vgl. Abschnitt 3.1).

Durch die Werkzeugmodule Nachrichten-Scheduling und Task Scheduling können im Laufe der späteren Entwicklungsphasen auf Grundlage der ersten Planungsdaten Verfeinerungen vorgenommen werden. Dazu werden die zuvor definierten globalen Kommunikationsbeziehungen in lokale Kommunikationsbeziehungen aufgeteilt und bestimmten Knoten zugeordnet. Auf diese Weise wird die Datenbasis sukzessive erweitert und vervollständigt.

Die Zeiteigenschaften des zeitgesteuerten Systems haben Einfluss auf die verteilten regelungstechnischen Anwendungen. Für die Parametrisierung der Modelle wird deshalb die Latenzzeit von Signalen benötigt. Da sich während des Entwicklungszeitraums die Zeiteigenschaften des zeitgesteuerten Systems ändern können, müssen stets die aktuellen Latenzzeiten verfügbar sein. Dieser wichtige Aspekt wurde bei bestehenden Ansätzen bisher nicht beachtet. Zur Unterstützung der Entwicklung regelungstechnischer Anwendungen besitzt ViETTA eine Schnittstelle zu Software-Entwicklungswerkzeugen, mit denen die zeitgesteuerte Anwendungssoftware entwickelt wird. Aus der Datenbasis können die Latenzzeiten von Signalen analysiert und dem Software-Entwicklungswerkzeug zur Verfügung gestellt werden. Das Software-Entwicklungswerkzeug kann diese Informationen wiederum verwenden, um die Parameter der Modelle automatisch zu aktualisieren.

Die WCET-Analyse spielt bei ViETTA, im Gegensatz zu existierenden Ansätzen, eine herausragende Rolle. Insbesondere wird berücksichtigt, dass die Anwendungssoftware modellbasiert mit Software-Entwicklungswerkzeugen erstellt wird. Nach der Implementierung bzw. Codegenerierung kann aus den erstellten Modellen auf Basis des generierten Codes automatisch die WCET-Analyse durchgeführt werden. Die Ergebnisse der WCET-Analyse können dann über eine Schnittstelle in die Datenbasis eingetragen werden.

Nach erfolgter WCET-Analyse muss auf einfache Weise nachprüfbar sein, ob die neu bestimmten WCET-Analyse-Ergebnisse alle Zeitanforderungen weiterhin erfüllen. Aus diesem Grund ist in der Werkzeugumgebung ViETTA eine Schedulability-Analyse integriert, die Informationen aus der Datenbasis nutzt.

Für bestimmte Implementierungen von Betriebssystemen und Treibern für Kommunikationssysteme wird es zukünftig von den jeweiligen Herstellern spezielle Konfigurationswerkzeuge geben. ViETTA kann als übergeordnetes Planungswerkzeug diesen Konfigurationswerkzeugen aus der Datenbasis Eingangsdaten zur Verfügung stellen. Die von ViETTA geplanten Abläufe stellen Zeitanforderungen dar, die von den Konfigurationswerkzeugen in jedem Fall erfüllt werden müssen. Darüber hinaus können dann weitere Tasks und Nachrichten, die nicht an verteilten Abläufen beteiligt sind, eingeplant werden.

6.2 Planung zeitlicher Abläufe

6.2.1 Spezifikation von Kommunikationsbeziehungen

Die Spezifikation von Kommunikationsbeziehungen erfolgt bei ViETTA grafisch in Form von Ablaufplänen, sog. Gantt-Diagrammen. Die Berechnungs- und Kommunikationsaktivitäten der Kommunikationsbeziehungen sind über der Zeitachse angeordnet. Zur Unterstützung der schrittweisen Verfeinerung der Kommunikationsbeziehungen wurden zwei Darstellungsarten definiert, eine globale und eine knotenbasierte Darstellungsart.

Globale Darstellungsart

Die globale Darstellungsart (siehe Abbildung 6.2) ermöglicht eine erste Spezifikation globaler Kommunikationsbeziehungen (GKB i) und deren Zeitanforderungen. Deren lokale Kommunikationsbeziehungen (LKB i-j) und Kommunikationsaktivitäten werden in ihrer sequenziellen Abfolge über der Zeit angeordnet. Lokale Kommunikationsbeziehungen werden als Balken dargestellt. Die Länge des Balkens entspricht der Deadline der Kommunikationsbeziehung, die Position des Balkens bezüglich der Zeitachse bestimmt den Aktivierungszeitpunkt. Die Kommunikationsaktivitäten werden durch Pfeile gekennzeichnet, die somit eine Abhängigkeitsbeziehung zum Ausdruck bringen. In dieser Darstellungsart lassen sich sehr gut die Relationen zwischen den einzelnen Kommunikationsbeziehungen grafisch darstellen.

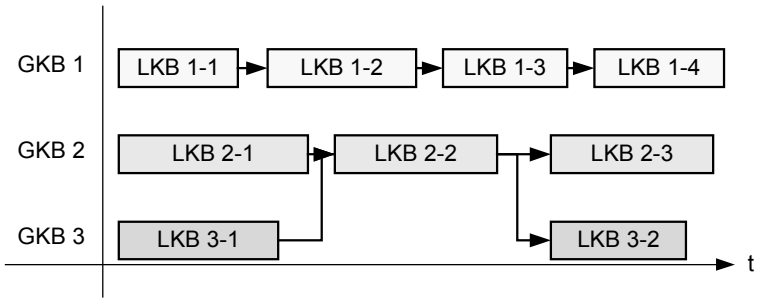


Abbildung 6.2: Globale Darstellungsart von Kommunikationsbeziehungen

Knotenbasierte Darstellungsart

Im Gegensatz zur globalen sind in der knotenbasierten Darstellungsart die lokalen Kommunikationsbeziehungen (LKB i-j) bestimmten Knoten zugeordnet (siehe Abbildung 6.3). Dadurch wird die Kommunikation zwischen den Knoten über das Kommunikationssystem erkennbar. Der Übersichtlichkeit halber sind der Ablauf der Aktionen auf den Knoten und die Kommunikation zwischen den Knoten für jeden Knoten untereinander über der Zeitachse angeordnet. In dieser Darstellungsart ist zu erkennen, welche Prozessorauslastung auf den einzelnen Knoten geplant ist.

Der Übergang von der globalen Darstellungsart in die knotenbasierte Darstellungsart erfolgt durch die Zuordnung der lokalen Kommunikationsbeziehungen zu bestimmten Knoten. Mehrere Signale eines Knotens, die über das Kommunikationssystem versendet werden, können bestimmten Nachrichtenrahmen sog. *Frames* des Knotens zugeordnet werden.

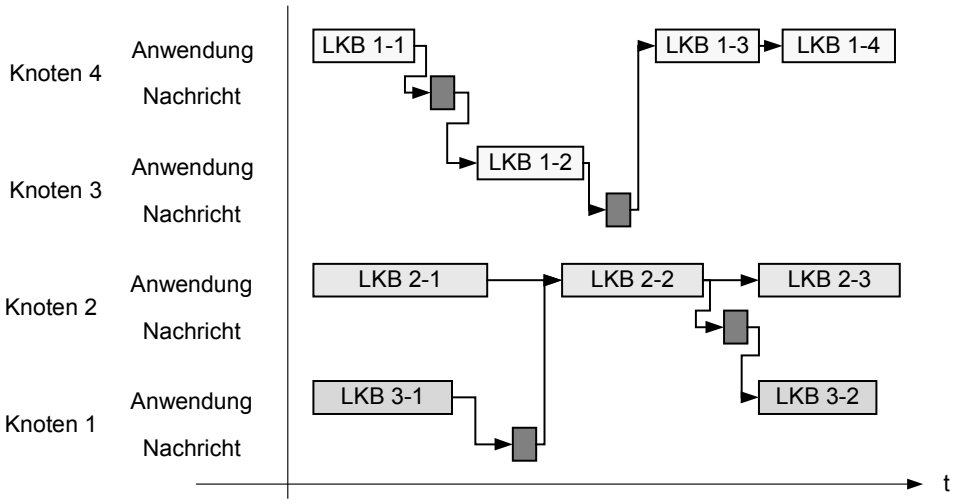


Abbildung 6.3: Knotenbasierte Darstellungsart von Kommunikationsbeziehungen

6.2.2 Nachrichten-Scheduling

In der knotenbasierten Darstellungsart wird das Nachrichten-Scheduling durchgeführt, indem die zeitliche Abfolge der Nachrichtenrahmen festgelegt wird. Das Werkzeug beachtet dabei bestimmte Restriktionen, die sich aus dem TDMA-Verfahren ergeben. Bei dem Kommunikationsprotokoll TTP/C beispielsweise kann jeder Knoten in einem Kommunikationszyklus nur genau einmal senden. Der Vorteil dieser Art des Nachrichten-Schedulings ist, dass im Vergleich zu den bestehenden Werkzeugen eine Beziehung zwischen den zeitlichen Abläufen auf den Knoten und die Nachrichten, die sie senden, visuell dargestellt wird. Die Startzeitpunkte der lokalen Kommunikationsbeziehungen auf den Knoten können entsprechend angepasst und verschoben werden.

6.2.3 Task Scheduling

Das Task Scheduling erfolgt analog zum Nachrichten-Scheduling in der knotenbasierten Darstellungsart durch die Verfeinerung der lokalen Kommunikationsbeziehungen in bestimmte Tasks. Der Startzeitpunkt einer Task wird durch das Positionieren des Balkens bezüglich der Zeitachse festgelegt. Die Länge des Balkens repräsentiert die geplante Deadline der Task.

6.3 Spezifikationssprache für zeitgesteuerte Systeme

Zur Spezifikation eines zeitgesteuerten Systems wird ein definiertes Datenformat benötigt, das zwischen Automobilherstellern und Zulieferern ausgetauscht werden kann. Bisher wird für ereignisgesteuerte Systeme die *OSEK/VDX Implementation Language (OIL)* [OSEK00b] eingesetzt. OIL beschreibt die Implementierungs- und Anwendungsdefinition für jeden Knoten eines ereignisgesteuerten Systems einzeln. Verteilte zeitgesteuerte Systeme erfordern darüber hinaus zusätzliche Definitionen:

- Für die Spezifikation zeitlicher Abläufe von Aktionen und deren Zeitanforderungen ist die Spezifikation von globalen und lokalen Kommunikationsbeziehungen erforderlich.
- Für die Zuordnung der lokalen Kommunikationsbeziehungen zu bestimmten Knoten müssen die Topologie des Kommunikationssystems und die an der Kommunikation beteiligten Knoten definiert werden.
- Die Produkte der Scheduling-Vorgänge, der Nachrichten-Schedule sowie die Task Schedules, müssen ebenfalls spezifiziert werden.

Die ViETTA-Spezifikationssprache für zeitgesteuerte Systeme berücksichtigt diese notwendigen Definitionen. Sie ist als Ergänzung zu OIL entwickelt und analog zu OIL in der Backus-Naur-Form definiert. Die wesentlichen Konstrukte der ViETTA-Spezifikationssprache sind in Abbildung 6.4 zusammengefasst. Eine globale Kommunikationsbeziehung (*Global_Transaction*) beginnt mit dem Schlüsselwort *Global_Transaction.* Anschließend folgen ihr Name und in geschweifte Klammern gesetzt eine Beschreibung ihrer Eigenschaft (*Description*) sowie die Zeiteigenschaften Startzeitpunkt (*StartTime*), Periode (*Period*) und End-to-End Deadline (*Deadline*) sowie die WCET (*WCET*). Anschließend werden Eingangs- und Ausgangssignale angegeben. Durch das Schlüsselwort *Local_Transactions* wird eine Liste der enthaltenen lokalen Kommunikationsbeziehungen (*LT_List*) angegeben. Die Definition von lokalen Kommunikationsbeziehungen und Tasks erfolgt analog. Bei den lokalen Kommunikationsbeziehungen wird eine Liste der enthaltenen Tasks angegeben (*Task_List*). Eine Task wird als atomar angesehen und enthält keine weiteren Elemente.

Die Spezifikation der Topologie des Kommunikationssystems beginnt mit dem Schlüsselwort *Topology*, gefolgt von den Eigenschaften des Kommunikationssystems (*Com_System)* sowie einer Liste der beteiligen Knoten. Die Eigenschaften des Kommunikationssystems werden in *Com_System* durch die Parameter Typ des Kommunikationssystems (z. B. FlexRay, TTP/C), Übertragungsgeschwindigkeit sowie Eigenschaften der Kanäle spezifiziert.

Der Nachrichten-Schedule (*Message_Schedule*) wird angeführt durch das Schlüsselwort *Message Schedule*, gefolgt von einer Liste der gesendeten Nachrichten. Durch das Konstrukt *Message* werden die Eigenschaften einer Nachricht (Länge, Deadline, Periode, sendender Kno-

ten und Nachrichteninhalt) definiert. Zur Vermeidung von redundanter Information werden die Task-Schedules implizit durch die lokalen Kommunikationsbeziehungen und die Eigenschaft *StartTime* der Tasks angegeben.

```
<Global_Transaction> ::=”Global Transaction” <Name> ”{” <Description>
                        <Period> <StartTime> <Deadline> <WCET>
                        <Input_Signal> <Output_Signal> ”Local Transactions”
                        ”{” <LT_List> ”}” ”}” ”;”

<Local_Transaction> ::= ”Local Transaction” <Name> ”{” <Description>
                        <Period> <StartTime> <Deadline> <WCET>
                        <Input_Signal> <Output_Signal>
                        ”Tasks” ”{” <Task_List> ”}” ”}” ”;”

<Task> ::=              ”Task” <Task_Name> ”{” <TaskID> <Description>
                        <StartTime> <Period> <WCET> ”}” ”;”

<Topology> ::=          ”Topology” <Com_System> ”{” <Node_List> ”}” ”;”

<Com_System> ::=        ”Com System” ”{” <Com_Type> <Transmission_Speed>
                        <Channels> ”}” ”;”

<Message_Schedule> ::=  ”Message Schedule” ”{” <Message_List> ”}” ”;”

<Message> ::=           ”Message” <Name> ”{” <Message_ID> <Message_Length>
                        <SendTime> <Deadline> <Period> <Sender_Node>
                        <Message_Content> ”}” ”;”
```

Abbildung 6.4: Auszug aus der Definition der ViETTA-Spezifikationssprache

6.4 Zeitanalyse im Werkzeugkonzept ViETTA

6.4.1 Latenzzeit-Analyse

Für die Parametrisierung der Regler werden die Latenzzeiten der Signale und die Zykluszeit der Regler benötigt. Abweichungen bei diesen Zeiten wirken sich sehr stark auf die Reglergüte aus. Die Latenzzeit-Analyse ist notwendig, da sich die Zeitpunkte der tatsächlich eingeplanten Tasks und Nachrichten von den anfangs spezifizierten relativ losen Zeitanforderungen unterscheiden und im Verlauf der Entwicklung mehrmals ändern können.

Die Latenzzeiten der Signale sind abhängig von den Anwendungszykluszeiten und Kommunikationszykluszeiten. Sie sind im Allg. nicht klar aus der knotenbasierten Darstellungsart ersichtlich. Da die Zykluszeiten der Kommunikationsbeziehungen kleiner sein können als deren Deadlines, kann es zu Überlappungen von verschiedenen Instanzen von Kommunikationsbe-

ziehungen kommen. Für die Analyse wird innerhalb der globalen Kommunikationsbeziehung die lokale Kommunikationsbeziehung durchlaufen und die tatsächlich eingeplante Aktivierungszeit der Tasks analysiert. Im Beispiel in Abbildung 6.5 hängen die Latenzzeiten der Signale von den Aktivierungszeitpunkten der folgenden Tasks auf den verschiedenen Steuergeräten, die die Signale bearbeiten, ab: die Task zur Erfassung des Sensorsignals, die Task zur Ausführung des Reglers und die Task zur Ansteuerung der Aktoren. Die Zykluszeit des Reglers ist die Zeit, mit der er zyklisch ausgeführt wird. Sie entspricht der tatsächlich eingeplanten Zykluszeit der Globalen Kommunikationsbeziehung.

Zum Erreichen einer durchgängigen Werkzeugkette können die ermittelten Latenzzeiten an die Software-Entwicklungswerkzeuge übergeben werden, die diese wiederum als Parameter automatisch einlesen können.

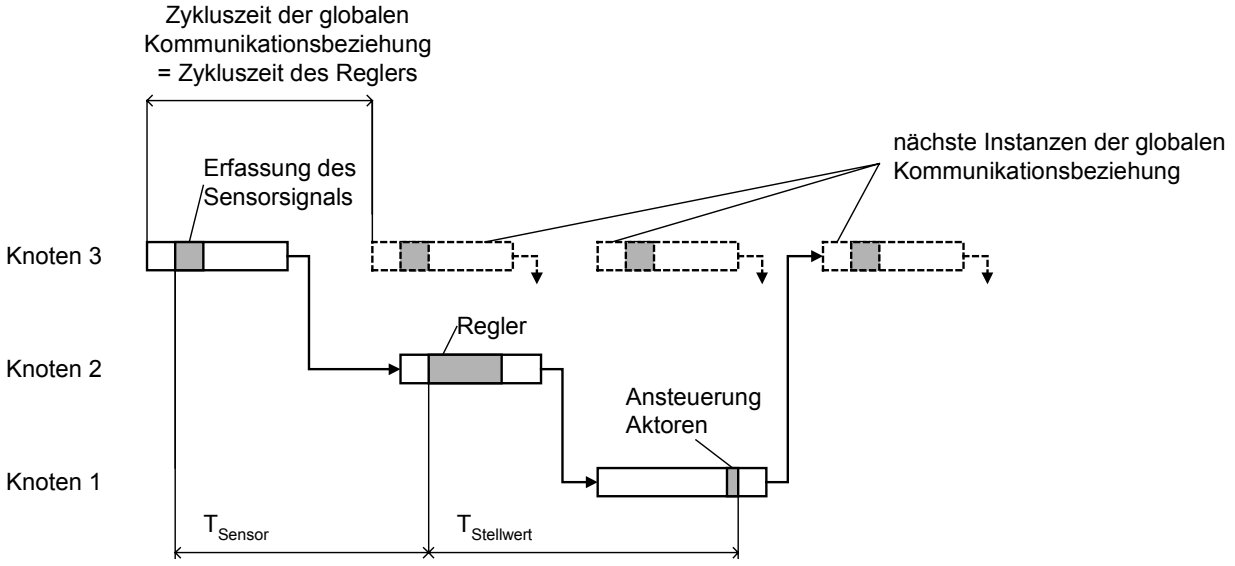


Abbildung 6.5: Latenzzeiten der an einem Regler beteiligten Signale

6.4.2 Schedulability-Analyse

Mit der Schedulability-Analyse wird überprüft, ob alle Tasks eines Knotens alle zeitlichen Anforderungen in jedem Fall einhalten können (siehe Abschnitt 2.1). Die Schedulability-Analyse ist abhängig von der Scheduling-Strategie des Betriebssystems. Beim betrachteten Betriebssystemstandard OSEKtime ist eine Stackbased-Scheduling-Strategie spezifiziert.

Die OSEKtime Spezifikation erlaubt Interrupts, zur Mitteilung von seltenen bzw. einzeln auftretenden Ereignissen, sofern eine minimale Zeitdauer (MINT) garantiert werden kann, in der mit Sicherheit kein weiterer Interrupt auftritt (siehe Abschnitt 3.1.2.2). Bei der Schedulability-Analyse muss deshalb der Nachweis erbracht werden, dass jede Task ihre Deadline bei geplanter Unterbrechung durch andere Tasks und auch beim Auftreten von Interrupts einhält.

Für den Fall, dass mehrere Interrupt-Quellen vorhanden sind, muss die schlechtest mögliche Phasenlage aller Interrupts beachtet werden.

In Abbildung 6.6 ist ein beispielhaftes Worst-Case-Szenario dargestellt. Es gebe eine Task A und eine Task B, die zu den Zeitpunkten t_A und t_B eingeplant sind. Weiterhin seien zwei mögliche Interrupt-Quellen vorhanden. Die Interrupt Service Routinen (ISR) besitzen die $WCET_{I1}$ und $WCET_{I2}$ sowie die $MINT_{I1}$ und $MINT_{I2}$. Der Interrupt 2 habe die höchste Priorität, die Task A die geringste.

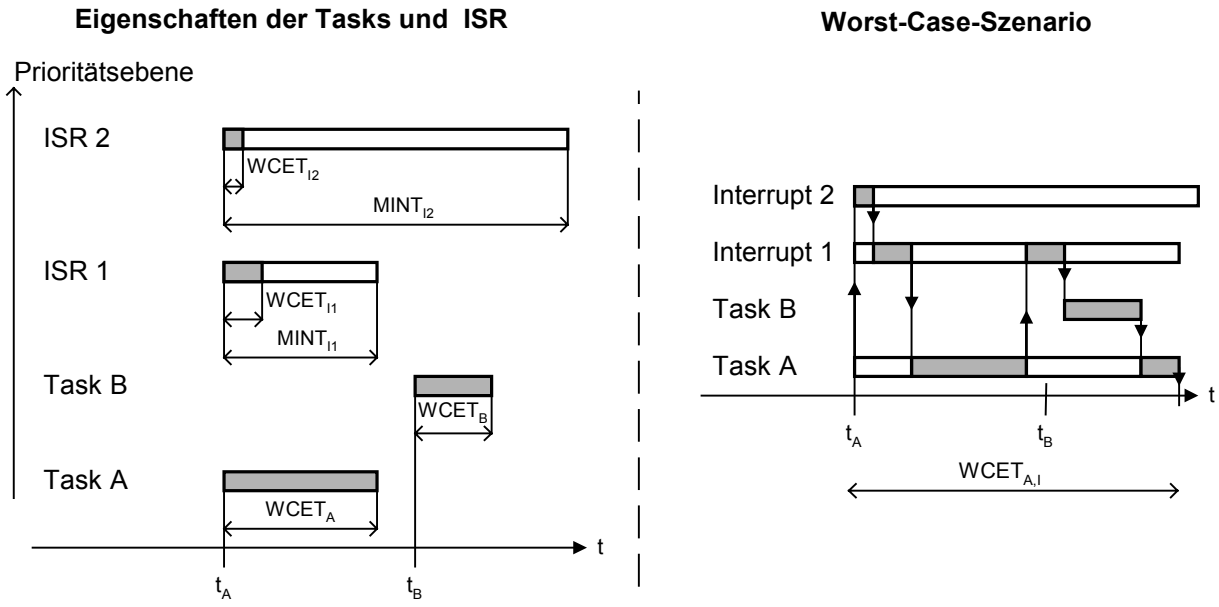


Abbildung 6.6: Bestimmung der maximalen Verzögerung einer Task

Für die Berechnung der maximalen Ausführungszeit der Task A inklusive Unterbrechungen $WCET_{A,I}$ muss der schlechteste Fall angenommen werden. Dies ist der Fall, wenn alle möglichen Interrupts sofort bei der Aktivierung von Task A auftreten und mit ihrer MINT wiederkehren (schlechtest mögliche Phasenlage). In Abbildung 6.6 ist ersichtlich, dass die Task A erheblich durch die Interrupts verzögert werden kann. Für die Berechnung der maximalen Verzögerung der Task B muss wiederum angenommen werden, dass alle Interrupts zum Zeitpunkt der Aktivierung der Task B auftreten. Die Stackbased-Scheduling-Strategie garantiert hierbei, dass der Start einer Task nicht durch die Ausführung einer anderen Task verzögert wird.

6.4.3 WCET-Analyse

Die Schedulability-Analyse basiert auf den bestimmten Ausführungszeiten der WCET-Analyse. Das Werkzeugkonzept ViETTA unterstützt die modellbasierte WCET-Analyse, indem es eine Schnittstelle für verschiedene Software-Entwicklungswerkzeuge bietet. Die Anbindung der WCET-Analyse an konkrete Software-Entwicklungswerkzeuge im Werkzeugkonzept ViETTA wird dazu im nächsten Kapitel gesondert betrachtet.

Das in diesem Kapitel vorgestellte Werkzeugkonzept ViETTA unterstützt die Entwicklung zeitgesteuerter Systeme über den gesamten Entwicklungszeitraum hinweg. Der freie Datenaustausch wird durch eine standardisierte Datenbasis gewährleistet. Die Planung zeitlicher Abläufe im zeitgesteuerten System wird durch die grafische Spezifikation von Kommunikationsbeziehungen vorgenommen. Die Entwicklung von regelungstechnischen Anwendungen wird durch die Latenzzeit-Analyse von Signalen unterstützt. Darüber hinaus ermöglicht das Werkzeugkonzept ViETTA den Nachweis der geplanten Zeitanforderungen durch die WCET- und die Schedulability-Analyse. Auf die modellbasierte WCET-Analyse wird im nächsten Kapitel ausführlich eingegangen, indem das Konzept aus Kapitel 4 auf konkrete Software-Entwicklungswerkzeuge übertragen wird.

7 Anbindung der WCET-Analyse an Software-Entwicklungswerkzeuge

In diesem Kapitel wird die Anbindung der WCET-Analyse an Software-Entwicklungswerkzeuge im Werkzeugkonzept ViETTA unter Anwendung des Konzepts der modellbasierten WCET-Analyse aus Kapitel 4 vorgenommen. Es wird aufgezeigt, auf welche Weise die während der Entwicklung zeitgesteuerter Systeme mit Software-Entwicklungswerkzeugen erstellten Modelle automatisch einer WCET-Analyse unterzogen werden können.

Zunächst wird die konkrete Umsetzung des Konzepts auf das komponentenbasierte Software-Entwicklungswerkzeug ViPER/ESTEREL, das am IAS entwickelt wurde, vorgestellt. Durch die Anwendung des Konzepts an dem kommerziellen Werkzeug MATLAB wird gezeigt, dass das Konzept nicht nur für akademische Werkzeuge, sondern auch erfolgreich bei kommerziellen Werkzeugen eingesetzt werden kann. MATLAB wurde aus einer Vielzahl kommerziell verfügbarer Werkzeuge ausgewählt, da es in der Automobilindustrie für die Entwicklung von Fahrwerksfunktionen favorisiert und voraussichtlich zukünftig auch für By-Wire-Systeme eingesetzt wird.

7.1 Übertragung des Konzepts auf ViPER/ESTEREL

7.1.1 Eigenschaften der Werkzeugumgebung ViPER/ESTEREL

Die Werkzeugumgebung ViPER (Visual Programming Environment for Embedded Real-Time Systems) wurde am IAS für die komponentenbasierte Entwicklung zeitgesteuerter Anwendungen mit synchronen Softwarekomponenten konzipiert [GuRi99]. Die synchronen Softwarekomponenten [Gunz99] verwenden die synchrone Programmiersprache ESTEREL [BeGo92]. ESTEREL basiert auf präzise definierter mathematischer Semantik. Durch die synchronen Softwarekomponenten wird das Zusammenbauen deterministischer Anwendungssoftware ermöglicht. In Verbindung mit einem zeitgesteuerten Kommunikationssystem lassen sich verteilte deterministische Systeme realisieren [Gunz99]. Synchrone Softwarekomponenten sind somit prädestiniert für die Entwicklung von verteilten Anwendungen für By-Wire-Systeme.

Synchrone Programmiersprachen beruhen auf einer Entwurfsannahme, der sog. *Synchronitätshypothese*. Sie nimmt an, dass Softwaremodule parallel und synchron zu einem gemeinsamen Takt unendlich schnell ausgeführt werden. Auf diese Weise kann der Entwurf von korrekten Systemen vereinfacht werden, da Indeterminismen, verursacht durch asynchrone Vorgänge, nicht auftreten können. Der ESTEREL-Compiler generiert aus den parallelen Modulen unter

Beachtung der Kommunikation zwischen den Modulen ein sequenzielles ausführbares Programm.

Die Synchronitätshypothese hat nur Gültigkeit, wenn das ausführbare Programm in jedem Falle beendet ist, bevor es erneut durch den eintreffenden Takt gestartet wird. Für den Einsatz von synchronen Programmen ist somit die WCET-Analyse der generierten Programme Grundvoraussetzung [Ring00b].

Modellierung

In ViPER/ESTEREL wird die Anwendungssoftware auf der Modellebene aus vorgefertigten synchronen Softwarekomponenten grafisch erstellt, die in einer Komponentenbibliothek zur Verfügung gestellt werden (siehe Abbildung 7.1). Die Softwarekomponenten werden ausgewählt, parametrisiert und im Modellierungseditor über Signale miteinander verbunden. Das so erstellte Modell wird in einer Projektdatenbank gespeichert. Die Implementierung der Softwarekomponenten ist für den Anwendungsentwickler nicht sichtbar.

Eine synchrone Softwarekomponente besteht aus einem reaktiven Teil und einem Datenverarbeitungsteil (DV-Teil). Der reaktive Teil legt das Verhalten der Softwarekomponente im Inneren fest. Der DV-Teil besteht aus Datentyp-Definitionen und einer Menge von datenverarbeitenden Funktionen und Prozeduren. Diese werden vom reaktiven Teil aufgerufen. Für die Spezifikation des reaktiven Teils wird die Programmiersprache ESTEREL benutzt, während für den DV-Teil der Softwarekomponenten die Programmiersprache C verwendet wird.

Codegenerierung

Der Generierungsprozess besteht aus mehreren Schritten (siehe Abbildung 7.1). Zunächst wird aus der Projektdatenbank ESTEREL-Code erzeugt, der die reaktiven Teile der Komponenten verbindet. Die Softwarekomponenten stehen in der Komponentenbibliothek bereit. Alle ESTEREL-Programme werden vom ESTEREL-Compiler in einen flachen Mealy-Automat[13] übersetzt. Der Zustandsautomat kann auf zwei grundlegend verschiedene Arten dargestellt werden [Berr98], aus denen anschließend C-Code generiert wird. Bei der *Object-Code*-Darstellung (*OC*) werden alle Zustände und deren mögliche Übergänge explizit gespeichert. Sehr kleine Ausführungszeiten sind die Folge. Der Nachteil dieser Darstellung ist die Codegröße. Sie ist nur für sehr kleine Programme geeignet. Große Anwendungen führen meistens zu einer Explosion der Codegröße [Berr98]. Die *Sorted-Circuit-Code*-Darstellung (*SSC*) beschreibt den Zustandsautomat implizit durch boole'sche Gleichungen. Diese Darstellungsart hat einen geringen Speicherplatzbedarf, führt jedoch zu längeren Ausführungszeiten, da die Gleichungen zur Laufzeit gelöst werden müssen. Ein an die Zielhardware angepasster C-Compiler erzeugt

[13] Beim Mealy-Automat werden die Ausgangsgrößen durch Ausführen von Aktionen beim Zustandsübergang ausgegeben, während sie beim Moore-Automat in einem Zustand ausgegeben werden (siehe Abschnitt 4.1.2).

abschließend aus dem Zustandsautomaten und den Datenverarbeitungsteilen ein ausführbares Programm.

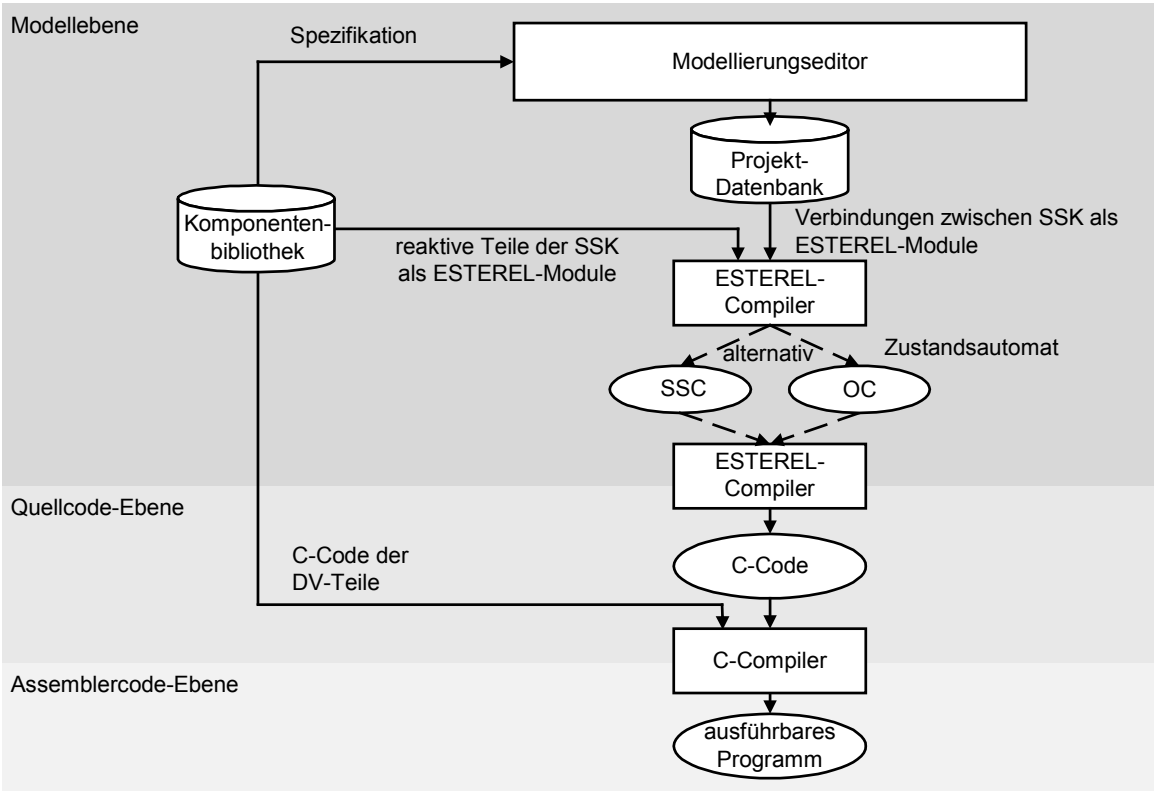


Abbildung 7.1: Codegenerierungsprozess bei ViPER/ESTEREL

Bewertung der Eigenschaften der Werkzeugumgebung

Die für die vorliegende Arbeit relevanten Eigenschaften der Werkzeugumgebung ViPER sind in Tabelle 7.1 anhand der Kriterien aus Abschnitt 4.8 zusammengestellt. Das Modell ist in einer

Tabelle 7.1: Eigenschaften der Werkzeugumgebung ViPER/ESTEREL

Kriterien	**ViPER/ESTEREL**
Kriterium 1: Ist eine offen gelegte Schnittstelle zum Modell vorhanden und enthält das Modell verwertbare semantische Informationen?	ja, offene Datenbank ja, offen gelegter Zustandsautomat in den OC- und SSC-Darstellungen mit mathematisch definierter Semantik
Kriterium 2: Ist ein offen gelegter Codegenerator vorhanden?	ja, ViPER-Modelle in ESTEREL-Programme nein, ESTEREL-Compiler nicht offen gelegt
Kriterium 3: Sind offen gelegte Bibliotheksfunktionen der Modellelemente vorhanden?	ja, als veränderbarer C-Code
Kriterium 4: Ist eine offen gelegte Schnittstelle zur Oberfläche vorhanden?	ja

Mircosoft Access Datenbank gespeichert und somit offen gelegt. Die Codegenerierungsvorschriften, die aus den in der Datenbank gespeicherten Informationen ESTEREL-Programme erzeugen, sind bekannt. Die OC- und SSC-Darstellungen sind als lesbare ASCII-Dateien gespeichert. Deren Semantik ist mathematisch präzise definiert. Nicht offen gelegt sind die Codegenerierungsvorschriften des ESTEREL-Compilers. Eine offen gelegte Schnittstelle zur Oberfläche ermöglicht die Darstellung der WCET-Analyse-Ergebnisse auf der Modellebene.

7.1.2 Betrachtung des generierten Codes

7.1.2.1 DV-Teile

Bei ViPER/ESTEREL liegen die DV-Teile der synchronen Softwarekomponenten als Quellcode in der Programmiersprache C vor. Sie wurden manuell erstellt bzw. können vom Anwendungsentwickler erstellt werden. Für die WCET-Analyse der DV-Teile können die in Abschnitt 3.2 vorgestellten Verfahren angewandt werden. Die Implementierungen der automatisch generierten Zustandsautomaten bedürfen jedoch näherer Betrachtung. Nachfolgend werden die Eigenschaften des generierten Codes analysiert. Sie sind in Tabelle 7.2 anhand der Kriterien aus Abschnitt 4.8 dargestellt.

7.1.2.2 Explizite Object-Code-Darstellung

Bei der C-Code-Implementierung der expliziten Object-Code-Darstellung ist der Zustandsautomat in unveränderlichen Datenstrukturen gespeichert, die von einer Schleife gelesen werden. Abhängig vom Eingangsvektor der anliegenden Signale der Umgebung, dem aktuellen Zustand und dem Inhalt der Datenstruktur werden die DV-Teile ausgeführt. Die Kontrollstruktur des Codes besteht im Wesentlichen aus einer einfachen Schleife, die abhängig von den Inhalten der Datenstruktur iteriert und über Zeiger dynamisch Funktionen aufruft. Der eigentliche Kontrollfluss ist in der Datenstruktur gespeichert. Die im Abschnitt *Stand der Technik* vorgestellten WCET-Analyse-Ansätze können nicht angewandt werden, da sie die Kontrollstruktur in Form von Kontrollstruktur-Anweisungen einer Programmiersprache erwarten und darüber hinaus keine dynamischen Funktionsaufrufe zulassen.

7.1.2.3 Implizite Sorted-Circuit-Code-Darstellung

Die C-Code-Implementierung der Sorted-Circuit-Code-Darstellung besteht hauptsächlich aus einer sequenziellen Folge von boole'schen Gleichungssystemen und zugeordneten bedingten Funktionsaufrufen (DV-Teil der Softwarekomponenten). Der Eingangsvektor und der Vektor des aktuellen Zustands sind die Eingangsgrößen der boole'schen Gleichungssysteme, die den Zustandsübergang berechnen. Ein Funktionsaufruf wird ausgeführt, wenn das Ergebnis des zugehörigen Gleichungssystems *wahr* ist. Bei der WCET-Analyse des Codes treten zwei Schwierigkeiten auf: einerseits ist der Code stark verzweigt, anderseits sind die Pfadinformationen schwer auf Codeebene (Quellcode oder Assemblercode) zu analysieren.

Tabelle 7.2: Eigenschaften des generierten Codes von ViPER/ESTEREL

	ViPER/ESTEREL	
Kriterien	**Explizite Object-Code-Darstellung**	**Implizite Sorted-Circuit-Code-Darstellung**
Kriterium 5: Sind die möglichen Pfade des generierten Codes statisch analysierbar?	nein nicht mit herkömmlichen Analysetechniken	ja, Compiler erzeugt jedoch stark verzweigte Codestruktur
Kriterium 6: Sind Änderungen am generierten Code notwendig?	ja, dazu müsste jedoch die gesamte Codestruktur verändert werden, die dann der ursprünglichen Intention widerspricht, deshalb ist eine spezialisierte Analyse sinnvoller	ja, Reduktion der Anzahl von möglichen Ausführungspfaden
Kriterium 7: Sind Pfadinformationen notwendig, um die Analyse zu ermöglichen?	ja	nein, da keine Schleifen oder Rekursionen auftreten
Kriterium 8: Führen zusätzliche Pfadinformationen zu einer Verbesserung der oberen Schranke der WCET?	ja	ja, stets gemeinsam ausgeführte DV-Teile

Stark verzweigter Code

Bei der Übersetzung der boole'schen Gleichungen durch den kommerziellen Tasking C-Compiler [Taks98] in Assemblercode erzeugt die nicht abschaltbare *Logical Expression Optimization* eine sehr stark verzweigte Codestruktur. Der erzeugte Code ist ungeeignet für die WCET-Analyse aus folgenden Gründen:

- Die Anzahl möglicher Pfade auf Assemblercode-Ebene wird sehr stark erhöht. Diesen möglichen Pfaden steht keine Korrespondenz auf höheren Ebenen (Quellcode oder Zustandsautomat) gegenüber. Die Pfadanalyse der WCET-Analyse wird dadurch erheblich erschwert.

- Die Codestruktur führt zu kürzeren Best-Case-Ausführungszeiten, verschlechtert jedoch die WCET, da im Worst-Case sehr viele Sprunginstruktionen notwendig sind, die insbesondere bei komplexen Hardwarearchitekturen erheblich größere Ausführungszeiten benötigen.

- Die Grundblöcke des erzeugten Codes bestehen aus nur wenigen Instruktionen (zwei bis drei). Da bei der Grundblock-Analyse jeweils an den Blockgrenzen pessimistische Annahmen getroffen werden, wirkt sich diese Codestruktur sehr negativ auf die Qualität der ermittelten WCET aus.

Fehlende Pfadinformation

Bei der C-Code Implementierung der Sorted-Circuit-Code-Darstellung sind die strukturell möglichen Pfade offensichtlich erkennbar, da der Code keine Schleifen enthält. Die prinzipielle Co-

destruktur, die die boole'schen Gleichungen und DV-Teile als Black Box betrachtet, ist in Abbildung 7.2 dargestellt. Da es bei den Tests keine alternativen Aktionen zu den Aktionen i, j, k (DV-Teile) gibt, ist der strukturell längste Pfad einfach zu bestimmen. Er ist der Pfad, der alle Aktionen durchläuft (siehe Abbildung 7.2 Pfad A). Dieser Pfad ist im Allg. nicht ausführbar, denn dies würde bedeuten, dass es einen Zustandsübergang im Zustandsautomaten gäbe, bei dem alle vorhandenen Aktionen ausgeführt werden. Die WCET dieses Pfads ist im Allg. sehr weit von der des ausführbaren Pfades entfernt und somit sehr pessimistisch.

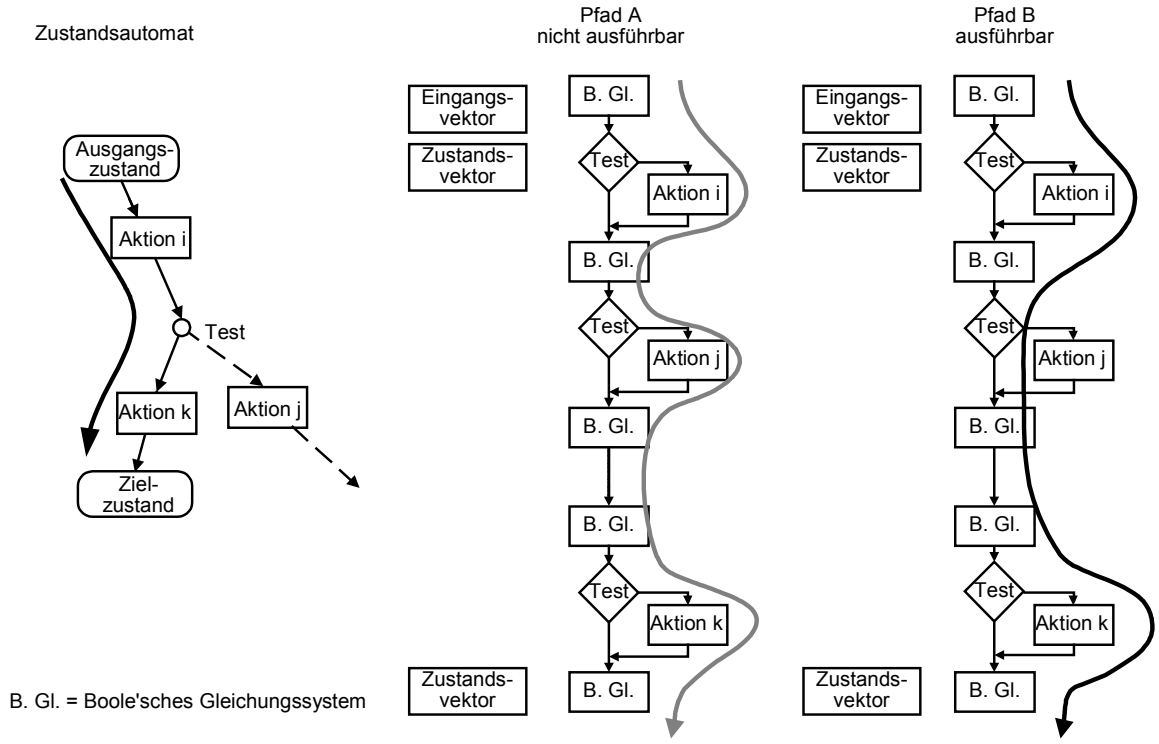


Abbildung 7.2: Pfade in der C-Code-Implementierung der Sorted-Circuit-Code-Darstellung

Es sind nur die Pfade ausführbar, für die ein zugehöriger Zustandsübergang im Zustandsautomaten existiert. In dem Beispiel in Abbildung 7.2 ist der Pfad A mit den Aktionen (i, j, k) nicht ausführbar, da es im Zustandsautomaten keinen Zustandsübergang gibt, der die Aktionen i, j und k enthält. Der Pfad B, der die Aktionen i und k durchläuft, ist jedoch ausführbar.

Für die Berechnung der ausführbaren Pfade auf Basis der Sorted-Circuit-Code-Darstellung müssten die boole'schen Gleichungssysteme für alle möglichen Eingangsvektoren und Zustandsvektoren gelöst werden, sowohl auf Quellcode- als auch auf Assemblercode-Ebene. Die im Zustandsautomaten explizit vorhandenen Informationen über die möglichen Zustandsübergange müssten aus der impliziten Sorted-Circuit-Code-Darstellung wiedergewonnen werden. Die Idee liegt somit auf der Hand, diese Pfadinformationen direkt von der Modellebene aus der expliziten Zustandsautomaten-Darstellung zu gewinnen.

7.1.3 WCET-Analyse-Werkzeugkonzept

7.1.3.1 Übertragung des Konzepts

Die Betrachtung der generierten Codestruktur hat gezeigt, dass die WCET-Analyse mit bekannten Ansätzen nur eingeschränkt möglich bzw. die Qualität der gelieferten Ergebnisse unzureichend ist. Weiterhin wurde festgestellt, dass durch die Nutzung der zusätzlichen Informationen, die auf der Modellebene schon explizit bekannt sind, die Qualität der WCET erheblich verbessert werden kann. Bei der Betrachtung der ViPER/ESTEREL Werkzeugumgebung wird Folgendes deutlich:

- Erst der ESTEREL-Compiler legt die Codestruktur fest. Die Art und Struktur des ViPER-Modells hat keinen bedeutenden Einfluss und bietet keine Pfadinformationen.
- Der ESTEREL-Compiler ist nicht offen gelegt und kann somit nicht verändert werden, um zusätzliche Pfadinformationen zu erzeugen.
- ESTEREL bietet mit den Sorted-Circuit-Code- und den Object-Code-Darstellungen ein zugängliches Modell des Zustandsautomaten mit präzise definierter mathematischer Semantik.
- Nur die implizite Sorted-Circuit-Code-Darstellung ist aufgrund der Speicherplatzeffizienz einsetzbar.
- Bei dem verwendeten C-Compiler kann die *Logical Expression Optimization* nicht verändert werden.

Das in Abbildung 7.3 dargestellte Werkzeugkonzept für die Anbindung der WCET-Analyse an ViPER/ESTEREL resultiert aus diesen Überlegungen. Ein neu geschaffener *SSC2Ass*-Codegenerator erzeugt aus der Sorted-Circuit-Code-Darstellung einfach zu analysierenden Code, indem die sich negativ auswirkenden Code-Optimierungen umgangen werden. Der neu entwickelte *SSC2Ass*-Codegenerator erzeugt direkt aus dem Zustandsautomaten in der Sorted-Circuit-Code-Darstellung linearen Assemblercode und ersetzt somit den herkömmlichen ESTEREL-Compiler und den C-Compiler (gestrichelt gezeichnet).

Die WCET-Analyse bestimmt aus dem generierten Assemblercode zusammen mit den WCET der DV-Teile die WCET des gesamten synchronen Programms. Die WCET der DV-Teile stammen aus der Komponentenbibliothek, in der auch die Spezifikation der synchronen Softwarekomponenten, der reaktive Anteil und die C-Implementierung gespeichert sind. In ihr werden die WCET der DV-Teile abgelegt, sobald sie für eine Plattform bestimmt wurden. Mit Hilfe eines weiteren Teilwerkzeuges, dem *OC2Path*-Generator, werden die notwendigen Pfadinformationen aus der expliziten Darstellung des Zustandsautomaten erzeugt und über eine gesonderte Schnittstelle der WCET-Analyse übergeben. Ein weiteres Teilwerkzeug *WCET2Model* übernimmt die Übergabe der WCET auf die Modellebene.

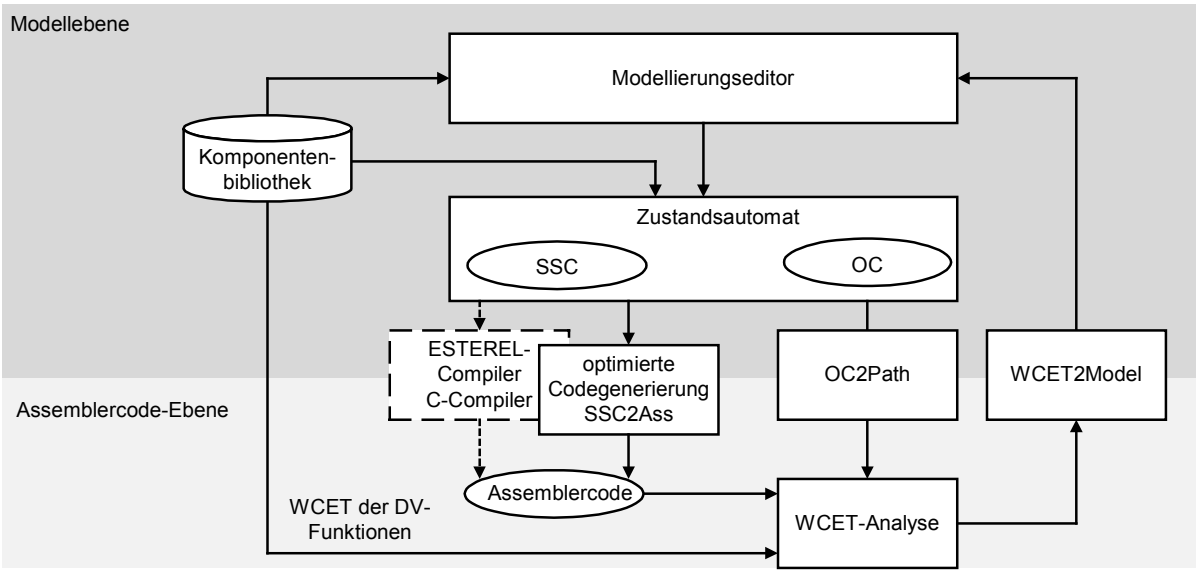


Abbildung 7.3: WCET-Analyse-Werkzeugkonzept für ViPER/ESTEREL

7.1.3.2 Generierung von geeigneten Codestrukturen

Bei der Generierung von linearen Codestrukturen werden die boole'schen Gleichungen durch den *SSC2Ass*-Codegenerator als boole'sche Instruktionen, die sequenziell ausgeführt werden, implementiert. Dadurch lässt sich ein boole'sches Gleichungssystem als ein Grundblock darstellen, der stets vollständig ausgeführt wird. Das vereinfacht die WCET-Analyse (Pfadanalyse und Pipeline-Simulation) erheblich, denn der Worst-Case-Pfad ist schon im Voraus für das Codestück bekannt.

Der kommerzielle Compiler hingegen verwendet Lade-, Vergleichs- und bedingte Sprunginstruktionen um die boole'schen Gleichungen zu implementieren. In Tabelle 7.3 ist eine boole'sche Gleichung in der Sorted-Circuit-Code-Darstellung dem stark verzweigten Assemblercode des kommerziellen Compilers und dem linearen Assemblercode des *SSC2Ass*-Codegenerators gegenübergestellt. In der linken Spalte ist ein Eintrag der Gleichungstabelle der SSC Darstellung dargestellt: In *wire 0* wird das Ergebnis der *oder*-Verknüpfung der Register *R1* und *R2* gespeichert.

Steht bei dem verzweigten Code das Endergebnis einer Gleichung bereits sicher fest, muss der Rest der Gleichung nicht weiter berechnet werden. Ist beispielsweise das Register R1 = wahr, dann ist das Ergebnis sicher wahr, ungeachtet des Werts im Register R2. Der Wert in Register R2 muss nicht berücksichtigt werden (entsprechend den fett markierten Assembler-Instruktionen in Tabelle 7.3 mittlere Spalte). Diese Implementierung wirkt sich insbesondere bei großen Gleichungen aus, bei denen die Wahrscheinlichkeit, dass das Ergebnis schon frühzeitig feststeht, sehr groß ist. Für die WCET-Analyse sind diese Fälle jedoch nicht relevant. Im Worst-Case

führt die Implementierung nur zu einer Verschlechterung der Ausführungszeit.[14] So ist in dem Beispiel in Tabelle 7.3 die WCET des verzweigten Assemblercodes beim Siemens 80C167 Mikrocontroller 2,6µs, während der lineare Code nur 1,6µs benötigt.

Tabelle 7.3: Gegenüberstellung von verzweigtem und linearem Assemblercode einer boole'schen Gleichung

boole'sche Gleichung in der Sorted-Circuit-Code-Darstellung	**verzweigter Assemblercode**	**linearer Assemblercode**

`wire 0: (R1 or R2)`	**`MOVB RL1,(R1)`**
	`CMPB RL1,#00h`
	`JMPR cc_NZ,_7`
	`JMPR cc_UC,_10`
	`_10: MOVB RL1,(R2)`
	`CMPB RL1,#00h`
	`JMPR cc_NZ,_7`
	`JMPR cc_UC,_8`
	`_7: MOVB RL1,#01h`
	`JMPR cc_UC,_9`
	`_8: MOVB RL1,#00h`
	`_9: MOVB _4,RL1`
		`MOVB RL1,(R1)`
		`ORB RL1,(R2)`
		`MOVB _4,RL1`

7.1.3.3 Generierung von Pfadinformationen

In der Zustandstabelle der Object-Code-Darstellung sind für alle Zustände alle möglichen Zustandsübergänge aufgeführt. Für jeden Zustandsübergang sind sowohl die Tests als auch die Aktionen, die abhängig von diesen Tests ausgeführt werden, und zusätzlich der zu erreichende Zielzustand explizit aufgelistet. Der OC2Path-Generator extrahiert aus der Zustandstabelle für jeden Zustandsübergang die Abfolge der ausgeführten Aktionen:

```
...
Zustand_x: Aktion_i Aktion_j Zustand_y
Zustand_x: Aktion_i Aktion_k Zustand_z
...
```

Jeweils eine Abfolge der ausgeführten Aktionen charakterisiert einen Zustandsübergang. Die Menge aller Abfolgen der ausgeführten Aktionen charakterisiert alle möglichen Zustandsübergänge. Diese Informationen werden vom OC2Path-Generator der WCET-Analyse über eine gesonderte Schnittstelle übergeben.

[14] Außerdem wird erheblich mehr Programm-Speicherplatz benötigt.

7.1.3.4 WCET-Analyse

Die Grundidee zur Pfadanalyse von ESTEREL-Programmen ist die folgende: Bei der WCET-Analyse von ESTEREL-Programmen soll die explizite Information über mögliche Zustandsübergänge aus dem Modell (der OC-Darstellung) verwendet werden, um auf Assemblercode-Ebene im Kontrollflussgraf die ausführbaren Pfade zu ermitteln. Diese Vorgehensweise ist in Abbildung 7.4 dargestellt. Die vom OC2Path-Generator extrahierten möglichen Zustandsübergänge werden an den Kontrollflussgraf übergeben. Die Zuordnung der high-level Informationen auf Modellebene zu den low-level Informationen auf der Kontrollflussgraf-Ebene erfolgt über die Aktionen, die beim Zustandsübergang ausgeführt werden. Sie können im Assemblercode als Funktionsaufrufe wiedergefunden werden. Aus der Vielzahl strukturell möglicher Pfade sind nur jene ausführbar, die exakt die Aktionen der Zustandsübergänge beinhalten. Alle anderen Kombinationen von DV-Teilen sind nicht ausführbar. Als Ergebnis liegen die ausführbaren Pfade fest.

Die Ausführungszeit der ausführbaren Pfade ist die Summe der WCET aller boole'schen Gleichungssysteme und die WCET der ausgeführten Aktionen (DV-Teile). Aus der Menge der ausführbaren Pfade kann anschließend der Pfad mit der längsten Ausführungszeit gesucht werden.

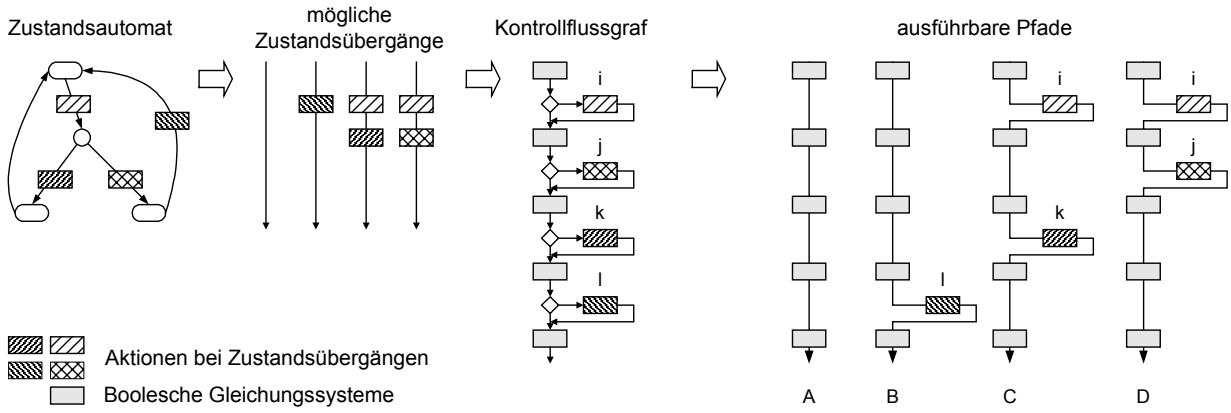


Abbildung 7.4: Grundidee zur Pfadanalyse von ESTEREL-Programmen

Das Berechnungsverfahren entspricht der Suche im Kontrollflussgraf. Die Implicit Path Enumeration Methode ist nicht einsetzbar, da keine Abhängigkeiten, die stets Gültigkeit haben, zwischen Zweigen (Aktionen) im Kontrollflussgraf formuliert werden können.[15] Abhängigkeiten zwischen Zweigen gelten nur für bestimmte Pfade. In Abbildung 7.4 könnte der Pfad C zwar folgendermaßen beschrieben werden: „Aktion i wird stets mit Aktion k und keiner weiteren Aktion ausgeführt". Dies widerspricht jedoch dem Pfad D, da dort nur die Aktion i und Aktion j

[15] Aktionen, die stets zusammen ausgeführt werden, können gleich zusammen *en bloc* nach einem booleschen Gleichungssystem ausgeführt werden.

ausgeführt werden. Die impliziten Beschreibungen aller ausführbaren Pfade widersprechen sich und können deshalb nicht als ein lineares Optimierungsproblem dargestellt werden. Möglich wäre jedoch, die statische Kontrollflussstruktur implizit zu beschreiben und jeweils nur einen ausführbaren Pfad durch Abhängigkeiten zwischen Zweigen zu beschreiben. Das lineare Optimierungsproblem müsste dann für alle ausführbaren Pfade maximiert werden.

7.1.3.5 Zuordnung der WCET-Analyse-Ergebnisse zu Modellelementen

Bei einer deterministischen Hardwarearchitektur ohne Caches ist es möglich, die WCET der DV-Teile einmal zu bestimmen und in der Komponentenbibliothek abzulegen. Somit kann bei der Auswahl einer Komponente bereits die WCET eingesehen und eine Auswahl auf Grundlage der WCET getroffen werden. In Abbildung 7.5 ist die entsprechende Tabelle der Komponentenbibliothek dargestellt. In der ersten Spalte *Component* sind die einzelnen Komponenten verzeichnet, in der zweiten die zugehörigen DV-Teile (*Function-Name*) der Komponenten. In der Spalte *WCET* sind die WCET der DV-Teile für entsprechende Hardware und Speicher zu finden.

Bei einer komplexen Hardwarearchitektur mit Caches hängt die Ausführungszeit der DV-Teile sehr stark von dem umgebenen Code ab. In diesem Fall ist eine Analyse des gesamten Code notwendig. Die Einträge in der Tabelle können jedoch erste Anhaltspunkte für die WCET liefern.

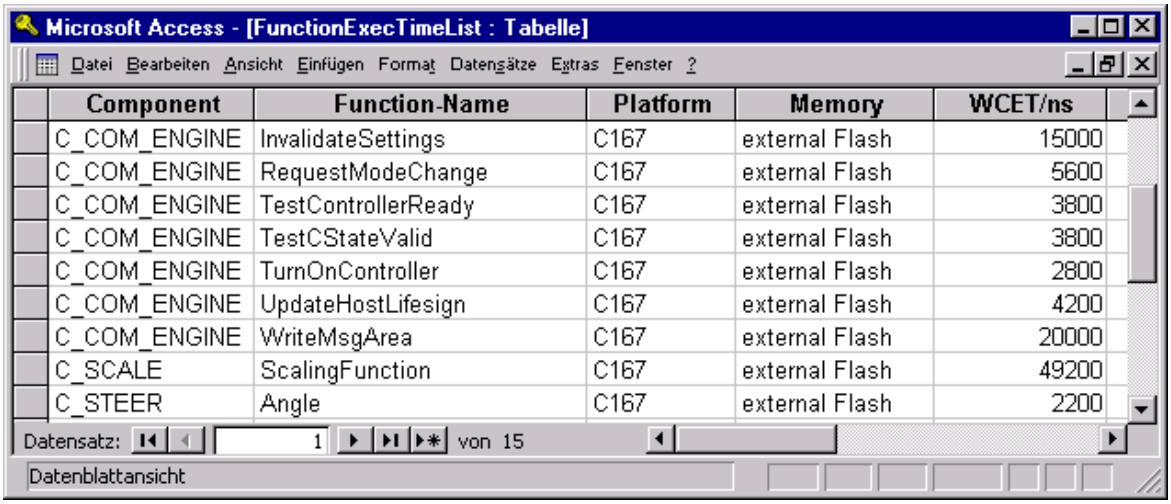


Abbildung 7.5: Auflistung der WCET der DV-Teile der synchronen Softwarekomponenten in der Komponentenbibliothek

Der Beitrag der reaktiven Teile ist jedoch nicht im Voraus bekannt. Aus allen parallelen ESTEREL-Modulen wird ein geschlossener Automat generiert. Der Beitrag einer einzelnen Komponente an der Gesamt-WCET hängt von den übrigen Komponenten im Modell und wie sie miteinander verbunden sind ab. Die WCET-Analyse muss deshalb nach der Generierung erfolgen. Die Beiträge der einzelnen synchronen Softwarekomponenten sind nach der Generierung nicht mehr identifizierbar und müssen deshalb als eine Einheit dargestellt werden. Nach der

WCET-Analyse kann der Anwender folgende Informationen einsehen:

- die WCET des gesamten synchronen Programms
- die Ausführungszeit für bestimmte Zustandsübergänge und die dabei ausgeführten DV-Teile
- die Ausführungszeit des reaktiven Kerns, im Vergleich zu den DV-Teilen

7.2 Übertragung des Konzepts auf MATLAB/Simulink/ Stateflow[16]

7.2.1 Eigenschaften der Werkzeugumgebung MATLAB/Simulink/ Stateflow

Die Werkzeugumgebung MATLAB (MATrix LABoratory) [Math] ist ein umfangreiches Software-Entwicklungswerkzeug für die Entwicklung von dynamischen und ereignisdiskreten Systemen. MATLAB ermöglicht den grafischen Entwurf von Software auf abstrakter Modellierungsebene, Simulation und Codegenerierung und bietet eine Familie von anwendungsspezifischen Lösungen, sog. *Toolboxes*. Die Werkzeugumgebung MATLAB besteht aus den Werkzeugen *Simulink* [Math98b], *Stateflow* [Math98d] und *Real-Time Workshop* [Math98e].

Modellierung
Simulink ist ein Werkzeug zur Modellierung, Simulation und Analyse dynamischer Systeme. Die Modelle werden grafisch als Blockdiagramme (siehe Abschnitt 4.1.2) erstellt. Die Modellierung erfolgt durch die Auswahl von Modellelementen, sog. Simulink-Blöcken, aus der Toolbox, das Parametrisieren und Verbinden dieser Blöcke mit anderen Blöcken. In der Simulink-Toolbox sind zahlreiche regelungstechnische Blöcke enthalten, die den grafischen Entwurf regelungstechnischer Anwendungen sehr gut unterstützen.

Das Werkzeug Stateflow dient zur Modellierung und Simulation von komplexen ereignisdiskreten Systemen, basierend auf hierarchischen und parallelen endlichen Zustandsautomaten (siehe Abschnitt 4.1.2). Die Zustandsautomaten werden als Block in ein Simulink-Modell integriert und können über Signale mit dem Simulink-Modell kommunizieren.

Codegenerierung
Der Real-Time Workshop (RTW) erzeugt automatisch aus den Simulink- und Stateflow-Modellen einen auf eine bestimmte Zielhardware speziell angepassten Zielcode. Der Codegenerierungsprozess wird von dem *Target Language Compiler (TLC)* koordiniert (siehe Abbildung 7.6). Die Codegenerierungsvorschriften sind in den sog. TLC-Dateien festgelegt. Bei Simulink

[16] Die nachfolgenden Ausführungen beziehen sich auf die MATLAB Version 5.3 (R11).

sind die TLC-Dateien fest, während bei Stateflow der Codegenerator *sfc.m* bei jeder Codegenerierung zunächst die TLC-Dateien erzeugt.

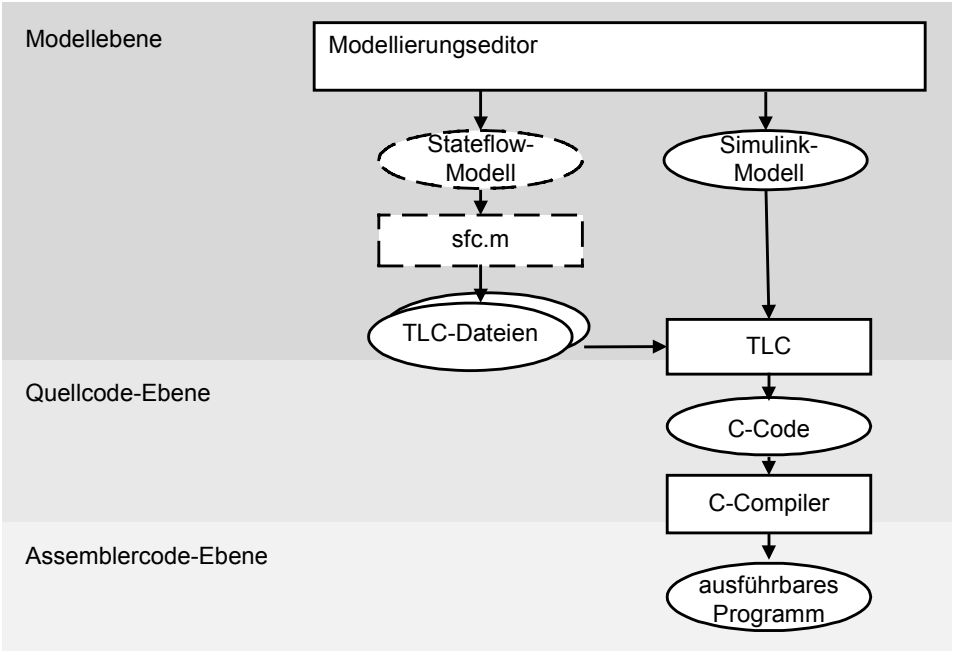


Abbildung 7.6: Codegenerierungsprozess bei MATLAB/Simulink/Stateflow

Bewertung der Eigenschaften der Werkzeugumgebung

Die für die vorliegende Arbeit relevanten Eigenschaften von MATLAB/Simulink/Stateflow sind in Tabelle 7.4 anhand der Kriterien aus Abschnitt 4.8 zusammengestellt. Die Modelldateien und die Codegenerierungsvorschriften in den TLC-Dateien sind offen gelegt und können frei verändert werden. In der Simulink-Modelldatei werden die verwendeten Blöcke, ihre Position und Verbindungen sowie Werte eventueller Parameter der Modellblöcke in Form einer ASCII-Datei gespeichert.

In der Stateflow-Modelldatei ist die grafische Beschreibung der Zustandsautomaten als ASCII-Datei gespeichert. Semantische Informationen wie beispielsweise Hierarchieebenen werden nicht explizit gespeichert, sie liegen lediglich implizit durch die Informationen über die Position im Zeichenblatt vor. Erst der Codegenerator deutet die grafische Beschreibung. Ebenfalls offen gelegt sind die Bibliotheksfunktionen und Algorithmen der Modellelemente. Eine Schnittstelle zur Oberfläche ist vorhanden, sodass neue Modellelemente definiert werden können. Eine Veränderung bestehender Modellelemente ist nur mit sehr großem Aufwand möglich.

Tabelle 7.4: Eigenschaften der Werkzeugumgebung MATLAB/Simulink/Stateflow

Kriterien		**MATLAB/Simulink**	**MATLAB/Stateflow**
Kriterium 1:	Ist eine offen gelegte Schnittstelle zum Modell vorhanden und enthält das Modell verwertbare semantische Informationen?	ja nein	ja nein
Kriterium 2:	Ist ein offen gelegter Codegenerator vorhanden?	ja	ja
Kriterium 3:	Sind offen gelegte Bibliotheksfunktionen der Modellelemente vorhanden?	ja	ja
Kriterium 4:	Ist eine offen gelegte Schnittstelle zur Oberfläche vorhanden?	eingeschränkt	eingeschränkt

7.2.2 Betrachtung des generierten Codes

7.2.2.1 Simulink

Die Eigenschaften des generierten Codes sind in Tabelle 7.5 dargestellt. Aus den verknüpften Simulink-Blöcken resultiert eine sequenzielle Kontrollstruktur des generierten Codes mit nur wenigen lokalen Verzweigungen. Globale Abhängigkeiten zwischen Zweigen lassen sich nicht erkennen. Das dynamische Anlegen von Datenstrukturen bei der Initialisierung ist optional und kann über Codegenerierungs-Optionen in der Oberfläche ausgeschaltet werden. Der zyklisch ausgeführte Code selbst besitzt keine Schleifen. Die Initialisierung der Variablen der einzelnen Blöcke erfolgt jedoch über Schleifen, deren Iterationszahl statisch und im Modell definiert ist. Zusätzlich werden Compiler-Bibliotheksfunktionen aufgerufen, in denen sich Schleifen befinden. Deren Iterationszahl ist statisch und hängt von den Modelleigenschaften (wie beispielsweise Anzahl der Integrationsblöcke) ab. Die Informationen über maximale Iterationszahlen können somit für die WCET-Analyse genutzt werden.

Tabelle 7.5: Eigenschaften des generierten Codes von MATLAB/Simulink/Stateflow

Kriterien	**MATLAB Simulink**	**MATLAB Stateflow**
Kriterium 5: Sind die möglichen Pfade des generierten Codes statisch analysierbar?	ja, Compiler-Bibliotheksfunktionen enthalten jedoch Schleifen	nein, der Code enthält Rekursionen, ist somit nicht ohne weiteres Zusatzwissen analysierbar
Kriterium 6: Sind Änderungen am generierten Code notwendig?	nein, was generierbar ist, ist auch statisch analysierbar, jedoch Abschalten des dynamischen Anlegens von Datenstrukturen notwendig	eventuell, andere Implementierung des Event Broadcastings
Kriterium 7: Sind Pfadangaben notwendig, um die Analyse zu ermöglichen?	ja, maximale Iterationszahl der Schleifen für Modellinitialisierung und Schleifen in Compiler-Bibliotheksfunktionen	ja, Informationen zur Rekursionstiefe
Kriterium 8: Führen zusätzliche Pfadinformationen zu einer Verbesserung der oberen Schranke der WCET?	nein, da keine Abhängigkeiten zwischen Zweigen erkennbar sind	ja, Berücksichtigung der ausführbaren Pfade durch Abhängigkeiten zwischen Zweigen, die ein Ereignis aussenden oder empfangen

7.2.2.2 Stateflow

Bei dem aus Stateflow-Modellen generierten Code sind die Zustände und die möglichen Zustandsübergänge direkt im Code ersichtlich. Der Code enthält keine dynamischen Konstrukte. Da ein Zustandsautomat stets nur einen aktiven Zustand besitzen kann, sind die Zustände eines Zustandsautomaten durch eine *switch*-Anweisung implementiert. Die Zweige verschiedener Zustände schließen sich somit gegenseitig aus. Es ist keine weitere Pfadinformation notwendig, um der WCET-Analyse sich gegenseitig ausschließende Pfade mitzuteilen.

Für die Kommunikation zwischen parallelen Zustandsautomaten über das Senden von Ereignissen wird das sog. *Event Broadcasting* angewandt. Dieser Mechanismus führt zu einer sehr komplexen Codestruktur mit rekursiven Funktionsaufrufen. Beim Event Broadcasting sendet ein Zustandsautomat ein Ereignis an andere parallel arbeitende Zustandsautomaten. In Abbildung 7.7 ist dazu ein einfaches Beispiel dargestellt. Der Zustandsautomat *A* besitzt zwei parallele Zustandsautomaten *A1* und *A2*. Der Zustandsautomat A1 reagiert auf das Ereignis *Tick* von außen. Der Zustandsübergang des Zustandsautomat A1 von *Zustand_A1_S1* zum Zustand *Zustand_A1_S2* sendet das Ereignis *E1* aus. Dieses Ereignis führt bei A2 ebenfalls zu einem Übergang.

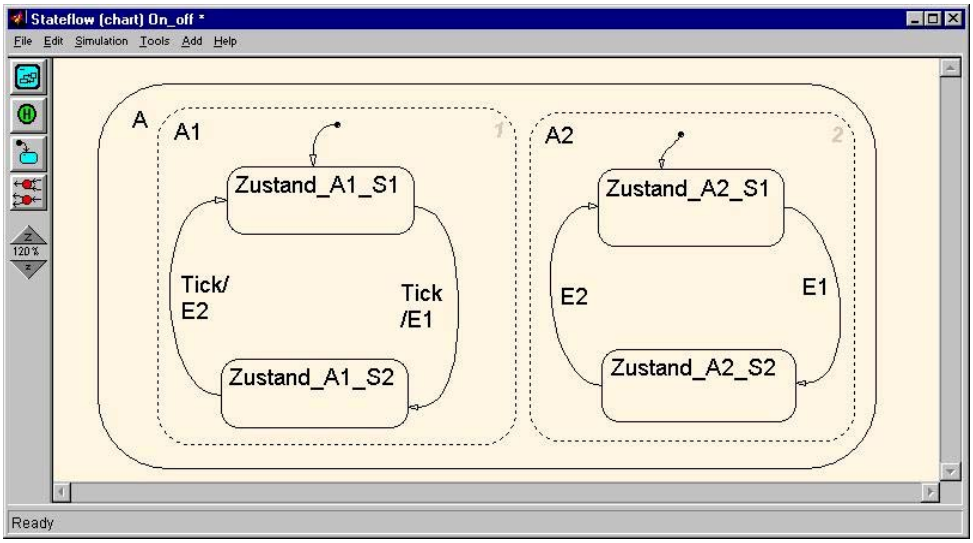


Abbildung 7.7: Event Broadcasting bei zwei parallelen Zustandsautomaten A1 und A2

Der Kontrollflussgraf des generierten Codes ist in Abbildung 7.8 dargestellt. Die Funktion des hierarchisch höchsten Zustandsautomaten A wird beim Event Broadcasting rekursiv mit dem gesendeten Ereignis als Parameter aufgerufen, der das Ereignis an die hierarchisch tieferen Zustandsautomaten A1 und A2 weiterleitet. Beim rekursiven Aufruf, der sich über mehrere Funktionsaufrufe erstreckt, werden die Pfade der parallelen Zustandsautomaten durchlaufen. Sendet ein durch Event Broadcasting ausgelöstes Ereignis selbst wieder ein Ereignis aus, dann wird ebenfalls rekursiv die Funktion des hierarchisch höchsten Zustandsautomaten (A) aufgerufen. Es entsteht eine „verkettete" Rekursion.

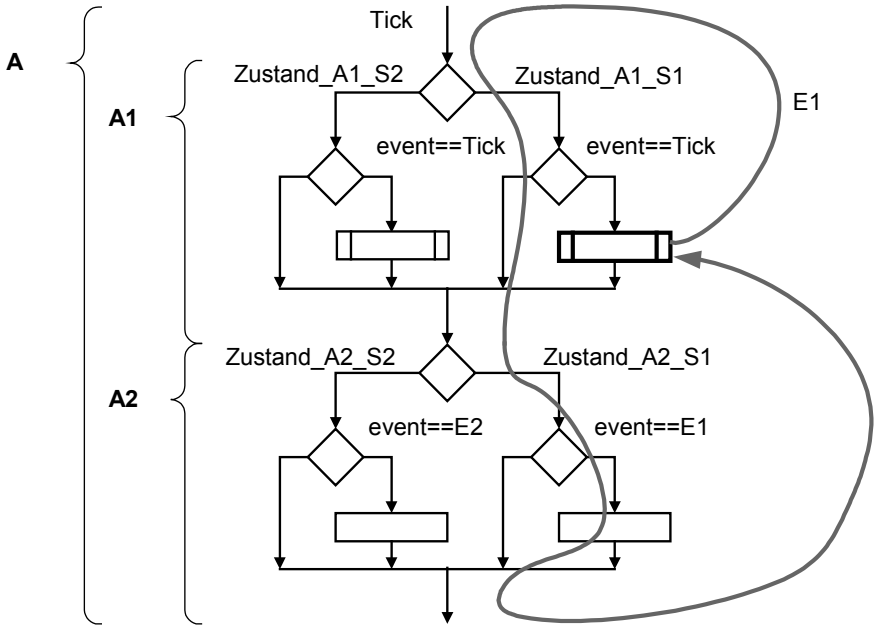


Abbildung 7.8: Kontrollflussgraf des generierten Codes des Event Broadcastings

Bei dem aus Stateflow-Modellen generierten Code sind die möglichen Pfade des generierten Codes nicht statisch analysierbar, da die maximale Rekursionstiefe auf der Assemblercode-Ebene nicht ersichtlich ist. Somit ist der generierte Code nicht ohne weiteres Zusatzwissen analysierbar. Es bleibt abzuwägen, ob eine andere Implementierung des Event Broadcastings notwendig ist oder ob diese Information aus dem Modell zur Verfügung gestellt werden kann. Weitere Verbesserungen der oberen Schranke der WCET können durch die Berücksichtigung der ausführbaren Pfade erreicht werden, sowohl mit also auch ohne Event Broadcasting. Die ausführbaren Pfade sind jedoch nicht explizit bekannt, sondern nur implizit durch Abhängigkeiten zwischen Zweigen, die ein Ereignis aussenden oder empfangen.

7.2.3 WCET-Analyse-Werkzeugkonzept

7.2.3.1 Übertragung des Konzepts

Die Betrachtung der generierten Codestruktur in Tabelle 7.5 zeigt, dass die Informationen auf Assemblerebene nicht ausreichen, um die WCET-Analyse des generierten Codes zu ermöglichen. Diese notwendigen Informationen sind jedoch auf Modellebene bzw. bei der Codegenerierung bekannt und können somit der WCET-Analyse übergeben werden. Die Umsetzung des allgemeinen Konzepts ist abhängig von den Eigenschaften der Werkzeugumgebung:

- MATLAB bietet offen gelegte Modelle und Codegenerierungsvorschriften.
- Im Modell sind jedoch keine ausreichenden semantischen Informationen enthalten, die direkt genützt werden könnten.
- Die Codegenerierungsvorschriften können sehr gut erweitert und verändert werden.

Das allgemeine Konzept lässt sich somit durch die Änderung und Erweiterung der Codegenerierungsvorschriften übertragen (siehe Abbildung 7.9). Dabei muss zwischen Stateflow und Simulink unterschieden werden. Während bei Simulink die TLC-Dateien geändert werden müssen, muss bei Stateflow der Codegenerator (sfc.m), der die TLC-Dateien erzeugt, verändert werden. Pfadinformationen müssen generiert werden und Informationen über die Beiträge der Modellelemente an Codestücken müssen an den Code angeheftet werden. Um eine Zuordnung von Pfadinformationen und Beiträgen der Modellelemente an Codestücken zu dem korrespondierenden Code zu ermöglichen, muss der Codegenerator die Pfadinformationen an den generierten Quellcode an entsprechender Stelle anheften. Zur Pfadmodellierung bietet sich die Implicit-Path-Enumeration-Methode von Li und Malik (siehe Abschnitt 3.2.1) an, mit der Abhängigkeiten zwischen Zweigen sehr gut implizit dargestellt werden können. Ein zusätzliches Werkzeug WCET2Modell übernimmt die Zuordnung der WCET-Analyse-Ergebnisse zu den Modellelementen.

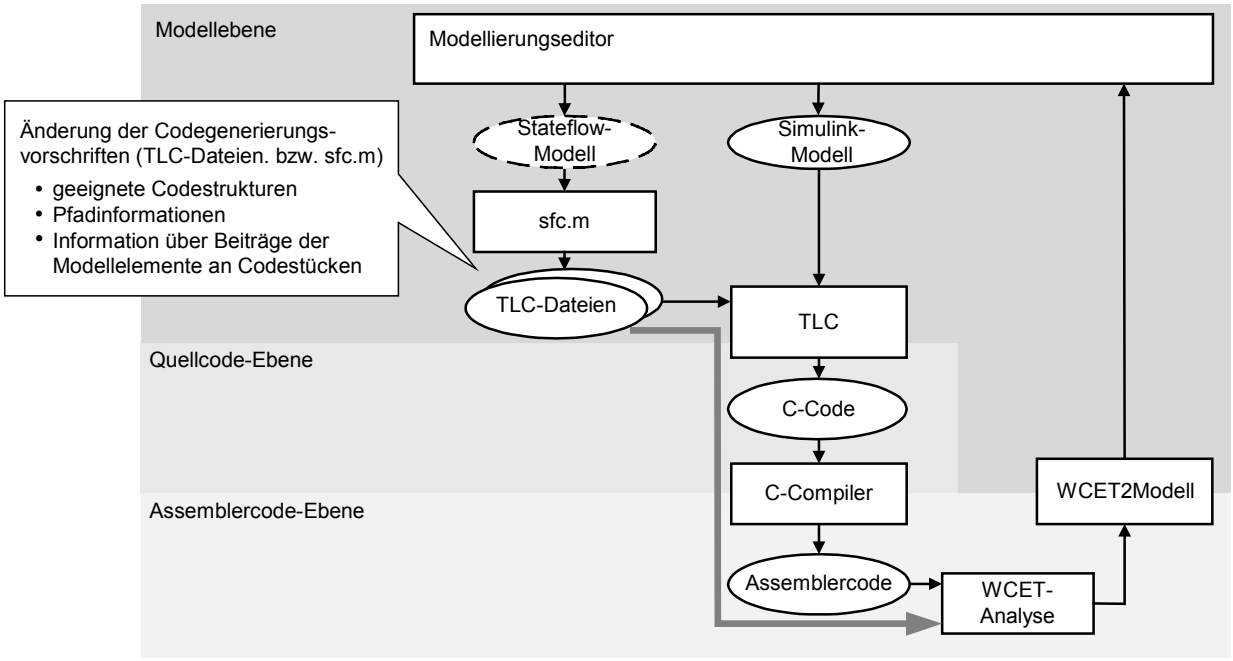


Abbildung 7.9: Anbindung der WCET-Analyse an MATLAB/Simulink/Stateflow

7.2.3.2 Generierung von geeigneten Codestrukturen

In den nachfolgenden Abschnitten werden für Simulink und Stateflow die Möglichkeiten bei der Generierung von geeigneten Codestrukturen diskutiert.

Simulink

Die generierbaren Codestrukturen sind auch statisch analysierbar. Änderungen der Codegenerierungsvorschriften sind deshalb für Simulink nicht notwendig. Simulink ermöglicht dem Anwendungsentwickler, zusätzliche Funktionen, sog. *S-functions,* zu entwickeln und einzubinden. Dabei müssen jedoch die gängigen Codierungsrichtlinien beachtet werden.

Stateflow

Die WCET-Analyse des generierten Event Broadcasting Codes ist nicht ohne Änderungen möglich. Das strikte Verbot der Verwendung des Event Broadcastings ist eine starke Einschränkung der Modellierung und wird als nicht zufriedenstellend angesehen. Für die Lösung gibt es zwei Möglichkeiten:

- Die Codegenerierungsvorschriften können derart verändert werden, dass keine rekursiven Funktionsaufrufe beim Event-Broadcasting auftreten. Dieses Vorgehen ist möglich, da die maximale Rekursionstiefe eines Aufrufs stets gleich eins ist (siehe Seite 95). Dazu muss auf Modellebene analysiert werden, welche Zustandsautomaten die gesendeten Ereignisse empfangen. Deren Funktionen müssen direkt aufgerufen werden, ohne wiederum die Funktion des hierarchisch höchsten Zustandsautomaten aufzurufen.

- Auf der Modellebene bekannte semantische Informationen über das dynamische Verhalten der rekursiven Funktionsaufrufe können der WCET-Analyse zur Verfügung gestellt werden, um die WCET-Analyse des unverändert generierten Codes zu ermöglichen.

Die zweite Möglichkeit ist der ersten vorzuziehen, da hier die Codegenerierung des Event Broadcastings unverändert bleibt, während ansonsten weitreichende Änderungen am Codegenerator vorgenommen werden müssen. Aus diesem Grund wird diese zweite Möglichkeit im nächsten Abschnitt weiter erläutert.

7.2.3.3 Generierung von Pfadinformationen

In den nachfolgenden Abschnitten wird für Simulink und Stateflow beschrieben, wie die Codegenerierungsvorschriften erweitert werden müssen, um notwendige zusätzliche Pfadinformationen für die WCET-Analyse zu erzeugen.

Simulink

Die maximale Iterationszahl der Schleifen bei der Initialisierung der Simulink-Modelle ist zwar statisch festgelegt, sie müsste jedoch für die WCET-Analyse aufwändig aus dem Assemblercode extrahiert werden. Es bietet sich daher die Möglichkeit an, diese Information direkt aus der Modellebene in einem für die WCET-Analyse lesbaren Format zu generieren. Lediglich einmal muss die Codegenerierungsvorschrift (in der TLC-Datei *roller.tlc*) für die Implementierung einer Schleife in der Programmiersprache C verändert werden. Dazu werden die interne Variable des Codegenerators `numIterations` verwendet. Auf diese elegante Weise wird für alle Schleifen automatisch die maximale Iterationszahl bei der Generierung als zusätzlicher Quellcode-Kommentar `MAX_ITERATION` angegeben. Diese Quellcode-Kommentare sind durch vier Schrägstriche gekennzeichnet und können somit von der WCET-Analyse im Assemblercode leicht identifiziert und Pfaden auf Assemblercode-Ebene zugeordnet werden:

```
...
  for (%<loopVariable>=0; %<loopVariable> < %<numIterations>;
%<loopVariable>++) ////MAX_ITERATION:%<numIterations>
...
```

Um die maximale Iterationszahl auch für aufgerufene Compiler-Bibliotheksfunktionen der WCET-Analyse angeben zu können, wurde der Quellcode-Kommentar `CALL_ITERATION` definiert. Ihm wird bei einem Funktionsaufruf die maximale Iterationszahl, die in der Compiler-Bibliotheksfunktion auftreten wird, angeheftet:

```
...
memset(b, 0, sizeof(BlockIO));      ////CALL_ITERATION:10
...
```

In der entsprechenden Compiler-Bibliotheksfunktion muss dazu die Schleife mit dem Quellcode-Kommentar `EXTERN_ITERATION` versehen werden:

```
memset( void *s, register int c, register size_t n )
{
...
      for( ; n--; *so++=(char)c )     ////EXTERN_ITERATION
...
}
```

Dieses Vorgehen setzt voraus, dass die Abhängigkeit der maximalen Schleifeniterationen von den übergebenen Daten bekannt ist.

Sind mehrere Schleifen in Bibliotheksfunktionen vorhanden, kann die Übergabe von Schleifeniterationen prinzipiell durch zusätzliche Parameter bei dem Quellcode-Kommentar `CALL_ITERATION` erweitert werden. In den Compiler-Bibliotheksfunktion muss dann beim Quellcode-Kommentar `EXTERN_ITERATION` eine Zuordnung zum Parameter erfolgen.

Stateflow

Bei Stateflow können die auf der Modellebene bzw. bei der Codegenerierung bekannten Informationen verwendet werden, um die WCET-Analyse zu ermöglichen. Durch die definierte Semantik der Zustandsautomaten ist Folgendes bekannt:

- Der Zustandsautomat wird für jedes externe Ereignis einzeln aufgerufen.
- Ein Zustand, der ein Ereignis aussendet, kann nicht selbst wieder auf dieses Ereignis reagieren. Die Rekursionstiefe ist somit pro Rekursionsaufruf auf eins beschränkt.
- Die maximale Anzahl verketteter rekursiver Aufrufe ist die maximal mögliche direkte Folge von Event Broadcasts.
- Pro Event Broadcast wird eine Rekursionsebene erzeugt. Wird innerhalb einer Rekursionsebene wiederum ein Event Broadcast durchgeführt, dann wird eine neue Rekursionsebene erzeugt. Innerhalb einer Rekursionsebene existiert nur ein Ereignis. In Abbildung 7.10 wird beispielsweise beim zyklischen Aufruf des Zustandsautomaten das Ereignis *Tick* übergeben. Erfolgt ein Event Broadcast mit dem Ereignis E1 im Zustandsautomat A1, dann wird eine neue Rekursionsebene erzeugt, in der nur das Ereignis E1 existiert. Wird ein weiteres Ereignis im Zustandsautomat A2 ausgesendet, dann wird wiederum eine neue Rekursionsebene erzeugt. Eine Rekursionsebene ist somit eindeutig durch ein Ereignis gekennzeichnet.
- Bei einem rekursiven Aufruf werden sicher nicht die Zweige durchlaufen werden, die auf ein anderes Ereignis reagieren. Es kann somit ausgeschlossen werden, dass ein aufrufender Zweig und Zweige einer tieferen Rekursionsebene, die auf andere Ereignisse reagieren, zum

selben ausführbaren Pfad gehören. In Abbildung 7.8 beispielsweise kann sicher nicht beim rekursiven Aufruf mit dem Ereignis *E1* der Pfad *Zustand_A2_S2* / *E2* durchlaufen werden.

- Es ist jedoch nicht sicher, dass Zweige, die auf ein Ereignis reagieren, stets zusammen mit dem aufrufenden Zweig ausgeführt werden. In Abbildung 7.8 beispielsweise muss nicht zwangsläufig beim rekursiven Aufruf mit dem Ereignis *E1* der Zweig *Zustand_A2_S1 / E1* durchlaufen werden. Es könnte auch der Zweig *Zustand_A2_S2 / E1* durchlaufen werden. Es hängt somit von weiteren Vorbedingungen ab, insbesondere vom aktuellen Zustand des Zustandsautomaten, welche Zweige zusammen durchlaufen werden.

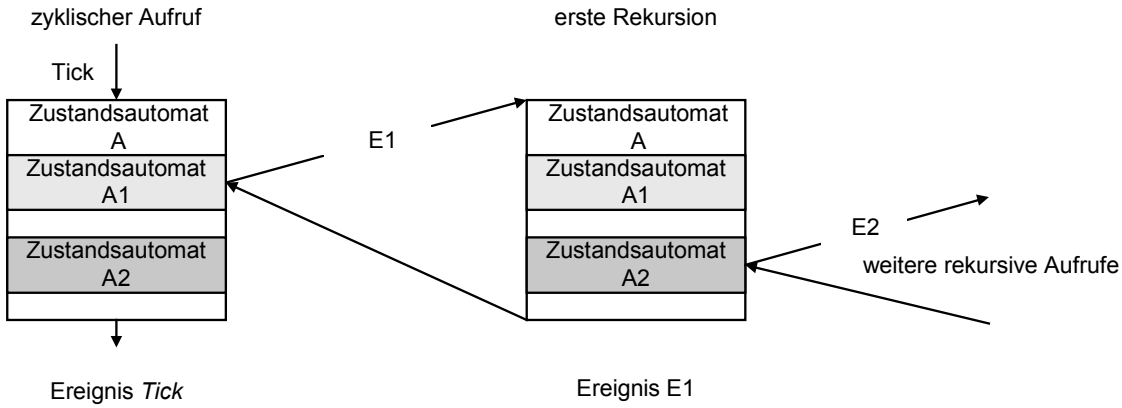


Abbildung 7.10: Austausch von Ereignissen zwischen verschiedenen Rekursionsebenen

Die bekannten Informationen auf Modellebene und Codegenerator-Ebene können durch die Erweiterung der Codegeneratorvorschriften der WCET-Analyse übergeben werden. Dazu muss die Codegenerierungsvorschrift bei einem rekursiven Funktionsaufruf wie folgt erweitert werden:

```
...
////MAX_RECURSION:1
...
```

Die Zuordnung von Zweigen zu bestimmten Ereignissen kann ebenfalls vom Codegenerator an die WCET-Analyse übergeben werden. Die Information, welches Ereignis den Event Broadcast auslöst, wird als Zweigvariable an den rekursiven Aufruf geheftet und so an die WCET-Analyse weitergeleitet. Dazu wird die interne Variable `Event` des Codegenerators verwendet:

```
...
////PATH:%<Event>
...
```

Die Zweige, die auf ein Ereignis reagieren, werden der WCET-Analyse auf die gleiche Weise mitgeteilt. Zur eindeutigen Identifizierung wird für jede Zweigvariable ein eindeutiger Bezeichner aus Zustand und Ereignis gebildet:

```
...
////PATH:%<State>%<Event>
...
```

Die Abhängigkeiten von Zweigen werden durch lineare Gleichungen bzw. Ungleichungen beschrieben (siehe Abschnitt 3.2.1, Ansatz von Li und Malik). Für jeden Zweig, der ein Ereignis aussendet, werden Abhängigkeiten zu Zweigen, die nicht auf dieses Ergebnis reagieren, aufgestellt:

```
...
////PATH_EQUATION:%<Event> + %<State>%<Event> <= %<count>
...
```

Die Variable **count** drückt hierbei aus, wie oft beim Auftreten des Ereignisses der entsprechende Pfad auftreten kann. Dies hängt von den möglichen rekursiven Aufrufen ab und kann durch die Analyse des Modells gewonnen werden. Für das Beispiel in Abbildung 7.8 ergeben sich folgende Zweiggleichungen:

```
...
////PATH_EQUATION:E1 + Zustand_A1_S2_E2 <=1
////PATH_EQUATION:E2 + Zustand_A1_S2_E1 <=1
...
```

Da bei diesem Beispiel pro Aufruf des Zustandsautomaten höchstens ein rekursiver Aufruf auftreten kann, können die aufgeführten Ereignisse nicht zu den entsprechenden Pfaden führen.

7.2.3.4 Zuordnung der WCET-Analyse-Ergebnisse zu Modellelementen

Die Initialisierung des Modells, die zyklische Ausführung und die Terminierung sind bei Simulink und Stateflow in feste Funktionen gekapselt, deren WCET auf Modellebene direkt angezeigt werden kann.

Zur Zuordnung der WCET-Analyse-Ergebnisse zu Modellelementen wird der WCET-Analyse während der Codegenerierung mitgeteilt, welche Codesequenzen welchen Modellelementen entstammen. Dazu können die Codegenerierungsvorschriften so erweitert werden, dass der Beginn und das Ende des Beitrags mit in den generierten Code einfügt werden. Dazu kann die interne

Variable `Blockname` genutzt werden:

```
////BLOCK_NAME:%<Blockname>
...
////BLOCK_END
```

Bei Stateflow kann die Ausführungszeit zusätzlich bestimmten externen Ereignissen zugeordnet werden.

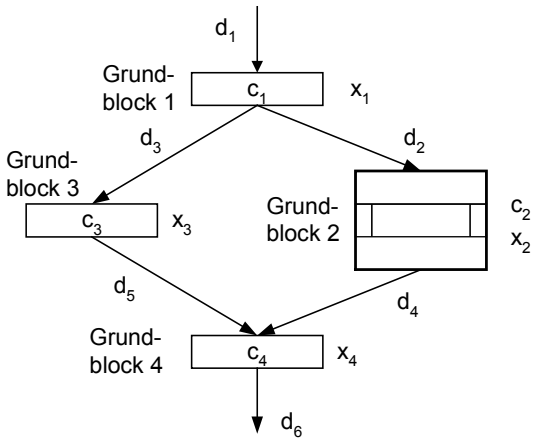
7.2.3.5 WCET-Analyse

Für die WCET-Analyse von Simulink und Stateflow Code eignet sich die Implicit-Path-Enumeration-Methode. Bei dieser Methode werden die Pfade implizit durch die Ausführungshäufigkeiten von Grundblöcken sowie durch die strukturellen und funktionalen Randbedingungen beschrieben. Rekursionen wurden bei Li und Malik nicht zugelassen. Um Rekursionen und Abhängigkeiten von Zweigen unterschiedlicher Rekursionsebenen darstellen zu können, muss die Funktion des gesamten Zustandsautomaten als ein geschlossener Kontrollflussgraf dargestellt werden. Dazu müssen alle rekursiven Aufrufe im Kontrollflussgraf in einen geschlossenen Kontrollflussgraf überführt werden.

In Abbildung 7.11 ist die Darstellung der Rekursion im Kontrollflussgraf veranschaulicht. Auf der linken Seite ist ein Kontrollflussgraf einer einfachen Funktion dargestellt. Die Grundblöcke sind mit den Ausführungshäufigkeiten x_i, den maximalen Ausführungszeiten c_i und Zweigvariablen d_i beschriftet. Die ausführbaren Pfade sind implizit durch Zweiggleichungen beschrieben. Innerhalb des Grundblocks 2 wird dieselbe Funktion rekursiv aufgerufen.

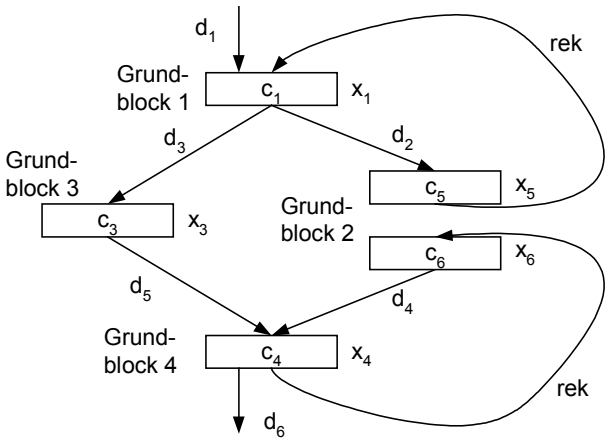
Zur Überführung in eine rekursive Darstellung wird der aufrufende Grundblock 2 aufgeteilt in den Grundblock 5 vor dem Aufruf und Grundblock 6 nachher (siehe rechte Seite der Abbildung 7.11). Zusätzlich wird ein Zweig zwischen Grundblock 5 und Grundblock 1 eingefügt. Der Zweig erhält die Gewichtung rek, die die maximale Rekursionszahl darstellt. Genauso oft wie die Funktion rekursiv aufgerufen wird, muss sie nach Beendigung auch wieder verlassen werden; deshalb wird ein Zweig von Grundblock 4 zum Grundblock 6 mit der Zweigvariable rek eingeführt.

Mit dieser geschlossenen Darstellung von rekursiven Funktionsaufrufen kann die WCET von Stateflow-Code ermittelt werden. Dazu wird der gesamte Kontrollflussgraf, mit allen aufgelösten rekursiven Aufrufen, als lineares Optimierungsproblem formuliert, das anschließend von einem externen Werkzeug [Berk] gelöst wird. Als Ergebnis wird die Ausführungshäufigkeit der Grundblöcke für den Worst-Case-Pfad zurückgegeben.



Zweiggleichungen ohne Rekursion:

$d_1=1;$
$x_1=d_1=d_2+d_3;$
$x_2=d_2=d_4;$
$x_3=d_3=d_5;$
$x_4=d_4+d_5=d_6;$



Zweiggleichungen mit Rekursion:
x_2 und c_2 entfallen, x_5,x_6 und c_5,c_6, rek kommen hinzu

$d_1=1;$
$x_1=d_1+rek=d_2+d_3;$
$x_3=d_3=d_5;$
$x_4=d_4+d_5=d_6+rek;$
$x_5=d_2=rek;$
$x_6=rek=d_4;$

Abbildung 7.11: Darstellung einer Rekursion im Kontrollflussgraf

7.3 Bewertung der Übertragbarkeit des Konzepts

Die Umsetzung des Konzepts der modellbasierten WCET-Analyse auf konkrete Software-Entwicklungswerkzeuge ist sehr stark von den spezifischen Eigenschaften der Werkzeuge abhängig. Nachfolgend werden die wesentlichen Erkenntnisse aus diesem Kapitel zusammengefasst. Die Bewertung der Übertragbarkeit des Konzepts erfolgt hierbei für die folgenden Teilaspekte des Konzepts getrennt:

- Generierung von geeigneten Codestrukturen
- Generierung der notwendigen Pfadinformationen aus dem Modell heraus und
- Zuordnung der Ergebnisse zu den Modellelementen

Generierung von geeigneten Codestrukturen

Bei der Betrachtung des generierten Codes wurde deutlich, dass bei beiden Werkzeugen aus jeweils unterschiedlichen Gründen die direkte WCET-Analyse des generierten Codes nicht möglich ist. Bei ViPER/ESTEREL ist der generierte C-Code der Object-Code-Darstellung nicht mit herkömmlichen WCET-Analyse-Ansätzen statisch analysierbar. Der generierte C-Code der Sorted-Circuit-Code-Darstellung mit seiner stark verzweigten Struktur ist für die WCET-Analyse sehr ungeeignet. Es wurde deshalb ein neuer Codegenerator entwickelt, der die negativen Eigenschaften behebt. Bei MATLAB hingegen ist der generierte Code sehr gut statisch analysierbar, sodass keine größeren Veränderungen an der Codegenerierung vorgenommen werden

müssen. Eine Ausnahme stellt die erzeugte Codestruktur beim Event Broadcasting dar, die gesondert betrachtet wurde.

Generierung von Pfadinformationen

Bei beiden untersuchten Software-Entwicklungswerkzeugen ist die Generierung von Pfadinformationen notwendig. Es muss hierbei unterschieden werden zwischen Pfadinformationen, die die WCET-Analyse erst ermöglichen und Informationen, die zu einer Verbesserung der oberen Schranke der WCET führen. Beim Vergleich von Simulink mit Stateflow und ViPER/ ESTEREL wird deutlich, dass erst, wenn eine Ablauflogik im Modell vorhanden ist, die zu unterscheidbaren Pfaden auf Codeebene führt, Pfadinformationen zu einer Verbesserung der oberen Schranke der WCET herangezogen werden können. Bei Simulink handelt es sich um weitgehend sequenziellen Code mit nur wenigen Schleifen und keinen globalen Abhängigkeiten zwischen den Zweigen im Code. Pfadinformationen zur Verbesserung der oberen Schranke der WCET sind kaum möglich, da der Code auch sehr wenige Pfade mit wenig variierenden Ausführungszeiten besitzt.

Beim Vergleich von MATLAB/Stateflow mit ViPER/ESTEREL wird ein weiterer Unterschied deutlich. Bei ViPER/ESTEREL dient der Zustandsautomat lediglich als internes Beschreibungsformat, das komplex und unübersichtlich sein darf, solange es mathematisch korrekt ist. Der Zustandsautomat ist mathematisch exakt als ein flacher Zustandsautomat beschrieben. Die Gewinnung der Pfadinformationen ist deshalb relativ einfach möglich. Bei MATLAB/Stateflow hingegen werden die Zustandsautomaten als grafisches Beschreibungsmittel zur Modellierung eingesetzt. Bei Stateflow liegt der Zustandsautomat nicht als mathematisches Modell vor, sondern lediglich als grafische Beschreibung. Die Pfadinformationen müssen deshalb bei der Codegenerierung *aufgelesen* werden. Hierbei ergeben sich lediglich implizite Informationen über Zusammenhänge zwischen Pfaden und keine expliziten Informationen, wie das bei ViPER/ ESTEREL der Fall ist. Die Art der Pfadinformationen hat wiederum Auswirkung auf die verwendete Pfadmodellierung und auf das Berechnungsverfahren.

Insgesamt hat sich bestätigt, dass auf der Modellebene das abstrakte Modell mehr explizite Information über das Ausführungsverhalten des Codes enthält als der generierte Code selbst, beziehungsweise dass diese Informationen einfacher aus dem Modell zu erhalten sind als aus dem Code selbst. Bei ViPER/ESTEREL sind im Zustandsautomaten explizit alle Pfade bekannt, die auf der Codeebene durch das Lösen einer Vielzahl boole'scher Gleichungssysteme hätte aufwändig errechnet werden müssen. Bei MATLAB/Stateflow sind auf Modellebene im Modell Rekursionstiefen und Abhängigkeiten zwischen Pfaden bekannt, die im Quellcode nicht mehr so einfach ersichtlich sind.

Zuordnung der Ergebnisse zu den Modellelementen

Die Zuordnung der WCET-Analyse-Ergebnisse zu einzelnen Modellelementen kann bei den zwei untersuchten Software-Entwicklungswerkzeugen ebenfalls erfolgen. Bei ESTEREL sind die Beiträge der reaktiven Teile nicht mehr identifizierbar, während das bei MATLAB möglich ist.

In diesem Kapitel wurde das zuvor erstellte Konzept der modellbasierten WCET-Analyse auf zwei Software-Entwicklungswerkzeuge ViPER/ESTEREL und MATLAB erfolgreich übertragen. Dazu wurden bekannte WCET-Analyse-Ansätze angewandt und erweitert. Es ist somit möglich, die während der Entwicklung zeitgesteuerter Systeme mit Software-Entwicklungswerkzeugen erstellten Modelle automatisch einer WCET-Analyse zu unterziehen. Die Umsetzung der erstellten Konzepte als prototypische Werkzeugumgebung für die Entwicklung und Analyse zeitgesteuerter Systeme wird im folgenden Kapitel vorgestellt.

8 Implementierung der Softwarewerkzeuge

Im Rahmen der vorliegenden Arbeit wurde eine Werkzeugumgebung für die Entwicklung und Analyse zeitgesteuerter Systeme erstellt. Die Werkzeugumgebung unterstützt die Entwicklung zeitgesteuerter Systeme im Hinblick auf die zeitlichen Planungs- und Analyseaktivitäten. Die Unterstützung reicht von der Planung zeitlicher Abläufe im verteilten System bis hin zur Zeitanalyse und modellbasierten WCET-Analyse von Software, die mit Software-Entwicklungswerkzeugen erstellt wurde. Im Folgenden wird nach einer Übersicht die Implementierung der Teilwerkzeuge des Werkzeugkonzepts vorgestellt.

8.1 Übersicht über die Teilwerkzeuge des Werkzeugkonzepts

Die erstellte Werkzeugumgebung besteht aus den folgenden Teilwerkzeugen (siehe Abbildung 8.1). Sie wurden im Rahmen von Studien- und Diplomarbeiten am IAS realisiert:[17]

- Grafisches Planungs- und Analysewerkzeug für zeitgesteuerte Systeme (ViETTA)
- WCET-Analyse für Software-Entwicklungswerkzeuge
  - WCET-Analyse für ViPER/ESTEREL (ViPER/WCETA)
  - und WCET-Analyse für MATLAB/Simulink/Stateflow als zusätzliche Simulink-Bibliothek
- WCET-Analyse Basiswerkzeug für den 80C167 Mikrocontroller für Assemblercode (PipeSim) und C-Code (WCETA)

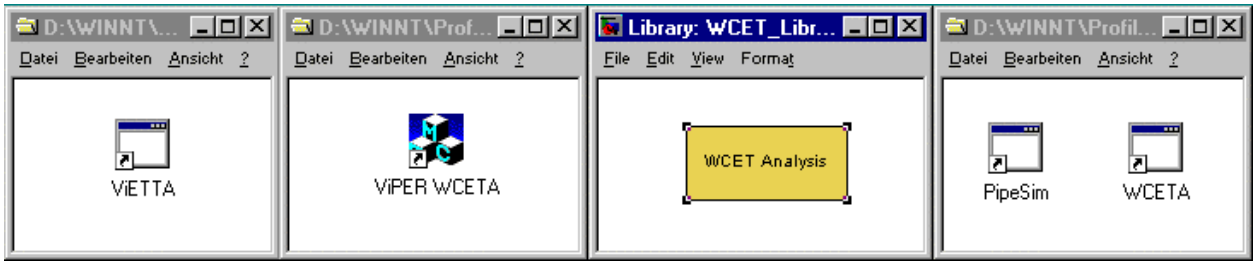


Abbildung 8.1: Übersicht über die Teilwerkzeuge des Werkzeugkonzepts

[17] Thomas Kersten: ViPER Zeitanalyseverfahren für synchrone Applikationssoftware,
Lars Braitinger: Parser für WCETA,
Jens Ruh: Grafisches Planungs- und Analysewerkzeug für zeitgesteuerte Systeme ViETTA und Weiterentwicklung von WCETA,
Bernd Heintel: WCETA und MATLAB-Anbindung

8.2 Planungs- und Analysewerkzeug für zeitgesteuerte Systeme

Das prototypische Werkzeug ViETTA wurde speziell zur Unterstützung der Planung des Ablaufs von Aktionen im zeitgesteuerten System und zur Überprüfung der Einhaltung von Zeitbedingungen entwickelt [Ruh00]. Die Implementierung von ViETTA wurde mit Hilfe von Visual C++ Version 5.0 unter Verwendung der Bibliothek Microsoft Foundation Classes (MFC) vorgenommen. Eine Datenbasis speichert alle Daten der zeitlichen Abläufe von Aktionen. Zum Einlesen der Datenbasis, die als ASCII-Datei vorliegt, wurden die frei verfügbaren Parser-Werkzeuge Flex-Bison[18] verwendet. Das Basiswerkzeug übernimmt intern die Datenhaltung und schreibt Änderungen über einen Generator zurück in die Datenbasis. Durch das Basiswerkzeug werden die weiteren Teilprogramme und Dialoge aufgerufen.

Das Basiswerkzeug

Über die Menüleiste und Schaltflächen in der Symbolleiste des Basiswerkzeugs (siehe Abbildung 8.2) wird der grafische Planungs-Editor, die Latenzzeit-Analyse und der Schedulability-Analyse-Dialog geöffnet. Die Systemparameter *Bus-Topologie*, *Kommunikationsbeziehungen* und *Schedules* können über weitere Schaltflächen bearbeitet werden.

Abbildung 8.2: Schaltflächen des Basiswerkzeugs

Der grafische Planungs-Editor

Für die Implementierung des grafischen Planungs-Editors wurde das kommerzielle Grafikprogramm Visio 5 [Micr] verwendet. Visio ist ein leistungsfähiges Grafikprogramm zur Erstellung von technischen Diagrammen. Es besitzt eine offene Programmierschnittstelle und wird über die COM-Schnittstelle[19] von dem Basiswerkzeug ferngesteuert. Dem Benutzer stehen Schablonen mit vielen vorgefertigten Symbolen, sog. *Shapes,* zur Verfügung. Die Symbole können von der Schablone via *Drag and Drop* auf das Arbeitsblatt übertragen werden.

In Abbildung 8.3 ist der grafische Planungs-Editor dargestellt. Es wurde eine neue Schablone *Zeitplanung* entworfen, die Symbole zur Planung zeitlicher Abläufe im verteilten zeitgesteuerten System enthält. Die Einplanung lokaler Kommunikationsbeziehungen erfolgt durch die Zuord-

[18] Flex ist eine frei verfügbare Version des lexikalischen Analyse Generators Lex und Bison ist eine frei verfügbare Version des Parsergenerator Yacc [Hero92]. Beide Werkzeuge stammen aus dem GNU Projekt [GNU].
[19] Microsoft Component Object Model

nung der Balken zu einem bestimmten Knoten und die Positionierung der Balken bezüglich der Zeitachse. Die Kommunikation zwischen lokalen Kommunikationsbeziehungen wird durch das Verbinden der Balken mit Pfeilen erreicht. Es ist sowohl eine globale als auch eine knotenbasierte Darstellungsart realisiert. In Abbildung 8.3 ist die knotenbasierte Darstellungsart dargestellt. Das Nachrichten-Scheduling wird durch das Verschieben der entsprechenden Rechtecksymbole über der Zeit durchgeführt. Der Editor gewährleistet dabei, dass eine sequenzielle Abfolge von Nachrichten (TDMA-Verfahren) eingehalten wird.

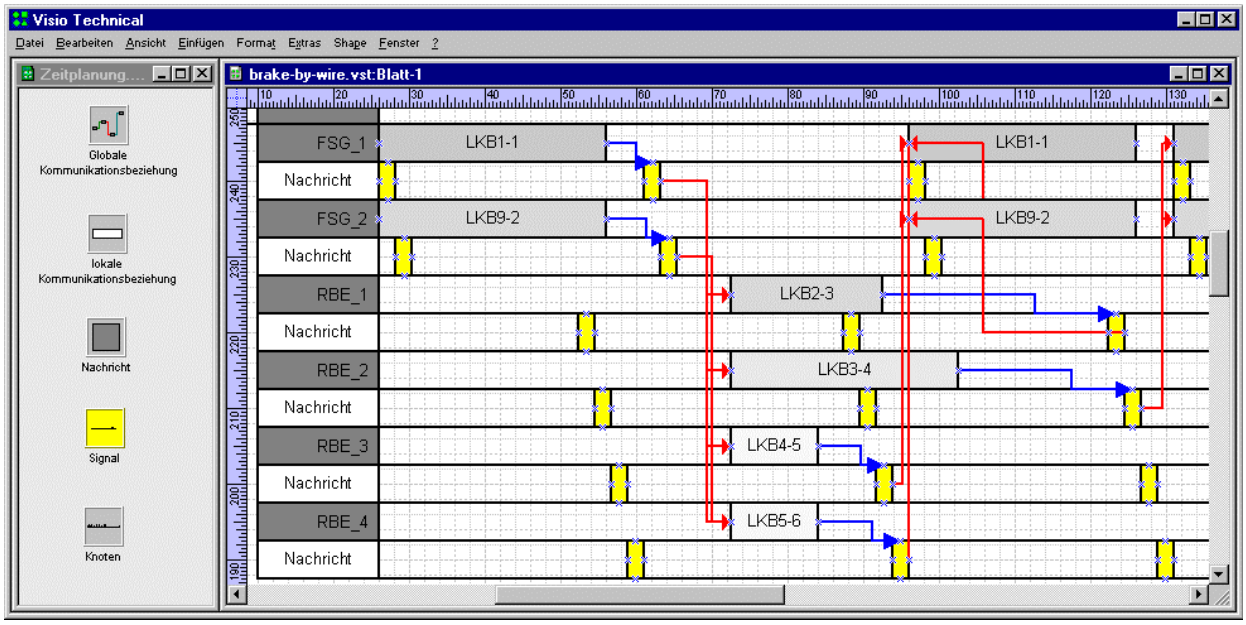


Abbildung 8.3: Grafischer Planungs-Editor für die Planung der zeitlichen Abläufe im zeitgesteuerten System

8.3 WCET-Analyse-Werkzeuge

8.3.1 WCET-Analyse-Werkzeug für ViPER/ESTEREL

Das Werkzeug ViPER/WCETA stellt der Entwicklungsumgebung ViPER/ESTEREL eine im Hinblick auf die WCET-Analyse optimierte Codegenerierung und WCET-Analyse-Funktionalität zur Verfügung [Kers99]. In Abbildung 8.4 sind Ausschnitte der Benutzungsoberfläche des Werkzeugs ViPER/WCETA dargestellt. Über die Oberfläche kann die WCET der DV-Teile der synchronen Softwarekomponenten ermittelt werden. Über den Menüpunkt *Component Library* können die Zeitanalyseergebnisse in die Komponentendatenbank eingetragen werden. Dabei können die Ausführungszeiten für eine bestimmte Zielhardware und Programmspeicher angegeben werden. Die Ausführungszeiten werden in die Datenbank eingetragen und stehen damit für die WCET-Analyse zur Verfügung.

Über den Menüpunkt *WCET-Analysis* wird die Codegenerierung durch den SSC2Ass-Codegenerator angestoßen, die Generierung von Pfadinformationen durch den OC2Path-Generator vorgenommen und dem Werkzeug WCETA übergeben. Nach erfolgter WCET-Analyse werden die Ergebnisse grafisch aufbereitet. Es werden die bei den Zustandsübergängen ausgeführten Pfade dargestellt. Die Ausführungszeiten des Pfades werden aufgeschlüsselt in die Teilbeiträge der boole'schen Gleichungssysteme (als BB gekennzeichnet) und Teilbeiträge der DV-Teile.

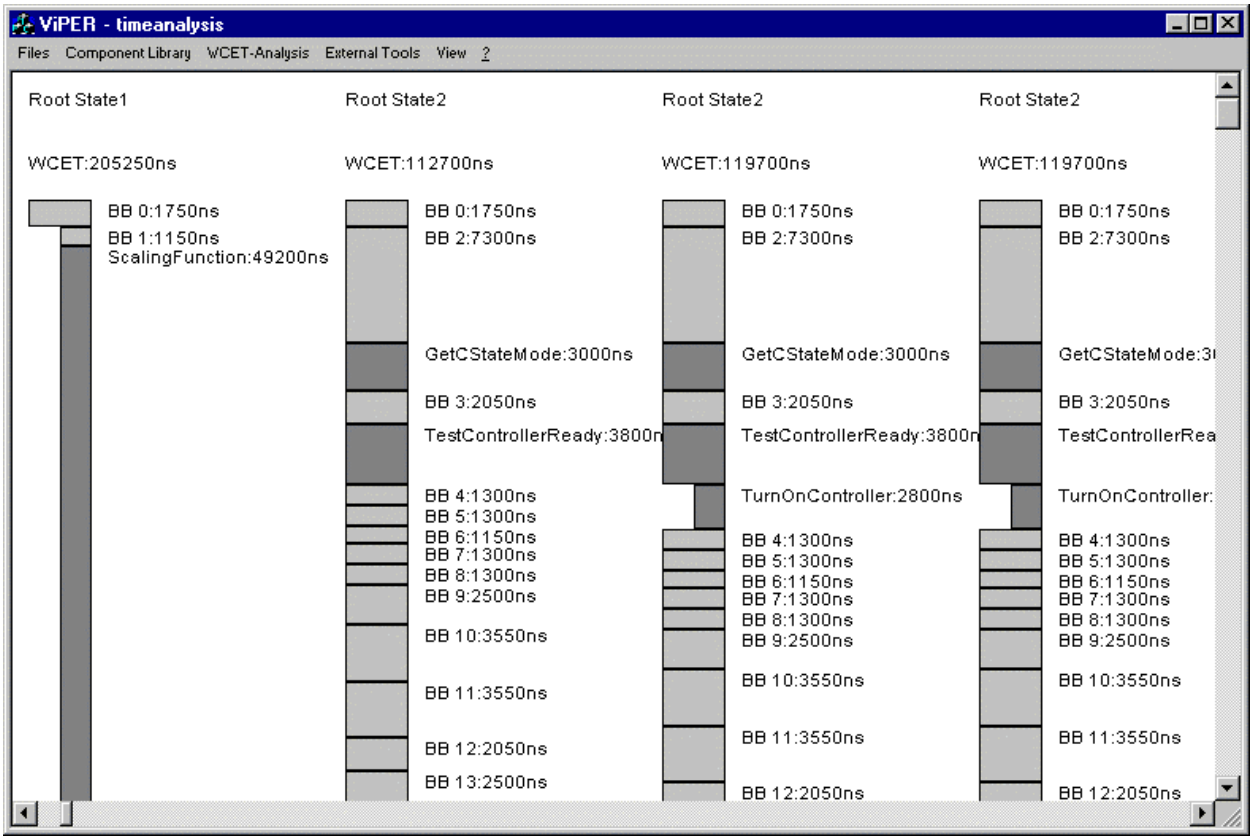


Abbildung 8.4: Darstellung der WCET von ESTEREL-Programmen

8.3.2 WCET-Analyse-Werkzeug für MATLAB/Simulink/Stateflow

Durch einen Doppelklick auf den WCET-Analysis-Block in Abbildung 8.1 öffnet sich ein Dialog, mit dem die WCET-Analyse in MATLAB gesteuert wird (Abbildung 8.5). Dieser Dialog wurde mit dem in MATLAB integrierten *GUI Layout Tool* erstellt [Hein00]. Hinter den Schaltflächen *WCET-Analysis* und *Show-Results* verbergen sich MATLAB-Programme, mit denen auf die gesamte Funktionalität von MATLAB und auch auf Windows-Kommandos zugegriffen werden kann. Per Knopfdruck wird die Codegenerierung einschließlich Generierung der Pfadinformationen, die Compilierung des erzeugten Codes und die WCET-Analyse aktiviert. Hinter der Show-Result-Schaltfläche verbirgt sich ein MATLAB-Programm, das die von der WCET-Analyse textuell gespeicherten WCET-Analyse-Ergebnisse als MATLAB-Balkengrafik aufbereitet.

Um die WCET-Analyse des aus Stateflow generierten Codes zu ermöglichen, wurde der MATLAB/Stateflow Codegenerator um die Generierung von Pfadinformationen erweitert. Die Gewinnung der notwendigen Pfadinformationen erfolgt während des Codegenerierungsprozesses. Zunächst wird die Modelldatei nach Ereignissen untersucht, die an einem Event-Broadcasting beteiligt sind. Anschließend werden einerseits die Zweige gekennzeichnet, die einen Event Broadcast auslösen, sowie die Zweige, die auf ein Ereignis reagieren. Andererseits werden die Zweiggleichungen erstellt, die die Abhängigkeiten zwischen Zweigen ausdrücken. Die Zweigvariablen werden aus den Bezeichnern der Zustände und Ereignisse ermittelt. Der eigentliche Code des Codegenerators bleibt dabei unverändert. Zur Speicherung der Zweigvariablen werden im Codegenerator zusätzliche globale Matrizen angelegt. Dabei werden die Variablen der Zweige die ein Ereignis auslösen und die Variablen der Zweige die auf das Ereignis reagieren, getrennt gespeichert. Nach Abschluss des Codegenerierungsvorganges werden die Zweiggleichungen erzeugt.

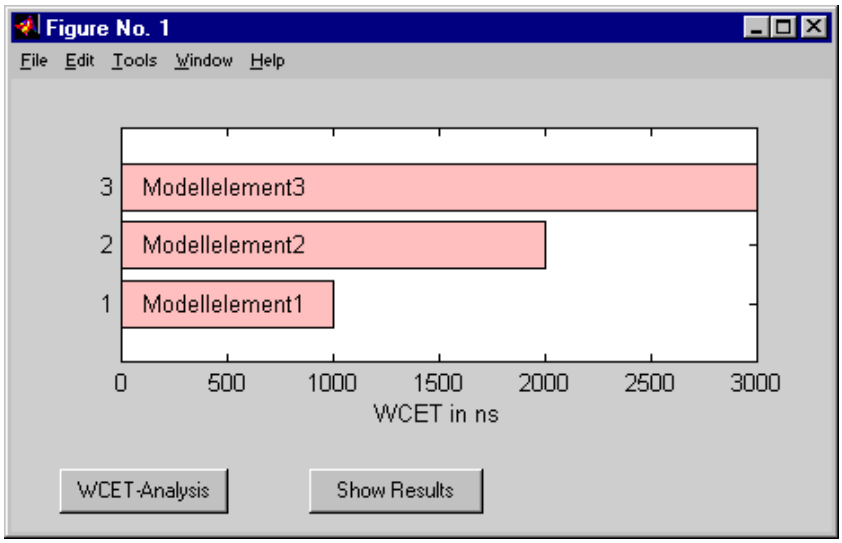


Abbildung 8.5: Benutzungsoberfläche der WCET-Analyse für MATLAB/Simulink/Stateflow

8.3.3 Basiswerkzeug WCETA

Das Werkzeug WCETA [Hein00] führt die WCET-Analyse von C-Programmen durch, die entweder von Hand geschrieben wurden oder von dem Software-Entwicklungswerkzeug stammen. Es besitzt selbst keine Oberfläche, sondern wird von den anderen Werkzeugen aufgerufen. Das Werkzeug selbst verarbeitet Assemblercode, der zuvor vom Compiler erzeugt wurde. Es berechnet die WCET und stellt die Ergebnisse für die Übergabe zur Modellierungsebene bereit. Es verarbeitet Schleifen sowie Rekursionen und erwartet Pfadinformationen und Informationen über Beiträge von Modellelementen als zusätzliche Quellcode-Kommentare. Diese Informationen können an den Assemblercode angeheftet oder in einer extra Datei abgelegt sein (im Falle

von ViPER). In Abbildung 8.6 sind die Komponenten des Werkzeugs WCETA dargestellt. WCETA führt die Strukturanalyse und die Pfadanalyse durch und nutzt die Grundblock-Analyse für den 80C167 und das frei verfügbare Werkzeug *lp_solve* [Berk]. Die externen Programme werden von WCETA gestartet, der Datenaustausch erfolgt über ASCII-Dateien.[20]

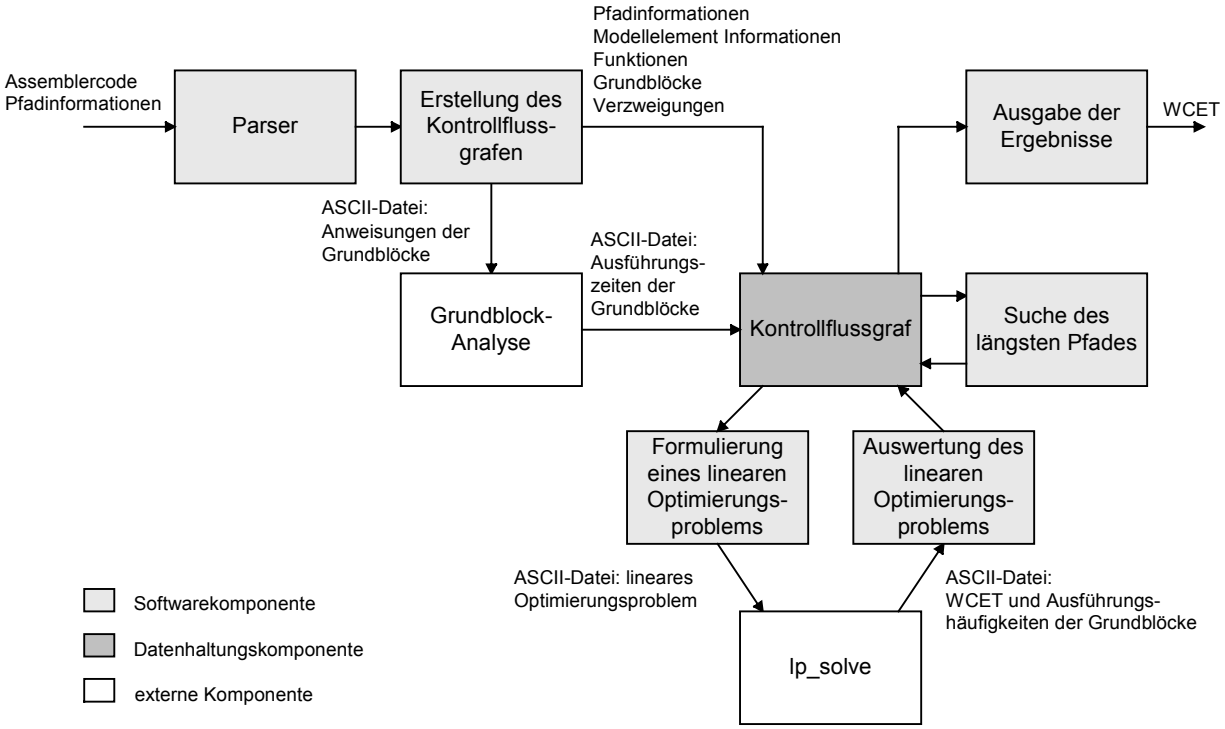


Abbildung 8.6: Systemarchitektur des Werkzeugs WCETA

Strukturanalyse

Ein Parser, der ebenfalls mit Hilfe des Parsergenerators Flex-Bison erstellt wurde, analysiert den Assemblercode. Er identifiziert Assembler-Instruktionen, Grundblöcke und Sprünge sowie die Pfadinformationen und erzeugt daraus einen Kontrollflussgrafen. Der Kontrollflussgraf ist die zentrale Datenstruktur, die schrittweise mit Informationen gefüllt wird.

Die wesentlichen Elemente des Kontrollflussgrafen und deren Attribute sind in Abbildung 8.7 dargestellt. Ein Programm besteht aus mehreren Funktionen, die wiederum aus einer Reihe von Grundblöcken (*Basic Block*) bestehen. Jede Funktion kennt den ersten Grundblock und alle Grundblöcke, in denen der Rücksprung erfolgt. Ein Grundblock kennt seinen sequenziellen Nachfolger im Programm und im Falle einer bedingten Sprunginstruktion den Grundblock des Sprungziels. Ein Grundblock nimmt die ermittelte WCET und die Überlappungszeit mit seinen Nachfolgern auf. Zusätzlich werden in den Grundblock die Pfadangaben gespeichert (maximale

[20] Die ASCII-Dateien als Schnittstellen ermöglichten die unabhängige Entwicklung und Test der Teilprogramme.

Ausführungshäufigkeit, maximale Rekursionszahl, Pfadgleichungen und die maximale Iterationszahl in einer aufgerufenen Funktion).

Zusätzlich werden die Instruktionen der Grundblöcke an die Grundblock-Analyse übergeben. Sie werden hierzu in einer ASCII-Datei abgespeichert. Die Grundblock-Analyse liefert als Ergebnis die WCET jedes Grundblocks und dessen Überlappungszeit mit dem nachfolgenden Grundblock. Mit diesen Zeitinformationen wird der Kontrollflussgraf versehen.

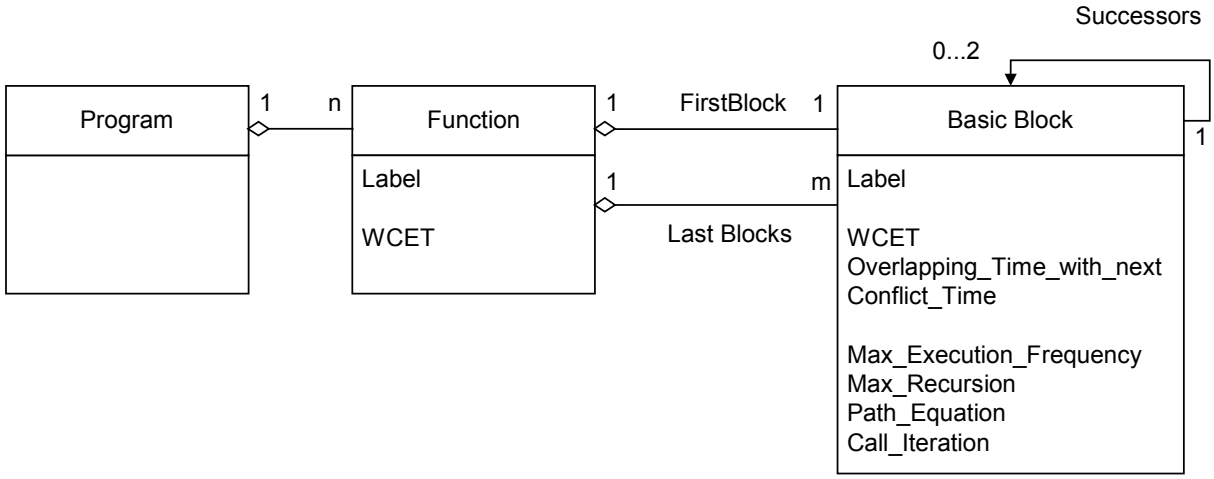


Abbildung 8.7: Elemente der internen Datenstruktur des Kontrollflussgrafen als Klassendiagramm in UML Notation

Pfadanalyse

Je nach Art der vorliegenden Pfadinformation kann zwischen den Berechnungsverfahren, Suche im Kontrollflussgraf oder Formulierung eines linearen Optimierungsproblems gewählt werden. Für die Formulierung des linearen Optimierungsproblems wird zunächst die oberste Funktion bzw. *main*-Funktion gesucht und die zugehörige Zielfunktion formuliert. Anschließend wird die Abbildung der Struktur in ein lineares Optimierungsproblem vorgenommen, indem für jeden Grundblock Strukturgleichungen erzeugt werden. Dazu werden die Summengleichungen der eingehenden und ausgehenden Kanten aufgestellt. Zusätzlich werden die übergebenen Pfadinformationen als Gleichungen formuliert. Die Kanten werden mit der maximalen Iterationszahl gewichtet. Die angehefteten Zweigvariablen werden der Ausführungshäufigkeit der Grundblöcke gleichgesetzt, während Zweiggleichungen direkt übergeben werden. Anschließend wird das Programm *lp_solve* gestartet und das Ergebnis, die Ausführungshäufigkeiten der Grundblöcke und die WCET der Funktion in den Kontrollflussgraf eingetragen. Abschließend können beliebige Informationen über WCET der Funktionen, Beiträge von Modellelementen an der WCET etc. aus der Datenstruktur des Kontrollflussgrafen berechnet werden.

8.3.4 Grundblock-Analyse

Die Grundblock-Analyse hängt stark von der verwendeten Hardwarearchitektur ab. Im Folgenden werden die wesentlichen Eigenschaften der Hardwarearchitektur des Mikrocontrollers 80C167 vorgestellt, der im Rahmen der vorliegenden Arbeit als Beispiel herangezogen wird. Anschließend wird auf die Modellierung zur Grundblock-Analyse eingegangen.

Eigenschaften des Mikrocontrollers 80C167
Der Mikrocontroller 80C167 besitzt eine vierstufige Pipeline. Jede Instruktion muss die vier Pipeline-Stufen (Fetch-, Decode-, Execute- und Write-Back-Stufe) vollständig durchlaufen. Im besten Fall benötigt die Ausführung einer isolierten Instruktion vier Maschinenzyklen[21]. Pipelining ermöglicht die parallele Verarbeitung von bis zu vier Instruktionen gleichzeitig. Nach gefüllter Pipeleine verlässt im besten Fall pro Maschinenzyklus eine Instruktion die Pipeline. Die minimalen Ausführungszeiten der Instruktionen, bei Ausführung im internen Speicher ohne Pipelinekonflikte, sind im Prozessorhandbuch [Siem96, Siem97] angegeben. Die Ausführungszeit einer Instruktion verlängert sich in den folgenden Fällen:

- **Pipelining-Konflikte:** Durch direktes Aufeinanderfolgen bestimmter Instruktionen, die gleichzeitig auf interne Register zugreifen wollen, kommt es zur Verzögerung der Instruktionsausführung.

- **Verzweigungen:** Die Ausführungszeit eines Sprungbefehls ist abhängig davon, ob er ausgeführt wird oder nicht. Wird der Sprung nicht ausgeführt, treten keine Abweichungen vom normalen Programmfluss auf und es wird keine zusätzliche Ausführungszeit benötigt. Wird der Sprung ausgeführt, so wird die erste Instruktion des bereits in der Fetch-Stufe geladenen Grundbocks nicht weiter ausgeführt. Statt dessen muss eine neue Instruktion am Sprungziel erst in die Fetch-Stufe geladen werden. Dadurch kommt es zu zusätzlichen Verzögerungen.

- **Externer Buszugriff:** Neben dem Zugriff auf den internen Speicher kann über den sog. External Bus Controller der Zugriff auf externe Speicherbausteine erfolgen. Im Gegensatz zu Instruktionen, die im internen ROM gespeichert sind, hängt die Ausführungszeit einer extern gespeicherten Instruktion von der Länge der Instruktion, deren Operanden und von den Eigenschaften des externen Busses (Adress- und Datenbusbreite, Bus-Timing etc.) ab.

- **Konflikte durch externen Buszugriff:** Durch Instruktionen in unterschiedlichen Pipeline-Stufen, die gleichzeitig auf den externen Speicher zugreifen, können zusätzliche Konflikte entstehen. Die Konflikte treten beim Zugriff auf den internen Speicher nicht auf, da sie parallel erfolgen können. Zur Auflösung der Konflikte sind den Pipeline-Stufen Prioritäten für den externen Buszugriff zugeordnet. Die oberste Priorität besitzt die Write-Back-Stufe,

[21] 1 Maschinenzyklus = 2 Takte = 100 ns bei 20 Mhz

gefolgt von der Fetch-Stufe und der Decode-Stufe. Bei der Execute-Stufe treten keine Buszugriffe auf.

Simulationswerkzeug zur Grundblock-Analyse

Mit einem Simulationswerkzeug für den Mikrocontroller 80C167 wird die Grundblock-Analyse durchgeführt. Als Eingangsdaten erhält das Werkzeug Grundblöcke von Assemblerinstruktionen. Es bestimmt die maximale Ausführungszeit dieser Grundblöcke durch die Simulation der Instruktionsausführung in der Pipeline.

Das Werkzeug wurde als objektorientiertes C++ Programm unter Visual C++ 5.0/MFC implementiert. Das zugehörige Klassendiagramm ist in Abbildung 8.8 dargestellt. In der Basisklasse *Pipeline Stage* sind die generellen Eigenschaften und das generelle Verhalten aller Pipeline-Stufen festgelegt. Eine Pipeline-Stufe bearbeitet zu einem Zeitpunkt genau eine Instruktion, die bestimmte Eigenschaften wie Instruktionstyp, Operanden und Ausführungszeiten besitzt. Die Hardwareeigenschaften wie Taktfrequenz, Bus-Timing etc. sind parametrisierbar in der Klasse *Hardware* festgelegt. Die speziellen Pipeline-Stufen *Fetch Stage*, *Decode Stage*, *Execute Stage* und *Write Back Stage*, in denen spezifisches Verhalten modelliert ist, werden von der Basisklasse abgeleitet. Gesteuert wird die Simulation in der Klasse *Pipeline Simulation*, die die vier Pipeline-Stufen besitzt.

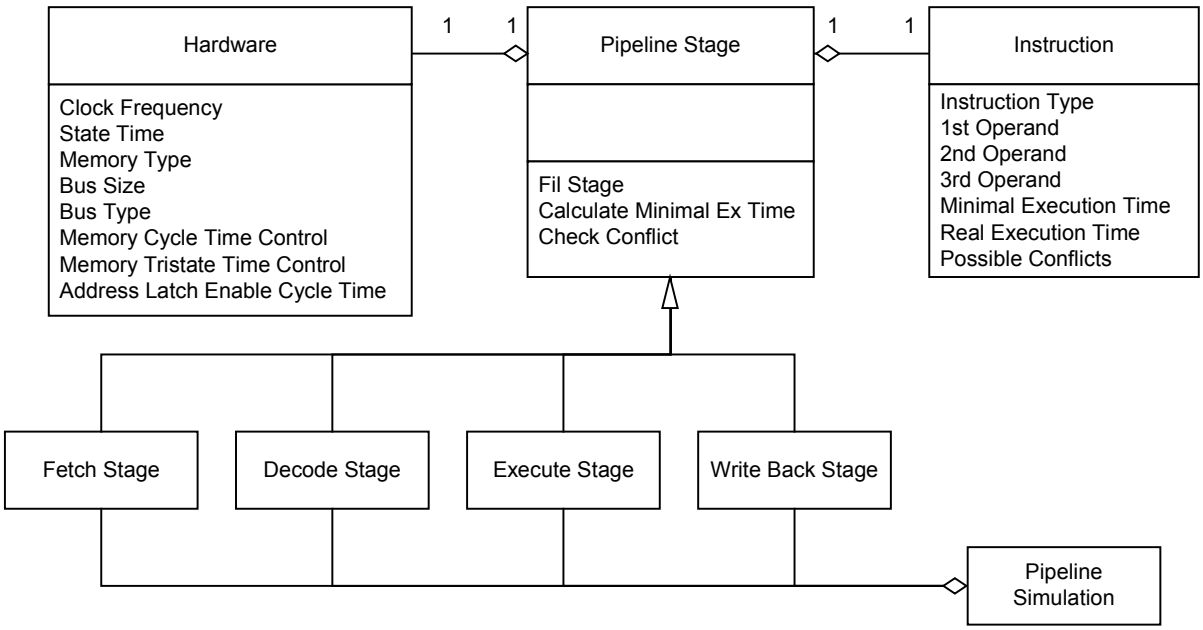


Abbildung 8.8: Klassendiagramm des Simulationsmodells der Pipeline

Simulation der Instruktionsausführung

Jede Instruktion eines Grundblocks wird einzeln eingelesen und durchläuft nacheinander die vier Pipeline-Stufen. Die erste Pipeline-Stufe (Fetch Stage) wird mit den Instruktionen gefüllt. Die nachfolgenden Stufen (Decode, Execute und Write Back) erhalten die Instruktionen von

ihren Vorgängerstufen. In den Pipeline-Stufen werden die benötigten Verweilzeiten der Instruktionen bestimmt. Dazu werden die Instruktionen und deren Operanden analysiert, um mögliche Konflikte zwischen anderen Pipeline-Stufen aufzuspüren. Dabei können bei einer Instruktion in einer Pipeline-Stufe gleichzeitig mehrere Konflikte mit anderen Pipeline-Stufen auftreten. Am Ende eines jeden Simulationsschritts werden die möglichen Konflikte verglichen, um den nächsten Simulationsschritt zu bestimmen. Wird ein Konflikt entdeckt, so wird die Verweildauer um eine bekannte Anzahl von Maschinenzyklen erhöht.

Bestimmung der WCET von Grundblöcken und Berücksichtigung von Überlappungen
Die Bestimmung der WCET von Grundblöcken und die Berücksichtigung von Überlappungen sind stark von der verwendeten Hardware-Architektur abhängig. Aufgrund der relativ einfachen Hardwarearchitektur des Mikrocontrollers 80C167 kann die Simulation der Instruktionsausführung sequentiell über alle Grundblock-Grenzen hinweg erfolgen. Die Ausführungszeiten der einzelnen Grundblöcke werden dabei folgendermaßen bestimmt (siehe Abbildung 8.9):

- Jedem Grundblock wird die Ausführungszeit zugeschlagen, vom Laden der ersten Instruktion in die Fetch-Stufe bis zum Verlassen der letzten Instruktion aus der Write-Back-Stufe.
- Zusätzlich wird die Ausführungszeit der Überlappung mit den nachfolgenden Grundblöcken aufgezeichnet.

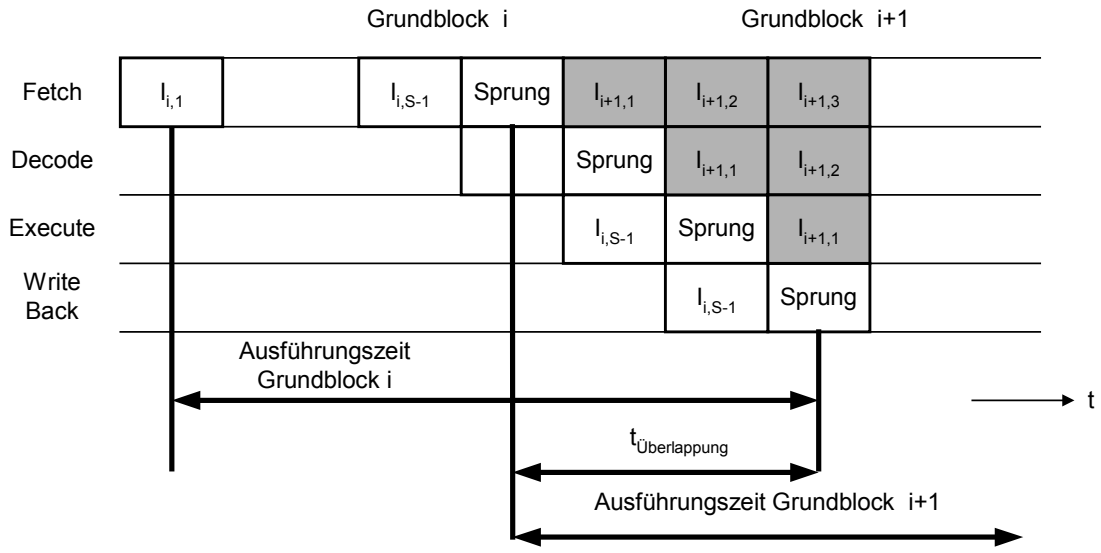


Abbildung 8.9: Zuordnung der Ausführungszeiten der Instruktionen zu Grundblöcken

Bei der Pfadmodellierung wird berücksichtigt, ob ein Sprung ausgeführt wird oder nicht. Wird ein Sprung nicht ausgeführt, so tritt keine Veränderung der Ausführung auf. Von der Kante zwischen den zwei Grundblöcken wird die Überlappungszeit abgezogen (siehe Abbildung 8.10). Wird der Sprung ausgeführt, dann wird die erste Instruktion des Grundblocks i+1, die bereits in

der Fetch-Stufe geladen wurde, nicht ausgeführt. Stattdessen muss die neue Instruktion am Sprungziel (erste Instruktion des Grundblocks i+2) erst geladen werden. Dazu wird zusätzliche Zeit benötigt. Hier treten Konflikte zwischen Grundblock i, Grundblock i+1 und Grundblock i+2 auf:

- Grundblock i hat unabhängig davon, ob der Sprung ausgeführt wird oder nicht, stets die gleiche Ausführungszeit und die gleichen Konflikte mit dem Grundblock i+1. Da die Write-Back-Stufe die höchste Priorität beim Buszugriff hat, machen sich die Konflikte nicht bei der Ausführung von Grundblock i bemerkbar, sondern nur bei Grundblock i+2.

- Die Ausführungszeit des Grundblocks i+2 ist abhängig von seinen Vorgängern. Da ein Grundblock beliebig viele Vorgänger haben kann, müssten streng genommen alle potenziellen Vorgänger beachtet werden. Die genauere Betrachtung zeigt, dass im Falle eines Sprunges nur ein Konflikt der vorletzten Instruktion des vorhergehenden Grundblocks in der Write-Back-Stufe mit der ersten Instruktion im Grundblock i+2 auftreten kann. Dieser Konflikt tritt nur auf, wenn die vorletzte Instruktion des vorhergehenden Grundblocks auf den externen Speicher zurückschreibt. Mit relativ geringerem Pessimismus ($t_{Konflikt}$ = 100 ns) wird stets ein Konflikt zwischen dem Laden der ersten Instruktion des Grundblocks i+2 und dem Zurückschreiben der Daten der vorletzten Instruktion des Vorgängers angenommen.

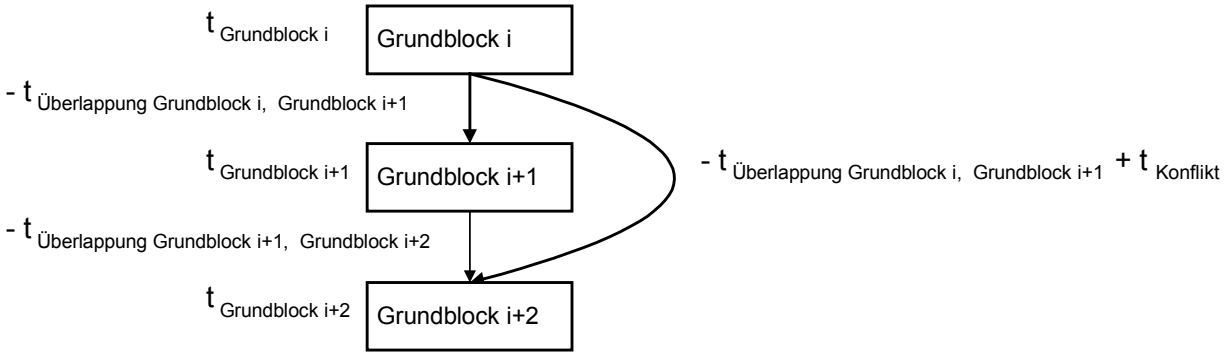


Abbildung 8.10: Modellierung der Überlappungszeit zwischen Grundblöcken

In diesem Kapitel wurde eine prototypische Realisierung der Werkzeugumgebung für die Entwicklung und Analyse zeitgesteuerter Systeme vorgestellt. Mit dieser Werkzeugumgebung ist es möglich, den Ablauf in zeitgesteuerten Systemen zu planen, Latenzzeiten für die Reglerentwicklung aus den Planungsdaten zu ermitteln und die Zeitanalyse durchzuführen. Mit dem WCET-Analyse-Werkzeug für den Siemens 80C167 Mikrocontroller kann automatisch der von den Software-Entwicklungswerkzeugen ViPER und MATLAB generierte Code analysiert werden. Die vorgestellte Werkzeugumgebung wird im nächsten Kapitel in einer Fallstudie angewandt.

9 Fallbeispiel

In diesem Kapitel wird die Anwendung des vorgestellten Verfahrens anhand eines Fallbeispiels beschrieben. Als Fallbeispiel wird ein Modell einer Steer-by-Wire-Anwendung herangezogen, das im Rahmen der vorliegenden Arbeit am IAS aufgebaut wurde. Dazu werden die im vorherigen Kapitel vorgestellten Werkzeuge eingesetzt. Anhand dieses Fallbeispiels wird zunächst die Planung eines zeitgesteuerten Systems, entsprechend dem in Kapitel 5 beschriebenen Entwicklungsprozess, vorgenommen. Anschließend wird die WCET-Analyse der verteilten Anwendungssoftware in dem zeitgesteuerten System, die mit Software-Entwicklungswerkzeugen entwickelt wurde, vorgenommen.

9.1 Beschreibung des Modellprozesses IAS-Kart

Der Modellprozess *IAS-Kart* wurde als Beispiel für eine zeitgesteuerte Steer-by-Wire-Anwendung aufgebaut [Maga98]. Dazu wurde die vorhandene mechanische Lenkung eines Kettcars durch eine elektronische Lenkung ersetzt. Das technische System besteht aus elektronischen Sensoren zur Erfassung des Lenkwunsches am Lenkrad und zur Erfassung der tatsächlichen Auslenkung an der Lenkstange sowie einem Lenkaktor, der über die Lenkstange den Lenkeinschlag der Vorderräder einstellt (siehe Abbildung 9.1). An der Lenkstange sind darüber hinaus zwei Schalter angebracht, die den Endanschlag signalisieren.

In der derzeitigen Realisierung sind Mechanik, Sensoren und Aktoren einfach ausgelegt, da die Realisierung von Fehlertoleranzkonzepten stets sekundäre Bedeutung hatte. Im Mittelpunkt steht, anhand des Modellprozesses neue Softwarekonzepte für By-Wire-Systeme anschaulich darstellen zu können sowie die Konzepte der vorliegenden Arbeit evaluieren zu können.

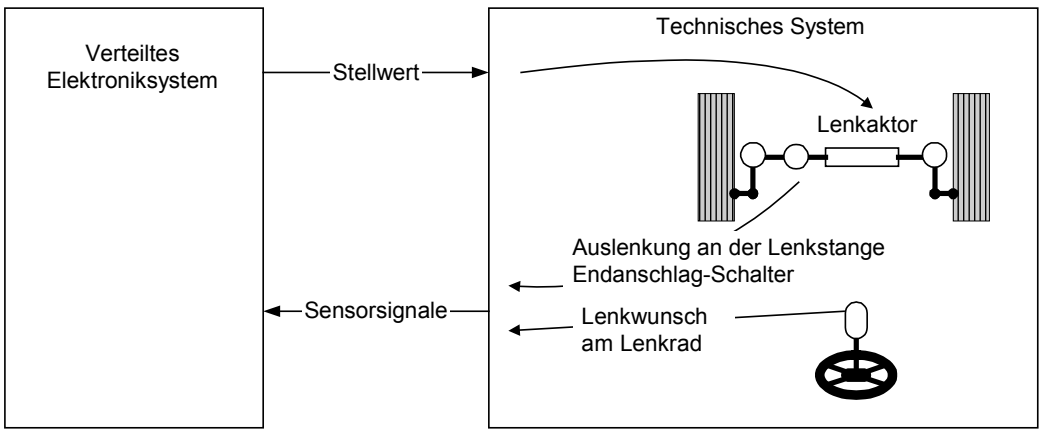


Abbildung 9.1: Systemarchitektur des Fallbeispiels IAS-Kart

9.2 Entwicklung des zeitgesteuerten Systems

In diesem Abschnitt wird das in Abbildung 9.1 dargestellte verteilte Elektroniksystem durch beispielhaftes Durchlaufen des in Abschnitt 5.3 vorgestellten Entwicklungsprozesses erstellt und die Ergebnisse der einzelnen Phasen diskutiert.

Analyse und Definition

Mit Hilfe des Elektroniksystems soll eine Steer-by-Wire-Anwendung realisiert werden. Die Regelung der Auslenkung der Räder soll dabei über ein zeitgesteuertes Elektroniksystem erfolgen. Das Lenkverhältnis soll variabel eingestellt werden. Aufgrund dieser ersten Anforderungsdefinition können die folgenden globalen Kommunikationsbeziehungen (GKB) festgestellt werden (siehe Abbildung 9.2):

- Die GKB1 erfasst die Auslenkung der Räder und berechnet das Stellsignal für den Lenkaktor.
- Die GKB2 erfasst den Lenkwunsch des Fahrers und berechnet das Stellsignal für den Lenkaktor.
- Die GKB3 erfasst das aktuelle Lenkverhältnis und berechnet das Stellsignal für den Lenkaktor.

Bei ersten prototypischen Versuchen hat sich gezeigt, dass die End-to-End Deadline von GKB1 $D_1 = 1$ ms und die Zykluszeit $t_{z1} = 1$ ms erfordert, da ansonsten ein Zittern an den Vorderrädern auftritt. Die GKB2 und GKB3 sind weniger zeitkritisch.

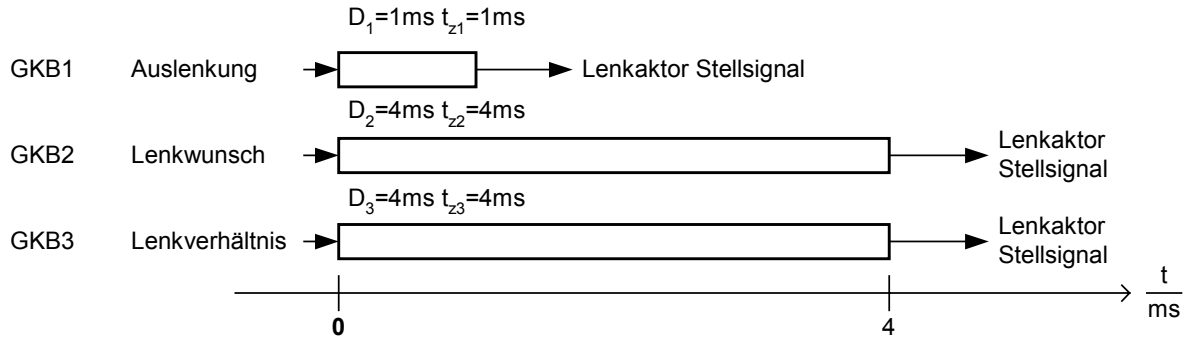


Abbildung 9.2: Darstellung der globalen Kommunikationsbeziehungen des IAS-Kart

Grobentwurf

Aufgrund der kleinen End-to-End Deadline von GKB1 muss diese lokal auf einem Knoten angeordnet sein. Die Erfassung des Lenkwunsches und der Geschwindigkeit erfolgt ortsbedingt durch verschiedene Steuergeräte. Die Forderung nach Tolerierung jedes Einfachfehlers durch

das Elektroniksystem führt zu einer doppelten Auslegung aller Steuergeräte. Somit entsteht die in Abbildung 9.3 dargestellte Elektronikarchitektur mit sechs Steuergeräten. Zur Veranschaulichung ist der Verlauf der GKB eingezeichnet.

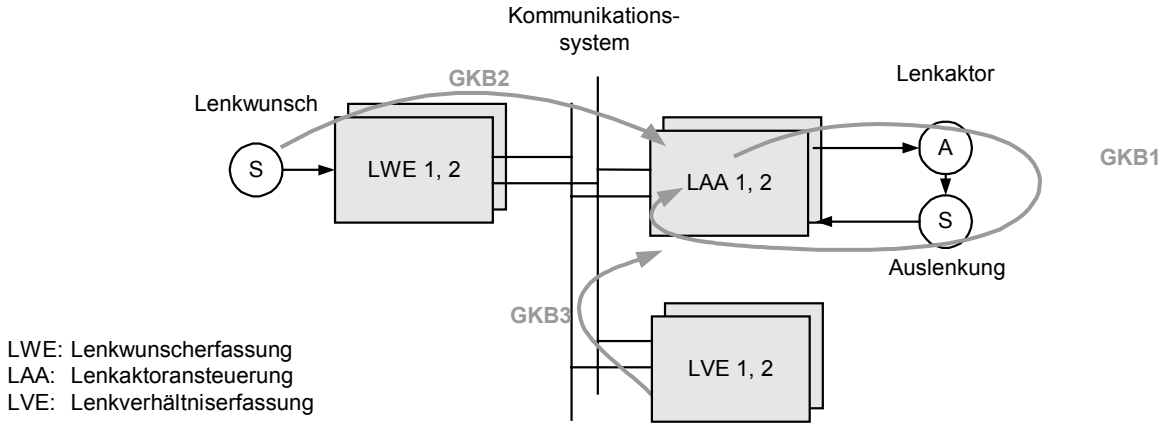


Abbildung 9.3: Elektronikarchitektur des IAS-Karts mit eingetragenen globalen Kommunikationsbeziehungen

In den GKB können bestimmte lokale Kommunikationsbeziehungen (LKB) identifiziert und bestimmten Steuergeräten zugeordnet werden. Dabei werden die Bearbeitungszeiten der LKB abgeschätzt. Daraufhin kann der Nachrichten-Schedule erstellt werden und die Definition von Zeitanforderungen an LKB erfolgen. In Abbildung 9.3 ist das Resultat dieser Schritte schematisch dargestellt. Die zyklische Abfolge der Nachrichten der Knoten ist durch die dunklen Rechtecke dargestellt. Die weißen Balken stellen Berechungsaktivitäten der GKB dar. Aus der Darstellung wird folgendes ersichtlich:

- Die GKB1 wurde vollständig auf dem Steuergerät LAA1 bzw. LAA2 angeordnet.
- Die GKB2 wurde in eine LKB2.1 auf den Steuergeräten LWE1 bzw. LWE2 und in eine LKB2.2 auf LAA1 bzw. LAA2 aufgeteilt.
- Die GKB3 wurde ebenfalls in eine LKB3.1 und LKB3.2 aufgeteilt und verschiedenen Steuergeräten zugeordnet. Die LKB2.2 und LKB3.2 werden somit zusammen mit der GKB1 ausgeführt.

Das Aufnehmen des Lenkwunsches (LKB2.1) und des Lenkverhältnisses (LKB3.1) wurde so spät wie möglich kurz vor dem Versenden der Nachrichten eingeplant. Um dieselben Messwerte zu erhalten, wurden die LKB2.1 und LKB2.2 sowie LKB3.1 und LKB3.2 auf den redundanten Steuergeräten zeitgleich eingeplant.

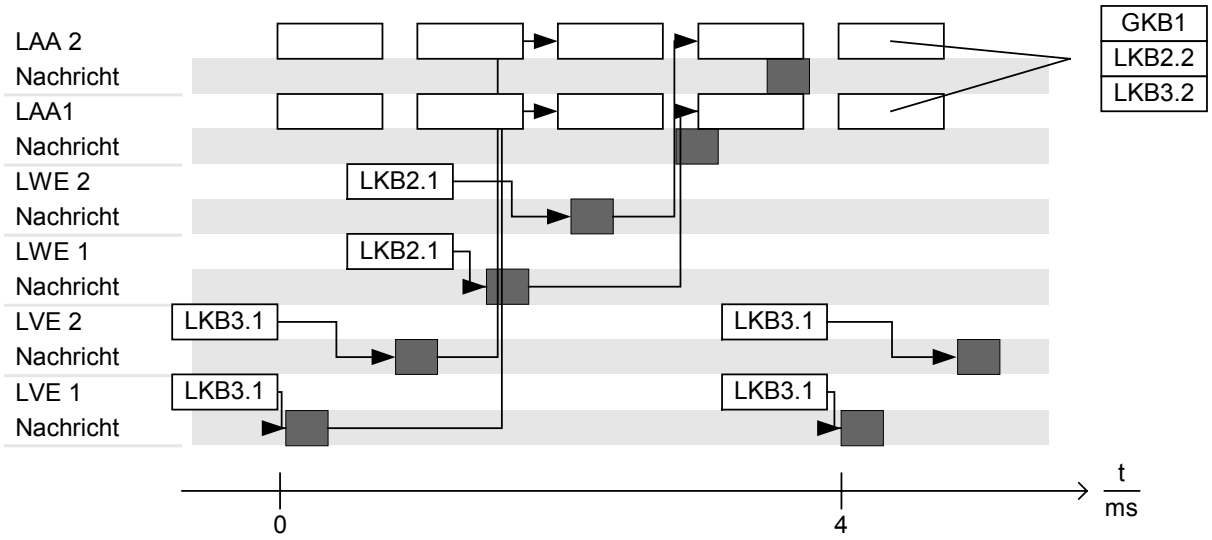


Abbildung 9.4: Nachrichten-Schedule des IAS-Karts mit eingetragenen globalen Kommunikationsbeziehungen

Feinentwurf

Beim Feinentwurf fiel die Entscheidung, die LWE-Knoten mit der Entwicklungsumgebung ViPER/ESTEREL und die restlichen Knoten mit MATLAB/Simulink/Stateflow zu entwickeln.

- Bei LWE wird der Zustandsautomat zyklisch äquidistant jeweils zu Beginn eines Sende-Slots alle 0,666 ms gestartet. Er beinhaltet alle relevanten Aufgaben eines Betriebssystems und übernimmt die Ansteuerung der Kommunikation. Die Deadline für den Zustandsautomaten ist somit 0,666 ms.

- Bei LAA werden die LKB1.1, die LKB2.2 und die LKB3.2 in weitere Tasks verfeinert (siehe Abbildung 9.5). Da die Tasks alle voneinander abhängig sind, können sie als eine Kette von Tasks (sog. Task Chain) eingeplant werden. Ihre gemeinsame Deadline ist $D_1 = D_{LKB1.1} = 1$ ms.

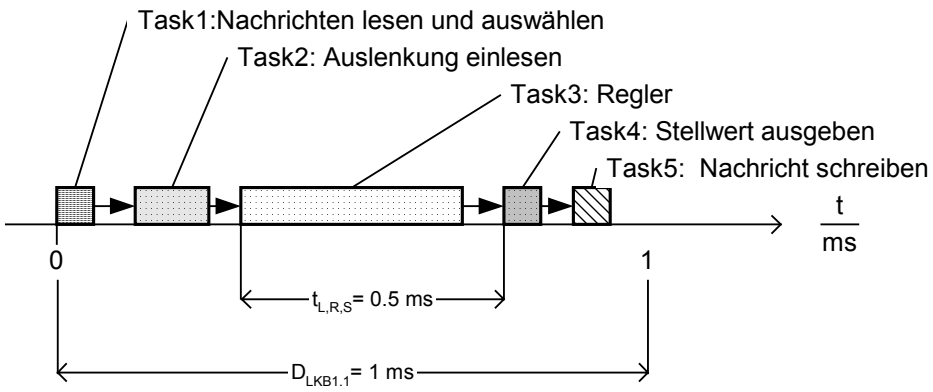


Abbildung 9.5: Lokale Kommunikationsbeziehung auf dem Knoten LAA1

Mit einer zyklischen Ausführung der Lokalen Kommunikationsbeziehung auf dem Knoten LAA1 mit 1 ms wird die anfangs geforderte Zykluszeit des Reglers von 1 ms eingehalten.

Implementierung
Die Latenzzeit-Analyse der Signale für die notwendige Parametrisierung der MATLAB-Regler ergibt folgende Latenzzeiten:

- Zwischen der Aktivierung des Reglers und Stellwert ausgeben $t_{L,R,S}$ = 0,5 ms.
- Zwischen der Lenkwunscherfassung und Aktivierung des Reglers $t_{L,LW,R}$ = 2,35 ms.

Hierbei wird deutlich, dass die GKB lediglich erste Anforderungen sind. Erst nach dem Nachrichten- und Task Scheduling sind die tatsächlichen Aktivierungszeitpunkte bekannt, die die Latenzzeiten von Signalen bestimmen.

Die WCET-Analyse der erstellten Tasks ergibt für die Steuergeräte:

- LWE: 0,257 ms für das synchrone Programm
- LAA: 0,7 ms für die Task Chain

Komponenten-Integration
In der Komponenten-Integration wird nachgewiesen, dass jede Task einer lokalen Kommunikationsbeziehung ihre Deadline einhalten kann.

- Die Task auf LWE1 bzw. LWE2 kann mit der WCET 0,25 ms die Deadline von 0,666 ms einhalten.

- Zusätzlich zu der Task Chain auf den Knoten LAA1 und LAA2 ist eine Interrupt-Quelle am Steuergerät vorhanden, die über das Eintreffen neuer Nachrichten über ein weiteres Bussystem informiert. Der Interrupt hat eine garantierte MINT von 5 ms und eine WCET von 0,1 ms. Er kann somit für die Task Chain höchstens einmal auftreten. Die Task Chain kann somit mit einer Ausführungszeit von 0,8 ms inklusive der möglichen Unterbrechung ihre Deadline von $t_{LKB1.1}$=1 ms einhalten.

System-Integration
In der abschließenden System-Integration wird insbesondere überprüft, ob die anfangs aufgestellten Zeitanforderungen an die globalen Kommunikationsbeziehungen zu dem gewünschten Lenkverhalten des Fahrzeugs führen.

9.3 Ergebnisse der WCET-Analyse

9.3.1 Übersicht über die Evaluierung der modellbasierten WCET-Analyse

In den nachfolgenden Abschnitten wird die Anwendbarkeit des in Kapitel 4 vorgestellten Konzepts der modellbasierten WCET-Analyse nachgewiesen. Die modellbasierte WCET-Analyse basiert auf der herkömmlichen WCET-Analyse, die im Rahmen der vorliegenden Arbeit ebenfalls als Werkzeug umgesetzt wurde. Es wird zunächst die Leistungsfähigkeit der angewandten Mikroarchitekturmodellierung für den 80C167 Mikrocontroller dargestellt, damit anschließend eine differenzierte Ergebnisbewertung der modellbasierten WCET-Analyse möglich wird. Für die Darstellung des Konzepts werden ausgewählte Teile des zuvor geplanten Steer-by-Wire-Beispiels herangezogen. Zur Demonstration der Konzepte aus Kapitel 7 wurde sowohl ViPER/ ESTEREL als auch MATLAB verwendet.

9.3.2 Eigenschaften der Mikroarchitekturmodellierung

Die typischen Ergebnisse der Mikroarchitekturmodellierung sind in Tabelle 9.1 für die Analyse von linearem Code, Sprüngen, Funktionsaufrufen und Rücksprüngen getrennt dargestellt. Zur Bewertung der statischen Analyseergebnisse sind ihnen Messergebnisse gegenübergestellt.

Die Messung erfolgte mit einem internen Timer, der durch Instrumentierung des Codes mit zusätzlichen Messinstruktionen gestartet und gestoppt wird. Es handelt sich um jeweils eine Maschineninstruktion, die direkt auf das interne Timer-Register zugreift. Die Instrumentierung verändert prinzipbedingt den Code und somit auch die möglichen Konflikte innerhalb der Pipeline. Bei dem verwendeten Mikrocontroller handelt es sich hierbei um ein lokales Problem, das sich nur auf die unmittelbar benachbarten Instruktionen auswirkt. Die Veränderung der Ausführungszeit beschränkt sich auf wenige Maschinenzyklen. Die Auflösung des Timers von 200 ns im Vergleich zu mindestens 400 ns für die vollständige Verarbeitung einer Instruktion ermöglicht eine genaue Bestimmung der Ausführungszeit. Durch eine Vielzahl gleichartiger Instruktionssequenzen wird es möglich, Ausführungszeiten zu ermitteln, die kleiner als die Auflösung des Timers sind.

Bei dem herangezogenen Mikrocontroller handelt es sich um eine deterministische Hardwarearchitektur. Bei wiederholtem Messen mit den gleichen Eingangsparametern eines Codestücks treten somit stets die selben Messergebnisse auf.

Tabelle 9.1: Abweichungen der statischen Analyseergebnisse von der Referenzmessung

Instruktionen	**Abweichung**
linearer Code	keine nennenswerte
Sprung, nicht ausgeführt	keine nennenswerte
Sprung, ausgeführt	+ 100 ns pro Sprung
Funktionsaufruf	+ 100 ns pro Aufruf
Rücksprung	+ 100 ns pro Rücksprung

Anhand des linearen Codes wird die Genauigkeit der Pipelinesimulation aufgezeigt. Die Ausführungszeit der analysierten Anweisungen weicht nicht nennenswert von der Messung ab. Innerhalb eines Grundblocks kann somit die WCET sehr gut bestimmt werden. Die Verarbeitung von Blockgrenzen wird durch eine Sequenz von Sprüngen dargestellt. Da der Grundblock des Sprungziels lediglich aus einer Instruktion besteht, wird weitestgehend die Verarbeitung von Sprüngen isoliert betrachtet. Hierbei ist zu unterscheiden, ob ein Sprung ausgeführt wird oder nicht. Wird der Sprung nicht ausgeführt, sind sehr gute Ergebnisse ohne nennenswerte Abweichungen zu erkennen. Demgegenüber ist eine Abweichung bei einem Sprung mit zusätzlich 100 ns = 2 Taktzyklen pro Sprung zu erkennen. Das liegt daran, dass hierbei ein Konflikt zwischen Instruktionen verschiedener Grundblöcke auftreten kann. Da bei einem Sprung Konflikte zwischen einer Instruktion am Sprungziel mit vielen Vorgänger-Instruktionen möglich sind, wurde hier stets ein Konflikt angenommen. Bei dem herangezogenen Testprogramm betrug die Abweichung insgesamt ca. 10 % der Ausführungszeit. Da typische Programme zwischen Sprunginstruktionen linearen Code aufweisen, ist der Pessimismus vertretbar. Funktionsaufrufe und Rücksprünge besitzen gleich große Abweichungen wie Sprünge.

9.3.3 Ergebnisse bei ViPER/ESTEREL

Die grafische Modellierung der Steuergeräte zur Lenkwunscherfassung in ViPER/ESTEREL ist in Abbildung 9.6 dargestellt. Die Komponente *C_COM_ENGINE* beinhaltet den Treiber des Kommunikationssystems. Außerdem startet sie die DV-Teile synchron zum Kommunikationssystem. Die Modellierung der Anwendung umfasst das Starten der Analog/Digital-Wandlung, das Lesen des Lenkwunsches von einem Port des Mikrocontrollers, die Skalierung des ermittelten Wertes sowie das Senden der Nachricht über das Kommunikationssystem. In Abbildung 9.6 sind die Zeiten der DV-Teile der Komponenten im Vordergrund dargestellt.

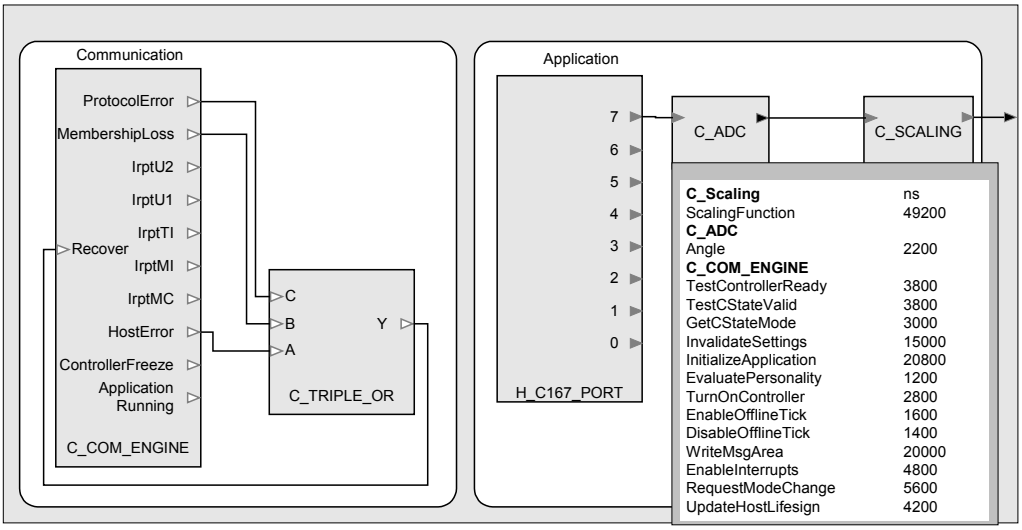


Abbildung 9.6: Grafische Modellierung der Lenkwunscherfassung in ViPER

Generierter Code

Der generierte Zustandsautomat besitzt 9 Zustände und 119 Übergänge, bei denen 14 verschiedene DV-Teile ausgeführt werden. Der Zustandsautomat wird durch 168 boole'sche Gleichungen dargestellt, die in 16 boole'sche Gleichungssysteme gruppiert sind. Der interne Zustand wird in 19 boole'schen Registern gespeichert. Der generierte Assemblercode umfasst ca. 1000 Assemblerinstruktionen. Durch den SSC2Ass-Generator werden die boole'schen Gleichungssysteme jeweils als ein linearer Grundblock dargestellt.

Generierte Pfadinformationen

Der OC2Path-Generator erzeugt die Beschreibung der 119 Zustandsübergänge durch Kombination von Aktionen. Der Worst-Case-Pfad ist beispielsweise durch folgende DV-Teile beschrieben:

```
ScalingFunction, Angle, GetCStateMode, TestControllerReady,
EnableOfflineTick, InitializeApplication, TestCStateValid,
UpdateHostLifesign
```

Bewertung der Ergebnisse

Durch generierte explizite Pfadinformationen können die Pfade exakt ermittelt werden. Theoretisch sind 2^{16} Pfade[22] möglich. Im Zustandsautomaten existieren 119 Zustandsübergänge, die zu 37 verschiedenen Pfaden führen. In Abbildung 9.7 sind die analysierten Pfade des Programms und deren Ausführungszeiten dargestellt. Die weißen Linien repräsentieren nicht ausführbare Pfade, die grauen Linien ausführbare. Die unterste Linie zeigt die BCET = 0,164 ms. Bei ihr werden keine DV-Teile ausgeführt. Die oberste Linie mit $WCET_C$ = 0,365 ms ist nicht ausführbar. Der längste ausführbare Pfad hat eine $WCET_{C,ex}$ = 0,257 ms. Die tatsächliche maximale Ausführungszeit wurde durch ausführliche Messungen zu $WCET_A$ = 0,250 ms bestimmt. Somit stellt die $WCET_{C,ex}$ eine sehr enge obere Schranke der $WCET_A$ dar. Die sehr guten Ergebnisse sind darauf zurückzuführen, dass mit dem SSC2Ass-Codegenerator langer linearer Code erzeugt wird. Die pessimistischen Annahmen bei Sprüngen wirken sich deshalb kaum aus.

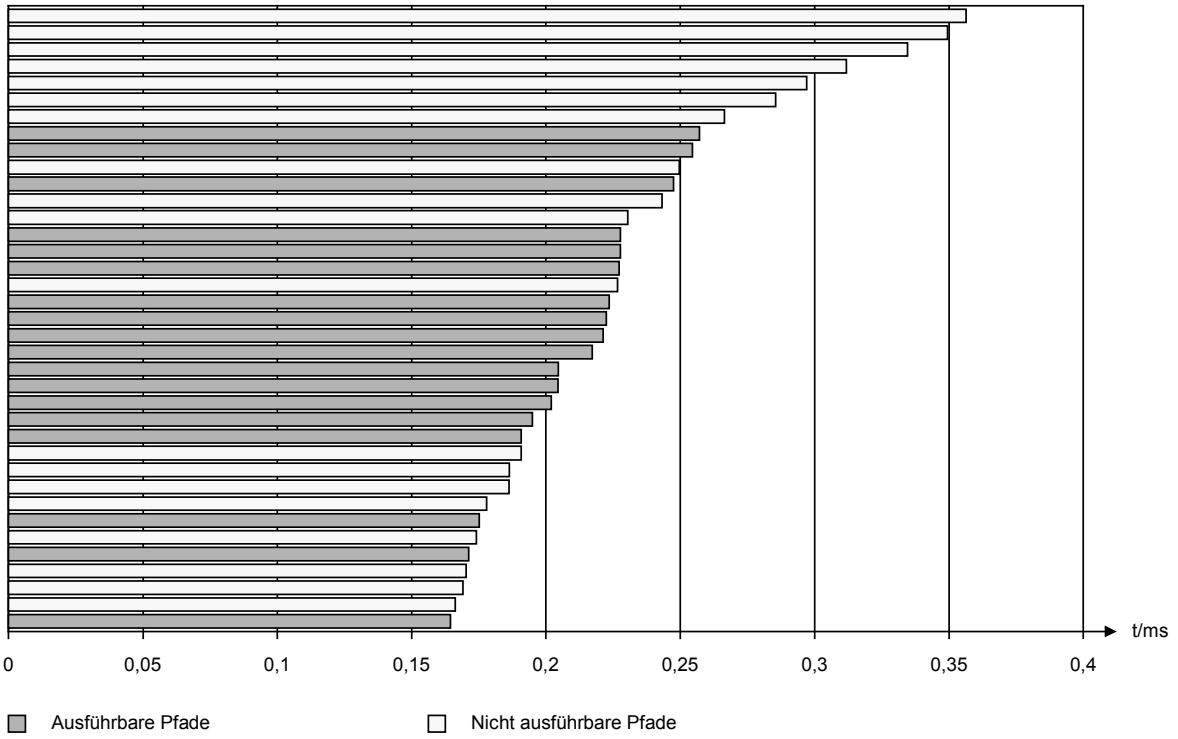


Abbildung 9.7: Ausführungszeiten der analysierten Pfade des Programms

In diesem Beispiel führt die Berücksichtigung der Pfadinformation zu einer Verbesserung der WCET um ca. 40%. Das Ausmaß der Verbesserung hängt entscheidend davon ab, welche DV-Teile zusammen ausgeführt werden können und welche nicht. Mit zunehmender Programmgröße wird im Allg. die Anzahl gemeinsam ausgeführter DV-Teile weniger stark ansteigen, als die Gesamtzahl der DV-Teile. Somit wird die Berücksichtigung der Pfadinformation zu noch stärkeren Verbesserungen der $WCET_C$ führen.

[22] Die Anzahl Pfade steigt exponentiell mit der Anzahl der booleschen Gleichungssysteme.

9.3.4 Ergebnisse bei MATLAB/Simulink/Stateflow

Die Ansteuerung des Lenkaktors wurde mit MATLAB entwickelt. Sie wird von einem zeitgesteuerten Betriebssystem [Fisc98], einem Vorläufer des OSEKtime-Betriebssystems, ausgeführt. Sie besteht sowohl aus dynamischen Anteilen, die in Simulink modelliert sind, als auch aus ereignisdiskreten Anteilen, die in Stateflow entworfen wurden.

9.3.4.1 Ergebnisse bei Simulink

In Abbildung 9.8 ist ein Ausschnitt der Modellierung des dynamischen Teils der Lenkaktoransteuerung dargestellt. Aus den vorverarbeiteten Signalen für Lenkwunsch und Auslenkung sowie dem Parameter für das variable Lenkverhältnis wird in einem P-Regler die Drehzahl für den Lenkaktor bestimmt. Der Proportionalanteil des Reglers errechnet sich aus den Zeitparametern, die zuvor in der Latenzzeit-Analyse bestimmt wurden (siehe Abschnitt 9.2). Die Verbindung zu anderen Subsystemen erfolgt durch Verbinder, die als abgerundete Rechtecke dargestellt sind. Zur Modellierung wurde die *Fix-Point Block Set Toolbox* [Math98c] verwendet, mit deren Hilfe reiner *Integer*-Code generiert werden kann.[23] Kommazahlen werden dazu mit einer festen Position des Kommas auf Integer-Datentypen abgebildet.

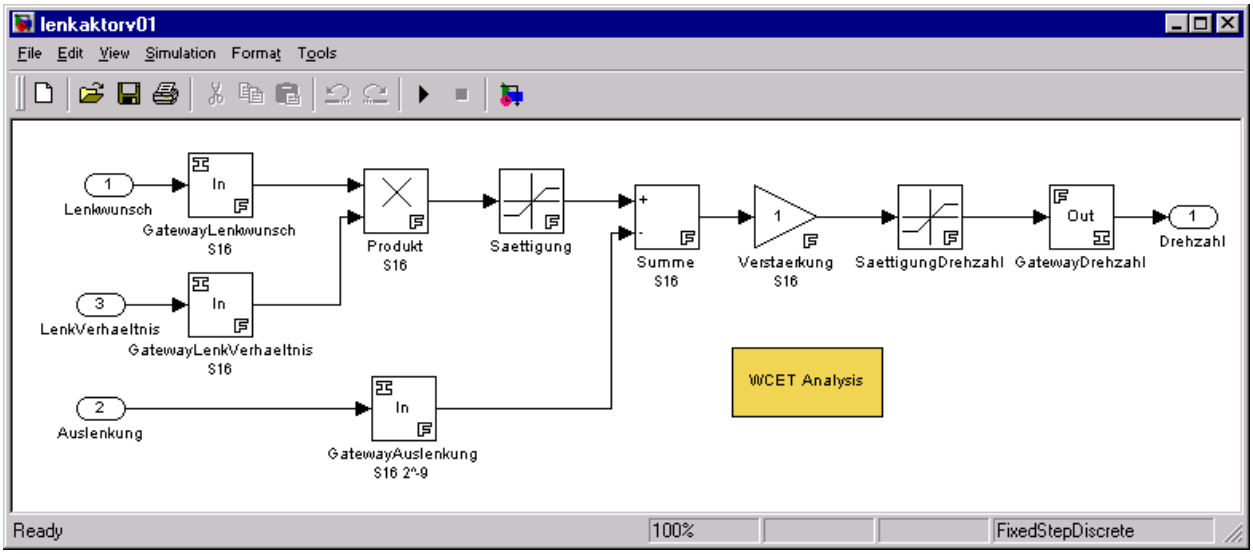


Abbildung 9.8: MATLAB/Simulink-Modell des Reglers

Generierter Code

Der generierte Code ist ohne Veränderungen seitens des Anwenders statisch analysierbar. Der generierte Code besteht aus 9 Funktionsaufrufen in 39 Grundblöcken. Er ist größtenteils linear und weist keine globalen Abgängigkeiten zwischen Pfaden auf. Bei jedem Durchlauf wird ein

[23] Sie wurde gewählt, da der verwendete Mikrocontroller keine Fließkommaeinheit besitzt. Die Verwendung des Fixed Point-Blockset stellt dabei keine Einschränkung dar, es ist ebenso die Verwendung der normalen Simulink-Blöcke möglich.

Großteil des Codes ausgeführt. Der Code, der den Sättigungsblock realisiert, enthält *if-else* Konstrukte, die jedoch auf Assemblerebene als sich gegenseitig ausschließende Pfade erkannt werden.

Generierte Pfadinformationen

Pfadinformationen werden für das Initialisieren der Variablen benötigt und für Schleifen in der Bibliotheksfunktion *memset* generiert. Zusätzlich werden die Code-Beiträge der Modellelemente gekennzeichnet. Der generierte Code ist sequentiell, sodass keine Pfadinformationen zur Einschränkung nicht ausführbarer Pfade benötigt werden.

Bewertung der Ergebnisse

In Abbildung 9.9 sind die Ergebnisse der WCET-Analyse dargestellt, wie sie dem Benutzer präsentiert werden. Die Ausführungszeit für die zyklische Ausführung ist oben ersichtlich. Die berechnete $WCET_C$ liegt mit 75,350 µs oberhalb der gemessenen maximalen Ausführungszeit von 70,600 µs. Die Abweichungen sind darauf zurückzuführen, dass in der eigentlich linearen Codestruktur zahlreiche Funktionsaufrufe auftreten. Jeder Funktionsaufruf führt zu einem zusätzlichen Grundblock und somit zu pessimistischen Annahmen an den Grenzen.

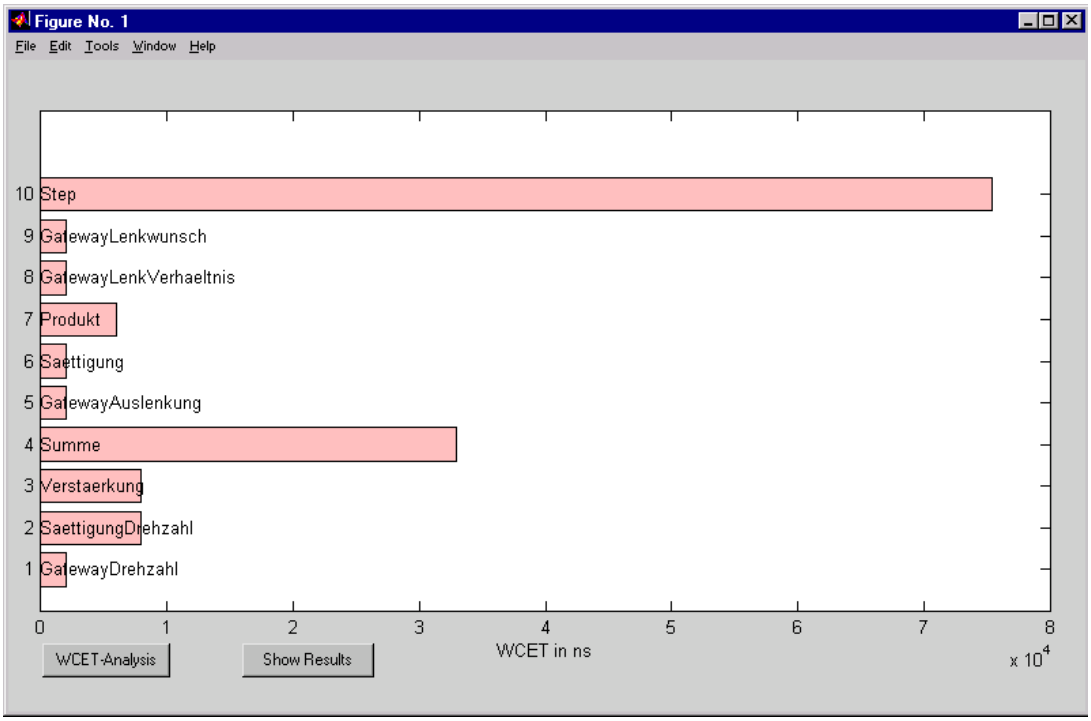


Abbildung 9.9: Darstellung der WCET von MATLAB/Simulink-Modellelementen

Im unteren Teil in Abbildung 9.9 sind die Beiträge der Modellelemente an der WCET einzeln aufgeschlüsselt. Die Aufschlüsselung stellt dem Anwender wichtige Informationen zur Verfügung. So ist beispielsweise zu erkennen, dass der Block *Summe* eine erstaunlich große Ausführungszeit besitzt. Darüber hinaus können die Änderungen an den Blockparametern genau be-

trachtet werden. So variiert beispielsweise bei Änderungen des Rundungsverfahrens beim Block *Summe* die Ausführungszeit zwischen 30 µs und 40 µs.

9.3.4.2 Ergebnisse bei Stateflow

In Abbildung 9.10 ist ein Ausschnitt der ereignisdiskreten Modellierung der Steer-by-Wire-Anwendung dargestellt. In dem Zustandsautomat *Nachrichten* ist die Anzahl vorhandener redundanter Nachrichten modelliert. Er teilt den anderen Zustandsautomaten über Event Broadcasting mit (Ereignis *N_OK* bzw. *N_N_OK*), ob mindestens eine gültige Nachricht vorhanden ist oder nicht. Der Zustandsautomat *Watchdog* steuert beim Vorhandensein mindestens einer gültigen Nachricht einen externen Überwachungsbaustein an. Ist keine gültige Nachricht vorhanden, dann geht er in den Zustand *Passiv*; die Einheit wird dann abgeschaltet. Ein weiterer Zustandsautomat *Endanschlag* modelliert die möglichen Zustände der Endlageschalter der Lenkung, der nicht direkt im Kontakt mit den anderen Zustandsautomaten steht. Im Zustand *Kein_Endanschlag* ruft er die Funktion *OK()* auf.

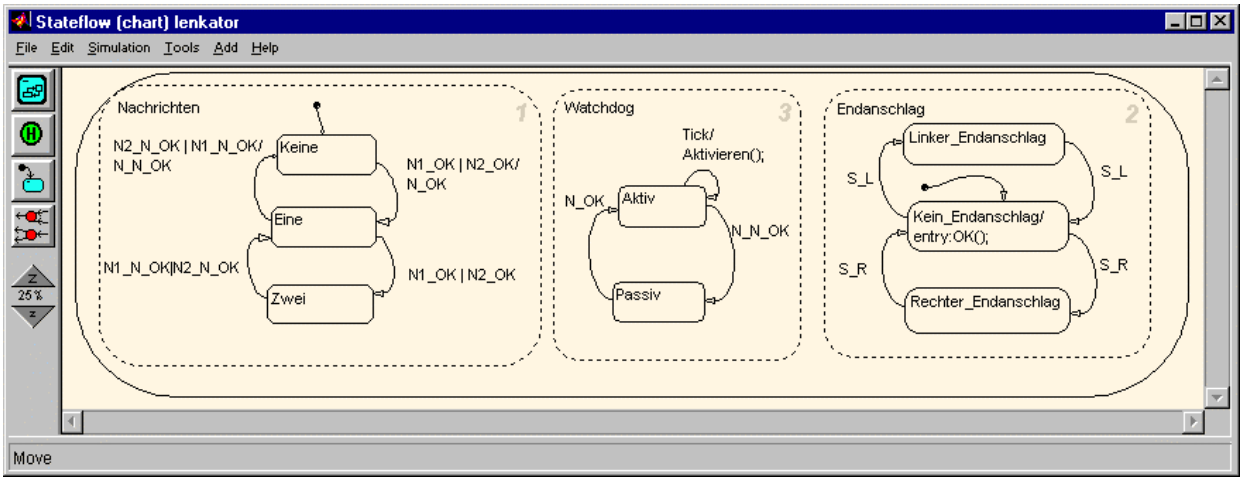


Abbildung 9.10: Modellierung von parallelen Zustandsautomaten mit MATLAB/Stateflow

Generierter Code

Die drei Zustandsautomaten Nachrichten, Watchdog und Endanschlag werden jeweils als eigenständige Funktionen implementiert, die von dem hierarchisch höchsten Zustandsautomaten aufgerufen werden. Die Implementierung des Zustandsautomaten wird für jedes der folgenden sieben externen Ereignisse einzeln aufgerufen:

```
EVENT_c1_e1_N1_OK, EVENT_c1_e2_N1_N_OK, EVENT_c1_e3_N2_OK,
EVENT_c1_e4_N2_N_OK, EVENT_c1_e5_S_L, EVENT_c1_e6_S_R,
EVENT_c1_e10_Tick
```

Aufgrund des Event Broadcastings der internen Ereignisse *N_OK* und *N_N_OK* werden hierbei zwei rekursive Aufrufe, ausgehend von dem Zustandsautomaten Nachricht, erzeugt.

Generierte Pfadinformationen

Bei der Codegenerierung werden die Zweige mit Zweigvariablen versehen, die Rekursionstiefe der Aufrufe angegeben und Zweiggleichungen generiert. Der folgende Gleichungssatz gibt beispielsweise an, dass beim Eintreten des Ereignisses N1_OK die Zweige der Zustände 2, 3, 4, 8, 9, 11, die auf andere Ereignisse reagieren, nicht ausgeführt werden können. Auf diese Weise werden für alle Ereignisse die Pfadgleichungen erzeugt.

```
...
////PATH_EQUATION:EVENT_c1_e1_N1_OK+S11Ec1_e4_N2_N_OK_c1_e2_N1_N_OK<1;
////PATH_EQUATION:EVENT_c1_e1_N1_OK+S4Ec1_e2_N1_N_OK_c1_e4_N2_N_OK<1;
////PATH_EQUATION:EVENT_c1_e1_N1_OK+S3Ec1_e5_S_L<1;
////PATH_EQUATION:EVENT_c1_e1_N1_OK+S2Ec1_e6_S_R<1;
////PATH_EQUATION:EVENT_c1_e1_N1_OK+S2Ec1_e5_S_L<1;
////PATH_EQUATION:EVENT_c1_e1_N1_OK+S8Ec1_e6_S_R<1;
////PATH_EQUATION:EVENT_c1_e1_N1_OK+S9Ec1_e10_Tick<1;
////PATH_EQUATION:EVENT_c1_e1_N1_OK+S9Ec1_e9_N_N_OK<1;
...
```

WCET-Analyse-Ergebnisse

Die WCET-Analyse ist nur möglich aufgrund der Generierung der maximalen Rekursionstiefe. Die Ergebnisse der WCET-Analyse sind in Abbildung 9.11 dargestellt. Die gesamte maximale Ausführungszeit des zyklischen Aufrufs ist im obersten Balken ersichtlich. Die WCET des Zustandsautomaten ist abhängig von dem beim Aufruf übergebenen externen Ereignis. Die Beiträge der einzelnen Ereignisse sind in den folgenden Balken getrennt dargestellt. Aufgrund der zeitaufwändigen Funktion *Aktivieren()* mit einer WCET = 76 µs ist die WCET des Zustandsautomaten für das Ereignis *EVENT_c1_e10_Tick* im Vergleich zu den WCET des Zustandsautomaten für die anderen Ereignisse sehr lange. Hier wird deutlich, dass für die Bestimmung von engen Schranken die WCET für die verschiedenen Ereignisse individuell ermittelt werden sollte. Würde stets von der WCET des Ereignisses *EVENT_c1_e10_Tick* ausgegangen, läge die berechnete WCET mit 7*101 µs ca. 100% über der tatsächlichen WCET von 355 µs.

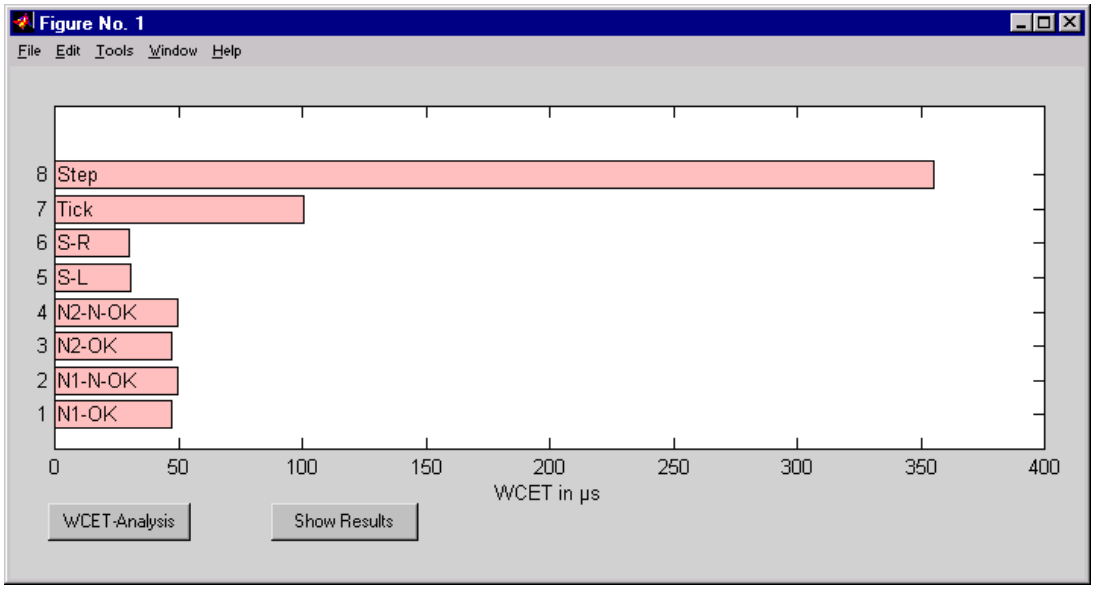


Abbildung 9.11: Darstellung der WCET bei MATLAB/Stateflow

Die Berücksichtigung der Pfadgleichungen führt zu einer erheblichen Verbesserung der oberen Schranke der WCET, wie Abbildung 9.12 zeigt. Es sind die Ausführungszeiten von ausführbaren (weiße Linien) und nicht ausführbaren Pfaden (graue Linien) für einen Aufruf des Zustandsautomaten dargestellt. Die grauen Linien entsprechen den Pfaden für die sieben externen Ereignisse. Darüber hinaus existieren zahlreiche Pfade, die nicht ausführbar sind. Bei der Anwendung des Ansatzes von Li und Malik sind die Pfade nicht explizit gegeben. Es wird stets der längste Pfad für ein gegebenes lineares Optimierungsproblem ermittelt. In Abbildung 9.12 sind die nicht ausführbaren Pfade dargestellt, die sich durch Variation der Pfadgleichungen erzeugen lassen. Darüber hinaus gibt es weitere Pfade, die aufgrund der gegebenen Eingangsdaten nicht maximal sind. Die oberste Linie stellt eine Ausführungszeit für den Fall dar, dass nur bekannt ist, dass die rekursiven Aufrufe eine Rekursionstiefe von eins haben und ansonsten keine weiteren Pfadgleichungen verwendet werden. Der resultierende Pfad schließt dann zwei Event Broadcasts, die der Ereignisse *N_OK* und *N_N_OK*, mit ein sowie die beiden Funktionsaufrufe der Funktion Aktivieren() und OK(). Die unterste Linie entspricht dem minimalen Pfad des Programms für den Fall, dass kein Ereignis angelegt wird.

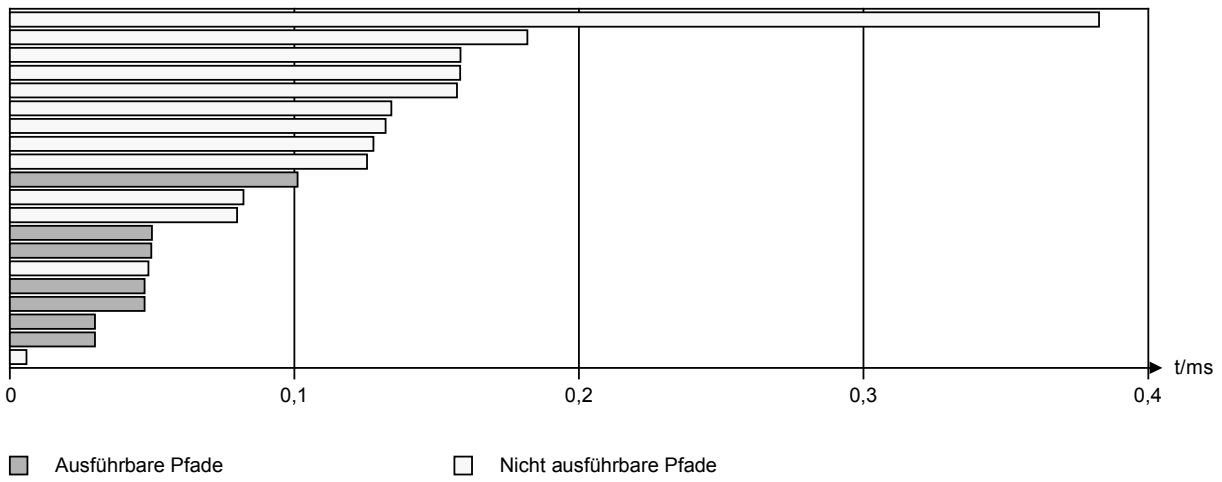


Abbildung 9.12: Ausführungszeiten der analysierten Pfade des Programms

Die Ergebnisse der WCET-Analyse sind in Abbildung 9.13 den Messergebnissen für das Ereignis EVENT_c1_e1_N1_OK exemplarisch gegenübergestellt. Die gemessene Ausführungszeit hängt sehr stark von dem Zustand ab, in dem sich der Zustandsautomat befindet. Bei dem Ereignis EVENT_c1_e1_N1_OK ist die Ausführungszeit davon abhängig, ob ein Event Broadcast ausgelöst wird (Messung 3 und 4) oder ob keiner stattfindet (Messung 1 und 2). Die WCET-Analyse liegt mit $WCET_C = 46{,}150\ \mu s$ sehr dicht an der gemessenen $WCET_A = 44{,}2\ \mu s$ (Messung 4).

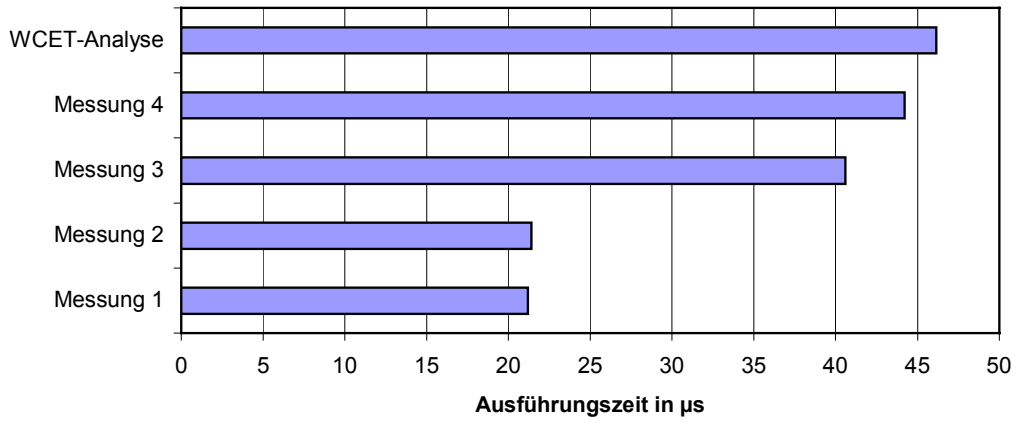


Abbildung 9.13: Vergleich der WCET-Analyse-Ergebnisse bei MATLAB/Stateflow mit Messungen

Bei der Betrachtung der Ergebnisse in Abbildung 9.13 wird ein weiterer Aspekt deutlich. Die Bestimmung der maximalen Ausführungszeit durch eine dynamische Analyse ist insbesondere bei komplexen Zustandsautomaten sehr unsicher, denn die Zustandsautomaten müssen zunächst in den Zustand gebracht werden, der anschließend zum Worst-Case-Pfad führt. Dieser Zustand ist jedoch nicht durch die Betrachtung des Codes einfach ersichtlich.

In diesem Kapitel wurde die Anwendung der vorgestellten Verfahren anhand eines Fallbeispiels beschrieben. Die Entwicklung eines zeitgesteuerten Systems wurde beispielhaft aufgezeigt und die Anwendbarkeit des in Kapitel 4 vorgestellten Konzepts der modellbasierten WCET-Analyse an konkreten Software-Entwicklungswerkzeugen nachgewiesen. Der generierte Code kann automatisch, ohne Eingreifen des Anwenders, statisch analysiert werden. Die zusätzlich generierten Pfadinformationen tragen dabei zu einer erheblichen Verbesserung der WCET bei. In dem herangezogenen Beispiel konnte bei ViPER/ESTEREL unter Zuhilfenahme der Pfadinformationen eine Verbesserung um 40% erreicht werden, im Vergleich zur WCET ohne Pfadinformationen. Im Falle von MATLAB/Stateflow wären die Ergebnisse ohne Pfadinformationen völlig unbrauchbar. Die Zuordnung der WCET-Analyse-Ergebnisse zu den Modellelementen stellt eine wichtige Information für den Anwender dar, die Entscheidungen bei Auswahl und Parametrisierung unterstützt. Im nächsten Kapitel werden abschließend die Eigenschaften des Verfahrens zur Entwicklung und Analyse zeitgesteuerter Systeme zusammengefasst und ein Ausblick auf weitere Arbeiten gegeben.

10 Schlussfolgerung und Ausblick

10.1 Eigenschaften des Verfahrens zur Entwicklung und Analyse zeitgesteuerter Systeme

In der vorliegenden Arbeit wurde ein Verfahren zur Entwicklung und Analyse zeitgesteuerter Systeme erarbeitet. Zu diesem Zweck wurde ein Konzept der modellbasierten WCET-Analyse erstellt und ein Entwicklungsprozess für zeitgesteuerte Systeme definiert. Darauf aufbauend wurde ein Werkzeugkonzept zur Entwicklung zeitgesteuerter Systeme mit Schwerpunkt auf Anbindung der WCET-Analyse an Software-Entwicklungswerkzeuge erstellt.

Entwicklungsprozess für zeitgesteuerte Systeme
Der erarbeitete Entwicklungsprozess ist eine Erweiterung des bereits in der Automobilindustrie etablierten Entwicklungsprozesses, der auf dem V-Modell basiert. Dieser wurde um zusätzliche Aktivitäten zur Planung und Analyse zeitgesteuerter Systeme erweitert. Die Planung erfolgt durch die Spezifikation von globalen Kommunikationsbeziehungen zwischen Berechnungseinheiten, die auf den Knoten des zeitgesteuerten Systems ausgeführt werden sollen. Diese werden in der Definitionsphase festgelegt und im Laufe der Entwicklung sukzessive verfeinert, bis hin zu sehr genauen Implementierungsdetails. Nach erfolgter Implementierung wird mit Hilfe der modellbasierten WCET-Analyse und der Schedulability-Analyse nachgewiesen, dass die geplanten Zeitanforderungen auch im schlechtest möglichen Fall eingehalten werden.

Das Vorgehen wurde in der vorliegenden Arbeit anhand eines einfachen Fallbeispiels dargestellt. Darüber hinaus fand es bei der Entwicklung eines ersten Brake-by-Wire-Forschungsfahrzeugs bei DaimlerChrysler [RSBH98] erfolgreich seinen Einsatz, wo bereits eine Arbeitsteilung zwischen der Forschungsabteilung des Automobilherstellers und einem Zulieferer stattfand. Wie jeder Entwicklungsprozess stellt auch der vorgeschlagene Entwicklungsprozess eine Idealisierung dar. Es wurden die wesentlichen Aspekte herausgegriffen und aufgezeigt. Darüber hinaus bedarf es bei der praktischen Umsetzung der zusätzlichen Berücksichtigung weiterer, auch firmenspezifischer Abläufe. In der Realität werden die einzelnen Entwicklungsschritte der einzelnen Phasen wahrscheinlich fließend ineinander übergehen.

Werkzeugkonzept für zeitgesteuerte Systeme
Trotz seines Prototypcharakters findet das erstellte Werkzeug Einsatz bei der Entwicklung erster By-Wire-Forschungsfahrzeuge. Insbesondere die einfache, direkte Spezifikation zeitlicher Abläufe hat großen Anklang gefunden, weit mehr als die bisher verwendeten kommerziellen Werkzeuge für die Konfiguration von Kommunikationssystemen. Die einfache und übersichtliche Darstellung der zeitlichen Abläufe im verteilten System ist eine sehr gute Grundlage bei inter-

disziplinären Diskussionen zwischen Regelungstechnikern, Kommunikationsentwicklern und Anwendungsentwicklern.

Für die Zukunft sind eine Reihe weiterer Verbesserungen denkbar. Beispielsweise sollte ein weiteres Beschreibungsmittel zur exakten Spezifikation des Zeitpunktes, zu dem Signale eingelesen, bearbeitet und ausgegeben werden, eingeführt werden. Bisher werden lediglich die zugehörigen Tasks spezifiziert.

Modellbasierte Worst-Case-Execution-Time-Analyse
In der vorliegenden Arbeit wurde ein Konzept der modellbasierten WCET-Analyse erarbeitet und erfolgreich auf einzelne Software-Entwicklungswerkzeuge angewandt. Teilweise musste die generierte Codestruktur geändert werden, um die statische Analyse zu ermöglichen bzw. zu vereinfachen. Dazu wurde eigens ein neuer Codegenerator entwickelt, der für die WCET-Analyse optimierten Code generiert. Die aus den Software-Entwicklungswerkzeugen generierten Pfadinformationen tragen sowohl dazu bei, die WCET-Analyse zu ermöglichen, als auch die Qualität der WCET-Analyse-Ergebnisse zu verbessern. Mit der Zuordnung der WCET-Analyse-Ergebnisse zu bestimmten Modellelementen wurde eine wichtige Unterstützung des Anwenders der Software-Entwicklungswerkzeuge bei Auswahl und Parametrisierung von Modellelementen geschaffen.

Als Fazit kann festgestellt werden, dass die automatische WCET-Analyse der während des Entwicklungsprozesses mit Software-Entwicklungswerkzeugen erstellten Modelle ohne die Angabe zusätzlicher Informationen durch den Anwender möglich ist. Dies wurde erreicht, indem aus den Modellen heraus Informationen über das Ausführungsverhalten des Codes gewonnen werden. Diese Informationen wären auf tieferen Codeebenen, wenn überhaupt, nur mit sehr viel mehr Aufwand zu erhalten gewesen. Die Hardwaremodellierung, das zweite Teilgebiet der WCET-Analyse, wurde im Rahmen der vorliegenden Arbeit nur am Rande betrachtet und nicht weiter entwickelt. Es wurde dennoch, um die Konzepte evaluieren zu können, eine Mikroarchitekturmodellierung für den in der Automobilindustrie weit verbreiteten Siemens 80C167 Mikrocontroller vorgenommen. Dieser wurde in jüngster Zeit zu einem leistungsfähigen Mikrocontroller weiterentwickelt, sodass dessen Einsatz auch zukünftig gesichert ist [Born01].

Das vorgestellte Konzept kann auch mit komplexer Hardwaremodellierung kombiniert werden. Gerade bei komplexen Hardwarearchitekturen ist eine exakte Pfadmodellierung extrem wichtig. Yau-Tsun Steven Li hat diesen Sachverhalt in seiner Dissertation folgendermaßen zum Ausdruck gebracht [LiMa99]:

> *"The cache modeling technique, no matter how accurate it is, will not help tightening the estimated bound, if the execution path it is modeled on does not closely follow the real execution path."*

10.2 Ausblick auf weiterführende Arbeiten

Die im Rahmen der vorliegenden Arbeit erstellten Konzepte bieten noch Potenzial für weiterführende Arbeiten.

Werkzeugkonzept für zeitgesteuerte Systeme
Bei der Entwicklung zeitgesteuerter Systeme sind Automobilhersteller und eine Vielzahl von Zulieferern durch den gemeinsamen Nachrichten-Schedule sehr stark voneinander abhängig. Während der mehrjährigen Entwicklungszeit wird die Organisation des Änderungsmanagements bei der verteilten Entwicklung eine sehr große Herausforderung darstellen. Die modernen Webtechnologien können einen entscheidenden Beitrag zur Vereinfachung der Kommunikation und des Austauschs von Entwicklungsdaten leisten. Das vorgeschlagene Werkzeugkonzept kann web-basiert realisiert werden. Der Automobilhersteller könnte das zeitgesteuerte System planen. Über eine web-basierte Datenbank könnten die Zulieferer dann die aktuellen Planungsdaten abrufen und nötigenfalls Änderungswünsche über diese Web-Schnittstelle mitteilen.

Worst-Case-Execution-Time-Analyse
Die kommerziellen Software-Entwicklungswerkzeuge besitzen sehr komplexe und umfangreiche Codegeneratoren. Das Konzept der modellbasierten WCET-Analyse wurde deshalb im Rahmen der vorliegenden Arbeit beispielhaft bei MATLAB/Simulink/Stateflow umgesetzt. Weitere Beispiele von Software-Entwicklungswerkzeugen sind Statemate [ILog], ein Werkzeug zur Modellierung von ereignisdiskreten Systemen, und ASCET-SD, ein Werkzeug, das die Modellierung sowohl ereignisdiskreter als auch dynamischer Systeme ermöglicht. Statemate verfügt über offene Schnittstellen zum Modell mit bekannter Semantik, sodass die Analyse des Modells möglich sein müsste. Bei ASCET-SD ist das Modell nicht offen gelegt und auch die Codegenerierungsvorschriften sind nicht in der benötigten Weise veränderbar, sodass hier der Hersteller des Software-Entwicklungswerkzeugs selbst das Konzept übertragen müsste.

Die vollständige Umsetzung des Konzepts erfordert einen sehr hohen Aufwand. Bei der nachträglichen Erweiterung eines bestehenden Codegenerators zur Generierung von Pfadinformationen für die WCET-Analyse können eventuelle Seiteneffekte nicht ausgeschlossen werden. Diese Erweiterungen sollten deshalb bereits vom Hersteller berücksichtigt werden. Die Hersteller werden sich zu diesem Schritt jedoch erst entscheiden, wenn ein Markt für die modellbasierte WCET-Analyse vorhanden ist. Dies wird spätestens dann er Fall sein, wenn die ersten By-Wire-Systeme mit zeitgesteuerten Systemen in Serie produziert werden.

Literaturverzeichnis

[ASU88] A. V. Aho, R. Sethi, J. D. Ullman: Compilerbau, Addison-Wesley, ISBN 3-89319-150-x, 1988

[Arin90] ARINC: *Specification 629*, Part 1, 1990

[AHP99] P. Atanassov, S. Haberl, P. Puschner: Heuristic Worst-Case Execution Time Analysis, 10th European Workshop on Dependable Computing, Wien, Österreich, 1999

[Balz96] H. Balzert: Lehrbuch der Software-Technik: Software Entwicklung, Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 1996

[BBB+00] R. Belschner, J. Berwanger, C. Bracklo, Ch. Ebner, B. Hedenetz, W. Kuffner, P. Lohrmann, J. Minuth, M. Peller, A. Schedl, V. Seefried: Anforderungen an ein zukünftiges Bussystem für fehlertolerante Anwendungen aus Sicht Kfz-Hersteller, VDI Berichte 1547: S. 23 - 42, Elektronik im Kraftfahrzeug, Baden-Baden, 2000

[BHH+98] R. Belschner, B. Hedenetz. A. Heni, J. Nell, P. Willimowski, H. Kopetz: Trockenes Brake-by-Wire mit fehlertolerantem TTP/C-Kommunikationssystem, VDI Berichte 1415: S. 325 - 344, Elektronik im Kraftfahrzeug Tagung Baden-Baden, 1998

[Berk] M. Merkelaar: lp_solve (mixed integer) linear programming problem solver, ftp://ftp.es.ele.tue.nl/pub/lp_solve

[BeGo92] G. Berry and G. Gonthier: The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation, Science of Computer Programming, 19(2): S. 87-152, November 1992

[Berr98] G. Berry: The Foundations of Esterel, Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte, editors, MIT Press, 1998

[BPG00] J. Berwanger, M. Peller, R. Griessbach: byteflight - A New Protocol for Safety Critical Applications, Proceedings FISITA 2000, F2000G316, Seoul, 2000

[BES+01] J. Berwanger, Ch. Ebner, A. Schedl, R. Belschner, S. Fluhrer, P. Lohrmann E. Fuchs, D. Millinger, M. Sprachmann, F. Bogenberger, G. Hay, A. Krüger, M. Rausch, W. O. Budde, P. Fuhrmann, R. Mores: FlexRay – The Communication System for Advanced Automotive Control Systems, SAE 2001 World Congress, 2001

[Blie00] J. Blieberger: Data-Flow Frameworks for Worst-Case Execution Time Analysis, WCET 2000 – Deutschsprachige WCET Tagung, Paderborn, 20.10.2000

[Born01] G. Born: 16-bit-Controller liefert 32-bit-Power, Elektronik 50 (2001), Heft 3: S. 52 - 56

[BSW00] J. Bortolazzi, St. Steinhauer, Th. Weber: Entwicklung und Qualitätsmanagement von Steuergerätesoftware, VDI Berichte 1547: S. 355 - 370, Elektronik im Kraftfahrzeug, Baden-Baden, 2000

[Bosc91] Robert Bosch GmbH: CAN Specification, Version 2.0, Stuttgart, 1991

[BrDr95] A. P. Bröhl, W. Dröschel: Das V-Modell: Der Standard für die Softwareentwicklung mit Praxisleitfaden, 2. Aufl. - München, Wien, Oldenbourg, 1995

[BGP98] K. Brouwer, W. Gellerich, E. Plödereder: Myths and Facts about the Efficient Implementation of Finite Automata and Lexical Analysis, Compiler Construction - 7th International Conference, CC'98, Lecture Notes in Computer Science 1383: S. 1-15, Springer Verlag, 1998

[Byte] byteflight Homepage: http://www.byteflight.com

[Daim98] Daimler-Benz AG: Presse Information zur Technologiepressekonferenz, Stuttgart-Möhringen, September 1998

[DCS] Dependable Computer Systems KEG: Homepage: http://www.decomsys.com

[DCS00] Dependable Computer Systems KEG: xDesigner Dokumentation, Wien, Österreich, 2000

[DFM+97] E. Dilger, T. Führer, B. Müller, S. Poledna, T. Thurner, X-By-Wire: Design of Distributed Fault Tolerant and Safety Critical Applications in Modern Vehicles, VDI Convertions, Wolfsburg, 1997

[dSpa] dSpace GmbH Homepage: http://www.dspace.de

[dSpa01] dSpace GmbH Paderborn: Coming Soon: Steer-by-Wire at ZF Lenksysteme, dSpace News, Spring 2001

[Echt90] K. Echtle: Fehlertoleranz-Verfahren, Springer-Verlag, Berlin, Heidelberg, New-York, 1990

[ErGu97] A. Ermedahl, J. Gustafsson: Deriving Annotations for Tight Calculation of Execution Time, Proceedings of Euro-Par'97, LNCS 1300: S. 1298-1307, Passau, 1997

[ETAS] ETAS Homepage: http://www.etas.de

[ETAS00a] ETAS GmbH & Co. KG: ASCET-SD - Development Environment for Embedded Control Systems User's Guide, 2000

[ETAS00b] ETAS GmbH & Co. KG: ERCOS^{EK} User's Guide, 2000

[FeWi99] C. Ferdinand, R. Wilhelm: Efficient and Precise Cache Behavior Prediction for Real-Time Systems, Real-Time Systems 17 (1999), Heft 2/3: S. 131-181

[Fisc98] M. Fischer: Entwicklung eines fehlertoleranten TTP-basierten Multi-Level-Schedulers für den 80C167, Diplomarbeit Nr. 1638, IAS, Universität Stuttgart, 1998

[Flex] FlexRay Homepage: http://www.flexray-group.com

[Föll78] O. Föllinger: Regelungstechnik Einführung in die Methoden und ihre Anwendung, 2. Auflage, Elitera-Verlag, Berlin, 1978

[FMD+01] Th. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther: Time Triggered Communication on CAN (Time Triggered CAN- TTCAN), CAN in Automation (CiA), Frankfurt, 2001

[GNU] Homepage des GNU Projekts: http://www.gnu.org

[GöFu98] J. Göthel, M. Fuchs: Semiformale Entwurfsmethoden für die Funktionsmodellierung am Beispiel der Fahrzeugklimatisierung, VDI Berichte 1415: S. 425 - 434, 1998

[GJSB00] J. Gosling, B. Joy, G. Steele, G. Bracha: The Java Language Specification, Addison-Wesley, Boston, MA, 2. Auflage, 2000

[Gunz99] M. Gunzert: Building Safety Critical Real-Time Systems with Synchronous Software Components, IFAC Workshop on Real-Time Programming WRTP'99, Schloß Dagstuhl, Germany, 1999

[GuRi99] M. Gunzert, Th. Ringler: ViPER – A Component-Based Approach for Designing Real-Time Systems, Proceedings ISA TEC INTERKAMA ’99, Düsseldorf, 1999

[Hans01] P. Hansen: BMW and Mercedes Choose FlexRay, The Hansen Report on Automotive Electronics 13 Heft 10, December 2000/January 2001

[Hare87] D. Harel: Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, Elsevier Science Publichers (North Holland): S. 231-274, 1987

[HMFH01] F. Hartwich, B. Müller, Th. Führer, R. Hugel: CAN Network with Time Triggered Communication, CAN in Automation (CiA), Frankfurt, 2001

[HRK+00] B. Hedenetz, J. Ruh, M. Kühlewein, A. Schedl, E. Fuchs, A. Krüger, Y. Domaratsky, P. Pelcat, S. Boutin, E. Dilger, Th. Führer, S. Poledna, M. Glück, Th. Ringler: OSEKtime – Die neue fünfte OSEK/VDX, Arbeitsgruppe: Ein zuverlässiges fehlertolerantes Echtzeit Betriebssystem und Kommunikationsschicht für By-Wire Applikationen, VDI Berichte 1547: S. 59, Elektronik im Kraftfahrzeug Tagung Baden-Baden, 2000

[Hede98] B. Hedenetz: Development Framework for Ultra-Dependable Automotive Systems Based on a Time-Triggered Architecture, 19th IEEE Systems Symposium (RTSS98), Madrid, Spanien, 1998

[Hein00] B. Heintel: Konzeption und Evaluierung der Anbindung von Zeitanalyseverfahren an Software-Entwicklungsverfahren für verteilte zeitgesteuerte Systeme, Diplomarbeit Nr. 1773, IAS, Universität Stuttgart, 2000

[Hero92] H. Herold: Lex und Yacc – lexikalische und syntaktische Analyse, Addison-Wesley, Bonn 1992

[Ilog] I-Logix Homepage: http://www.ilogix.com

[ISO99] ISOTC 22/SC 3 N 11898: Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication, ISO/WD11898-4, 1999

[Jams86] K. Jamsa: Bibliothek der C-Routinen, McGraw-Hill Book Company GmbH, Hamburg, 1986

[Kaus89] E. Kausen: Numerische Mathematik mit TURBO-PASCAL, Dr. Alfred Hüthig Verlag Heidelberg, 1989

[KeRi88] B. W. Kernighan, D. M. Ritchie: The C Programming Language, Prentice-Hall, Englewood Cliffs, NJ, USA, 2. Auflage, 1988

[Kers99] Th. Kersten: Entwicklung eines Zeitanalyseverfahrens für synchrone TTP-Applikationssoftware, Diplomarbeit Nr. 1726, IAS, Universität Stuttgart, 1999

[KoGr94] H. Kopetz, G. Grünsteidl: TTP - A Protocol for Fault-Tolerant Real-Time Systems, IEEE Computer 27 (1994) Heft 1: S. 14-23, 1994

[KoGr92] H. Kopetz, G. Grünsteidl: TTP – A time-triggered protocol for fault-tolerant real-time systems, Research Report 12/92, Institut für Technische Informatik, TU Wien, Österreich, 1992

[KoNo95] H. Kopetz, R. Nossal: The Cluster Compiler – A Tool for the Design of Time-Triggered Real-Time Systems, ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, Vol. 30, La Jolla, California, 1995

[Kope97a] H. Kopetz: Real-Time Systems - Design Principles for Distributed Real-Time Systems, Kluwers Academic Publishers, 1997

[Kope97b] H. Kopetz, et. al.: A Prototype Implementation of a TTP/C Controller, Proceedings SAE World Congress 1997, Detroit, MI, USA, Society of Automotive Engineers, SAE Press, SAE Paper No. 970296, Feb. 1997

[KFMN99] H. Kopetz, E. Fuchs, D. Millinger, R. Nossal: An Interface as a Design Object, ISORC ’99, 1999

[KoPl96] R. Koschke, E. Plödereder (1996): Ansätze des Programmverstehens, Softwarewartung und Reengineering - Erfahrungen und Entwicklungen: S. 159-176, Deutscher Universitätsverlag / Gabler Vieweg Westdeutscher Verlag, 1996

[KWB98] H. Krimmel, A. Welte, O. Buchhold: Übergang von der C-Programmierung zur graphischen problemorientierten Beschreibung toolunabhängige Methoden und Strukturierungsansätze, VDI Berichte 1415: S. 735 – 756, 1998

[KDH+00] A. Krüger, Y. Domaratsky, B. Holzmann, A. Schedl, C. Ebner, R. Belschner, B. Hedenetz, E. Fuchs, A. Zahir, S. Boutin, E. Dilger, Th. Führer, R. Nossal, B. Pfaffeneder, S. Poledna, M. Glück, C. Tanzer, Th. Ringler: OSEKtime: Highly Dependable Applications - Objectives, Basics and Concepts, VDI Berichte 1528: S. 51-70, OSEK/VDX Open Systems in Automotive Networks 3rd International Workshop, Bad Homburg, 2000

[LaGö99] R. Lauber, P. Göhner: Prozessautomatisierung II, 1. Auflage, Springer-Verlag, 1999

[LiMa95] Y. Li, S. Malik: Performance Analysis of Embedded Software Using Implicit Path Enumeration, ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems Vol. 30, La Jolla, Carlifornia, 1995

[LiMa99] Y. Li, S. Malik: Performance Analysis of Real-Time Embedded Software, Kluwer Academic Publishers, Bosten, Dordrecht, London, 1999

[LIN99] Motorola Semiconductors: LIN Specification, Revision 0.7.1b, 1999

[Maga98] J. Maga: Aufbau und Inbetriebnahme einer auf dem 80C167 und TTP basierenden elektronischen Lenkung für den Modellprozess IAS-Kart, Studienarbeit Nr. 1876, IAS, Universität Stuttgart, 1999

[Math] MathWorks Homepage: http://www.mathworks.com/

[Math98a] The MathWorks: Using MATLAB, The MathWorks, Nattick, MA USA, 1998

[Math98b] The MathWorks: Simulink User Guide, The MathWorks, Nattick, MA USA, 1998

[Math98c] The MathWorks: Fixed-Point Blockset User Guide, The MathWorks, Nattick, MA USA, 1998

[Math98d] The MathWorks: StateFlow User Guide, The MathWorks, Nattick, MA USA, 1998

[Math98e] The MathWorks: Real-Time Workshop User Guide, The MathWorks, Nattick, MA USA, 1998

[Micr] Microsoft Homepage: http://www.microsoft.de

[Mok89] A. K. Mok et. al: Evaluating Tight Execution Time Bounds of Programs by Annotations, Proceedings of the 6th IEEE Workshop on Real-Time Operating Systems and Software: S. 74-80, 1989

[MOST] MOST Homepage: http://www.mostnet.de/main/index.htlm

[NoGa97] R. Nossal, T. M. Galla: Solving NP-Complete Problems in Real-Time System Design by Multichromosome Genetic Algorithms, ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems, 1997

[Noss97] R. Nossal: An Interface-Focused Methodology for the Development of Time-Triggered Real-Time Systems Considering Organizational Constraints, Dissertation, Institut für Technische Informatik, TU Wien, Österreich, 1997

[OSEK00a] OSEK/VDX: Time-Triggered Operating System Specification, Version 0.2.19, 2000

[OSEK00b] OSEK/VDX: System Generation OIL: OSEK Implementation Language, Version 2.2, 2000

[Park92] C. Y. Park: Predicting Deterministic Execution Times of Real-Time Programs, Ph. D. Thesis, University of Washington, Seattle 98195, 1992

[PBG99] M. Peller, J. Berwanger, R. Grießbach: byteflight specification, BMW AG, Version 0.5, 1999

[Plöd00] E. Plödereder: Lecture Notes Real-Time Programming, WS 2000/2001, Universität Stuttgart, Abteilung Programmiersprachen und Übersetzerbau, 2000

[PGT+00] S. Poledna, M. Glück, C. Tanzer, S. Boutin, E. Dilger, Th. Führer C. Ebner, E. Fuchs, R. Belschner, B. Hedenetz, B. Holzmann, A. Schedl, R. Nossal, B. Pfaffeneder, Th. Ringler, Y. Domaratsky, A. Krüger, A. Zahir: OSEKtime: A Dependable Real-Time Fault-Tolerant Operating System and Communication Layer as an Enabling Technology for By-Wire Applications, SAE Word Congress 2000, Detroit Michigan USA, 2000

[PuKo89] P. Puschner, C. Koza: Calculating the Maximum Execution Time of Real-Time Programs, Journal of Real-Time Systems 1 (1989), Heft 2: S. 159-176

[Pusc93] P. Puschner: Zeitanalyse von Echtzeitprogrammen, Doktorarbeit, Institut für Technische Informatik, TU Wien, Österreich, 1993

[PuSc97] P. Puschner, A. Schedl: Computing Maximum Task Execution Times – a Graph-Based Approach, Real-Time Systems 13 (1997) Heft 1: S. 67-91

[PuVr96] P. Puschner and A. Vrchoticky: Problems in Static Worst-Case Execution Time Analysis, Research Report No. 6/96, Institut für Technische Informatik, TU Wien, Österreich, 1996

[Ring00a] Th. Ringler: Entwicklung und Analyse von verteilten zeitgesteuerten Systemen, 3. ITG/GI/GMM-Workshop, Forschungs-Report Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Frankfurt, VDE Verlag Berlin, Offenbach: S. 261 - 270, 2000

[Ring00b] Th. Ringler: Static Worst-Case Execution Time (WCET) Analysis of Synchronous Programs, 5th International Conference on Reliable Software Technologies, Ada-Europe 2000, Potsdam, Lecture Notes in Computer Science 1845: S. 56-68, Springer Verlag, 2000

[RSBH98] Th. Ringler, J. Steiner, R. Belschner and B. Hedenetz: Increasing System Safety for by-wire Applications in Vehicles by using a Time Triggered Architecture, Proceedings 17th International Conference, SAFECOMP'98, Heidelberg, Lecture Notes in Computer Science 1516: S. 243 - 253, Springer Verlag, 1998

[Ruh00] J. Ruh: Konzeption eines Entwurfsprozesses für das zeitliche Design verteilter zeitgesteuerter Systeme, Diplomarbeit Nr. 1761, IAS, Universität Stuttgart, 2000

[SHS+97] C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz and C. Temple: Time-Triggered Architecture (TTA), EMMSEC'97, Florenz, Italien, 1997

[SFB98] C. Scheidler, E. Fuchs and A. Bäcker: SETTA: A Systems Engineering Environment for Time-Triggered Architectures, EMMSEC'98, Bordeaux, Frankreich, 1998

[Schn91] H.-J. Schneider (Hrsg.): Lexikon der Informatik und Datenverarbeitung, 3. Auflage, Oldenbourg, 1991

[Shaw89] Alan C. Shaw: Resoning about time in higher-level language software, IEEE Transactions on Software Engineering 15 (1989) Heft 7: S. 875-889

[Siem96] Siemens AG: C167 Derivatives User's Manual, Siemens AG, München, 1996

[Siem97] Siemens AG: C166 Family Instruction Set, Version 1.2, 1997

[Task98] Tasking Inc.: C166/ST10 C Compiler Manual, Tasking, 1998

[TTA] TTA Projekt Homepage: http://www.vmars.tuwien.ac.at/projects/tta/index.html

[TTTe] TTTech Computertechnik GmbH Homepage: http://www.ttech.com

[TTTe99a] TTTech Computertechnik GmbH: TTPplan/ TTPbuilt Dokumentation, Wien, Österreich, 1999

[TTTe99b] TTTech Computertechnik: TTP/C Protocol – Specification of the TTP/C Protocol, Version 0.5, Wien, Österreich, 1999

[Vect] Vector-Informatik Homepage: http://www.vector-informatik.de

[WiKr98] M. Winterkorn, W. Kreft: Plattformstrategie in der Fahrzeug-Elektrik/ -Elektronik, VDI Berichte 1415: S. 3 - 20, Elektronik im Kraftfahrzeug Tagung Baden-Baden, 1998

[Wind] WindRiver Homepage: http://www.windriver.com

Lebenslauf

Persönliche Daten:

01.10.1970	geboren in Abtsgmünd

Schulbildung:

Juni 1987	Mittlere Reife, Realschule Abtsgmünd
Mai 1990	Fachgebundene Hochschulreife, Technisches Gymnasium Aalen

Wehrdienst:

Juli 1990 - Juni 1991	Grundwehrdienst im 4./Panzergrenadierbatallion 302, Ellwangen

Studium:

Okt. 1991 - Juni 1997	Studium der Elektrotechnik an der Universität Stuttgart Studienrichtung Regelungstechnik und Prozessautomatisierung achtmonatiger Studienaufenthalt in Porto Alegre, Brasilien Fachpraktikum bei der Firma Telenot, Aalen Fachpraktikum bei der Daimler-Benz Forschung, Esslingen
01.07.1997	Abschluss des Studiums als Diplom-Ingenieur

Berufstätigkeit:

Sept. 1997 - Juli 2001	Wissenschaftlicher Mitarbeiter am Institut für Automatisierungs- und Softwaretechnik der Universität Stuttgart
Seit Okt. 2001	Mitarbeiter bei der DaimlerChrysler Forschung, Esslingen