Alexander Burst

Rapid Prototyping eingebetteter elektronischer Systeme auf Basis des CDIF-Datenaustauschformats

Institut für Technik der Informationsverarbeitung
Universität Karlsruhe (TH)

Rapid Prototyping eingebetteter elektronischer Systeme auf Basis des CDIF-Datenaustauschformats

Zur Erlangung des akademischen Grades eines

DOKTOR–INGENIEURS

von der Fakultät Elektrotechnik und Informationstechnik
der Universität Fridericiana Karlsruhe
genehmigte

DISSERTATION

von
Dipl.-Ing. Alexander Burst
aus Karlsruhe

Tag der mündlichen Prüfung : 15. Februar 2000
Hauptreferent : Prof. Dr.-Ing. Klaus D. Müller-Glaser
Korreferent : Prof. Dr. Dr. h.c. mult. Manfred Glesner

Kurzfassung

Die vorliegende Dissertation befaßt sich mit der schnellen Erzeugung von Prototypen (engl. *Rapid Prototyping*) für elektronische Systeme. Durch die schnell wachsende Komplexität der Entwurfsaufgaben und den damit verbundenen hohen Anteil an schwer nachvollziehbaren Wechselwirkungen von Systemkomponenten finden verstärkt formale Hilfsmittel zur Spezifikation und abstrakten Modellierung Einzug in den Entwurfsprozeß. Diese Techniken erlauben den Einsatz von rechnergestützter Analyse, Simulation und Codeerzeugung zur Validierung und Verifikation.

Als Ausgangspunkt der Arbeit dienten Ansätze bestehender CASE-Werkzeuge, die aus einer Modellierung automatisch ablauffähigen Code erzeugen können. Diese Ansätze weisen jedoch Schwächen bei der Verwendung mehrerer Modellierungswerkzeuge aus unterschiedlichen Bereichen (z.B. diskrete Zustandsautomaten und kontinuierliche Blockdiagramme) sowie bei der Erzeugung von Code für harte Echtzeitbedingungen auf. In dieser Arbeit wurde deswegen eine Überführung der proprietären Modelldaten in das standardisierte CASE Datenaustauschformat CDIF vorgenommen. Die zugrunde liegende Idee besteht in der Verwendung dieses Datenformats als einheitliche Basis zur integrierten, heterogenen Analyse, Simulation und Codeerzeugung unter Verwendung mehrerer Modellierungsarten. Auf diese Weise wird eine weitestgehende Unabhängigkeit von Modellierungswerkzeugen erreicht. Die Einbindung neuer Modellierungswerkzeuge erfordert bei diesem Ansatz lediglich die Unterstützung von CDIF, während die Komponenten für Analyse, Simulation und Codeerzeugung unverändert bleiben.

Wesentlicher Bestandteil der Arbeit war auch die Entwicklung eines optimierten Schedulingalgorithmus für den erzeugten Code. Aus dem formalen Prozeßmodell wurde eine Klassifikation des Schedulings abgeleitet und eine geeignete Metrik für die Randbedingungen des Rapid Prototyping ausgewählt. Das aus der Literatur bekannte zeitgesteuerte, ratenmonotone Scheduling wurde zu pseudoraten-basierten Scheduling unter Optimierung der vorhandenen Systemressourcen verfeinert. Durch die Erweiterung des Zeitmodells konnte die Überlegenheit des pseudoraten-basierten Schedulings rechnerisch und im Experiment anhand einer industriellen Fensterheberansteuerung aus dem Automobilbereich nachgewiesen werden.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Technik der Informationsverarbeitung (ITIV) der Universität Karlsruhe (TH). Ich möchte mich bei den beiden Leitern des Instituts, Herrn Prof. Dr.–Ing. Klaus D. Müller–Glaser und Herrn Prof. Dr.–Ing. Dr.–Ing.h.c. Dr.h.c. H.M. Lipp, für die Möglichkeit bedanken, an ihrem Institut zu arbeiten.

Mein besonderer Dank gilt Herrn Prof. Dr.–Ing. Klaus D. Müller–Glaser, der diese Arbeit ermöglichte und dessen Meinung und Rat mir immer eine wertvolle Hilfe war. Durch beständiges Interesse, fachliche Diskussionen, Anregungen und sein Vertrauen hat er wesentlich zum Gelingen der Arbeit beigetragen.

Mein Dank gilt in gleicher Weise Herrn Prof. Dr.–Ing. Manfred Glesner für die Übernahme des Korreferats meiner Arbeit und die damit einhergehende Unterstützung.

Ich möchte mich bei allen Kollegen und Mitarbeitern am Institut bedanken, die durch interessante Diskussionen meine Arbeit unterstützten. Auch das freundschaftliche und angenehme Umfeld am ITIV trug seinen Anteil bei. An dieser Stelle möchte ich mich namentlich bei meinen Kollegen Peter Elter, Johannes Ernst, Markus Kühl, Gunther Lehmann, Bernhard Spitzer, Michael Wolff und Bernhard Wunder bedanken. Insbesondere die Entlastung von den Institutsarbeiten in der Endphase dieser Arbeit half mir sehr. Wichtig für das Gelingen dieser Arbeit waren auch Studenten, die im Rahmen ihrer Studien- und Diplomarbeit im Themengebiet Rapid Prototyping arbeiteten. Hierbei möchte ich besonders Georg Doll, Jürgen Kock, Wolfgang Meier, Kilian Schnellbacher und Matthias Volz erwähnen. Für die Durchführung der aufwendigen Meßreihen danke ich Thomas Schmerler. Für die sorgfältig angefertigten Zeichnungen möchte ich mich bei Frau Gerda Neidhard bedanken.

Mein herzlicher Dank gilt meiner Frau Tina sowie meinen Eltern Bärbel und Heinz, die durch ihre liebevolle Unterstützung einen wesentlichen Beitrag leisteten und mir die notwendigen Freiräume zur Durchführung dieser Arbeit schufen.

Wenn man sehr jung ist und wenig weiß,
sind Berge Berge, Wasser ist Wasser und Bäume sind Bäume.

Hat man studiert und ist aufgeklärt,
sind Berge nicht mehr Berge, Wasser ist nicht mehr Wasser und
Bäume sind nicht länger Bäume.

Hat man wirkliches Verständnis gewonnen,
sind Berge wieder Berge, Wasser ist Wasser und Bäume sind Bäume.

Zen–Weisheit

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Elektronische Systeme bestehen aus anwendungsspezifischen, integrierten Schaltungen oder programmierbaren Standardbausteinen mit Speichern oder Zusatzhardware (Bild 1-1). Im Gegensatz zur klassischen Datenverarbeitung zeichnen sich diese Systeme durch eine intensive Interaktion mit der Umgebung und fest definierten Aufgaben mit zeitlichen Randbedingungen aus. Speziell der Bereich der Mikroelektronik erfordert immer komplexere Funktionalität, die in einem wachsenden in-

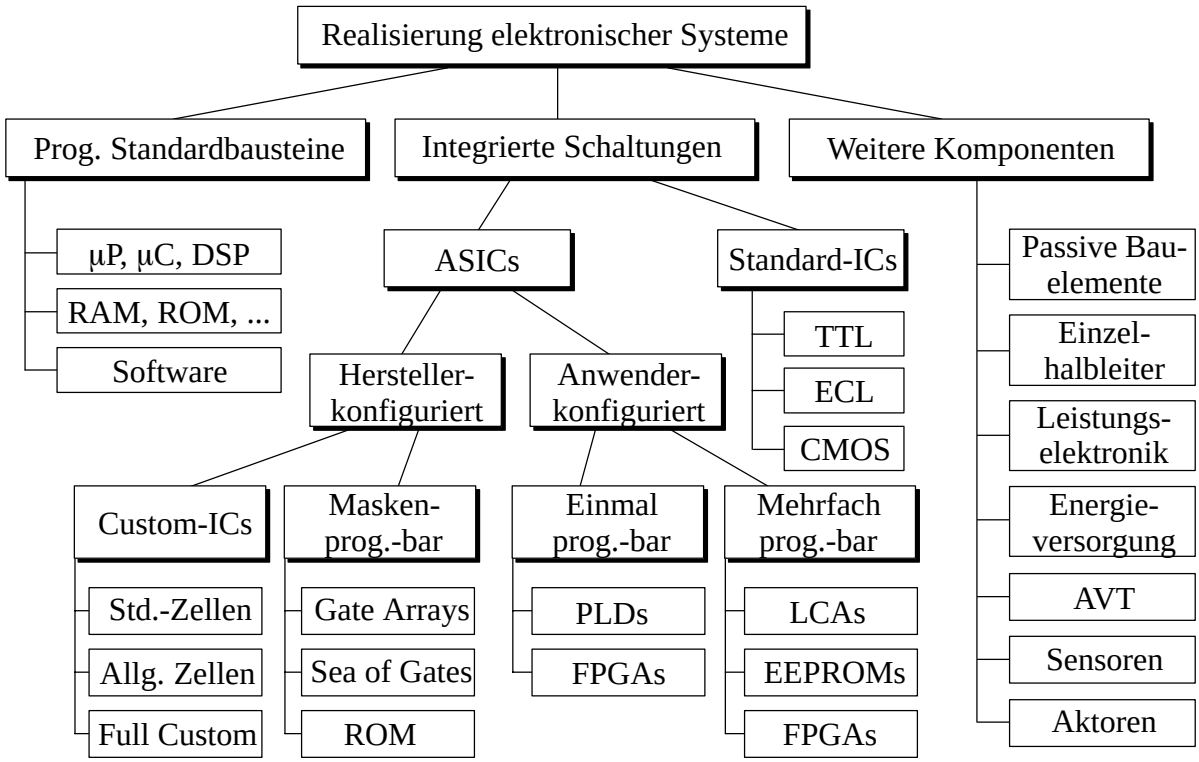


Bild 1-1: Entwurfsalternativen für elektronische Systeme

ternationalen Wettbewerb mit strengen Richtlinien zu entscheidenden Wettbewerbsvorteilen führen können. Die Einhaltung der Spezifikationsvorgaben und die Erfüllung gesetzlicher Vorschriften sind insbesondere für elektronische Systeme mit “harten” Echtzeitbedingungen (engl. *hard real-time systems*) wichtig. Bei der Verletzung der Echtzeitbedingungen können schwere Störungen auftreten, die von der Zerstörung eines Werkstücks bis zur Gefährdung von Mensch und Umwelt führen können. Gerade sicherheitskritische Systeme wie beispielsweise im Automobilbereich ABS-Bremssysteme oder Gear/Brake/Steer-By-Wire Systeme müssen hohe Anforderungen bei der Entwicklung berücksichtigen. Für diese Arbeit wurde das Umfeld der Automobilindustrie ausgewählt,

da der Bedarf an modernen Entwicklungsmethoden wegen des starken Kostendrucks sehr hoch ist und für diesen Bereich bereits eine detaillierte Klassifikation der möglichen Einsatzgebiete [ChEr93] erstellt worden war. In einem Oberklassen-Automobil kommen heute schon mehr als 70 Mikrocomputersysteme zum Einsatz. Bild 1-2 zeigt einen Teil dieser Steuergeräte in einem modernen Fahrzeug. Entsprechend wird der Anteil der Elektronik und Software auf über 30% in der Wertschöpfungskette prognostiziert. Durch die zunehmende Komplexität und die Zahl der Aufgaben eines elektronischen Systems werden jedoch auch die Entwurfsaufgaben zunehmend anspruchsvoller. Zusätzlich wird eine sehr geringe Zeit vom Beginn der Entwicklung bis zur Marktreife (engl. *time-to-market*) gefordert, die ohne aufwendige nachträgliche Korrekturen am Endprodukt in ihrer Einsatzumgebung auskommen müssen.

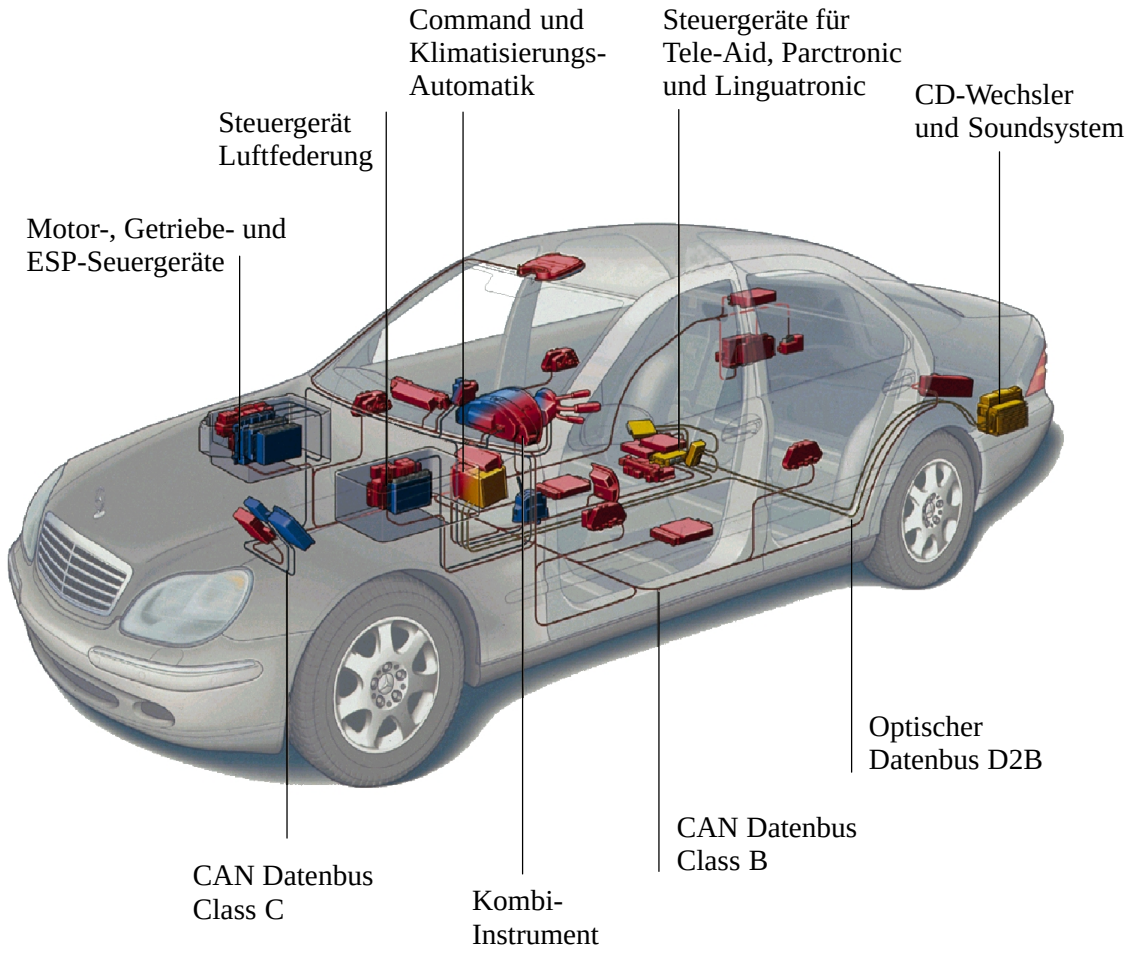


Bild 1-2: Vernetzte Steuergeräte in der Automobiltechnik

Die schnell wachsende Komplexität der Entwurfsaufgaben ist durch den hohen Anteil an Wechselwirkungen für den Menschen nur noch schwer nachzuvollziehen. Auch die Qualitätssicherung gestaltet sich als zunehmendes Problem. Um Entwurfsfehler in frühen Phasen zu vermeiden, werden verstärkt formale Hilfsmittel zur Spezifikation und zur abstrakten Modellierung verwendet. Mit den formalen Hilfsmitteln finden auch die Simulation und formale Verifikation der Modellierung Einzug in den Entwurf elektronischer Systeme. Diese Techniken sind in vielen Jahren gewachsen.

Deswegen existieren mehrere parallele Beschreibungsformen, mit denen dieselbe Funktionalität modelliert werden kann. Zusätzlich sind die Beschreibungsformen sowie Simulation und formale Verifikation oftmals auf einzelne Werkzeuge festgelegt, die keine oder eine eingeschränkte Interaktion mit weiteren Werkzeugen erlauben.

Simulation und formale Verifikation besitzen im Entwurfsverlauf Schwächen und Einschränkungen, die beispielsweise in der Notwendigkeit der Modellierung der Umgebung eines elektronischen Systems und der eingeschränkten zeitlichen Analyse im Zusammenspiel mit Echtzeitbetriebssystemen sichtbar werden. Als Ergänzung wird bereits seit vielen Jahren die Fertigung von Prototypen in den Entwurfsablauf miteinbezogen. Der Prototyp kann direkt in die endgültige Systemumgebung eingebettet werden und erlaubt eine “Erfahrbarkeit” des modellierten Systems. Dies ist insbesondere dann wichtig, wenn elektronische Systeme nach dem subjektiven Empfinden des Menschen optimiert werden sollen, beispielsweise bei der Abstimmung einer automatischen Gangschaltung.

Der Aufwand zur Erstellung eines Prototypen ist jedoch meist sehr hoch, da einerseits eine komplette Implementierung des elektronischen Systems vorgenommen werden muß und andererseits die Nachbildung der physikalischen Schnittstellen erforderlich ist. Durch den hohen Implementierungsaufwand besteht die Möglichkeit, daß Inkonsistenzen zwischen Spezifikation und Prototyp auftreten. Prototypen werden daher meist erst sehr spät im Entwurfsablauf und in einem eingeschränkten Umfang verwendet. Diese späte Einbindung bringt jedoch ein erhöhtes Entwurfsrisiko mit sich.

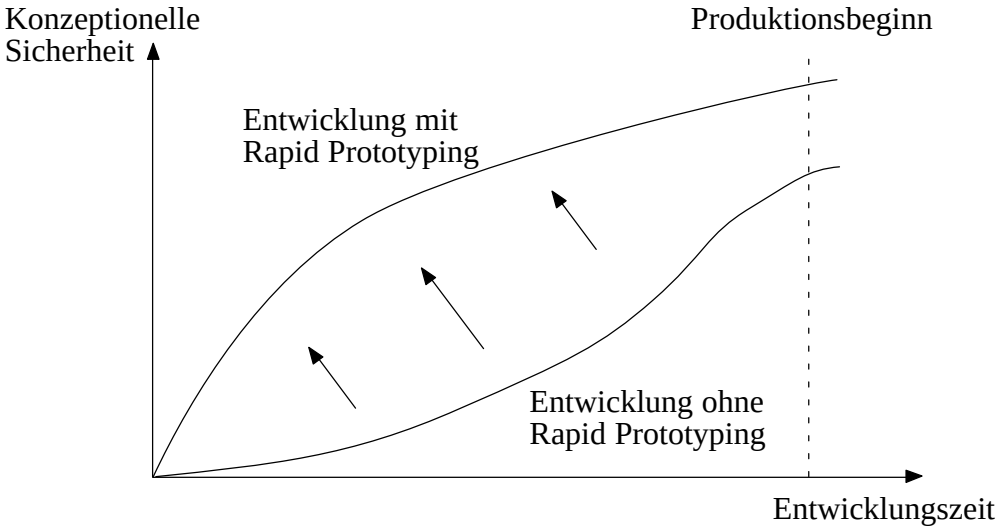


Bild 1-3: Erhöhung der Planungssicherheit durch Rapid Prototyping

Die nachfolgend vorgestellte Methode des schnellen Prototypenentwurfs (engl. *rapid prototyping*) verspricht den notwendigen Schritt zu einem Einsatz von Prototypen bereits in frühen Phasen, die damit die konzeptionelle Sicherheit (Bild 1-3) entscheidend erhöhen können. Ein Rapid Prototyping System setzt eine abstrakte Modellierung weitgehend automatisch in ein Hardware-/Software-System um, das die Funktionalität der Modellierung nachbildet. Der Aufbau und Kosten dieses Systems sind dabei von untergeordneter Bedeutung. Bild 1-4 zeigt die Komponenten, aus denen ein heterogenes elektronisches System aufgebaut ist und den Bereich, der von einem Rapid Prototyping System umfaßt wird.

Modellierungstechniken für Rapid Prototyping umfassen u.a. die Bereiche Steuerungs– und Regelungstechnik, objekt-orientierte Analyse und Design, Datenmodellierung und -ablage (für Datenbanken) und Datenflußmodellierung. Bestehende Rapid Prototyping Systeme sind jedoch nicht in der Lage, “Best–of–Point” Werkzeuge unterschiedlicher Modellierungstechniken kombiniert einzusetzen. Jedes der kommerziellen Werkzeuge behandelt zwar auch Modellierungen von Werkzeugen anderer Modellierungstechniken, meist jedoch ohne Unterstützung des vollen Funktionsumfangs. Unterschiede zwischen den Werkzeugen bestehen auch bei der Codeerzeugung. Manche Werkzeuge können keinen Code erzeugen, andere nur für die Beschleunigung von Simulationsabläufen und nur die wenigsten unterstützen die für Rapid Prototyping wichtige Erzeugung von Echtzeitcode. An dieser Situation wird sich selbst langfristig nichts ändern, da Anwender unterschiedliche Anforderungen an Werkzeuge stellen. Oftmals sind Anwender auch aus historischen Gründen auf spezielle Werkzeuge festgelegt, vorhandenes Know–how der Mitarbeiter soll weiterhin genutzt werden oder bereits existierende, umfangreiche Modellbibliotheken würden einen hohen Konvertierungsaufwand mit sich bringen.

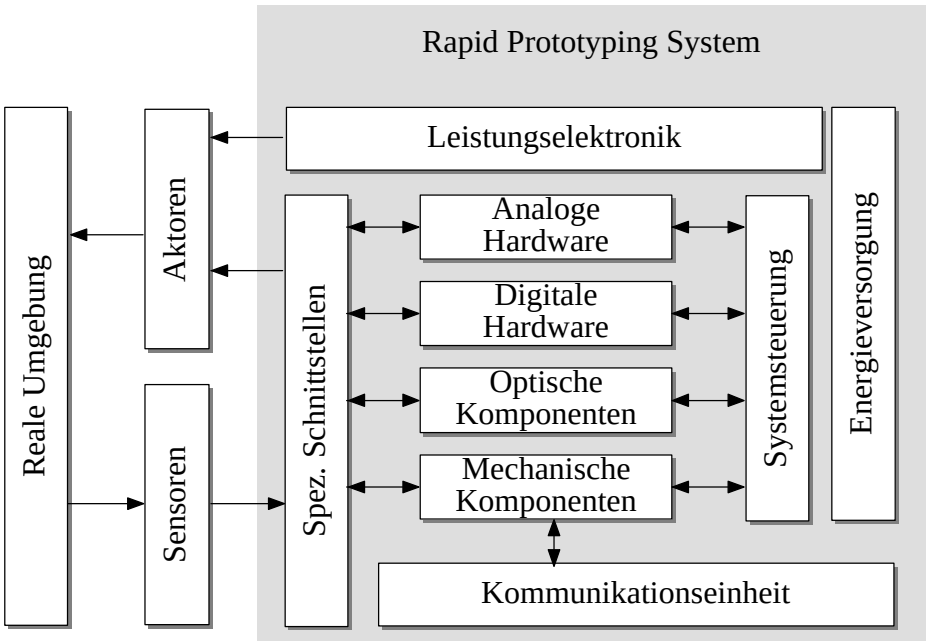


Bild 1-4: Komponenten eines heterogenen Systems

Wird Rapid Prototyping eingesetzt, folgt oftmals der Wunsch nach automatischer Erzeugung eines möglichst effizienten, seriennahen Echtzeit–Codes. Die Echtzeit–Codeerzeugung heutiger Werkzeuge benötigt im Vergleich zu handoptimierten Echtzeit–Code meist mehr Speicherplatz und längere Rechenzeiten. Dies schränkt die Verwendung in manchen Bereichen (z.B. Motorsteuerung) stark ein. Die Benutzung unterschiedlicher Werkzeuge und die daraus resultierende Verwendung von mehreren Codegeneratoren fällt ebenfalls nicht optimal aus. Die Codestücke müssen meist per Hand für jeden Rapid Prototyping Durchlauf erneut angepaßt werden. Zur Erleichterung wird bei der heutigen Codeerzeugung mit “Templates” gearbeitet, die eine Vorlage für den Aufbau des zu erzeugenden Codes darstellen und somit eine gezielte Beeinflussung erlauben. Die Codeerzeugung

für ein gesamtes Projekt ist jedoch auch damit nicht möglich. Für Steuergeräte im Automobilbau gilt darüber hinaus, daß der Echtzeit–Code für das automobilspezifische Echtzeitbetriebssystem OSEK (*Open Systems and the Corresponding Interfaces for Automotive Electronics*) [OSEK97] generiert werden muß.

Abhängig vom Werkzeug ist auch die Palette von einsetzbaren, käuflichen Hardwaremodulen. Jeder Hersteller besitzt eigene, auf das jeweilige Werkzeug abgestimmte Hardwaremodule, die wegen der Verwendung eigener Bussysteme oder wegen nicht offenliegenden Schnittstellen nicht allgemein angepaßt werden können. Somit ist die Wiederverwendung von Modellen oder Modellbibliotheken, der Datenaustausch zwischen Werkzeugen und die beliebige Verwendung von Hardware nur eingeschränkt möglich. Auch im Bereich der Simulation spielt dieses Problem eine wesentliche Rolle. Der Wunsch, eine große Anzahl von Steuergeräten gemeinsam zu simulieren, die mit unterschiedlichen Werkzeugen modelliert wurden, führte zu zahlreichen proprietären Simulatorkopplungen. Es existiert daher ein dringender Wunsch nach standardisierten Datenformaten, die eine gemeinsame Nutzung von Analysewerkzeugen, Simulatoren und Codegeneratoren ermöglichen können.

1.2 Eigener Beitrag

Bei der Entwicklung eines Rapid Prototyping Systems für frühe Phasen muß eine nahtlose Einbindung in den Entwurfsprozeß ermöglicht werden, der durchgängig von der abstrakten Modellierung unter Beachtung von Analyse und Simulation bis zur Erzeugung von leistungsfähigem, möglichst kompakten Echtzeit-Code reicht. In dieser Arbeit werden deswegen mehrere Bereiche betrachtet und verbessert, die zur Erreichung dieses Ziels notwendig sind. Im einzelnen wurden die folgenden Gebiete behandelt:

- Für die Durchführung der Modellierung wird eine heterogene Entwurfsumgebung unter Berücksichtigung der horizontalen und vertikalen Kooperation (siehe Kapitel 5) sowie der strukturellen Verbindung von Modellen auf unterschiedlichen Abstraktionsebenen konzeptioniert und aufgebaut. Die Komponenten des Systementwurfs, d.h. Modellierung, Analyse, Simulation, Verifikation und Codeerzeugung, können eng in die Entwurfsumgebung integriert werden. Durch die Bibliotheksbildung und Implementierungsunabhängigkeit im Bereich der Modellierung wird eine Wiederverwendbarkeit bereits erstellter Modelle möglich. Für die Modellierung werden vorhandene CASE-Werkzeuge (Computer-Aided Software/Systems Engineering) aus unterschiedlichen Entwurfsbereichen in eine einheitliche Entwurfsumgebung eingebunden. Die Struktur des Gesamtsystems wird durch Funktionsblöcke festgelegt, die entweder eine strukturelle Verfeinerung oder eine Verhaltensbeschreibung enthalten können. In der Arbeit wird detailliert der Aufbau der benötigten Klassen zur Schnittstellendefinition, Implementierung, Verbindung und Konfiguration beschrieben.
- Durch den gewählten Aufbau eignet sich dieser Ansatz in idealer Weise für den Systementwurf und wird durch Verwendung eines standardisierten CASE Datenaustauschformats offen für die Anbindung weiterer Werkzeuge im Modellierungsbereich oder im weiteren Verlauf des Entwurfs (Analyse, Simulation, Codeerzeugung, etc.). Als Datenformat wird in der Arbeit das Datenaustauschformat CDIF (CASE Data Interchange Format) verwendet, das eine Vier-Ebenen Architektur zur Beschreibung von Modellierungselementen besitzt. Wegen dieser Architektur kann der Funktionsumfang des Datenformats erweitert werden, ohne den verwendeten Stan-

dard aufgeben zu müssen und ist somit offen für zukünftige Modellierungstechniken. Gerade im dynamischen Umfeld der Modellierung elektronischer Systeme ist diese Eigenschaft von großem Vorteil. Zum ersten Mal wurde CDIF in dieser Arbeit nicht nur zum Daten*austausch* von CASE-Werkzeugen, sondern als einheitliches Datenformat für den Systementwurf eingesetzt.

- Da heutige CASE-Werkzeuge noch keine Unterstützung für dieses Datenaustauschformat anbieten, wurden Compiler für die Umwandlung der Modellierungsdaten erstellt, die im proprietären Datenformat des Werkzeugs vorlagen. Um Werkzeuge aus verschiedenen Entwurfsbereichen zu verwenden, wurden Compiler für die CASE-Werkzeuge STATEMATE™ und MATRIX$_X$™ erstellt. Dabei wurde gezeigt, daß sämtliche diskreten und kontinuierlichen Modellierungselemente erfolgreich in CDIF überführt werden konnten. Um die erstellten CDIF-Beschreibungen miteinander zu verbinden, wurde ein weiterer Compiler erstellt, der beliebige CDIF-Beschreibungen miteinander koppeln kann.

- Aus dem Gesamtsystem, das im CDIF-Datenformat vorliegt, wird hochoptimierter Echtzeit-Code erzeugt. Der Echtzeit-Code wurde in drei Teile aufgeteilt, um einen möglichst modularen Aufbau zu erreichen. Der Modellcode (1) repräsentiert die Verhaltensbeschreibung des zu entwickelnden Systems. Für den diskreten Bereich werden u.a. Systemzustände und Zustandsübergänge, für den kontinuierlichen Bereich u.a. algebraische Ausdrücke und Differentialgleichungen in Programmcode nachgebildet. Bei der Ausführung des Echtzeitcodes werden hierarchische und parallele Strukturen im diskreten Bereich genutzt, um die Abarbeitung zu beschleunigen. Sämtliche Modellierungskomponenten werden in der Arbeit detailliert im CDIF-Datenformat und als Echtzeitcode dargestellt. Der Schedulingcode (2) ist für die Ablaufsteuerung und Überwachung des Software-Prototypen verantwortlich. Da diese Komponente mit jedem Modellschritt abgearbeitet wird, ist der optimierte Aufbau von zentraler Bedeutung für die Abarbeitungsgeschwindigkeit des Gesamtsystems. Der Ein-/Ausgabe-Code (3) dient der Einbindung von Ein-/Ausgabe-Hardwarekomponenten in das Rapid Prototyping System. Für jede Hardwarekomponente besitzt der Programmcode eine standardisierte Schnittstelle in Form von Funktionsaufrufen für Initialisierung, Ansteuerung der Schnittstellen und Beendigung der Bearbeitung. Die durch die Trennung in drei Teile gewonnene Unabhängigkeit der Einzelteile führt zu einer verbesserten und vereinfachten Systemübersicht und -wartung.

- Zusätzlich wird eine genaue Klassifikation des Schedulings unter Berücksichtigung der speziellen Anforderungen von Rapid Prototyping vorgenommen. Es werden unterschiedliche Schedulingstrategien vorgestellt, wobei das zeitgesteuerte Scheduling dem ereignisgesteuerten Scheduling vorgezogen wird. Das für das zeitgesteuerte Scheduling verwendete inkrementelle Fortschreiten der Zeit ist für die Abarbeitung der Differenzengleichungen im kontinuierlichen Bereich erforderlich. Es wird gezeigt, daß Vorteile bei der Abarbeitung erzielt werden können, wenn das Zeitinkrement äquidistant ist. Dies führt auf die Verwendung von ratenmonotonem Scheduling, das formal eingeführt wird. Nachteilig bei der Verwendung von ratenmonotonem Scheduling ist die Modellschrittweite, die für jeden Prozeß unterschiedlich ausfallen kann. Dies bedeutet, daß für die Überwachung der Echtzeitbedingungen jeweils ein eigener Prozeß für jeden vorhandenen Prozeß erstellt werden muß. Der somit vorhandene zusätzliche Aufwand kann für große Modelle einen erheblichen Zeitverlust bei der Abarbeitung bedeuten. Um eine optimale Systemleistung zu erreichen, wird das ratenmonotone Scheduling zu pseudoraten-basiertem Scheduling verfeinert, das nur eine einheitliche Zeitbasis für alle Prozesse verwendet und

somit auch nur einen zusätzlichen Prozeß zur Überwachung der Echtzeitbedingungen erforderlich macht. Beide Schedulingarten werden formal untersucht und um ein verfeinertes Zeitmodell mit Zeiten für Interrupts, Watchdog-Timern oder Auxiliary-Timern (siehe Kapitel 7) erweitert. Auf diese Weise konnte formal nachgewiesen werden, daß pseudoraten-basiertes Scheduling für die Abarbeitung von Echtzeitcode in einem Rapid Prototyping System besser geeignet ist. Die Auswirkungen, die die Verwendung dieser einheitlichen Zeitbasis mit sich bringt, wird ebenfalls diskutiert.

- Für den Test des Echtzeitcodes werden Vergleiche mit der Codeerzeugung bestehender CASE-Werkzeuge durchgeführt. Zusätzlich wird das Rapid Prototyping System unter industriellen Randbedingungen am Beispiel eines Fensterhebers mit Einklemm- und Diebstahlschutz getestet.
- Dazu wird die Entwicklung von softwarekonfigurierbarer Hardware auf Basis des VMEbus durchgeführt. Je nach Anwendung müssen zur Prozeßkopplung individuelle Schnittstellenschaltungen meist in Form von Leiterplatten entworfen werden. Dieses zeitraubende und fehlerträchtige Vorgehen kann den Rapid Prototyping Prozeß jedoch stark verlangsamen. Um dies zu vermeiden, werden software-konfigurierbare Hardwarekomponenten entwickelt, die eine möglichst allgemeine Anbindung an unterschiedliche Applikationen gewährleisten.

Insgesamt stellt der Ansatz eine Verbesserung in mehreren Bereichen für Rapid Prototyping dar. Die mit dieser Entwurfsumgebung mögliche Modellierung heterogener Systeme auf Basis eines standardisierten Datenformats stellt erstmals eine systematische Methodik zur Echtzeitcodeerzeugung dar. Die methodischen Betrachtungen und Verbesserungen im Bereich des Schedulings sowie bei der Abarbeitung des Echtzeitcodes ermöglichen den Einsatz von Rapid Prototyping bei Anwendungen, die höchste Anforderungen an die Abarbeitungsgeschwindigkeit stellen. Eine Analyse des erzeugten Echtzeit-Codes und eine Bewertung der erreichten Fortschritte schließen die Arbeit ab.

1.3 Gliederung

Die Arbeit wurde in die folgenden Abschnitte gegliedert:

- In Kapitel 2 werden allgemeine Aspekte des Entwurfs elektronischer Systeme wiedergegeben. Dabei werden zuerst die Grundlagen der Modellierung diskreter, kontinuierlicher und heterogener Systeme behandelt. Nach Einführung von Entwurfsphasen und Lebenszyklusmodellen, werden Rapid Prototyping und Hardware-in-the-Loop als ergänzende Systementwurfstechniken eingeführt. In einem weiteren Abschnitt wird gezielt auf den Entwurf von Echtzeitsystemen und die Verwendung von Echtzeitbetriebssystemen eingegangen.
- Kapitel 3 behandelt eine Analyse des Stands der Technik im Bereich von Rapid Prototyping Systemen. Dabei werden sowohl kommerzielle als auch universitäre Ansätze vorgestellt. Die vorgestellten Systeme umfassen FPGA-basierte Systeme, die über die Synthese von VHDL-Code den Aufbau von elektronischen Systemen erlauben, als auch DSP-, Mikrocontroller- oder Mikroprozessor-basierte Systeme. Zum Ende des Kapitels erfolgt eine Bewertung der vorgestellten Systeme.

- In Kapitel 4 erfolgt die Vorstellung des eigenen Ansatzes. Dazu werden zuerst detailliert die Anforderungen an ein Rapid Prototyping System erarbeitet. Aufbauend auf diesen Anforderungen wird das Gesamtkonzept für das Rapid Prototyping System, bestehend aus einer integrierten Entwurfsumgebung, der Echtzeitcode-Erzeugung und der Anbindung an die Systemumgebung vorgestellt.
- In Kapitel 5 wird auf Basis der in Kapitel 4 erarbeiteten Anforderungen eine Entwurfsumgebung für elektronische Systeme vorgestellt, die den heterogenen Systementwurf und die Kopplung mehrerer Entwurfswerkzeuge erlaubt.
- Kapitel 6 präsentiert die nach der Modellierung vorgenommene Abstraktion der Modelldaten. Dazu erfolgt eine Einführung in die Meta-Modellierung, die als einzige Technik die komplexen Anforderungen erfüllt, die bei der Modellierung heterogener elektronischer Systeme auftreten. Als Datenformat wurde das standardisierte Datenaustauschformat CDIF (*CASE Data Interchange Format*) ausgewählt. Erstmals werden komplexe Systeme über dieses Datenformat gekoppelt, da die eigentliche Absicht von CDIF im Austausch von Daten zwischen CASE-Werkzeugen liegt. Zum Ende des Kapitels erfolgt eine Bewertung des Datenformats.
- Kapitel 7 behandelt die Systemkopplung in Echtzeit. Dazu wird auf ereignis- und zeitgesteuertes Scheduling eingegangen. Der Schwerpunkt wird auf zeitgesteuertes Scheduling gelegt, da die Überprüfbarkeit der Echtzeitbedingungen und die zur Abarbeitung kontinuierlicher Systeme benötigte Zeitbasis ereignisgesteuertes Scheduling ausschließt. Um eine optimale Systemleistung zu erreichen, wird das ratenmonotone Scheduling zu pseudoraten-basiertem Scheduling verfeinert. Ein weiterer Abschnitt befaßt sich mit der Abschätzung von Ausführungszeiten von diskreten und kontinuierlichen Modellen, die für die Berechung der minimalen zeitlichen Auflösung benötigt wird. Eine Betrachtung der Auswirkungen des pseudoraten-basierten Schedulings auf den Systementwurf schließt dieses Kapitel ab.
- Nach Einführung der notwendigen Schedulingalgorithmen wird in Kapitel 8 auf die Erzeugung des Echtzeitcodes eingegangen. Der Echtzeitcode umfaßt drei Komponenten, die als Modellcode, Ein-/Ausgabecode und Schedulingcode bezeichnet werden. Der Modellcode besteht aus den Komponenten für diskrete und kontinuierliche Systeme sowie der Kopplung der beiden Systeme. Der Aufbau des Ein-/Ausgabecode wird anhand der eingesetzten Hardware erläutert. Die Algorithmen zum Scheduling werden hier konkret vorgestellt und die Zeitverhältnisse diskutiert.
- Kapitel 9 faßt die erzielten Ergebnisse zusammen. Zu Beginn wird die entwickelte Entwurfsumgebung vorgestellt. Danach folgen die Ergebnisse der Codegenerierung, die detailliert nach Übersetzungszeit, Abarbeitungsgeschwindigkeit und Codegröße jeweils für die einzelnen Bereiche wiedergegeben sind. In diesen Abschnitten wird auch eine Analyse der Ergebnisse vorgenommen. Um die industrielle Einsetzbarkeit des Systems nachzuweisen, wird im letzten Abschnitt die Entwicklung einer Fensterheberansteuerung beschrieben.
- In Kapitel 10 wird eine Zusammenfassung der Arbeit präsentiert und die Grenzen der Arbeit beleuchtet. Die Arbeit wird durch einen Ausblick auf die möglichen zukünftigen Entwicklungen in diesem Bereich abgeschlossen.

2 Grundlagen

Zum Verständnis der im Rahmen dieser Arbeit entwickelten Konzepte, Methoden und Werkzeuge sind sowohl die detaillierte Kenntnis des Vorgehens beim Systementwurf, der dabei verwendeten Modellierungskonzepte sowie Kenntnisse im Bereich der Echtzeitverarbeitung notwendig.

Definition 2.1: Unter einem *System* versteht man eine Anzahl an verbundenen Elementen, um ein vorgegebenes Ziel unter Anwendung von vorgegebenen Funktionen zu erreichen.

Dieses Kapitel wird deswegen einen Überblick über den Systementwurf unter Verwendung von Lebenszyklusmodellen vermitteln, dann auf die diskreten und kontinuierlichen Modellierungskonzepte von rechnergestützten Entwurfswerkzeugen eingehen und daraus die Motivation für das zentrale Thema dieser Arbeit, den schnellen Prototypenentwurf (engl. *Rapid Prototyping*), herleiten. Abschließend wird auf die Verbindung von realem Steuergerät und simulierter Umgebung (engl. Hardware-in-the-Loop), die in Ergänzung zu Rapid Prototyping eingesetzt werden kann, sowie auf Echtzeitsysteme eingegangen.

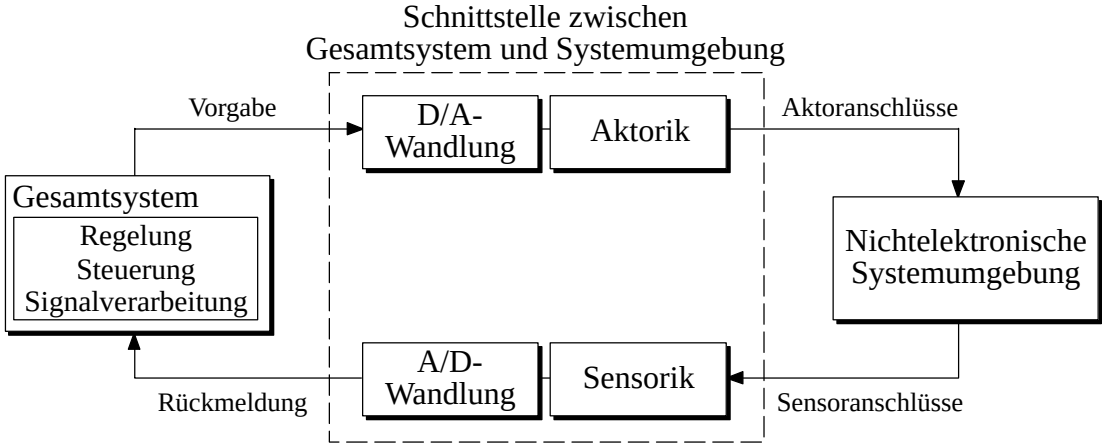


Bild 2-1: Struktur eines Meß-, Steuerungs- und Regelungssystems

Die hier betrachteten Systeme gehören einer Klasse von Systemen an, die mit dem Begriff ’Meß-, Steuerungs- und Regelungssysteme’ bezeichnet wird.

Definition 2.2: Unter der Steuerung eines Systems wird die Einflußnahme von Eingangsgrößen auf weitere Größen, die als Ausgangsgrößen bezeichnet werden, vestanden. Kennzeichen für die Steuerung ist ein offener oder ein geschlossener Wirkungsweg, bei dem die durch die Eingangsgrößen beeinflußten Ausgangsgrößen nicht fortlaufend und nicht wieder über dieselben Eingangsgrößen auf sich selbst wirken (nach DIN-Norm 19226).

Die Begriff der Steuerung wird oftmals nicht nur für den Vorgang des Steuerns, sondern auch für die Gesamtanlage verwendet, in der die Steuerung stattfindet.

Definition 2.3: Eine Regelung ist ein Vorgang, bei dem fortlaufend eine Größe (Regelgröße) mit einer anderen Größe (Führungsgröße) verglichen und im Sinne einer Angleichung an die Führungsgröße beeinflußt wird. Kennzeichen für eine Regelung ist ein geschlossener Wirkungsablauf, bei dem die Regelgröße im Wirkungsweg des Regelkreises fortlaufend sich selbst beeinflußt (nach DIN-Norm 19226).

Der grundsätzliche Aufbau eines Meß-, Steuerungs- und Regelungssystems ist in Bild 2-1 dargestellt. Die *Sensorik* dient der Wandlung nichtelektronischer Primärgrößen der Systemumgebung in analoge, elektrische Sekundärgrößen. Diese werden über eine analog/digital Wandlung (A/D-Wandlung) in das Gesamtsystem eingespeist. Das Gesamtsystem führt die meß-, steuer- und regelungstechnischen Aufgaben durch und gibt seine Vorgabesignale über eine D/A-Wandlung und die entsprechende *Aktorik*, die eine Wandlung der elektronischen in nichtelektronische Größen vornimmt, an die Systemumgebung weiter.

2.1 Systemmodellierung

Der Begriff *Modell* kennzeichnet ein System als Abbild eines realen oder hypothetisch angenommenen Untersuchungsobjekts (Original), das alle für ein bestimmtes Untersuchungsziel relevanten Eigenschaften des Originals aufweist, um die Erfassung oder Beherrschung des Originals zu ermöglichen oder zu erleichtern, bzw. es zu ersetzen [WUSL86]. Im Zusammenhang mit der Systemmodellierung kann ein Modell wie folgt definiert werden.

Definition 2.4: Ein Modell ist die Abbildung eines Systems in ein anderes begriffliches oder gegenständliches System, das aufgrund der Anwendung bekannter Gesetzmäßigkeiten, einer Identifikation oder auch getroffener Annahmen gewonnen wird und das System bezüglich ausgewählter Fragestellungen hinreichend genau abbildet (nach DIN-Norm 19226).

Modelle können mittels unterschiedlicher Notationen beschrieben werden, die für die Darstellung eines Teilgebiets entwickelt wurden. Diese Teilgebiete umfassen beispielsweise die Modellierung diskreter Systeme, kontinuierlicher Systeme oder die Modellierung von Datenflüssen. Zur Vereinfachung der Modellierung existieren sogenannte CASE-Werkzeuge, die eine grafische Oberfläche für die Modellierung zur Verfügung stellen.

Definition 2.5: *CASE* (Computer Aided Software/Systems Engineering) stellt einen Überbegriff von allen rechnergestützten Tätigkeiten bei der Software- oder Systementwicklung dar.

Definition 2.6: Unter *Systems Engineering* versteht man die Anwendung von mathematischen und physikalischen Wissenschaften, um Systeme zu entwickeln, die ökonomisch die Materialien und Kräfte der Natur zum Nutzen der Menschheit benutzen.

Definition 2.7: Unter *Concurrent Engineering* wird i.a. das gleichzeitige, nebenläufige Arbeiten aller an einem Projekt beteiligten Fachabteilungen in allen Phasen der Planung, der Entwicklung und der Fertigung eines Produkts verstanden [Müll94].

Definition 2.8: *Requirements Engineering* umfaßt Methoden, Beschreibungsmittel und rechnergestützte Werkzeuge zur Entwicklung einer vollständigen und konsistenten Spezifikation des Zielsystems. Es unterstützt eine ingenieursmäßige Ermittlung aller Anforderungen, Ziele und Randbedingungen einer Systementwicklung sowie ihre Aufbereitung zur Realisierung in Hardware und Software [Müll94].

Definition 2.9: Der Begriff *Simultaneous Engineering* wird häufig synonym zu Concurrent Engineering benutzt, steht aber in engerem Sinne dafür, daß mehrere Entwicklungsteams parallel nebeneinander verschiedene Module eines Gesamtsystems entwickeln [Müll94].

Die Modellierung jedes der oben erwähnten Teilgebiete ist jedoch nicht nur durch eine einzige Notation möglich. Durch parallele Entwicklungen oder Übereinstimmungen entstanden unterschiedliche Notationen für den gleichen Sachverhalt. Die wichtigsten Notationen werden in den folgenden Kapiteln für jedes Teilgebiet vorgestellt.

2.1.1 Modellierung diskreter Systeme

Bei diskreten Systemen umfaßt die Systemausgabe eine Reihe von festen, diskreten Werten. Neben der Ausgabe spielt häufig auch die Vorgeschichte (Historie) eines bestimmten Zustands eine wesentliche Rolle. Die Vorgeschichte wird dabei durch die zuvor durchlaufenen Systemzustände gebildet. In den nachfolgenden Abschnitten werden drei Methoden für den Entwurf diskreter Systeme betrachtet.

Endliche Automaten

Endliche Automaten (engl. *finite state machines*) wurden bereits in den fünfziger Jahren [Meal55] [Moor56] entwickelt und verarbeiten im Gegensatz zu Schaltnetzen nicht nur die aktuellen Eingaben, sondern auch die Vorgeschichte des Systems. Die allgemeine, rekursive Struktur eines Automaten ist in Bild 2-2 dargestellt. λ stellt dabei die Ausgabefunktion und δ die Überführungsfunktion des Automaten dar. Beide Funktionen sind reine schaltalgebraische Verknüpfungen ohne interne Rückführung.

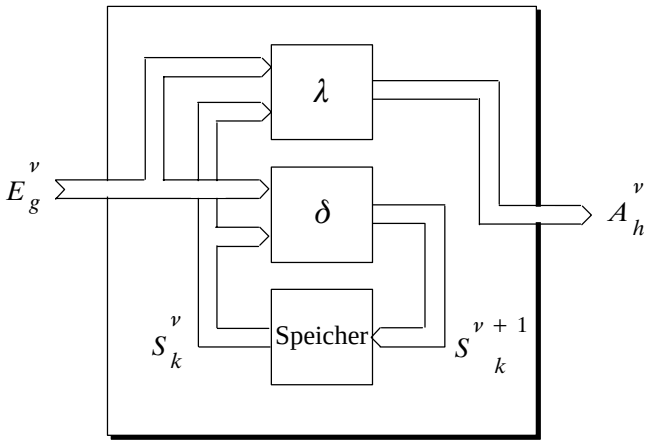


Bild 2-2: Allgemeine, rekursive Struktur eines Automaten

Der Automat besitzt das endliche Eingabealphabet $E = \{E_1, E_2, \ldots, E_g, \ldots, E_u\}$ und das Ausgabealphabet $A = \{A_1, A_2, \ldots, A_h, \ldots, A_v\}$. Eine als Speicher $S = \{S_1, S_2, \ldots, S_k, \ldots, S_w\}$ be-

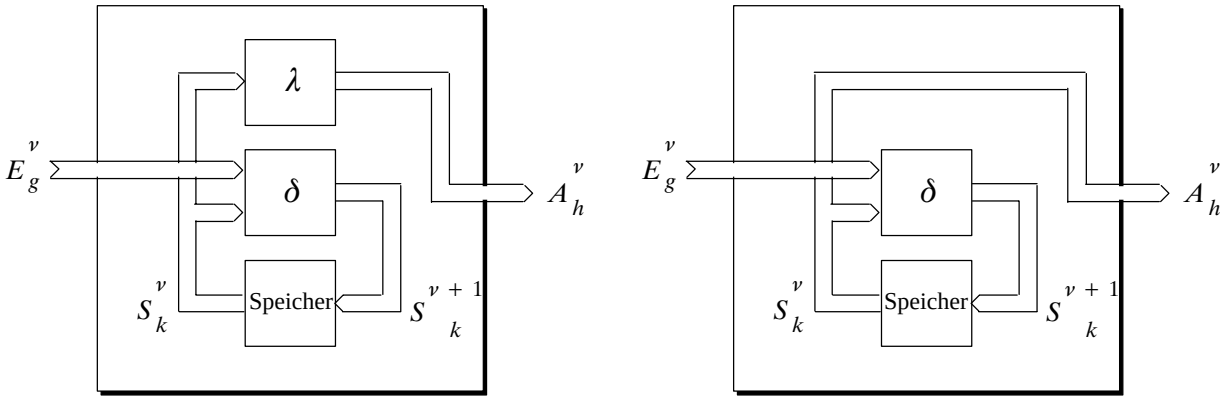
zeichnete Einheit nimmt den momentanen Zustand S^{ν} auf und hält ihn fest. Der Ordnungsindex ν dient der Unterscheidung der zeitlichen Folge der Elemente. Beim Anlegen eines neuen Eingabeelements muß der Speicher $S^{\nu+1}$ übernehmen und den Funktionen δ und λ als Argument zur Verfügung stellen. Mit dem inneren Zustand S lassen sich somit für einen endlichen Automaten die folgenden Gleichungen bilden:

$$A_h^{\nu} = \lambda\left(E_g^{\nu},\ S_k^{\nu}\right) \qquad (2.1)$$

$$S_k^{\nu+1} = \delta\left(E_g^{\nu},\ S_k^{\nu}\right) \qquad (2.2)$$

Insgesamt ist ein endlicher Automat also durch das Quintupel $(E, A, S, \delta, \lambda)$ gekennzeichnet, das aus drei endlichen Mengen und zwei Abbildungen zwischen diesen Mengen besteht.

Bei den endlichen Automaten unterscheidet man gewisse Typklassen. Eine Typklasse stellt der sogenannte *Mealy-Automat* dar, bei dem die Ausgabe abhängig vom Zustand und der Eingabe ist und somit den allgemeinsten Fall darstellt. Dieser Automat wurde bereits in Bild 2-2 wiedergegeben. Der *Moore-Automat* kann aus dem Mealy-Automat abgeleitet werden, da bei ihm die Ausgabe allein vom Zustand abhängt (Bild 2-3 a). Einen Spezialfall des Moore-Automaten stellt der *Medwedew-Automat* dar, bei dem als Ausgabe der Zustand selbst dient (Bild 2-3 b).



a. b.

Bild 2-3: Struktur des Moore-Automaten (a) und des Medwedew-Automaten (b)

Es existieren grafische und tabellarische Methoden zur Festlegung des Verhaltens eines Automaten, beispielsweise in Form eines *Automatengraphen* (Zustände werden als Knoten, Zustandsübergänge als verbindende, gerichtete Kanten repräsentiert) und einer *Automatentafel* (tabellenförmige Darstellung, die durch Bildung des karthesischen Produkts aus Eingabe- und Zustandsmenge erreicht wird [Lipp95]). Diese Methoden sind jedoch aus Sicht praktischer Aufgabenstellungen teilweise unbefriedigend, da die Betrachtung aller Variablen den Wunsch nach kompakteren Formen aufkommen läßt. Einen grafischen Ansatz dazu stellt das *Ablaufdiagramm* dar. Das Ablaufdiagramm nutzt aus, daß nicht alle Komponenten des Eingabevektors für den Übergang zum Folgezustand notwendig sind. Die Komponenten, die für den Übergang notwendig sind, bezeichnet man als relevante Eingabevariablen. Die vier Grundelemente Zustand, Ausgabe, Abfrage und Anfangszustand eines Ablaufdiagramms sind in Bild 2-4 abgebildet. Verbunden werden die Grundelemente mit unidirektionalen Pfeilen.

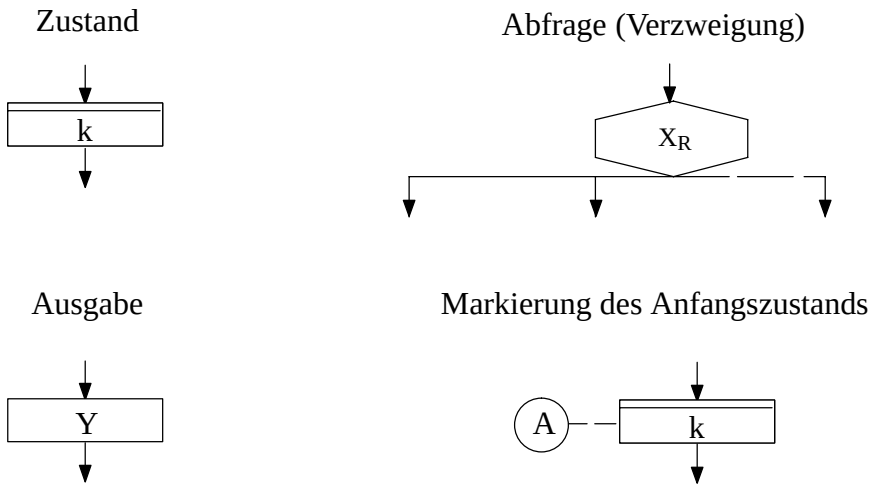


Bild 2-4: Grafische Symbole des Ablaufdiagramms

Das Ablaufdiagramm der endlichen Automaten vom Typ Mealy und Moore, das eine Zusammenstellung der eingeführten Elemente beinhaltet, ist in Bild 2-5 dargestellt.

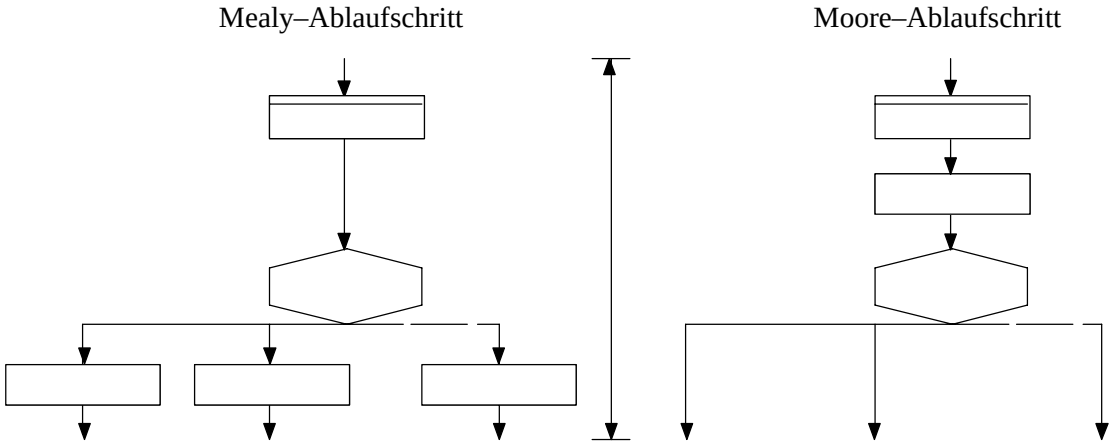


Bild 2-5: Ablaufdiagramm eines Mealy- und Moore-Automaten [Lipp95]

Durch ihre grafische Notation eignen sich die endlichen Automaten für den Einsatz in CASE-Werkzeugen. Dies wurde beispielsweise im Entwurfsprogramm LOG/iC™ der Firma Isdata mit STATE/view™ realisiert. Die endlichen Automaten können bei der Entwicklung komplexer Systeme allerdings nur bedingt eingesetzt werden, da diese Beschreibungsform bei einer hohen Anzahl an Zuständen schnell unübersichtlich wird.

Als Entwurfsbeispiel für die Modellierung diskreter Systeme wird die Modellierung eines einfachen Fensterhebersteuergeräts mit Positionserkennung vorgenommen. Der Fensterhebersteuergerät besteht aus einem Kippschalter, die mit `H` (für Fahrt hoch) und `R` (für Fahrt runter) bezeichnet sind. Ist keine der beiden Tasten gedrückt, so befindet sich der Fensterhebermotor in Ruhe (`MH`=’0’, `MR`=’0’). Ist die Taste `H` betätigt, fährt das Fenster nach oben (`MH`=’1’), bei `R` nach unten (`MR`=’1’). Durch die Auswertung der vom Motor gelieferten Hallsignale, die jede Motorumdrehung mit einer Veränderung des Hallsignals `Hall` von ’1’ nach ’0’, bzw. von ’0’ nach ’1’ quittieren, kann eine Positionserkennung durchgeführt werden. Die Drehrichtung des Motors kann über das Signal `MH` ermittelt werden.

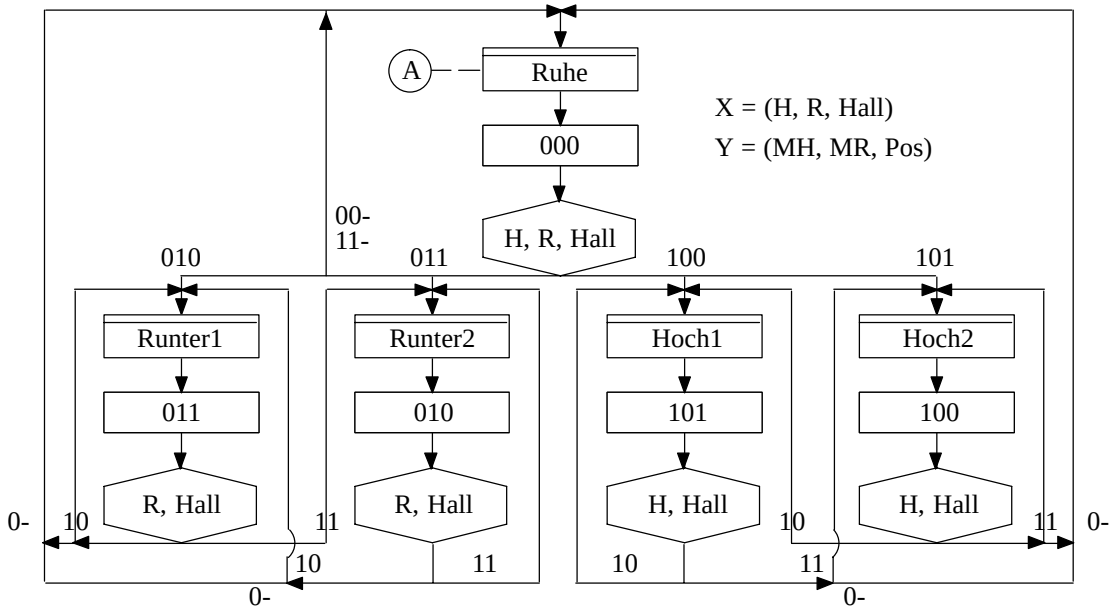


Bild 2-6: Modellierung eines einfachen Fensterhebers mit endlichen Automaten

In Bild 2-6 ist die Modellierung dieses Systems mit Hilfe von endlichen Automaten dargestellt. Zu Beginn befindet sich das Modell im Zustand Ruhe, der als Ausgaben die Motoren in ihren Ruhezustand versetzt, das Positionserkennungssignal `Pos` auf 0 setzt. Das Abfrageelement enthält drei Eingabevariablen. Je nach Wert dieser Eingabevariablen werden unterschiedliche Folgezustände angenommen. Besitzen `H` und `U` den Wert 0, so wird unabhängig von `Hall` als Folgezustand Ruhe angenommen und der Motor bleibt ausgeschaltet. Dieser Zustand wird auch dann angenommen, wenn beide Tasten gleichzeitig gedrückt werden. Wird die Taste `R` gedrückt, so wird entweder in Zustand Runter1 oder Runter2 gewechselt. Dies ist abhängig vom Wert des Hallsignals zum Zeitpunkt des Wechsels. Hatte das Hallsignal den Wert 0, so wird in den Zustand Runter1 gewechselt. Die Ausgabevariablen dieses Zustands steuern den Fensterhebermotor so an, daß `MH`=0 ist und `MR`=1 und der Fensterhebermotor bewegt sich nach unten. Das Signal `Pos` wird auf 1 gesetzt, um einen Positionswechsel anzuzeigen. Wechselt das Signal `R` wieder auf 0, so wird der Zustand Ruhe angenommen. Bleibt `R` bestehen, so wird bei einem Wechsel des Hallsignals (`Hall`=1) zum Zustand Runter2 übergegangen, anderenfalls der Zustand Runter1 beibehalten. In derselben Art wird der Rest des Modells für H=1 abgearbeitet, dabei ist allerdings `MH`=1 und `MR`=0.

Es ist einzusehen, daß diese Form der Modellierung nur für Anforderungen mit geringer Komplexität eingesetzt werden kann.

Petri-Netze

Eine weitere Beschreibungsform für diskrete Systeme stellen Petri-Netze [Petr62] zur Verfügung. Neben der grafischen Repräsentation durch einen gerichteten, bipartiten Graph besteht auch ein mathematischer Formalismus zur Beschreibung von Systemen. Ein Graph besteht aus *Stellen*, die einer Zwischenablage von Informationen entsprechen, *Transitionen*, die durch *Flußrelationen* mit Stellen verbunden sind sowie aus *Marken*, die als dynamische Elemente die aktiven Stellen charakterisieren. Bei der grafischen Darstellung werden Stellen als Kreise, Transitionen als Balken, Flüsse

durch Pfeile und Marken als kleine, ausgefüllte Kreise repräsentiert (siehe Bild 2-7). Dabei dürfen Stellen über Flußrelationen mit Transitionen verbunden werden und umgekehrt.

Bild 2-7: Grafische Symbole des Petri-Netzes

Zur Beschreibung der dynamischen Vorgänge werden Stellen durch Marken gefüllt. Eine Stelle kann dabei mehrere Marken aufnehmen. Der Bewegungsablauf der Marken im Netz wird durch folgende Schaltregel definiert:

- Eine Transition *T* kann schalten, wenn jede Eingabestelle von *T* mindestens eine Marke enthält.
- Schaltet eine Transition, wird aus jeder Eingabestelle eine Marke entfernt und zu jeder Ausgabestelle eine Marke hinzugefügt.

Mathematisch läßt sich ein Petri-Netz durch ein 6–Tupel (P, T, F, K, W, M_0) beschreiben. Dabei bedeuten $P = \{p_1, p_2, \ldots, p_m\}$ die Menge der Stellen, $T = \{t_1, t_2, \ldots, t_n\}$ die Menge der Transitionen, F die Flußrelation, $K : P \rightarrow \mathbb{N} \cup \{\infty\}$ die Kapazitäten der Stellen, $W : F \rightarrow \mathbb{N}$ die Kantengewichte und $M_0 : P \rightarrow \mathbb{N}_0$ die Anfangsmarkierungen. Dabei gelte $P \cap T = \emptyset$ und $F \subseteq (P \times T) \cup (T \times P)$. Durch Verwendung dieses Formalismus kann ein Petri-Netz vollständig beschrieben werden und eine rechnergestützte Analyse des Systems ist möglich.

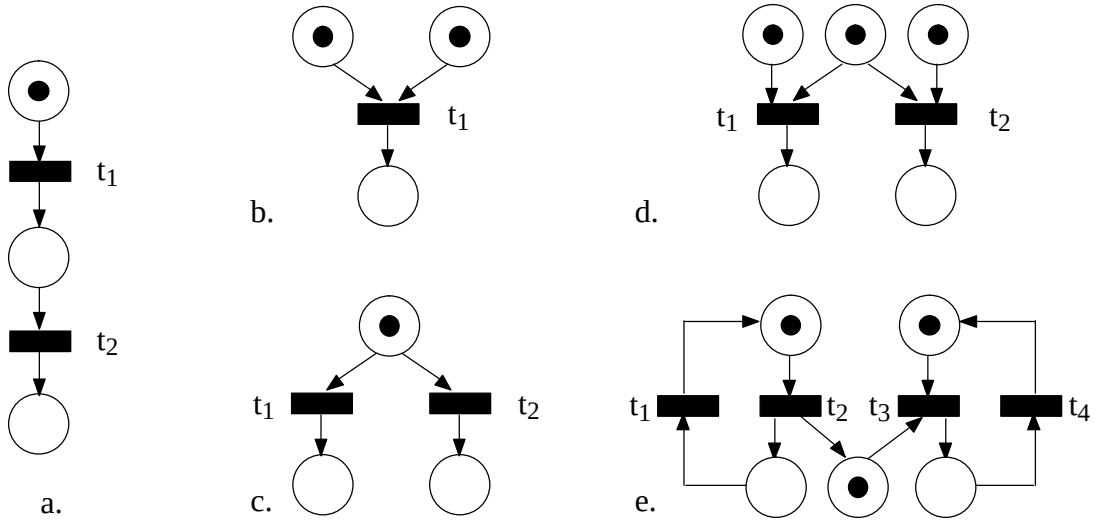


Bild 2-8: Modellierungen mit Petri-Netzen

Petri-Netze haben im Vergleich zu Zustandsautomaten gewisse Ähnlichkeiten, besitzen aber darüberhinaus einige Erweiterungen. Die Stellen eines Petri-Netzes lassen sich als Zustände, die Transitionen als Zustandsänderungen auffassen. Im Gegensatz zu einem Zustandsautomaten kann sich ein durch ein Petri-Netz beschriebenes System zu einem Zeitpunkt in mehreren Zuständen aufhalten, die durch die aktuelle Markenbelegung definiert sind. Je nach Struktur des Petri-Netzes können unabhängig voneinander mehrere Zustandsübergänge stattfinden. Bild 2-8a zeigt, wie *Sequentialität* mit Hilfe eines Petri-Netzes modelliert wird. Die Synchronisation eines Petri-Netzes ist in

Bild 2-8b wiedergegeben. Beide Marken müssen dabei vorhanden sein, damit geschaltet wird. Bild 2-8c stellt eine nicht-deterministische Verzweigung dar, da die Transitionen t_1 und t_2 einen Konflikt aufweisen. Ein Konflikt entsteht, wenn zwei oder mehrere Transitionen schaltbereit sind, die mindestens eine gemeinsame Eingangsstelle haben und das Schalten der einen Transition die Schaltbereitschaft der anderen zerstört. Bild 2-8d zeigt einen Ressourcenkonflikt, da die Transitionen t_1 und t_2 sich um die mittlere Marke streiten. Die Modellierung der Synchronisation einer Nebenläufigkeit ist in Bild 2-8e dargestellt [Teic97]. Die Transitionen t_2 und t_3 können simultan schalten und die Synchronisation zwischen den nebenläufigen Systemen kann durch die spezielle Netzstruktur erzwungen werden.

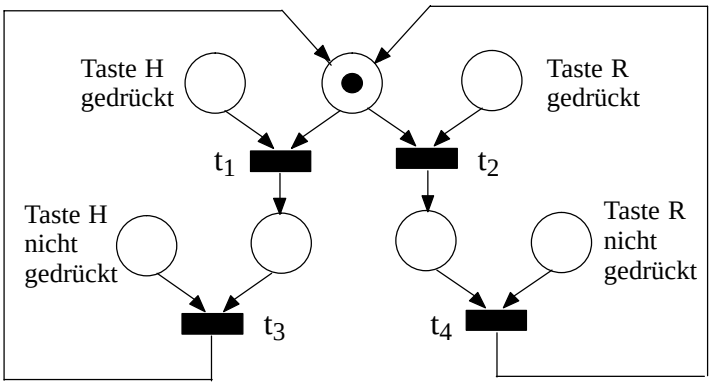


Bild 2-9: Modellierung eines Fensterhebers mit Petri-Netzen

Am Beispiel der Modellierung eines Fensterhebers soll die Modellierung mit Petri-Netzen verdeutlicht werden. Dabei soll dieselbe Funktionalität modelliert werden, die bereits in Kapitel 2.1.1 für die endlichen Automaten verwendet wurde. In Bild 2-9 ist die Modellierung mit Hilfe eines Petri-Netzes dargestellt. Zu Beginn wird das System keine Aktion durchführen, da sowohl die Transition t_1 als auch Transition t_2 nicht schaltbereit sind. Erst wenn die Taste `H` gedrückt wird, wird eine Marke in die entsprechende Stelle gegeben, so daß die Transition t_1 schalten kann und das Fenster nach oben fahren wird. Der Zustand des Hochfahrens hält solange an, bis die Taste `H` nicht mehr gedrückt wird und eine Marke in die entsprechende Stelle gegeben wird. Daraufhin wird die Transition t_3 schalten und das System in den Wartezustand zurückversetzen. Derselbe Ablauf wie beim Hochfahren geschieht, wenn nun die Taste `R` betätigt wird.

Wie bei den Automaten wird die Darstellung eines komplexen Systems aufgrund der großen Anzahl von Stellen und Transitionen sehr unübersichtlich. Die Modellierung des Fensterhebers mit Petri-Netzen wurde hier im Vergleich zur Modellierung mit endlichen Automaten vereinfacht (keine Positionserkennung), da die resultierende Modellierung sehr aufwendig ausfällt. Eine Darstellung von Hierarchien oder das Einbinden von Zeitverhalten ist nur durch höhere Petri-Netze möglich, deren Notation nicht einheitlich ist. Diese höheren Petri-Netze sind nur schwer zu erstellen und zu analysieren.

Statecharts

Wegen der Mängel zustandsbasierter Beschreibungen wie Zustandsautomaten oder Petri-Netzen (mangelnde Übersichtlichkeit bei hoher Zustandszahl, Unterstützung rein sequentieller Abläufe, keine Modularisierungs– oder Strukturierungsmöglichkeit) wurde nach Ansätzen gesucht, die

diese prinzipiellen Nachteile nicht besitzen. Einen der bekanntesten Ansätze stellen die Statecharts dar, die von Harel vorgeschlagen wurden [Hare87]. Statecharts erweitern die klassischen Zustandsautomaten um:

- Hierarchie
- Nebenläufige Zustände (parallele Ausführung von mehreren Teilen eines Automaten)
- Bedingte Zustandsübergänge
- Hybride Zustandsautomaten (Kombination von Mealy– und Moore-Automaten)
- Zustände mit Rücksprungadresse

Dabei werden zur Darstellung von Zuständen (engl. *states*) Rechtecke mit abgerundeten Ecken verwendet. Bei Statecharts können drei Arten von Zuständen unterschieden werden:

- Elementarzustände (engl. *basic states*), die keine weiteren Unterzustände besitzen.
- Parallele Zustände (engl. *AND states*), die unabhängige oder zusammenwirkende Unterzustände aufweisen. Zwischen diesen Unterzuständen besteht eine Und-Beziehung, d.h. alle Unterzustände werden parallel (engl. *orthogonal*) ausgeführt.
- Hierarchische Zustände (engl. *OR states*), die weitere Unterzustände besitzen. Die Unterzustände befinden sich in einer Exklusiv-Oder-Beziehung, d.h. zu einem Zeitpunkt ist genau ein Zustand aktiv.

Startzustände werden durch einen Pfeil mit einem kleinen, schwarz ausgefüllten Kreis am Pfeilanfang (engl. *default transition*) gekennzeichnet. Zur Darstellung eines Oberzustands (engl. *super–state*), der aus mehreren, eventuell gleichzeitig auszuführenden Unterzuständen besteht, wird eine gestrichelte Linie eingeführt, die den Oberzustand in mehrere nebenläufige Komponenten aufteilt. Ein Zustand mit Gedächtnis (engl. *history connector*) wird als nicht ausgefüllter Kreis mit innenstehendem H dargestellt. Zustandsübergänge (engl. *transitions*) werden durch Pfeile repräsentiert und durch Ereignisse ausgelöst. Die auslösenden Ereignisse werden an den jeweiligen Zustandsübergang geschrieben.

Definition 2.10: Unter einem *Ereignis* (engl. *event*) versteht man eine zweiwertige Variable, die den Wert “wahr” nur für infinitesimale Zeit annimmt.

Ereignisse können auch von Bedingungen erzeugt werden, die in eckigen Klammern dargestellt sind.

Definition 2.11: Unter einer *Bedingung* (engl. *condition*) versteht man eine zweiwertige Variable mit den Werten “wahr” und “falsch”

In diesem Fall findet ein Zustandsübergang statt, wenn zu dem Übergangszeitpunkt die entsprechende Bedingung gültig ist. Aktionen (engl. *actions*), die bei einem Zustandsübergang ausgeführt werden sollen, werden angehängt an Ereignisse nach einem Schrägstrich-Zeichen aufgeführt. Die an einen Zustandsübergang gebundenen Ereignisse und Aktionen werden als *labels* bezeichnet. Die verwendeten Symbole sind in Bild 2-10 zusammenfassend aufgeführt.

Eine Hierarchiebildung, mit der eine Modularisierung von Systemen erfolgen kann, geschieht mit Hilfe von Oberzuständen, in die weitere Zustände eingetragen werden. Befindet sich ein System in

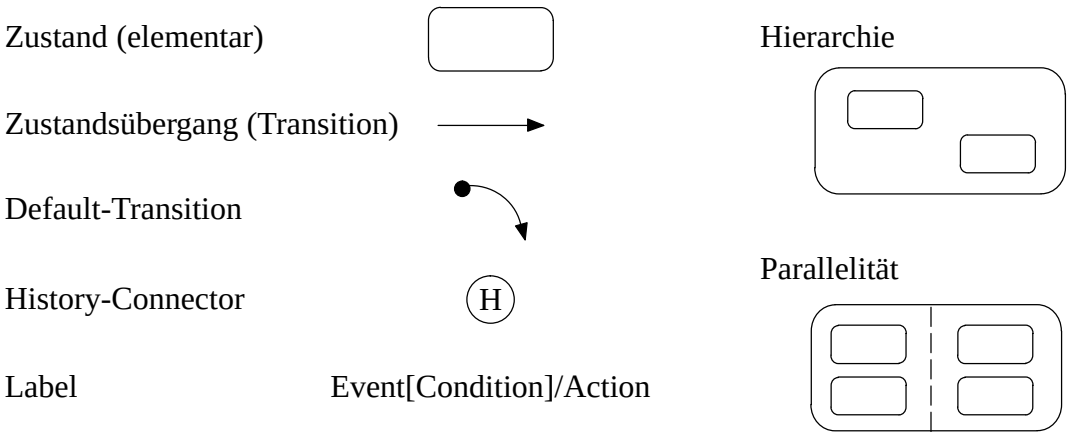


Bild 2-10: Symbole eines Statecharts

solch einem Oberzustand, so ist genau ein darin enthaltener Zustand aktiv. Wird der Oberzustand verlassen, wird auch der gerade aktive Unterzustand verlassen. Bild 2-11 zeigt die Modellierung

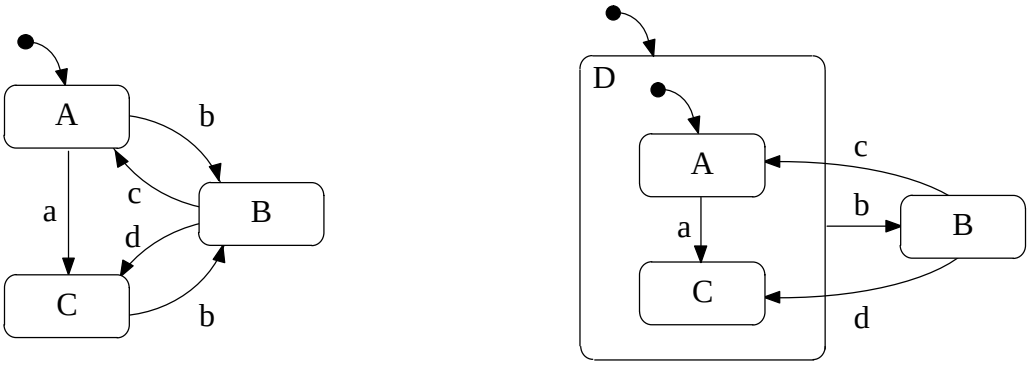


Bild 2-11: Modellierung ohne und mit Hierarchie

von Statecharts ohne und mit Verwendung von Hierarchie. Beide Automaten modellieren die gleiche Funktionalität. Durch Einführung des Oberzustands D, der die Zustände A und C umfaßt, können die aus allen seinen Zuständen entspringenden Transitionen b zu einer einzigen Transition b vom Oberzustand D aus zusammengefaßt werden. Die Semantik von D beschreibt eine exklusive Oder–Beziehung der innen liegenden Zustände A und C, d.h. innerhalb des Zustands D kann der Automat gleichzeitig nur einen der beiden Zustände annehmen. Der Zustand, der als Startzustand bei Eintritt in D angenommen wird, wird durch eine Default-Transition gekennzeichnet. Gerade bei komplexen Systemen kann durch Einsatz der Hierarchie die Anzahl der Pfeile wesentlich vermindert werden. Besonders geeignet ist die Hierarchie (in Verbindung mit dem History-Connector) für Interruptsteuerungen, da nicht für jeden von einem Interrupt Ereignis betroffenen Zustand der Übergang zum Interrupt Folgezustand modelliert werden muß. Die Darstellung gewinnt dadurch deutlich an Übersichtlichkeit. Der Einsatz eines History-Connectors erlaubt den Rücksprung zu dem Zustand, der vor Eintreffen des Interrupt Ereignisses aktiv war (siehe auch Bild 2-15).

Viele technische Systeme zeichnen sich dadurch aus, daß parallel mehrere Arbeitsschritte ausgeführt werden. Eine Modellierung dieses Verhaltens, bei dem sich ein System gleichzeitig in mehreren Zuständen befindet, wird unter Verwendung von Oberzuständen möglich, die aus mehreren, gleichzeitig ausgeführten Unterzuständen bestehen. Bild 2-12 zeigt die Modellierung von State-

charts mit und ohne Verwendung von Parallelität. Der Oberzustand A besitzt zwei Komponenten B und C. Wird in den Oberzustand A eingetreten, befindet sich das System gleichzeitig in D und F. In diesem Fall spricht man vom *kombinierten Zustand* (D, F). Tritt das Ereignis a ein, so erfolgt ein Zustandsübergang von D nach E. Die Komponente C bleibt unverändert im Zustand F. Es ergibt sich also der kombinierte Zustand (E, F). Tritt das Ereignis b ein, so erfolgen simultan die Zustandsübergänge von E nach D und von F nach G.

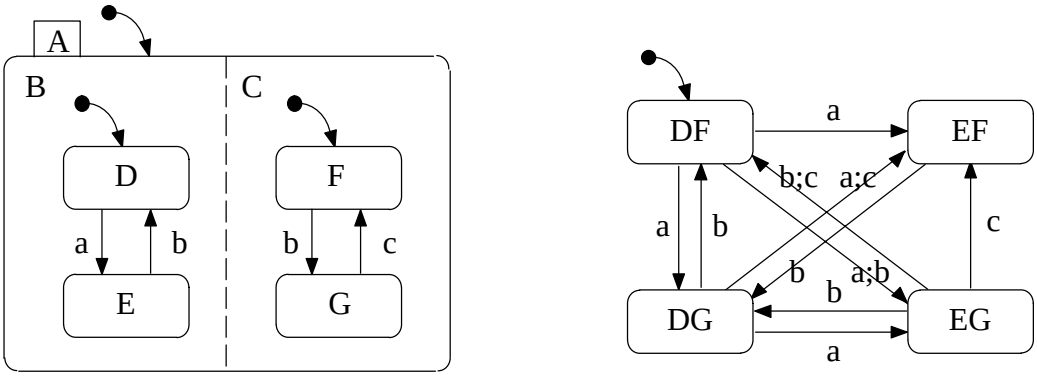


Bild 2-12: Modellierung mit und ohne Parallelität

Ohne Verwendung der Parallelität (siehe Bild 2-12 rechts) müssen alle möglichen kombinierten Zustände einzeln modelliert werden. Die Zahl der verwendeten Zustände kann dabei stark steigen. Im ungünstigsten Fall wächst die Zahl von Zuständen auf die Multiplikation der Zustandszahl aller verwendeten Komponenten (*Produktautomat*). In Bild 2-12 bedeutet dies, daß vier Zustände dargestellt werden müssen. Zusätzlich wächst jedoch auch die Anzahl der Zustandsübergänge, was zu einer extrem unübersichtlichen Darstellung führt.

Neben Aktionen, die bei einem Zustandswechsel ausgeführt werden, gibt es zusätzlich die Möglichkeit, Aktionen beim Eintritt in einen Zustand, im Zustand selbst oder bei Verlassen des Zustands auszuführen. Diese Modellierungsmöglichkeit wird als *Static Reaction* bezeichnet und besitzt die gleiche Syntax wie Labels einer Transition. Um die Bindung an den Zustand zu erreichen, existiert für jeden Zustand ein Zustandsformular (engl. *state form*), das die textuellen Informationen aufnehmen kann. Ist ein Zustandsformular eines Zustands ausgefüllt, so wird dem Namen des Zustands ein zusätzliches “>”-Zeichen vorangestellt.

Zur Vereinfachung und Vervollständigung der Statecharts gibt es zusätzliche Konstrukte, die dem Einsparen und Zusammenfassen von Zustandsübergängen oder der Erweiterung der hierarchischen Darstellung dienen. Diese Konstrukte werden als *Connectoren* bezeichnet.

Mit Condition- und Switch-Connectoren ist es möglich, Zustandsübergänge auf mehrere Zielzustände aufzuspalten, um die Verwendung von mehreren Zustandsübergängen zu vermeiden und so die Übersichtlichkeit zu erhöhen. Der *Condition-Connector* besitzt die Möglichkeit, die Verzweigung eines Zustandsübergangs in Abhängigkeit von Bedingungen (engl. *conditions*) darzustellen. Das in Bild 2-13 links dargestellte Beispiel zeigt die Auslösung eines Zustandsübergangs von Zustand A auf Zustand B oder C, wenn die Auslösung einer der Teil-Zustandsübergänge erfolgt. Dabei wirkt der Condition-Connector als Und-Verknüpfung der Teil-Zustandsübergänge, die im Beispiel durch das Eintreffen des Events E und der Condition C1 bzw. des Events E und der Condition C2 gekennzeichnet sind. Falls keine der beiden Übergangsbedingungen erfüllt ist, wird kein Zustands-

übergang ausgelöst. Ist mehr als eine Übergangsbedingung erfüllt, entsteht ein *Nichtdeterminismus*, d.h. das Verhalten der Modellierung kann nicht vorhergesagt werden, da gleichzeitig mehrere Fälle eintreten können.

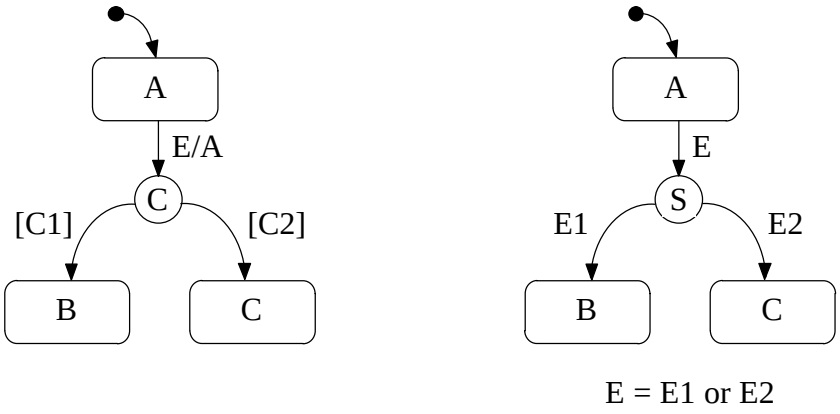


Bild 2-13: Condition-Connector und Switch-Connector

Der *Switch-Connector* dient der Verzweigung von Zustandsübergängen aufgrund von parametrisierten Ereignissen. Existiert, wie in Bild 2-13 rechts dargestellt, ein bereits definiertes Ereignis E = E1 or E2, wird ein Zustandsübergang vom Zustand A dann durchgeführt, wenn eines der Ereignisse E1 oder E2 auftritt. Ist mehr als eine Übergangsbedingung erfüllt, tritt ein Nichtdeterminismus auf.

Der *Junction-Connector* (Bild 2-14) führt mehrere Zustandsübergänge in einem Knoten (engl. *junction*) zusammen und führt von dort an als ein einziger Zustandsübergang weiter zu einem Zielzustand. Mit dem Junction-Connector kann die Übersichtlichkeit der Modellierung verbessert werden.

Der *Diagram-Connector* (Bild 2-14) kann verwendet werden, wenn der Start- und Zielzustand eines Zustandsübergangs räumlich entfernt liegt oder auf mehrere Statecharts verteilt ist. Ohne die explizite Verbindung durch einen Zustandsübergang stellen die beiden verwendeten Diagram-Connectoren eine Verbindung dar. Beispielsweise wurde in Bild 2-14 rechts eine Verbindung von Zustand A nach Zustand B über den Diagram-Connector D1 hergestellt.

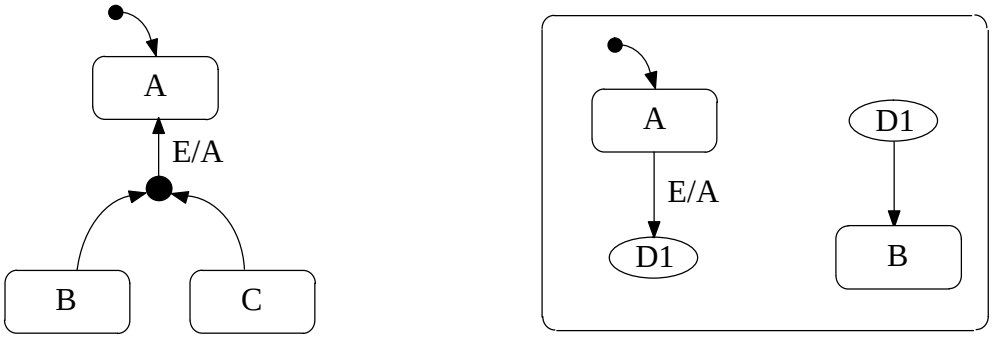


Bild 2-14: Junction- und Diagram-Connector

Nach dem Verlassen eines komplexen Zustands, der aus vielen Hierarchieebenen und Parallelitäten bestehen kann, existieren normalerweise keine Informationen darüber, welche Unterzustände aktiv waren. Die Statecharts bieten jedoch ein Speichermechanismus an, der einen Wiedereintritt in die verlassenen Systemzustände erlaubt und mit *History-Connector*, bzw. *Deep-History-Connector* be-

zeichnet wird. Während der History-Connector den Aktivierungsstatus der Zustände auf seiner Ebene speichert, speichert der Deep-History-Connector die Zustände auf seiner Ebene und aller Ebenen unter ihm. Bild 2-15 verdeutlicht die Unterschiede der beiden Connectoren.

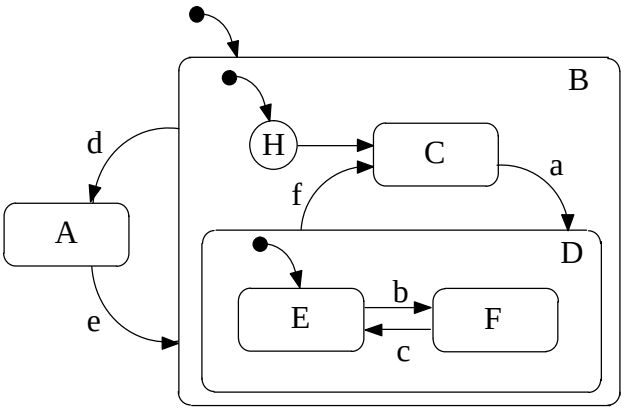


Bild 2-15: Statechart mit History-, bzw. Deep-History-Connector

Über die Default-Transition wird der Oberzustand B aktiviert. Die Default-Transition von B ist mit einem History-Connector verbunden. Beim ersten Eintritt in den Zustand B tritt der History-Connector nicht in Erscheinung (der Zustandsspeicher ist noch leer) und der Zustand C wird betreten. Nach dem Betreten des Zustands B über den History-Connector wird der aktuelle Unterzustand auf der Ebene des History-Connectors gespeichert. Bei späterem Wiedereintritt in den Oberzustand B veranlaßt der History-Connector wegen des Zustandsspeichers den Eintritt in den Unterzustand, der beim letztmaligen Verlassen gespeichert wurde. Im Beispiel können die beiden Zustände C und D im Zustandsspeicher abgelegt werden. Beim Eintritt in den Oberzustand B wird also der Zustand C im Zustandsspeicher abgelegt. Wird das Event a ausgelöst, so geht das System in Zustand E über. Da nur der Unterzustand auf der Ebene des History-Connectors gespeichert wird, befindet sich im Zustandsspeicher der Oberzustand D. Auch nach Auslösen des Events b und dem Übergang des Systems von E nach F verbleibt im Zustandsspeicher der Oberzustand D. Sollte nun ein Event d ausgelöst werden, so wird Zustand F verlassen und es erfolgt ein Sprung auf Zustand A. Bei Rückkehr von Zustand A in Zustand B durch Auslösen von Event e wird der nun gefüllte Zustandsspeicher durchlaufen. Im Zustandsspeicher befindet sich Oberzustand D, der direkt nach Betreten des History-Connectors angesprungen wird. Obwohl sich das System beim Verlassen des Zustands B im Zustand F befand, wird nun die Default-Transition von Oberzustand D betreten und in Zustand E übergegangen.

Bild 2-16: History- und Deep-History-Connector

Wird statt des History-Connectors ein Deep-History-Connector verwendet (Bild 2-16), so können auch die Zustände unterhalb gespeichert werden. In obigem Beispiel würde also ein Rücksprung zum Zustand F erfolgen. Der Deep-History-Connector besitzt das gleiche Aussehen wie der History-Connector, allerdings ist das H mit einem zusätzlichen Stern gekennzeichnet.

Bei der Einführung von Parallelität muß auch das Zusammenwirken und Synchronisieren der parallelen Komponenten gewährleistet werden. Dies wird durch das *Broadcasting*-Prinzip unterstützt, das einen Kommunikationsmechanismus für parallele Systemzustände bietet. Die Kommunikation kann dabei durch spezielle Events, Conditions oder *Joint-Transitions* erfolgen. Eine Joint-Transition wird von mehreren Zuständen gemeinsam (engl. *joint*) benutzt, wenn die Zustände gleichzeitig aktiv sind und mit gleichem Label und Ziel ausgestattet sind. Zum Broadcasting geeignete Events stellen beispielsweise en(S) oder ex(S) dar. Das Event en(S) wird erzeugt, wenn ein Zustand S betreten (engl. *entered*) wird, das Event ex(S) wird erzeugt, wenn ein Zustand S verlassen (engl. *exited*) wird. Die Condition in(S) kann als Abfrage verwendet werden, ob sich das System gerade im Zustand S befindet. Es existieren noch weitere Events und Conditions, mit denen ein Broadcast ausgeführt werden kann, auf die jedoch in diesem Rahmen nicht weiter eingegangen wird.

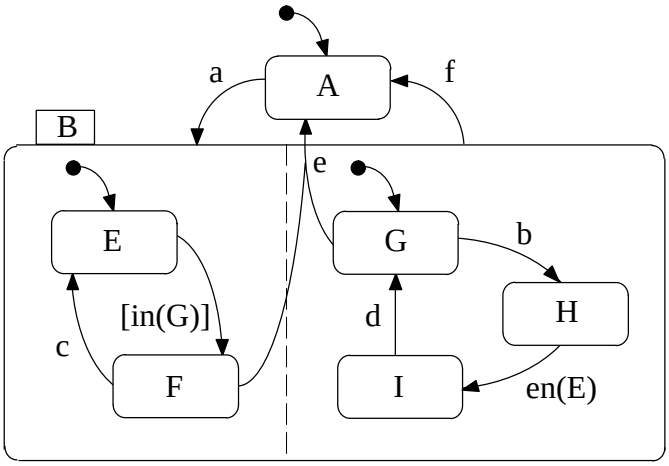


Bild 2-17: Statechart mit Broadcasting-Mechanismus

Bild 2-17 zeigt ein Beispiel für die Verwendung des Broadcasting-Mechanismus. Befindet sich das System in Zustand A, so gelangt man bei Eintreffen des Events a in die beiden parallelen Zustände E und G. Da sich das System in Zustand G befindet, ist die Condition in(G) erfüllt und ein Zustandsübergang von Zustand E nach F wird in der linken Komponente des Statecharts durchgeführt, während die rechte Komponente im Zustand G verbleibt. Durch die Verwendung der Condition in(G) wird der Ablauf des linken Teils durch den rechten Teil beeinflußt und damit ein Broadcasting durchgeführt. Erst nach Empfang des Events b wird ein Zustandsübergang im rechten Teil von Zustand G nach H durchgeführt. Trifft das Event c ein, so wird ein Zustandsübergang von F nach E im linken Teil durchgeführt. Durch Betreten des Zustands E wird das Event en(E) ausgelöst und ein Zustandsübergang von H nach I wird durchgeführt. Man beachte hierbei jedoch, daß das Broadcasting erst im folgenden Verarbeitungsschritt ausgeführt wird, d.h. zuerst geschieht der Übergang von F nach E und erst danach der Übergang von H nach I. Dieser zusätzliche Verarbeitungsschritt kann zu unerwünschten Effekten führen.

Befindet sich das System in den Zuständen F und G und wird ein Event e ausgelöst, so wird die Joint-Transition benutzt, die von Zuständen aus beiden Statechart-Teilen ausgeht und ein Verlassen des Oberzustands B in den Zustand A bewirkt.

Das zeitliche Verhalten von Systemen kann mit Hilfe von Events modelliert werden, auf die mit einer zeitlichen Verzögerung reagiert wird (*timeout*) oder die erst nach Ablauf einer bestimmten Zeit-

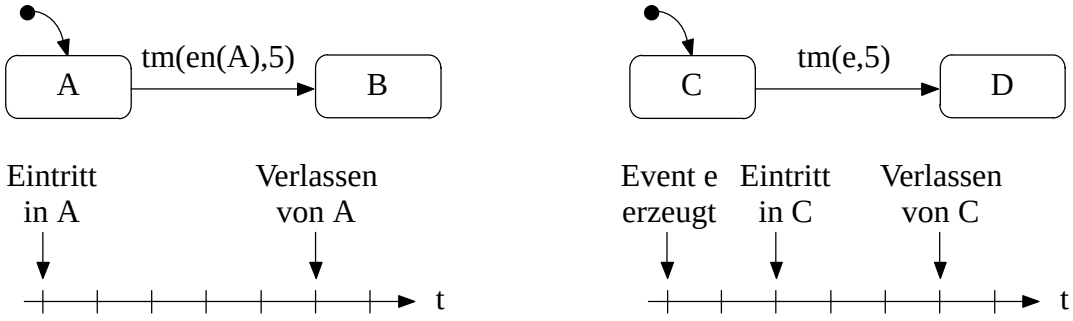


Bild 2-18: Timeout-Event

spanne zur Ausführung gelangen (*scheduled action*). Eine Transition, die das Label timeout(E,N) besitzt, löst N Zeiteinheiten nach dem Auftreten von Ereignis E ein Event aus. Bild 2-18 zeigt zwei Beispiele für die Benutzung von Timeout-Events. Auf der linken Seite wird fünf Verarbeitungsschritte nach Betreten des Zustands A ein Event ausgelöst, das einen Zustandsübergang von A nach B bewirkt. Das Auslösen des Timeout-Events auf der rechten Seite hängt direkt mit dem Erzeugungszeitpunkt des Events e zusammen. Findet der Eintritt in Zustand C beispielsweise zwei Verarbeitungsschritte nach der Erzeugung des Events e statt, so erfolgt bereits drei Verarbeitungsschritte später das Verlassen des Zustands C.

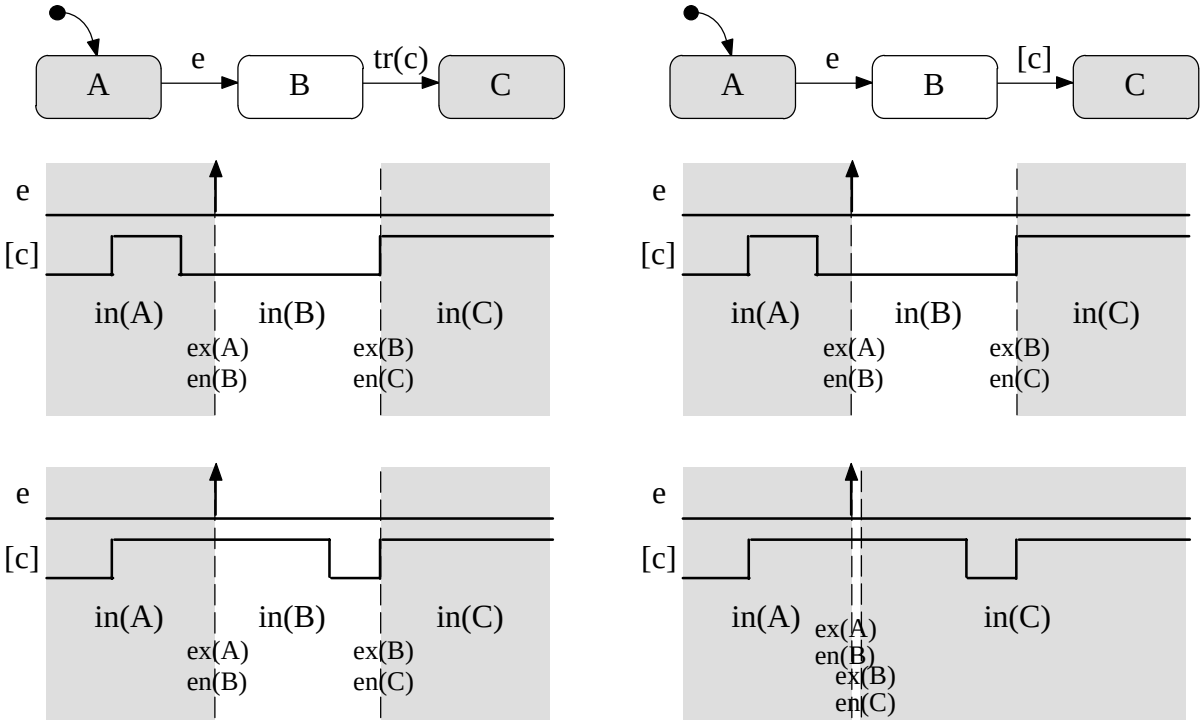


Bild 2-19: Flanken- und pegelsensitive Systeme

Mit Statecharts können sowohl flanken- als auch pegelsensitive Systeme modelliert werden.

Definition 2.12: Ein System heißt *flankensensitiv,* wenn sein Verhalten von einem bestimmten Pegelwechsel mindestens eines Eingabe- oder Zustandssignals abhängig ist.

Definition 2.13: Ein System heißt *pegelsensitiv,* wenn ausschließlich die anliegenden Signalpegel sein Verhalten bestimmen.

Bild 2-19 zeigt auf der linken Seite die Modellierung eines flankensensitiven Systems und auf der rechten Seite die Modellierung eines pegelsensitiven Systems. Das Event tr(c) wird ausgelöst, wenn eine Flanke des Signals c von ’0’ nach ’1’ detektiert wird, während die Condition c ausgelöst wird, wenn der Wert von c ’1’ beträgt. Diese Modellierung kann zu unterschiedlichen Reaktionen führen. Bei dem obigen Verlauf des Events e und der Condition c ist der Wert von c im Moment des Eintretens in Zustand B ’0’. Damit wird bei keiner der beiden Modellierungen ein Zustandsübergang ausgelöst. Bei geändertem Verlauf der Condition c besitzt c im Moment des Eintretens in Zustand B den Wert ’1’. Da auf der linken Seite auf eine Flanke des Signals c von ’0’ nach ’1’ gewartet wird, wird erst nach Wechsel der Condition von ’1’ auf ’0’ und wieder nach ’1’ der Zustandsübergang ausgelöst. Im rechten Fall wird der Wert der Condition c ausgewertet. Da diese den Wert ’1’ besitzt, wird der Übergang nach C durchgeführt. Man beachte jedoch auch hier, daß sich für die Dauer eines Verarbeitungsschritts das System im Zustand B befindet.

Definition 2.14: Unter einem *Verarbeitungsschritt* versteht man die Zeitspanne, die ein System zur Verarbeitung der Eingabesignale und des Systemzustands zur Ermittlung des folgenden Systemzustands benötigt.

Um einen Vergleich zu den endlichen Automaten und den Petri-Netzen herstellen zu können, wurde das Modellierungsbeispiel Fensterheber auch für Statecharts angewendet. Die resultierende Modellierung (Bild 2-20) fällt im Vergleich zu Bild 2-6 wesentlich übersichtlicher aus. Dieser Effekt verstärkt sich noch weiter, je komplexer das Modell ausfällt.

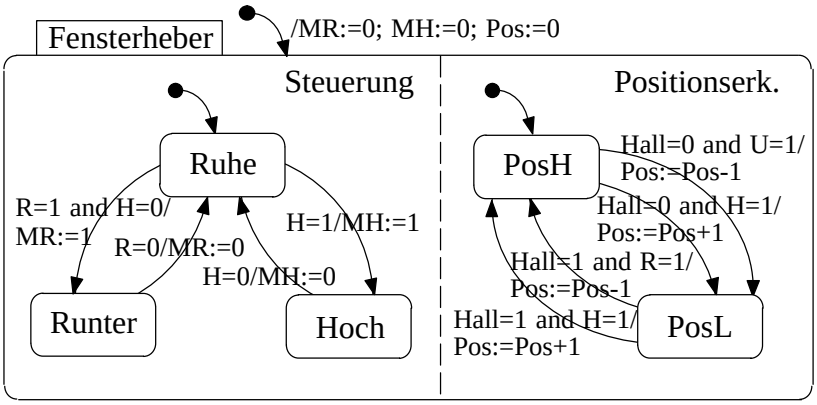


Bild 2-20: Modellierung eines einfachen Fensterhebers mit Statecharts

Auch in diesem Beispiel besteht das Fensterhebersteuergerät aus einem Kippschalter, der die Tasten `H` (für Fahrt hoch) und `R` (für Fahrt runter) besitzt. Ist keine der beiden Tasten gedrückt, so befindet sich der Fensterhebermotor in Ruhe (`MH`=’0’, `MR`=’0’). Ist die Taste `H` betätigt, fährt das Fenster nach oben (`MH`=’1’), bei `R` nach unten (`MR`=’1’). Durch die Auswertung der vom Motor gelieferten Hallsignale, die jede Motorumdrehung mit einer Veränderung des Hallsignals `Hall` von ’1’ nach ’0’, bzw. von ’0’ nach ’1’ quittieren, wird eine Positionserkennung durchgeführt. Die Variable Pos wird bei einen Fahrt nach oben bei jeder Veränderung des Hallsignals inkrementiert, bei einer Fahrt nach unten und einer Veränderung des Hallsignals dekrementiert.

CASE-Werkzeuge zur Beschreibung von diskreten Systemen sind beispielsweise STATEMATE ™ der Firma I-logix oder BetterState ™ der Firma ISI.

2.1.2 Modellierung kontinuierlicher Systeme

Kontinuierliche Systeme bestehen aus Funktionseinheiten zur Verarbeitung und Übertragung von Signalen, wobei die Systemeingänge zusammen mit dem Systemverhalten Auswirkungen auf die Systemausgänge besitzen [Föll92]. Im Unterschied zu diskreten Systemen können die Ein– und Ausgangssignale kontinuierlicher Systeme beliebig viele Zwischenwerte einnehmen.

Daher sind Systeme dieser Art für Aufgaben aus dem Bereich der Regelungstechnik geeignet und können in anschauliche grafische Darstellungen überführt werden. Diese Darstellung wird als Signalflußbild oder Strukturbild bezeichnet. Um kontinuierliche Systeme berechnen zu können, kann das Signalflußbild in eine Zustandsraumdarstellung transformiert werden und mit Methoden der numerischen Integration aufgelöst werden. Bei der numerischen Verarbeitung werden die ursprünglich kontinuierlichen Signale nur noch abgetastet und gehen deswegen in zeit– und wertdiskrete Signale über. Die abgetasteten, quasikontinuierlichen Werte unterliegen dann der Rechengenauigkeit des verwendeten Zahlenformats und der Algorithmen.

Differentialgleichungssysteme

Beschreibt man ein dynamisches System mittels physikalischer Gesetze, erhält man für den Zusammenhang zwischen Ein– und Ausgangsgrößen Differentialgleichungen bzw. Differentialgleichungssysteme. Die Gleichungen stellen dabei ein Modell der im allgemeinen räumlich verteilten und parallelen Vorgänge dar. Es existieren mehrere Modellierungsmethoden, die problemangepaßt zur Anwendung gelangen:

- Modellierungsmethoden mit konzentrierten Parametern, die zu linearen oder nichtlinearen gewöhnlichen Differential– bzw. Differenzengleichungssystemen führen.
- Modelle mit verteilten Parametern, die durch partielle Differentialgleichungen beschrieben werden. Zur Lösung dieser Gleichungen können beispielsweise Finite–Elemente–Methoden eingesetzt werden. Aus der Berechnung dieser Gleichungen können häufig Parameter für gewöhnliche Differentialgleichungen extrahiert werden.
- Nichtparametrische Beschreibungen wie Kennlinien und Kennlinienfelder

Zur Beschreibung linearer, zeitinvarianter, kontinuierlicher Systeme mit konzentrierten Parametern werden Systeme gewöhnlicher Differentialgleichungen verwendet. Als Beispiel für ein solches dynamisches System kann dabei ein Tiefpaß verwendet werden (Bild 2-21).

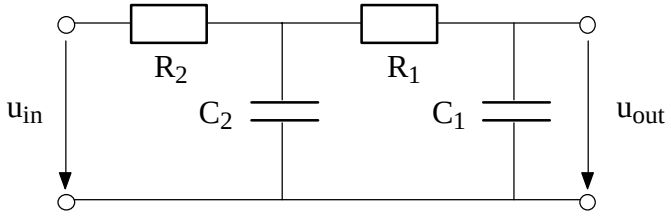


Bild 2-21: Schaltbild eines Tiefpasses

Um das Verhalten dieses Systems zu beschreiben, müssen zunächst die physikalischen Zusammenhänge aufgestellt werden. Für einen Kondensator gilt:

$$\dot{u} = \frac{i}{C} \qquad (2.3)$$

Für einen Widerstand gilt:

$$i = \frac{u}{R} \tag{2.4}$$

An einem Knotenpunkt gilt die erste Kirchhoff‘sche Gleichung:

$$\sum_{n=1}^{N} i_n = 0 \tag{2.5}$$

und in einer Masche die zweite Kirchhoff‘sche Gleichung:

$$\sum_{M} U_M = 0 \tag{2.6}$$

Durch Anwendung dieser Gleichungen läßt sich eine lineare Differentialgleichung zweiter Ordnung erstellen, die die Beziehung zwischen Eingangs– und Ausgangsspannung definiert.

$$\left(C_1 C_2 R_1^2 R_2\right) \ddot{u}_{out} + \left(C_2 R_1 R_2 + 2 C_1 R_1\right) \dot{u}_{out} + u_{out} = u_{in} \tag{2.7}$$

Mit dieser Differentialgleichung kann eine analytische Untersuchung des Systems vorgenommen werden. Da die entstehenden Differentialgleichungen bei komplexeren Systemen eine wesentlich höhere Ordnung aufweisen können, ist diese Darstellung für eine rechnergestützte Verarbeitung nicht geeignet.

Zustandsraumdarstellung

Durch die Transformation einer Differentialgleichung n–ter Ordnung in n Differentialgleichungen erster Ordnung kann die rechnergestützte Verarbeitung erheblich vereinfacht werden. Die Differentialgleichung (2.7) zweiter Ordnung kann in folgendes Differentialgleichungssystem erster Ordnung transformiert werden.

$$\begin{aligned} \dot{u}_1 &= -\frac{1}{R_1 C_1} u_1 + \frac{1}{R_1 C_1} u_2 \\ \dot{u}_2 &= \frac{1}{R_1 C_2} u_1 - \frac{1}{C_2}\left(\frac{1}{R_1} + \frac{1}{R_2}\right) u_2 + \frac{1}{R_2 C_2} u_{in} \\ u_{out} &= u_1 \end{aligned} \tag{2.8}$$

Dabei stehen die Variablen u_1 und u_2 für die Spannung an den Kondensatoren. Dieses Gleichungssystem läßt sich durch Verwendung von Vektoren und Matritzen auf folgende Form bringen:

$$\begin{aligned} \underline{\dot{u}} &= \begin{bmatrix} -\frac{1}{R_1 C_1} & \frac{1}{R_1 C_1} \\ \frac{1}{R_1 C_2} & -\frac{1}{C_2}\left(\frac{1}{R_1} + \frac{1}{R_2}\right) \end{bmatrix} \underline{u} + \begin{bmatrix} 0 \\ \frac{1}{R_2 C_2} \end{bmatrix} u_{in} \\ u_{out} &= \begin{bmatrix} 1 & 0 \end{bmatrix} \underline{u} \end{aligned} \tag{2.9}$$

Diese Darstellung als Vektordifferentialgleichungen entspricht bereits der allgemeinen Form des Zustandsraums, der aus einem Eingangsvektor $\underline{u}$ (im Beispiel: u_{in}), einem Ausgangsvektor $\underline{y}$ (im Beispiel: u_{out}) und einem Zustandsvektor $\underline{x}$ (im Beispiel: u_1 und u_2) besteht. Die Elemente des Zustandsvektor x werden auch als Zustandsvariablen bezeichnet. Wie aus Gleichung (2.9) zu erken-

nen ist, wird die Verknüpfung dieser Variablen durch vier Matritzen beschrieben. Allgemein lassen sich alle linearen, zeitinvarianten, kontinuierlichen Systeme in einer Zustandsraumdarstellung wiedergeben:

$$\begin{aligned} \dot{\underline{x}} &= \underline{A}\,\underline{x} + \underline{B}\,u \\ y &= \underline{C}\,\underline{x} + \underline{D}\,u \end{aligned} \qquad \text{mit} \qquad \underline{x}(t_0) = \underline{x}_0 \qquad (2.10)$$

Die Gesamtheit der Zustandsvariablen kennzeichnet den Zustand des Systems, von dem die Systemausgänge direkt abhängig sind. Es ist dabei für die Ausgabe unerheblich, wie das System in einen bestimmten Zustand gelangt ist. Nur in seltenen Fällen besteht zusätzlich eine direkte Abhängigkeit der Ausgänge von den aktuellen Eingangsgrößen.

Signalflußbilder

Soll unter Verwendung von CASE–Werkzeugen ein kontinuierliches System beschrieben werden, so muß der mathematische Zusammenhang aus den Gleichungen (2.7) und (2.9) modelliert werden. Dies kann entweder textuell oder grafisch in einem Signalflußbild geschehen. Zur grafischen Darstellung muß die Differentialgleichung nach der höchsten Ableitung aufgelöst werden und kann dann wie in Bild 2-22 dargestellt werden.

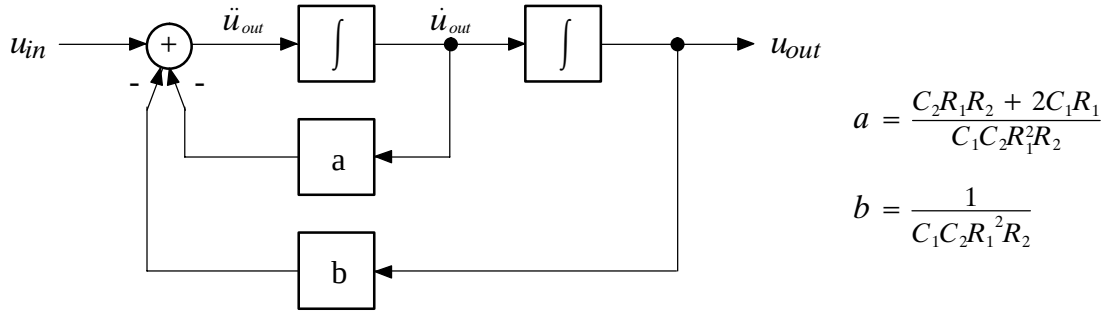


Bild 2-22: Signalflußbild des Tiefpaßbeispiels

Auch ein Zustandsraum kann durch ein Signalflußbild dargestellt werden (Bild 2-23). Da die Struktur dabei immer gleich ist, wird ein Zustandsraum oft als ein einzelnes Symbol dargestellt, dem die Werte der vier Matritzen und der Anfangsbelegung des Zustandsvektors zugeordnet werden.

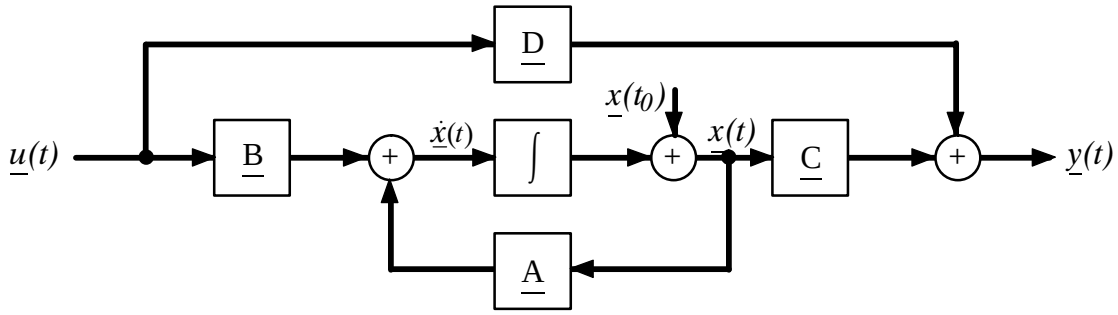


Bild 2-23: Signalflußbild eines Zustandsraums

Um die Darstellung weiter zu vereinfachen, wechselt man beim Übergang zu den Signalflußbildern vom Zeit– in den Frequenzbereich, d.h. man wendet bei kontinuierlichen Systemen die Laplace– oder Fouriertransformation an. Das Verhalten eines Systems läßt sich somit in Form einer Übertragungsfunktion *G(s)* beschreiben.

$$G(s) = \frac{Y(s)}{U(s)} = \frac{b_0 + b_1 s + ... + b_n s^n}{a_0 + a_1 s + ... + a_n s^n} \qquad (2.11)$$

Die Übertragungsfunktion wird durch Bildung des Quotienten von Ausgangssignal und Eingangssignal in Abhängigkeit der komplexen Variablen s aufgestellt. Werden bei einem Systementwurf die Teilsysteme mit Übertragungsfunktionen beschrieben, so kann das Verhalten des gesamten Systems unter Anwendung einfacher Grundrechenarten bestimmt werden. Differentialgleichungen können somit durch Verknüpfung von Integrationsblöcken, Summationsstellen, Multiplikatoren und Rückkopplungen dargestellt werden. Aus dem entstehenden Signalflußbild kann dann die zu Grunde gelegte Funktionalbeziehung rekonstruiert werden. Für ein Systemmodell existieren oftmals mehrere äquivalente Darstellungen. Die Interpretation der Signalflußbilder dagegen ist immer eindeutig. Gleichzeitig eignet sich diese Darstellung zur analytischen Betrachtung eines Systems. Beispielsweise erhält man durch eine Faktorzerlegung der Zähler–, bzw. Nennerpolynome die Pol– und Nullstellen der Übertragungsfunktion, die unmittelbar Aussagen über das Systemverhalten zulassen. Diese Eigenschaften führten dazu, daß sich im Gegensatz zur Modellierung physikalischer Zusammenhänge im Bereich des Reglerentwurfs die Systembeschreibung im Frequenzbereich durchgesetzt hat.

CASE–Werkzeuge wie MATRIX$_X$™ der Firma ISI oder Matlab™ der Firma Mathworks zum Entwurf von Steuerungs– und Regelungssystemen bauen direkt auf der Darstellung der Signalflußbilder auf. Die Modellierung wird mittels eines grafischen Editors erstellt. Für den Entwurf von Regelstrecken kann dabei auf eine Reihe von elementaren Funktionsgliedern zurückgegriffen werden, deren Übertragungsfunktion bekannt ist. In Bild 2-24 sind vier Funktionsglieder mit ihren Übertragungsfunktionen abgebildet.

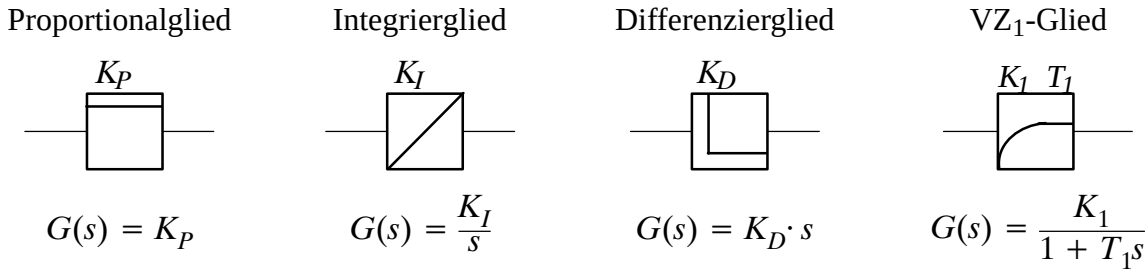


Bild 2-24: Funktionsglieder und ihre Übertragungsfunktionen

Mit den Funktionsgliedern und den Rechenanweisungen nach [Föll85] läßt sich das Verhalten einer Regelstrecke wie in Bild 2-25 folgendermaßen beschreiben:

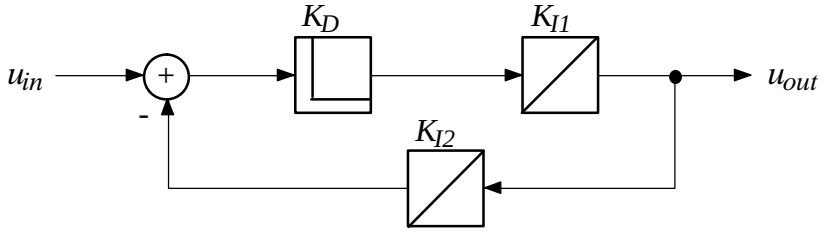


Bild 2-25: Regelstrecke mit Rückkopplung

$$G(s) = \frac{K_D K_{I1} \cdot s}{s + K_D K_{I1} K_{I2}} \qquad (2.12)$$

Anhand der Modellierung einer Gleichstrommaschine [Fabe97] kann gezeigt werden, wie die physikalischen Gegebenheiten in ein Modell umgewandelt werden können (siehe auch Kapitel 9.3). In Bild 2-26 ist das Prinzipschaltbild einer Gleichstrommaschine dargestellt. Bei der Aufstellung des Systems von Differentialgleichungen kann mit dem Feldkreis begonnen werden. Man erhält als Spannungsgleichung:

$$U_f = R_f I_f + \frac{d\phi}{dt} \qquad (2.13)$$

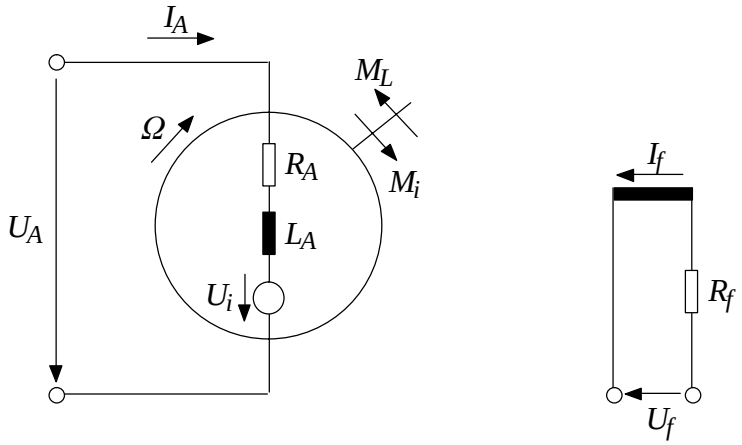


Bild 2-26: Prinzipschaltbild einer Gleichstrommaschine

Hierbei ist ϕ der magnetische Fluß. Wegen der Sättigung des magnetischen Kreises ist ϕ eine nichtlineare Funktion des Feldstroms.

$$\phi = f(I_f) \qquad (2.14)$$

Nach der zweiten Kirchhoff'sche Gleichung (siehe Gleichung (2.6)) ergibt sich die folgende Spannungsgleichung für den Ankerkreis:

$$U_A = U_i + I_A R_A + L_A \frac{dI_A}{dt} \qquad (2.15)$$

R_A stellt dabei den resultierenden ohmschen Widerstand der Ankerwicklung und der übrigen im Ankerkreis liegenden Wicklungen dar. Die Induktivität L_A berücksichtigt die magnetischen Eigenschaften der Ankerwicklung. Rotiert der Anker mit der Winkelgeschwindigkeit Ω im Erregerfeld mit dem magnetischen Fluß ϕ, wird eine Spannung U_i induziert. U_i ergibt sich zu:

$$U_i = c\,\phi\,\Omega \qquad (2.16)$$

Die Konstante c stellt dabei die Maschinenkonstante dar, die das Verhältnis der effektiven Ankerwindungszahl zur Windungszahl der Feldwicklung und einen Faktor, der die Aufteilung des Erregerflusses in Hauptfluß und Streufluß berücksichtigt. Das von der Gleichstrommaschine erzeugte Drehmoment M_i ergibt sich zu:

$$M_i = c\,\phi\,I_A \qquad (2.17)$$

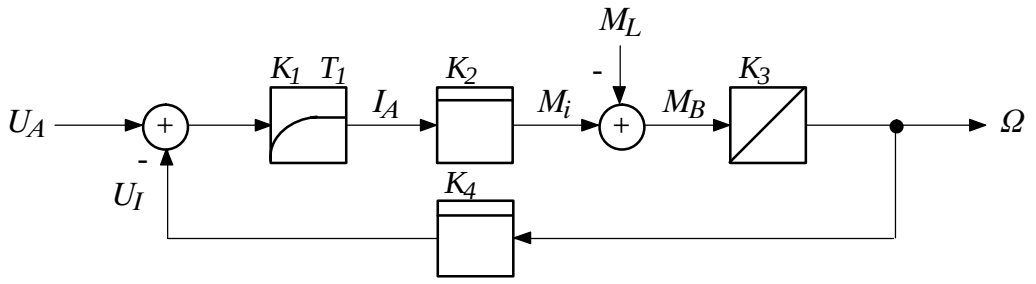
Diesem Drehmoment wirkt von außen das Lastmoment M_L entgegen. Aus der Differenz der beiden Momente ergibt sich das Beschleunigungsmoment M_B:

$$M_B = M_i - M_L \qquad (2.18)$$

Dieses Beschleunigungsmodell bewirkt eine Änderung der Winkelgeschwindigkeit Ω:

$$J \frac{d\Omega}{dt} = M_B \qquad (2.19)$$

Dabei wird das Trägheitsmoment J von allen bewegten Massen (Maschinenanker und angekoppelte Belastungs- und Antriebsmaschinen) gebildet. Da in vielen Fällen die Gleichstrommaschine mit konstanter Erregung betrieben wird, ist der Erregerfluß ϕ eine Konstante und Gleichung (2.14) muß nicht berücksichtigt werden. Die restlichen Gleichungen bilden ein System gekoppelter Differentialgleichungen, die in einem Signalflußbild dargestellt werden können (Bild 2-27).



$$K_1 = \frac{1}{R_A} \qquad T_1 = \frac{L_A}{R_A} \qquad K_2 = c\,\phi \qquad K_3 = \frac{1}{J} \qquad K_4 = c\,\phi$$

Bild 2-27: Signalflußbild einer Gleichstrommaschine

2.1.3 Modellierung von Daten- und Kontrollflüssen

Ein Datenfluß unterscheidet sich grundsätzlich von einem Kontrollfluß. Während Kontrollflüsse die Verarbeitung steuern, beschreiben Datenflüsse den Weg von Daten zwischen einzelnen Prozessen oder Funktionsblöcken.

Definition 2.15: Unter einem Datenfluß versteht man einen informationstragenden Fluß, der zu einem bestimmten Zeitpunkt ein Datum eines beliebigen Datentyps übertragt [Bort94].

Deswegen sind in reinen Datenflüssen weder Entscheidungen noch Schleifen vorhanden. Außerdem werden keine Aussagen über den Beginn oder die Terminierung von Prozessen vorgenommen.

Definition 2.16: Kontrollflüsse sind als nicht informationstragende Flüsse definiert, die den Zustand einer logischen Variablen repräsentieren [Bort94].

Eine verbreitete Form der Modellierung stellen funktionsorientierte Blockdiagramme dar, die den Datenaustausch zwischen einzelnen Blöcken darstellen. Durch die funktionale Aufteilung (engl. *decomposition*) entsteht mit Hilfe dieser Beschreibungsweise eine hierarchisch gegliederte Struktur des Gesamtsystems. Je nach gewünschter Verfeinerung kann die Aufgliederung bis auf die Ebene elementarer Grundeinheiten reichen, die aus logischen Primitivfunktionen wie AND, OR oder auch arithmetischen Grundoperationen wie ADD, SUB bestehen können. Im folgenden werden zwei Formen zur Modellierung von Datenflüssen untersucht, die mit Datenflußdiagramm und Activity-Chart bezeichnet werden. Auf weitere Modellierungen wie N^2-Diagramme [Lano90] wird in diesem Zusammenhang nicht weiter eingegangen.

Datenflußdiagramme

Ein Datenflußdiagramm (engl. *data flow diagram*) besteht aus einem gerichteten, kanten- und knotenmarkiertem Graph, der die Wege von Daten zwischen Funktionen, Speichern und Schnittstellen

beschreibt. Wie dem Namen Datenflußdiagramm zu entnehmen ist, waren in der ursprünglichen Fassung keine Mechanismen zur Darstellung von Kontrollflüssen vorhanden. Spätere Erweiterungen des Datenflußdiagramms [Bort94] wurden um diese Darstellung erweitert. Da verschiedene grafische Repräsentationen für Datenflußdiagramme existieren, wird im folgenden ausschließlich die verbreitetste Darstellung von DeMarco [DeMa79] vorgestellt.

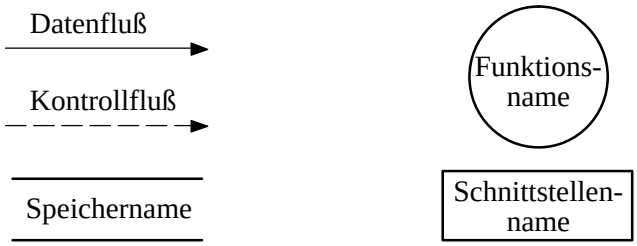


Bild 2-28: Symbole des Datenflußdiagramms nach DeMarco

Ein Datenflußdiagramm besteht aus den in Bild 2-28 abgebildeten vier Symbolen, die die folgende Bedeutung besitzen. Ein Datenfluß wird durch einen Pfeil mit beistehendem Namen dargestellt; eine Funktion oder Prozeß wird durch einen Kreis mit innenstehendem Namen dargestellt; ein Datenspeicher wird durch zwei parallele Linien mit dazwischenstehendem Namen dargestellt und die Schnittstelle zur Umwelt wird durch ein Rechteck mit enthaltenem Schnittstellennamen dargestellt. Speicher stellen dabei Hilfsmittel zur Ablage von Informationen dar. In einen Speicher können Informationen hineinfließen und wieder ausgelesen werden. Ein Kontrollfluß wird durch einen strichlinierten Pfeil mit beistehendem Namen dargestellt.

Mit einem Datenflußdiagramm kann die Struktur eines Systems in immer feinere Teilstrukturen abgebildet werden. Dies ist möglich, da Prozesse einer Ebene wiederum durch Datenflußdiagramme beschrieben und so hierarchisch organisiert werden können. Die Vorteile der Datenflußdiagramme liegen in der leichten Erstellung, der guten Lesbarkeit und in der intuitiven Erfaßbarkeit der Beschreibung auch von wenig erfahrenen Anwendern. Dies ermöglicht eine gute Kommunikation zwischen Entwickler und Auftraggeber. Die formalisierte Beschreibung der Datenflußdiagramme erlaubt auch einfache Konsistenz- und Vollständigkeitsüberprüfungen. Der Nachteil der Datenflußdiagramme liegt in der Schwierigkeit, ein einheitliches Abstraktionsniveau einzuhalten. Außerdem kann nicht von dem Datennamen auf den Datenaufbau oder -struktur geschlossen werden.

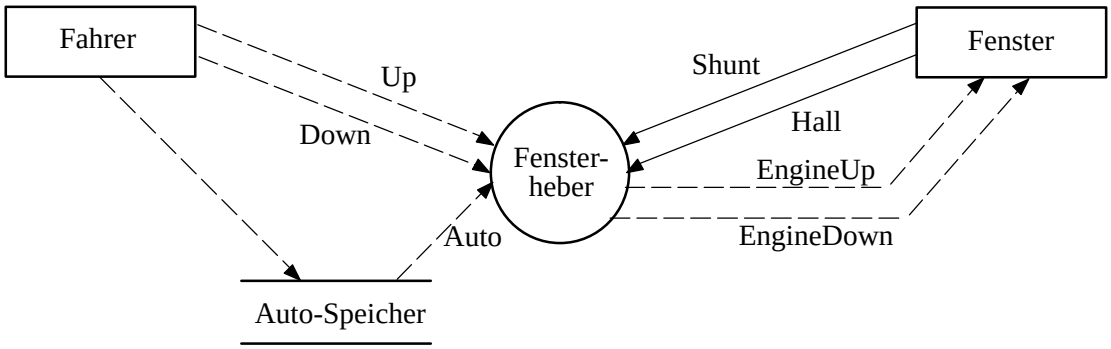


Bild 2-29: Datenflußdiagramm einer Fensterhebersteuerung

In Bild 2-29 ist das Datenflußdiagramm einer Fensterhebersteuerung dargestellt. Der Prozeß ’Fensterheber’ wird von der äußeren Schnittstelle ’Fahrer’ mit den Kontrollflüssen ’Up’, ’Down’ und ’Auto’ versorgt, die die vom Fahrer gewünschte Hoch- oder Herunterfahrfunktion auslösen soll. Der Kontrollfluß ’Auto’ wurde mit einem Datenspeicher versehen, der das automatische Hoch- und

Herunterfahren des Fensters auch ohne Interaktion des Fahrers weiterführt. Der Fensterheber steuert mit den Kontrollflüssen 'EngineUp' und 'EngineDown' die äußere Schnittstelle 'Fenster', die die beiden Datenflüsse 'Shunt' und 'Hall' zurückliefert, die vom Fensterhebersteuergerät als eingehende Datenflüsse zur Bestimmung der Fensterposition und -geschwindigkeit genutzt werden.

Activity-Charts

Activity-Charts [HLNP90] sind in Aufbau und Bedeutung ähnlich zu den Datenflußdiagrammen des vorigen Abschnitts. Sie werden in Verbindung mit der Verhaltensbeschreibung der Statecharts eingesetzt und dienen der Beschreibung der Funktionsstruktur und des Datenflusses, um Verhaltensbeschreibungen funktional besser aufteilen zu können. Ergänzt werden diese beiden Beschreibungsformen durch die Module-Charts, die die physikalische Systemstruktur wiedergeben. Abweichend von den Datenflußdiagrammen fällt die grafische Repräsentation aus. Prozesse (in diesem Zusammenhang *Activities* genannt) werden durch Rechtecke beschrieben. Die zugehörigen Kontroll- oder Steuerprozesse werden durch Rechtecke mit abgerundeten Ecken dargestellt. Die Activity-Charts bieten die Möglichkeit, mittels durchgezogener Linien einen Datenfluß und mittels gestrichelter Linien einen Kontrollfluß zu modellieren. Um Diagramme übersichtlicher zu gestalten, können zusätzliche Connectoren und Transitionen verwendet werden, die denen der Statecharts ähnlich sind. Ein Junction-Connector führt mehrere Daten- oder Kontrollflüsse in einem Knoten (engl. *junction*) zusammen und führt von dort als ein einziger Zustandsübergang weiter zu einem Zielzustand. Mit dem Junction-Connector kann die Übersichtlichkeit der Modellierung verbessert werden. Eine Joint-Transition kann von mehreren Activities gemeinsam (engl. *joint*) benutzt werden, wenn die beteiligten Activities das gleiche Ziel haben. Zusätzlich kann sich ein Datenfluß auftrennen (engl. *fork*) und in unterschiedlichen Activities münden.

Ein Activity-Chart besteht aus einer Activity auf der höchsten Ebene, die die Struktur des modellierten Systems gegenüber der Umgebung abgrenzt. Für das System relevante externe Activities, die nicht vom System selbst beschrieben werden sollen und die Schnittstelle nach außen darstellen, werden durch gestrichelte Rechtecke dargestellt. Diese externen Activities können nicht weiter verfeinert werden und tauschen auch untereinander keine Informationen aus.

Die Activity-Charts unterscheiden zwischen aktiven und passiven Activities. Dabei existieren verschiedene Activityklassen, die je nach Art der Terminierung einer anderen Klasse zugehörig sind:

- *Reactive-Controlled Termination Activities* führen Operationen aus und reagieren auf Ereignisse, solange sie aktiv sind. Sie können von außen gestartet und gestoppt werden, können jedoch nicht selbständig terminieren.
- *Reactive-self Termination Activities* haben das gleiche Verhalten wie Reactive-Controlled Termination Activities, können jedoch darüberhinaus auch von einer Control Activity gesteuert werden, deren Verhalten durch ein externes Statechart beschrieben wird. Dieses Statechart kann einen Termination Connector enthalten, der die Terminierung der Activity bewirkt.
- *Procedure-like Termination Activities* werden aktiviert, führen eine Reihe von Operationen aus und terminieren dann selbständig.

Obwohl eine Modellierung mit Activity-Charts sehr komfortabel möglich ist, sollte man beachten, daß hierbei eine Vermischung von Funktionalität und Verhalten durch die Terminierung von Activities vorgenommen wird.

Bild 2-30: Activity-Chart einer Fensterhebersteuerung

Als Beispiel für eine Datenflußmodellierung mit Activity-Charts dient eine Fensterhebersteuerung, die in Bild 2-30 dargestellt ist. Die Activity Fensterheber, die als Rechteck mit durchgezogener Linie in der Mitte dargestellt ist, kommuniziert mit zwei Activities, die die Umgebung modellieren und als gestrichelte Rechtecke gezeichnet sind. Als Umgebung wird dabei sowohl der Fahrer angesehen, der die Bedienung des Fensterhebers mit den Tasten 'Up', 'Down' und 'Auto' vornimmt, als auch das Fenster selbst, das vom Fensterhebersteuergerät mit den Befehlen 'EngineUp' und 'EngineDown' das Fenster in die gewünschte Richtung bewegt. Diese fünf Signale stellen einen Kontrollfluß dar, der mit strichlinierten Pfeilen repräsentiert ist. Die beiden Datenflüsse 'Shunt' und 'Hall' werden vom Fensterhebersteuergerät als eingehende Datenflüsse zur Bestimmung der Fensterposition und -geschwindigkeit genutzt.

2.2 Systementwurf

2.2.1 Entwurfsphasen

Eine der wichtigsten Aufgaben beim Entwurf elektronischer Systeme stellt die Konsistenz von den Anforderungen an das Produkt und dem Produkt selbst dar. Bei den Anforderungen handelt es sich um qualitative und quantitative Eigenschaften des Produkts aus Sicht des Auftraggebers. Oftmals ist ein Auftraggeber jedoch nicht in der Lage, das gewünschte Produkt vollständig unter Beachtung aller Randbedingungen zu beschreiben.

Definition 2.17: Mit dem Begriff Randbedingung *(*engl. *constraint)* wird eine zwingende Beziehung zwischen bestimmten Größen bezeichnet.

Deswegen wird mit Hilfe einer von Auftraggeber und Entwicklungsteam gemeinsam durchgeführen *Systemanalyse* (engl. *requirements engineering*) die Anforderungen in einem iterativen Prozeß ermittelt. Dies kann bereits mit Hilfe einer formalisierten Beschreibung geschehen, damit die Konsistenz dieser Phase zu den folgenden Phasen gewährleistet bleibt.

In der *Spezifikationsphase* werden Lasten-/Pflichtenhefte formuliert, die das Verhalten, den technischen Aufbau sowie weitere technische und finanzielle Randbedingungen festlegen.

Definition 2.18: Eine *Spezifikation* besteht aus einem Dokument, das in einer kompletten, präzisen und überprüfbaren Art die Anforderungen, Methoden und Werkzeuge des Entwurfs, Verhalten und andere Charakteristika eines Systems oder einer Systemkomponente beschreibt. Oft werden auch die Verfahren zur Feststellung, ob die Anforderungen erfüllt wurden, angegeben.

Das Lastenheft stellt dabei eine Zusammenfassung aller fachlichen Basisanforderungen dar, die das Produkt aus Sicht des Auftraggebers erfüllen muß. Die Basisanforderungen umfassen dabei nur die fundamentalen Eigenschaften des Produkts und eine Beschreibung auf einem hinreichenden Ab-

straktionsniveau. Das Pflichtenheft hingegen enthält eine Zusammenfassung aller fachlichen Anforderungen, die das zu entwickelnde Produkt aus Sicht des Auftragnehmers erfüllen muß und fällt damit meist wesentlich genauer als das Lastenheft aus. Das Pflichtenheft stellt die vertragliche Beschreibung des Lieferumfangs dar.

In der *Systementwurfsphase* setzt sich ein Entwicklerteam mit der Spezifikation auseinander. Das Entwicklerteam nimmt eine Aufteilung des Systems in kleinere Komponenten (Subsysteme) vor und beschreibt somit die Struktur des Systems. In dieser Phase werden bereits die Schnittstellen zwischen den Komponenten festgelegt. Dieses Vorgehen wird auch als funktionale Dekomposition bezeichnet. Aus dieser Beschreibung, die meist unterstützt von Werkzeugen (CASE-Tools) vorgenommen wird, kann eine erste ausführbare Spezifikation generiert werden. Diese ausführbare Spezifikation kann entweder als Animation der Modelle innerhalb eines Werkzeugs, als getrennte Applikation oder als erster Prototyp Verwendung finden. Diese Vorgehensweise erleichtert die Absprache zwischen Entwicklerteam und Auftraggebern und trägt deswegen entscheidend zur konzeptionellen Absicherung bei.

Die in der Systementwurfsphase erstellten Subsysteme werden in der folgenden Phase entwickelt (*Subsystementwurfsphase*). Hierbei wird bereits auf weitere Randbedingungen wie zeitliche Abläufe, Umfang der erstellten Komponenten sowie Kostenkriterien eingegangen. Für die Realisierung der Subsysteme werden Programmiersprachen wie C oder Hardwarebeschreibungssprachen wie VHDL-AMS eingesetzt.

In der *Realisierungsphase* wird das System aus allen Subsystemen zusammengestellt und kann in der realen Systemumgebung getestet werden. Ohne unterstützende Systementwurfstechniken wie Rapid Prototyping (siehe Kap. 2.3) kann erst in dieser Phase eine Beurteilung geschehen, ob das entwickelte System den Wünschen des Auftraggebers entspricht. Dieser relativ späte Zeitpunkt und die hohen Kosten, die bei Nichterfüllung der Anforderungen bei einer Neuentwicklung anfallen, trugen entscheidend dazu bei, daß heutige Systeme bereits in früheren Phasen genauer untersucht werden. Meist wird dazu eine Emulation durchgeführt.

Definition 2.19: Unter *Emulation* versteht man die Ausführung einer Funktion auf einer Hardware, die nicht mit der Ziel-Hardware identisch ist.

Schritt für Schritt wird das realisierte System ausgetestet. Dabei werden zuerst die einzelnen Subsysteme getestet (*Subsystemtestphase*) und danach das gesamte System (*Systemtestphase*). Sind die Tests zur Zufriedenheit des Auftraggebers abgeschlossen worden, wird das System industriell hergestellt und ausgeliefert (*Systemauslieferungsphase*). Eine *Einsatzanalysephase*, in der das hergestellte System im Einsatzgebiet bewertet wird, schließt die Entwicklung ab.

Der Systementwurf wird in der Literatur mittels mehrerer unterschiedlicher Modelle beschrieben. Die dabei verwendeten Modelle unterteilen sich in Entwurfssichten und Lebenszyklusmodelle.

2.2.2 Lebenszyklusmodelle und Entwurfssichten

Um die Aktivitäten und Abläufe während des Entwurfs elektronischer Systeme verstehen zu können, wurden mehrere Lebenszyklusmodelle entwickelt. Diese Modelle sind bereits durch verschiedene Standards normiert worden (z.B. IEEE 830–1993, MIL–STD–498). Das älteste Lebenszyklusmodell ist nach dem Aussehen als Wasserfallmodell benannt und wurde bereits 1970 veröffentlicht [Royc70]. Üblicherweise benutzen alle Lebenszyklusmodelle die Phasen, die in Kap. 2.2.1 be-

schrieben wurden. Auch das in Bild 2-31 dargestellte Wasserfallmodell besteht aus den Phasen Sy-

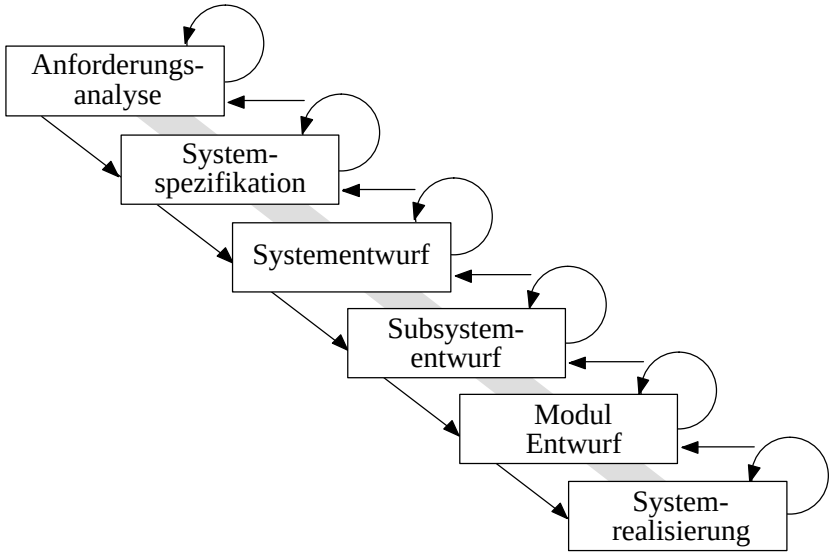


Bild 2-31: Wasserfallmodell

stemanalyse, Spezifikationsphase, Systementwurfsphase, Subsystementwurfsphase und Implementierungsphase. Beim ursprünglichen Wasserfallmodell waren diese Entwurfsphasen abgeschlossene Komponenten ohne Rückführungen und Wiederholungen. Diese sehr idealisierte Betrachtungesweise, die nur Realisierungsfehler betrachtet, wurde einige Jahre später überarbeitet und einem realen Entwicklungsverlauf angepaßt [Boeh76]. Dabei wurden Rückführungen und Wiederholungen in das Modell eingeführt. Trotzdem ist bei dem Wasserfallmodell die späte Überprüfung durch den Systemtest kritisch, da erst zu diesem Zeitpunkt Spezifikationsfehler entdeckt werden können. Im ungünstigsten Fall kann dies eine Neuimplementierung zur Folge haben, die nochmals alle Phasen durchlaufen muß.

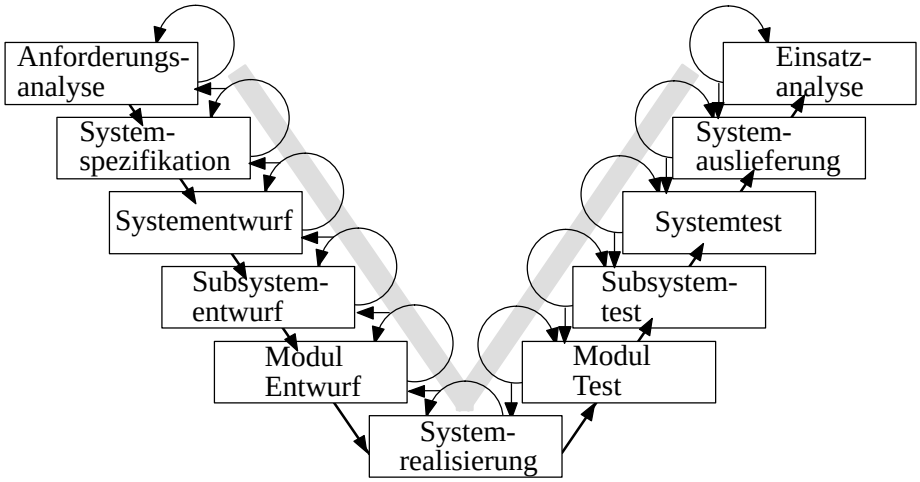


Bild 2-32: V-Modell

Das V-Modell [Calv93] [IABG99] ist eine Weiterentwicklung des Wasserfallmodells. Zusätzlich zu den Phasen des Wasserfallmodells werden nun die korrespondierenden Phasen für Verifikations- und Validierungsaktivitäten eingeführt.

Definition 2.20: Der Begriff *Verifikation* umfaßt zwei mögliche Bedeutungen: 1. Der Evaluierungsprozeß eines Systems oder Systemkomponente, um zu bestimmen, ob die Resultate einer Entwicklungsstufe die auferlegten Bedingungen beim Start dieser Phase erfüllen. 2. Formaler Beweis der Korrektheit eines Programms.

Definition 2.21: Unter *Validierung* wird der Evaluierungsprozeß eines Systems oder Systemkomponente während oder zum Ende eines Entwicklungsprozesses verstanden, um festzustellen, ob die spezifizierten Anforderungen erfüllt wurden.

Dabei basiert das V-Modell auf der Annahme, daß der Spezifikations- und Entwurfsprozeß hauptsächlich durch Zerlegung und Verfeinerung eines top-down-orientierten Entwicklungsprozeß charakterisiert ist. Ergänzt wird dies durch einen auf zunehmender Zusammensetzung beruhenden bottom-up-orientierten Prozeß, der die Implementierungs- und Integrationsphasen charakterisiert. Jede Phase auf der Spezifikations- und Entwurfsseite hat ihre Entsprechung auf der gegenüberliegenden Seite auf den Implementierungs- und Integrationsphasen. Begleitend zu den Implementierungs- und Integrationsphasen erfolgt die Verifikation und Validierung, die die Konsistenz und Vollständigkeit der Entwurfsbeschreibung auf jeder Ebene gewährleistet. Während der horizontale Verlauf des V-Modells die Projektphasen und ihre relative Dauer darstellt, repräsentiert der vertikale Verlauf die unterschiedlichen Abstraktionsebenen.

Man unterscheidet beim Entwurf elektronischer Systeme neben den Entwurfsphasen auch die drei Entwurfssichten Verhalten, Struktur und Geometrie. Anschaulich kann dies durch das *Y–Diagramm* nach Gajski-Walker [WaTh85] dargestellt werden, das sowohl die drei Entwurfssichten als

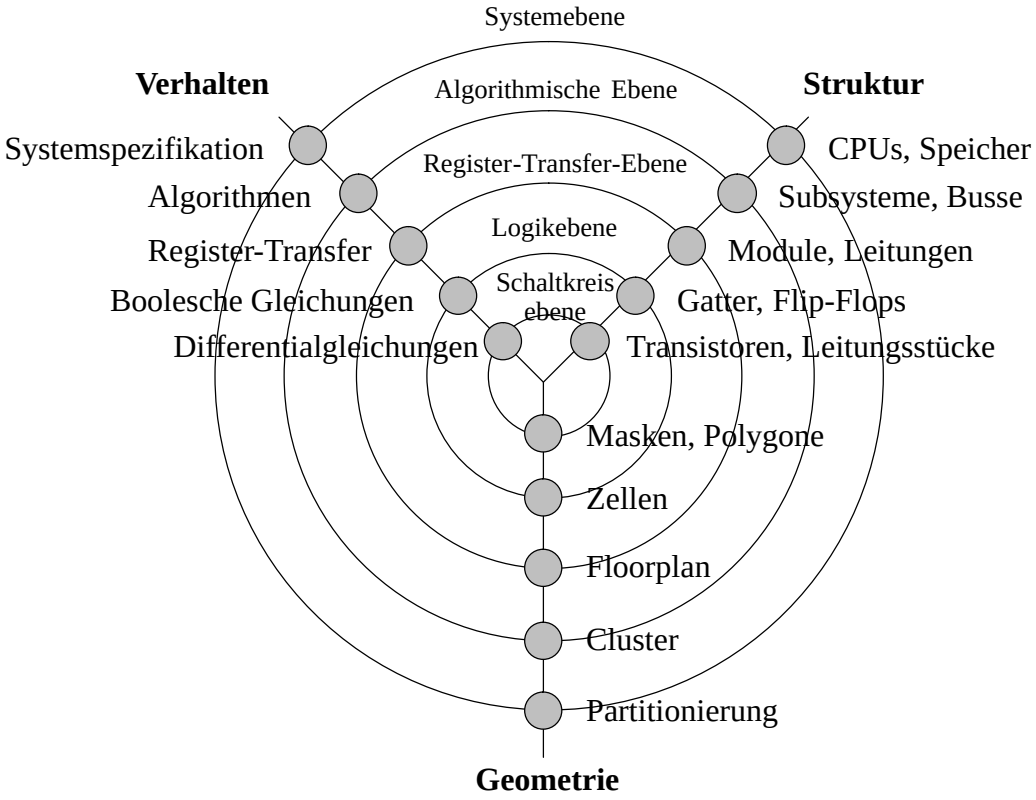


Bild 2-33: Y-Diagramm nach Gajski-Walker für den IC-Entwurf

auch verschiedene Abstraktionsebenen der Entwurfssichten darstellen kann. Während Verhalten,

Struktur und Geometrie durch die Äste des Y–Diagramms repräsentiert werden, stellen die Kreise die unterschiedlichen Abstraktionsebenen dar. Ein großer Kreis bedeutet dabei eine hohe Abstraktion. Der Entwurf elektronischer Systeme kann gemäß des Y–Diagramms als eine Verkettung von Transformationen (Wechsel der Sichtweise auf einem Abstraktionskreis) und Verfeinerungen (Wechsel der Abstraktionsebene innerhalb einer Sichtweise) angesehen werden. Dabei beginnt der Entwurf üblicherweise auf dem Verhaltensast auf Systemebene und wird dann durch mehrere Transformationen und Verfeinerungen bis zum Layout auf dem geometrischen Ast durchgeführt. Eine reine Top–Down Vorgehensweise kann dabei nicht immer eingehalten werden. Fehler können durch Verifikationsschritte zwischen einzelnen Ebenen aufgezeigt werden. In einem Fehlerfall muß der Entwurf von höheren Phasen aus wiederholt werden. Das Y–Diagramm stellt eine technologische Sicht des Entwurfs elektronischer Systeme dar, gibt aber keinen Ablauf des Entwurfs wieder.

Um eine technologische und ablaufgeprägte Sichtweise zu erhalten, kann das Y–Diagramm mit dem V–Diagramm verbunden werden. Das entstehende YV–Diagramm , das im Rahmen dieser Ar-

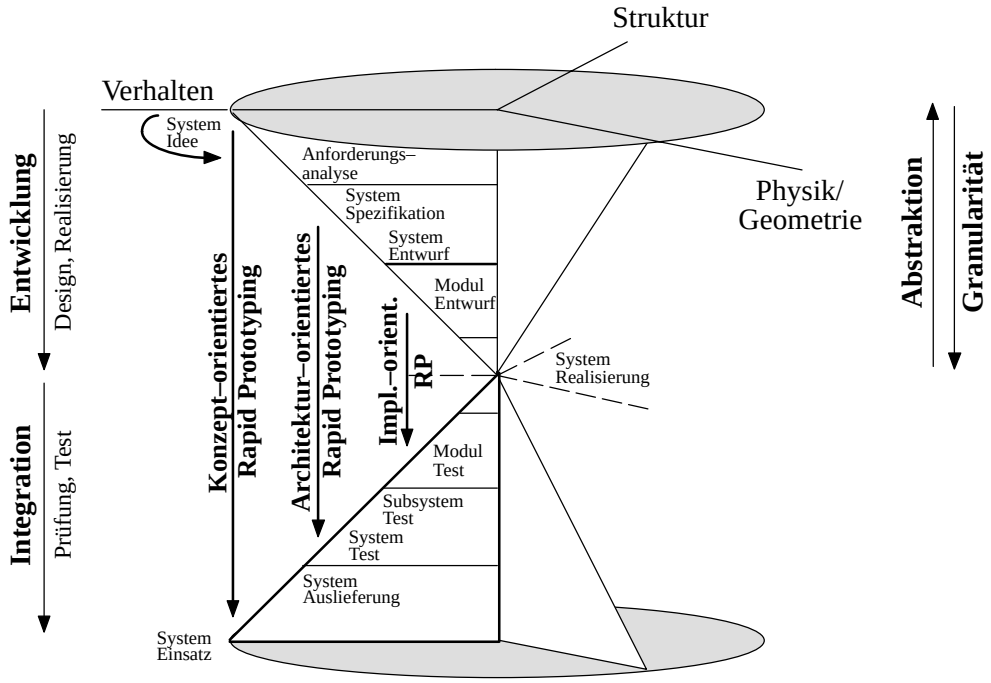


Bild 2-34: YV-Diagramm

beit entwickelt wurde, ist in Bild 2-34 dargestellt. Mit ihm ist es möglich, den technologischen Stand auf den Kreisen der horizontalen Ebene und die Entwurfsphase auf der vertikalen Ebene darzustellen. Ein Entwurf mit dem YV-Diagramm stellt sich als Spirallinie dar, die auf einem Punkt im Bereich des Verhaltensentwurfs beginnt und mit zunehmender Verfeinerung (und somit einer stärkeren technologischen Festlegung) immer mehr durch Struktur- und Geometrieentwurf beschrieben werden kann. In der unteren Hälfte lassen sich die Testphasen wiedergeben und ebenfalls technologisch klassifizieren.

In der Literatur existiert noch eine Vielzahl von weiteren Lebenszyklusmodellen, die entweder stark an das V-Modell angelehnt sind und eigene Erweiterungen vornehmen (z.B. das Spiralmodell [Boeh86], bei dem die einzelnen Phasen nicht mehr starr getrennt sind und nicht zwingend sequen-

tiell verlaufen; das X-Modell, das Wiederverwendungsmechanismen in das V-Modell einführt) oder die für den Entwurf elektronischer Systeme ungeeignet sind (z.B. das Fontänenmodell [HeEd90], das für die objekt-orientierte Systementwicklung eingesetzt werden kann). Im weiteren Verlauf wird auf ergänzende Systementwurfstechniken eingegangen, die in Kapitel 2.3 in das V-Modell eingearbeitet werden.

2.2.3 Ergänzende Systementwurfstechniken

Der Entwurf von elektronischen Systemen bereitet durch steigende Komplexität und wachsenden Anforderungen an heutige Produkte zunehmende Probleme, die nur durch Einsatz geeigneter Systementwurfstechniken gelöst werden können. Durch Verwendung von Lebenszyklusmodellen mit Rückführungen und Wiederholungen konnte man den Entwurf elektronischer Systeme besser verstehen. Es trat eine Verschiebung des Entwicklungsschwerpunkts hin zu den frühen Phasen des Entwurfs ein (Bild 2-35).

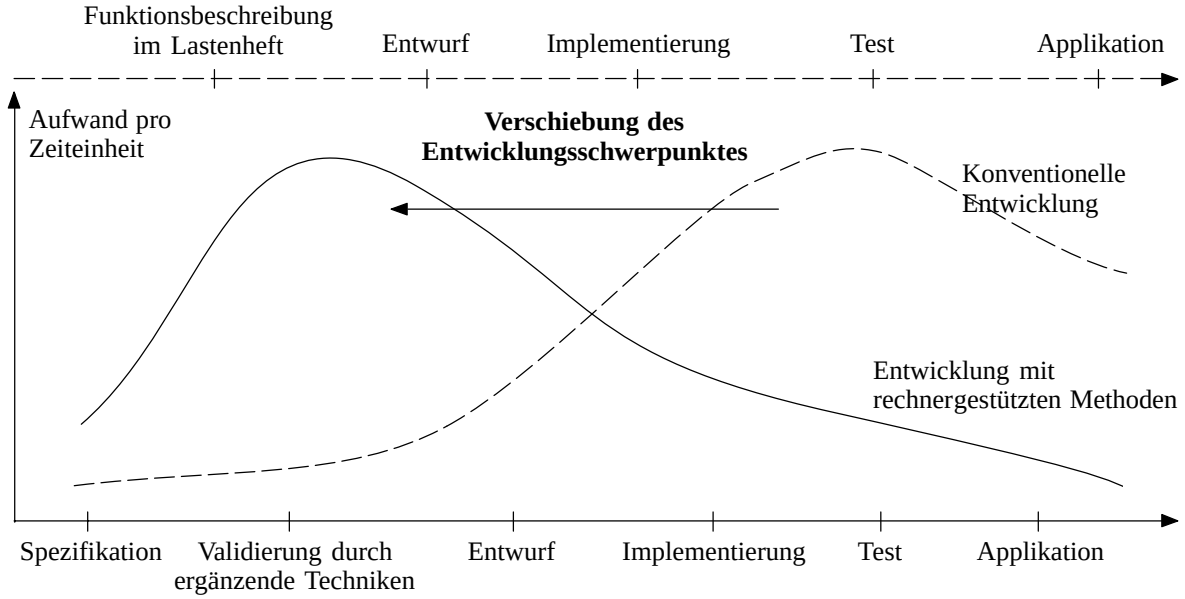


Bild 2-35: Verschiebung des Entwicklungsschwerpunktes

Danach liegt der Hauptaufwand beim Entwurf auf den Phasen Spezifikation und Entwurf. Die Phasen Implementierung, sowie Integration und Test werden nur mit einem geringeren Arbeitsaufwand unterstützt. Sollen in frühen Phasen Aussagen über das endgültige System getroffen werden, so muß die Möglichkeit einer genauen Untersuchung der Spezifikation und des Entwurfs ohne Fertigung des Endprodukts zur Verfügung gestellt werden. Zuerst kann eine einfache Modellanalyse, eine syntaktische Überprüfung und eine Vollständigkeitsanalyse des Modells vorgenommen werden. Seit vielen Jahren ist die Systemsimulation zu einem festen Bestandteil des Systementwurfs geworden.

Definition 2.22: Unter einer *Simulation* versteht man die Ersetzung eines Systems durch ein Modell, das sich wie das System verhält oder arbeitet.

Dabei wird eine Überprüfung der funktionalen Korrektheit des Entwurfsschrittes von der Spezifikation zum abstrakten Verhaltensmodell durchgeführt. Die vorhandenen Werkzeuge unterstützen dies durch Komponenten, die eine Ausführung des Modells erlauben. Bei der Simulation unterscheidet man zwischen *interpretierenden Simulationstechniken* und *compilierenden Simulations-*

techniken. Die interpretierenden Simulationstechniken stellen dabei die klassische Methode dar, bei der ein Interpreter das Modell mit einem Simulator abarbeitet. Kennzeichnend für die interpretierende Simulation sind die kurzen Zeiten bei der Simulationsvorbereitung und die längeren Zeiten bei der Simulation selbst. Bei den compilierenden Simulationstechniken wird das Modell bei der Simulationsvorbereitung in eine Programmiersprache übersetzt, compiliert und als eigenständige Applikation ausgeführt. Dabei muß für die Vorbereitung eine längere Zeitspanne berücksichtigt werden. Die Simulation hingegen kann stark beschleunigt ausgeführt werden. Interpretierende Simulationstechniken eignen sich für eine schnelle Überprüfung des Entwurfs bei der Modellierung oder für kleinere Modelle. Große Modelle und lange Simulationszeiten machen den Einsatz von compilierenden Simulationstechniken erforderlich. Zusätzlich zur Simulation des Verhaltensmodells kann die Systemumgebung modelliert und simuliert werden, damit das entworfene Steuergerät unter Einsatzrandbedingungen getestet werden kann (Bild 2-36).

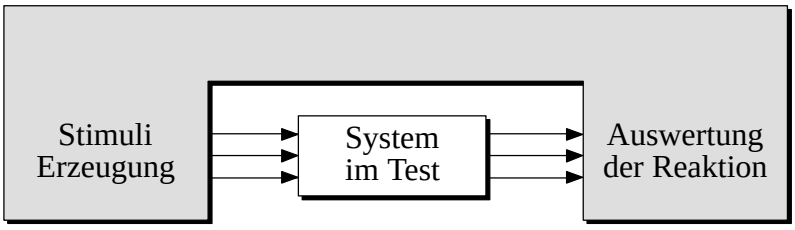


Bild 2-36: Testbenchstrategie

Eine Simulation kann jedoch nur die funktionalen Aspekte des Verhaltensmodells in einer vereinfachten Systemumgebung überprüfen. Die Modellierung einer Systemumgebung ist meist so komplex, daß nur die wesentlichen Aspekte modelliert werden und das Zeitverhalten nur in groben Zügen nachgebildet werden kann. Um eine noch realistischere Überprüfung ohne Einsatz des realen Steuergeräts gewährleisten zu können, wird das simulierte Steuergerät in der realen Umgebung eingesetzt. Dieses Verfahren wird *Rapid Prototyping* genannt.

Definition 2.23: Unter *Rapid Prototyping* versteht man eine Art der Prototypenerzeugung, bei der der Schwerpunkt auf der Entwicklung eines Prototypen früh in dem Entwicklungsprozeß ermöglicht wird.

Um das simulierte Steuergerät in der realen Umgebung ausführen zu können, muß das Modell in eine ausführbare Form gebracht werden und die Ein-/Ausgänge der Modellvariablen in physikalische Signale umgesetzt werden. Die Abarbeitung des Modells muß in Echtzeit erfolgen, um eine zeitlich korrekte Interaktion mit der realen Umgebung zu gewährleisten. Rapid Prototyping kann in mehreren Entwurfsphasen eingesetzt werden. Eine ausführliche Beschreibung zu Rapid Prototyping ist in Kap. 2.3 zu finden.

Liegt ein reales Steuergerät vor, muß es ausführlich getestet werden. Tests in der realen Umgebung können aus verschiedenen Gründen gemieden werden, z.B. können diese Tests kostenintensiv, gefährlich (im Fall eines Airbag-Tests, Crashtests, etc.) oder überhaupt nicht möglich sein, da die physikalischen Einsatzbedingungen zum Zeitpunkt des Steuergerätetests noch nicht existieren (z.B. ist dieser Fall oftmals bei der Entwicklung von Steuergeräten für den Automobilbereich gegeben, wenn das Steuergerät, jedoch keine fertiggestellte Karosserie vorliegt). Um trotzdem reproduzierbare, realistische Tests durchführen zu können, wird das reale Steuergerät in einer simulierten Um-

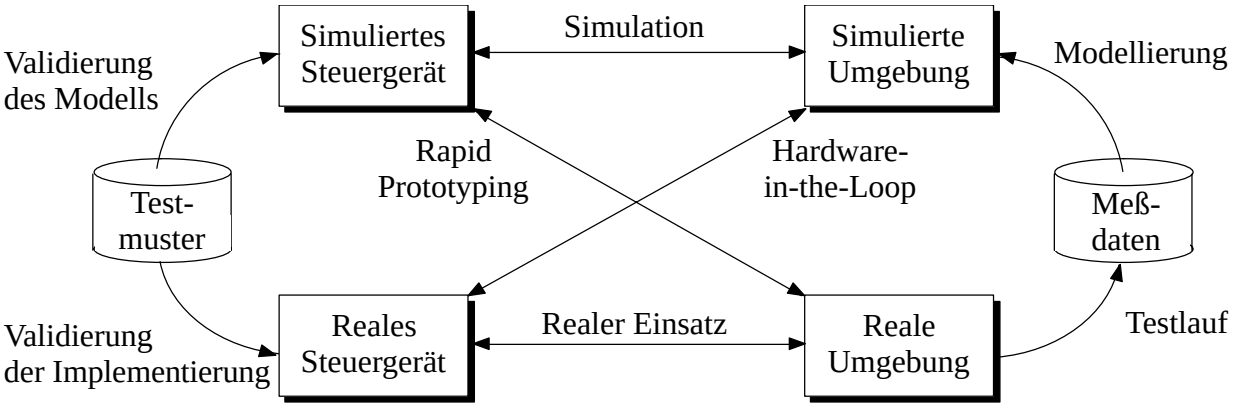


Bild 2-37: Kombinationen bei der Entwicklung elektronischer Systeme

gebung getestet. Dieses Verfahren wird als *Hardware-in-the-Loop* (siehe hierzu Kapitel 2.4) bezeichnet.

Definition 2.24: *Hardware-in-the-Loop* bezeichnet die Ausführung eines entwickelten Systems in einer simulierten Umgebung, um eine Überprüfung auf Konformität mit der Spezifikation durchführen zu können.

Abschließend muß das reale Steuergerät in der realen Umgebung getestet werden. Erst in dieser Phase können endgültige Aussagen über die Fehlerfreiheit des entwickelten Steuergeräts getroffen werden. Die ergänzenden Systementwurfstechniken Rapid Prototyping und Hardware-in-the-Loop dienen der Vermeidung konzeptioneller Fehler und einer möglichst realistischen Untersuchung des Steuergeräts. Dies führt zu einer erhöhten Konzeptsicherheit bei der Entwicklung eines Steuergeräts und vermeidet kostspielige Mehrfachdurchläufe von Entwicklungsphasen.

2.3 Rapid Prototyping

Unter einem Prototypen wird in der Technik im allgemeinen ein Produkt verstanden, dessen Entwicklungsstand zwischen Entwurf und Massenherstellung liegt. Er stellt eine erste funktionsfähige Einheit dar, die getestet werden kann. Um Aussagen über das fertige Produkt ableiten zu können, muß sichergestellt sein, daß der Prototyp ein möglichst genaues Abbild des fertigen Produktes ist. Während eine weitestgehende Verhaltensübereinstimmung zwischen Prototyp und fertigem Produkt erzielt werden soll, gibt es bei der Erstellung Unterschiede. Prototyp und fertiges Produkt können sich in der Beschreibungsart, Programmcodegröße, Ein-/Ausgabehardware und äußerem Design unterscheiden. Prototypen werden zielgerichtet auf Basis festgelegter Eigenschaften in Form eines Anforderungskatalogs für die Ermöglichung bestimmter Untersuchungen entwickelt [Fisc88].

Im Bereich des Echtzeitsystementwurfs wird unter dem Begriff *Rapid Prototyping* eine Art der Prototypenerzeugung verstanden, bei der bereits in den ersten Entwurfsphasen ein Prototyp zur konzeptionellen Überprüfung elektronischer Systeme hergestellt wird. Durch Rapid Prototyping wird ein Einsatz der erstellten Modellierung in der realen Umgebung ermöglicht. Die frühzeitige konzeptionelle Überprüfung führt i.a. zu einer Verkürzung der Entwicklungszeiten bei wachsender Konzeptsicherheit. Die Prototypen können hinsichtlich ihrer Funktionalität und Leistungsfähigkeit untersucht werden und erlauben somit schnelle Rückschlüsse auf die Richtigkeit und Vollständigkeit der Spezifikation. Zusätzlich wird die Entwicklung verschiedener Varianten und der Vergleich

unterschiedlicher Ansätze stark vereinfacht und ein schnelles, iteratives Vorgehen beim Entwicklungsprozeß ermöglicht. Mit Rapid Prototyping findet ein Übergang von der abstrakten Simulation der Systemfunktionalität zu einem Test unter Einbeziehung der realen Umgebung und Zeitanforderungen statt. Rapid Prototyping unterstützt dabei die folgenden Möglichkeiten:

- Automatische Abbildung der Verhaltensspezifikation in ein ausführbares Programm (Generierung eines Software–Prototypen).
- Bereitstellung einer frei konfigurierbaren Hardwareplattform zur Ausführung des Prototypen.
- Bereitstellung von Analyse– und Meßverfahren zur Untersuchung des zeitlichen Verhaltens des generierten Prototypen.
- Bestimmung der Grenze zwischen Soft– und Hardware, d.h. nach Feststellung des zeitlichen Verhaltens können Systemteile, die in Software implementiert waren, in Hardware implementiert werden und umgekehrt (dieses Spezialgebiet wird als Hardware/Software Codesign bezeichnet).

Im Gegensatz zur Simulation der Systemfunktionalität, die in der Regel nicht in der realen Zeit abgearbeitet wird, muß bei einer Einbettung des Prototypen in der realen Umgebung außer der Systemfunktionalität auch die Echtzeitfähigkeit gewährleistet sein. Durch entsprechende Analysemethoden kann das zeitliche Verhalten des Prototyps überprüft werden. Stimmen sowohl Funktionalität als auch zeitliches Verhalten mit der geforderten Spezifikation überein, kann der Entwickler in den nächsten Schritten eine weitere Verfeinerung des Prototypen vornehmen. Können Funktionalität oder zeitliches Verhalten nicht gewährleistet werden, muß ein Rücksprung zur Spezifikationsphase vollzogen werden. Bei Rapid Prototyping entfällt im Gegensatz zur Simulation die Notwendigkeit, die Systemumgebung für vollständige Betrachtungen zu modellieren. Diese Modellierung der Systemumgebung kann unter Umständen (beispielsweise bei der Modellierung des Kaltlaufverhaltens eines Verbrennungsmotors) sehr umfangreich werden und deswegen zu einer vereinfachten Modellierung führen oder in Extremfällen überhaupt nicht erstellbar sein. Insbesondere unvorhergesehene Störeinflüsse der realen Umgebung und Aspekte des Echtzeitverhaltens bleiben in einer Simulation unberücksichtigt.

Durch die iterative Vervollkommnung eines Prototypen, die durch Akzeptanz von Entwickler und Auftraggeber abgeschlossen werden kann, können auch mehrere Arten eines Prototypenaufbaus klassifiziert werden. In der Literatur existieren mehrere Varianten, von denen sich jedoch nur zwei Ansätze durchgesetzt haben (weitere Ansätze sind unter [RBKG00] und [Spre96] zu finden). Der erste Ansatz nach Watkins untergliedert den Prototypenaufbau in fünf Arten [CoHu93]:

- *Exploratory Prototyping:* Diese Form beschreibt einen Prototyp, der in der Spezifikationsphase zu einem Zeitpunkt entwickelt wird, an dem Entwickler und Auftraggeber die Systemziele festlegen können.
- *Solution Prototyping:* Hierbei wird eine Evaluierung von Systemanforderungen vorgenommen, die sowohl für Funktions- als auch für Performanzuntersuchungen verwendet werden können. Dabei werden Animations- und Simulationstechniken angewendet.
- *Investigative Prototyping:* Hierbei wird eine Evaluierung von alternativen Lösungswegen vorgenommen, die sowohl Hardware- als auch Softwareaspekte berücksichtigt. Dabei werden oftmals Simulationstechniken eingesetzt.

- *Verification Prototyping:* Diese Form verwendet formale, mathematische Methoden, um die Korrektheit von Software- und Hardwareentwürfen nachzuweisen.
- *Evaluation Prototyping:* Hierbei werden in der tatsächlichen Systemumgebung funktionierende Prototypen eingesetzt, um Performanz, Benutzerakzeptanz und die Auswirkung von Systemänderungen testen zu können.

Die Klassifikation dieser fünf Arten wurde dabei nach der Zielsetzung des Prototypen vorgenommen. Dieser nicht immer ganz einfach vorzunehmenden Einteilung steht ein zweiter Ansatz gegenüber, der sich am V-Modell orientiert und den Zeitpunkt des Prototypeneinsatzes in den Vordergrund stellt. Dieser zweite Ansatz verwendet drei Phasen und geht auf Ratcliffe [Ratc88] zurück. Sein Vorschlag wurde im Rahmen dieser Arbeit verfeinert und mit dem V-Modell in Verbindung gebracht. Der traditionelle Entwurfsablauf nach dem V–Modell (siehe Kapitel 2.2.2) wird durch den Einsatz von Rapid Prototyping auf mehreren Ebenen der Systementwicklung unterstützt und beschleunigt. Dies kann anhand eines erweiterten V-Modells gezeigt werden, das mit *VP-Modell* (Bild 2-38) bezeichnet wird.

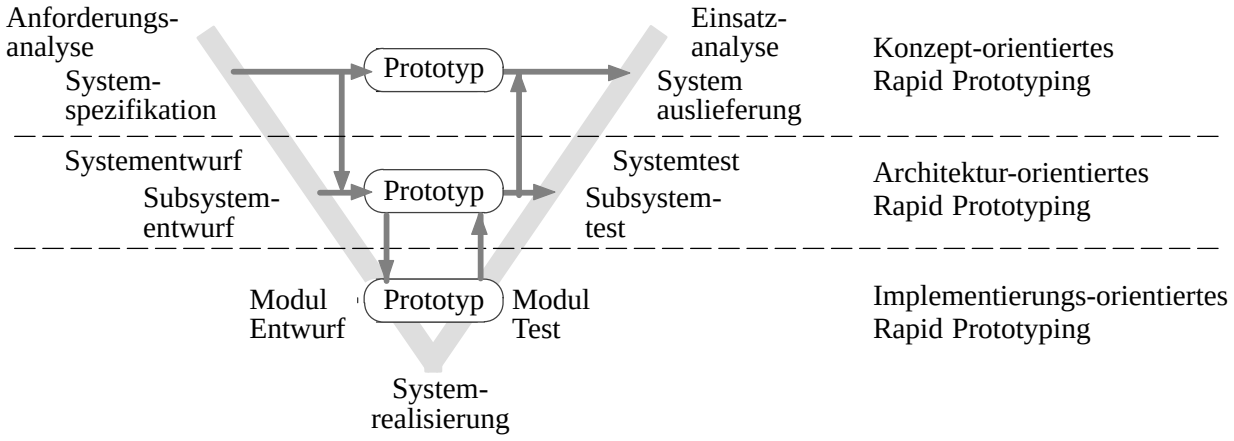


Bild 2-38: VP-Modell

Beim VP-Modell werden drei Phasen des Rapid Prototyping unterschieden: das konzept-orientierte, das architektur-orientierte sowie das implementierungs-orientierte Rapid Prototyping. Da Zielsetzung und zeitliches Auftreten der drei Ansätze unterschiedlich sind, wird jede Form des Rapid Prototypings in bestimmten Phasen des Systementwurfs eingesetzt. Die drei Formen stehen somit nicht in Konkurrenz zueinander, sondern ergänzen sich.

Konzept-orientiertes Rapid Prototyping

Auf der höchsten Ebene des VP–Modells wird das konzept-orientierte Rapid Prototyping eingesetzt und dient hauptsächlich der Klärung der Systemziele. Es repräsentiert dabei die schnelle Umwandlung einer ausführbaren Spezifikation mittels CASE-Werkzeuge wie z.B. $MATRIX_X$™ oder STATEMATE™ in einen funktionellen Prototyp. Die CASE-Werkzeuge können nach einer Prüfung auf Konsistenz und Vollständigkeit aus der Verhaltensbeschreibung des Modells Programmcode (z.B. C, C++, Ada, VHDL-AMS, etc.) generieren. Der konfigurierte und kompilierte Software Prototyp wird danach auf einer geeigneten Zielplattform zur Ausführung gebracht. Die Hardware des Rapid Prototyping Systems besteht aus einem leistungsfähigen Rechner (Zielplattform), der mit geeigneten Schnittstellen zur realen Umgebung ausgestattet ist. Die Schnittstellen müssen dabei einen weiten Signalbereich abdecken können und modular erweiterbar sein. Die Kosten eines kon-

zept-orientierten Rapid Prototyping Systems sind nicht kritisch, da die Hardware für den Aufbau von weiteren Prototypen verwendet werden kann und nicht auf einen Prototyp begrenzt ist.

Beim konzept-orientierten Rapid Prototyping steht ein schneller Systemaufbau im Vordergrund. Stellt man bei der Ausführung des Systemmodells ein Fehlverhalten fest, können in den CASE–Werkzeugen Änderungen der Spezifikation vorgenommen und innerhalb weniger Minuten das geänderte Modell erneut getestet werden. Auch die hardwareseitige Anpassung an die Systemumgebung muß unterstützt werden. Deswegen sollten die Ein-/Ausgabeschnittstellen vom Bildschirm aus konfigurierbar sein und Ein-/Ausgabehardware für die benötigten Signalbereiche vorliegen.

Die Antwortzeiten heutiger Rapid Prototyping Systeme liegt im Bereich weniger Millisekunden und darunter. Damit eignet sich konzept-orientiertes Rapid Prototyping bereits für eine Vielzahl von Anwendungen, beispielsweise für Entwicklung und Test von Innenraumelektronik oder Getriebesteuerungen bei Automobilen. Um weitere Anwendungen miteinbeziehen zu können, die Reaktionszeiten im Bereich von wenigen Mikrosekunden erfordern (beispielsweise Motorsteuerungen) ist eine hohe Codeoptimierung notwendig.

Architektur-orientiertes Rapid Prototyping

Im Gegensatz zum konzept-orientierten Rapid Prototyping wird beim architektur-orientierten Rapid Prototyping bereits die Zielarchitektur des Systems berücksichtigt, um genauere Aussagen über die endgültige Leistungsfähigkeit eines Systems vornehmen zu können. Einige Komponenten des Systems wurden bereits entwickelt oder mittels Standardkomponenten realisiert, während andere Teile sich noch im Test befinden. Dabei können für die Evaluation Prozessor- oder Mikrocontrollerboards eingesetzt werden, die über Schnittstellen mit der realen Umgebung kommunizieren und mit zusätzlicher meist programmierbarer Hardware ausgestattet sind. Die Softwarekomponenten, die auf den Prozessor- oder Mikrocontrollerboards eingesetzt werden, werden bereits auf schnelle Reaktionszeiten und geringen Speicherbedarf hin optimiert, um den Anforderungen des Zielsystems zu entsprechen.

Beim Rapid Prototyping von Prozessoren oder Prozessorkomponenten werden auch mehrfach programmierbare Hardwareplattformen als Basis für Prototypen eingesetzt, wobei meist eine Anordnung von programmierbaren Bausteinen benutzt wird, beispielsweise ein Array von FPGAs (Field Programmable Gate Arrays). Gemäß den Leistungsanforderungen des Systems findet eine erste Partitionierung und Abbildung des Prototyps auf die programmierbaren Bausteine statt. Ein Problem stellt dabei die bislang sehr geringe Ausnutzung der programmierbaren Bausteine dar, die bei etwa 10% liegt [Turn99].

Das Systemmodell wird bei diesen Systemen meistens in Form einer Hardware-Beschreibungssprache erstellt, z.B. VHDL oder Verilog. Ebenso ist die Verwendung von Programmiersprachen (z.B. ADA, C) in Verbindung mit einem Mikrocontroller oder digitalen Signalprozessor möglich. Die Verhaltensbeschreibung kann entweder per Hand oder mittels einer grafischen Spezifikation von CASE-Werkzeugen erzeugt werden. Die Ein-/Ausgabeschnittstellen, die beim konzept-orientierten Rapid Prototyping eingesetzt wurden, können unter Umständen zur Anbindung des Prototypen an die Systemumgebung verwendet werden.

Das architektur-orientierte Rapid Prototyping erreicht nicht die kurzen Änderungszyklen des konzept-orientierten Rapid Prototypings, da die Partitionierung und Synthese des Modells wesentlich mehr Zeit in Anspruch nehmen. Somit ist dieser Ansatz weniger für eine Klärung der Systemziele geeignet, sondern dient der Überprüfung, ob mit der beabsichtigten Zielplattform die Leistungsan-

forderungen erfüllt werden können. Insbesondere bei den programmierbaren Bausteinen kann diese Überprüfung nur eine erste Abschätzung darstellen.

Durch die implementierungs-näheren Reaktionszeiten und die größere Nähe zur endgültigen Realisierung besitzen die architektur-orientierten Prototypen eine höhere Aussagekraft über das zeitliche Verhalten des zu entwickelnden Systems im Vergleich zum konzept-orientierten Rapid Prototyping. Dies wird um den Preis von längeren Änderungszyklen und einer stärkeren Bindung an den zu erstellenden Prototypen erreicht.

Implementierungs-orientiertes Rapid Prototyping

Beim implementierungs-orientierten Rapid Prototyping werden leistungsfähige, hochspezialisierte Prototypen erzeugt, die auf den expliziten Anwendungsfall zugeschnitten sind und meist keine Wiederverwendung erlauben. Prototypen dieser Art dienen der genauen Analyse der Leistungsfähigkeit eines Systems und bestehen oft aus nahe an der endgültigen Systemrealisierung stehenden Prototypen oder bereits vorhandenen Systemen, die um neue Teile erweitert wurden [WeRG95].

Ein Prototypenaufbau auf dieser Ebene ist bereits seit vielen Jahren ein fester Bestandteil des Systementwurfs. Dabei müssen die Systemziele und die Systemstruktur bereits festliegen und Fragen der Realisierung und des Tests im Vordergrund stehen. Der zeitliche Aufwand zur Generierung von implementierungs-orientierten Prototypen ist vergleichsweise hoch. Ein Spezifikationsfehler in dieser Phase verursacht bereits häufig einen erheblichen Änderungsaufwand des Prototypen.

Bewertung

Die Vorteile von Rapid Prototyping liegen in der Steigerung der Konzeptsicherheit und der damit verbundenen Qualitätserhöhung sowie der Verkürzung von Entwicklungszeiten und der Reduzierung von Entwicklungskosten. Dies wird durch schnelle Änderungszyklen erreicht, die zusätzlich die Kreativität des Entwicklers unterstützen. Insbesondere das konzept-orientierte Rapid Prototyping, das erst in den letzten Jahren von Firmen eingesetzt wird, unterstützt diese Ziele. Jedoch ergibt erst ein abgestimmtes Vorgehen unter Verwendung der Prototypen auf den weiteren Ebenen eine optimale Unterstützung des Systementwurfs.

Der Einsatz von Rapid Prototyping Systemen rentiert sich wegen der hohen Kosten für Hard- und Software nicht bei der Entwicklung sehr kleiner, überschaubarer Systeme. Es handelt sich vielmehr um eine Methode zur Verkürzung der Entwicklungszeiten von komplexen Systemen. Rapid Prototyping Systeme können jedoch nicht in allen Umgebungen eingesetzt werden. Beispielsweise bereitet die Kopplung an bewegte Mikrosysteme, deren mechanisches Verhalten durch Anbindung an die Prototypenhardware verändert wird, oder der Einsatz in Systemen mit schwierigen Umgebungsbedingungen (z.B. hohe Luftfeuchtigkeit, hohe mechanische Belastung), starke Probleme. Ein weiteres Problem besteht bei der Entwicklung von heterogenen Systemen, deren Teilsysteme oftmals von unterschiedlichen Werkzeugen beschrieben werden. Die Zusammenführung der Ergebnisse dieser Werkzeuge muß vom Anwender übernommen werden.

Definition 2.25: Ein System wird als *heterogen* bezeichnet, wenn es neben digitalen auch analoge und nichtelektronische Komponenten umfaßt.

Aufgrund der Vorteile, die Rapid Prototyping bei der Entwicklung komplexer Systeme bietet, wird von Unternehmen und Forschungsinstituten nach neuen Ansätzen gesucht, die die Entwicklungszeiten weiter verkürzen und die Planungssicherheit in frühen Entwurfsphasen erhöhen sollen. Gerade das konzept-orientierte Rapid Prototyping, das für die zukünftige Entwicklung elektronischer

Systeme von besonderem Interesse ist, wurde nach Ansicht des Autors noch keiner ausreichenden wissenschaftlichen Untersuchung unterzogen und die Einbettung in den Gesamtsystementwurf sowie die Optimierung des Prozesses selbst ausreichend betrachtet. Aktuell sind u.a. die Bereiche Durchgängigkeit bei Verwendung mehrerer CASE-Werkzeuge, Codeerzeugung für Endprodukte, Unterstützung bei der Fehlersuche sowie stochastische Untersuchungen Gegenstand der Forschung.

Eine Übersicht über die wichtigsten industriellen und universitären Rapid Prototyping Systeme wird in Kapitel 3 vorgestellt.

2.4 Hardware-in-the-Loop

Unter dem Begriff Hardware-in-the-Loop versteht man die Verbindung von realem Steuergerät und einer simulierten Umgebung (siehe Bild 2-37). Das Ziel von Hardware-in-the-Loop besteht in der Überprüfung realer Steuergeräte auf Konformität mit ihrer Spezifikation. Die Überprüfung kann dabei beliebig oft wiederholt werden oder bestimmte Aspekte des Tests, z.B. eine Grenzsituation, herausgegriffen werden, ohne daß Zeitaufwand und Kosten eines realen Tests erreicht werden. Bild 2-39 zeigt die allgemeine Struktur eines Hardware-in-the-Loop Aufbaus. Die Umgebung wird auf einem Echtzeitrechner mittels eines Modells nachgebildet. Die Schnittstelle zwischen dem Umgebungsmodell und dem Steuergerät besteht dabei aus einer Simulation von Sensorik oder Last und einer anschließenden Fehlersimulation. Im Rahmen der Fehlersimulation werden gezielt verschiedene Fehlerarten wie Kurzschlüsse, Masseschlüsse, etc. zwischen Umgebungsmodell und Steuergerät simuliert, um die Reaktion des Steuergeräts im Fehlerfall testen zu können. Die Ansteuerung geschieht dabei über ein Ansteuerungsmodul, das eine Synchronisation von Echtzeit-Rechner und der Fehlersimulation zu Auswertungszwecken herstellt. Das Ansteuerungsmodul sowie Sensor- und Fehler-Simulation kann entweder als reine Software-Lösung oder mit Hilfe einer eigenen Hardware ausgeführt werden.

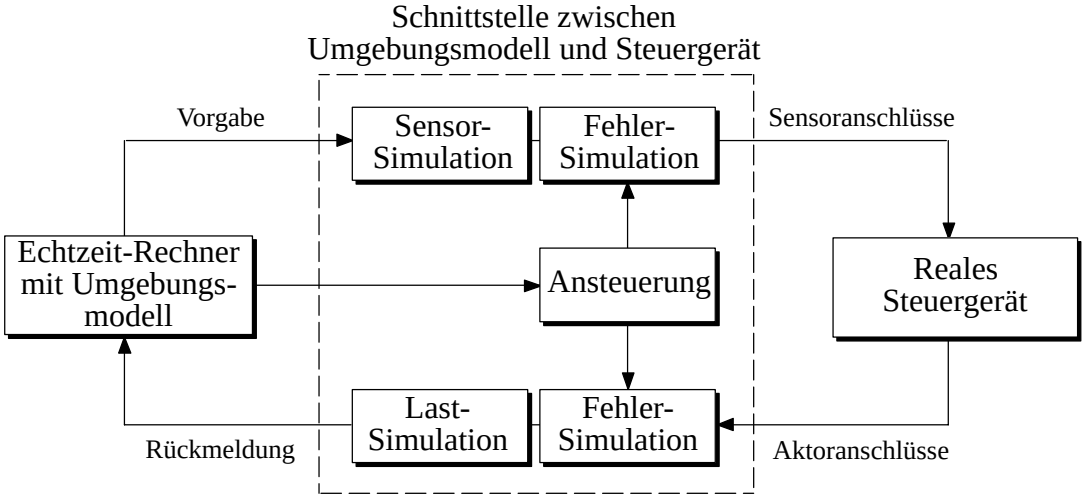


Bild 2-39: Hardware-in-the-Loop Aufbau für elektronische Systeme

In der Automobiltechnik werden Steuergeräte häufig parallel zu den mechanischen Teilen eines Automobils entwickelt. Damit kann es vorkommen, daß Steuergeräte zu einem Zeitpunkt getestet werden müssen, zu dem das Fahrzeug, also die Umgebung des Steuergeräts, noch nicht existiert. Um aussagekräftige Tests bei großen Systemen wie der Gesamtelektronik eines Automobils vornehmen

zu können, können dabei mehrere Steuergeräte gemeinsam getestet werden. Somit kann die Interaktion der Steuergeräte untereinander und ihr Verhalten in einer bestimmten Umgebung getestet werden. Diese erweiterte Form des Steuergerätetests wird auch als Integrationstest bezeichnet.

Der Entwurfsablauf gestaltet sich bezüglich der Werkzeugverwendung ähnlich zu Rapid Prototyping, allerdings wird hier versucht, die Umgebung mittels eines mathematischen Modells nachzubilden. Im allgemeinen werden nur die für das Steuergerät entscheidenden Aspekte der Umgebung modelliert. Oftmals muß auch diese Modellierung noch vereinfacht werden, um die Echtzeitbedingungen für Rapid Prototyping einhalten zu können. Zur einfachen Erstellung der Modellierung existieren spezielle CASE-Werkzeuge, die Mehrkörpersysteme (ADAMS, SIMPACK, etc.), hydraulische Systeme (EASY5, Flowmaster, etc.), chemische Systeme (ASPEN Plus, SpeedUp, etc.) und weitere Systeme modellieren können [Harr98].

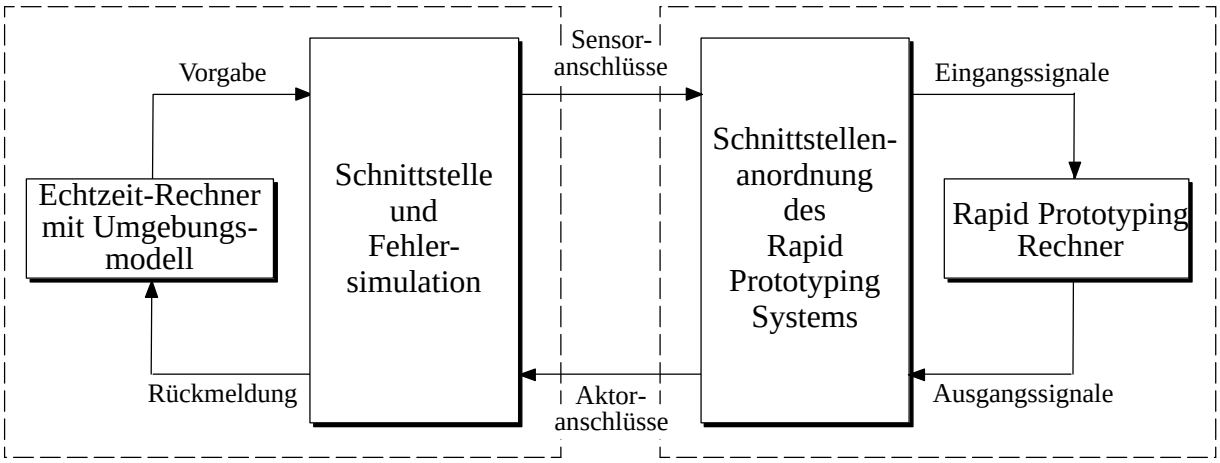


Bild 2-40: Gemeinsamer Einsatz von Hardware-in-the-Loop und Rapid Prototyping

Hardware-in-the-Loop und Rapid Prototyping können auch gemeinsam eingesetzt werden. Dabei werden die Steuergeräte nicht erst nach der kompletten Erstellung, sondern bereits in frühen Entwurfsphasen als Prototyp mit weiteren Steuergeräten innerhalb einer Umgebung eingesetzt und getestet (Bild 2-40). Die daraus gewonnenen Erkenntnisse sind insbesondere für sehr komplexe Systeme wichtig, deren Test in hohem Maß von ihrer Gesamtumgebung beeinflußt wird.

2.5 Echtzeitsysteme

Von elektronischen Systemen wird verlangt, daß strenge Zeitbedingungen eingehalten werden müssen. Charakteristisch für diese Klasse von Systemen ist, daß auf Prozeßstörungen in Echtzeit (engl. *real–time*) reagiert werden kann und sofort korrigierende Maßnahmen eingeleitet werden können. Je nach Anwendungsgebiet unterscheiden sich die geforderten Zeitbedingungen stark und damit existieren verschiedene Interpretationen von dem Begriff “Echtzeit”.

Definition 2.26: Unter dem Begriff Echtzeitbetrieb wird ein Rechnersystem verstanden, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, daß die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind (nach DIN 44300).

Somit ist ein Echtzeitsystem eine Hardware/Softwarekombination, die Daten empfängt, diese verarbeitet und die Ergebnisse innerhalb einer definierten Zeitspanne an den Prozeß weitergibt

[ReLe94]. Diese Definition nimmt jedoch keine quantitative Aussage über die Länge der Zeitspanne vor. Diese wird anwendungsspezifisch für unterschiedliche Bereiche festgelegt und kann von einigen Mikrosekunden bei Spracherkennungssystemen bis zu einigen Sekunden bei Feueralarmsystemen reichen.

Während bei konventionellen Systemen die Richtigkeit einer Berechnung im Vordergrund steht und der Zeitpunkt für die Bereitstellung des Ergebnisses zweitrangig ist, muß im Echtzeitsystem zu einem geforderten Zeitpunkt das Ergebnis zwingend vorliegen. Echtzeitsysteme werden in ihrer Verarbeitung von externen Ereignissen (z.B. Interrupts) unterbrochen. Daraufhin muß die Wichtigkeit des externen Ereignisses bewertet werden und gegebenenfalls direkt in die Verarbeitung eingegriffen werden. Die zur Verarbeitung anfallenden Daten und Ereignisse können dabei zu zyklischen Zeitpunkten auftreten oder einer zufälligen Verteilung unterliegen. Unabhängig von den eingehenden Daten und Ereignissen muß ein Echtzeitsystem die vorgegebene Reaktionszeit auch im schlechtesten anzunehmenden Fall (engl. *worst case*) einhalten.

Neben der Einteilung in anwendungsspezifische Klassen unterscheidet man bei Echtzeitsystemen auch die Folgen der Nichteinhaltung einer Reaktionszeit. Führt die Nichteinhaltung zu schwerwiegenden Störungen oder Systemabstürzen, spricht man von *harten Echtzeitbedingungen*. Beispiele für solche Systeme sind Anti–Blockier–Systeme bei Bremsanlagen von Automobilen oder Systeme zur Schnellabschaltung bei kritischen Zuständen in Kraftwerken. Hat die Nichteinhaltung lediglich eine tolerierbare Rechenungenauigkeit zur Folge, bezeichnet man dies als *weiche Echtzeitbedingung*. Ein Beispiel hierfür sind Klimaanlagen.

Echtzeitbetriebssysteme

Bei Echtzeitsystemen wird häufig von einer parallelen oder verteilten Verarbeitung Gebrauch gemacht. Unter Berücksichtigung der Zeitbedingungen ist die Synchronisation der Tasks eines Echtzeitsystems eines der vordringlichsten Probleme. Zur Unterstützung bei der Programmierung von Echtzeitsystemen gelangen deswegen spezielle Echtzeitbetriebssysteme (engl. *real–time operating systems*) zum Einsatz, die Mechanismen zur Verwaltung der Betriebsmittel zur Verfügung stellen und als Schnittstelle zwischen Systemumgebung und der Anwendung dienen. Betriebsmittel stellen in diesem Zusammenhang alle Mittel dar, die von einer Task benötigt werden, um durchgeführt werden zu können. Dazu zählen Prozessor, Speicher, Peripheriegeräte sowie Programme, Daten, Variablen und Dateien. Beim Echtzeitbetrieb werden die Aufträge direkt in Form von einzelnen Programmen vergeben, die abgearbeitet werden müssen. Um den zeitlichen Anforderungen zu entsprechen, besitzt das Echtzeitbetriebssystem eine Tasksteuerung (engl. *task control*), die die einzelnen Tasks verarbeiten, unterbrechen oder beenden kann. Die Ablaufplanung dieser Tasksteuerung bestimmt die Zuteilungsstrategie, nach der die Prozessorleistung an Tasks verteilt werden. Dieser Vorgang wird als *Task–Scheduling* bezeichnet. Die Ablaufplanung muß auch im worst case Fall die korrekte Verarbeitung gewährleisten. Ist ein System jedoch mit der Abarbeitung von Tasks überlastet, so daß keine korrekte Verarbeitung mehr gewährleistet ist, muß das elektronische System bei der Erkennung von Zeitverletzungen in einen sicheren Zustand übergehen. Es sollte bereits bei der Auslegung des Systems beachtet werden, daß eine Zeitverletzung ausgeschlossen werden kann. Um dies zu gewährleisten, muß man die Machbarkeit des Schedulings (engl. *schedulability*) nachweisen.

Echtzeitbetriebssysteme sind in der Regel modular aufgebaut und sind bezüglich ihrer Funktionalität skalierbar (Bild 2-41). Dies gilt insbesondere für die Klasse der eingebetteten Systeme, denen

oftmals nur wenig Speicher und eine geringe Prozessorleistung zur Verfügung stehen. Neben den stets vorhandenen Grundfunktionen lassen sich bei Bedarf weitere Funktionalität z.B. zur Datei– oder Netzwerkverwaltung einbinden. Die spezifische Anpassung eines Echtzeitbetriebssystems an eine Hardwareplattform geschieht über ein *Board Support Package*, das dem Betriebssystem das Ansprechen der vorhandenen Hardware (z.B. Timer, Speicher, Ein–/Ausgabekanäle) ermöglicht.

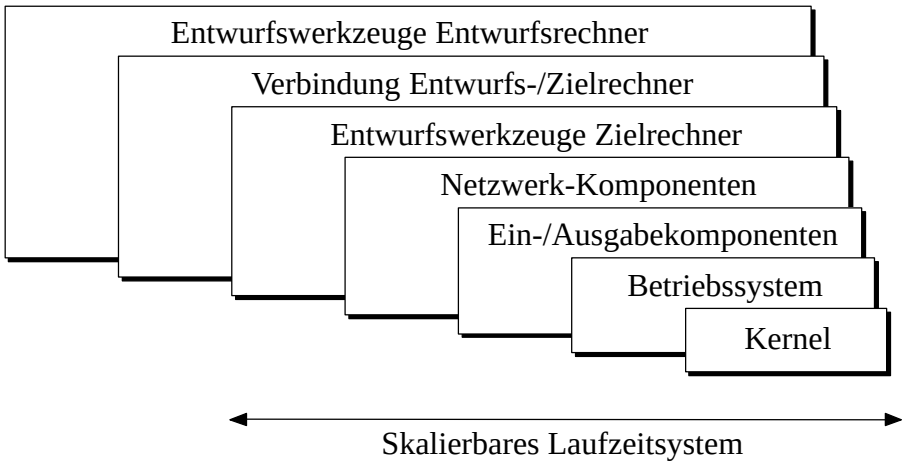


Bild 2-41: Skalierbares Echtzeitbetriebssystem VxWorks™

Der Kern (engl. *kernel*) eines Echtzeitbetriebssystems besteht aus den elementaren Systemfunktionen zur Pflege des Systemtakts sowie der Synchronisation und Verwaltung von Tasks. Er unterscheidet sich insbesondere bei der Unterbrechbarkeit zu konventionellen Betriebssystemen. Echtzeitbetriebssysteme sind voll unterbrechbar (engl. *full–preemptive*) und besitzen damit die Möglichkeit, einen Interrupt direkt bei dessen Auftreten zu bearbeiten. Betriebssysteme mit Unterbrechungspunkten (z.B. Microsoft Windows) bearbeiten einen Interrupt nach dem Erreichen des nächsten Unterbrechungspunktes (engl. *preemption points*). Bei einem nicht unterbrechbaren Kernel (engl. *non–preemptive*) kann erst nach Beendigung eines Systemaufrufs auf einen Interrupt reagiert werden.

Bei der Erstellung von Echtzeitsystemen wird häufig versucht, unabhängige Aufgaben mit verschiedenen Tasks zu realisieren und die zeitlich korrekte Abarbeitung dem Echtzeitbetriebssystem zu überlassen. Um bestimmen zu können, welche Task als nächste ausgeführt werden soll, bzw. welche Tasks überhaupt ausgeführt werden können, teilt ein Echtzeitbetriebssystem Tasks in unterschiedliche Klassen ein. Insgesamt existieren sechs solcher Klassen [ReLe94]:

- Existent
- Nicht existent
- Eine Task ist bereit (engl. *ready*), wenn lauffähig ist, d.h. sie hat alle Betriebsmittel außer dem Prozessor zugeteilt bekommen.
- Eine Task ist laufend (engl. *running*), wenn sie in diesem Augenblick ausgeführt wird und damit den Prozessor benutzt.

- Eine Task ist blockiert (engl. *blocked*), wenn sie nicht ausgeführt werden kann, weil sie auf die Zuteilung von Betriebsmitteln oder ein Unterbrechungssignal wartet.
- Eine Task ist beendet (engl. *terminated*), wenn sie alle ihre Anweisungen abgearbeitet und die ihr zugeteilten Betriebsmittel zurückgegeben hat.

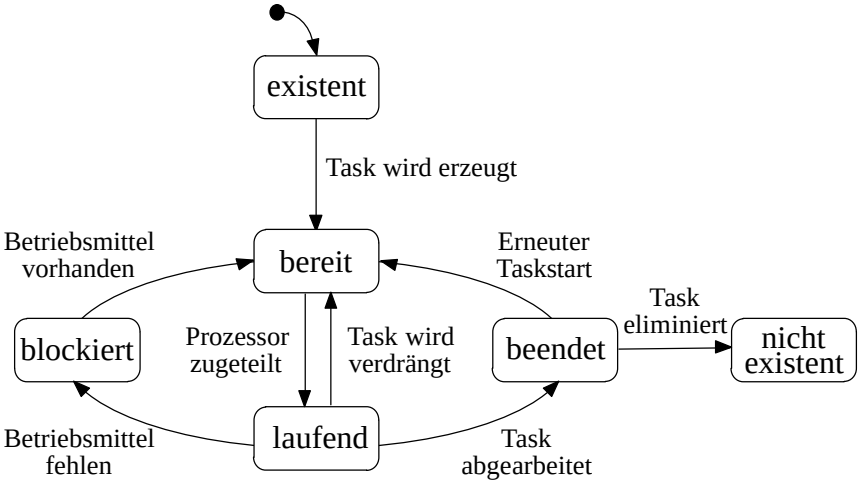


Bild 2-42: Mögliche Taskzustände und Zustandsübergänge

In Bild 2-42 ist dargestellt, welche Übergänge zwischen den sechs Zuständen möglich sind. Eine Task darf sich dabei nur in einem der Zustände aufhalten. Die Zustandsübergänge werden vom sogenannten *Dispatcher* durchgeführt.

Task-Scheduling

Der Scheduler eines Echtzeitbetriebssystems bestimmt, wann welche Task ausgeführt wird. Hierbei stehen mehrere unterschiedliche Verfahren zur Verfügung:

- Beim *Round–Robin Scheduling* werden die Tasks zyklisch geordnet und erhalten den Prozessor für eine definierte Dauer (einem sogenannten *Time–Slice*) zugeteilt. Nach Ablauf dieser Dauer wird die Task in jedem Fall unterbrochen und die nächste Task wird aktiviert. Dieses Verfahren wird hauptsächlich dann eingesetzt, wenn alle Tasks gleich wichtig sind und die Prozeßlaufzeiten abgeschätzt werden können.
- Beim *prioritätsbasierten Scheduling* erhält jede Task eine bestimmte Priorität zugewiesen. Der Scheduler führt dann die Task mit der höchsten Priorität aus. Wird eine sich gerade in der Ausführung befindende Task zugunsten einer freigeschalteten Task mit höherer Priorität unterbrochen, bevor sie ihre eigene Abarbeitung beendet hat, spricht man von preemptiven Scheduling. Prioritätsbasiertes Scheduling eignet sich für Echtzeitbetriebssysteme.
- Das *Hybrid–Scheduling* stellt eine Verbindung der beiden oben genannten Scheduling–Mechanismen dar. Dabei werden Tasks, die die gleiche Priorität besitzen, per Round–Robin Scheduling abgearbeitet.

Trotz der prioritätsbasierten Scheduling–Verfahren kann es zu unerwünschten Effekten bei der Abarbeitung von Tasks kommen. Dies ist in Bild 2-43 dargestellt. Greift eine Task C mit niedriger Priorität auf eine Ressource (beispielsweise über eine Semaphore, siehe dazu nächstes Unterkapitel) zu, während zur gleichen Zeit Task A, die eine höhere Priorität besitzt, die Ressource benötigt, muß Task A warten, bis die Ressource von Task C freigegeben wird. Dieser Vorgang ist beabsichtigt und dient der Datenkonsistenz. Wird zum Zeitpunkt t_2 allerdings Task B mit einer mittleren Priorität

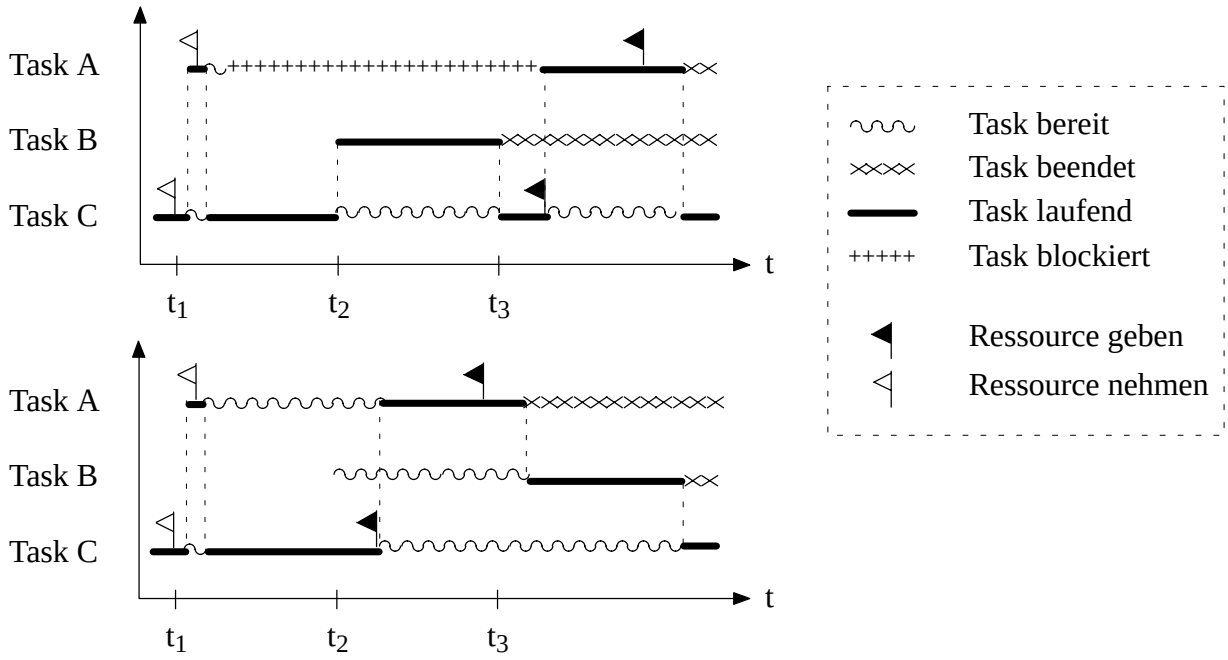


Bild 2-43: Prioritätsumkehr und Prioritätsvererbung

erzeugt, so wird sie Task C unterbrechen, da sie mit einer höheren Priorität ausgestattet ist. Damit ergibt sich der unerwünschte Effekt, daß die mit höchster Priorität versehene Task A sowohl auf Task B als auch auf Task C warten muß. Dieser Effekt wird als *Prioritätsumkehr* oder *Prioritätsinversion* bezeichnet.

Um dieses Problem zu umgehen, vererbt Task A seine Priorität an Task C. Dabei wird bei der Beanspruchung einer blockierten Ressource die Priorität der höher priorisierten Task an die niedriger priorisierte Task weitergereicht. Dieses Verfahren führt unter Beachtung der Prioritäten weiterer Tasks zu der schnellstmöglichen Freigabe der Ressource. Task B ist in diesem Fall nicht mehr in der Lage, die Abarbeitung von Task C zu unterbrechen. Man spricht in diesem Fall von *Prioritätsvererbung*. Bild 2-43 unten zeigt die Abarbeitung der drei Tasks nach Einführung der Prioritätsvererbung.

Interprozeßkommunikation

Die Bereitstellung von Mechanismen zur Interprozeßkommunikation zählt zu den wichtigsten Aufgaben eines Echtzeitbetriebssystems. Unter Interprozeßkommunikation versteht man die Möglichkeit, Informationen zwischen verschiedenen Tasks auszutauschen. Da einzelne Tasks im allgemeinen keine selbständigen Einheiten darstellen, sondern auf Datenaustausch mit anderen Tasks angewiesen sind, werden Kommunikationsmittel benötigt, die den Datenaustausch und die Synchronisation der Tasks übernehmen. Kommunikationsmechanismen, die für diese Aufgaben verwendet werden, sind im folgenden aufgeführt.

- *Semaphoren* zeigen die Verfügbarkeit von Ressourcen an und können auf nahezu alle Betriebsmittel angewendet werden. Häufig werden Semaphoren auf speziell für diesen Zweck definierte Variablen angewendet und dienen der Synchronisation von Tasks. Solange eine Task eine Semaphore besitzt, muß jede andere Task warten, die auf dieselbe Semaphore zugreifen will, bis die Semaphore wieder freigegeben wird. Die wartende Task geht in den Zustand ”blockiert” über.

Erst nach Freigabe der Semaphore kann die wartende Task auf die Ressource zugreifen und kann weiter verarbeitet werden.

- *Message Queues* dienen der Übermittlung von Daten beliebiger Länge.
- Ein *Shared Memory* stellt einen Bereich dar, der von mehreren Tasks zur gleichen Zeit benutzt werden kann. Beim Datenaustausch über Shared Memory besitzen die Tasks einen Zeiger auf einen global definierten Speicherplatz, der sowohl lesend als auch schreibend benutzt werden kann. Normalerweise wird vor einem Lese– oder Schreibvorgang eine Semaphore auf diesen Bereich gesetzt, um eine definierte Benutzung zu gewährleisten.
- *Asynchrone Signale* entsprechen einem Software–Interrupt und können zwischen Tasks das Auftreten von Ereignissen signalisieren.

Entwicklung eines Echtzeitsystems

Um ein Echtzeitsystem möglichst komfortabel entwickeln zu können, bietet sich die Benutzung einer Cross–Entwicklungs–Umgebung (z.B. Tornado™ von Wind River Systems) an. Dabei geschieht die Entwicklung des Echtzeitsystems auf einem eigenen Entwicklungssystem, das oftmals nicht dem Zielsystem entspricht. Das Entwicklungssystem wird Host–Rechner genannt, das Zielsystem wird als Target–System bezeichnet. Auf dem Entwicklungssystem stehen eine Reihe von Entwicklungswerkzeugen zur Verfügung, die mittels eines Simulators die komplette Funktionalität des auf dem Zielrechner benutzten Echtzeitbetriebssystems nachbilden. Die so entwickelte Applikation kann zu jedem Zeitpunkt über eine serielle Schnittstelle oder über eine Netzanbindung auf dem realen Zielsystem abgearbeitet werden.

Da Echtzeitbetriebssysteme auf unterschiedlichen Plattformen vom gleichen Programmcode ausgehen, ist auch eine Portierung auf andere Zielsysteme mit geringem Aufwand verbunden. Dies kann dazu genutzt werden, spezielle Zielsysteme in der Testphase zu benutzen, die beispielsweise mit zusätzlicher Hardware hochgenaue zeitliche Messungen vornehmen können. Das endgültige Zielsystem kann mit einem wesentlich geringeren Hardwareaufwand ausgestattet sein.

3 Stand der Technik

Im folgenden Abschnitt werden die wichtigsten industriellen und universitären Rapid Prototyping Ansätze aufgeführt. Dabei wird sowohl auf FPGA-basierte Systeme als auch auf DSP-, Mikrocontroller- oder Mikroprozessor-basierte Systeme eingegangen.

3.1 EASY5

Das *Engineering Analysis System Version 5* (*EASY5*™) ist ein Softwarepaket zur Modellierung und Simulation dynamischer Systeme und wird von der Boeing Corporation zur Entwicklung von zivilen Passagierflugzeugen eingesetzt. Seinen Ursprung hat *EASY5*™ bereits 1976, als eine Werkzeugkopplung zwischen linearen und nichtlinearen Modellierungswerkzeugen eines Boeing-Mitarbeiters von Boeing übernommen und neu überarbeitet wurde. Von 1980 an wurde *EASY5*™ von den Boeing Computer Services kommerziell vertrieben und wird seitdem kontinuierlich erweitert. Die entwickelten Systeme können digitale und analoge, hydraulische, pneumatische, mechanische und thermische Subsysteme beinhalten. Systeme werden im Entwurfssystem *EASY5*™ aus funktionalen Blöcken (Summierer, Multiplikator, Signalgeneratoren, Integratorblöcken, etc.), vordefinierten Komponenten (Motoren, Getriebe, Pumpen) und Softwarefragmenten zusammengesetzt. Bild 3-1 zeigt einen Screenshot der Oberfläche von *EASY5*™.

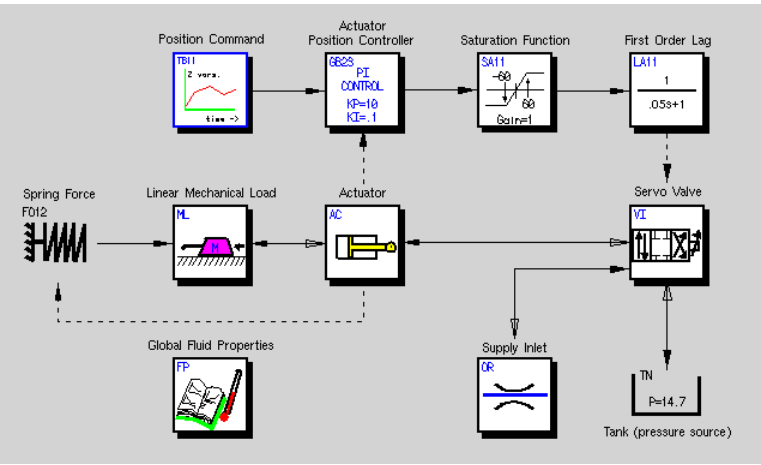


Bild 3-1: Screenshot der Entwicklungsumgebung von *EASY5*™

EASY5™ bietet Schnittstellen zu anderen CAE-Softwaresystemen aus den Bereichen Mehrkörper, Finite Elemente, Integrated-Circuit und CACSD (Computer-Aided Control System Design). Aus diesen Beschreibungen kann zu Emulationszwecken Fortran-Code generiert werden, der in C-Code umgewandelt werden kann. *EASY5*™ unterstützt auch mehrere Zielplattformen von dSPACE, SUN sowie verschiedene VMEbus Systeme.

Die Stärken von *EASY5*™ liegen in der Breite der unterstützten Entwurfs- und Analysemethoden. Das System war ursprünglich jedoch nicht für Rapid Prototyping elektronischer Steuergeräte aus-

gelegt, sondern diente in erster Linie der Beschleunigung der Offline-Simulation. Der erzeugte Code erfüllt deswegen keine harten Echtzeitbedingungen. Da Boeing jedoch stark an der Entwicklung eines CASE Datenaustauschformats (CDIF) beteiligt ist und gleichzeitig an einem Standard für den Produktmodelldatenaustausch (STEP) arbeitet, kann man hier in den nächsten Jahren die aussichtsreichsten Entwicklungen erwarten. Die vordefinierten Komponenten sind ausschließlich auf die Flugzeugindustrie ausgerichtet. In diesem Bereich müssen jedoch noch weitere Bibliotheken erstellt werden, um einen breiten Einsatz in weiteren Bereichen, beispielsweise in der Automobilindustrie, zu ermöglichen.

3.2 dSPACE

Das Entwurfssystem der *dSPACE GmbH* basiert auf den CASE-Werkzeugen MATLAB™/Simulink und MATRIX$_X$™/SystemBuild, die zu Entwurf, Analyse, Optimierung und Offline Simulation eingesetzte werden. Der Entwurf geschieht bei beiden CASE-Werkzeugen grafisch und kann ständig verfeinert werden. Die von den CASE-Werkzeugen mitgelieferten Blöcke werden durch die Blöcke der dSPACE Block-Bibliothek ergänzt. Darin enthalten sind auch Blöcke zur Integration der dSPACE Hardware (Ein-/Ausgabe Hardware).

Wurde das System oder ein Teilsystem implementiert, so kann der zugehörige C-Code bei Verwendung des CASE-Werkzeugs MATLAB™/Simulink mit Hilfe des Werkzeugs *Real-Time Workshop* generiert werden. Danach modifiziert das *dSPACE*™ Werkzeug *Real-Time Interface* den Code, steuert den Download auf die Zielhardware und startet die Emulation des Modells. Zur Emulation des Modells stellt die dSPACE Umgebung eine Reihe von weiteren Werkzeugen zur Steuerung und Kontrolle des Prototypen zur Verfügung. Die Werkzeuge *Cockpit* und *RealMotion* dienen in erster Linie zum Monitoring und Analyse der Modell-Emulation. Sie erlauben die Darstellung aller Steuer- und Ausgabegrößen.

Die Unterstützung des *dSPACE*™ Entwurfssystems umfaßt Zielhardware, die auf dem Prozessor DEC Alpha AXP von Digital Equipment und dem digitalen Signalprozessor TMS320Cx0 von Texas Instruments basiert. Dabei stehen sowohl Single-Board Lösungen, die über einen Prozessor, bzw. Signalprozessor und Ein-/Ausgabe Hardware bestehen, als auch modulare Systeme zur Verfügung, die auf Basis des proprietären *dSPACE*™ Hochgeschwindigkeitsbusses PHS (peripheral high-speed) den Aufbau eines skalierbaren Systems ermöglichen. Darüberhinaus stehen Schnittstellen zu anderen Bussystemen wie CAN und VMEbus zur Verfügung, mit denen weitere Hardwarekomponenten eingesetzt werden können. Allerdings vermindert der Einsatz dieser Schnittstelle die Leistungsfähigkeit der Hardwarekomponenten. Der Einsatz eines Echtzeitbetriebssystems für die Zielhardware ist nicht vorgesehen. Dadurch ergeben sich gegenüber Lösungen mit Echtzeitbetriebssystemen Performancevorteile, die jedoch bei Eigenlösungen im Hard- und Softwarebereich zu erhöhtem Integrationsaufwand führen. Außerdem ist die Portierbarkeit des erzeugten Programm-Codes nicht mehr gewährleistet.

Durch die umfangreiche Hardwareunterstützung hat das *dSPACE*™ Entwurfssystem eine starke Verbreitung in der Automobil- und Luftfahrtindustie gewonnen. Akzeptanzprobleme treten bei dieser Lösung häufig auf, wenn bereits bestehende Entwicklungssysteme auf Basis anderer CASE-Werkzeuge im Einsatz sind, die nicht direkt oder nur unzureichend unterstützt werden (beispielsweise STATEMATE™). Da das *dSPACE*™ Entwurfssystem seine Hardwarekomponenten bereits bei der Modellierung verwendet, tritt eine Vermischung von Modellierung und Implementierung

auf. Eine Integration von diskreten Teil-Modellen erfolgt auf Basis einer C-Code Beschreibung, die in das (kontinuierliche) Modell integriert werden kann. Während diese Vorgehensweise zur Code-Erzeugung genügt, kann eine Gesamtsimulation mit Unterstützung des CASE-Werkzeugs nicht gewährleistet werden. Mit der MATLAB™/Simulink Ergänzung Stateflow steht in Zukunft eine Möglichkeit zur Verfügung, diese Problematik zu umgehen. Allerdings verfügt Stateflow über einen geringeren Funktionsumfang im Vergleich zu STATEMATE™.

3.3 CAMeL

Das *Computer–Aided Mechatronic Laboratory* (*CAMeL™*) ist eine offene Computer Aided Control System Design (CACSD) Entwurfsumgebung, die den Entwurf von mechatronischen Systemen unterstützt. *CAMeL™* wurde am Mechatronik Labor der Universität Paderborn von 1991 an entwickelt. Neben den mitgelieferten Werkzeugen zur Analyse von linearen und nichtlinearen Systemen sowie deren Optimierung basierend auf der Systembeschreibungssprache DSL, unterstützt *CAMeL™* auch die Erweiterung durch andere Entwicklungsumgebungen. *CAMeL™* besitzt drei Ebenen, die sich im Abstraktionsgrad unterscheiden:

- **Objective-Dynamic System Structure (ODSS):** In dieser fachspezifischen Ebene werden die einzelnen Systemteile mit den dafür üblichen Beschreibungsformen modelliert. Das Gesamtsystem wird mit der objekt-orientierten Beschreibungssprache ODSS zusammengefügt. Die Beschreibungssprache verfügt über Basiselemente, hierarchische Elemente und Koppelelemente. Die Informationen können in textueller Form zum Austausch zur Verfügung gestellt werden.
- **Objective-Dynamic System Language (ODSL):** Eine Systemrepräsentation in ODSS kann in eine mathematische Beschreibung überführt werden. Die dabei anfallenden Gleichungen werden von ODSL beschrieben. Auch hier ist eine textuelle Repräsentation der Daten möglich.
- **Dynamic System Code (DSC):** Mit DSC wird die verarbeitungsorientierte Sichtweise beschrieben. Hierbei wird das System in einer abstrakten, maschinenunabhängigen Weise beschrieben. Somit steht ein einheitlicher Ausgangspunkt zur Erzeugung von Echtzeitcode zur Verfügung. Die Darstellung erfolgt ebenfalls in textueller Form.

CAMeL™ erlaubt keinen Wechsel der Beschreibungsebene, der zur abstrakteren Ebene hin gerichtet ist. Ein Datentransfer kann also nur von ODSS nach ODSL und DSC durchgeführt werden. Somit kann ein System nur auf der ODSS Ebene vollständig rekonstruiert werden. Die Systembeschreibungssprache ODSL hat eine starke Ähnlichkeit mit der block-orientierten Sprache BDL der BEACON Entwicklungsumgebung.

CAMeL™ greift zum ersten Mal die Idee eines standardisierten Zwischenformats in Form der mathematischen Beschreibung ODSL auf. Allerdings können mit dieser Darstellung nur Systeme repräsentiert werden, die in Form von Differentialgleichungen darstellbar sind. Systeme, die ereignisbasiert oder dem Bereich OOAD (Object-Oriented Analysis and Design) entstammen, können mit dieser Darstellung nicht beschrieben werden. Da *CAMeL™* nicht kommerziell vertrieben wird, ist die Akzeptanz von *CAMeL™* und seinem Datenaustauschformat nur gering.

3.4 CASE-Werkzeuge mit zusätzlichen Hardwarekomponenten

Seit einigen Jahren existieren Ergänzungen für gängige CASE-Werkzeuge im diskreten und kontinuierlichen Bereich, die ein Rapid Prototyping ermöglichen sollen. Oftmals bestehen diese Ergänzungen aus speziellen Hardwarekomponenten, für die ein Programm-Code passend zur Modellierung des CASE-Werkzeugs erzeugt werden kann. Dieser Programm-Code wird auf einer Prozessorplatine ausgeführt und steuert über ein Bussystem Ein-/Ausgabekomponenten an, die die Verbindung zur Umgebung herstellen. Die Komponenten sind genau aufeinander abgestimmt, so daß die Einbindung fremder Werkzeuge oder Codeteile nicht oder nur schwer durchgeführt werden kann.

RealSim AC-104

Für das CASE-Werkzeug MATRIX$_X$™ existiert von der Herstellerfirma ISI die Möglichkeit, Code zu generieren und diesen auf dem Rapid Prototyping System *RealSim AC-104* auszuführen. Die Hardware von *RealSim AC-104* besteht aus einem Intel 486 oder Pentium Prozessor, 8 MByte RAM und einer 4 MByte großen Flash-Disk. Als Eingabe-/Ausgabe Hardware sind 16 A/D Kanäle, 8 D/A Kanäle und 40 digitale Kanäle vorgesehen. In industriellen Projekten wurden zusätzlich VMEbus basierte Hardwarekomponenten oder Industry Pack (IP) Module benutzt, um eine Signalanpassung an die Sensorik/Aktorik des Prozesses vornehmen zu können. Wegen ungenügender Leistungsfähigkeit des generierten Codes mußten außerdem weitere Hardwarekomponenten (beispielsweise zur Ansteuerung von Motorventilen) eingebunden werden. RealSim AC-104 eignet sich nur zum Rapid Prototyping von Modellen, die mit MATRIX$_X$™ entworfen wurden. Zusätzlich ist die Unterstützung von Ein-/Ausgabe Hardware stark begrenzt.

System Engineering Workbench

Die *System Engineering Workbench*™ (*SEW*) wurde ab 1996 von Siemens entwickelt und wird von HP vertrieben. Sie erweitert das CASE-Werkzeug MATRIX$_X$™ um einige automobilspezifische Blöcke (beispielsweise Kennlinienfelder, Zähler) und erzeugt über MATRIX$_X$™ Programm-Code. Der Programm-Code wird auf einer mitgelieferten Zielhardware ausgeführt, die aus zwei digitalen Signalprozessoren C40/50 MHz von Texas Instruments mit 2 MByte RAM besteht. Ein Prozessor wird zur Applikationsausführung benutzt, während der zweite zur Aufbereitung der Daten für CAN/Flash/Ethernet-Schnittstellen dient. Als Ein-/Ausgabe Hardware ist ein Modul namens *SARA* vorgesehen, das aus 32 A/D Kanälen, 16 D/A Kanälen, 16 digitalen Eingangssignalen und 16 digitalen Ausgangssignalen besteht. Da die Ausführungszeit mit 2 ms recht hoch ausfällt, kann ein kleinerer Zustandsautomat unter Umgehung des von MATRIX$_X$™ generierten Programm-Codes direkt auf der Ein-/Ausgabe Hardware ausgeführt werden. Auch dieser Ansatz ist speziell auf ein CASE-Werkzeug ausgerichtet. Eine größere Anzahl von Hardwarekomponenten wird benötigt, um die Signalanpassung an die Sensorik/Aktorik unterschiedlicher Prozesse vornehmen zu können.

3.5 BEACON

Die Entwicklungsumgebung *BEACON*™ ist eine grafische Entwicklungsumgebung für den Entwurf von Steuerungs- und Regelungssystemen. Dabei erlaubt *BEACON*™ die Darstellung von Systemen in Form von Blockdiagrammen, die Simulation des Systems und die anschließende Trans-

formation des Systems in ausführbaren Echtzeit-Code. *BEACON™* wurde seit 1992 von General Electric Corp. entwickelt und wird von Applied Dynamics International vermarktet.

Durch die offene Architektur von *BEACON™* ist es möglich, bei Bedarf neue Funktionsblöcke hinzuzufügen. Dazu verfügt *BEACON™* über eine einfache, funktionale Beschreibungssprache namens *BEACON Block Description Language* (BDL), die auch die Beeinflussung der Code-Generator Eigenschaften ermöglicht. *BEACON™* besteht im wesentlichen aus drei Komponenten:

- einem grafischen Blockdiagramm-Editor, mit dem der Entwurf eines Systems erfolgt.
- einer Netzliste (Zwischenformat), die aus dem Diagramm generiert wird und eine vollständige Beschreibung des Diagramms beinhaltet. Die Netzliste ist schlüsselwort-orientiert und ihre Syntax ist vergleichbar mit der Syntax der Hochsprachen Pascal oder Ada.
- einem automatischen Codegenerator, der das grafische Modell aus Blockdiagrammen in einen simulations- oder echtzeitfähigen Code transformiert.

BEACON™ bietet keine Unterstützung für bestehende CASE-Werkzeuge wie MATRIX_X™ oder STATEMATE™. Da keine Weiterentwicklung des bestehenden Systems erfolgt, wird auch keine Einbindung von CASE-Werkzeugen mehr stattfinden. Durch die unflexible BDL können nicht alle notwendigen Konstrukte abgebildet werden oder erfordern einen hohen Implementierungsaufwand (beispielsweise History-Connectoren bei Statecharts). Trotzdem stellt *BEACON™* einen ersten ernstzunehmenden Rapid Prototyping Ansatz dar, der bereits mit einem einfachen Zwischenformat arbeitet und ein konzept-orientiertes Rapid Prototyping als Ziel hat.

3.6 DUCADE

Das *Domain Unified CAD Environment* (*DUCADE™*) [WaRW95] wurde an der University of California, Berkeley entwickelt und verbindet Werkzeuge des mechanischen und elektronischen Entwurfs in einer Entwurfsumgebung. Dabei werden mechanische CAD-Werkzeuge wir ARIES und elektronische CAD-Werkzeuge aus dem Printed Circuit Board (PCB) Bereich wie RACAL/ FINNSE gekoppelt. Die heterogenen Datenbanken der einzelnen Werkzeuge wird dabei in eine vereinheitlichte Datenbank überführt. Diese Datenbank stellt sich jedoch nur nach außen hin als einheitlich dar, besitzt jedoch nach wie vor eine getrennte Repräsentation ohne eine gemeinsame übergeordnete Datenstruktur. Mit dem Environment soll eine gemeinsame Simulation und die anschließende Fertigung des mechanischen Aufbaus, sowie die Herstellung der Leiterplatte ermöglicht werden. Die Arbeiten an *DUCADE™* sind seit 1996 beendet.

3.7 ASCET-SD

ASCET-SD™ ist eine Entwicklungsumgebung für eingebettete Steuerungs- und Regelungssysteme im Automobilsektor der Firma ETAS. Ausgangspunkt von *ASCET-SD™* war eine Dissertation an der Universität Paderborn [Eppi93], die zu einem Spin-off von der Firma Bosch führte. Zur Modellierung werden von *ACSET-SD™* eigene Entwurfs-Werkzeuge zur Beschreibung von funktionalen Modellen, Zustandsautomaten und echtzeit-spezifischen Code-Elementen eingesetzt. Durch die eigenentwickelten Werkzeuge liegt eine vollständige Einbettung in die Entwurfsumgebung vor. Zu anderen CASE-Werkzeugen bietet *ASCET-SD™* Import-Schnittstellen an, die eine Integration von MATRIX_X™ oder MATLAB™ Modellen ermöglichen. Eine echte Systemintegration mit diesen

Werkzeugen liegt nicht vor. *ASCET-SD*™ verfügt über eine umfassende Projektverwaltung, benutzer-definierbare Komponenten-Bibliotheken und Visualisierungselemente für eine Offline- und Online-Simulation. Des weiteren ist ein Code-Generator vorhanden, der die Erzeugung von echtzeitfähigem Code zur Ausführung auf mehreren Zielplattformen erlaubt.

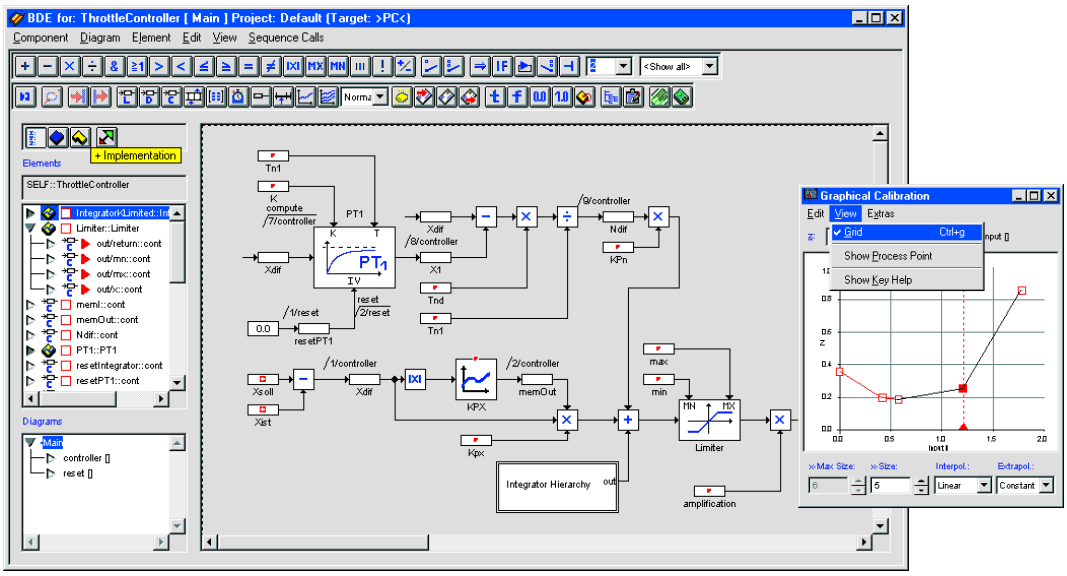


Bild 3-2: Screenshot der Entwicklungsumgebung von *ASCET-SD*™

Zu den Zielplattformen gehören Motorola PowerPC basierende Systeme, Mikrocontroller der Siemens 16x-Serie und ein Transputer-System. Auf den Zielplattformen wird der Code mit dem Echtzeitbetriebssystem *ERCOSEK*™ ausgeführt. Ein-/Ausgabehardware Komponenten liegen mit CAN- und VMEbus Interface vor und unterstützen eine Vielzahl von Anwendungen im Automobilbereich.

Das Echtzeitbetriebssystem *ERCOSEK*™ unterstützt den im Automobilsektor gängigen Standard OSEK. *ERCOSEK*™ zeichnet sich durch geringen Ressourcenverbrauch, statisches und dynamisches Scheduling sowie kooperatives und preemptives Multitasking aus. *ERCOSEK*™ unterstützt die Prozessorfamilien Motorola 68k, PowerPC und Siemens C16x.

Im Bereich der Entwurfsmethoden fehlen auf Seiten der diskreten Modellierung die Erweiterungen, die Statecharts bei der diskreten Modellierung eingebracht haben (Hierarchie, Parallelität, Zeitbedingungen). *ASCET-SD*™ unterstützt nur einfache Zustandsautomaten, die die Modellierung komplexerer Systeme nur unzureichend unterstützen. An dieser Stelle sind jedoch bereits Entwicklungen im Gange, die eine verbesserte Unterstützung bei der diskreten Modellierung ermöglichen. Die Ein-/Ausgabe Hardwareunterstützung hat noch nicht die Breite der Firma dSPACE erreicht. Im Zielbereich der Automobilindustrie besitzt ETAS jedoch bereits eine wesentlich größere Tiefe im Vergleich zu dSPACE. *ASCET-SD*™ wird ständig weiterentwickelt und soll zukünftig mit weiteren Bibliotheken und einem vergrößerten Hardware-Angebot ausgestattet werden. Durch die enge Verbindung mit der Firma Bosch können Entwicklungen im Fahrzeugbereich aufeinander abgestimmt werden.

3.8 PROCORS

Im Rahmen einer Doktorarbeit bei der Firma BMW wurde unter der Bezeichnung *Prototype Code for Real-Time Systems* (*PROCORS*) [Spre95] von 1993 bis 1996 ein echtzeitfähiger Codegenerator für STATEMATE™-Modelle entwickelt. Der Codegenerator geht von einer eigenen Semantik von Statecharts aus, die den Funktionsumfang von STATEMATE™ einschränkt. Die Semantik ermöglicht eine vereinfachte und somit schnellere Verarbeitung. *PROCORS* löst die Hierarchie eines Modells auf, während parallele Prozesse getrennt voneinander abgearbeitet werden. Um eine gegenseitige Beeinflussung der Prozesse zu vermeiden, werden diejenigen Zustandsübergänge verboten, die die Grenzen der parallelen Prozesse überschreiten. Deswegen müssen die parallelen Teilmodelle nicht gleichzeitig abgearbeitet werden, sondern können auch sequentiell ablaufen.

Durch die Verwendung einer eigenen Semantik und der gleichzeitigen Unverträglichkeit mit dem STATEMATE™-Simulator hat sich dieser Ansatz nicht durchgesetzt. *PROCORS* wurde insbesondere wegen des effizienten Codes für einige Projekte bei der Firma BMW eingesetzt, die Arbeiten an *PROCORS* sind jedoch mittlerweile eingestellt.

3.9 Forschungsansätze

Arbeiten im Bereich Rapid Prototyping werden seit 1996 von der Deutschen Forschungsgemeinschaft DFG gefördert. Im Rahmen des Projektes *Rapid Prototyping für integrierte Steuerungssysteme mit harten Zeitanforderungen* werden gezielt Forschungsvorhaben im Bereich Architektur und Automation bei der Umsetzung einer Spezifikation zum Prototypen gefördert. Die relevanten Arbeiten aus diesem Forschungsprojekt werden in den nächsten Abschnitten behandelt.

Emulation von Steuergeräten für die Fahrzeugtechnik und Mechatronik

Das Projekt, das am Lehrstuhl für Prozeßrechner an der Technischen Universität München durchgeführt wird, zielt auf die Implementierung von umfangreichen, komplexen eingebetteten Systemen. Als Zielsystem dient ein konfigurierbares, heterogenes Multiprozessorsystem mit unterschiedlich komplexen Verarbeitungseinheiten (Mikroprozessoren und Digitale Signalprozessoren), die untereinander mit einem leistungsfähigen Bussystem verbunden sind. Als konfigurierbare Ein-/Ausgabeeinheit (Configurable I/O Processor) dient ein FPGA von Xilinx.

Das Konzept sieht den Einsatz der Modellierungswerkzeuge STATEMATE™, ObjecTime™ und MATRIX_X™ vor. Die automatische Generierung von C- und VHDL-Code wird unterstützt, allerdings werden zumindest in der ersten Phase nur die mitgelieferten Codegeneratoren der Modellierungswerkzeuge eingesetzt. Somit muß von einer geringen Leistungsfähigkeit des erzeugten Codes ausgegangen werden. Der Idee, eine Beschleunigung der Abarbeitung im Bereich Rapid Prototyping durch programmierbare Hardware zu erreichen, steht allerdings eine deutlich erhöhte Zeitdauer für die Programmierung der Hardware (Place & Route) gegenüber.

Softwaremapping für Rapid Prototyping Systeme

Das Projekt *Softwaremapping für Rapid Prototyping Systeme* wird im Fachbereich Informatik an der Technischen Universität Dortmund durchgeführt. Es befaßt sich mit der Entwicklung von Echtzeit-Compilern, die effizienten Code für eingebettete Systeme generieren können. Da traditionelle Compiler keine zeitlichen Randbedingungen berücksichtigen, liegt der Schwerpunkt dieses Vorha-

bens in der Entwicklung geeigneter Compiler Methodiken, die die Einbindung dieser Bedingungen erlauben. Dabei soll zusätzlich eine Einbindung der Compiler in das Hardware-Software Co-Design System *Cool* erfolgen.

3.10 FPGA-basiertes Rapid Prototyping

Eine weitere Möglichkeit für ein Rapid Prototyping stellen Hardware- oder Logik-Emulatoren dar. Diese werden z.B. von der Firma *Quickturn* (*Logic Animator*™) oder *Zycad* (*Paradigm RP*™) hergestellt und verwenden eine Anordnung von programmierbaren Hardware-Bausteinen (fast ausschließlich FPGAs) und programmierbaren Verbindungsbausteinen (beispielsweise *APTIX*). Erstellt wurden diese Emulatoren ursprünglich zur prototypischen Realisierung von digitalen IC-Bausteinen. Da einige CASE-Werkzeuge über die Generierung von Hardware-Beschreibungssprachen verfügen, können diese ebenfalls Modelle für Emulatoren erzeugen. Durch die hohen Taktraten, die bei diesen Emulatoren bis in den MHz Bereich reichen können, sind diese Lösungen gerade bei sehr leistungsfähigen Applikationen von Vorteil. Allerdings spricht die geringe Auslastung der einzelnen FPGAs (max. 10% Auslastung [Turn99]) und die im Vergleich zur Erzeugung von Programmcode für Mikrocontroller oder digitale Signalprozessoren lange Reprogrammierungszeit gegen diese Lösung. Ein Rapid Prototyping in diesem Bereich entspricht mehr dem architektur-orientierten Rapid Prototyping, nicht dem konzept-orientierten Rapid Prototyping (siehe Kap. 2.3). Der Hauptanwendungsbereich dieser Systeme liegt deswegen auch im Bereich der Prozessorentwicklung.

3.11 Bewertung

Da *Rapid Prototyping* im Bereich elektronischer Systeme erst seit einigen Jahren eine weitere Verbreitung gefunden hat, könnte man vermuten, daß Ansätze in diesem Bereich noch relativ jung sind. Dies ist allerdings nicht der Fall. Bild 3-3 zeigt eine zeitliche Übersicht der Rapid Prototyping Ansätze. Dabei fällt vor allen der frühe Beginn von Rapid Prototyping in der Flugzeugindustrie (*EASY5*™) auf. Gerade in diesem Bereich trat bereits vor anderen Industriezweigen die Notwendigkeit einer durchgängigen Entwicklung unter Verwendung von Prototypen auf. Alle weiteren Anstrengungen im Bereich der Flugzeugindustrie brachten jedoch keine Verbesserungen der Methodik mit sich, sondern beschränkten sich auf den Ausbau und die bessere Unterstützung der vorhandenen Ansätze. Die komplette Neustrukturierung eines Rapid Prototyping Systems wurde nicht mit aller Konsequenz vorangetrieben.

Interessanterweise gingen seit Beginn der neunziger Jahre Impulse von der Automobilindustrie aus, um einen komplett durchgängigen Entwicklungsprozeß zu erreichen. Der Kostendruck und die zunehmende Konkurrenz im Bereich der elektronischen Steuergeräte erfordert eine enge *Integration* der verwendeten CASE-Werkzeuge. Dies schließt auch die Einführung einer einheitlichen Analyse, Simulation, Rapid Prototyping und Implementierung ein. Dieser wesentlich über die bestehenden Lösungen hinausgehende Prozeß bereitet allerdings große Probleme, weil die verfügbaren Werkzeuge keine Lösung für die geforderte Integration zur Verfügung stellen.

Die oben genannten Anforderungen nach einer Betrachtung des Gesamtsystems müssen mit den ursprünglichen Zielen von Rapid Prototyping (siehe Kap. 2.3) wie schnellen Iterationszyklen vereinbar sein. Die dabei im heutigen Entwicklungszyklus auftretenden Probleme müssen beseitigt

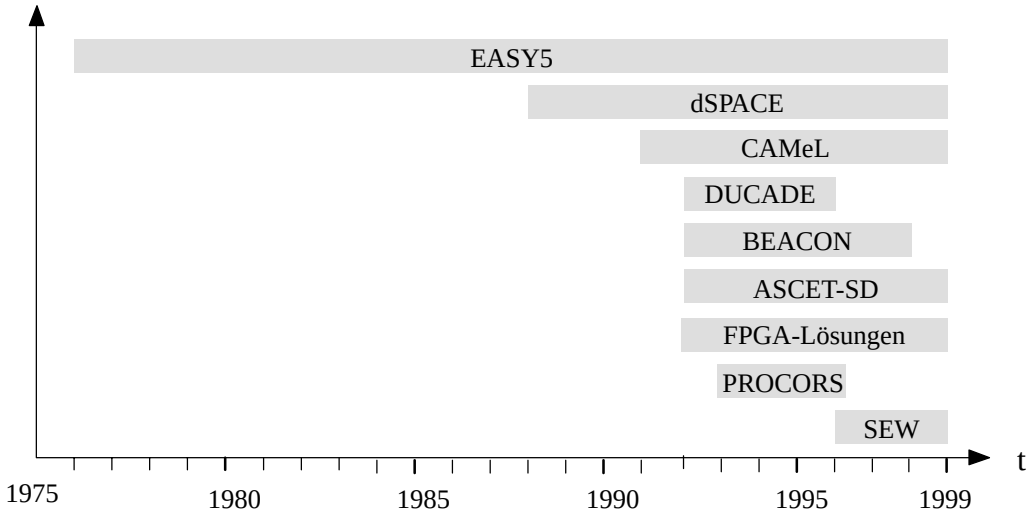


Bild 3-3: Zeitliche Übersicht der Rapid Prototyping Ansätze

werden. Besonders kritisch ist die heute verbreitete, enge Verknüpfung zwischen einem CASE-Werkzeug und den darauf aufbauenden Werkzeugen zur Analyse, Simulation und Rapid Prototyping. Eine Unabhängigkeit ist insbesondere deshalb wichtig, damit eine gemeinsame Verwendung von mehreren Modellierungswerkzeugen ermöglicht werden kann. Weist ein Ansatz ein werkzeugunabhängiges Zwischenformat auf (wie beispielsweise *CAMeL*), so ist das verwendete Zwischenformat so aufgebaut, daß nicht alle Modellierungsarten unterstützt werden. Auch die enge Verknüpfung zwischen einem CASE-Werkzeug und der verwendeten Rapid Prototyping Hardware, bzw. der Zielplattform für die Serienentwicklung muß vermieden werden.

Im folgenden Kapitel wird ein neues Konzept für den Aufbau eines Rapid Prototyping Systems unter Berücksichtigung der Gesamtsystementwicklung vorgestellt, das die oben genannten Forderungen erfüllt.

4 Konzept

4.1 Anforderungen

Im letzten Jahrzehnt wurden hauptsächlich CASE-Werkzeuge entwickelt, die für eine alleinstehende Nutzung konzipiert wurden und über nur eine Modellierungstechnik verfügten, beispielsweise für den Bereich State/Event *oder* Regelungstechnik (Bild 4-2a). Der Fall der gemeinsamen Nutzung mehrerer CASE-Werkzeuge wurde nicht oder nur ungenügend berücksichtigt. Nachdem in den letzten Jahren zunehmend der Wunsch von Anwenderseite nach einer gemeinsamen Nutzung geäußert wurde, wurde im universitären Umfeld begonnen, Kopplungen von CASE-Werkzeugen durchzuführen [ChEr93] [JäKl91]. Direkte werkzeugspezifische Kopplungen von CASE-Werkzeugen sind problematisch (Bild 4-1), da die Anzahl *N* der maximal notwendigen Filter pro neu eingebundenem CASE-Werkzeug quadratisch nach der folgenden Gleichung anwächst:

$$N = \sum_{i=1}^{n} i = \frac{n\,(n-1)}{2} \qquad (4.1)$$

Zusätzlich darf der Aufwand für die Pflege der Filter bei einer Aktualisierung der CASE-Werkzeuge nicht unterschätzt werden.

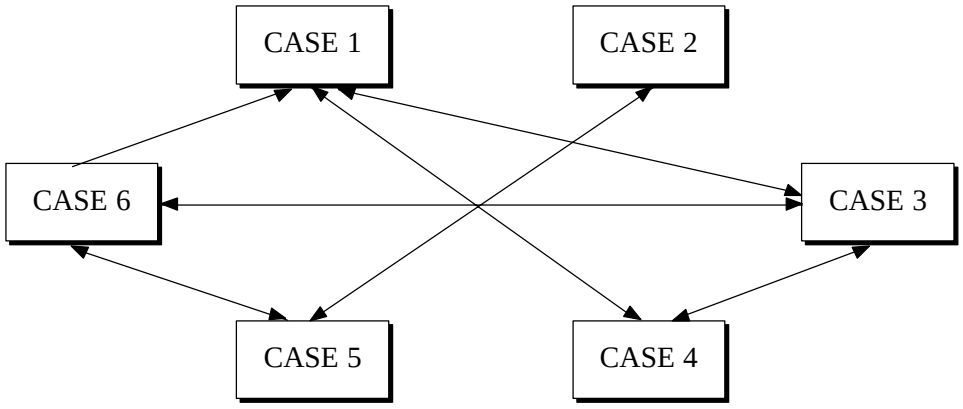


Bild 4-1: Anstieg der Anzahl der benötigten Filter

Die ersten Entwicklungen beschäftigten sich deswegen mit der Einbindung von automatisch erzeugtem C-Code eines CASE-Werkzeugs in ein anderes CASE-Werkzeug (Bild 4-2b). Beispielsweise ist die Integration eines von MATRIX$_X$™ generierten C-Codes in ein Activity-Chart von STATEMATE™ möglich. Da jedoch dabei eine Transformation von Modellebene zur Implementierungsebene vorgenommen wird, kann das entstehende Gesamtmodell nicht mehr analytisch untersucht werden. Eine Codegenerierung ist möglich, doch können unterschiedliche Modellierungsbereiche nicht mehr unterschieden werden.

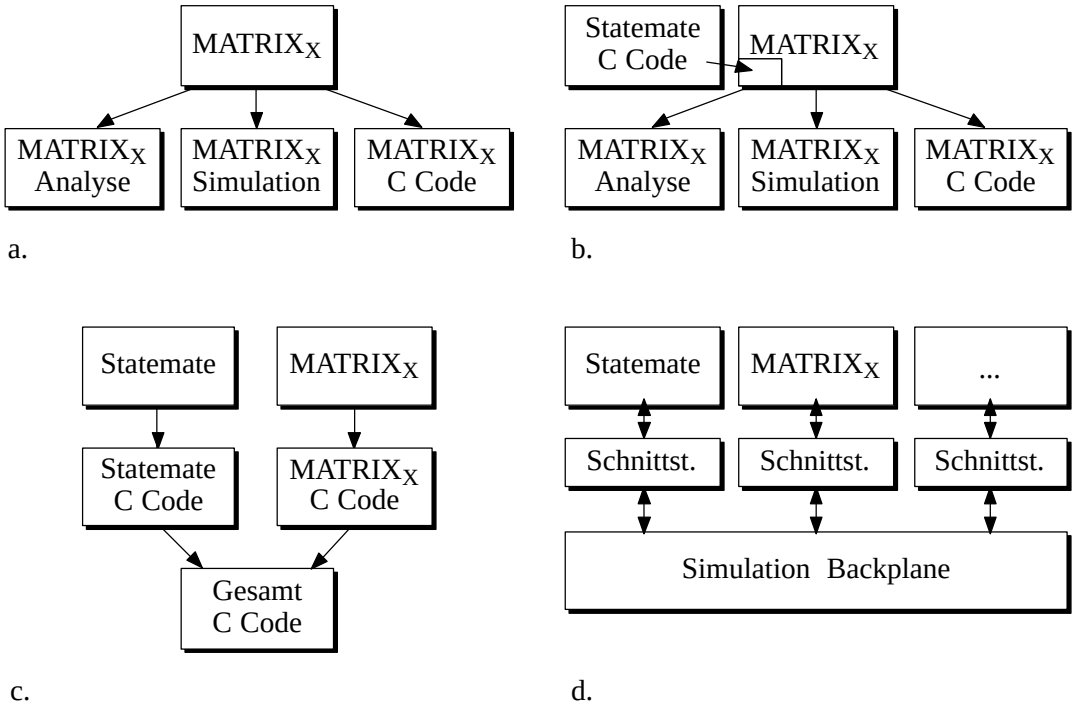


Bild 4-2: Evolution der Kopplung von CASE-Werkzeugen

Die Integration von Code, der von den CASE-Werkzeugen getrennt erzeugt wurde, bietet keine Lösung zu einer Gesamtanalyse und -simulation und setzt außerdem fehlerträchtige Handarbeit bei der Adaption der Codeteile voraus (Bild 4-2c). Die Idee einer Simulationbackplane (Bild 4-2d) [ScTa95], die die Simulatoren mehrerer CASE-Werkzeug Hersteller miteinander koppelt, erfüllt die Anforderungen an die Simulation. Eine Möglichkeit zur Codeerzeugung ist jedoch nicht vorgesehen.

Rapid Prototyping und die damit verbundene Codeerzeugung kann nicht als isoliertes Problem betrachtet werden. Vielmehr ist eine Einbindung in den gesamten Entwicklungsprozeß notwendig. Um die Einbindung in den gesamten Entwicklungsprozeß zu gewährleisten, müssen die folgenden Anforderungen erfüllt werden:

- Enge Integration von verschiedenen Modellierungstechniken (beispielsweise diskreter Bereich, kontinuierlicher Bereich, objekt-orientierter Bereich) und Modellierungswerkzeugen (beispielsweise STATEMATE ™, MATRIX$_X$ ™).
- Unabhängigkeit von speziellen Entwurfswerkzeugen, d.h. ein echter Datenaustausch zwischen Werkzeugen muß realisiert werden (einheitliches Datenaustauschformat).
- Strikte Trennung von Modellierung und Implementierung, um eine weitestgehende Unabhängigkeit von der gewählten Zielplattform zu erreichen.
- Unterstützung von Analyse, Simulation, Rapid Prototyping und Implementierung des gesamten Systems (hierbei muß auch die Einzelsimulation weiterhin möglich sein).
- Analyse, Simulation, Rapid Prototyping und Implementierung müssen basierend auf der jeweiligen Modellierungstechnik durchführbar sein und nicht von einem speziellen Werkzeug abhängen.
- Die Wiederverwendbarkeit von Systemkomponenten muß unterstützt werden.

Speziell für den Bereich Rapid Prototyping lassen sich noch weitere Punkte hinzufügen, die eine schnelle Umsetzung, eine komfortable Fehlerfindung und Parametrisierung sowie eine größtmögliche Portabilität gewährleisten sollen.

- Erzeugung von effizientem, hochoptimiertem Echtzeit-Code.
- Weitestgehende Unabhängigkeit des erzeugten Echtzeit-Codes von der verwendeten Zielhardware.
- Flexible, werkzeug-unterstützte Verbindung zwischen Modellierungsvariablen und physikalischen Signalen des Prototyps.
- Unterstützung einer Echtzeit-Prozeßvisualisierung.

4.2 Gesamtkonzept für das Rapid Prototyping System

4.2.1 Integrierte Entwurfsumgebung

Durch die bereits weit fortgeschrittene Verwendung von alleinstehenden CASE-Werkzeugen und dem gleichzeitigen Mangel an flexiblen, zur Kopplung fähigen CASE-Werkzeugen besteht die Notwendigkeit, den Rapid Prototyping Prozeß unabhängig von den Modellierungswerkzeugen zu gestalten. In diesen Prozeß müssen auch Analyse, Simulation und Codeerzeugung miteinbezogen werden. Um diese Unabhängigkeit zu erreichen, wird zwischen Modellierung und den weiterverarbeitenden Werkzeugteilen eine Zwischenschicht eingefügt (siehe Bild 4-3) [Volz95] [Doll95].

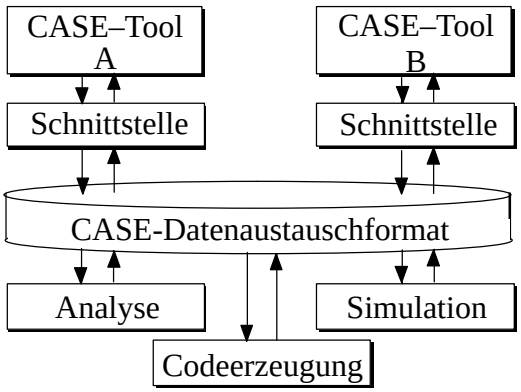


Bild 4-3: Trennung von Modellierung und weiterverarbeitenden Werkzeugen

Die Werkzeuge zur Modellierung werden im folgenden als *Front-end Werkzeuge* bezeichnet, die weiterverarbeitenden Werkzeuge werden als *Back-end Werkzeuge* bezeichnet. Diese Zwischenschicht dient als einheitliche Datenbank für alle am Entwurf beteiligten Werkzeuge und erfüllt die folgenden Aufgaben:

- Die Modellierungsinformation muß eineindeutig abgelegt sein, d.h. es darf keine mehrfache Beschreibung einer Modellierungsinformation möglich sein.
- Alle Modellierungsinformationen, die einen Bereich (z.B. diskreter Bereich) betreffen, müssen im selben Datenschema abgelegt werden.
- Back-end Werkzeuge können unabhängig von den Modellierungswerkzeugen entwickelt und eingesetzt werden. Die Back-end Werkzeuge können somit eine rein an der Modellierungstechnik orientierte Analyse, Simulation und Codeerzeugung durchführen.

- Die Durchführung einer integrierten, gemeinsamen Analyse, Simulation und Codeerzeugung unter Verwendung aller Modellierungsarten wird ermöglicht.
- Die Einbindung weiterer CASE-Werkzeuge erfordert nur noch eine Schnittstelle zum CASE-Datenaustauschformat. Die maximal benötigte Anzahl *N* an Filtern zu CDIF beträgt somit:

$$N = n \qquad (4.2)$$

Dabei benötigen die Back-end Werkzeuge keine keine weiteren Modifikationen, da ihre Modellierungsdaten aus dem werkzeugunabhängigen CASE-Datenaustauschformat entnommen werden.

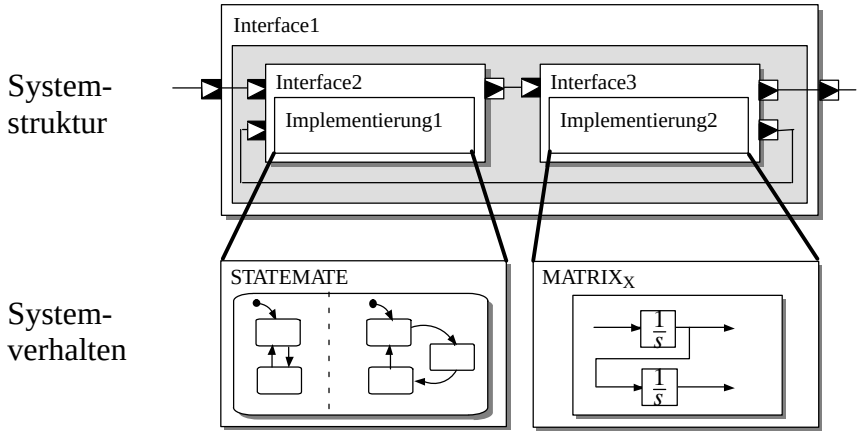


Bild 4-4: Modellierung von Systemstruktur und -verhalten

Bei allen oben genannten Vorteilen hat dieser Aufbau allerdings den Nachteil, daß eine wirkliche Kooperation zwischen CASE-Werkzeugen nicht stattfindet. Um diesem Problem zu begegnen, unterstützt das Konzept die Möglichkeit, die CASE-Werkzeuge bereits auf Modellierungsebene zu verbinden. Um ein Gesamtsystem zu modellieren, müssen Struktur und Verhalten festgelegt werden. Mit Hilfe eines weiteren Modellierungswerkzeugs kann die Struktur des Systems mit Funktionsblöcken festgelegt werden. Ein Funktionsblock kann weitere Funktionsblöcke beinhalten, die entweder die Systemstruktur verfeinern oder eine weitere Verhaltensbeschreibung enthalten können. In Bild 4-4 ist die Verfeinerung eines Funktionsblocks auf höchster Ebene und zwei Verhaltensbeschreibungen dargestellt, die mit unterschiedlichen CASE-Werkzeugen vorgenommen werden können (Mixed-Mode Beschreibungen). Die eine Implementierung umgebenden Blöcke werden Interfaces genannt und charakterisieren die Ein- und Ausgänge der benutzten Implementierung. Im Inneren dieser Interfaces können mehrere Implementierungen hinterlegt werden, die sowohl verschiedene Versionierungsstände als auch verschiedene Implementierungsebenen (Mixed-Level Beschreibungen) enthalten können.

Der Systementwurf kann auf diese Weise heterogen und integriert durchgeführt werden, ohne gleichzeitig starr auf eine Modellierungssprache fixiert zu sein. Die dabei verwendeten Modellierungsdaten können sowohl von mehreren existierenden CASE-Werkzeugen als auch von heterogenen Modellierungssprachen (siehe Kapitel 2) erzeugt werden.

4.2.2 Echtzeit-Codegenerierung und Struktur des Echtzeit-Codes

Viele kommerzielle CASE-Werkzeuge können aus einem Systementwurf, beispielsweise mit Statecharts oder Regel-Blöcken, Programm-Code erzeugen, der für den Einsatz in einem Rapid Prototy-

ping System benutzt werden kann (siehe beispielsweise [HaGe97]). Der eigentliche Zweck der Codeerzeugung dieser CASE-Werkzeuge liegt jedoch in der Beschleunigung der Systemsimulation, die durch Übersetzung in Programm-Code gegenüber der interpretierten Simulation große Geschwindigkeitsvorteile mit sich bringt. Der erzeugte Code ist jedoch nicht auf Größe, Geschwindigkeit oder Einhaltung von Echtzeitbedingungen optimiert und damit für einen Einsatz als Echtzeit-Code ungeeignet. Zusätzlich ist der Programm-Code spezifisch auf einen Entwurfsbereich zugeschnitten (beispielsweise erzeugt STATEMATE™ nur Code für Statecharts) und nur auf speziellen Zielplattformen ausführbar (beispielsweise die RealSim AC-104 Series für den Einsatz mit MATRIX$_X$™). Ansätze zur Verbesserung der Codegeneratoren von CASE-Werkzeugen [Spre95] konzentrierten sich auf die Echtzeitfähigkeit des erzeugten Codes, nicht jedoch auf heterogenen Echtzeit-Code oder Unabhängigkeit von der Zielplattform.

Ausgangspunkt dieses Ansatzes ist die Systemrepräsentation in einem CASE-Datenformat. Mit Hilfe der dort vorhandenen Datenstrukturen läßt sich das modellierte System formal analysieren und eine Simulation und Echtzeit-Codegenerierung durchführen. Der aus dem CASE-Datenformat erzeugte Code gibt das logische Verhalten des modellierten Systems wieder, ohne eine Prozeßanbindung zur Umgebung oder eine Ausführungsreihenfolge festzulegen. Es bietet sich somit an, die Struktur des Echtzeit-Codes in drei Teile zu untergliedern [Volz95] [Doll95]:

- der *Modell-Code*, der das logische Verhalten des modellierten Systems wiedergibt.
- der *Scheduler-Code*, der für das zeitliche Verhalten und die Synchronisation des Software-Prototypen verantwortlich ist.
- der *Ein-/Ausgabe-Code*, der die Konfiguration der Hardwarekomponenten und die Verbindung der modellierten Variablen mit den physikalischen Signalen der Umgebung vornimmt.

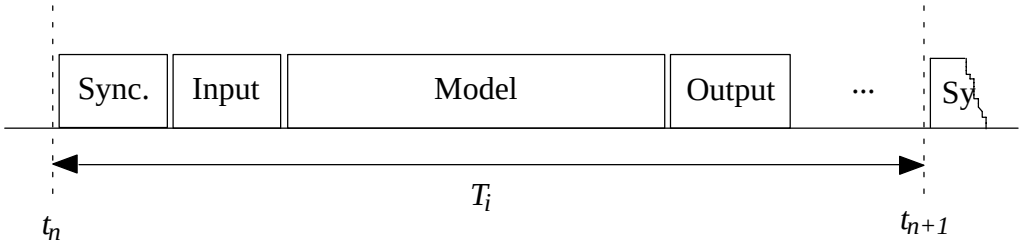


Bild 4-5: Struktur eines Modellschritts

Der Modell-Code und der Ein-/Ausgabe-Code werden bei jeder Codeerzeugung erneut dynamisch erzeugt, während der Scheduler-Code eine statische Komponente repräsentiert.

Die Ausführung des Echtzeit-Codes hängt von zeitlichen Randbedingungen ab. Prinzipiell besteht der Code aus einer unbegrenzten Schleife, die einen Modellschritt wiederholt abarbeitet (siehe Bild 4-5). Zuerst wird dabei ein Codesegment durchlaufen, das zur Synchronisation dient und dem Scheduler-Code zugeordnet ist. Dieser ist für die Ausführung des Software-Prototypen verantwortlich und kann die folgenden Codeteile so schnell wie möglich, mit feststehenden Zeitintervallen oder mit adaptierbaren Zeitintervallen abarbeiten. Danach werden die Eingabevariablen sowie die Verbindung der Variablen zwischen den Modellkomponenten aktualisiert. Nachdem die Aktualisierung der Daten vorgenommen wurde, wird das Modell selbst durchlaufen. Dabei können sowohl diskrete als auch kontinuierliche Anteile verarbeitet werden. Nach Beendigung des Modellschritts werden die Ausgabevariablen aktualisiert.

4.2.3 Anbindung an die Prozeßumgebung

Um den Rapid Prototyping Prozeß zu beschleunigen, muß die bislang mühevolle Anbindung an die reale Prozeßumgebung über die Ein-/Ausgabehardware verfeinert werden. Aufgrund der Vielzahl der Hardwarekomponenten, die entweder käuflich erworben oder selbst entwickelt sein können, muß ein Rapid Prototyping System mehrere Aspekte erfüllen:

- Software-unterstützte Auswahl der Ein-/Ausgabehardware.
- Software-unterstützte Zuordnung der Modellvariablen zu den physikalischen Ein/-Ausgabekanälen der Hardware.
- Automatische Erzeugung von Codesegmenten zur Initialisierung und Rücksetzung der Ein-/Ausgabehardware.
- Automatische Generierung von Codesegmenten zur Unterstützung der verwendeten Ein-/Ausgabeprotokolle (beispielsweise CAN-Bus, etc.).

Die Anforderungen an die Schnittstellensignale sind sehr vielfältig. Im Automobilbereich werden Spannungen bis 24V und Strömen bis zu 40A, digitale Signale mit einstellbaren Schwellspannungen, analoge Eingänge in Form von Spannungs-, Strom- und Widerstandsänderungen sowie frequenz- und pulsweitenmodulierte Signale benötigt. Hierzu müssen eine Reihe von voll-programmierbaren Schnittstellenmodulkarten entwickelt werden, die aus verschiedenen Signalerfassungskarten mit Businterface und universell einsetzbaren, softwarekonfigurierbaren Signalanpassungsmodulen ohne Businterface bestehen. Für die Kommunikation mit anderen Steuergeräten sind beispielweise CAN-Bus oder RS232-Module vorzusehen.

Der Aufbau der Hardware hängt jedoch stark vom Einsatzgebiet ab. In der vorliegenden Arbeit wurden exemplarisch Module entwickelt, die für die Verwendung bei der Steuergeräteentwicklung in der Automobiltechnik geeignet sind. Studien [BuSc97] haben gezeigt, daß eine Adaption der Hardware auch für weitere Bereiche (z.B. Bahntechnik) möglich ist.

4.2.4 Vorgehensweise

In den folgenden Kapiteln werden die vorgestellten Themengebiete näher untersucht. Der Schwerpunkt der Arbeit liegt dabei insbesondere auf den noch ungelösten Fragen der Verbindung von heterogenem Systementwurf und Rapid Prototyping, der Kopplung von Komponenten unterschiedlicher Entwurfsbereiche auf Entwurfs- und Implementierungsebene sowie der Erzeugung eines möglichst effizienten Echtzeit-Codes.

Nachdem oben bereits eine Übersicht der für ein Rapid Prototyping System notwendigen Komponenten präsentiert wurde, gehen die folgenden Kapitel im Detail auf die einzelnen Probleme ein. Dabei werden Teilaspekte, die sich auf bekannte Methoden und Verfahren der Software-Entwicklung zurückführen lassen, nicht oder nur kurz behandelt. Insbesondere betrifft dies die Datenablage in einer Datenbank, die zur Klärung der wesentlichen Problemstellungen nicht relevant ist. Statt dessen werden die folgenden Fragen behandelt, die bisher im Bereich der akademischen und industriellen Entwicklung eines Rapid Prototyping Systems nicht oder nur unvollständig Berücksichtigung fanden.

- Wie muß eine Entwurfsumgebung aufgebaut sein, um einen heterogenen Entwurf mit Unterstützung heutiger und zukünftiger Werkzeuge leisten zu können?

- Welche Eigenschaften muß ein Datenformat aufweisen, das zur Ablage von diskreten, kontinuierlichen und objekt-orientierten Modellierungsdaten geeignet ist und das die in Kapitel 4.1 dargestellten Anforderungen erfüllt? Existiert bereits ein Datenformat, das diese Anforderungen erfüllt?
- Welche Eigenschaften muß die Kopplung von Modellierungsdaten auf Codeebene aufweisen? Welche Kommunikationsmechanismen sind für die Kopplung der einzelnen Modellierungsteile notwendig?
- Wie sieht die Struktur des Echtzeit-Codes aus, der für ein Rapid Prototyping Prozeß geeignet ist? Wie erreicht man höchste Leistungsfähigkeit bei einem gleichzeitigen hohen Portabilitätsgrad?

Die Fokussierung auf die Lösung dieser Fragen spiegelt sich anhand des Aufbaus der weiteren Kapitel wider. In Kapitel 5 wird zunächst auf die Struktur einer neuartigen Entwurfsumgebung eingegangen, die den heterogenen Systementwurf unterstützt. Kapitel 6 gibt dann erstmals geschlossen einen Überblick auf die Abstraktion dieser Modelldaten, die die Basis der in Kapitel 5 eingeführten Entwurfsumgebung bildet. Auf Echtzeitebene wird danach in Kapitel 7 die Systemkopplung in Echtzeit untersucht und in Kapitel 8 auf die Erzeugung von Echtzeitcode eingegangen. Die erzielten Ergebnisse sind Bestandteil von Kapitel 9. Kapitel 10 beinhaltet eine Zusammenfassung der erzielten Ergebnisse, beleuchtet die Grenzen dieser Arbeit und bietet einen Ausblick auf die zukünftige Entwicklung des Bereichs Rapid Prototyping.

5 Entwurf heterogener elektronischer Systeme

5.1 Problematik

Wie bereits in Kapitel 4 dargestellt, bereitet die Modellierung, Simulation, Analyse und Codeerzeugung aus einem einzelnen Werkzeug keine Schwierigkeiten. Erst bei der Modellierung und Weiterverarbeitung heterogener Systeme treten neue Aufgabenstellungen auf. Die Zusammenarbeit mehrerer Werkzeuge auf verschiedenen Abstraktionsebenen, die konsistente Datenhaltung bei der Weiterverarbeitung einzelner Modellteile, die Analyse und Simulation auf verschiedenen Abstraktionsebenen sowie die einheitliche Codeerzeugung bereiten zunehmend Probleme. Als Stand der Technik kann in diesem Bereich nur die vertikale Weitergabe von Daten angesehen werden. Oftmals geschieht die Zusammenarbeit über herstellerspezifische Formate, die durch weite Verbreitung einen Quasi-Standard gebildet haben. Hierfür ist die Weitergabe von Daten von synthetisierten VHDL-Dateien im XNF-Format an die Xilinx Xact™ Place&Route Software ein Beispiel.

Es existiert eine Vielzahl von rechnergestützten Entwurfswerkzeugen, die einen bestimmten Bereich des Entwurfs elektronischer Systeme unterstützen. Beispielsweise unterstützt das Werkzeug STATEMATE™ den Entwurf mit Statecharts und das Werkzeug Synopsys die Analyse und Synthese von VHDL-Code.

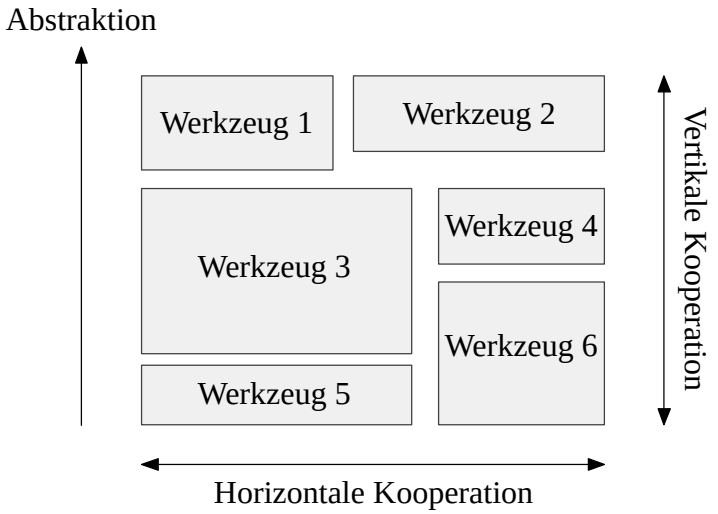


Bild 5-1: Kooperation bestehender Werkzeuge auf unterschiedlichen Ebenen

Weder bei den existierenden Werkzeugen, die zur Zeit industriell genutzt werden, noch bei Werkzeugen der nähereren Zukunft ist zu erwarten, daß ein Werkzeug alle Phasen des Systementwurfs von der Spezifikation bis zur Implementierung im jeweils gewünschten fein-granularen Detaillie-

rungsgrad abzudecken vermag. Bei der Kooperation von Entwurfswerkzeugen wird zwischen vertikaler und horizontaler Kooperation unterschieden. Die *vertikale Kooperation* beinhaltet die Zusammenarbeit von Entwurfswerkzeugen unterschiedlicher Abstraktionsstufen (engl. *mixed level*). Ein Beispiel für die vertikale Kooperation ist die Erzeugung von VHDL-Code aus dem CASE-Werkzeug STATEMATE™, das mit einem Syntheseprogramm (beispielsweise Synopsys) weiterbearbeitet wird. Unter *horizontaler Kooperation* wird die Zusammenarbeit von Entwurfswerkzeugen einer Abstraktionsebene verstanden, die unterschiedliche Entwurfsbereiche unterstützen können (engl. *mixed mode*). Ein Beispiel hierfür wäre die Zusammenarbeit der CASE-Werkzeuge STATEMATE™ und MATRIX$_X$™, die den diskreten und kontinuierlichen Modellierungsbereich abdecken.

Allein die notwendige Breite der horizontalen und vertikalen Entwurfsebenen (Bild 5-1) ist für einzelne Werkzeuge nahezu nicht unterstützbar. Eine Überführung der Daten der jeweiligen Entwurfsebene von einem Werkzeug in ein nächstes (*vertikale Integration*) oder die Zusammenarbeit von mehreren Werkzeugen (*horizontale Integration*) ist meist äußerst aufwendig und fehlerträchtig. Insbesondere können Systemänderungen auf niedrigeren Entwurfsebenen nicht in höhere Entwurfsebenen überführt werden und Modellierungen von mehreren Entwurfswerkzeugen nicht gemeinsam weiterverwendet werden. Daraus resultiert eine Verminderung der Entwurfssicherheit, die im ungünstigsten Fall zu einer fehlerhaften Implementierung führen kann.

Auch die Integration von ergänzenden Systementwurfstechniken wie Rapid Prototyping und Hardware-in-the-Loop müssen von der Entwurfsumgebung berücksichtigt werden und können somit nicht einzeln und isoliert betrachtet werden. Der dazu notwendige Aufbau einer Entwurfsumgebung wird in dieser Arbeit ebenfalls betrachtet und in Kapitel 5.4 vorgestellt.

Das aufkeimende Bewußtsein der Industrie für diese Problematik wird von mehreren Artikeln [Harr98] [IsLa97] untermauert, die sich dieser Problematik stellen. Insbesondere Flugzeughersteller, die mit ihren Produkten die strengsten Auflagen zu erfüllen haben, forschen seit mehreren Jahren auf diesem Gebiet. Eine wirkliche Lösung der Probleme ist nicht gelungen. Dies ist insbesondere an den aufwendigen Werkzeugkopplungen, deren ungenügender Wartbarkeit als auch an der ungenügenden Codeerzeugung bei Verwendung mehrerer Werkzeuge zu erkennen.

5.2 Anforderungen

Durch die weite Verbreitung von Verhaltensmodellierungen von elektronischen Systemen besteht keine Notwendigkeit, neue Beschreibungsformen einzuführen. Die vorhandenen Beschreibungen erlauben für nahezu alle Fälle, eine ausreichende Systemmodellierung durchzuführen. Als Anforderungen an eine Systemumgebung können die folgenden Punkte genannt werden:

- Kombination von bereichsspezifischen Modellierungen durch Festlegung einer Systemstruktur (Daten- und Kontrollflußmodellierung).
- Größtmögliche Flexibilität bei der Einbindung von unterschiedlichen Verhaltensmodellierungen in die Systemstruktur.
- Vertikale und horizontale Kooperation der beteiligten Werkzeuge.
- Möglichkeit zur hierarchischen Gliederung der Systemstruktur.
- Kombination von Modellen auf unterschiedlichen Abstraktionsstufen. Modelle unterschiedlicher Abstraktionsstufen bleiben dabei verbunden.

- Einbindung von Testbenchs und zusätzlichen Randbedingungen in die Modellierung ohne Verlust der Synthesefähigkeit.
- Wiederverwendbarkeit und Bibliotheksbildung von erstellten Modellen und Teilmodellen.

5.3 Eignung bestehender Ansätze

Entwurfsumgebungen, die eine Unterstützung des ganzheitlichen Entwurfsprozesses unter den oben aufgeführten Anforderungen gewährleisten, weisen oft Lücken auf. Im folgenden werden zwei mögliche Entwurfsumgebungen auf Eignung untersucht.

5.3.1 Frameworks

Frameworks vereinen die Verwaltung von Daten, die von CASE-Werkzeugen generiert werden können, mit der Datenverwaltung von Systemdateien.

Definition 5.1: Ein Framework kann als ein Werkzeug verstanden werden, das eine kooperative Aneinanderreihung von CASE-Werkzeugen erlaubt.

Mit einem Framework wird der Entwurf eines Systems von der abstrakten Systemspezifikation bis zur Implementierung innerhalb einer Umgebung möglich. Entscheidend ist dabei die nahtlose Integration der CASE-Werkzeuge zu einem übergeordneten Entwurfsprozeß.

Innerhalb einer zentralen Komponente des Frameworks kann der Entwurfspfad festgelegt werden. Unter einem Entwurfspfad wird die Reihenfolge des Werkzeugeinsatzes innerhalb des Entwurfsprozesses verstanden. Als Schnittstelle zwischen den Werkzeugen dienen dabei die von den CASE-Werkzeugen generierten Daten. Das Framework ist für den Aufruf der CASE-Werkzeuge und die Übergabe der korrekten Daten zuständig.

Ein Problem, das bei Frameworks auftritt, ist die Unterschiedlichkeit der einzelnen CASE-Werkzeuge in Bezug auf die Datenhaltung und die Aktivierung. Um diesem Problem zu begegnen, wurde mit der *Tool Encapsulation Specification* (TES) von der *CAD Framework Initiative* (CFI) eine Spezifikation zur Kapselung von Werkzeugen geschaffen [CADF95]. Diese Spezifikation definiert eine Sprachsyntax und Grammatik, die die Aufrufeigenschaften der Werkzeuge und deren Interpretation beschreibt. Als eines der wenigen Frameworks, die diesen standardisierten Weg zur Einbindung von Werkzeugen benutzen, ist *Lean Integration Platform* von C-Lab anzusehen. Sollte bei der Einbindung eines Werkzeuges in dieses Framework eine Beschreibung in TES-Format fehlen, muß der Benutzer den Aufruf des Werkzeugs selbst konfigurieren.

Trotz der Zusammenführung der CASE-Werkzeuge, die durch ein Framework möglich ist, werden eine Reihe von Problemen des Systementwurfs nicht gelöst. Im Gegensatz zu den in Kapitel 5.2 genannten Anforderungen, ist ein Framework nicht mehr als die Summe seiner Einzelkomponenten. Die eingebundenen CASE-Werkzeuge gewinnen nicht an Mächtigkeit, sie werden lediglich in einer bestimmten Reihenfolge ausgeführt. Dies wirkt sich insbesondere bei der horizontalen Kooperation von CASE-Werkzeugen aus. Da ein Framework lediglich administrative Aufgaben erfüllt, kann es die Koordination mehrerer CASE-Werkzeuge einer Abstraktionsebene nicht übernehmen. Erzeugt man aus den CASE-Werkzeugen C- oder VHDL-Code und arbeitet mit diesen Daten weiter, kann beispielsweise eine Simulation auf der ursprünglichen Abstraktionsebene nicht mehr vorgenommen werden. Zusätzlich muß die Kopplung des erzeugten Codes von Hand vorgenommen werden (siehe hierzu auch Kapitel 4.1).

Ein weiterer Schwachpunkt betrifft die Datenhaltung der CASE-Werkzeuge. Hier ist das Framework auf die Kooperation der CASE-Werkzeuge und des Benutzers angewiesen. Der Benutzer muß die erzeugten Daten in dem vorgesehenen Knoten des Datenbaums ablegen, um eine korrekte Weiterverarbeitung zu gewährleisten. Dies erfordert eine detaillierte Kenntnis des Datenbaums und somit eine zeitintensive Einarbeitung des Benutzers in die Datenhaltung. Zusätzlich erfolgt die Datenhaltung dateibasiert, was die Einbindung eines leistungsfähigen Datenbanksystems erschwert.

Auch die gemeinsame Nutzung von Systemressourcen und -funktionen ist mit einem Framework nicht möglich, da die beteiligten CASE-Werkzeuge einzeln arbeiten, ohne den Gesamtablauf zu berücksichtigen. Dies erhöht den Entwicklungsaufwand für CASE-Werkzeuge unnötig, da viele Funktionen wie beispielsweise die Datenhaltung oder die grafische Benutzeroberfläche gemeinsam genutzt werden könnten. Mit diesem Vorgehen würde zudem ein einheitliche Bedienung verwirklicht werden und zusätzliche Einarbeitungszeit könnte eingespart werden.

5.3.2 Aviatis' Zero-Latency Engineering Platform

Ein vielversprechendes Projekt wurde im Jahr 1998 von der kalifornischen Firma Aviatis begonnen. Dort wird erstmals versucht, über die Funktionalität eines Frameworks hinaus zu gehen und eine wirkliche Entwurfsumgebung zu implementieren, die für den Einsatz in großen Entwicklergruppen an verteilten Orten geeignet ist. Der Term "Zero-Latency Engineering" verdeutlicht, daß als Ziel der sofortige Zugriff auf alle relevanten Modellierungsdaten für jeden berechtigten Entwickler vorgesehen ist. Die Entwurfsumgebung bietet die Möglichkeit, bestehende WWW-Browser zur Ansicht von Modellierungsdaten zu verwenden. Dabei ist die Entwurfsumgebung unabhängig von der Art des verwendeten CASE-Werkzeugs. Die Realisierung dieser Umgebung ist allerdings noch nicht abgeschlossen und kann deswegen nicht abschließend bewertet werden.

5.4 Offene Entwurfsumgebung

Im folgenden wird eine Entwurfsumgebung für den heterogenen Systementwurf vorgestellt, die die Integration vorhandener CASE-Werkzeuge aus unterschiedlichen Entwurfsbereichen erlaubt und die Einbindung von weiteren CASE-Werkzeugen mit zukünftigen Beschreibungsmitteln ermöglicht [Hart97] [Grei97]. Es sei jedoch bereits an dieser Stelle darauf hingewiesen, daß keine eigenen Beschreibungsmittel für heterogene Systeme erstellt werden sollen, sondern die bereits vorhandenen Beschreibungsmittel zu einem Gesamtsystementwurf vereint werden sollen.

5.4.1 Schnittstellendefinition

Die im Vordergrund stehende Systemstrukturierung macht die Betrachtung der einzelnen Modellierungen als Objekte erforderlich, die zu einem Gesamtsystemmodell zusammengefügt werden können. Ein Objekt ist dabei eine funktionale Einheit, die durch eine Schnittstelle mit ihrer Umgebung in Verbindung tritt. Es ist unerheblich, welcher Inhalt sich im Inneren des Objekts befindet. Des weiteren besitzt ein Objekt Eigenschaften, die seinen Zustand wiedergeben. Die Objekte sind gekapselt, so daß nur das Objekt selbst direkten Zugriff auf seinen inneren Aufbau, Funktionen und Variablen besitzt. Ein Benutzer des Objekts und somit auch die Entwurfsumgebung selbst greift auf das Objekt indirekt über eine definierte Schnittstelle zu (Bild 5-2 links). Auf diese Weise kann die äußere Schnittstelle des Objekts beibehalten werden, auch wenn die innere Systemstruktur geändert

wird. Wird eine Änderung der Schnittstelle vorgenommen, so geschieht dies transparent für Objekt und Entwurfsumgebung.

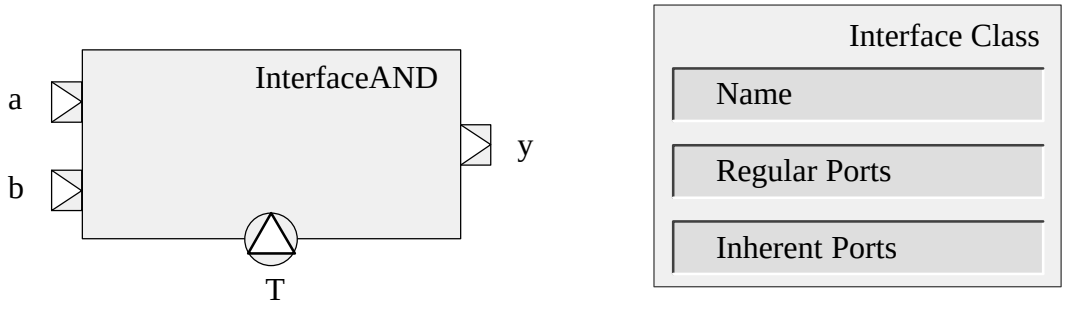


Bild 5-2: Elemente der Interface Class

Um Objekte zu erzeugen, können Schnittstellen-Klassen (engl. *interface class*) definiert werden (Bild 5-2 rechts). Diese besitzen einen eindeutigen Namen und können instanziiert werden, um ein Schnittstellen-Objekt zu erzeugen. Die instanziierten Schnittstellen-Klassen können als eine Bibliothek von Elementen gelten, die zur Modellierung verwendet werden können. Beispielsweise kann ein logisches UND-Gatter durch die Schnittstelle von Bild 5-2 beschrieben werden, das zwei Ports als Eingänge und einen Port als Ausgang definiert.

Um die Bestandteile eines Systems ebenfalls zu erfassen, die nicht unmittelbar Bestandteil der Systemfunktionalität sind, wird zwischen regulären und inhärenten Ports unterschieden. *Reguläre Ports* dienen dem Transport der eigentlichen Signale, die für die unmittelbare Systemfunktionalität verantwortlich sind. Für ein kontinuierliches System wären dies die Eingänge für Stell- und Meßgrößen und der Ausgang für die Regelgröße. Das zu implementierende System wird diese Schnittstellen besitzen. *Inhärente Ports* werden benötigt, da ein System auch von weiteren Einflüssen der Umgebung abhängig ist. Beispielsweise besitzen elektrische und mechanische Systeme oftmals eine starke Temperaturabhängigkeit. Diese Sekundärgrößen müssen ebenfalls festgelegt werden und bei der Simulation mit berücksichtigt werden. Diese Ports werden im implementierten System nicht mehr berücksichtigt, da zu diesem Zeitpunkt die Sekundärgrößen von der Umwelt vorgegeben sind. Das logische UND-Gatter von Bild 5-2 besteht also aus zwei regulären Ports als Eingänge und einem regulären Port als Ausgang. Zusätzlich wird die Umgebungstemperatur durch einen inhärenten Port in das System eingebracht.

Die Objekte kommunizieren untereinander über Daten- und Kontrollflüsse (siehe Kapitel 2.1.3). Über diese Kommunikationswege können Ereignisse, Bedingungen, beliebige Werte oder komplexe Protokolle (z.B. für Bussysteme) ausgetauscht werden. Die Spezifikation der Objektschnittstellen sind jeweils eng mit der Art der Kommunikationswege verknüpft.

5.4.2 Implementierungen

Die Objektschnittstelle definiert das Vorhandensein einer bestimmten Funktionalität. Es wird jedoch keine Aussage über die Implementierung der Funktionalität gemacht. Diese Eigenschaft kann ausgenutzt werden, um mehrere Implementierungen unter einer Schnittstelle zu hinterlegen (*Polymorphismus*). Es ist auf diese Art möglich, dieselbe Funktionalität in verschiedenen Abstraktionsstufen zu behandeln oder unterschiedliche Beschreibungsarten zu verwenden. Dies ist insbesondere für automatisch erzeugte Beschreibungen auf niedrigeren Abstraktionsebenen wichtig, die mit der Beschreibung auf höherer Ebene verbunden bleiben. Beispielsweise kann eine Modellierung in STATEMATE ™ und der zugehörige erzeugte C-Code innerhalb einer einzigen Komponente hinter-

legt werden. Dadurch werden Simulationen zur Funktionalitätsüberprüfung auf höherer Abstraktionsebene durchgeführt, während eine zeitlich präzise Simulation über den C-Code erfolgen kann.

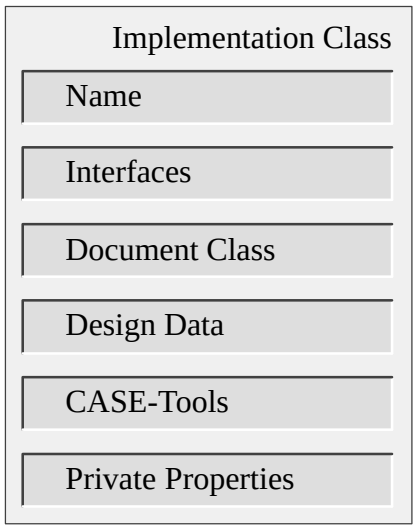


Bild 5-3: Elemente der Implementation Class

Um Systemimplementierungen aufnehmen zu können, wird wie bei den Schnittstellen eine eigene Klasse eingeführt, die als Implementierungs-Klasse (engl. *implementation class*) bezeichnet wird. Als Implementierungen kann eine Verhaltensbeschreibung oder eine strukturelle Beschreibung gewählt werden. Die Eigenschaften der Implementation Class (Bild 5-3) umfassen die folgenden Elemente:

- *Name*: Über einen eindeutigen Namen wird der Zugriff auf die Implementation Class vorgenommen.
- *Interfaces*: Die hier hinterlegte Liste umfaßt Verweise auf die von dem referenzierten Objekt einer Implementation Class unterstützten Schnittstellen. Die Schnittstelle ist dabei eine Referenz auf eine Interface Class aus der Liste der unterstützten Schnittstellen.
- *Document Class*: Dieses Element beinhaltet Informationen, wie die Daten des referenzierten Objekts der Implementation Class zu interpretieren sind. Über dieses Element kann die Art der Beschreibungsmethodik (z.B. Statechart, VHDL oder CDIF) ermittelt werden.
- *Design Data*: Hinter diesem Element ist die eigentliche Systembeschreibung im Format der Document Class abgelegt. Um eine Interpretation der Entwurfsdaten vorzunehmen, ist das verwendete CASE-Werkzeug notwendig.
- *CASE-Tool*: Das Element enthält Informationen über das CASE-Werkzeug, mit dem die Beschreibungsdaten dargestellt und modifiziert werden können. Hierbei ist es möglich, mehrere CASE-Werkzeuge für die Darstellung und Modifikation zu hinterlegen.
- *Private Properties*: Benutzerdefinierte Eigenschaften zur erweiterten Beschreibung eines referenzierten Objekts einer Implementation Class.

Dieses Konzept ist offen für alle Verhaltensbeschreibungen und schließt dabei auch zukünftige Beschreibungen ein, da bei der Einbindung der Entwurfsdaten ebenfalls das Prinzip der Kapselung verwendet wird. Die Entwurfsdaten werden durch ein Datenobjekt mit einer standardisierten Schnittstelle erfaßt, das die Einbindung gewährleistet. Das Datenobjekt ist Instanz einer Dokumentenklasse für Entwurfsdaten und stellt für jeden Datentyp eine eigene Dokumentenklasse zur Verfü-

gung. Datenobjekte für Statecharts können ihre Modellierung beispielsweise in Form eines Baums halten, während für VHDL Beschreibungen die Daten in Form eines ASCII Datenstroms gehalten werden können. In Kapitel 6 wird ein standardisiertes Datenformat vorgestellt, das für die Repräsentation aller Entwurfsdaten verwendet wird.

Um die Entwurfsdaten innerhalb der Entwurfsumgebung ohne Kenntnis des Aufbaus nutzen zu können, wird für jede Implementierung eine Schnittstelle definiert, die mit der Schnittstelle der Entwurfsdaten übereinstimmt. Bei ausschließlicher Verwendung des in Kapitel 6 verwendeten Datenformats wäre diese allgemeine Vorgehensweise nicht notwendig gewesen. Allerdings sichert diese Vorgehensweise die einfache Austauschbarkeit des Datenformats.

Eine Implementierung besitzt genau eine Schnittstelle. Dies stellt eine Einschränkung dar, da ein System abhängig von der Modellierungsgenauigkeit und dem Abstraktionsgrad unterschiedliche Schnittstellen besitzen kann. Beispielsweise kann ein Addierer je nach Abstraktionsgrad zwei Eingänge und zwei Ausgänge vom Typ 'Integer' (Ergebnis und Übertrag) haben oder in einer niedrigeren Beschreibungsebene zwei Eingänge und zwei Ausgänge mit vier Bits. Bild 5-4 zeigt zwei äußere Schnittstellen I1_Add4 und I2_Add4. Auch ein Syntheseschritt kann die äußeren Schnittstellen verändern. Die Entwurfsumgebung muß somit einen Mechanismus vorsehen, daß eine Implementierung mehrere äußere Schnittstellen referenzieren kann.

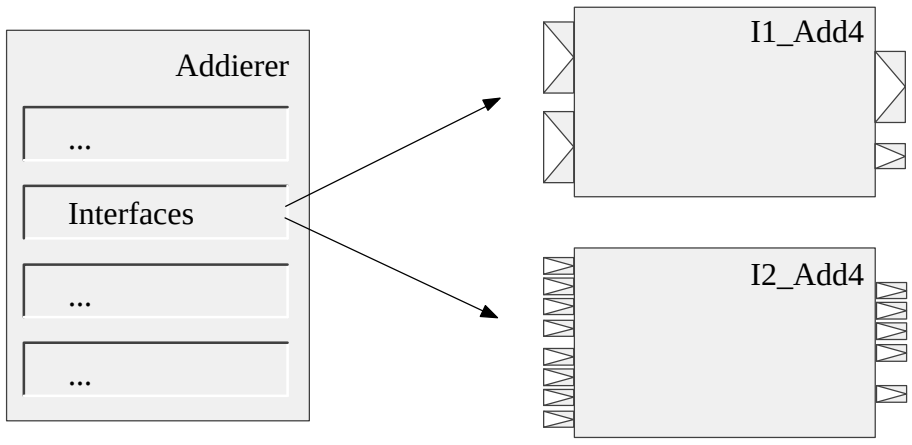


Bild 5-4: Äußere Schnittstellen des Modul-Objekts Summation4

Dieses Element ist in Bild 5-4 auf der linken Seite dargestellt (Addierer). Um Systembeschreibungen mit mehreren Implementierungen und Schnittstellen aufnehmen zu können, wird eine weitere Klasse eingeführt, die als Modul-Klasse (engl. *module class*) bezeichnet wird. Die Elemente der Modul-Klasse (Bild 5-5) sind im folgenden beschrieben:

- *Name*: Über den eindeutigen Namen wird der Zugriff auf die Module Class vorgenommen.
- *Specification*: In der Spezifikation ist eine textuelle Beschreibung des Verhaltens des referenzierten Objekts einer Module Class hinterlegt.
- *Interfaces*: Die hier hinterlegte Liste umfaßt Verweise auf die von dem referenzierten Objekt einer Module Class unterstützten Schnittstellen.
- *Implementations*: Liste von Verweisen auf die Implementierungen. Ein referenziertes Objekt einer Module Class kann dabei mehrere Implementierungen besitzen.

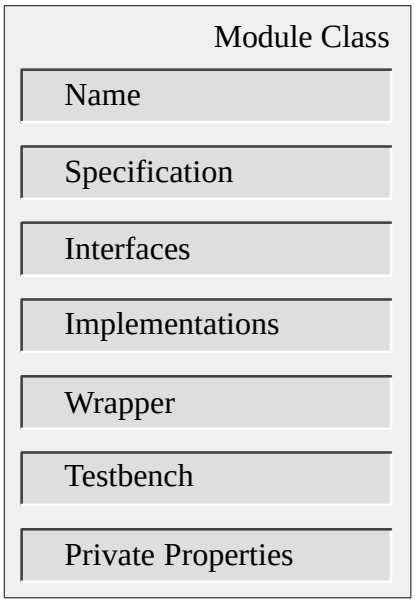


Bild 5-5: Elemente der Module Class

- *Wrapper*: Liste der Vermittler (engl. *wrapper*), die eine Implementierung mit der Schnittstelle verbinden (siehe Kapitel 5.4.3).
- *Testbenchs*: Die hier hinterlegte Liste enthält die Modellierung der Umgebung des referenzierten Objekts einer Module Class. Mit Hilfe einer Testbench können Implementierungen simuliert werden. Die Angabe von Testbenchs ist optional.
- *Private Properties*: Benutzerdefinierte Eigenschaften zur erweiterten Beschreibung eines referenzierten Objekts einer Module Class.

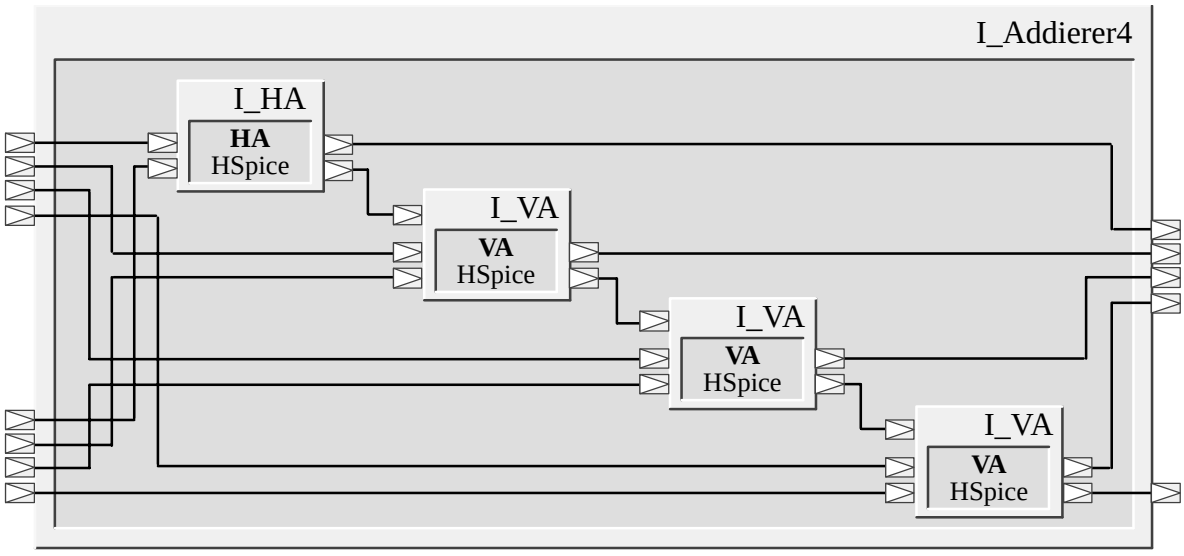


Bild 5-6: Implementierung eines Addierers

Der beschriebene Addierer ist in Bild 5-6 dargestellt. Die Schnittstelle I_Addierer enthält als Implementierung eine strukturale Modellierung, die aus den Schnittstellen eines Halbaddierers (I_HA) und drei Volladdierern (I_VA) besteht. Mit Hilfe von Verbindungen werden die inneren Schnittstellen zu einem nicht kaskadierbaren 4-Bit Ripple-Carry Volladdierer zusammengesetzt. Darüber hin-

aus ist zu erkennen, daß die Implementierung aller inneren Schnittstellen mit dem Werkzeug HSpice vorgenommen wurde.

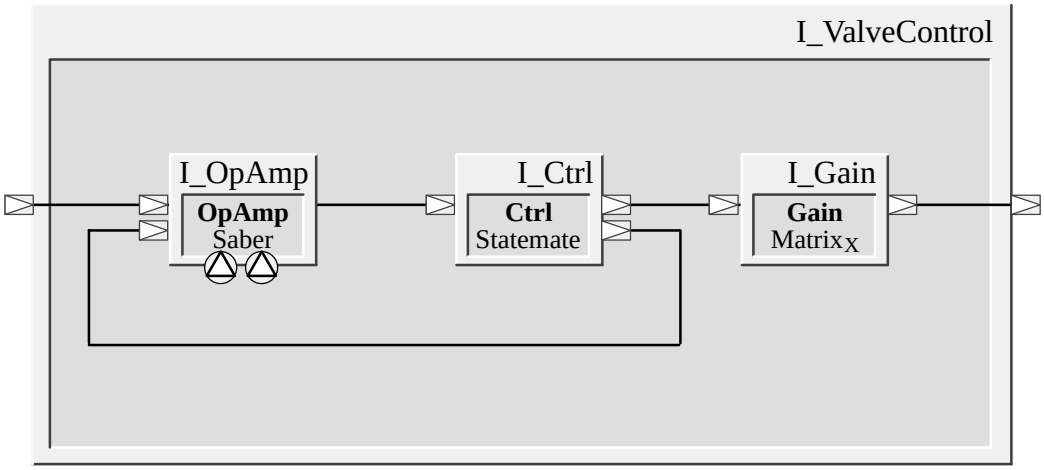


Bild 5-7: Implementierung eines diskret/kontinuierlichen Modells

Bild 5-7 zeigt die Implementierung von diskret/kontinuierlichen Modellen mit der Entwurfsumgebung für die Ansteuerung eines Ventils. Die zu regelnde kontinuierliche Größe wird über einen Operationsverstärker, der in dem Entwurfswerkzeug Saber (Werkzeug zur Mixed-Signal Modellierung mit der Beschreibungssprache MAST) implementiert ist, an einen diskreten Zustandsautomaten geleitet, der in STATEMATE™ implementiert ist. Der Zustandsautomat beeinflußt das eingehende Signal und führt die Ansteuerung des Ventils über einen Verstärker durch, der mit MATRIX$_X$™ implementiert ist.

5.4.3 Wrapper

Um die Umsetzung von Signalen zwischen Interface und einer Implementierung vorzunehmen, wird ein Vermittler (engl. *wrapper*) eingesetzt. Der Vermittler umhüllt die Schnittstelle einer Implementierung und setzt Signale, die durch die Ports einer Implementierung führen, in Signale um, die durch die Ports der Implementierung führen. Diese Vermittlertätigkeit ist insbesondere dann wichtig, wenn fein-granulare Implementierungen eingebettet werden. Beispielsweise müssen die Integer Signale eines Ports des Addierers von Bild 5-8 auf die zugehörigen bitweisen Signale der Implementierung umgesetzt werden.

Die Vermittler-Klasse (engl. *wrapper class*) besitzt den folgenden Aufbau:

- *Name*: Über den eindeutigen Namen wird der Zugriff auf die Wrapper Class vorgenommen.
- *Interfaces*: Die hier hinterlegte Liste umfaßt Verweise auf die von dem referenzierten Objekt einer Wrapper Class unterstützten Schnittstellen. Die zugehörige Implementierung ist im nächsten Element abgelegt.
- *Implementations*: Verweis auf eine Implementierung, bzw. auf das von der Implementierung referenzierte Interface. Durch diesen Aufbau ist es möglich, zwei Implementierungen, die dasselbe Interface referenzieren, mit einem Wrapper zu beschreiben.
- *Translators*: Verweis auf das Übersetzungsmodul, das die Daten der Ports der äußeren Schnittstelle auf die Ports der inneren Schnittstelle zuweist oder konvertiert.

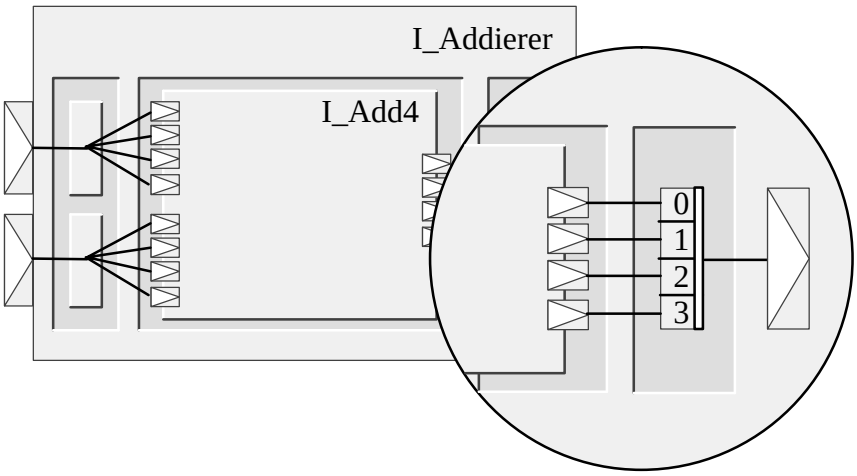


Bild 5-8: Wrapper mit zugehöriger Übersetzungsregel

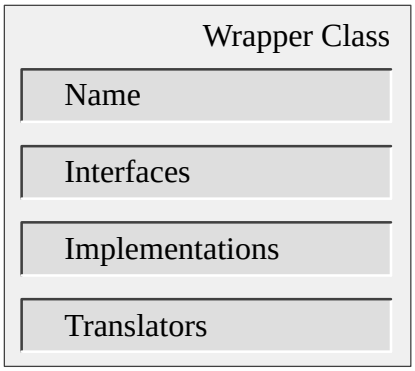


Bild 5-9: Elemente der Wrapper Class

Durch die Vermittler wird die in Kapitel 5.4.2 beschriebene Einbettung mehrerer Implementierungen in eine Schnittstellendefinition sehr mächtig. Insbesondere das für Rapid Prototyping wichtige Mitführen von Systembeschreibungen auf unterschiedlichen Abstraktionsebenen wird auf diese Weise unterstützt. Bild 5-10 zeigt die Auswahl zweier Implementierungen (Saber und CDIF) innerhalb einer Schnittstelle. Das Datenformat CDIF wird in Kapitel 6 behandelt.

Bild 5-8 zeigt die Umsetzung zwischen Implementierung und Schnittstelle in Form einer grafischen Modellierung. Die notwendigen Regeln können auch mit einer höheren, objekt–orientierten Programmiersprache realisiert werden. Der folgende Ausschnitt liefert dafür ein Beispiel:

```
 1    class TranslationRule1 extends TranslationRule
 2    { ...
 3    // Inner ports y0, y1, y2, y3 to outer port y
 4    public void rule1(Bit iy0, Bit iy1, Bit iy2, Bit iy3, Integer iy)
 5      {
 6      oy.setValue( iy3.getValue() << 3 +
 7                   iy2.getValue() << 2 +
 8                   iy1.getValue() << 1 +
 9                   iy0.getValue() << 0 );
10      }
11    ...
12    } // End TranslationRule1
```

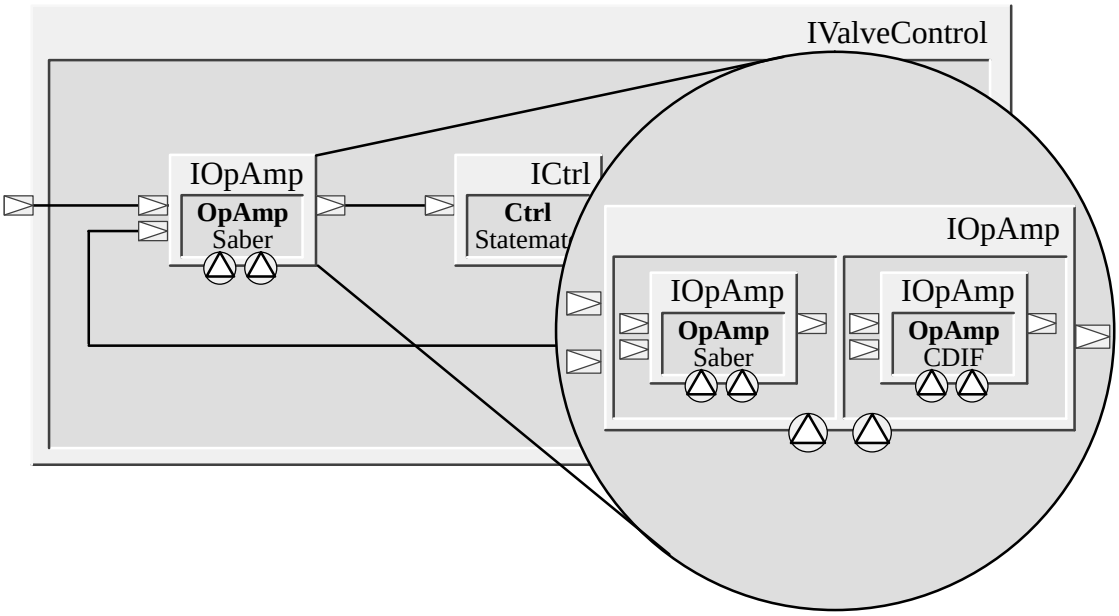


Bild 5-10: Mehrere Implementierungen in einer Schnittstelle

5.4.4 Konfigurationen

Um eine Zuweisung von Implementierung und Vermittler vornehmen zu können, müssen zusätzlich Konfigurationen (engl. *configurations*) eingeführt werden. Da Implementierungen eine Schnittstelle nur referenzieren, besteht die Möglichkeit, daß zwei Implementierungen dasselbe Interface referenzieren. Dies kommt dann vor, wenn sich die Implementierungen in ihrer Beschreibung unterscheiden, jedoch dieselbe Anzahl und Art von Signalen verwenden. Die Angabe eines Vermittlers alleine reicht also nicht aus, um einen Zusammenhang zwischen Schnittstelle, Vermittler und Implementierung herzustellen. Die für diesen Zweck erstellten Konfigurationen stellen eine eindeutige Zuordnung aller Komponenten eines Moduls dar. Mit dieser eindeutigen Konfiguration kann eine Simulation, Analyse, Codeerzeugung und Implementierung durchgeführt werden.

Die Wiederverwendbarkeit der Objekte liegt durch den oben beschriebenen Aufbau zwangsweise vor. Durch die Eigenschaften Kapselung und Polymorphismus eignet sich dieser Ansatz in nahezu idealer Weise für den Systementwurf. Die Trennung von Schnittstelle und Implementierung zusammen mit dem Polymorphismus erlaubt die Beschreibung von Systemen auf verschiedenen Abstraktionsebenen mit identischen Schnittstellen. Zusätzlich gewährleistet die Kapselung einen änderungsfreundlichen Entwurf, da sich die Modifikationen innerhalb eines Objekts wegen der festgelegten Schnittstelle nicht auf die Umgebung auswirken. Um eine einheitliche Datenhaltung zu gewährleisten, werden im nächsten Kapitel standardisierte Datenformate auf Eignung untersucht.

6 Abstraktion der Modelldaten

6.1 Einführung

Im Bereich der Entwurfswerkzeuge für elektronische Systeme ist zu beobachten, daß jedes Entwurfswerkzeug über spezielle Möglichkeiten zur Bearbeitung eines bestimmten Problems verfügt. Für das jeweilige Problem bietet das Entwurfswerkzeug leistungsstarke, optimierte Lösungen an. Solange sich die Probleme in kleinen, überschaubaren Bereichen eines Entwurfsbereichs bewegen, kann man mit den meisten heutigen Werkzeugen komfortabel arbeiten. Möchte man jedoch kompliziertere Abläufe modellieren, die nach mehreren Entwurfsmethoden verlangen, so ist eine Verbindung von Entwurfswerkzeugen notwendig. Diese kann entweder als feste interne Kopplung oder als lose externe Kopplung realisiert werden. Die feste Kopplung hat den Nachteil, daß sich hiermit nur spezielle Werkzeuge, die meist von einem Hersteller angeboten werden, verbinden lassen und ein Hinzufügen weiterer Werkzeuge nahezu unmöglich ist. Der Anwender ist jedoch an einer 'best–of–breed' Lösung interessiert, die ihm die freie Auswahl seiner Werkzeuge ermöglicht.

Dies führte wie bereits in Kapitel 4 erläutert Anfang der neunziger Jahre zu den ersten externen Kopplungen von Entwurfswerkzeugen. Bei der Durchführung dieser Kopplungen [ChEr93] stieß man jedoch schnell auf die Grenzen, da die Werkzeuge auf die Lösung eines bestimmten Problems ausgerichtet sind. So "verstanden" sich die Werkzeuge untereinander nicht, was zu erheblichen Anstrengungen im Bereich der Datenkonvertierung führte. Noch 1997 wurde von [Balz97] festgestellt: *Die Integrierbarkeit in CASE–Plattformen ist – falls überhaupt – nur in herstellereigene gegeben, sonst so gut wie gar nicht anzutreffen.* Zusätzlich führten Verfeinerungen der Entwurfstechnik wie der Wunsch zu Gesamtanalyse, Gesamtsimulation, Rapid Prototyping und Hardware–in–the–Loop zu Anforderungen, die keine einfache Werkzeugkopplung zu leisten im Stande war.

Nach [ThNe92] können vier Arten der Integration eines CASE–Werkzeugs identifiziert werden. Integration wird dabei definiert, daß Dinge als Teile eines kohärenten Ganzen funktionieren. Die Ziele der verschiedenen Integrationsarten lauten:

- **Oberflächen–Integration:** Verbesserung der Effizienz der Werkzeugbedienung innerhalb einer CASE–Umgebung durch Reduktion der kognitiven Belastung.
- **Steuerungs–Integration:** Flexible Kombination von Umgebungsfunktionen in Abhängigkeit von Entwicklungspräferenzen und getrieben durch den unterliegenden Prozeß, den die CASE–Umgebung unterstützt.
- **Prozeß–Integration:** Sicherstellung, daß die Werkzeuge effektiv zusammenarbeiten, um einen definierten Prozeß zu unterstützen.
- **Daten–Integration:** Sicherstellung, daß alle Informationen in der CASE–Umgebung als konsistentes Ganzes verwaltet werden.

Dabei stellt die Daten–Integration den bislang am wenigsten gelösten Bereich dar. Bei Daten–Integration werden nach [RaSt92] nochmals drei Stufen unterschieden:

Die **Black–Box Integration** (siehe Bild 6-1, links) stellt die schwächste Daten–Integrationsart dar. Dabei arbeitet das CASE–Werkzeug weiterhin isoliert auf seinen eigenen Datenstrukturen und benutzt dieselben Datenzugriffe wie im nicht–integrierten Fall. Durch eine Check–In/Check–Out Technik werden die benötigten Daten aus dem Repository (Verwahrungsort) zur Verfügung gestellt. Das Repository dient zur Datenverwaltung und kann beispielsweise unter Verwendung eines Dateisystems oder eines Datenbanksystems realisiert werden. Diese Technik erlaubt eine Zugriffs– und Versionskontrolle. Vorteile dieser Integrationsart liegen darin, daß ein Zugriff auf werkzeugübergreifende Daten über das Repository erfolgen kann und daß keine Kenntnisse über die Interna des Werkzeugs erforderlich sind, so daß fast alle gängigen Werkzeuge für diese Integrationsart benutzt werden können. Unter Interna werden dabei Rückschlüsse auf den inneren Aufbau des CASE–Werkzeugs, ungeschützte Veröffentlichung von Programm–Code oder weitere Informationen, die der CASE–Werkzeug Hersteller nicht öffentlich machen möchte, zusammengefaßt. Allerdings sind bei dieser Daten–Integrationsart die anderen Integrationsarten nur schwer oder überhaupt nicht zu erfüllen.

Die **Grey–Box Integration** (siehe Bild 6-1, Mitte) stellt einen mittleren Daten–Integrationsgrad dar. Hierbei werden die Datenzugriffe eines CASE–Werkzeugs durch Repository Dienste ersetzt. Eine Interoperabilität zwischen Werkzeugen und eine Unterstützung der Datenintegrität wird ermöglicht. Kenntnisse über Interna des Werkzeugs sind erforderlich, allerdings können diese Kenntnisse in einem Modul für Zugriffe auf externe Daten gekapselt werden. Die eigentlichen Bestandteile der Werkzeug–Implementierung müssen nicht offengelegt werden. Mit dieser Daten–Integrationsart können alle Integrationsarten außer der Oberflächen–Integration erreicht werden.

Die **White–Box Integration** (siehe Bild 6-1, rechts) stellt den stärksten Daten–Integrationsgrad dar und ist als ideale Integrations–Lösung zu betrachten. Für alle verwendeten Werkzeuge und den Prozeßablauf wird ein einheitliches Datenschema feiner Granularität definiert. Auf dieses Datenschema setzen über die Zugriffs– und Versionsverwaltung des Repository alle CASE–Werkzeuge auf. Somit können verschiedene CASE–Werkzeugs simultan und kooperierend auf gemeinsame Datenbestände zugreifen, so daß es zu effizienter und enger Kooperation kommt, ohne daß Datenbestände kopiert und transformiert werden müssen. Die Konsistenz und Sicherheit der Daten ist einfach zu gewährleisten. Auch bei genauer Einhaltung der Konventionen des Repository müssen für diese Art der Daten–Integration Interna des Werkzeugs offenstehen (im minimalsten Fall betrifft dies die Schnittstelle zum Repository). Gibt es Abweichungen im Datenschema, müssen die Werkzeuge grundlegend geändert werden. Mit dieser Daten–Integrationsart stehen dafür jedoch alle Integrationsarten zur Verfügung.

Die anzustrebende Lösung ist also die White–Box Integration. In [Balz97] wird jedoch berichtet, daß die White–Box Integration sich "*aber auf neu entwickelte Werkzeuge beschränken wird und auch nur dann Erfolg haben wird, wenn Normen für konzeptionelle Schemata und dazugehörige Repository–Schnittstellen geschaffen werden*".

Da die White–Box Integration die oben erwähnten Nachteile hat, wird in dieser Arbeit eine verbesserte Grey–Box Integration vorgeschlagen, die schrittweise in eine White–Box Integration übergehen kann. Diese besondere Form der Integration wird im folgenden als **Color–Box Integration** bezeichnet. Wie bereits in Kapitel 4 gezeigt, existiert ein allen Werkzeugen gemeinsames Repository.

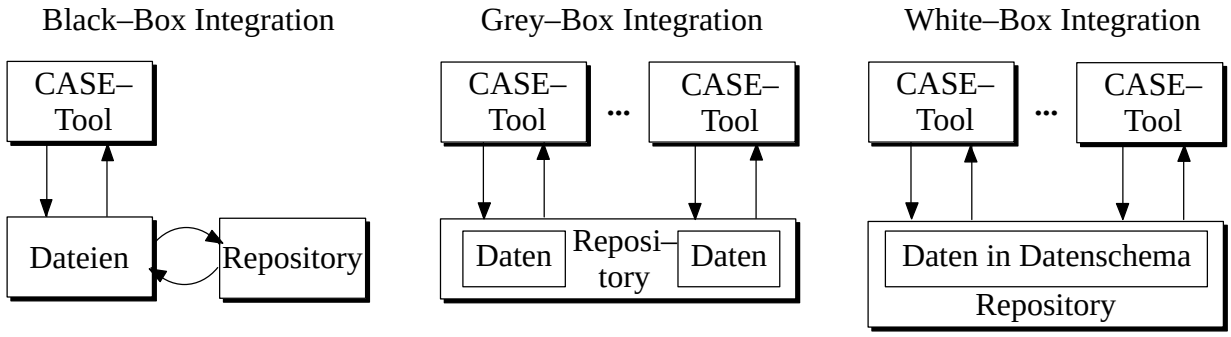


Bild 6-1: Stufen der Daten–Integration

Dieses Repository wird jedoch im Gegensatz zur Grey–Box Integration von einem CASE–Werkzeug verwaltet, das auch die strukturelle Verbindung der einzelnen Entwurfsmodule modelliert. Durch diese Zusammenfassung in einem Werkzeug kann eine sehr enge Kopplung erreicht werden, die sich durch Verwendung eines gemeinsamen, einheitlichen Datenschemas bei verteilten Werkzeugen auszeichnet. Die einheitliche Verwaltung der zur Verhaltens–Modellierung eingesetzten CASE–Werkzeuge wird mittels Kapselung der einzelnen CASE–Werkzeuge erreicht, die nach außen über Data Ports angesprochen werden können. Viele der heutigen Werkzeuge können auch von außen fremdbestimmt werden, was zur weiteren Verbesserung der Integration ausgenutzt werden kann. Die Vorgehensweise erlaubt die Unterstützung der Datenintegrität, wobei keine Interna der CASE–Werkzeuge offengelegt werden müssen. Somit stellt die Color–Box Integration einen beachtlichen Fortschritt zur White–Box Integration dar, da bereits bestehende CASE–Werkzeuge eng miteinander verknüpft werden können, ohne die sonstigen Nachteile der White–Box Integration aufzuweisen. Der in Kapitel 4 vorgestellte Ansatz kann von den vier möglichen Integrationsarten drei erfüllen. Die Oberflächenintegration kann erst durch Einsatz eines Meta–CASE–Werkzeugs (siehe Kap. 6.2) ermöglicht werden.

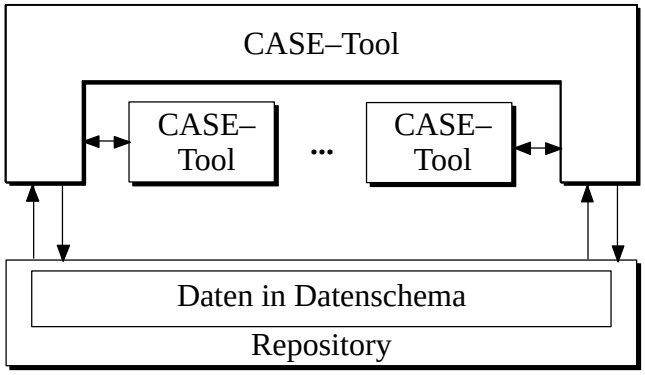


Bild 6-2: Color–Box Integration

Allgemein stellt sich beim Einsatz eines Datenformats die Frage, welches Datenformat für die Problemstellung am besten geeignet ist. Während beispielsweise die Aufstellung eines Standards im Bereich von Pixelgrafiken sich noch als verhältnismäßig einfach erweist, ist eine Standardisierung im weiten Bereich der Entwurfswerkzeuge ungleich schwieriger. Folgende Kriterien für Datenformate bilden dabei eine Bewertungsgrundlage:

- **Anwendungsbereich:** Wurden die in einem Anwendungsbereich auftretenden Daten ermittelt, muß das Datenformat die Möglichkeit bieten, diese in vollem Umfang beschreiben zu können.

- **Technische Qualität:** Datenformate können Daten auf unterschiedliche Art beschreiben. Dabei muß ein Datenformat eine eindeutige Repräsentation der Daten unterstützen. Darüber hinaus muß gewährleistet sein, daß eine Änderung eines Teils des Datenformats nicht Änderungen vieler anderer Teile mit sich bringt.
- **Erweiterbarkeit:** Um zukünftigen Anforderungen zu genügen, muß ein Datenformat erweiterbar sein. Dies muß jedoch verträglich mit nicht erweiterten Versionen des Datenformats sein, um bereits bestehende Beschreibungen nicht unbrauchbar zu machen.
- **Qualität der Dokumentation:** Die Dokumentation eines Datenformats muß so gefaßt sein, daß die Semantik eindeutig definiert ist und keine Interpretationen zuläßt. Nur so kann ein fehlerfreier Datenaustausch gewährleistet werden.
- **Integration:** Ein Datenformat darf nicht nur die gerade aktuellen Probleme lösen, sondern sollte sich nahtlos in Datenformate anderer Bereiche eingliedern.
- **Akzeptanz:** Ein Datenformat hat wenig Sinn, wenn niemand damit arbeiten will. Neben den funktionalen Anforderungen muß daher auch die Unterstützung von mehreren Werkzeugen, bzw. deren Anwendern sichergestellt sein.

Im Laufe der letzten Jahre wurden von verschiedenen Organisationen und Firmen Datenformate entwickelt, die für den Austausch von Daten zwischen CASE–Werkzeugen konzipiert wurden. Da diese Datenaustauschformate jedoch speziell für einzelne herstellerspezifische Werkzeugfamilien entwickelt wurden, oder von Organisationen erstellt wurden, die nur einen einzigen Bereich betrachteten, konnten sich Datenaustauschformate in dem Bereich von CASE–Werkzeugen nicht durchsetzen.

Cadre-CDIF

Ein Datenaustauschformat für eine hersteller–spezifische Werkzeugfamilie ist mit Cadre–CDIF der Firma CADRE gegeben (das nicht mit dem CASE Data Interchange Format CDIF verwechselt werden darf). Dieses Datenaustauschformat bietet die Möglichkeit zum Austausch aller in dieser Produktfamilie anfallenden Daten, unter anderem auch die Beschreibung von diskreten und kontinuierlichen Modellen. Der Funktionsumfang von Cadre–CDIF umfaßt jedoch keine vollständige Beschreibung des diskret–kontinuierlichen Bereichs, sondern ist eng an den Funktionsumfang der Cadre–Werkzeuge orientiert. Aus diesem Grund können nicht alle diskreten und kontinuierlichen Systeme abgebildet werden. Der Austausch von Daten unter Verwendung dieses Datenformats wurde anhand der Entwurfswerkzeuge STATEMATE™ und MATRIX$_X$™ untersucht [Erns94]. Die dabei festgestellten Mängel von Cadre–CDIF ließen sich nur durch eine spezielle Anpassung des Funktionsumfangs beheben, wodurch allerdings die Basis für den Datenaustausch zwischen den Werkzeugen verlassen werden mußte. Da die Möglichkeiten zu individuellen Erweiterungen nicht gegeben sind, konnte sich Cadre–CDIF nicht durchsetzen. Die Firma CADRE ist mittlerweile in der Firma Cayenne Software aufgegangen. Cayenne Software unterstützt das in Kapitel 6.3 vorgestellte Datenaustauschformat CDIF.

CAMeL

Ein weiteres Datenaustauschformat wurde am Mechatronik Labor der Universität Paderborn (MLaP) mit dem Entwurfswerkzeug CAMeL (Computer–Aided Mechatronic Laboratory) entwikkelt. CAMeL stellt eine offene CAE–Entwurfsumgebung zur Modellierung und Simulation mecha-

tronischer Systeme dar und kann mechanische, elektronische und Software–Komponenten [JäKl91] darstellen. Die Daten werden zuerst fachspezifisch modelliert und in eine mathematische Beschreibung in Form von Differentialgleichungen überführt. Die dabei verwendeten Differentialgleichungen eignen sich zwar zur Repräsentation von zeit–kontinuierlichen Modellierungen, können jedoch nicht zur Repräsentation von State/Event oder objekt–orientierten Modellierungen herangezogen werden. Eine Rekonstruktion von Daten einer niederen Ebene auf eine höhere Ebene ist nicht mehr möglich.

AIRE

Das AIRE (Advanced Intermediate Representation with Extensibility / Common Environment) Konsortium [Will97] beschäftigt sich mit der Interoperabilität von einzelnen Komponenten, bzw. bereits kompilierten Modellen von Hardwarebeschreibungssprachen (z.B. VHDL, Verilog). Hierzu wurde eine Internal Intermediate Representation (IIR), die eine speicherbasierte Klassenstruktur und Methoden zur Interaktion zur Verfügung stellt, und eine File Intermediate Representation (FIR) eingeführt, die das entsprechende File Format zur Verfügung stellt, um Werkzeuge lesend und schreibend auf teilweise kompilierte HDL Modelle zugreifen zu lassen. Dabei ist sowohl der Aufbau von Werkzeugen vorgesehen, die auf AIRE basieren, als auch die Standardisierung des Formats.

Das Format ist allerdings auf Hardwarebeschreibungssprachen zugeschnitten und soll rein datei–basierend ablaufen. Eine Verbindung zu anderen Standards, beispielsweise der objekt–orientierten Programmierung, ist nicht vorgesehen. Es ist somit zu befürchten, daß hier an einer weiteren Insellösung gearbeitet wird.

SLDL

Die System–Level Design Language (SLDL) [Schu98] wurde im Oktober 1996 von Mitgliedern des EDA Industry Council ins Leben gerufen. Der hauptsächliche Grund für diese Sprache liegt in Schwierigkeiten, die bei einem System–on–a–Chip Design auftreten. Allerdings kann diese Sprache letztendlich auch allgemein für verteilte elektronische Systeme eingesetzt werden. Als erster Schritt soll SLDL in Ergänzung zu VHDL als Beschreibung von Randbedingungen eingesetzt werden. In einem zweiten Schritt soll die Sprache mit Komponenten zur Beschreibung von Verhalten und Struktur ausgestattet werden. Zu diesem Zeitpunkt sollen mit SLDL Sprachen wie VHDL, Verilog, Esterel aber auch C/C++ und Java über 'Bridging'–Konzepte verbunden werden können. Der Aufbau dieser Konzepte ist derzeit noch nicht bekannt. Erste Prototypen der Sprache sollen ab dem Jahr 2000 erhältlich sein. Der Sprachstandard ist allerdings noch ganz am Anfang und schwer einschätzbar. Die Ideen, die mit dieser Sprache verbunden sind, versprechen Abhilfe für eine ganze Reihe von Problemen:

- Unterstützung von frühen, abstrakten Modellen von Mikroprozessoren und Echtzeitbetriebssystemen.
- Unterstützung von semantischer Konsistenz für Interfaces auf unterschiedlichen Abstraktionsebenen.
- Beschreibung einer Komponente ohne Angabe von Implementierungsdetails.
- Gleichzeitige Einarbeitung von Informationen auf mehreren Abstraktionsebenen.
- Unterstützung von semantisch getrennten Bereichen und Bibliotheksbildung.

- Unterstützung von verschiedenen Sichten auf eine Beschreibung. Nur die wirklich gewünschte Information soll dem Anwender zur Verfügung gestellt werden.

Zum gegenwärtigen Zeitpunkt kann noch keine abschließende Bewertung von SLDL vorgenommen werden. Ein offenes ASCII Format von SLDL, das sowohl für direkte Benutzereingabe als auch für automatische Generierung aus EDA–Werkzeugen geeignet ist, muß in Zukunft ebenfalls beachtet werden.

Um die oben genannten Kriterien zu erfüllen, eignen sich die bisherigen Ansätze nur bedingt. Eine neue Form der Datenmodellierung, genannt Meta–Modell Beschreibung, die sich bislang hauptsächlich im Bereich der Programmiersprachen [BoJa97] durchgesetzt hat, kann auch in dem Bereich des Entwurfs elektronischer Systeme eine Lösungsmöglichkeit darstellen und eine ideale Verbindung zu objekt–orientierten Sprachen bieten.

6.2 Meta-Modell Beschreibungen

Um einen Austausch von Informationen zu ermöglichen, ist es notwendig, diese Informationen in einem Modell zu beschreiben. Will z.B. ein Augenzeuge einem Polizisten den Ablauf eines Unfalles schildern, so wird er versuchen, die Vorgänge mit Hilfe von sprachlichen Konstrukten zu beschreiben. Er erzeugt somit ein Modell des Unfalles und übermittelt dieses Modell dem Polizisten. Dieser wird versuchen, anhand des Modells die ursprünglichen Informationen, also den tatsächlichen Ablauf des Unfalles, zu rekonstruieren. Damit dieser Informationsaustausch funktionieren kann, müssen alle Beteiligten die gleiche Sprache sprechen. Sie benötigen demnach den gleichen Wortschatz und die gleiche Grammatik. Die Problematik bei dieser Art der Kommunikation ist die korrekte und eindeutige Beschreibung der Zusammenhänge. So ist die Aussage des Satzes ”Ein Mann geht zu einer Bank” nicht eindeutig, da der Begriff Bank zum einen für ein Kreditinstitut aber auch für eine Sitzmöglichkeit verwendet werden kann.

Der Informationsaustausch zwischen verschiedenen Programmen folgt diesem Schema. Der Benutzer eines CASE–Werkzeugs, der das Verhalten eines diskreten Systems beschreiben will, muß dieses Verhalten mit Hilfe einer Modellierungstechnik erstellen, die durch das CASE–Werkzeug zur Verfügung gestellt wird. Unterstützt ein Werkzeug nur Statecharts, so wird er das Verhalten des diskreten Systems in Form von Statecharts modellieren. Um eine sinnvolle Analyse und Simulation des Systems durchführen zu können, müssen sowohl das CASE–Werkzeug wie auch der Benutzer dieses Modell gleichermaßen interpretieren. Man benötigt demnach ein weiteres Modell, das als *Meta–Modell* bezeichnet wird, das nicht die Informationen, sondern die Modellierungselemente beschreibt. In einem Meta–Modell werden demnach Elemente definiert, die zur Modellierung bestimmter Informationen zur Verfügung stehen. Der Augenzeuge verwendet bei der Beschreibung (also bei der Modellierung) des Unfalles sprachliche Konstrukte, die er nach den Regeln einer Grammatik (Semantik) und eines Wortschatzes (Syntax) bildet. Unter *Syntax* wird im allgemeinen eine Menge von Wortklassen (z.B. Nomina, Adjektive, etc.) und syntaktischen Konstituenten (z.B. Verbalgruppe, Satz, etc.) sowie Vorschriften, nach denen aus Wortklassen syntaktische Konstituenten gebildet werden dürfen, verstanden. Die Vorschriften definieren dabei, welche Wortklassen mit welchen anderen kombinierbar sind und welche syntaktischen Eigenschaften Wortklassen einer syntaktischen Konstituente (unter Umständen auch im Kontext anderer Konstituenten) haben müssen. Die *Semantik* hingegen stellt die Beziehung zwischen dem Sprachzeichen, dem Wort und seinem Bedeutungsgehalt dar. Dies sind die Beziehungen zur realen Umwelt, die in den Bedeutungen

begrifflich gefaßt werden [GaKn95]. Die Grammatik und der Wortschatz bilden somit das Meta–Modell des Informationsaustauschs zwischen den beiden Personen.

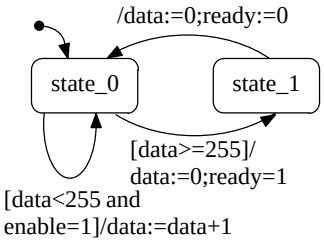
Werden die Modelle zwischen zwei CASE–Werkzeugen ausgetauscht, so müssen beide CASE–Werkzeuge das Meta–Modell kennen, um das zu übermittelnde Modell richtig interpretieren zu können. Ist das Meta–Modell nicht bekannt, so kann das CASE–Werkzeug nicht am Datenaustausch beteiligt werden. Um diese Problematik zu umgehen, besteht die Möglichkeit, das Meta–Modell in den Datenaustausch zu integrieren und somit jedem CASE–Werkzeug zur Verfügung zu stellen. Dazu ist es allerdings notwendig, ein Meta–Meta–Modell zu definieren. Dieses Meta–Meta–Modell beschreibt die Elemente, die zur Bildung eines Meta–Modells verwendet werden können. Somit ist es möglich, verschiedene Modellierungstechniken in verschiedenen Meta–Modellen zu definieren. Zwei CASE–Werkzeuge, denen das gleiche Meta–Meta–Modell bekannt ist, können dann, ohne Kenntnis der verwendeten Modellierungstechnik, ein übermitteltes Modell analysieren, da die Konstrukte im Meta–Modell beschrieben sind. Für solche Modelle ist es somit notwendig, neben dem Modell auch das Meta–Modell zu übertragen.

Meta–Modell Beschreibungen erlauben somit die Einführung einer weiteren Abstraktionsebene für die CASE Datenmodellierung und führen deswegen weg von den Implementierungsdetails der Werkzeugintegration. Dabei wird eine Partitionierung in verschiedene parallele Untergebiete vorgenommen, die einzeln leichter zu lösen sind als das Gesamtproblem. Diese Aufteilung fördert dabei die oben genannten Kriterien technische Qualität, Erweiterbarkeit und Integration, was letztendlich zu einer wachsenden Akzeptanz führt. Die weiteren Kriterien Anwendungsbereich und Qualität der Dokumentation hängen stark von der jeweiligen Implementierung der Meta–Modell Beschreibung ab.

Obwohl Meta–Modell Beschreibungen bereits seit ungefähr zehn Jahren existieren, ist erst mit Aufkommen des Intra– und Internets sowie der weiten Verbreitung der Unified Modeling Language UML [BoJa97] das Problem der Daten–Modellierung angegangen worden. Während CASE–Werkzeuge der ersten Generationen lediglich eine einzige Modellierungsart zuließen, werden zukünftige CASE–Werkzeuge den verzahnten Einsatz von mehreren Modellierungsarten zulassen. Dabei werden im Moment auch sogenannte Meta–CASE–Werkzeuge angedacht [IsLa97], die kein festes Meta–Modell mehr besitzen, sondern die erst durch Konfiguration des Benutzers mit einem speziellen Meta–Modell eine Modellierung zulassen. Das in Kapitel 4 vorgestellte Konzept unterstützt diese Modellierung in idealer Weise, da dann keine zusätzlichen Komponenten zur Übersetzung in ein gemeinsames Datenformat mehr notwendig sind und der Implementierungsaufwand als gering einzustufen ist.

Aus fortgeschrittenen Modellierungssprachen wie Modelica und VHDL–AMS, die auf einer reinen Syntax/Encoding Ebene definiert sind, können zwar entsprechende Meta–Modelle durch Reverse–Engineering gewonnen werden. Diese lassen jedoch auch nach Aufstellung eines Meta–Modells modellierungs–inhärente Mehrdeutigkeiten zu. Bild 6-3 zeigt die Darstellung eines Statecharts und zwei von mehreren möglichen Repräsentationen in VHDL–AMS [BiLa93]. Durch die unterschiedliche Repräsentation desselben Statecharts gestaltet sich eine Weiterverarbeitung auf Basis dieser Daten als schwierig. Manche Werkzeuge, die nur für die Synthese von VHDL ausgelegt sind, erfordern bereits eine ganz spezielle Beschreibung unter Einhaltung von Richtlinien, um eine erfolgreiche Synthese durchführen zu können (diese Einschränkungen sind auch bei VHDL–AMS zu beobachten). An dieser Stelle sei angemerkt, daß durch die obige Vorgehensweise Modellierungsspra-

chen zur *Datenmodellierung* eingesetzt werden. Damit wird jedoch der eigentliche Zweck der Modellierungssprache überschritten.




```
WAIT UNTIL clk ='1';
CASE current_state IS
 WHEN state_0 =>
   IF data < 255 THEN
    IF enable = '1' THEN
     data <= data + 1;
    END IF;
    current_state := state_0;
   ELSE
    data <= 0;
    ready <= '1';
    current_state := state_1;
   END IF;
 WHEN state_1 =>
   data <= 0;
   ready <= '0';
   current_state := state_0;
END CASE;
```

```
data <= 0;
ready <= '0';
WAIT UNTIL clk='1';
WHILE (data < 255) LOOP
 IF (enable='1') THEN
  data <= data + 1;
 END IF;
 WAIT UNTIL clk='1';
END LOOP;
data <= 0;
ready <= '1';
WAIT UNTIL clk='1';
```

Bild 6-3: Darstellung eines Statecharts und unterschiedliche Repräsentationen in VHDL

Im Gegensatz zu den oben genannten Modellierungssprachen haben native Meta–Modell Beschreibungen diese Nachteile nicht. Im weiteren soll nun auf ein CASE Datenaustauschformat eingegangen werden, das bereits mit Hilfe einer Meta–Modell Beschreibung erstellt wurde und über ein Meta–Meta Modell verfügt.

6.3 Systemmodellierung mit CDIF

6.3.1 Einführung

Die Electronic Industry Association (EIA) erkannte bereits früh den Bedarf an einem leistungsfähigen Datenaustauschformat im Bereich der CASE–Werkzeuge. Im Jahr 1987 wurde das CDIF–Komitee gegründet, das das CASE Datenaustauschformat CDIF entwickelt. Durch eine enge Einbindung von CASE–Werkzeug Herstellern und Anwendern soll CDIF möglichst alle Anforderungen in dem Bereich des CASE Datenaustauschs erfüllen. So sind unter anderen die Firmen Ford, Boeing, Aerospatiale, British Aerospace als Anwender und Integrated Systems Inc., Rational Software Corp., Platinum Technology, Cayenne Software Inc. als Hersteller an der Entwicklung beteiligt. Die erste Version wurde im Jahr 1991 unter der Bezeichnung EIA/CDIF 1991 verabschiedet. Diese erste Version wurde 1994 durch die überarbeitete Version EIA/CDIF 1994 ersetzt.

CDIF beinhaltet sowohl die funktionale Beschreibung als auch die Verhaltensbeschreibung von elektronischen Systemen und unterstützt dazu eine Reihe von Entwurfsmethoden aus den Bereichen Datenflußmodellierung (Yourdon/DeMarco [Your89]), Zustands–Ereignis–Modellierung (Zustandsautomaten [Lipp95], Petrinetze [Petr62], Statecharts [Hare87]), objekt–orientierte Analyse und Design (Booch [Booc94], Use Cases [Jaco92], OMT [RuBl91], UML [BoJa97]) und Regler–Modellierung (Regelungsglieder, Zustandsraum [Föll85]). Damit die Daten auch grafisch rekonstruierbar bleiben, können Informationen über die Darstellung (Präsentation) der Daten integriert werden.

Das CDIF–Komitee beschäftigt derzeit mehrere Arbeitsgruppen, die Methoden zur Darstellung der einzelnen Teilbereiche ausarbeiten und diese anhand von konkreten Anwendungsbeispielen auf

ihre Tauglichkeit prüfen. Am Ende dieses Entwicklungsprozesses soll CDIF der ISO (International Organization for Standardization) zur endgültigen Standardisierung vorgelegt werden (dies ist für das Jahr 2000 vorgesehen). Um Konflikte mit weiteren Standards auszuschließen, findet eine enge Zusammenarbeit mit der OMG (Object Management Group), IEEE (Institute of Electrical and Electronics Engineers) und ANSI (American National Standards Institute) statt. Nach der Standardisierung durch die ISO soll ISO/CDIF der OMG zur Fortführung der bisherigen Arbeiten übergeben werden.

Die offene, erweiterbare und methodenbezogene Struktur machen CDIF zu einer geeigneten Basis zur Kopplung von mehreren heterogenen Systemkomponenten. Um dies zu ermöglichen, verwendet CDIF zur Beschreibung von Systemen eine Entity–Relation Beschreibung, die durch eine mehrstufige Meta–Modell Architektur festgelegt wird.

6.3.2 Entity-Relation Modellierung

Die Entity–Relation–Modellierung (kurz ER–Modellierung) wurde 1976 zur Datenmodellierung entwickelt [Chen76]. Das Ziel der ER–Modellierung liegt in einer konzeptionellen Modellierung, die gegenüber einer Änderung der Funktionalität weitgehend stabil ist. Ausgangspunkt der ER–Modellierung ist die Zerlegung eines Systems in einzelne Entities. Eine Entity ist dabei ein individuelles und identifizierbares Exemplar von Dingen, Personen oder Begriffen der realen oder der Vorstellungswelt und wird durch Eigenschaften beschrieben. Eine Entity muß jedoch eindeutig identifizierbar sein. Es existiert eine Menge von Entity–Typen, die zur Modellierung verwendet werden können. Eine Entity wird durch ein Rechteck dargestellt, in dem der Typ der Entity eingetragen ist. Die Darstellung eines CASE–Werkzeugs als Entity zeigt Bild 6-4.

Bild 6-4: Darstellung einer Entity vom Typ CASE–Tool

Einer Entity können Attribute zugeordnet werden, die die Entity und somit das beschriebene Objekt näher spezifizieren. Bei einem CASE–Werkzeug könnte dies z.B. der Name und die Version des CASE–Werkzeugs sein. In einem ER–Diagramm werden diese Attribute in das Rechteck der Entity eingefügt. Die Beziehungen zwischen einzelnen Entities werden durch Relationen beschrieben. Eine Relation besteht immer zwischen genau zwei bekannten Entities und zeichnet sich durch ihren Typ und eine Richtung aus. Somit können allgemeine Beziehungen zwischen einem CASE–Werkzeug und einem Mensch wie in Bild 6-5 dargestellt werden.

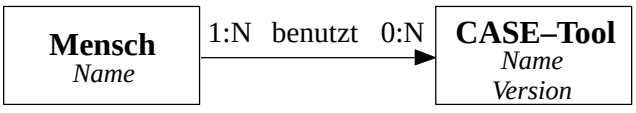


Bild 6-5: Relation zwischen zwei Entities

An den Endpunkten einer Relation ist die minimale und maximale Kardinalität angegeben, die die mögliche Anzahl dieser Relation zwischen Instanzen dieser Entity–Typen beschreibt. Durch die Angabe *0:N* wird beschrieben, daß ein *Mensch* keines oder beliebig viele *CASE–Tools* benutzen kann. Die inverse Beziehung, die mit der Kardinalität *1:N* beschrieben ist, gibt an, daß ein *CASE–Tool* von einem oder mehreren *Menschen* benutzt werden kann.

Durch die Aufteilung in mehrere Untertypen läßt sich eine Entity genauer spezifizieren. So kann ein *Mensch* die Übermenge von Arbeiter, Angestellter und freier Mitarbeiter bilden. Wurde dabei der Entity *Mensch* das Attribut *Name* zugeordnet, so ist dieses Attribut für alle Untertypen gültig. Ebenso gelten alle Relationen, die zur Entity *CASE–Tool* führen, für alle Untertypen. Bild 6-6 zeigt zwei ER–Diagramme, die den gleichen Sachverhalt beschreiben.

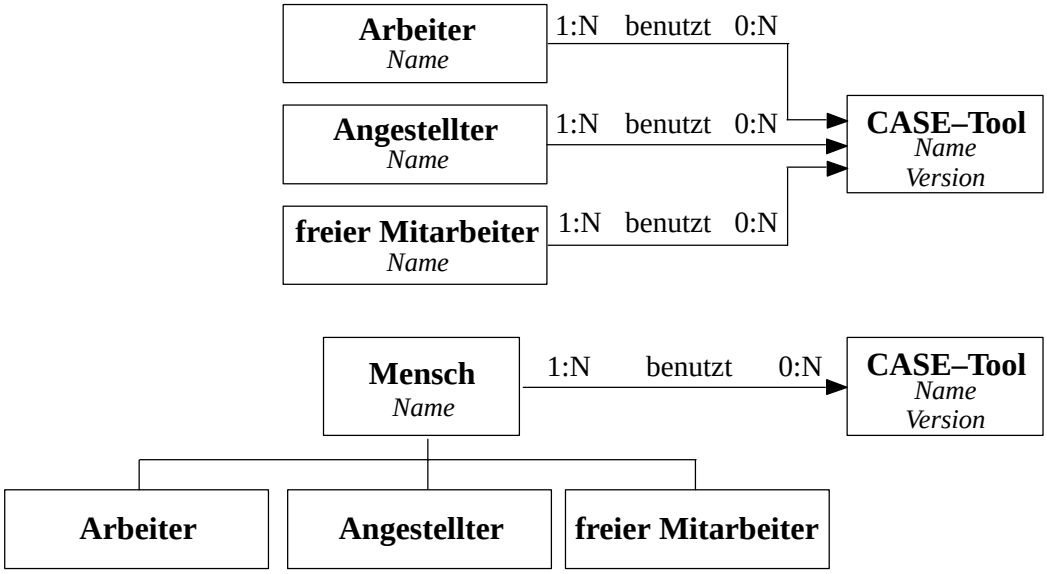


Bild 6-6: Untertypen von Entities

Bisher wurden nur Typen von Entities und Relationen betrachtet. Um einen konkreten Sachverhalt zu beschreiben, müssen Instanzen dieser Entities und Relationen gebildet werden. In Bild 6-7 wurde am Beispiel eines Angestellten und zweier CASE–Werkzeuge ein ER–Modell gebildet. Dazu wurden eine Instanz des Entity–Typs Angestellter und zwei Instanzen des Entity–Typs *CASE–Tool* benötigt. Die genaue Angabe, um welchen Angestellten und welche CASE–Tools es sich handelt, erfolgt unter Verwendung der Attribute. Diesen werden bei der Instanziierung bestimmte Werte zugewiesen, wie z.B. dem Attribut Name in der Entity vom Typ Angestellter der Wert ”J. Mustermann” zugewiesen wurde. Dabei muß aber nicht jedes verfügbare Attribut mit einem Wert versehen werden, um eine Entity eindeutig zu beschreiben.

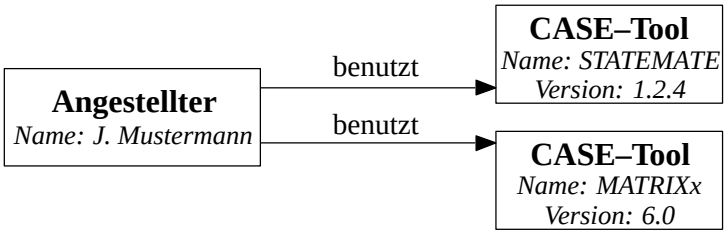


Bild 6-7: Instanziierung von Entities

6.3.3 Die Architektur von CDIF

CDIF ist in der Vier–Ebenen Architektur aufgebaut, die bei der Meta–Modellierung angewendet wird (siehe Kapitel 6.2). In Bild 6-8 ist diese Vier–Ebenen Architektur dargestellt. Die Information, die übermittelt werden soll, ist eine Addition der Form $y = a + b$. Die Modellierung dieser Information erfolgte unter Verwendung der ER–Modellierung. Durch vier Entities und drei Relationen wurde ein Modell gebildet, daß eine bestimmte Addition (eine Entity vom Typ *SummationDefini-*

tion) zwei Attribute (jeweils eine Entity vom Typ *Attribute*) verwendet, die die Namen *a* und *b* haben (die Entities haben ein Attribut vom Typ *Name* und dem Wert *a* bzw. *b*), um ein Attribut mit dem Namen *y* zu berechnen. In der Meta–Modell Ebene sind die Typen von Entities und Relationen definiert, die in der Modellebene verwendet wurden. Man spricht dabei von Meta–Entities und Meta–Relationen, die Meta–Attribute enthalten können. Dieses Meta–Modell besteht aus zwei Meta–Entities und zwei Meta–Relationen und bietet alle Elemente, die benötigt werden, um eine Addition zu beschreiben. Die Angabe der Kardinalität der Meta–Relationen gibt zusätzliche Modellierungsregeln vor. So muß eine Addition genau ein Attribut berechnen (1:1), während es für diese Berechnung mindestens zwei Attribute verwenden muß (2:N). Auf der Meta–Meta–Modell Ebene wird definiert, daß die Beschreibung eines Meta–Modells als ER–Beschreibung erfolgen muß. Weitere Ebenen sind nicht notwendig, da ausgehend vom Meta–Meta–Modell alle Arten von konformen Meta–Modellen beschreibbar sind (völlige Offenheit der Beschreibung) und das Meta–Meta–Modell sich selbst definiert.

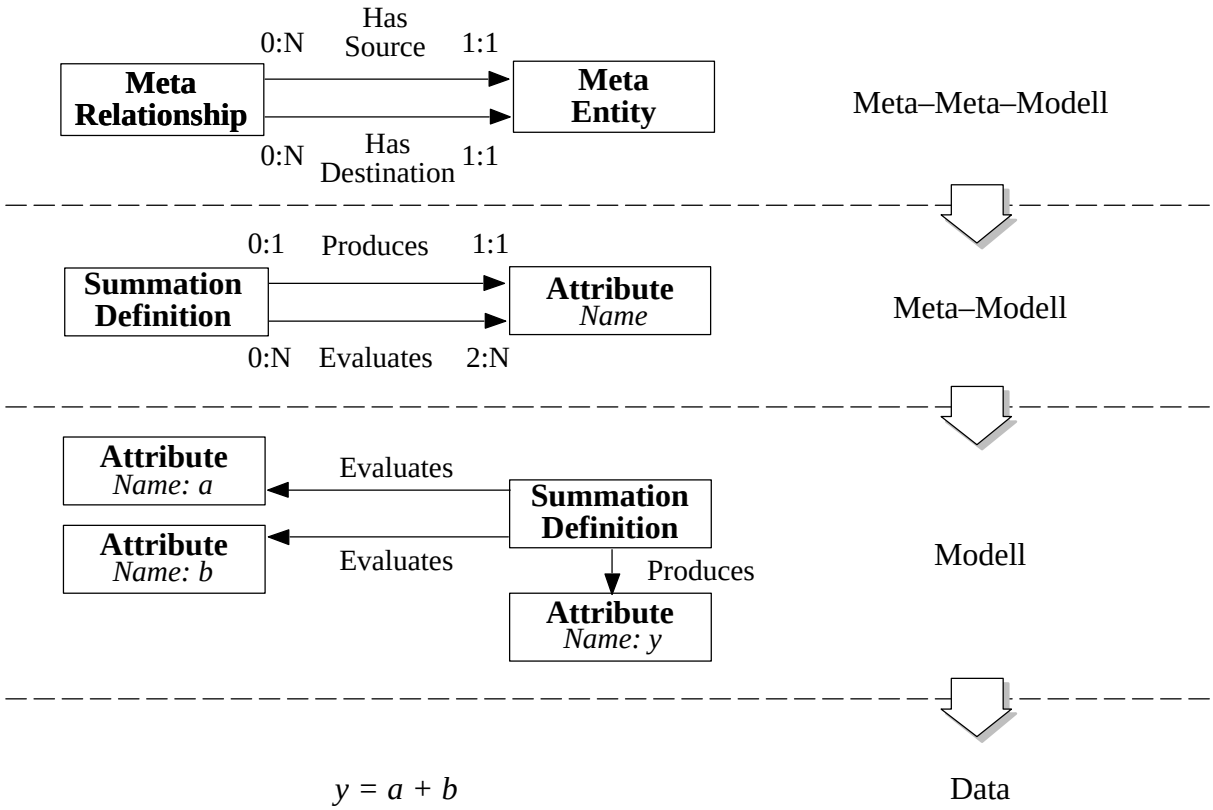


Bild 6-8: Vier–Ebenen Architektur von CDIF

Bei der Standardisierung eines Datenaustauschformats, das diese Architektur verwendet, wird das Meta–Meta–Modell und ein oder auch mehrere Meta–Modelle definiert. Bei der Verwendung dieses Datenformats wird aber nur das Meta–Modell und das Modell übertragen. Das Meta–Meta–Modell muß dem beteiligten Werkzeug bereits bekannt sein und braucht daher nicht zu übertragen werden. Die Übermittlung des Meta–Modells bringt den Vorteil mit sich, daß ein bereits bestehendes Meta–Modell unter Verwendung der Regeln des Meta–Meta–Modells um weitere Elemente erweitert werden kann. Somit kann der Funktionsumfang dieses Datenformats erweitert werden, ohne den verwendeten Standard aufgeben zu müssen.

CDIF zeichnet sich durch eine hierarchische Anordnung der einzelnen Komponenten aus. Beispielsweise hat eine Änderung der Komponente Meta–Modell keine Auswirkungen auf die Komponente Meta–Meta–Modell. Somit sind die einzelnen Komponenten voneinander unabhängig. Diese Architektur hat den Vorteil, daß einzelne Komponenten geändert werden können, ohne daß dabei Änderungen in anderen Komponenten vorgenommen werden müssen.

Anhand der in Kapitel 6.1 eingeführten Kriterien für Datenaustauschformate soll im folgenden CDIF auf seine Eignung geprüft werden.

Anwendungsbereich

Da CDIF aufgrund seiner Zielsetzung, möglichst alle Anforderungen beim Datenaustausch zwischen CASE–Werkzeugen erfüllen zu können, sehr umfangreich ist, bedarf es auch eines sehr umfangreichen Meta–Modells. Bei der Anwendung von CDIF werden aber in der Regel nur einige Teilbereiche des gesamten Meta–Modells benötigt werden. So werden zur Beschreibung eines diskreten Systems keine Elemente zur Beschreibung eines Regelglieds benötigt werden. Daher wurde das Meta–Modell in Gruppen geteilt, die eine sinnvolle Zusammenfassung einzelner Elemente darstellt. Diese Gruppen werden Subject Areas genannt. Da alle Subject Areas gleichzeitig verwendet werden können und somit ein einziges Meta–Modell repräsentieren, darf ein Element nur ein einziges Mal in einer Subject Area definiert sein. Wird ein grundlegendes Element in mehreren Subject Areas benötigt, so wird es einmal definiert und in allen weiteren Subject Areas nur noch referenziert. In CDIF wurden zwei grundlegende Subject Areas (Foundation und Common) definiert, die den Ursprung und das Grundgerüst für jede mögliche Kombination von Subject Areas festlegen. Sie sind somit integraler Bestandteil jedes Meta–Modells.

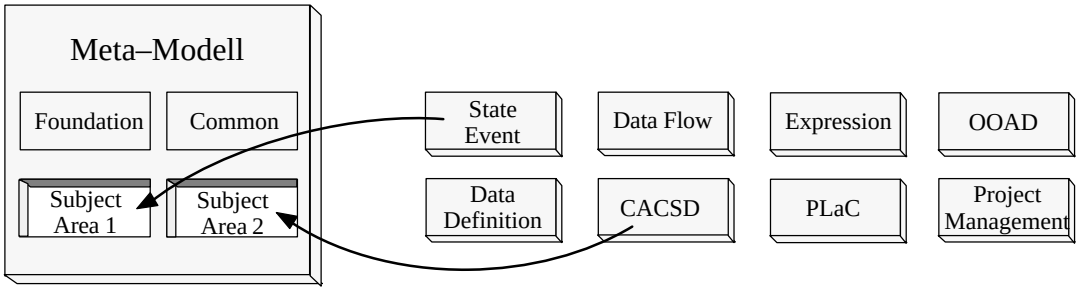


Bild 6-9: Zusammensetzung des Meta–Modells

In verschiedenen Arbeitsgruppen wurden mehrere Subject Areas erarbeitet und zum Teil bereits verabschiedet. Diese bestehen aus:

- *Foundation Subject Area*, die die grundlegenden Definitionen des Integrated Meta–Modells von CDIF bereitstellt. Sie definiert das *RootObject*, von dem sich die beiden Untertypen *RootEntity* und *RootEntity.IsRelatedTo.RootEntity* ableiten. Alle weiteren Entities, bzw. Relationen des Integrated Meta–Modells und der individuellen Erweiterungen sind Untertypen dieser beiden Objekte.
- *Common Subject Area*, die Konzepte beinhaltet, die allen Subject Areas gemein sind. Dies sind z.B. die Namengebung, das Erzeugen von Synonymen und die Strukturierung von Objekten. Darüberhinaus wird eine Trennung von semantischer Information und Präsentation vorgenommen. Hierfür wird die *RootEntity* aus der Foundation in ein *SemanticInformationObject* und ein *PresentationInformationObject* aufgeteilt.

- *Data Definition Subject Area* zur Definition von Datentypen. Dabei können vordefinierte Datentypen wie z.B. Integer, Float, Boolean und Strings verwendet werden. Auch eigene Datentypen und –strukturen können definiert werden.
- *Expression Subject Area* zur Repräsentation von mathematischen Ausdrücken. Der Umfang reicht von der Darstellung einfacher Operatoren wie Addition, Subtraktion, Multiplikation und Division bis zur Abbildung von Interpolationen und Bit–Operationen.
- *Data Modeling Subject Area* zur Repräsentation von Daten für Datenbanken. Dies umfaßt alle Elemente der klassischen Entity–Relation Modellierung als auch Erweiterungen wie Rollen, Schlüssel und N–Array Relationen.
- *Data Flow Subject Area* zur Repräsentation von Datenflüssen. Diese werden bei der Entwicklung komplexer Systeme zur funktionalen Dekomposition eingesetzt. Ein Beispiel hierfür sind die Activity–Charts von STATEMATE ™.
- *State/Event Subject Area* zur Repräsentation von erweiterten Finite State Machines (FSM). Mit dieser Subject Area sind diskrete Modellierungen mit Statecharts, Moore– und Mealy–Automaten oder Petri–Netze darstellbar.
- *Computer Aided Control System Design Subject Area* (CACSD) zur Repräsentation von zeit–kontinuierlichen Modellen wie z.B. Regelungskreise. Die einzelnen zeit–kontinuierlichen Modelle werden mit Datenflüssen untereinander verbunden und sind in statische und dynamische Prozesse untergliedert. Bei statischen Prozessen berechnet sich der Ausgang direkt aus den anliegenden Eingängen, bei dynamischen Prozessen hängt der Ausgang zusätzlich von internen Zuständen oder einer Systemzeit ab.
- *Physical Relational Data Base Subject Area* zur Repräsentation von tabellen–orientierten Inhalten. Im Gegensatz zur Data Modeling Subject Area werden hier einzelne Zellen zum Ablegen bzw. Auslesen von Daten angesprochen.
- *Object–oriented Analysis and Design Subject Area* zur Repräsentation von objekt–orientierten Modellierungsmethoden wie z.B. Booch, Rumbaugh oder die Unified Modeling Language (UML). Auf diesem Gebiet wird eng mit der Object Management Group (OMG) zusammengearbeitet, um CDIF als allgemeinen Standard zur Repäsentation von OOAD Modellen zu etablieren.
- *Presentation Location and Connectivity Subject Area*, die im Gegensatz zu den obigen Subject Areas keine Semantik, sondern nur die Abbildung von Informationen zu Präsentationszwecken beinhaltet. Sie enthält Konstrukte zur Darstellung von grafischen Elementen wie z.B. Koordinatenpunkte, Linien, Pfeile oder Rechtecke.
- *Project Management Subject Area* zur Repräsentation von Aspekten des Projekt–Managements. Dabei können beispielsweise Anfangs– und Endtermin, Anzahl der Mitarbeiter und Ressourcen dargestellt werden.

Technische Qualität

Die technische Qualität stellt ein wichtiges Kriterium dar, ist jedoch objektiv nur schwer zu bewerten. Für die Bewertung der technischen Qualität von CDIF stellt die Bewertung des Meta–Modells den wichtigsten Punkt dar. Meta–Modelle von ein und demselben Sachverhalt können unterschiedlich aufgestellt werden. Dies kann am Beispiel der diskreten Modellierung gezeigt werden. Die Se-

mantik (Bedeutung) eines Modells kann auf unterschiedliche Arten präsentiert werden. In Bild 6-10 ist ein Automat mit gleicher Semantik einmal als Zustands–Übergangs–Tabelle und einmal als Statechart dargestellt.

Zustand–Übergangstabelle

	E1	E2
S1	S2	–
S2	S2	S1/A

Statechart

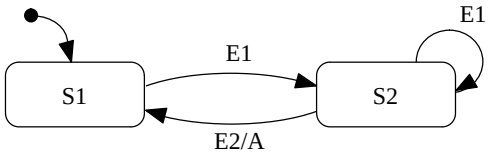


Bild 6-10: Beispiel: Unterschiedliche Präsentationsart – gleiche Semantik

Verschiedene Präsentationen bedeuten also nicht zwingend, daß es sich auch um Modelle unterschiedlicher Anwendungsbereiche handelt. Ebenso kann dieselbe Darstellung unterschiedliche Bedeutungen beinhalten. Die Semantik eines Statecharts etwa hängt davon ab, welche Bedeutung der Betrachter den einzelnen Elementen (Pfeile, Rechtecke) zuordnet.

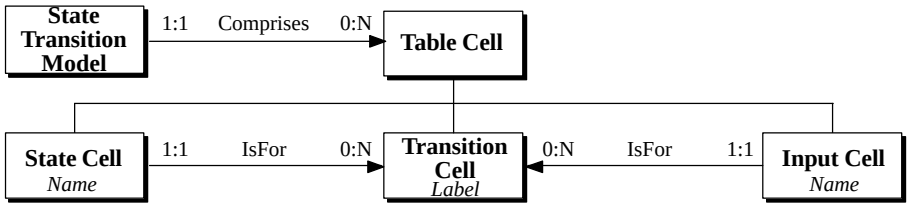


Bild 6-11: Qualitativ niederstehendes Meta–Modell

Ein Meta–Modell, das eine Trennung von Semantik und Präsentation nicht vornimmt, geht von der Präsentation der Zustands–Übergangs–Tabelle aus, weshalb sowohl die Zustände als auch die Ein– und Ausgänge des Automaten als Zellen einer Tabelle abgebildet werden. Dies ist in Bild 6-11 dargestellt. Hier gelingt zwar die Abbildung der Zustands–Übergangs–Tabelle (Bild 6-10 links), die Abbildung des Statecharts (Bild 6-10 rechts) gelingt jedoch nicht mehr, da der Begriff “Zelle” für Statecharts keinerlei Bedeutung hat. Ein so gewähltes Meta–Modell würde also keinen allgemeinen Modellaustausch zwischen verschiedenen CASE–Werkzeugen zulassen (Bild 6-12).

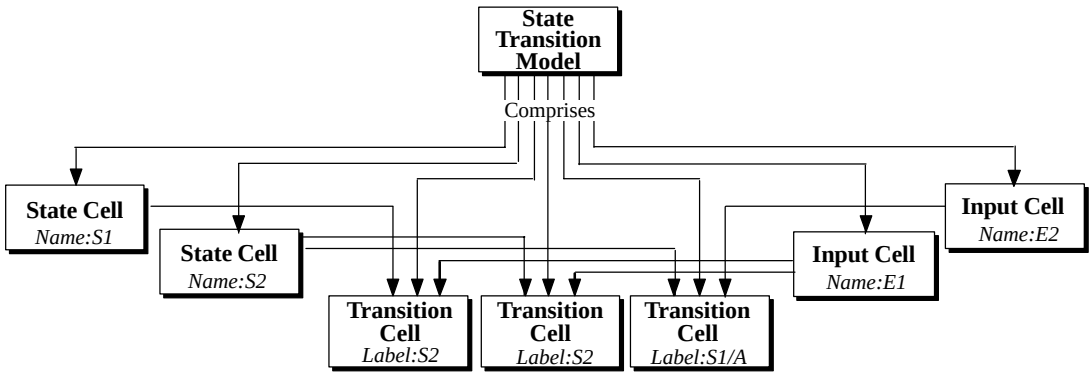


Bild 6-12: Modellierung mit qualitativ niederstehenden Meta–Modell

Die *State–Event Subject Area* von CDIF, die für die Darstellung von endlichen Automaten konzipiert ist, dient nur der Darstellung der Semantik eines solchen Automaten. Bild 6-13 zeigt einen

Ausschnitt der State–Event Subject Area. Zur Darstellung der äußeren Form, d.h. ob ein Übergang als Zelle einer Tabelle oder als Pfeil repräsentiert wird, wird die *Presentation, Location and Connectivity Subject Area* von CDIF benutzt. Findet ein Modellaustausch zwischen zwei CASE–Werkzeugen mit unterschiedlichen Präsentationsarten statt, kann das importierende Werkzeug die Präsentation des anderen Werkzeugs zwar nicht verstehen und übernehmen, bei einer strikten Trennung jedoch die Semantik verstehen und das Modell dann mit seiner Darstellungsweise präsentieren. Ohne eine Trennung von Semantik und Präsentation wäre in diesem Fall kein Datenaustausch ohne umständliche Konvertierung möglich.

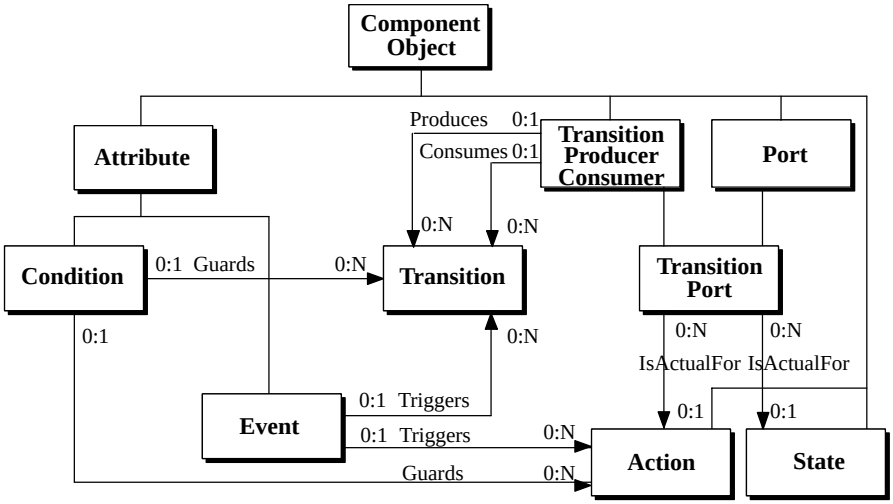


Bild 6-13: Ausschnitt aus dem CDIF Meta–Modell

Zur Abbildung der Zustände eines Automaten verwendet das CDIF Meta–Model die Entity *State*. Zustandswechsel werden durch die Entity *Transition* abgebildet, die über *TransitionPorts* mit den *States* verbunden werden. Übergangsbedingungen und damit verbundene Aktionen werden durch die Entities *Condition*, *Event* und *Action* unter CDIF abgebildet.

Einen weiteren Punkt der technischen Qualität stellt die Trennung von semantischer Information und physikalischer Übertragung dar. CDIF trennt zwischen den Definitionen, die die Bedeutung der zu übertragenden Information beschreiben, und den Definitionen, die festlegen, auf welche Weise diese Informationen übertragen werden. Die Beschreibungsart wird durch das verwendete Meta–Modell festgelegt. Die Regeln, auf welche Weise Informationen in eine standardisierte Form für den Transfer abgelegt werden, stellt CDIF im sogenannten Transfer Format bereit. Formal gesehen legt das Meta–Modell die Semantik fest, das Transfer Format legt die lexikalische Ebene und die Syntax des Transfers fest.

Mit der Erstellung des Meta–Modells und der Erzeugung eines Modells stehen alle Informationen für den Datenaustausch zur Verfügung. Für eine Datenübertragung muß jedoch noch festgelegt werden, wie diese Informationen physikalisch repräsentiert werden sollen. Ein Austausch an Informationen kann z.B. unter Verwendung einer Datenbank, eines Netzwerkes, einer Diskette oder eines gedrucktes Dokumentes erfolgen. Da das Modell unabhängig von der Art der Übertragung ist, wurde auch in CDIF eine Trennung zwischen der Modellierung und dem Transfer Format vorgenommen. Die Regeln, wie das Transfer Format aufgebaut ist, sind ebenfalls im Meta–Meta–Modell enthalten.

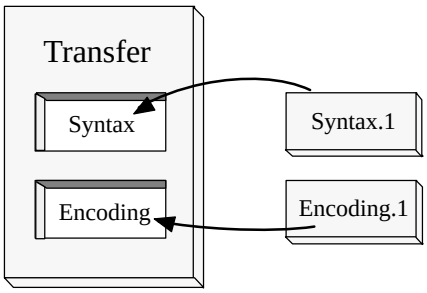


Bild 6-14: Struktur des Transfer Formats

Das Transfer Format gliedert sich in eine Syntax und eine Encoding, also einer Grammatik und einem Wortschatz. Derzeit existiert eine Syntax.1 und Encoding.1 für eine ASCII–Repräsentation. Eine Repräsentation in XML (Extended Markup Language) ist ebenso wie eine Datenübertragung unter Zuhilfenahme von CORBA (Common Object Request Broker Architecture) [Obje99] bereits realisiert. CORBA ermöglicht die Ablage von Objekten wie Daten, Funktionen und Klassen in einer offenen Umgebung. Auf diese Umgebung kann von dem Programm, das die Daten erzeugt hat, genauso zugegriffen werden, wie von Programmen, die auf verteilten Rechnern ablaufen. Die Trennung von Syntax und Encoding erlaubt eine leichte Anpassung an das Medium, mit dem die Daten ausgetauscht werden sollen. Die ASCII–Darstellung hat den Vorteil, daß sie von einem Menschen leicht erfaßt werden kann. Für die Übermittlung über ein Netzwerk ist dies jedoch von Nachteil, da der Datenumfang im Vergleich zu einer binären Darstellung wesentlich größer ist. Die Grammatik, die bei beiden Darstellungen genutzt wird, kann die gleiche sein. Somit muß nur eine Schnittstelle für den Datenaustausch mit CDIF implementiert werden, die mehrere Darstellungen der Schlüsselwörter verwenden kann.

Das Meta–Meta–Modell, das Meta–Modell und das Transfer Format bilden somit die Architektur von CDIF. Die gesamte CDIF Architektur ist in Bild 6-15 dargestellt. Unter Anwendung dieser Architektur können Modelle erzeugt und für den CDIF–Datentransfer bereitgestellt werden.

Erweiterbarkeit

Unter Verwendung der Subject Areas kann ein Meta–Modell zusammengestellt werden, das den Anforderungen der zu übertragenden Informationen angepaßt ist. Sollte der durch die Subject Areas gebotene Funktionsumfang nicht ausreichen, kann man nach den Regeln des Meta–Meta–Modells eine individuelle Erweiterung des Meta–Modells vornehmen. Modellierungen, die auf Basis eines erweiterten Meta–Modells vorgenommen wurden, müssen im Gegensatz zu gewöhnlichen Modellierungen das Meta–Modell in den Datentransfer mit aufnehmen (Bild 6-15).

Die Erweiterbarkeit gewährleistet einen fehlerfreien Datenaustausch von Semantik und Präsentation zwischen CASE–Werkzeugen, die nicht über das gleiche Meta–Modell verfügen. Somit stellt die Verwendung von CDIF nicht nur eine Lösung zur Zusammenarbeit gegenwärtiger CASE–Werkzeuge dar, sondern bietet darüberhinaus Sicherheit für zukünftige Entwicklungen.

Qualität der Dokumentation

Die Dokumentation von CDIF bietet eine genaue Definition aller beteiligten Elemente des Meta–Modells. Alle Elemente sind zum besseren Verständnis textuell beschrieben und mit zusätzlichen Randbedingungen versehen. Dabei werden die Attribute der Meta–Elemente unterteilt in vererbt / nicht vererbt und optional / obligatorisch. Attribute werden mit weiteren Beschreibungen wie Da-

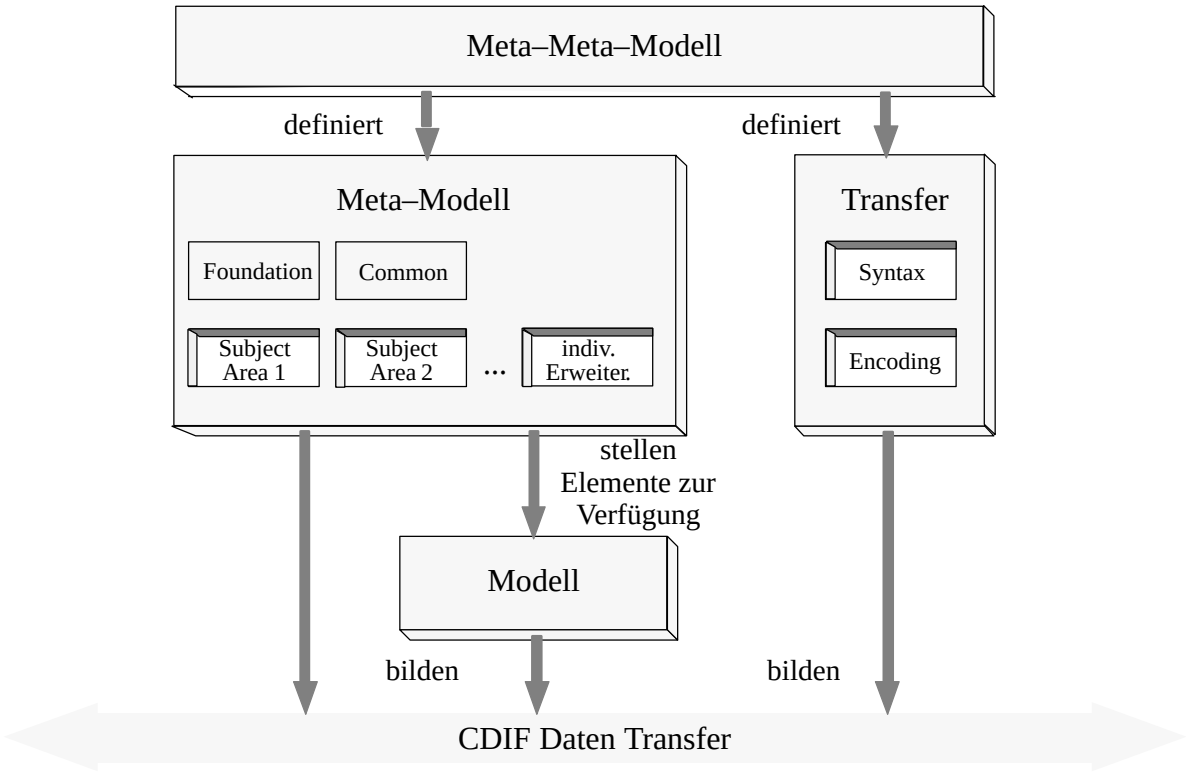


Bild 6-15: Architektur von CDIF

tentyp, Bereich, etc. näher gekennzeichnet. Durch den Einsatz einer Datenbank [Flat98], die alle Elemente konsistent verwaltet, wirken sich Änderungen in einzelnen Dokumenten auf andere, abhängige Dokumente aus. Die Qualität der Dokumentation ist somit als hoch einzustufen.

Integration

Große Anstrengungen werden unternommen, um sich in weitere Standards einzugliedern. Hierbei wird insbesondere auf Standards im Bereich der objekt–orientierten Modellierung und Datenübertragung eingegangen, die von der Object Management Group (OMG) gesteuert werden, wie beispielsweise der Unified Modeling Language UML oder CORBA [Obje99]. Bereits realisiert ist eine Anbindung von CDIF an CORBA, das eine Infrastruktur für die Erstellung von verteilten Anwendungen innerhalb heterogener Umgebungen zur Verfügung stellt (siehe auch den Abschnitt 'Technische Qualität'). Die OMG hat bereits beschlossen, als Datenformat für UML CDIF zu verwenden. Nach dieser Entscheidung der OMG ist mit einer weiten Verbreitung CDIFs im Bereich der objekt–orientierten Modellierung zu rechnen. Weitere Zusammenarbeit besteht mit der ECMA (European Computer Manufacturers Association) [ECMA99], die Standards für die Bereiche Informations– und Kommunikationssysteme erarbeiten und CDIF als ein Datenbankschema für Daten des Bereichs Software Engineering nutzen (Portable Common Tool Environment PCTE) und verschiedenen Organisationen wie ISO IRDS (Information Resource Dictionary System), ISO/IEC JTC1/SC7/WG11 (Software Engineering Data Definition and Representation), SC7/WG11, ANSI X3L8 (Data Representation), ANSI X3H4 (Open Systems Repository) und IEEE P1175 (Task Force on Professional Computing Tools).

Des weiteren wird an einer Harmonisierung der Meta–Modelle von UML, CDIF und STEP (Standard for the computer–interpretable representation and exchange of computer data, ISO 10303)

[ISO99] gearbeitet. Gerade STEP, das als Zielsetzung die Repräsentation von produktdefinierenden Daten während des gesamten Lebenszyklus hat, ist eine interessante Ergänzung zu CDIF. Die Einbindung von STEP ermöglicht die Modellierung physikalischer Elemente, die im Bereich CAD/CAM entstehen, wie beispielsweise das Material eines Produktes, seine Abmessungen und genaue Positionsangaben. Dieser Standard stellt also die physikalische Umgebung dar, für die CDIF Modelle erstellt wurden (beispielsweise als Programmierung für das Steuergerät eines Automobils). Gerade im Hinblick auf das Ziel der kompletten Modellierung eines technischen Transportmittels (Digital Mock–up) einiger Unternehmen im Luft– und Raumfahrtsektor sowie in der Automobilindustrie ist diese Verbindung interessant. Hiermit wird die Möglichkeit in Aussicht gestellt, daß eine Gesamtmodellierung physikalischer Komponenten mit elektronischen Verhaltensbeschreibungen Wirklichkeit werden kann. Die hierzu notwendige Zusammenführung unterschiedlichster CASE–Werkzeuge kann nur auf einer standardisierten Basis gelingen.

Ebenfalls erwähnt werden müssen an dieser Stelle jedoch auch die Probleme, die bei der Standardisierung von STEP auftreten. Hier treffen leider nationale und firmenspezifische Interessen aufeinander und verhindern die schnelle Bildung eines Standards.

Akzeptanz

Die Akzeptanz von CDIF ist schwer einzuschätzen, da noch keine endgültige Standardisierung vorliegt. Aufgrund der obigen Punkte kann jedoch festgehalten werden, daß die technischen Voraussetzungen für eine erfolgreiche und langlebige Standardisierung vorliegen. Trotz der soliden technischen Basis hat es bereits in der Vergangenheit viele Standards gegeben, die nicht zum Einsatz kamen oder sich am Markt nicht durchsetzen konnten. Durch Kooperationen, die weitere Standardisierungsgremien mit CDIF bereits eingegangen sind, und die aktive Mitarbeit großer Firmen wie Boeing ist die Wahrscheinlichkeit der Verbreitung von CDIF groß. Die Adoption von CDIF durch die OMG führt zu einer weiteren Verbreitung dieses Standards. Außerdem muß an dieser Stelle erwähnt werden, daß keine gleichwertige Konkurrenz für CDIF existiert und bereits mehrere Firmen erfolgreiche Implementierungen auf CDIF–Basis entwickelt haben.

6.3.4 CDIF Datentransfer

Damit das importierende Werkzeug feststellen kann, ob es die eingehenden Daten aufgrund einer bekannten Syntax und Encoding analysieren kann, müssen diese Informationen in den Transfer integriert werden. Dies erfolgt unter Verwendung des *Transfer Envelope*.

Der Transfer Envelope enthält eine CDIF–, eine Syntax– und eine Encoding–Kennung. Entsprechend der Syntax.1 und Encoding.1 lautet diese Kennung:

```
CDIF, SYNTAX ”SYNTAX.1” ”02.00.00”, ENCODING ”ENCODING.1” ”02.00.00”
```

Wurde beim Datenimport eine bekannte Syntax und Encoding mit der korrekten Versionsnummer erkannt, so können die weiteren Informationen analysiert werden. In der *Header Section* sind Informationen abgelegt, die das exportierende Werkzeug durch seinen Namen und Versionsnummer beschreiben. Zusätzlich können projektbezogene Informationen, wie das Exportdatum, Uhrzeit und der Name des Benutzers abgelegt werden. Diese Angaben sind jedoch nicht zwingend erforderlich, da sie nur Zusatzinformationen zu einem Modell enthalten.

```
(:Header
  (SUMMARY
    (ExporterName           "STATEMATE MAGNUM")
    (ExporterVersion        "01:02:04")
    (ExportDate             "2000/01/01")
    (ExportTime             "12:00:00")
    (PublisherName          "H. Mustermann")
  )
)
```

Die Meta–Model Section enthält alle Angaben über das verwendete Meta–Modell. Wird auf Subject Areas des CDIF–Standards zurückgegriffen, so werden nur Referenzen darauf in diesen Abschnitt aufgenommen.

```
(:META-MODEL
  (:SUBJECTAREAREFERENCE Foundation
    (:VERSIONNUMBER "01.00") )
  (:SUBJECTAREAREFERENCE Common
    (:VERSIONNUMBER "01.00") )
  (:SUBJECTAREAREFERENCE DataModeling
    (:VERSIONNUMBER "01.00") )
)
```

Bei der individuellen Erweiterung einer Subject Area müssen Instanzen von Meta–Meta Entities und Meta–Meta Relationen gebildet werden. Da dabei auch Informationen wie die Kardinalitäten einer Meta–Relation und die Vererbung von Eigenschaften dieser Meta–Objekte abgelegt werden müssen, kann dieser Abschnitt sehr umfangreich werden.

In der Model Section sind alle Informationen abgelegt, die zur Modellierung der zu übertragenden Daten notwendig sind. Dieser Abschnitt wird durch das Kennwort 'Model' eingeleitet und enthält dann eine beliebige Anzahl Instanzen von Meta–Entities bzw. Meta–Relationen.

```
(:MODEL
    ...
)
```

Bei der Modellierung gibt es in CDIF einige grundlegende Konstrukte, die zum Verständnis einer Modellbeschreibung bekannt sein müssen.

6.3.5 Modellierungsprinzipien

In CDIF wird grundsätzlich eine Trennung in zwei Gruppen von Meta–Entities vorgenommen. Eine Gruppe umfaßt alle Meta–Entities, die Objekte repräsentieren und die zweite Gruppe enthält alle Meta–Entities, die Definitionen darstellen. Die entsprechenden Elemente im Meta–Modell zeigt Bild 6-16.

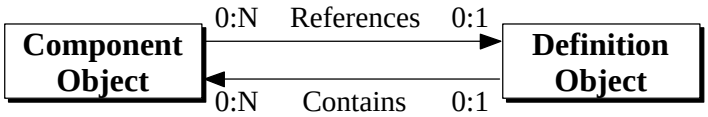


Bild 6-16: Objekte und Definitionen im Meta–Modell

Alle weiteren in den Subject Areas enthaltenen Meta–Entities sind Untertypen des *ComponentObject* bzw. des *DefinitionObject*. Das Verhalten und die Struktur eines Objektes wird durch eine Definition beschrieben. Die dazu notwendige Verknüpfung eines Objektes mit einer Definition erfolgt

mit der Meta–Relation *References*. Zur Beschreibung hierarchischer Systeme ist es notwendig, daß eine Definition weitere Objekte enthalten kann. Dies wird durch die Meta–Relation *Contains* ermöglicht. Anhand der Repräsentation einer Beispielschaltung (Bild 6-17) soll die Vorgehensweise bei der Modellierung aufgezeigt werden.

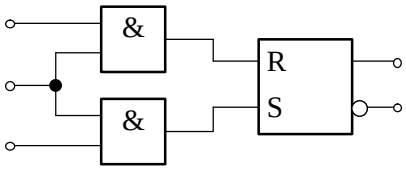


Bild 6-17: Beispielschaltung

Um ein abgeschlossenes System zu beschreiben, bedarf es einer Definition, die alle Elemente des Systems enthält. Die Schaltung besteht aus drei Elementen, die die eingehenden Informationen (Signale) verarbeiten. Solche Elemente werden in CDIF durch Prozesse repräsentiert. Das Verhalten und die Struktur dieser Prozesse ist wiederum in Definitionen beschrieben, die von den Prozessen referenziert werden. Da die beiden AND–Gatter das gleiche Verhalten und die gleiche Struktur haben, bedarf es für diese Prozesse nur einer Definition, die von beiden Prozessen referenziert wird. Zur Modellierung werden demnach drei Definitionen (System, AND, RS) und drei Prozesse (AND1, AND2, RS) benötigt.

Zur Modellierung des Informationsflusses werden *Flows* und *Ports* eingesetzt. Dabei wird zwischen gewöhnlichen und formalen Ports unterschieden. Ein formaler Port wird immer zur Beschreibung einer Schnittstelle verwendet, die aus einer Definition heraus führt. Die Definition des RS–FlipFlops enthält somit zwei formale *FlowInputPorts* und zwei formale *FlowOutputPorts*. Ein gewöhnlicher Port wird zur Beschreibung einer Schnittstelle eines Objektes verwendet. So sind dem Prozeß, der das RS–FlipFlop repräsentiert, zwei gewöhnliche *FlowInputPorts* und zwei gewöhnliche *FlowOutputPorts* zugewiesen. Diese Ports sind wie die Prozesse selbst in der Definition des Gesamtsystems enthalten. Somit kann der Informationsfluß zwischen den Schnittstellen durch *Flows* beschrieben werden. Diese werden in die Definition des Systems eingefügt und verbinden die jeweiligen *Ports* miteinander. Um einen Überblick über die CDIF Repräsentation zu erhalten, wird diese durch abstrahierte Elemente ersetzt. Diese Elemente sind in Bild 6-18 wiedergegeben.

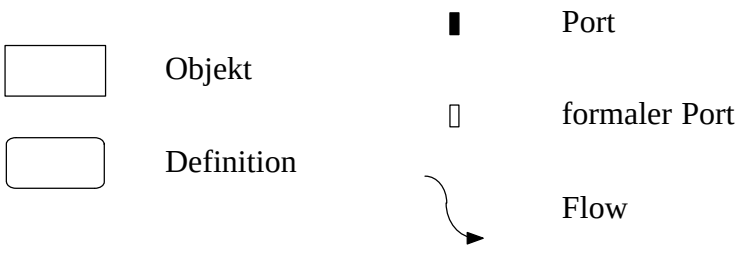


Bild 6-18: Abstrahierte Elemente

Die Modellierung der Struktur der Schaltung aus Bild 6-17 unter Verwendung der abstrahierten Elemente von Bild 6-18 zeigt Bild 6-19. Da in Bild 6-19 die Definition2 zweimal referenziert wird, bedarf es eines Mechanismus, der die Ports der Prozesse1 und 2 eindeutig einem formalen Port dieser Definition zuordnet. In CDIF wird dazu eine Meta–Entity *ReferencedElement* verwendet. In

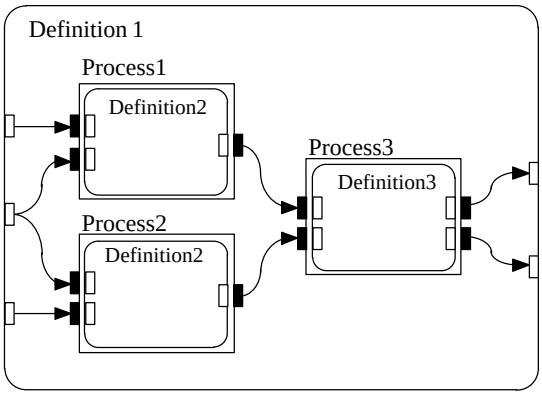


Bild 6-19: Modell der Schaltung

Verbindung mit der Meta–Relation *DefinesPath* kann eindeutig angegeben werden, in welchem Objekt (z.B. Prozeß1) ein darin enthaltenes Objekt (ein bestimmter formaler Port) angesprochen werden soll. Unter Verwendung der Meta–Entity *EquivalenceSet* und der Meta–Relation *HasMember* wird das *ReferencedElement* mit dem gewöhnlichen *Port* verbunden. Anhand eines Beispiels soll dieser Sachverhalt verdeutlicht werden.

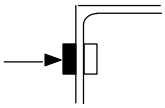


Bild 6-20: Darstellung eines Flows

Zur Modellierung der Elemente, die in Bild 6-20 dargestellt sind, bedarf es einiger Elemente, die in der Data Flow Model Subject Area definiert sind. Bild 6-21 gibt einen Ausschnitt aus dieser Subject Area wieder.

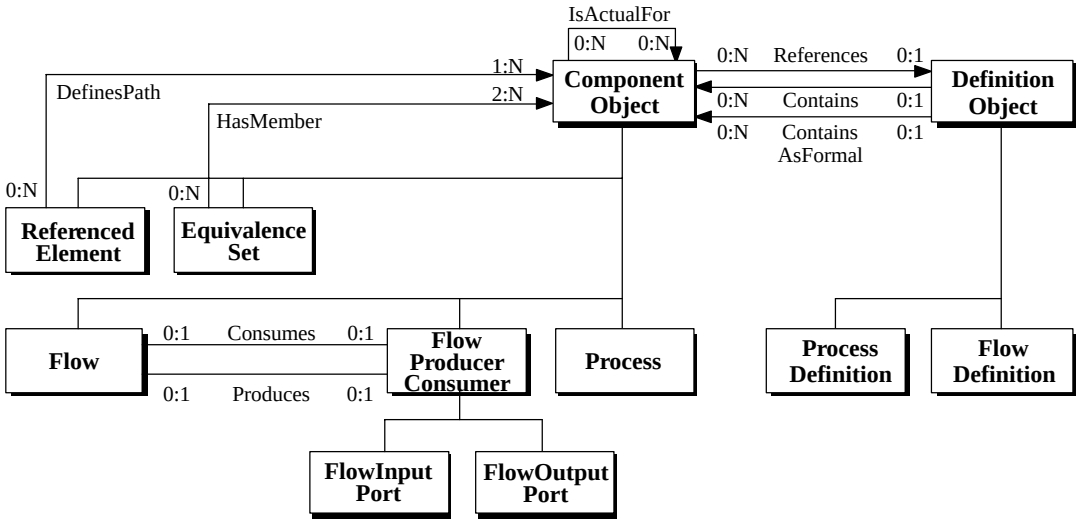


Bild 6-21: Ausschnitt aus der Data Flow Subject Area

Das *ComponentObject* hat eine Reihe von Meta–Entities als Untermenge, die in diesem Diagramm durch Linien mit dem *ComponentObject* verbunden sind. Jede dieser Meta–Entities stellt ein besonderes Objekt dar, das neben den allgemeinen Eigenschaften des *ComponentObject* weitere spezielle Eigenschaften besitzen kann. Im Falle des *FlowProducerConsumer* wurde eine weitere Untermenge gebildet, um Ports für Eingangs– und Ausgangssignale unterschiedlich darstellen zu kön-

nen. Ebenso besitzt das *DefinitionObject* zwei Untertypen, die von den entsprechenden Objekten referenziert werden können.

Alle Elemente aus Bild 6-21 befinden sich in einem Gesamtsystem, das durch eine *ProcessDefinition* repräsentiert wird. Diese Definition enthält somit einen *Flow*, einen *Port* und einen *Process*, da diese Elemente in dem System auf der gleichen Hierarchieebene liegen. Um den *Port* eindeutig dem *Process* zuordnen zu können, wird die Meta–Relation *IsActualFor* verwendet, die zwischen dem *Port* und dem *Process* gebildet werden kann. Das Meta–Attribut *Type* definiert dabei, ob der Informationsfluß in den *Process* hinein oder heraus führt. Der Prozeß selbst wird wiederum durch eine *ProcessDefinition* beschrieben, die einen formalen *FlowInputPort* enthält. Daß es sich um einen formalen Port handelt, wird durch die Meta–Relation *ContainsAsFormal* beschrieben. Da eine Definition und somit der formale Port von mehreren Prozessen referenziert werden kann, muß eine eindeutige Verbindung des formalen *Port*s mit dem gewöhnlichen *Port* gebildet werden. Dazu bedarf es eines *ReferencedElement* und zwei Instanzen der Meta–Relation *DefinesPath*. Die erste dieser Relationen verweist auf die Entity *Process*. Somit wird festgelegt, in welchem Objekt ein hierarchisch tiefer liegendes Objekt liegt. Dieses ”innere” Objekt wird durch die zweite Relation *DefinesPath* mit dem *ReferencedElement* verknüpft. Um die Reihenfolge der Relationen beschreiben zu können, wird dem Attribut *SequenceNumber*, das die Relationen näher spezifiziert, die Ordnungszahl zugeordnet. Unter Verwendung der Meta–Entity *EquivalenceSet* und der Meta–Relation *HasMember* wird nun das *ReferencedElement* mit dem gewöhnlichen *Port* verbunden. Das *ReferencedElement* und das *EquivalenceSet* sind dabei in der Definition des Systems enthalten. Eine direkte Verbindung der beiden Ports mit einem *EquivalenceSet* ist nicht möglich, da eine Definition mehrfach von einem Objekt referenziert werden kann. Somit würde ein formaler Port mit mehreren gewöhnlichen Ports verbunden werden, obwohl es sich um verschiedene Schnittstellen handelt. Daher wird das *ReferencedElement* eingesetzt, das genau angibt, in welchen *Process* welcher *Port* angesprochen werden soll.

Um die Entities und Relationen eindeutig identifizieren zu können, werden ihnen Identifier zugeordnet. Die Identifier werden dabei willkürlich vergeben und dienen nur der eindeutigen Zuordnung. Eine Relation enthält somit neben dem eigenen Identifier noch zwei weitere für den Ursprung und das Ziel der Relation. Die Einträge, die nach den Regeln der Syntax.1 und Encoding.1 in die Model Section vorzunehmen sind, sind demnach:

```
( ProcessDefinition ID01 )
( Process ID02 )
( ProcessDefinition ID03 )
( Flow ID04 )
( FlowInputPort ID05 )
( FlowInputPort ID06 )
( ReferencedElement ID07 )
( EquivalenceSet ID08 )

( DefinitionObject.Contains.ComponentObject ID09 ID01 ID02 )
( DefinitionObject.Contains.ComponentObject ID10 ID01 ID04 )
( DefinitionObject.Contains.ComponentObject ID11 ID01 ID05 )
( DefinitionObject.Contains.ComponentObject ID12 ID01 ID07 )
( DefinitionObject.Contains.ComponentObject ID13 ID01 ID08 )
( FlowProducerConsumer.Consumes.Flow ID14 ID05 ID04 )
( ComponentObject.IsActualFor.ComponentObject ID15 ID05 ID02
        ( Type ”IsInput” ) )
```

```
( ComponentObject.References.DefinitionObject ID16 ID02 ID03 )
( DefinitionObject.ContainsAsFormal.ComponentObject ID17 ID03 ID06 )
( ReferencedElement.DefinesPath.ComponentObject ID18 ID07 ID02
        ( SequenceNumber #D1 ) )
( ReferencedElement.DefinesPath.ComponentObject ID19 ID07 ID06
        ( SequenceNumber #D2 ) )
( EquivalenceSet.HasMember.ComponentObject ID20 ID08 ID07 )
( EquivalenceSet.HasMember.ComponentObject ID21 ID08 ID05 )
```

6.4 Darstellung diskreter Systeme

Zur Modellierung diskreter Systeme wie z.B. Zustandsautomaten, Petri–Netze und Statecharts, werden in CDIF die Elemente der State/Event Subject Area verwendet [Meie97]. Dabei werden nicht alle Elemente der unterschiedlichen Modellierungstechniken in speziellen Konstrukten abgebildet, sondern nur die Konstrukte, die zur eindeutigen Abbildung aller Modellierungstechniken ausreichend sind. Werkzeug–spezifische Erweiterungen werden dabei durch individuelle Erweiterungen ergänzt, z.B. bei STATEMATE™ der Deep–History Connector durch mehrfache Verwendung von History Connectoren.

Im folgenden werden Statecharts als Beschreibungsmittel diskreter Systeme betrachtet.

Zustandsdarstellung

Ein Zustand wird in CDIF durch die Meta–Entity *State* repräsentiert. Die Darstellung eines strukturierten Zustands wird durch die Meta–Entity *StateDefinition* ermöglicht. Diese Meta–Entity kann weitere Zustände, aber auch Transitionen, Attribute und Aktionen enthalten. Mit Hierarchien und Parallelitäten existieren grundsätzlich zwei Arten von strukturierten Zuständen. Bei einem hierarchischen Zustand ist nur ein Unterzustand aktiv, während bei einer Parallelität alle Unterzustände aktiv sind. Zur Unterscheidung der beiden Arten wird in CDIF einer *StateDefinition* mit dem Meta–Attribut *Operator* der Wert XOR für eine Hierarchie und der Wert AND für eine Parallelität zugewiesen. Im ER–Diagramm wird für jeden Zustand eine Instanz der Meta–Entity *State* gebildet, die bei strukturierten Zuständen eine Instanz der Meta–Entity *StateDefinition* referenziert.

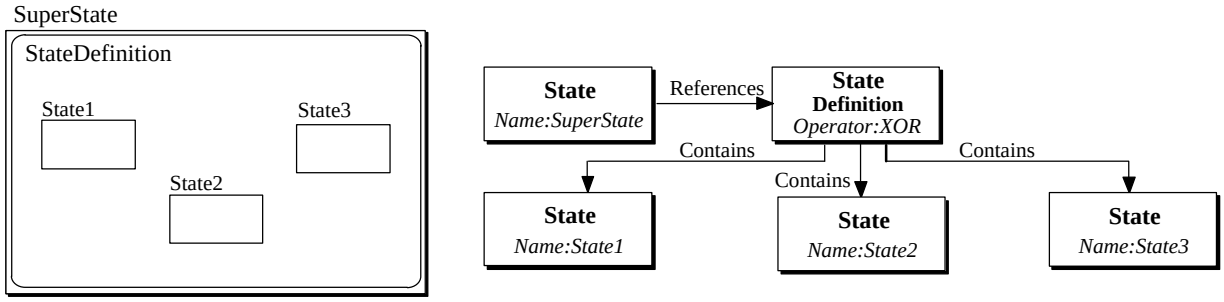


Bild 6-22: Modellierung einer Hierarchie

Transitionen

Eine Transition beschreibt einen möglichen Übergang von einem Zustand in einen anderen. Eine Transition führt dabei von einem *TransitionExitPort* zu einem *TransitionEntryPort*. Diese Ports müssen dabei in der gleichen Definition enthalten sein. Somit können nur Transitionen zwischen States modelliert werden, die in der gleichen Hierarchieebene liegen. Es existieren jedoch Pro-

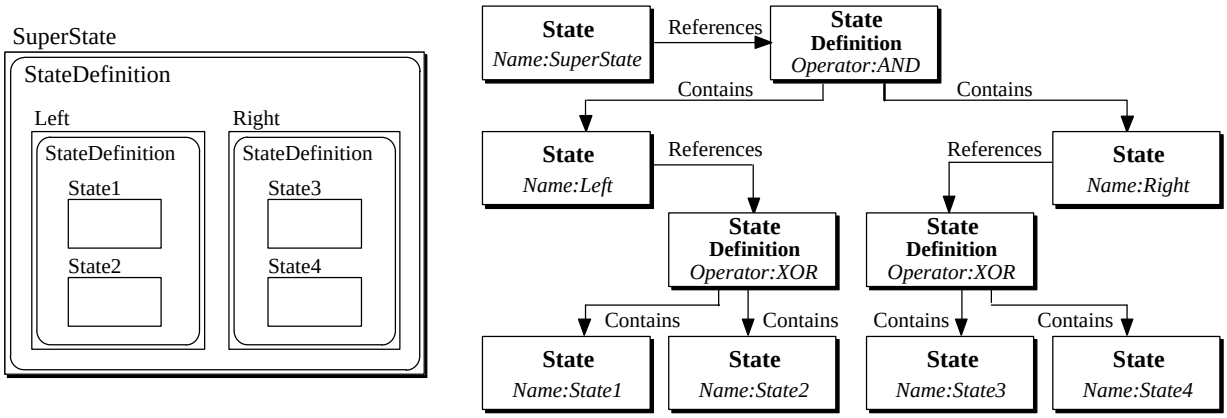


Bild 6-23: Modellierung einer Parallelität

gramme (z.B. STATEMATE ™), die Übergänge über Hierarchien hinweg erlauben. Dabei ist es notwendig, einen solchen Übergang in mehrere Transitionen zu zerlegen. Wird eine Hierarchieebene überschritten, so muß unter Verwendung eines formalen Ports eine Schnittstelle geschaffen werden. Bild 6-24 zeigt die wichtigsten Elemente, die zur Modellierung hierarchischer Übergänge benötigt werden. Die erste *Transition* führt dabei von dem *TransitionExitPort* des Startzustands zum *TransitionEntryPort* des hierarchischen Zustandes State2. Da die Transition innerhalb dieses Zustandes weiter führt, wird ein formaler *TransitionEntryPort* eingefügt und unter Verwendung eines *EquivalenceSet* und *ReferencedElement* mit dem gewöhnlichen *TransitionEntryPort* verbunden. Von dort führt eine zweite Instanz der Meta–Entity *Transition* zum *TransitionEntryPort* des Zielzustands State4.

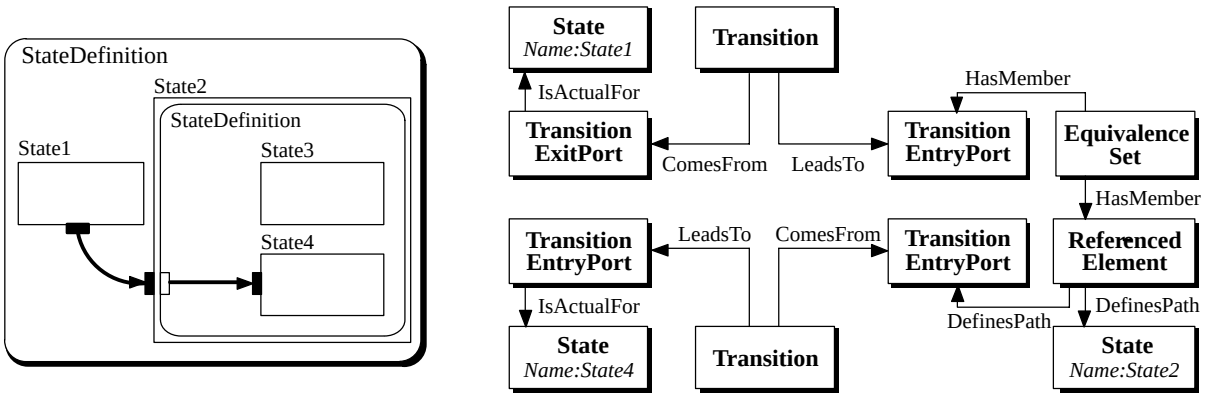


Bild 6-24: Modellierung einer Transition durch eine Hierarchieebene

Übergangsbedingungen

Um ein sinnvolles Übergangsverhalten des Systems beschreiben zu können, bedarf es der Modellierung von Ereignissen und Bedingungen, bei denen ein Übergang ausgeführt werden soll. Diese werden durch die Meta–Entities *Event* und *Condition* repräsentiert. Zur Darstellung eines strukturierten Events der Art (E1 and E2) wird ein hierarchisches Event gebildet, das als Definition eine Instanz der Meta–Entity *EventDataType* referenziert. Das Meta–Attribut *Operator* enthält die Information, wie die in dieser Definition enthaltenen Events miteinander verknüpft werden sollen. Zur Beschreibung strukturierter Conditions wird nach dem gleichen Schema die Meta–Entity *ConditionDa-*

taType verwendet. Durch die Meta–Relationen *Event.Triggers.Transition* und *Condition.Guards.Transition* werden die Bedingungen dem Übergang zugewiesen.

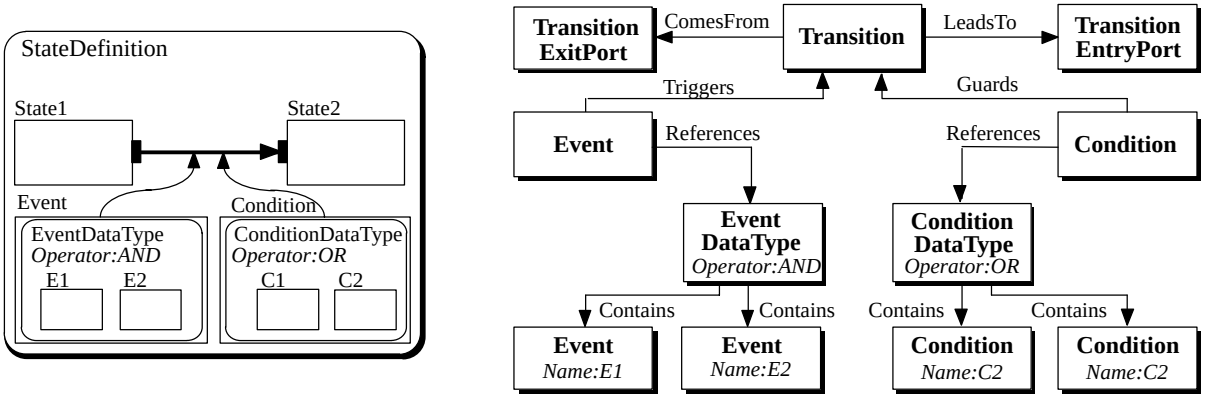


Bild 6-25: Bedingter Übergangswechsel

Aktionen

Bei der Ausführung eines Zustandswechsels können bei Statecharts auch Anweisungen ausgeführt werden. Diese können Variablen Werte zuweisen oder das Verhalten des Systems steuern. Sind mehrere Anweisungen vorgesehen, so muß festgelegt werden, in welcher Reihenfolge diese ausgeführt werden sollen. Jede Anweisung wird durch eine Instanz der Meta–Entity *Action* repräsentiert und die Anweisungen in einer *ActionDefinition* beschrieben. Eine Transition, die einen Zustandswechsel definiert, führt ”in die Action hinein”. Verfolgt man nun den Weg einer *Transition* von ihrem Startzustand zum Zielzustand, so kann dieser Weg durch eine oder mehrere *Actions* führen. Ein Übergang bestimmt demnach die Reihenfolge der Anweisungen, stellt aber keinen Informationsfluß dar.

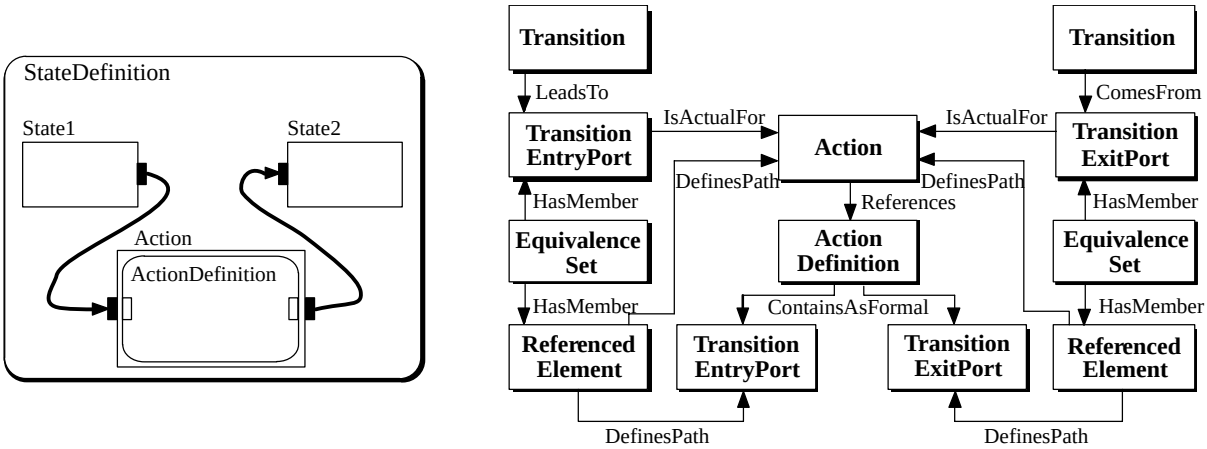


Bild 6-26: Modellierung einer Action

6.5 Darstellung kontinuierlicher Systeme

Die Modellierung kontinuierlicher Systeme, unabhängig vom Zeit– oder Frequenzbereich, zeichnet sich durch einen Informationsfluß durch ein System aus. Daher bildet die Data Flow Modeling Subject Area (DFM) die Grundlage für die Modellbildung kontinuierlicher Systeme. Durch die zu-

sätzliche Verwendung der CACSD Subject Area, die eine Erweiterung der DFM Subject Area darstellt, werden Elemente zur Beschreibung von Regelstrecken bereitgestellt [Meie97].

Struktur

Die Struktur eines kontinuierlichen Systems wird durch Prozesse und Datenflüsse dargestellt. Ein Prozeß manipuliert Daten, während ein Datenfluß diese nur weiterleitet. Die Funktion eines Prozesses wird durch die Meta–Entity *DFMProcessDefinition* beschrieben. Dabei ist die Bildung von Hierarchien möglich, indem eine *DFMProcessDefinition* weitere Instanzen der Meta–Entity *DFMProcess* enthält. Zur Beschreibung von Regelstrecken werden Untertypen der *DFMProcessDefinition* verwendet. Die Meta–Entity *StaticCACSDProcessDefinition* wird für Funktionsblöcke verwendet, die von keinem internen Zustand abhängen und somit das Ausgangssignal direkt aus den Eingangssignalen bestimmen können. Ein einfaches Beispiel ist ein Summierglied. Durch die Verwendung der Meta–Entity *DynamicCACSDProcessDefinition* werden zustandsabhängige Systeme dargestellt. Somit ist ein Integrator modellierbar, der zu seinem internen Zustand das Eingangssignal aufsummiert um das Ausgangssignal zu bestimmen. Der Eingang bzw. Ausgang eines Prozesses wird durch Schnittstellen beschrieben, die durch Instanzen der Meta–Entities *FlowInputPort* und *FlowOutputPort* modelliert werden. Der Verlauf der Signale wird durch *Flows* modelliert. Die Informationen über ein Signal, welchen Typ, Wertebereich und Struktur es hat, sind dabei in einer *FlowDefinition* abgelegt. Eine *FlowDefinition* kann mehrere *Flows* enthalten, wodurch mehrere Signale zusammengefaßt werden können (z.B. bei einer parallelen Verbindung).

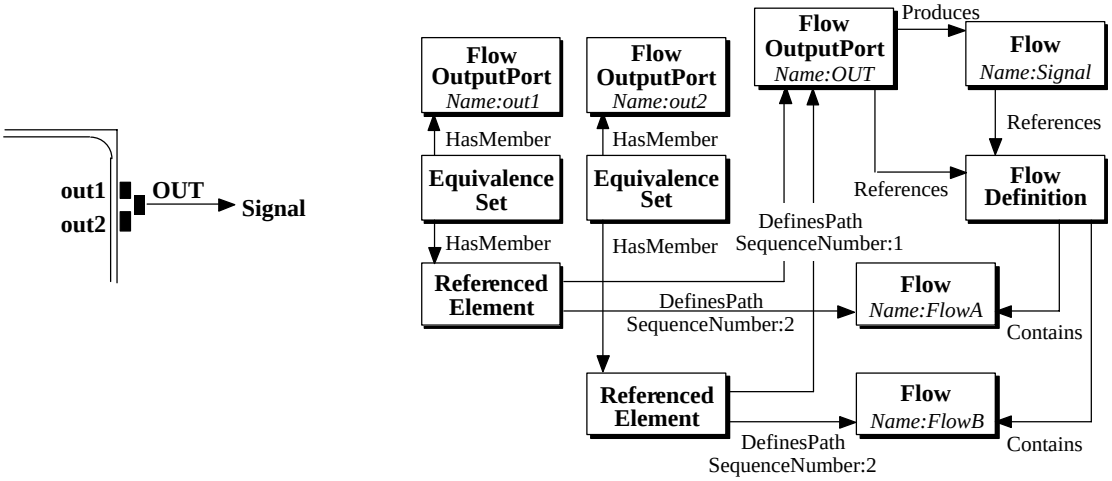


Bild 6-27: Modellierung eines strukturierten Flows

Zur Modellierung solcher *Flows* werden die Elemente wie in Bild 6-27 benötigt. Zunächst wird durch eine *FlowDefinition* beschrieben, wie dieser Datenfluß strukturiert ist. In diesem Falle enthält der Datenfluß *Signal* zwei weitere Flüsse *FlowA* und *FlowB*. Um nun diesen elementaren Datenflüssen einen Wert zuweisen zu können, muß jeder Fluß mit einem Port verbunden werden. Dazu referenzieren sowohl der strukturierte Fluß wie auch der *FlowOutputPort OUT* die gleiche *FlowDefinition*. Zusätzlich werden zwei *FlowOutputPorts out1* und *out2* eingefügt, die mit den elementaren Flows verknüpft werden. Unter Verwendung von je einem *ReferencedElement* und *EquivalenceSet* werden diese *Ports* mit den Datenflüssen in der *FlowDefinition* des *Ports OUT* verbunden.

Somit steht für jeden elementaren *Flow* ein *Port* zur Verfügung, während für die Beschreibung des eigentlichen Datenflusses nur ein einzelner *Port* verwendet wird.

Statische Funktionsblöcke

Funktionsblöcke, die von keinem internen Zustand abhängen, werden in CDIF als statisch bezeichnet. Dazu zählen alle Blöcke, die eine algebraische Verknüpfung der Eingangssignale vornehmen. Zur Beschreibung solcher Zusammenhänge werden Elemente der Expression Subject Area benötigt. Das CDIF–Komitee wird jedoch erst die Arbeit an dieser Subject Area aufnehmen, so daß derzeit nur ein Entwurf, der im Rahmen des MSR–Konsortiums erstellt wurde, zur Verfügung steht. Das Projekt MSR (Meß–, Steuer– und Regelungstechnische Systeme) war ein Entwicklungsvorhaben mehrerer deutscher Automobilhersteller und Elektrik/Elektronik–Zulieferer mit der Zielsetzung, die Zusammenarbeit zwischen Hersteller und Zulieferer zu verbessern und in der Effizienz zu steigern. Das 1990 initiierte Projekt wurde 1995 abgeschlossen.

Eine algebraische Verknüpfung wird durch die Meta–Entities *Expression* und *AlgebraicExpressionDefinition* beschrieben. Es stehen dabei eine Reihe von Untertypen der Meta–Entity *AlgebraicExpressionDefinition* zur Verfügung, die z.B. eine Summation oder Multiplikation beschreiben. Die benötigten Variablen werden durch Instanzen der Meta–Entity *Attribute* dargestellt. Durch die Meta–Relationen *AttributeExpressionDefinition.Evaluates.Attribute* und *AttributeExpressionDefinition.Produces.Attribute* werden diese in Bezug zueinander gebracht. Damit die Signale berechnet werden können, müssen die Ports des Prozesses mit den Attributen verbunden werden. Dies geschieht unter Verwendung der Meta–Entity *EquivalenceSet*, da damit eine Verbindung von Port und Attribute, bzw. *ReferencedElement* möglich ist. Die Definition eines Summierglieds ist in Bild 6-28 dargestellt. Der statische Prozeß enthält dabei drei formale *Ports* und eine *Expression*, die diese *Ports* miteinander verknüpft. Die Art dieser Berechnung wird durch eine *SummationExpressionDefinition* beschrieben, die zwei Attribute summiert und somit das Attribut mit Namen *out* berechnet. Damit die Ein– und Ausgangssignale mit diesen Attributen gekoppelt sind, wird für jeden *Port* über ein *EquivalenceSet* und *ReferencedElement* ein *Attribute* zugewiesen.

Zustandsabhängige Funktionsblöcke

Das Verhalten von zeitvarianten und zeitinvarianten dynamischen Prozessen ist nicht immer gleich, sondern es hängt von einem oder mehreren internen Zuständen ab. Es handelt sich dabei aber nicht um Zustände im Sinne der diskreten Modellierung, sondern um Zustände, die beliebige kontinuierliche Werte annehmen können. Alle Systeme, die zustandsabhängig sind, lassen sich durch eine Darstellung im Zustandsraum beschreiben.

Daher verwendet CDIF ausschließlich eine Darstellung dieser Systeme im Zustandsraum. Wurde ein System in Form einer Übertragungsfunktion definiert, so muß diese Darstellung in den Zustandsraum transformiert werden. Durch die zusätzliche Angabe eines Meta–Attributes, woraus der Zustandsraum erzeugt wurde, bleibt die Systembeschreibung nach dem Datentransfer eindeutig rekonstruierbar.

Für ein rationales Übertragungsglied gilt:

$$Y(s) = G(s)\,U(s) \qquad (6.1)$$

$$G(s) = \frac{Z(s)}{N(s)} = \frac{b_0 + b_1 s + \ldots + b_n s^n}{a_0 + a_1 s + \ldots + a_n s^n} \qquad (6.2)$$

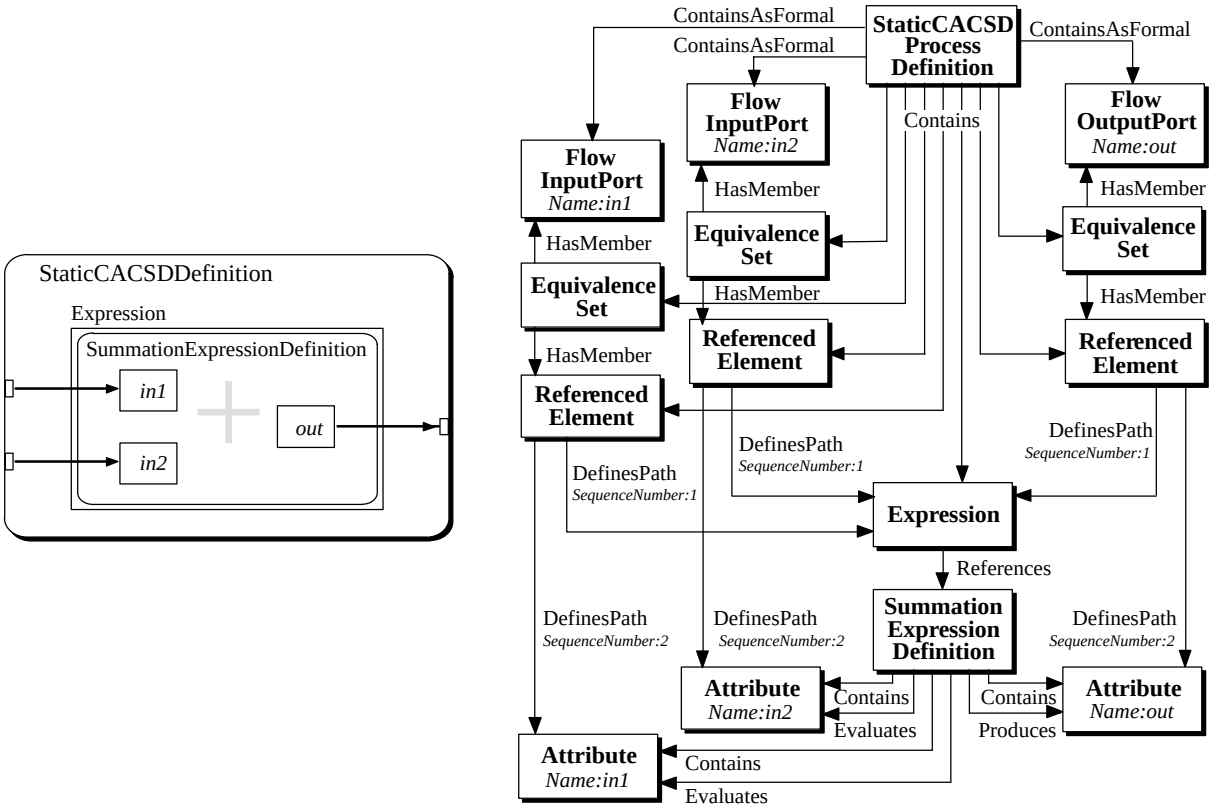


Bild 6-28: Definition eines Summierglieds

Dabei seien $a_n \neq 0$ und mindestens ein $b_\nu \neq 0, \nu \in \{1...n\}$. Zähler und Nenner seien ohne gemeinsame Wurzel. Die zugehörige Differentialgleichung im Zeitbereich lautet dann:

$$a_n \overset{(n)}{y} + ... a_1 \dot{y} + a_0 y = b_0 u + b_1 \dot{u} + ... + b_n \overset{(n)}{u} \qquad (6.3)$$

Nach Übergang in den Frequenzbereich und Umformung wird aus Gleichung (6.3):

$$Y(s) = b_0 \frac{1}{N(s)} U(s) + b_1 \frac{s}{N(s)} U(s) + ... + b_n \frac{s^n}{N(s)} U(s) \qquad (6.4)$$

Dann werden die Zustandsvariablen eingeführt, die untereinander einfache Beziehungen haben.

$$X_1 = \frac{1}{N(s)} U(s)\,,\; X_2 = \frac{s}{N(s)} U(s)\,,\; ...\,,\; X_n = \frac{s^{n-1}}{N(s)} U(s) \qquad (6.5)$$

$$sX_1 = X_2\,,\; sX_2 = X_3\,,\; ...\,,\; sX_{n-1} = X_n \qquad (6.6)$$

Falls man hier die Anfangswerte zu Null annimmt, folgt daraus im Zeitbereich:

$$\dot{x}_1 = x_2\,,\; \dot{x}_2 = x_3\,,\; ...\,,\; \dot{x}_{n-1} = x_n \qquad (6.7)$$

$$\dot{x}_1 = x_2\,,\; \ddot{x}_1 = x_3\,,\; ...\,,\; \overset{(n-1)}{x_1} = x_n \qquad (6.8)$$

Zusammen mit Gleichung (6.5) erhält man:

$$\dot{x}_n = -\frac{a_0}{a_n} x_1 - \frac{a_1}{a_n} x_2 - ... - \frac{a_{n-1}}{a_n} x_n + \frac{1}{a_n} u \qquad (6.9)$$

Diese Darstellung kann in eine Matrix–Schreibweise überführt werden, die in Gleichung (6.10) wiedergegeben ist. Die entstehende Darstellung wird als *Regelungsnormalform der Zustandsglei-*

chungen oder auch *Frobenius–Matrix* bezeichnet (Gleichungen (6.10) und (6.12)).

$$\dot{\underline{x}} = \begin{bmatrix} 0 & 1 & 0 & \vdots & 0 \\ 0 & 0 & 1 & \vdots & 0 \\ \vdots & \vdots & \vdots & \therefore & \vdots \\ 0 & 0 & 0 & \vdots & 1 \\ -\frac{a_0}{a_n} & -\frac{a_0}{a_n} & -\frac{a_2}{a_n} & \vdots & -\frac{a_{n-1}}{a_n} \end{bmatrix} \underline{x} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \frac{1}{a_n} \end{bmatrix} u \qquad (6.10)$$

Die Ausgangsgleichung wird durch Umwandlung von Y(s) in den Zeitbereich und Einsetzen der Zustandsvariablen gewonnen.

$$y = b_0 x_1 + b_1 x_2 + \ldots + b_{n-1} x_n + b_n \dot{x}_n \qquad (6.11)$$

$$y = \left[b_0 - a_0 \frac{b_n}{a_n} , \ldots , b_{n-1} - a_{n-1} \frac{b_n}{a_n} \right] \underline{x} + \frac{b_n}{a_n} u \qquad (6.12)$$

In Kurzschreibweise ergibt sich:

$$\dot{\underline{x}} = \underline{A}\,\underline{x} + \underline{B}\,u \qquad (6.13)$$

$$y = \underline{C}\,\underline{x} + \underline{D}\,u \qquad (6.14)$$

Die Darstellung im Zustandsraum besteht also aus einem Eingangs–, Ausgangs– und Zustandsvektor. Das Verhalten des Systems wird durch vier Matrizen beschrieben (A, B, C, D), deren Dimension sich aus denen der Vektoren ableiten lassen. Um ein eindeutiges Verhalten zu beschreiben, muß der Zustandsvektor mit einer Anfangsbelegung versehen werden.

Zur Darstellung des Zustandsraums in CDIF müssen zunächst die allgemeinen Gleichungen (6.13) und (6.14) repräsentiert werden. Diese setzt sich aus zwei Summationen und einer Multiplikation zusammen. Die Definition einer Summation wurde bereits in Bild 6-28 beschrieben. Bei der Definition einer Multiplikation wird die *SummationExpressionDefinition* durch eine *MultiplicationExpressionDefinition* ersetzt. Im Folgenden wird auf die Darstellung als ER–Diagramm verzichtet, da die entstehenden Diagramme zu umfangreich und damit unübersichtlich werden würden.

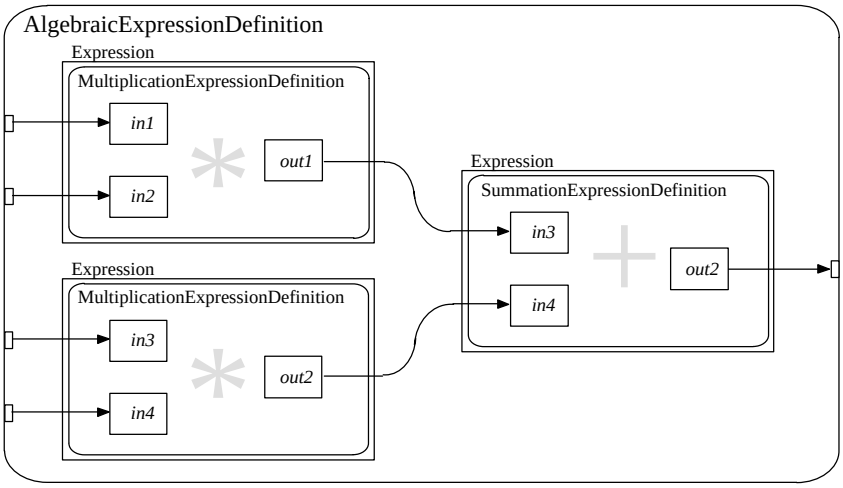


Bild 6-29: Definition der Gleichung *out = in1 * in2 + in3 * in4*

Unter Verwendung der Definitionen *SummationExpressionDefinition* und *MultiplicationExpressionDefinition* können die beiden Gleichungen beschrieben werden. In Bild 6-29 ist eine Entity vom Typ *AlgebraicExpressionDefinition* dargestellt, die vier Eingangs– und einen Ausgangswert

miteinander verbindet. Jeweils zwei Attribute werden durch eine Expression verknüpft, die eine *MultiplicationExpressionDefinition* referenziert. Das Ergebnis dieser Berechnung wird in zwei Attributen, hier als *out1* und *out2* bezeichnet, abgelegt. Eine weitere Expression summiert diese beiden Attribute und berechnet das Attribute *out*.

Zur Beschreibung des gesamten Zustandsraums besteht die Möglichkeit, gesamte Matrizen und Vektoren oder nur deren Elemente mit dieser Gleichung zu berechnen. Bei der Verwendung der einzelnen Elemente müßten für jeden Zustand und für jedes Ausgangssignal eine eigene Gleichung verwendet werden. Durch die Verwendung von Matrizen und Vektoren werden nur zwei Gleichungen benötigt. Eine Matrix wird durch eine Instanz der Meta–Entity *Attribute* repräsentiert, die eine Definition *NumericType* referenziert (siehe Bild 6-30). Diese Definition enthält die einzelnen Elemente, die wiederum durch ein *Attribute* repräsentiert werden. Diese Art der Datenbeschreibung ist in der Data Definition Subject Area beschrieben.

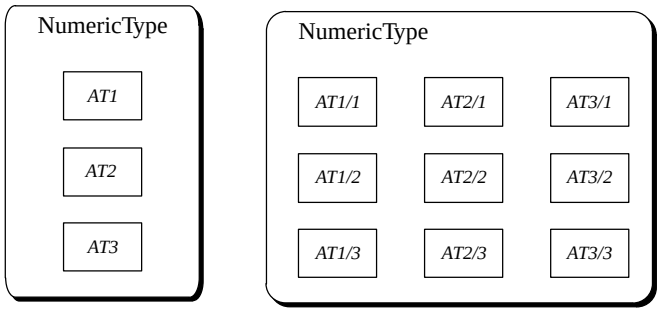


Bild 6-30: Darstellung von Vektoren und Matrizen

Die vier Systemmatrizen enthalten nur Konstanten, für deren Darstellung die Meta–Entity *ConstantAttribute* verwendet wird. Die Elemente der Eingangs– und Ausgangsvektoren wurden unter Verwendung der Meta–Entities *ReferencedElement* und *EquivalenceSet* mit den formalen *Ports* der Prozeßdefinition verbunden. Den Elementen des Zustandsvektors werden mit dem Meta–Attribut *DefaultValue* die Anfangswerte zugeteilt. Zur Beschreibung, welcher dieser Vektoren als Zustandsvektor des Prozesses dienen soll, wird die Meta–Relation *DynamicCACSDDefinition.HasState.Attribute* verwendet. Ebenso wird der Vektor der Ableitungen durch die Meta–Relation *DynamicCACSDDefinition.HasStateDerivate.Attribute* beschrieben. Unter Verwendung von zwei Entities *Expression*, die jeweils die Definition aus Bild 6-29 referenzieren, kann somit ein dynamischer Prozeß vollständig beschrieben werden. In Bild 6-31 ist die Definition eines Zustandsraums mit drei Eingangs– und zwei Ausgangssignalen dargestellt.

6.6 Kopplung diskret-kontinuierlicher Systeme

CDIF wurde als Basis für das Rapid Prototyping System gewählt, da es eine einheitliche Betrachtung heterogener Systeme erlaubt. Nachdem nun die einzelnen Komponenten beschrieben sind, muß nun die Kopplung der einzelnen Systeme betrachtet werden. In den Kapiteln 6.4 und 6.5 wurde bereits beschrieben, daß die Systemkomponenten innerhalb eines *CACSDProcess* abgelegt wurden. Verbindet man die *CACSDProcesses* untereinander, so muß das Gesamtsystem in eine *CACSDDefinition* eingebettet werden (siehe Bild 6-32). Hierbei wird das bereits in Kapitel 6.3.5 eingeführte Modellierungsprinzip von *ComponentObject* und *DefinitionObject* angewendet.

In Bild 6-33 ist die Kopplung zweier Systemkomponenten (einer zeit–kontinuierlichen und einer diskreten Systemkomponente) als Übersichtsbild sowie als ER–Diagramm wiedergegeben. Die

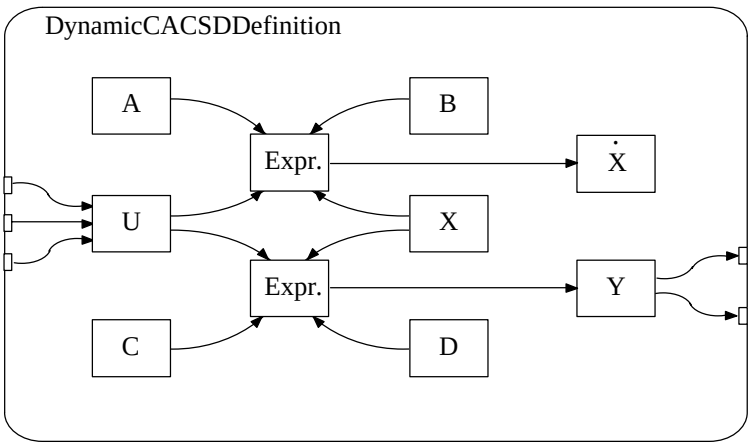


Bild 6-31: Definition des Zustandsraums

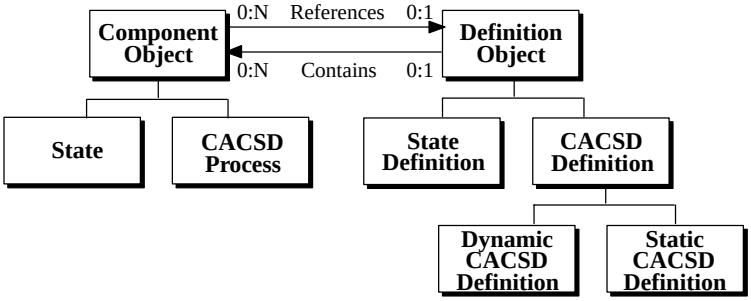


Bild 6-32: Ausschnitt aus dem CDIF Meta–Modell

CACSDDefinition, die das zeit–kontinuierliche Modell enthält, wird von einem formalen *FlowOutPort* über *ReferencedElement* und *EquivalenceSet* mit dem *FlowOutPort* des Datenflusses verbunden. Dieser produziert einen Datenfluß, der im *FlowInPort* des *CACSDProcess* des diskreten Modells konsumiert wird. Der *FlowInPort* ist dann wieder über *EquivalenceSet* und *ReferencedElement* mit dem formalen *FlowInPort* der *StateDefinition* verbunden.

An dieser Stelle sei angemerkt, daß CDIF das Modell in einem standardisierten Datenformat repräsentiert. Sind bei der Modellierung beispielsweise zwei Datentypen verbunden worden, die nicht kompatibel zueinander sind, so wird diese fehlerhafte Modellierung auch in CDIF übernommen. Die Aufgabe von CDIF besteht nicht in der Überprüfung sondern der Repräsentation der Modellierungsinformation. Eine Software zum Überprüfen des Modells kann, basierend auf der CDIF–Repräsentation, das Modell auf solche Fehler untersuchen.

6.7 Bewertung

Der Austausch von Modelldaten zwischen verschiedenen CASE–Werkzeugen wird insbesondere bei großen, verteilten Projekten immer wichtiger. Gerade die Automobil– und Flugzeugindustrie, die in besonderer Weise eine starke Vernetzung mit der Zulieferindustrie, aber auch von Entwicklungsteams innerhalb der Firma aufweisen, zeigen diesen Sachverhalt deutlich. Das Entwickeln von eigenen Lösungen, um den Datenaustausch zwischen CASE–Werkzeugen zu ermöglichen, ist wegen der zeitlichen Randbedingungen und der nicht gewährleisteten Konvertierungssicherheit nicht mehr ausreichend. Somit nehmen große Teile der Industrie Einfluß auf die CASE–Werkzeughersteller, um geeignete Lösungen zu finden. Um den schwierigen Prozeß zu beschleunigen, enga-

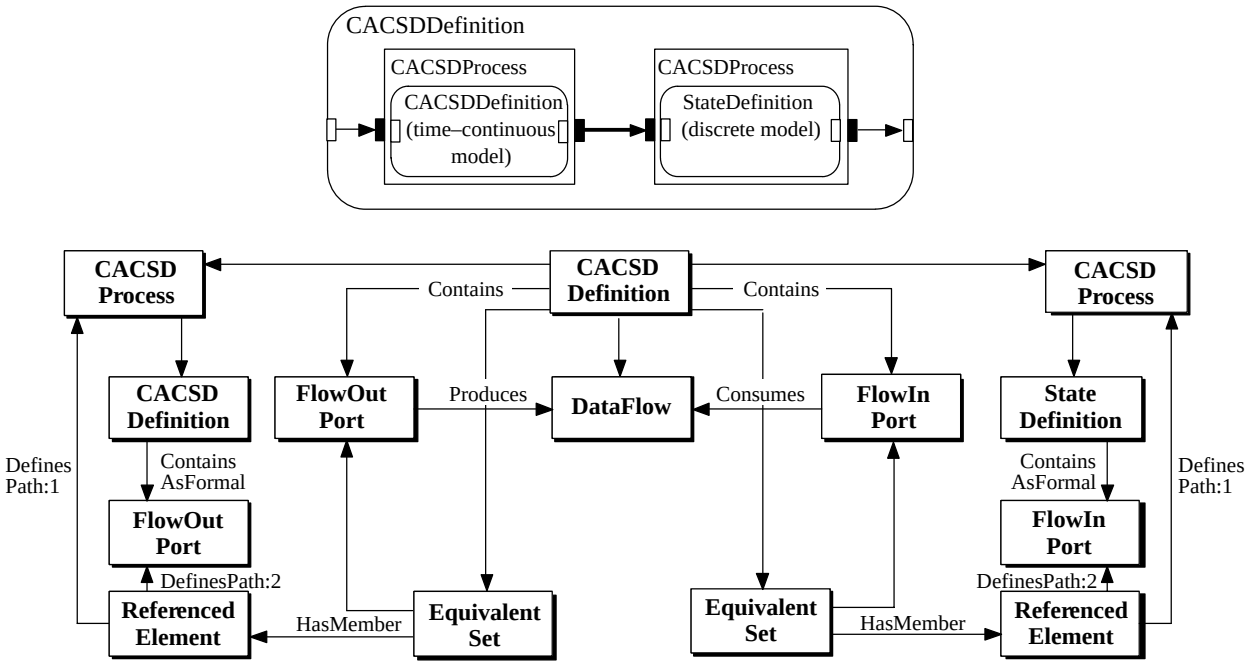


Bild 6-33: Verbindung zweier Modelle

gieren sich sowohl CASE–Werkzeughersteller wie auch Herstellerfirmen an der Entwicklung von Lösungen.

Umfangreiche Modellierungssprachen wie VHDL–AMS [Anal99] oder Modelica [ElMa97] bieten nur ihre eigenen Sprachkonstrukte zur Modellierung an. Diese sind jedoch entweder vom Modellierungsbereich zu stark eingeschränkt (VHDL–AMS) oder sehr komplex (Modelica), da unterschiedlichste Entwurfsbereiche unterstützt werden müssen (siehe Kapitel 6.2). Keine der oben genannten Modellierungssprachen unterstützt objekt–orientierte Techniken wie z.B. die Unified Modeling Language (UML) und eine Modellierung von relativen oder absoluten Zeitbedingungen. Zudem sind diese Modellierungstechniken nicht erweiterbar. UML andererseits bietet keine Lösungen beispielsweise zur Unterstützung von regelungstechnischen Problemen.

Die Verwendung von CDIF als Datenschema für den heterogenen Systementwurf bietet dagegen:

- eine eindeutige Repräsentation der dargestellten Modellierungsdaten
- den orthogonalen Aufbau von CDIF
- die Trennung zwischen Semantik und Präsentation der Daten
- die Trennung zwischen Syntax und Encoding
- eine Unterteilung des Meta–Modells in einzelne Subject Areas
- die Erweiterbarkeit des Meta–Modells

Die Vorteile und die Notwendigkeit eines CASE–Datenaustauschformats wie CDIF sind somit offensichtlich. An der Realisierbarkeit einer solchen Lösung bestehen allerdings Zweifel. Der allgemeine Aufbau des Meta–Modells birgt den Nachteil, daß die allgemeine Beschreibung der Modelle wesentlich umfangreicher im Vergleich zu speziellen Modellen ausfällt. Die starre Zuordnung, die beispielsweise MATRIXx™ bei der Datenrepräsentation vornimmt, stehen auf Seite von CDIF mehrere Entities und Relationen gegenüber (zu Details siehe Kapitel 9).

Wie aus Kapitel 9 ersichtlich werden wird, entstehen bei der momentan verfügbaren ASCII Codierung (Encoding.1) sehr große Transferfiles, die mehrere Mega–Bytes Umfang aufweisen können. Wegen ihrer Größe ist ein Datenaustausch über Netzwerke, Modem oder das Internet zu unhandlich. Aus diesem Grund ist die Entwicklung einer weiteren Codierung auf Basis eines binären Formats notwendig, die die Größe der Datenübertragung drastisch reduzieren kann. Diese Erweiterung von CDIF ist möglich, muß jedoch ebenfalls standardisiert werden.

Ein weiteres Problem, das sich auch durch eine binäre Codierung nicht verhindern läßt, ist die Weiterverarbeitung, also der Import der CDIF Modelle. Die Analyse von CDIF Daten mit mehreren Tausend Entities und Relationen führt selbst auf leistungsfähigen Rechnern zu Importzeiten von einigen Minuten (siehe Kapitel 9). Abhilfe kann durch die Entwicklung von hochoptimierten Compilern geschaffen werden, die den schnellen Import von CDIF Modellen ermöglichen. In Kapitel 9, das die Ergebnisse präsentiert, werden deswegen auch die Zeiten angegeben, die zur Übersetzung von Modellen benötigt werden.

Die Situation im Bereich der CASE–Werkzeuge erinnert an den Bereich der Programmiersprachen in den achziger Jahren. Damals wurde diskutiert, ob Hochsprachen wie C oder Pascal die optimierte Programmierung in Maschinensprache ersetzen könne. Viele Entwickler waren wegen der schlechten Performanz der Hochsprachen der Meinung, daß dies nicht gelingen könne. Die Komplexität der Aufgaben und die geforderten knappen Entwicklungszeiten ließen jedoch schnell erkennen, daß nur Hochsprachen den moderenen Anforderungen gewachsen sind. Dieselbe Situation existiert heute in der Modellierung elektronischer Systeme, die nur unter Zuhilfenahme einer Datenabstraktion gelöst werden kann. Der bereits weitentwickelte Standardisierungsprozeß von CDIF bietet dabei die ideale Basis zur Lösung der gegenwärtigen Modellierungsprobleme.

7 Systemkopplung in Echtzeit

7.1 Anforderungen und Eingrenzung

Der heterogene Aufbau elektronischer Systeme aus diskreten und kontinuierlichen Modellen erfordert bei der Abarbeitung des Modells mittels Simulation oder Rapid Prototyping eine Synchronisation der beteiligten Teilmodelle. Dabei unterscheidet man zwischen *kontinuierlicher* und *zeitdiskreter Abarbeitung*. Der Unterschied zwischen beiden Abarbeitungsarten liegt in der Charakteristik der verwalteten und berechneten Signalzeitverläufe und den damit verbundenen Berechnungsmethoden. Entsprechend der Änderungszeitpunkte und der Wertebereiche der Signale wird zwischen

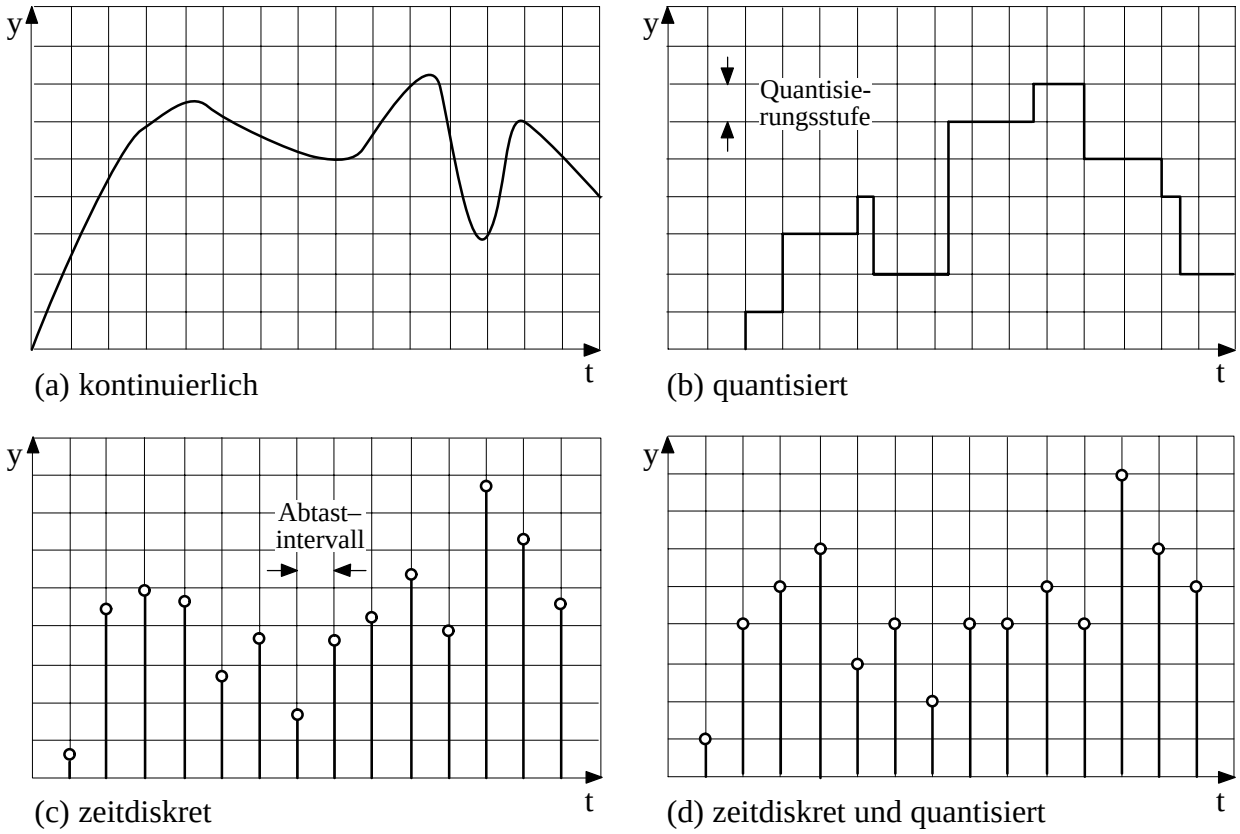


Bild 7-1: Klassifikation von Signalarten

(a) kontinuierlichen, (b) quantisierten, (c) zeitdiskreten und (d) zeitdiskreten und quantisierten Signalen unterschieden [Schm98]. Bei der zeitdiskreten Abarbeitung werden ausgewählte Werte zu unterscheidbaren Zeitpunkten berechnet. Die kontinuierliche Abarbeitung ermittelt dagegen sämtliche Werte des Systems in Abhängigkeit von der kontinuierlich fortschreitenden Zeit. Da dieses Vorgehen jedoch bei der Berechnung und Verwaltung einer unendlichen Menge von Zuständen ent-

spricht, muß die Abarbeitung auf digitalen Rechnern zeitdiskret erfolgen, d.h. es wird eine Rasterung des Ergebnisverlaufs bewußt in Kauf genommenen.

Im folgenden wird ausschließlich die zeitdiskrete Abarbeitung der heterogenen Modelle betrachtet. Da die Änderungen in diesem Fall lediglich zu bestimmten Zeitpunkten stattfinden können, findet auch die Synchronisation der Teilmodelle zeitdiskret statt. Bei der zeitdiskreten Abarbeitung kann eine feinere Einteilung in zeit- und ereignisgesteuerte Abarbeitung vorgenommen werden. Bei der *ereignisgesteuerten Abarbeitung* werden die eintreffenden Ereignisse als diejenigen diskreten Zeitpunkte festgelegt, an denen die Abarbeitung zu erfolgen hat. Dabei werden die eintreffenden Ereignisse der Reihe nach abgearbeitet, indem eine zeitlich geordnete Ereignisliste aufgestellt wird, die die Ereignisse und ihre Zeitinformationen enthalten. Die Ausführung der Ereignisse erfolgt dann nach der Reihenfolge der mitgelieferten Zeitinformationen.

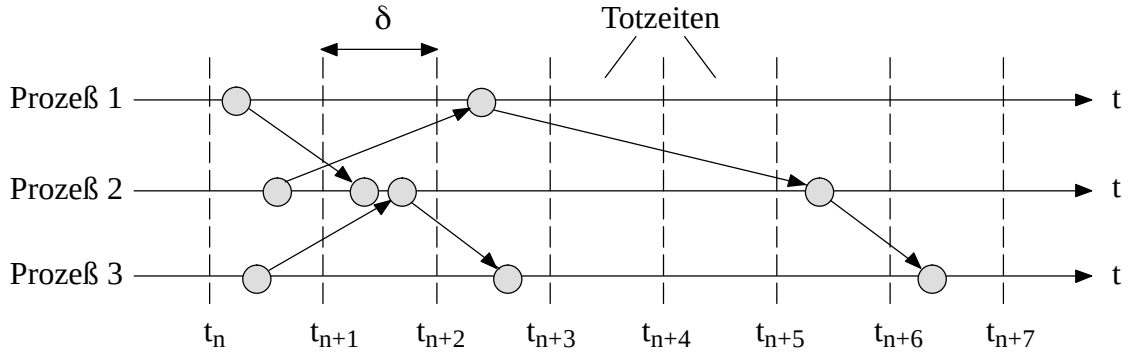


Bild 7-2: Zeit- und ereignisgesteuerte Abarbeitung [Schm98]

Die ereignisgesteuerte Abarbeitung ist insbesondere dann vorteilhaft, wenn über lange Zeiträume keine eingehenden Ereignisse (*Totzeiten*) auftreten. Gerade bei einer Abarbeitung ohne Echtzeitbedingungen, z.B. bei einer Simulation, kann bei dieser Vorgehensweise ein schnelles Fortschreiten der Simulationszeit gewährleistet werden. In Bild 7-2 sind diese Totzeiten dargestellt. Alle Ereignisse eines Prozesses sind als Punkte auf einer waagrechten Linie angeordnet. Die Pfeile zeigen, welche Ereignisse von welchen anderen erzeugt worden sind. Gestrichelte Linien stellen Zeitintervalle dar.

Bei der *zeitgesteuerten Abarbeitung* wird die Simulationszeit in Inkrementen δ fester oder variabler Schrittlänge festgelegt. In Bild 7-2 ist eine feste Schrittweite dargestellt. Nach einer Zeiterhöhung treffen die Ereignisse des Zeitintervalls $\left[t_n - \delta,\, t_n\right]$ in einer willkürlichen Reihenfolge ein. Das Zeitintervall muß dabei so gewählt sein, daß ein Ereignis kein weiteres Ereignis in demselben Intervall beeinflussen kann. Bereits hier wird die Wichtigkeit des Zeitintervalls δ für die Abarbeitung ersichtlich. Wird δ sehr klein gewählt, steigt die Rechengenauigkeit bei gleichzeitigem Anstieg der Rechendauer. Wird δ sehr groß gewählt, sinkt die Rechengenauigkeit aber auch die Rechendauer.

Im Unterschied zur Simulation muß bei Rapid Prototyping in Echtzeit auf Ereignisse reagiert werden. Ein Überspringen von Totzeiten ist in diesem Zusammenhang nicht sinnvoll, da eine gemeinsame Zeitbasis zwischen Umgebung und abzuarbeitendem Modell besteht, an der beide Teile unbedingt festhalten müssen. Lediglich die Zeitdauer vom Eintreffen eines Ereignisses bis zur Reaktion darauf kann von den Echtzeitrandbedingungen beeinflußt werden. In Kapitel 7.2 werden beide Abarbeitungsarten auf ihre Eignung für Rapid Prototyping näher untersucht.

Zeitparameter

Betrachtet man die in Bild 7-2 verwendeten Prozesse genauer, können die zeitlichen Anforderungen an einen Prozeß durch die folgenden Zeitparameter beschrieben werden:

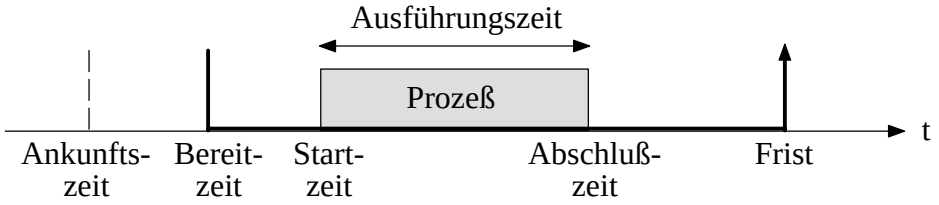


Bild 7-3: Zeitparameter eines Prozesses

- Die Ankunftszeit ***A*** (engl. *arrival time*) kennzeichnet den Zeitpunkt, zu dem ein Prozeß dem Betriebssystem bekannt gemacht wird.
- Die Bereitzeit ***R*** (engl. *ready time*) eines Prozesses bestimmt, wann der Prozeß gestartet werden kann. Ab diesem Zeitpunkt versucht der Prozeß, die Ressource Prozessor für seine Abarbeitung zu gewinnen.
- Unter der Startzeit ***S*** (engl. *start time*) versteht man den Zeitpunkt, an dem ein Prozeß zum ersten Mal Zugang zur Ressource Prozessor erhält.
- Die Ausführungszeit ***C*** (engl. *execution time*) bestimmt, wie lange ein Prozeß bis zur vollständigen Abarbeitung der Ressource Prozessor zugeordnet sein muß.
- Die Abschlußzeit ***F*** (engl. *completion time*) ist der Zeitpunkt, zu dem ein Prozeß beendet wird.
- Die Frist ***D*** (engl. *deadline*) ist der spätestmögliche Abschlußzeitpunkt eines Prozesses. Sie bestimmt, zu welchem Zeitpunkt ein Prozeß spätestens beendet sein muß.

Für ein konsistentes Prozeßsystem muß jeder Prozeß P die folgenden Bedingungen bezüglich seiner Zeitparameter erfüllen:

$$A(P) \leq R(P) \leq S(P) \leq F(P) - C(P) \leq D(P) - C(P) \qquad (7.1)$$

Im Rahmen dieser Arbeit wird von einer vernachlässigbaren Zeitspanne zwischen Ankunftszeit und Bereitzeit ausgegangen. Im folgenden wird nur von der Bereitzeit gesprochen und eine zeitlich davor liegende Ankunftszeit impliziert. Unter der *Antwortzeit* eines Prozesses wird diejenige Zeit verstanden, die von der Bereitzeit bis zur vollständigen Abarbeitung des Prozesses vergeht.

Sporadische Prozesse

Sporadische Prozesse treten nur gelegentlich in einem Prozeßsystem auf und sind im allgemeinen nicht voraussehbar. Treten willkürliche äußere Stimuli auf, die eine Reaktion des Echtzeitsystems bedingen, können die Zeitparameter eines solchen Prozesses nicht im vorhinein bestimmt werden. Eine Lösung bietet die Umwandlung eines sporadischen Prozesses in einen periodischen Prozeß. Dadurch wird eine immer wiederkehrende Reservierung von Ausführungszeit vorgenommen, in denen die Prozesse beim tatsächlichen Auftreten ablaufen können.

Periodische Prozesse

Periodische Prozesse wiederholen sich in einem bestimmten Zeitraum. Dieses Vorgehen kann von außen bedingt sein, indem beispielsweise ein äußerer Sensor periodisch einen Wert liefert oder von

dem Echtzeitsystem selbst, falls beispielsweise eine Prozeßüberwachung vorgenommen werden soll, die kritische Parameter eines technischen Systems überprüft.

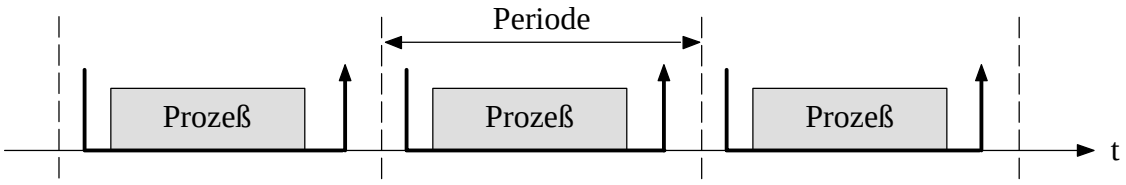


Bild 7-4: Periodischer Prozeß

Um auch solche Prozesse in das Prozeßmodell aufnehmen zu können, führt man den sich wiederholenden Zeitraum als Parameter ***T*** ein. Dieser Zeitraum wird als Periode (engl. *period*) bezeichnet. Er gibt den Zeitpunkt an, an dem ein periodischer Prozeß wieder von vorne abgearbeitet wird. Nichtperiodische Prozesse bekommen per Definition die Periode $T = 0$. Bild 7-4 zeigt den Ablauf eines periodischen Prozesses. Die Abarbeitung des Prozesses muß innerhalb einer Periode geschehen, falls die Echtzeitbedingung nicht verletzt werden soll.

Formales Prozeßmodell

Nachdem die zur Modellierung von Prozessen in Echtzeitsystemen notwendigen Parameter eingeführt wurden, kann nun die formale Beschreibung eines Prozesses vorgenommen werden. Das Prozeßmodell läßt sich wie folgt darstellen:

$$P_i = \left(R_i,\ C_i,\ D_i,\ T_i,\ Prio_i,\ V_i,\ Sp_i\right) \qquad (7.2)$$

wobei $R_i \in \mathbb{N}_0$ die frühestmögliche Startzeit, $C_i \in \mathbb{N}$ die Ausführungszeit, $D_i \in \mathbb{N}$ die spätestmögliche Endzeit, $T_i \in \mathbb{N}_0$ die Periode, $Prio_i \in \mathbb{N}_0$ die Priorität, $V_i \in \{true, false\}$ das Verdrängungsattribut und $Sp_i \in \{true, false\}$ das sporadische Attribut darstellt. Die Wichtigkeit der Priorisierung und des Verdängungsattributs wurde bereits bei der Prioritätsumkehr in Kap. 2.5 hervorgehoben.

Klassifikation des Schedulings

Um ein ganzes System charakterisieren zu können, das aus vielen Prozessen bestehen kann, unterscheidet man zwischen unterschiedlichen *Schedulingstrategien.* Eine der hauptsächlichen Unterschiede stellt die Abarbeitung auf einem (engl. *uniprocessor system*) oder mehreren Prozessoren (engl. *multiprocessor system*) dar. Komplexitätsuntersuchungen haben gezeigt, daß die meisten Probleme im Bereich der Abarbeitung auf mehreren Prozessoren np-hart sind [SSDB95] und durch die notwendige Synchronisation und Kommunikation viel zusätzlichen Aufwand mit sich bringen. Im Rahmen dieser Arbeit wird deswegen ausschließlich die Abarbeitung auf einem Prozessor untersucht.

Die Unterscheidung in statisches und dynamisches Scheduling beeinflußt die Fähigkeit des Schedulings, Korrekturen bei der Abarbeitung der Prozesse noch während der Laufzeit vornehmen zu können. Beim *statischen Scheduling* wird die Art der Abarbeitung bereits bei der Compilierung vorgenommen und besitzt somit ein festes Schema. *Dynamisches Scheduling* hingegen bestimmt die weitere Art der Abarbeitung während der Laufzeit des Systems. Dynamisches Scheduling wird hauptsächlich verwendet, wenn zusätzliche Prozesse zur Laufzeit erzeugt werden und das Ablaufverhalten nicht im vorhinein bestimmt werden kann. Damit ist dynamisches Scheduling nicht deterministisch und bedeutet zusätzlichen Berechnungsaufwand zur Laufzeit. Gerade in Echtzeitap-

plikationen wie Rapid Prototyping, bei denen eine hohe Abarbeitungsgeschwindigkeit bei sicherem, vorhersagbaren Verhalten im Vordergrund steht, kann die Zeit für die Berechnung der optimalen Schedulingstrategie für die Erfüllung der Echtzeitvorgaben erforderlich sein. Statisches Scheduling garantiert ein deterministisches Verhalten, wenn die Fristen, Ausführungszeiten und Randbedingungen bereits zur Compilierung feststehen und ist deswegen für Rapid Prototyping vorzuziehen.

Schedulingstrategien unterscheiden sich auch darin, ob ein Prozeß während seiner Abarbeitung unterbrochen und zu einem späteren Zeitpunkt auf derselben oder einer anderen Ressource fortgesetzt werden kann (engl. *preemption*). Liegt kein preemptives Scheduling vor, muß die gesamte Abarbeitung eines Prozesses ohne Unterbrechung erfolgen. Gerade bei stark unterschiedlicher Ausführungszeit und Prioritätenvergabe der beteiligten Prozesse kann die Unterbrechbarkeit notwendig sein, um die Echtzeitrandbedingungen einhalten zu können. Existiert beispielsweise ein Prozeß mit einer niedrigen Abtastrate und einer langen Ausführungszeit, muß er von einem Prozeß mit hoher Abtastrate und einer kürzeren Ausführungszeit unterbrochen werden können. Die Schedulingstrategie muß also Prozessen mit hoher Abtastrate eine höhere Priorität im Vergleich zu Prozessen mit einer niedrigeren Abtastrate zuordnen. Für periodische Tasks wurde gezeigt, daß dieses Vorgehen die optimale statische Schedulingstrategie darstellt [LiLa73]. Die Unterbrechbarkeit kostet allerdings zusätzliche Zeit (engl. *latency*), da neue Prozeßparameter zu laden sind. Man spricht hier von einem *Kontextwechsel.* Für Rapid Prototyping muß die oben vorgestellte Schedulingstrategie angewendet werden, um eine möglichst optimale Ausführung zu gewährleisten.

Da die Prozesse im allgemeinen miteinander interagieren müssen, muß auch der Informationsaustausch zwischen den beteiligten Prozessen beachtet werden. Dieser Vorgang wird als *Interaktion* bezeichnet. Grundsätzlich werden zwei Formen der Interaktion unterschieden:

- Im Fall der *Kooperation* tauschen die Prozesse Informationen über einen gemeinsamen Bereich im Speicher aus. Dieser Bereich wird als Shared Memory bezeichnet und stellt eine Überlappung der Datenbereiche dar, auf den sowohl schreibend als auch lesend zugegriffen werden kann. Um gleichzeitige Zugriffe zu vermeiden, werden Zugriffe auf diesen Bereich mit Semaphoren (siehe Kapitel 2.5) geschützt.
- Im Fall der *Kommunikation* findet ein echter “Informationsaustausch” zwischen den Prozessen statt. Dabei werden Informationen von dem lokalen Datenbereich eines Prozessen in den lokalen Datenbereich eines anderen Prozesses transportiert.

Um ein deterministisches Verhalten in einer Echtzeitumgebung zu gewährleisten, werden die Prozesse als gekapselte Objekte behandelt, die Eingaben und Ausgaben nur unter der Kontrolle des Schedulers zulassen. Die gekapselten Objekte haben eine begrenzte Ausführungszeit und bestehen aus vier Komponenten (Bild 7-5). Die Komponente *Synchronisation* ist für die Kommunikation des Prozesses mit dem Scheduler verantwortlich. Danach wird die Komponente *Eingabe* ausgeführt, die die Eingabevariablen aktualisiert. Die Komponente *Modell* arbeitet das logische Verhaltensmodell ab und speichert die Ausgabevariablen. In dieser Komponente können mehrere Modellteile enthalten sein, die im Rahmen einer Partitionierung auf die Komponenten verteilt werden können. Die Komponente *Ausgabe* stellt auf Anforderung des Schedulers die zwischengespeicherten Ausgabevariablen der Modellkomponente allen anderen Prozessen zur Verfügung. Würden Variablenänderungen innerhalb der Zeitspanne T_i ohne Kontrolle des Schedulers vorgenommen, würde dies zu einem nicht-deterministischen Verhalten führen, da andere Prozesse mit teilweise oder völlig geän-

derten Variablen rechnen könnten. Die Zwischenspeicherung der Ausgaben wird als *Double-Buffering* bezeichnet. Dieses Vorgehen ermöglicht jedem Prozeß die Verwendung von lokalem Speicher. Die Eingabevariablen werden vor der Ausführung des Prozesses in einen prozeß-eigenen Speicherbereich geschrieben. Die Eingabevariablen bleiben für die gesamte Dauer der Ausführung des Prozeßschritts gleich. Auch die Ausgabevariablen werden in einen eigenen, lokalen Speicherbereich geschrieben und erst nach Ablauf des Prozeßschritts für die restlichen Prozesse zur Verfügung gestellt. Nach diesem Ablauf wird die Verarbeitung wiederholt. Dies kann sofort nach Beendigung des Ablaufs geschehen oder nach einer gewissen Zeitspanne. In Kapitel 7.2 wird auf diese Möglichkeiten eingegangen. Dieses Vorgehen führt zu einer Unabhängigkeit der Prozeßdaten der ausgeführten Prozesse.

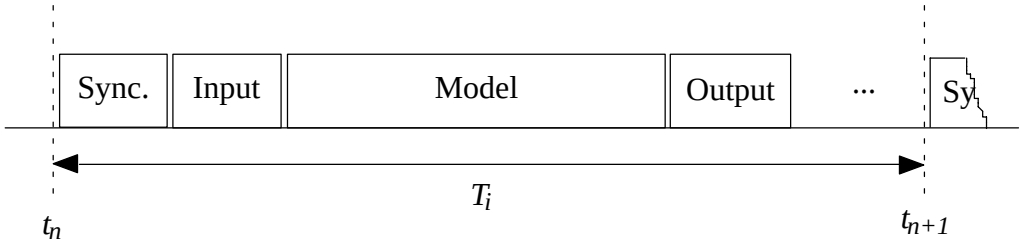


Bild 7-5: Innerer Aufbau eines Prozesses

Bei der Ausführung von Rapid Prototyping Applikationen wird in den meisten Fällen eine Ressourcenbeschränkung auftreten, da nicht für jeden Prozeß ein einzelner Prozessor vorhanden ist. Hauptsächlich betroffen sind funktionale Ressourcen (beispielsweise Prozessoren), während Speicher- und Kommunikations-Ressourcen weniger kritisch sind. Die Verwendung eines Multiprozessor-Echtzeitbetriebssystems verbessert zwar die Auslastung der funktionalen Ressourcen, führt jedoch zu einem hohen Kommunikationsaufwand. Wegen dieses Tradeoffs lohnt sich der Einsatz einer Multiprozessor-Architektur nicht immer. Im folgenden wird von der Abarbeitung auf einem Prozessor ausgegangen.

Um Algorithmen für diskrete und kontinuierliche Modelle lösen zu können, wird eine fein-granulare zeitliche Auflösung benötigt. Zur Abarbeitung von diskreten Modellen muß die Zeit bekannt sein, die für einen Übergang benötigt wird. Kontinuierliche Modelle, die zeitdiskret abgearbeitet werden, benötigen eine Zeitbasis, um Differentialgleichungen in Form von Differenzengleichungen abarbeiten zu können oder Signale erzeugen zu können.

Metriken

In dieser Arbeit wird die Notation nach Lawler [Lawl83] verwendet, die die Form $\alpha \mid \beta \mid \gamma$ besitzt. Dabei repräsentiert α die zur Ausführung zur Verfügung stehende Maschine (bei $\alpha = 1$ steht für die Abarbeitung ein Prozessor zur Verfügung), β gibt die charakteristischen Prozeßmerkmale an (z.B. gibt *prec* (engl. *precedence*) an, daß eine prioritätsbasierte Abarbeitung vorgenommen wird; *pmnt* (engl. *preemption*) gibt an, daß die Prozesse unterbrechbar sind; *resrc* (engl. *limited resources*) steht für limitierte Ressourcen; *indpt* (engl. *independence*) zeigt die Unabhängigkeit der Daten einzelner Prozesse an) und γ gibt die Optimierungsmethode (Metrik) der Abarbeitung an.

Es existieren unterschiedliche Metriken zur Abarbeitung der Prozesse, die je nach Anwendungsfall ausgewählt werden müssen. Klassische Schedulingstrategien benutzen beispielsweise die Summe der Abschlußzeiten, das Minimum der gewichteten Summe von Abschlußzeiten, das Minimum der Schedulinglänge, das Minimum der Anzahl der benötigten Prozessoren oder das Minimum der ma-

ximalen Verspätung als Metrik. Die für Echtzeitbetrachtungen wichtige Einbeziehung von Deadlines wird dabei im allgemeinen nicht betrachtet. Bezieht man Deadlines als zusätzliche Randbedingung ein, so kann man die Metriken nach den in den folgenden Abschnitten dargestellten Möglichkeiten klassifizieren [SSDB95].

Die Summe der Abschlußzeiten kann für Echtzeitsysteme nicht eingesetzt werden, da keine Einbeziehung von Zeitbedingungen (Deadlines oder Perioden) vorgenommen wird. Die gewichtete Summe von Abschlußzeiten kann hingegen von Interesse für Echtzeitsysteme sein, wenn nicht nur die Einhaltung von Deadlines im Vordergrund steht, sondern eine Kombination von Deadlines und Durchsatz betrachtet werden soll. Im Bereich Rapid Prototyping ist allerdings ausschließlich die Einhaltung der Deadlines wichtig. Die gewichtete Summe von Abschlußzeiten kann somit für diesen Fall nicht verwendet werden. Die minimale Schedulinglänge spielt eine bedeutende Rolle bei der Minimierung der benötigten Systemressourcen. Allerdings werden die Deadlines der einzelnen Prozesse nicht berücksichtigt. Ebenso wie die minimale Schedulinglänge kann die Minimierung der Anzahl der benötigten Prozessoren nicht für eine Echtzeitverarbeitung verwendet werden, da auch hier die Deadlines der einzelnen Prozesse nicht berücksichtigt werden. Das Minimum der maximalen Verspätung ist besonders nützlich, wenn zusätzliche Ressourcen hinzugefügt werden können, bis die maximale Verspätung kleiner gleich Null ist. In diesem Fall überschreitet kein Prozeß seine Deadline. Werden dem Prozeß weniger als die notwendigen Ressourcen zur Verfügung gestellt, kann nicht garantiert werden, daß nur die kleinste Anzahl an Prozessen ihre Deadline verfehlt. Tritt dieser Fall im Bereich Rapid Prototyping auf und ein Prozeß kann seine Deadline nicht erfüllen, muß die Verarbeitung jedoch in jedem Fall unterbrochen werden und in einen sicheren Notzustand übergegangen werden, da bei der Nichterfüllung einer Deadline kein deterministisches Verhalten der Applikation mehr garantiert werden kann.

Für Rapid Prototyping ist entscheidend, daß alle Prozesse im Rahmen ihrer Zeitbedingungen abgearbeitet werden können. Gemäß dieser Randbedingung kann die Minimierung der maximalen Verspätung L_{max} als geeignete Metrik verwendet werden.

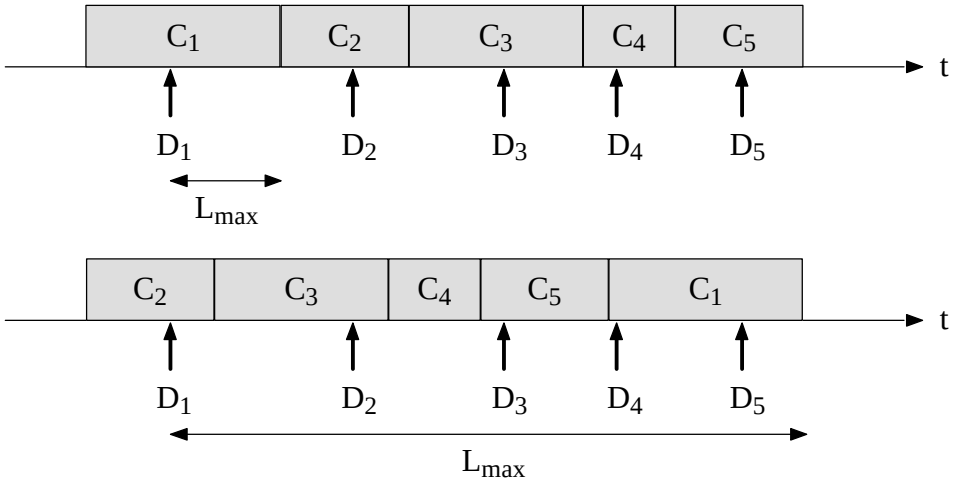


Bild 7-6: Maximale Verspätung

Definition 7.1: Die *Verspätung* eines Prozesses besteht aus dem Intervall zwischen Deadline und Abschlußzeit.

In Bild 7-6 ist ein Beispiel für die Überschreitung der Deadlines von Prozessen und die Betrachtung der maximalen Verspätung dargestellt [SSDB95]. Im oberen Fall wurde auf die Minimierung der maximalen Verspätung optimiert. Obwohl L_{max} wesentlich geringer ist als im unteren Fall, überschreiten alle Prozesse ihre Deadlines. Im unteren Fall, bei dem die maximale Verspätung nicht minimal ist, überschreitet nur Prozeß 1 seine Deadline. Da für Rapid Prototyping jedoch die Erfüllung aller Zeitbedingungen für jeden Prozeß im Vordergrund steht, ist bereits die Überschreitung einer Deadline nicht zu akzeptieren. Wichtig ist deswegen, einem Prozeß soviel Zeit wie möglich zur Beendigung seiner Abarbeitung zur Verfügung zu stellen. Dies wird durch die Verwendung der Minimierung der maximalen Verspätung (Bild 7-6 oben) erreicht.

Das im Bereich Rapid Prototyping zu lösende Problem kann also folgendermaßen klassifiziert werden:

$$1 \mid static,\ prec,\ pmtn,\ resrc,\ indpt \mid L_{max} \qquad (7.3)$$

7.2 Schedulingstrategien

In Kapitel 7.1 wurden die Anforderungen an das Scheduling für Rapid Prototyping Systeme festgelegt. Nun muß eine Schedulingstrategie erarbeitet werden, die eine Abarbeitung der Prozesse ohne Überschreiten einer Deadline gewährleistet. Als Schedulingstrategien kommen prinzipiell drei Verfahren in Frage: das ereignisgesteuerte Scheduling, das zeitgesteuerte Scheduling und das adaptive Scheduling, das auf Basis des ereignis- oder zeitgesteuerten Schedulings eine Anpassung der Schrittweite zur Laufzeit vornimmt. Wie bereits in Kapitel 7.1 erwähnt, kommt die adaptive Anpassung der Schrittweite, die zu einem dynamischen Scheduling führt, nicht in Frage. Im folgenden werden ereignis- und zeitgesteuertes Scheduling näher untersucht.

7.2.1 Ereignisgesteuertes Scheduling

Ereignisgesteuertes Scheduling eignet sich insbesondere für rein diskrete Systeme, die keine Synchronisation mit der realen Zeit benötigen. In diesem Fall existiert kein Systemtakt und die zeitliche Abfolge der Ereignisse besitzt nur ordnenden Charakter. Das Ereignis löst Systemreaktionen aus, die für die Verarbeitung der Reaktion eine bestimmte Berechnungsdauer in Anspruch nimmt. Die Berechnungsdauer hat keine feste, immer gleichbleibende Größe, sondern hängt vom Systemzustand und dem Ereignis ab. Tritt während der Verarbeitung eines Ereignisses ein weiteres Ereignis auf, wird dieses nicht sofort verarbeitet, sondern in einen Pufferspeicher geschrieben, der nach Reihenfolge des Eintreffens abgearbeitet wird.

Um eine Abarbeitung in Echtzeit zu gewährleisten, muß die Berechnungsdauer auf ein Ereignis deutlich unter den Echtzeitanforderungen liegen. Zum Nachweis der Echtzeit muß beim Eintreffen eines Ereignisses ein Timer gestartet werden, der überprüft, ob die Abarbeitung noch im Rahmen der geforderten Zeitspanne liegt. Diese Art des Echtzeitnachweises beruht auf einer vorherigen statistischen Auswertung der zeitlichen Verteilung von Ereignissen in dem System. Hierbei ergibt sich das Problem, daß Echtzeitsysteme immer innerhalb einer geforderten Frist ihre Abarbeitung beendet haben müssen, Statistiken hierfür jedoch keine gesicherten Werte liefern können, da nicht alle Fälle systematisch betrachtet werden. Bei Systemen, bei denen viele Ereignisse innerhalb kurzer Zeit auftreten, ist das ereignisgesteuerte Scheduling nicht anwendbar.

Bild 7-7 verdeutlicht dieses Problem. Beim Eintreffen von Ereignis 1 wird der zugehörige Prozeß P_1 ausgelöst. Noch bevor die Abarbeitung dieses Prozesses beendet ist, trifft Ereignis 2 ein. Der

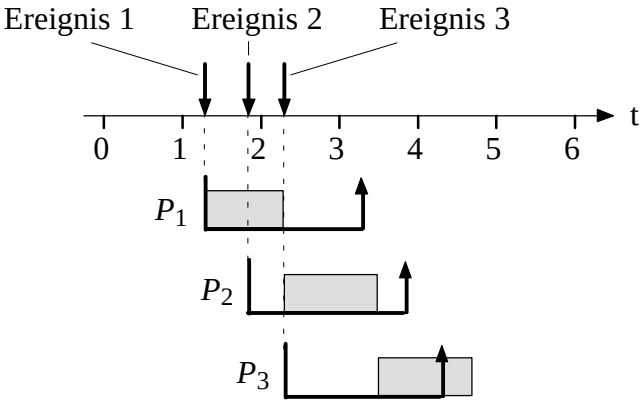


Bild 7-7: Echtzeitnachweis des ereignisgesteuerten Schedulings

zugehörige Prozeß P_2 kann jedoch noch nicht abgearbeitet werden, da zu diesem Zeitpunkt P_1 abgearbeitet wird. Ist die Abarbeitung von P_1 beendet, beginnt P_2 mit der Abarbeitung. Die für die Abarbeitung von P_2 einzuhaltende Zeitspanne steht somit nicht mehr vollständig zur Verfügung. Zu diesem Zeitpunkt trifft das Ereignis 3 ein. Da nun P_2 verarbeitet wird, kann P_3 nicht mit seiner Abarbeitung beginnen. Das Ende der Abarbeitung von P_2 findet innerhalb der Frist statt, die dem Prozeß zur Verfügung steht. Nun beginnt P_3 mit der Abarbeitung. Da dieser Prozeß jedoch wegen der verzögerten Abarbeitung von P_2 erst spät mit seiner Abarbeitung beginnen konnte, verfehlt er seine Abarbeitung innerhalb der ihm zur Verfügung stehenden Frist. Es tritt eine Verletzung der Echtzeitbedingungen auf.

Die Abtastraten zeitdiskreter kontinuierlicher Modelle oder die Taktraten synchroner Automaten geben zeitliche Rahmenbedingungen für die Abarbeitung von Teilmodellen vor. Bei der Simulation ist es möglich, diese zeitlichen Abläufe virtuell zu verändern. Wichtig ist dabei, daß die einzelnen Modellteile ihre Ergebnisse zu vorgegebenen Zeitpunkten austauschen können. Die Zeitpunkte selbst spielen nur eine untergeordnete Rolle. Bei der Abarbeitung einer Simulation kann aus diesem Grund eine Beschleunigung erreicht werden. Bei der Echtzeitverarbeitung hingegen stehen die Zeitpunkte und die unbedingte Einhaltung der vorgegebenen Fristen im Vordergrund. Rein ereignisgesteuertes Scheduling ist für die Abarbeitung dieser Prozesse ungeeignet, insbesondere wenn sowohl diskrete als auch kontinuierliche Prozesse betrachtet werden, da keine Zeitbasis zur Verfügung gestellt wird. Zur Echtzeitverarbeitung von kontinuierlichen Prozessen und der damit verbundenen Integrationsalgorithmen ist es notwendig, eine eigene Zeitbasis zur Verfügung zu stellen. Bei der Echtzeitverarbeitung von rein diskreten Prozessen ist der Einsatz einer Zeitbasis nicht zwingend erforderlich. Möchte man jedoch Aussagen über die Dauer von Übergängen bei diskreten Modellen vornehmen, muß ebenfalls eine Zeitbasis zur Verfügung gestellt werden. Die Überprüfung der Zeitbedingungen anhand einer Zeitbasis führt zu zeitgesteuertem Scheduling. Eine ebenfalls denkbare kombinierte Verwendung von ereignis- und zeitgesteuertem Scheduling bringt einen erhöhten Verwaltungsaufwand mit sich, so daß diese Möglichkeit für den Einsatz in einem Rapid Prototyping System nicht in Betracht gezogen wird. Im folgenden wird das zeitgesteuerte Scheduling für den Einsatz in einem Rapid Prototyping System eingeführt.

7.2.2 Zeitgesteuertes Scheduling

Um die Synchronisation zwischen System und Umgebung unter Beachtung von Echtzeitbedingungen zu vereinfachen, kann das zeitgesteuerte Scheduling verwendet werden. Im Gegensatz zum

ereignisgesteuerten Scheduling wird bei dieser Schedulingstrategie ein inkrementelles Fortschreiten der Zeit vorausgesetzt. Nach jeder Erhöhung der Zeit werden die innerhalb des letzten Inkrements aufgetretenen Ereignisse ausgewertet und ein neuer Systemzustand ermittelt. Ein einzelnes Inkrement wird als *Modellschritt* bezeichnet. Die innerhalb eines Modellschritts auftretende Verarbeitung kann nochmals in sog. *Mikroschritte* unterteilt werden. Modellschritte werden zu jedem Zeitpunkt ausgeführt, d.h. im Gegensatz zum ereignisgesteuerten Scheduling auch dann, wenn keine Ereignisse im jeweiligen Modellschritt stattfanden. Die Länge eines Modellschritts, die als *Modellschrittweite* bezeichnet wird, hat entscheidenden Einfluß auf die Genauigkeit, Geschwindigkeit und damit auch auf die Korrektheit der Verarbeitung.

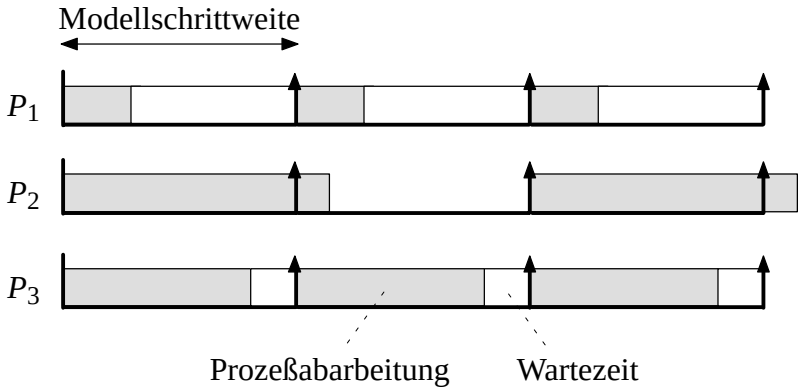


Bild 7-8: Wahl der Modellschrittweite

In Bild 7-8 ist dargestellt, wie entscheidend die Wahl der Modellschrittweite für die Abarbeitung ist. Der Prozeß P_1 besitzt im Verhältnis zur Prozeßabarbeitung eine lange Wartezeit. Damit kann die Genauigkeit der Berechnung sinken und die Abarbeitungsgeschwindigkeit ist langsam. Sinkt die Genauigkeit zu stark, so kann die Abarbeitung zu fehlerhaften Ergebnissen führen. Minimiert man die Modellschrittweite im Verhältnis zur Prozeßabarbeitung zu stark, so kann die Modellschrittzeit wie bei Prozeß P_2 zu kurz werden und es tritt eine Verletzung der Echtzeitbedingungen auf. Prozeß P_3 besitzt eine gute Annäherung der Prozeßabarbeitung an die Modellschrittweite ohne die Modellschrittweite zu überschreiten. Hierbei sei angemerkt, daß die Dauer der Prozeßabarbeitung für jeden Schritt unterschiedlich ausfallen kann (siehe Kapitel 7.3).

Da nach Definition der Echtzeit in Kapitel 2.5 die Eigenschaft verstanden wird, daß die Reaktionszeit eines Echtzeitsystems unter keinen Umständen über eine vorgegebene Frist hinausgeht, muß auch beim zeitgesteuerten Scheduling ein Echtzeitnachweis geführt werden. Bei der Wahl der Modellschrittweite muß gewährleistet werden, daß die Modellschrittweite größer ist als die Summe der ungünstigsten Fälle aller in diesem Inkrement abzuarbeitenden Prozesse (siehe Gleichung (7.4)). Da beim zeitgesteuerten Scheduling das Zeitinkrement durch das Echtzeitbetriebssystem bestimmt werden kann, bietet es sich an, dieses Zeitinkrement auch zur Prüfung der Echtzeitbedingungen einzusetzen.

Ratenmonotones Scheduling

Für den Fall der schnellen Abarbeitung einer begrenzten Anzahl von Prozessen bei nur einer funktionalen Ressource, eignet sich insbesondere die ratenmonotone Schedulingstrategie (engl. *rate-monotonic scheduling*). Für den bei Rapid Prototyping vorliegenden Fall gilt, daß die Deadline einer Periode immer der Dauer der Periode entspricht. Eine frühere Abarbeitung ist zulässig, doch wird wegen der Betrachtung der Prozesse als gekapselte Objekte erst zum Ende einer Deadline eine

Interaktion mit anderen Prozessen durchgeführt. Im Gegensatz zur deadlinemonotonen Schedulingstrategie (engl. *deadline-monotonic scheduling*), bei der *Ausführungszeit* $E_i \leq$ *Deadline* $D_i \leq$ *Periode* T_i ist, gilt bei der ratenmonotonen Schedulingstrategie:

$$E_i \leq D_i = T_i \qquad (7.4)$$

Definition 7.2: Für eine Prozeßschar, die nach einer beliebigen Schedulingstrategie abgearbeitet wird, tritt dann ein *Überlauf* auf, wenn die Deadline eines nicht beendeten Prozesses überschritten wird. Dabei gilt $E_i > D_i$.

Da ein ratenmonotones Scheduling betrachtet werden soll, ist zur Bestimmung der besten Strategie die Einführung des kritischen Moments (engl. *critical instant*) wichtig [LiLa73].

Definition 7.3: Der *kritische Moment* eines Prozesses ist der Moment, in der eine Anforderung an einen Prozeß die größte Antwortzeit ergibt.

Aus dieser Definition kann das kritische Intervall abgeleitet werden.

Definition 7.4: Das *kritische Intervall* eines Prozesses ist das Intervall, das zwischen dem kritischen Moment und dem Ende der Abarbeitung der zugehörigen Prozeßanforderung liegt.

Um den kritischen Moment eines Prozesses bestimmen zu können, legt das folgende Theorem die Randbedingungen fest.

Theorem 7.1: Der kritische Moment jedes Prozesses tritt auf, wenn der Prozeß im gleichen Augenblick wie alle weiteren Prozesse angefordert wird, die eine höhere Priorität besitzen.

Der Beweis zu Theorem 7.1 kann folgendermaßen geführt werden. Angenommen, es existieren m Prozesse $P_1, P_2, ..., P_m$, die in absteigender Priorität geordnet sind. Zum Zeitpunkt t_1 tritt eine Anforderung an P_m auf. Weiterhin werde angenommen, daß zwischen t_1 und $t_1 + T_m$ zusätzliche Anforderungen an alle Prozesse P_i, $i < m$ auftreten, die zu den Zeitpunkten t_2, $t_2 + T_i$, $t_2 + 2T_i$, ..., $t_2 + kT_i$ ausgeführt werden. Die Unterbrechung des Prozesses P_m durch die höherprioren Prozesse P_i wird zu einer Verzögerung bei der Beendigung von P_m führen, falls der Prozeß P_m nicht bereits vor t_2 beendet ist. Startet man die Prozesse P_i erst zu einem späteren Zeitpunkt, jedoch vor Ablauf von $t_1 + T_m$, so ergibt sich keine beschleunigte Abarbeitung von P_m. Verschiebt man t_2 in Richtung t_1, so gilt die Behauptung, daß die Verzögerung von P_m am größten ist, wenn t_2 mit t_1 zusammenfällt. Wiederholt man diese Vorgehensweise für alle P_i, $i = 2, ..., m - 1$, so ist Theorem 7.1 bewiesen. □

Mit diesem Theorem kann durch eine einfache Rechnung bestimmt werden, ob eine vorgegebene Prioritätsverteilung zur Durchführbarkeit (engl. *feasibility*) einer Schedulingstrategie führt.

Definition 7.5: Unter der *Durchführbarkeit* einer Schedulingstrategie versteht man das Scheduling einer Prozeßschar in der Art, daß die gewählte Schedulingstrategie zu keinem Überlauf führt.

Als Beispiel für die Durchführbarkeit einer Schedulingstrategie werden die beiden in Bild 7-9 dargestellten Prozesse P_1 und P_2 mit den Ausführungszeiten $C_1 = 1$ und $C_2 = 1$ sowie den Perioden $T_1 = 2$ und $T_2 = 5$. Vergibt man an den Prozeß P_1 eine höhere Priorität als an Prozeß P_2, so er-

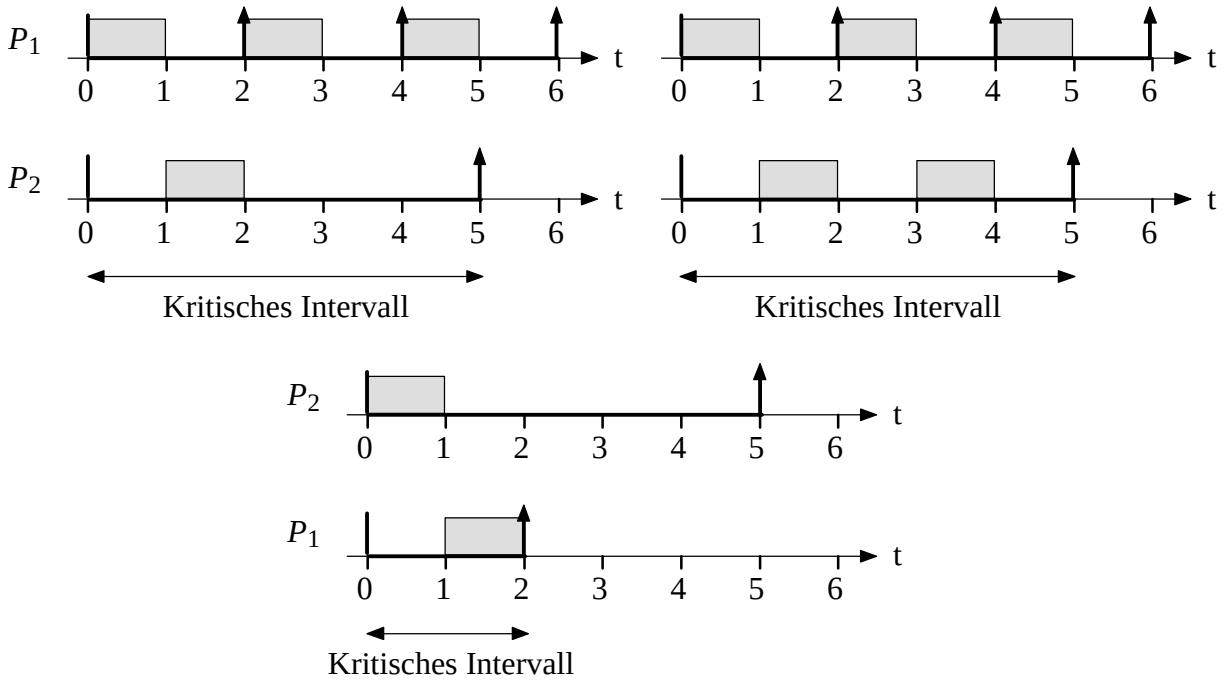


Bild 7-9: Schedulingstrategie mit unterschiedlichen Prioritäten

kennt man aus Bild 7-9 links oben, daß diese Prioritätenvergabe zur Durchführbarkeit der Schedulingstrategie führt. Weiterhin kann man erkennen, daß der Prozeß P_2 höchstens zu $C_2 = 2$ erhöht werden kann (Bild 7-9 rechts oben). Kehrt man die Prioritäten um und weist dem Prozeß P_2 die höhere Priorität zu, so kann keine der Ausführungszeiten C_1 oder C_2 einen höheren Wert als 1 annehmen. Das kritische Intervall dieser Prioritätsverteilung ist im unteren Teil von Bild 7-9 ist dargestellt.

Es ist also anzunehmen, daß aus Theorem 7.1 auch eine optimale Schedulingstrategie abgeleitet werden kann. Betrachtet man hierzu zwei Prozesse P_1 und P_2 mit $T_1 < T_2$, wobei der Prozeß P_1 die höhere Priorität aufweise, so muß nach Theorem 7.1 die folgende Bedingung erfüllt sein:

$$\left\lfloor \frac{T_2}{T_1} \right\rfloor C_1 + C_2 \leq T_2 \qquad (7.5)$$

$\lfloor x \rfloor$ beschreibt dabei die größte ganze Zahl, die kleiner oder gleich x ist. Entsprechend beschreibt $\lceil x \rceil$ die kleinste ganze Zahl, die größer oder gleich x ist. Zu beachten ist, daß Gleichung (7.5) notwendig, jedoch nicht hinreichend ist, um die Durchführbarkeit des Schedulings zu gewährleisten [LiLa73]. Weist hingegen der Prozeß P_2 die höhere Priorität auf, so gilt:

$$C_1 + C_2 \leq T_1 \qquad (7.6)$$

Da aber gleichzeitig

$$\left\lfloor \frac{T_2}{T_1} \right\rfloor C_1 + \left\lfloor \frac{T_2}{T_1} \right\rfloor C_2 \leq \left\lfloor \frac{T_2}{T_1} \right\rfloor T_1 \leq T_2 \qquad (7.7)$$

gilt, ergibt sich aus Gleichung (7.6) Gleichung (7.5). Dies bedeutet, daß falls zwei Prozesse P_1 und P_2 mit $T_1 < T_2$ und vorgegebenen Ausführungszeiten C_1 und C_2 existieren, und ein Scheduling bei höherer Priorität von P_2 durchführbar ist, auch ein Scheduling mit höherer Priorität von P_1 durchführbar ist. Die Umkehrung gilt jedoch nicht. Aus dieser Feststellung folgt, daß eine höhere

Priorität dem Prozeß mit der kürzeren Periode zugewiesen werden sollte (in obigem Fall P_1) unabhängig von der Ausführungsdauer.

Definition 7.6: Vergibt man an Prozesse mit kürzerer Periode höhere Prioritäten im Vergleich zu Prozessen mit längerer Periode, so wird die Prioritätsverteilung als *ratenmonotone Prioritätsverteilung* (engl. *rate-monotonic priority assignment*) bezeichnet.

Weiter folgt nach [LiLa73] das Theorem 7.2:

Theorem 7.2: Existiert eine durchführbare Prioritätsverteilung für eine Prozeßschar, ist auch eine ratenmonotone Prioritätsverteilung für diese Prozeßschar durchführbar.

Die ratenmonotone Schedulingstrategie kann also, verglichen mit allen Algorithmen mit statischen Prioritäten, eine gültige Schedulingstrategie bestimmen, falls eine gültige Strategie existiert. Der Beweis zu Theorem 7.2 ist folgendermaßen zu führen: Angenommen, es existieren *m* Prozesse P_1, P_2, ..., P_m, die eine eindeutig durchführbare Prioritätsverteilung besitzen. Dabei seien P_i und P_j zwei Prozesse, die aufeinander folgende Prioritäten besitzen, wobei P_i die höhere Priorität aufweise. Weiterhin gelte $T_i > T_j$. Vertauscht man die Prioritäten von P_i und P_j, kann nach Theorem 7.1 erkannt werden, daß die resultierende Prioritätsverteilung immer noch durchführbar ist. Weil die ratenmonotone Prioritätsverteilung aus jeder beliebigen Prioritätsverteilung durch paarweises Vertauschen der Prioritäten zweier aufeinander folgenden Prozesse erreicht werden kann, ist Theorem 7.2 bewiesen. □

Die maximal erreichbare Prozessorausnutzung *U* (engl. *processor utilization factor*) ist als Summe derjenigen Prozessorzeitscheiben definiert, die für die Ausführung einer Prozeßschar benötigt wird. Da der Quotient $\frac{C_i}{T_i}$ die Prozessorzeitscheibe für die Ausführung eines Prozesses P_i beschreibt, folgt für die Prozessorausnutzung von *n* Prozessen:

$$U = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \qquad (7.8)$$

Im folgenden wird die maximale Prozessorausnutzung mit ratenmonotonem Scheduling betrachtet. Je höher die Auslastung ausfällt, desto optimaler ist die Ausnutzung der funktionalen Ressourcen gelungen. Die Prozessorausnutzung kann zwar durch Vergrößerung der Werte von C_i oder durch Verkleinerung der Werte von T_i vergrößert werden. Jedoch existiert eine obere Schranke, die durch die Erfüllung der Fristen der jeweiligen Prozesse vorgegeben ist.

Definition 7.7: Eine Prozeßschar nutzt den Prozessor genau dann *vollständig* aus, wenn das Scheduling unter einer gegebenen Prioritätsverteilung durchführbar ist und ein Anstieg der Ausführungszeiten eines beliebigen Prozesses das Scheduling bei jeder Prioritätsverteilung verhindert.

Des weiteren kann eine kleinste obere Grenze der Prozessorausnutzung angegeben werden.

Definition 7.8: Die *kleinste obere Grenze* der Prozessorausnutzung ist durch das Minimum der Prozessorausnutzung aller Prozesse gegeben, die den Prozessor vollständig ausnutzen.

Für das Scheduling in einem Rapid Prototyping System ist diese kleinste obere Grenze das entscheidende Kriterium, da hierdurch die maximal mögliche Abtastrate bestimmt wird. Eine Prozessoraus-

nutzung oberhalb dieser Grenze ist nur dann möglich, wenn T_i vergrößert werden oder der Wert von C_i durch eine weiter optimierte Modellierung (oder Codeerzeugung) verkleinert werden kann. Meist steht dieser Wert jedoch als feste, nicht veränderbare Größe zur Verfügung, die stochastisch ermittelt werden muß.

Wie bereits gezeigt, kann die Prozessorausnutzung einer ratenmonotonen Prioritätsverteilung nicht durch eine andere Prioritätsverteilung übertroffen werden, da die ratenmonotone Prioritätsverteilung bereits das Optimum darstellt. Im folgenden wird deshalb mit einer ratenmonotonen Prioritätsverteilung die kleinste obere Grenze für alle Ausführungszeiten und Prozeßanforderungen zu bestimmen sein. Dies wird zuerst für zwei Prozesse bestimmt und danach für eine beliebige Anzahl von Prozessen verallgemeinert.

Theorem 7.3: Für eine Prozeßschar mit beliebiger statischer Prioritätsverteilung kann die kleinste obere Grenze der Prozessorausnutzung durch $U = 2\left(2^{\frac{1}{2}} - 1\right) \approx 0,828$ bestimmt werden.

Der Beweis zu diesem Theorem kann mit Hilfe von zwei Prozessen P_1 und P_2 geführt werden, die die Perioden T_1 und T_2 und die Ausführungszeiten C_1 und C_2 besitzen. Es werde angenommen, daß $T_2 > T_1$ sei. Gemäß der ratenmonotonen Prioritätsverteilung muß also P_1 eine höhere Priorität im Vergleich zu P_2 aufweisen. Im kritischen Intervall von P_2 treten $\left\lceil \frac{T_2}{T_1} \right\rceil$ Anforderungen von P_1 auf. C_2 wird dabei so gewählt, daß eine vollständige Prozessorausnutzung innerhalb des kritischen Intervalls auftritt. Nach [LiLa73] sind dann zwei Fälle zu unterscheiden. Im ersten Fall ist C_1 klein genug, um zu gewährleisten, daß alle Anforderungen von P_1 vor Auftreten der zweiten Anforderung von P_2 abgearbeitet sind. Aus dieser Annahme folgt Gleichung (7.9).

$$C_1 \leq T_2 - T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor \qquad (7.9)$$

Der größte mögliche Wert von C_2 beträgt also:

$$C_2 = T_2 - C_1 \left\lceil \frac{T_2}{T_1} \right\rceil \qquad (7.10)$$

Somit beträgt die Prozessorausnutzung U:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = 1 + C_1 \left(\frac{1}{T_1} - \frac{1}{T_2} \left\lceil \frac{T_2}{T_1} \right\rceil \right) \qquad (7.11)$$

In diesem Fall sinkt die Prozessorausnutzung monoton in C_1, da der abzuziehende Faktor stets kleiner als der zu addierende Faktor ist.

Für den zweiten Fall gelte, daß sich die Ausführung der $\left\lceil \frac{T_2}{T_1} \right\rceil$ ten Anforderung von P_1 mit der zweiten Anforderung von P_2 überlappt. In diesem Fall gilt für C_1:

$$C_1 \geq T_2 - T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor \qquad (7.12)$$

Hieraus folgt, daß der größte mögliche Wert von C_2 gemäß Gleichung (7.13) bestimmt werden kann.

$$C_2 = -C_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor + T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor \qquad (7.13)$$

Aus Gleichung (7.13) kann die Prozessorausnutzung U bestimmt werden:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{T_1}{T_2} \left\lfloor \frac{T_2}{T_1} \right\rfloor + C_1 \left(\frac{1}{T_1} - \frac{1}{T_2} \left\lfloor \frac{T_2}{T_1} \right\rfloor \right) \qquad (7.14)$$

In diesem Fall steigt die Prozessorausnutzung monoton in C_1. Da sich im ersten Fall eine monoton fallende und im zweiten Fall eine monoton steigende Funktion ergibt, tritt das Minimum der Prozessorausnutzung an der Grenze zwischen beiden Fällen auf. Mit

$$C_1 = T_2 - T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor \qquad (7.15)$$

ergibt sich für die die Prozessorausnutzung mittels Einsetzen von C_1 in Gleichung (7.11):

$$\begin{aligned} U &= 1 + \left(T_2 - T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor \right) \left(\frac{1}{T_1} - \frac{1}{T_2} \left\lceil \frac{T_2}{T_1} \right\rceil \right) \\ &= 1 + \frac{T_2}{T_1} - \left\lceil \frac{T_2}{T_1} \right\rceil - \left\lfloor \frac{T_2}{T_1} \right\rfloor + \frac{T_1}{T_2} \left\lceil \frac{T_2}{T_1} \right\rceil \left\lfloor \frac{T_2}{T_1} \right\rfloor \\ &= 1 - \frac{T_1}{T_2} \left(\left\lceil \frac{T_2}{T_1} \right\rceil - \frac{T_2}{T_1} \right) \left(\frac{T_2}{T_1} - \left\lfloor \frac{T_2}{T_1} \right\rfloor \right) \end{aligned} \qquad (7.16)$$

Mit dieser Gleichung ist eine allgemeine Abhängigkeit zwischen U und T_2 und T_1 hergestellt. Nun muß das Minimum von U gesucht werden. Gleichung (7.16) kann mit Hilfe von den beiden Hilfsvariablen $I = \left\lfloor \frac{T_2}{T_1} \right\rfloor$ und $f = \frac{T_2}{T_1} - \left\lfloor \frac{T_2}{T_1} \right\rfloor$ umgewandelt werden:

$$\begin{aligned} U &= 1 - \frac{T_1}{T_2} \left(\frac{T_2}{T_1} - \left\lfloor \frac{T_2}{T_1} \right\rfloor \right) \left(\left\lceil \frac{T_2}{T_1} \right\rceil - \frac{T_2}{T_1} \right) \\ &= 1 - \frac{\left(\frac{T_2}{T_1} - \left\lfloor \frac{T_2}{T_1} \right\rfloor \right) \left(1 + \left\lfloor \frac{T_2}{T_1} \right\rfloor - \frac{T_2}{T_1} \right)}{\left\lfloor \frac{T_2}{t_1} \right\rfloor + \frac{T_2}{T_1} - \left\lfloor \frac{T_2}{t_1} \right\rfloor} \\ &= 1 - \frac{f(1-f)}{I+f} \end{aligned} \qquad (7.17)$$

Da $0 \le f \le 1$ ist und, wie aus Bild 7-10 zu erkennen ist, U monoton mit I fällt, erhält man das kleinste U für den kleinsten möglichen Wert von I. Für $I = 1$ ergibt sich somit Gleichung (7.18).

$$U = 1 - \frac{f(1-f)}{1+f} \qquad (7.18)$$

Minimiert man U nach f, so ergibt sich:

$$\frac{\delta U}{\delta f} = \frac{(1-2f)(1+f) - (f+f^2)}{(1+f)^2} = \frac{1-2f-f^2}{(1+f)^2} \qquad (7.19)$$

Das Minimum von U ergibt sich zu:

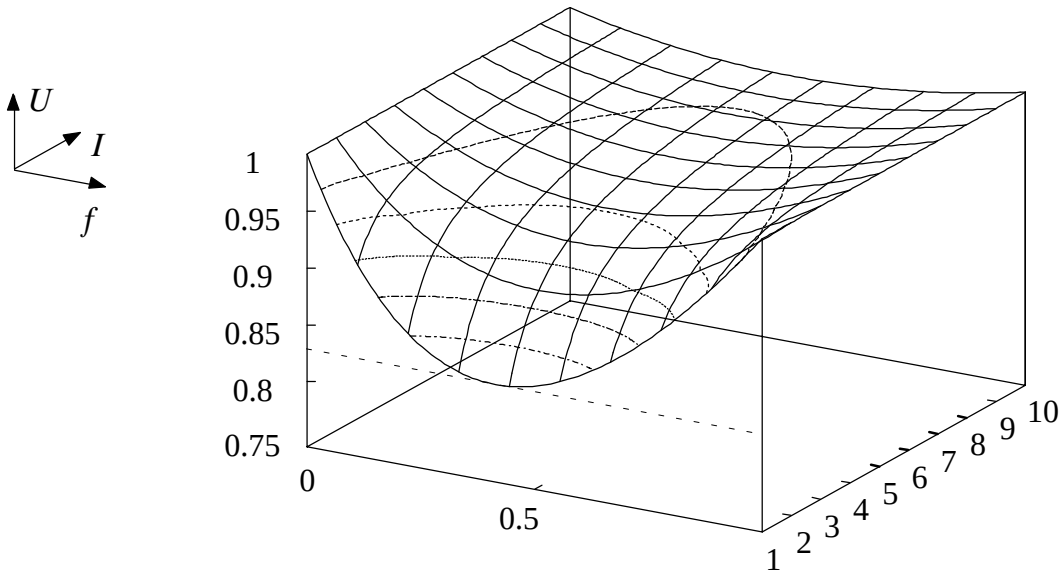


Bild 7-10: Grafische Repräsentation von Gleichung (7.17)

$$f_{\min} = \frac{2 \pm \sqrt{8}}{-2} = -1 \pm \sqrt{2} \qquad (7.20)$$

Da nur der positive Fall zugelassen ist, ergibt sich für $U_{\min}$:

$$U_{\min} = 2\left(\sqrt{2} - 1\right) \approx 0,828 \qquad (7.21)$$

Damit ist Theorem 7.3 bewiesen. An dieser Stelle sei angemerkt, daß die Prozessorausnutzung $U = 1$ wird, wenn $f = 1$ gilt. Das ist genau dann der Fall, wenn die Periode des niedriger priorisierten Prozesses ein Vielfaches der Periode des höher priorisierten Prozesses annimmt. Dies wird insbesondere für die Einführung des pseudoraten-basierten Schedulings im nächsten Teilkapitel zu beachten sein. □

Im folgenden muß Theorem 7.3, das nur für zwei Prozesse Gültigkeit besitzt, auf eine beliebige Anzahl von Prozessen erweitert werden. Zuerst wird der allgemeine Fall auf ein maximales Verhältnis der Anforderungsperioden von zwei zwischen zwei Anforderungen eingeschränkt.

Theorem 7.4: Für eine Prozeßschar von n Prozessen mit einer festgelegten Prioritätsordnung und der Einschränkung, daß das Verhältnis von zwei Anforderungsperioden geringer als zwei ist, kann die kleinste obere Grenze der Prozessorausnutzung durch $U = n\left(2^{\frac{1}{n}} - 1\right)$ bestimmt werden.

Der Beweis zu Theorem 7.4 kann mit Hilfe von n Prozessen P_1, P_2, ..., P_n geführt werden, die die Ausführungszeiten C_1, C_2, ..., C_n und den Perioden $T_n > T_{n-1} > ... > T_2 > T_1$ besitzen. Die Ausführungszeiten seien dabei so gewählt, daß die Prozesse den Prozessor vollständig ausnutzen und dabei die Prozessorausnutzung U minimieren.

Für die Berechnung von C_1 kann Gleichung (7.22) angenommen werden:

$$C_1 = T_2 - T_1 + \Delta, \quad \Delta > 0 \qquad (7.22)$$

Es seien:

$$\begin{aligned} C_1' &= T_2 - T_1 \\ C_2' &= C_2 + \Delta \\ C_3' &= C_3 \\ &\vdots \\ C_{n-1}' &= C_{n-1} \\ C_n' &= C_n \end{aligned} \qquad (7.23)$$

Aufgrund der obigen Annahmen werden auch C_1', C_2', ..., C_{m-1}', C_m' eine vollständige Prozessorausnutzung gewährleisten. Die Prozessorausnutzung für diese Prozeßschar sei mit U' bezeichnet. Für die Differenz der Prozessorausnutzungen folgt Gleichung (7.24).

$$U - U' = \frac{\Delta}{T_1} - \frac{\Delta}{T_2} > 0 \qquad (7.24)$$

Nun führt man eine zweite Prozeßschar ein, für die folgende Annahme gelte:

$$C_1 = T_2 - T_1 - \Delta \qquad (7.25)$$

Weiter seien:

$$\begin{aligned} C_1'' &= T_2 - T_1 \\ C_2'' &= C_2 - 2\Delta \\ C_3'' &= C_3 \\ &\vdots \\ C_{n-1}'' &= C_{n-1} \\ C_n'' &= C_n \end{aligned} \qquad (7.26)$$

Auch die Prozeßschar C_1'', C_2'', ..., C_{m-1}'', C_m'' soll eine vollständige Prozessorausnutzung gewährleisten. Die Prozessorausnutzung für diese Prozeßschar wird mit U'' bezeichnet. Für die Differenz der Prozessorausnutzungen folgt Gleichung (7.27).

$$U - U'' = -\frac{\Delta}{T_1} + \frac{2\Delta}{T_2} > 0 \qquad (7.27)$$

Aus den obigen Gleichungen kann geschlossen werden, daß falls U tatsächlich die kleinste Prozessorausnutzung gewährleistet, folgende Gleichung gelte:

$$C_1 = T_2 - T_1 \qquad (7.28)$$

Auf die gleiche Weise läßt sich dies auch auf die weiteren Gleichungen übertragen:

$$\begin{aligned} C_2 &= T_3 - T_2 \\ C_3 &= T_4 - T_3 \\ &\vdots \\ C_{n-1} &= T_n - T_{n-1} \end{aligned} \qquad (7.29)$$

Daraus ergibt sich:

$$C_n = T_n - 2\left(C_1 + C_2 + ... + C_{n-1}\right) \qquad (7.30)$$

Nun führt man zur Vereinfachung die Hilfsvariable g_i ein:

$$g_i = \frac{T_n - T_i}{T_i}, \quad i = 1, 2, \ldots, n \tag{7.31}$$

Mit dieser Hilfsvariablen ergibt sich C_i zu:

$$C_i = T_{i+1} - T_i = g_i T_i - g_{i+1} T_{i+1}, \quad i = 1, 2, , \ldots, n-1 \tag{7.32}$$

und C_n zu:

$$C_n = T_n - 2 g_1 T_1 \tag{7.33}$$

Auf die Prozessorausnutzung U bezogen ergibt sich:

$$\begin{aligned} U &= \sum_{i=1}^{n} \frac{C_i}{T_i} = \sum_{i=1}^{n-1} \left(g_i - g_{i+1} \frac{T_{i+1}}{T_i} \right) + 1 - 2 g_1 \frac{T_1}{T_n} \\ &= \sum_{i=1}^{n-1} \left(g_i - g_{i+1} \frac{g_i + 1}{g_{i+1} + 1} \right) + 1 - 2 \frac{g_1}{g_1 + 1} \\ &= 1 + g_1 \frac{g_1 - 1}{g_1 + 1} + \sum_{i=2}^{n-1} g_i \frac{g_i - g_{i-1}}{g_i + 1} \end{aligned} \tag{7.34}$$

Wie in Gleichung (7.18) wird die Prozessorausnutzung zu 1, wenn $g_i = 1$ ist. Somit wird auch hier Gleichung (7.34) nach g_j minimiert, um die kleinste obere Grenze der Prozessorausnutzung zu erhalten. Dazu wird die erste Ableitung von U nach g_j gebildet.

$$\frac{\delta U}{\delta g_j} = \frac{g_j^2 + 2g_j - g_{j-1}}{\left(g_j + 1 \right)^2} - \frac{g_{j+1}}{g_{j+1} + 1} = 0 \tag{7.35}$$

Die Lösung von Gleichung (7.35) ergibt sich zu:

$$g_j = 2^{\frac{n-j}{n}} - 1, \quad j = 0, 1, \ldots, n-1 \tag{7.36}$$

Aus Gleichung (7.36) folgt:

$$U = n \left(2^{\frac{1}{n}} - 1 \right) \tag{7.37}$$

Mit Gleichung (7.37) ist Theorem 7.4 bewiesen. □

Unschön ist allerdings, daß noch eine Einschränkung auf das Verhältnis von zwei Anforderungsperioden existiert, das bei weniger als zwei liegen muß. Um diese Einschränkung aufzuheben, muß Theorem 7.5 bewiesen werden.

Theorem 7.5: Für eine Prozeßschar von n Prozessen mit einer festgelegten Prioritätsordnung kann die kleinste obere Grenze der Prozessorausnutzung durch $U = n \left(2^{\frac{1}{n}} - 1 \right)$ bestimmt werden.

Der Beweis zu Theorem 7.5 kann mit Hilfe von n Prozessen $P_1, P_2, \ldots, P_n$ geführt werden, die eine vollständige Prozessorausnutzung gewährleisten sollen. Für beliebige i sei $\left\lfloor \frac{T_n}{T_i} \right\rfloor > 1$ und damit gilt $T_n + qT_i + r, \quad q > 1, r \geq 0$. Im folgenden ersetzt man den Prozeß P_i durch P_i', so daß $T_i' = qT_i$ und $C_i' = C_i$ gilt, und erhöht C_n so, daß eine vollständige Prozessorausnutzung vorliegt.

Die Erhöhung besteht hauptsächlich aus $C_i\,(q-1)$ und damit aus der Zeitspanne des kritischen Intervalls von P_n, die von P_i aber nicht von $P_i^{'}$ besetzt wird. Mit $U^{'}$ als Prozessorausnutzung des Prozesses $P_i^{'}$ ergibt sich:

$$U^{'} < U + (q-1)\,\frac{C_i}{T_n} + \frac{C_i}{T_i^{'}} - \frac{C_i}{T_i}$$

$$U^{'} \leq U + (q-1)\,C_i\left(\frac{1}{qT_i + r} - \frac{1}{qT_i}\right) \qquad (7.38)$$

Da $q - 1 > 0$ und $\frac{1}{qT_i + r} - \frac{1}{qT_i} \leq 0$ ist, gilt $U^{'} \leq U$. Aus dieser Beziehung kann geschlossen werden, daß nur Prozeßscharen betrachtet werden müssen, deren Verhältnis von Anforderungsperioden weniger als zwei beträgt, um die kleinste obere Grenze der Prozessorausnutzung zu bestimmen. Damit ist Theorem 7.5 bewiesen.

Durch Theorem 7.5 kann unter Annahme einer ratenmonotonen Prioritätsverteilung die kleinste obere Grenze in Abhängigkeit der Anzahl der verwendeten Prozesse aufgestellt werden. Für eine hohe Anzahl an Prozessen nimmt die Prozessorausnutzung den folgenden Wert an: $\lim_{n\to\infty} n\left(2^{\frac{1}{n}} - 1\right) = \ln 2 \approx 0,693$. Zur Verwendung von Theorem 7.5 muß angemerkt werden, daß die ermittelte kleinste obere Grenze eine notwendige, jedoch keine hinreichende Bedingung darstellt. Dies bedeutet für die Abschätzung, daß die Abschätzung schlechter als notwendig ausfällt und somit einen pessimistischen Charakter aufweist. Der Vorteil dieses Theorems liegt in der schnellen Abschätzbarkeit, ob ein ratenmonotones Scheduling möglich ist.

Das Anwendungsbeispiel in Tabelle 7-1 verdeutlicht diesen Vorteil. Die Prozeßschar besteht aus fünf Prozessen, die eine unterschiedliche Periode T_i aufweisen. Für die ersten vier Prozesse kann ein erfolgreiches Scheduling gefunden werden, da die tatsächliche Prozessorausnutzung unterhalb der kleinsten oberen Grenze liegt ($0,700 \leq 0,756$). Nimmt man jedoch den fünften Prozeß hinzu, so wird die kleinste obere Grenze verletzt. Dies bedeutet jedoch nach Theorem 7.5 nicht zwangsläufig, daß für die fünf Prozesse kein durchführbares Scheduling existiert, sondern nur, daß keine Aussage über die Durchführbarkeit vorgenommen werden kann. Theorem 7.5 eignet sich also genau dann, wenn die Prozessorausnutzung unterhalb der oberen Grenze liegt. Der Rechenaufwand zur Bestimmung dieser Aussage ist sehr gering.

Prozeß i	**Ausführungszeit E_i**	**Periode T_i**	**Verhältnis E_i/T_i'**	**Prozessorausnutzung (1 .. N)**	**Obere Grenze nach Theorem 7.5**
1	45	180	0,250	0,250	1,000
2	30	200	0,150	0,400	0,828
3	15	300	0,050	0,450	0,779
4	100	400	0,250	0,700	0,756
5	40	400	0,100	0,800	0,743

Tabelle 7-1: Beispiel für die Anwendung von Theorem 7.5

Im allgemeinen kann der ratenmonotone Schedulingalgorithmus jedoch eine höhere Prozessorausnutzung erreichen, als mit Theorem 7.5 ermittelt werden kann, da mit dem Theorem nur eine notwendige Bedingung formuliert wird. Praktische Messungen ergaben, daß größere Prozeßscharen

durchaus eine Prozessorausnutzung von 90% bei Verwendung eines ratenmonotonen Schedulings erreichen können. Das Verhalten ist in hohem Maße abhängig von den Werten der Perioden, die die Prozesse besitzen. Um genauere Aussagen über die Durchführbarkeit des Schedulings einer Prozeßschar in dem Bereich oberhalb der Grenze, die Theorem 7.5 vorgibt, treffen zu können, muß eine Betrachtung zu den einzelnen Perioden vorgenommen werden.

Theorem 7.6: Für eine Prozeßschar von *n* Prozessen mit einer festgelegten Prioritätsordnung ist ein Scheduling genau dann durchführbar, wenn die folgende Bedingung erfüllt ist:

$$\forall i : 1 \leq i \leq n \quad \min_{(k,l) \in W_i} \left(\sum_{j=1}^{i} \frac{C_j}{l\,T_k} \left\lceil \frac{l\,T_k}{T_j} \right\rceil \right) \leq 1 \qquad (7.39)$$

$$W_i = \left\{ (k,l) \mid 1 \leq k \leq i\,,\ l = 1, \ldots,\ \left\lfloor \frac{T_i}{T_k} \right\rfloor \right\}$$

Beweis: Wie bereits in Theorem 7.1 gezeigt, tritt der kritische Moment genau dann auf, wenn alle Prozesse zu einem Zeitpunkt gleichzeitig ausgeführt werden. Ausgehend von diesem Theorem konnte gezeigt werden, daß ein ratenmonotones Scheduling für die Prozeßschar gefunden werden kann, wenn der erste Prozeß einer Prozeßschar, der im kritischen Moment gestartet wurde, innerhalb seiner vorgegebenen Deadline ausgeführt werden kann. Dies wird durch Theorem 7.6 erweitert, indem eine notwendige und hinreichende Bedingung für den ersten Prozeß einer Prozeßschar unter den ungünstigsten Randbedingungen (engl. *worst case*) aufgestellt wird. Bei der Ausführung der Prozesse im kritischen Moment kann davon ausgegangen werden, daß der Prozeß P_i nur von Prozessen mit höherer Priorität unterbrochen werden kann. Somit müssen für die Bestimmung, ob ein Scheduling für Prozeß P_i erfolgreich sein kann, nur die Prozesse P_1 bis P_i betrachtet werden, da alle anderen Prozesse eine niedrigere Priorität als P_i aufweisen. Außerdem kann festgestellt werden, daß, ausgehend vom kritischen Moment, der Prozessor nicht unbeschäftigt sein wird, bis die Abarbeitung des Prozesses P_i abgeschlossen ist oder seine Deadline verpaßt wurde. Der Prozeß P_i beendet seine Ausführung zur Zeit $t \in \left[0, T_i\right]$ dann und nur dann, wenn alle Anforderungen von allen Prozessen mit einer höheren Priorität als P_i zur Zeit *t* abgearbeitet sind. Die vollständige Zeit, die zur Abarbeitung dieser Anforderungen benötigt wird, wird von $W_i(t)$ wiedergegeben:

$$W_i(t) = \sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil \qquad (7.40)$$

Die Anforderungen sind zum Zeitpunkt *t* dann und nur dann erfüllt, wenn $W_i(t) = t$ und $W_i(s) > s$ für $0 \leq t < s$ gilt. Dividiert man durch t, erhält man die Gleichung:

$$L_i(t) = \frac{W_i(t)}{t} = 1 \qquad (7.41)$$

Daraus folgt, daß eine notwendige und hinreichende Bedingung für die Einhaltung der Deadline eines Prozesses P_i dann gefunden ist, wenn die Existenz eines Zeitpunkts $t \in \left[0, T_i\right]$ nachgewiesen werden kann, für den gilt:

$$L_i = \min_{0 < t \leq T_i} L_i(t) \leq 1 \qquad (7.42)$$

Für die gesamte Prozeßschar kann genau dann und nur dann ein durchführbares Scheduling gefunden werden, wenn ein durchführbares Scheduling für jeden einzelnen Prozeß gefunden werden kann. Dies bedeutet, daß die folgende Gleichung für jedes *i* erfüllt sein muß:

$$L = \max_{1<i\le n} L_i \le 1 \qquad (7.43)$$

Gemäß Gleichung (7.42) muß das Minimum über der kontinuierlichen Variablen $t \in [0, T_i]$ gefunden werden. Da die Funktion $L_i(t)$ stückweise monoton fällt, muß nur eine begrenzte Anzahl von kritischen Stellen überprüft werden. Die Stellen, an denen die Sprünge auftreten, werden genau dann erreicht, wenn *t* einer der Perioden T_j entspricht, wobei $1 \le j \le i$ gilt. Diese Stellen werden als Schedulingzeitpunkte des Prozesses P_i bezeichnet. Die Funktion ist an diesen Stellen linksseitig stetig und springt rechtsseitig auf einen höheren Wert (Bild 7-11).

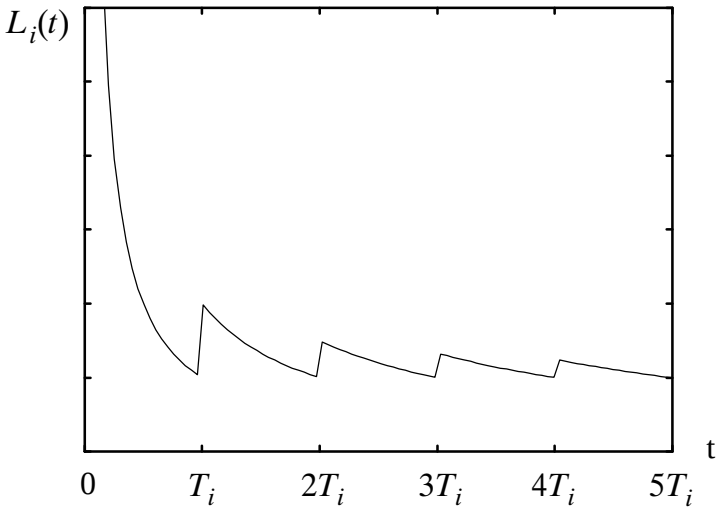


Bild 7-11: Qualitativer Verlauf von $L_i(t)$

Somit muß nicht über den ganzen Bereich von *t* ein Minimum gesucht werden, sondern nur an bestimmten Stellen, den lokalen Minima, die mit folgender Gleichung charakterisiert werden können:

$$S_i = \left\{ k \cdot T_j \mid j = 1, \ldots, i;\ k = 1, \ldots, \left\lfloor \frac{T_i}{T_j} \right\rfloor \right\} \qquad (7.44)$$

Die Elemente von S_i entsprechen den Schedulingzeitpunkten des Prozesses P_i, der Deadline von Prozeß P_i sowie den Schedulingzeitpunkten und Deadlines von allen Prozessen, die eine höhere Priorität als Prozeß P_i aufweisen und vor der Deadline von P_i abgearbeitet sind. Engt man die zulässigen Werte auf die in Gleichung (7.44) aufgeführten Punkte ein, so erhält man Gleichung (7.39).
□

Um die Anwendung von Theorem 7.6 nachvollziehen zu können, wird im folgenden die Durchführbarkeit des Schedulings einer Prozeßschar anhand Tabelle 7-2 untersucht. Das Beispiel umfaßt drei Prozesse, die die kleinste obere Grenze der Prozessorausnutzung verletzen [Zale95].

Prozeß i	Ausführungszeit E_i	Periode T_i	Verhältnis E_i/T_i'	Prozessorausnutzung (1 .. N)	Obere Grenze nach Theorem 7.5
1	45	135	0,333	0,333	1,000
2	50	150	0,333	0,667	0,828
3	80	360	0,222	0,889	0,779

Tabelle 7-2: Anwendungsbeispiel

Der Nachweis nach Gleichung (7.39) muß nur für den Prozeß P_3 geführt werden, da für die anderen beiden Prozesse bereits ein Scheduling nach Theorem 7.5 durchführbar ist. Somit ist bereits eine Variable $i = 3$ der Gleichung (7.39) festgelegt. Die weiteren Variablen lassen sich zu $j = 1, \ldots, 3$; $k = 1, \ldots, 3$ und $l = 1, \ldots, \left\lfloor \frac{T_i}{T_k} \right\rfloor$ festlegen. l ergibt sich bei variiertem k zu:

$$k = 1: \quad \left\lfloor \frac{T_i}{T_k} \right\rfloor = \left\lfloor \frac{360}{135} \right\rfloor = 2 \quad \Rightarrow \quad l = 1, 2$$

$$k = 2: \quad \left\lfloor \frac{T_i}{T_k} \right\rfloor = \left\lfloor \frac{360}{150} \right\rfloor = 2 \quad \Rightarrow \quad l = 1, 2$$

$$k = 3: \quad \left\lfloor \frac{T_i}{T_k} \right\rfloor = \left\lfloor \frac{360}{360} \right\rfloor = 1 \quad \Rightarrow \quad l = 1$$

Ausgehend von diesen Variablen lassen sich mittels Gleichung (7.39) für jede Periode und ihre Vielfache überprüfen, ob ein Scheduling möglich ist.

$k = 1, l = 1$:

$$\frac{C_1}{1 \cdot T_1} \left\lceil \frac{1 \cdot T_1}{T_1} \right\rceil + \frac{C_2}{1 \cdot T_1} \left\lceil \frac{1 \cdot T_1}{T_2} \right\rceil + \frac{C_3}{1 \cdot T_1} \left\lceil \frac{1 \cdot T_1}{T_3} \right\rceil = \frac{45 + 50 + 80}{135} = 1,296 > 1$$

$k = 1, l = 2$:

$$\frac{C_1}{2 \cdot T_1} \left\lceil \frac{2 \cdot T_1}{T_1} \right\rceil + \frac{C_2}{2 \cdot T_1} \left\lceil \frac{2 \cdot T_1}{T_2} \right\rceil + \frac{C_3}{2 \cdot T_1} \left\lceil \frac{2 \cdot T_1}{T_3} \right\rceil = \frac{2 \cdot 45 + 2 \cdot 50 + 80}{2 \cdot 135} \stackrel{!}{=} 1$$

$k = 2, l = 1$:

$$\frac{C_1}{1 \cdot T_2} \left\lceil \frac{1 \cdot T_2}{T_1} \right\rceil + \frac{C_2}{1 \cdot T_2} \left\lceil \frac{1 \cdot T_2}{T_2} \right\rceil + \frac{C_3}{1 \cdot T_2} \left\lceil \frac{1 \cdot T_2}{T_3} \right\rceil = \frac{2 \cdot 45 + 50 + 80}{150} = 1,467 > 1$$

$k = 2, l = 2$:

$$\frac{C_1}{2 \cdot T_2} \left\lceil \frac{2 \cdot T_2}{T_1} \right\rceil + \frac{C_2}{2 \cdot T_2} \left\lceil \frac{2 \cdot T_2}{T_2} \right\rceil + \frac{C_3}{2 \cdot T_2} \left\lceil \frac{2 \cdot T_2}{T_3} \right\rceil = \frac{3 \cdot 45 + 2 \cdot 50 + 80}{2 \cdot 150} = 1,05 > 1$$

$k = 3, l = 1$:

$$\frac{C_1}{1 \cdot T_3} \left\lceil \frac{1 \cdot T_3}{T_1} \right\rceil + \frac{C_2}{1 \cdot T_3} \left\lceil \frac{1 \cdot T_3}{T_2} \right\rceil + \frac{C_3}{1 \cdot T_3} \left\lceil \frac{1 \cdot T_3}{T_3} \right\rceil = \frac{3 \cdot 45 + 3 \cdot 50 + 80}{360} = 1,014 > 1$$

Da eine der obigen Gleichungen (für $k = 1$ und $l = 2$) erfüllt ist, ist das ratenmonotone Scheduling für die drei Prozesse aus Tabelle 7-2 durchführbar. Da diese Form des Nachweises oftmals zeitintensiv und unübersichtlich ausfällt, kann die Durchführbarkeit des Schedulings nach Theorem 7.5 und Theorem 7.6 auch in Form eines einfacher anwendbaren Algorithmus nach [LeSD89] berechnet werden.

- **Schritt 1:** Man wende die aus Theorem 7.5 bekannte Berechnung der kleinsten oberen Grenze an und beende den Algorithmus, wenn die Bedingungen erfüllt sind. Falls nicht, wende man für die Fälle, in denen Theorem 7.5 nicht erfüllt ist, Theorem 7.6 an (Schritt 2a-2c).
- **Schritt 2a:** Es müssen alle Zeitpunkte bestimmt werden, zu denen ein Scheduling möglich ist, indem man für alle nach Schritt 1 übrigbleibenden Prozesse auf einer Zeitachse alle Perioden und deren Vielfache einträgt. Dabei werden die Eintragungen von $t = 0$ begonnen und bis zum Ende des Prozesses mit der höchsten Periode vorgenommen.
- **Schritt 2b:** Für jeden in Schritt 2a ermittelten Zeitpunkt wird eine Ungleichung aufgestellt, die auf der linken Seite die Summe aller möglichen Ausführungszeiten der Prozesse, bzw. deren Vielfache, aufsummiert, die bis zu dem betrachteten Zeitpunkt möglich sind und auf der rechten Seite den Wert des Zeitpunktes selbst.
- **Schritt 2c:** Sind die Werte auf der linken Seite kleiner oder gleich den entsprechenden Werten auf der rechten Seite, so ist ein ratenmonotones Scheduling der Prozeßschar möglich. Wird keine der Ungleichungen erfüllt, so ist kein ratenmonotones Scheduling möglich.

Anhand des Beispiels in Tabelle 7-2 soll auch dieses Vorgehen demonstriert werden. Beginnend mit Schritt 1 wird Theorem 7.5 angewendet und ermittelt, daß der dritte Prozeß eine Verletzung der kleinsten oberen Grenze verursacht. Der zu untersuchende Fall beschränkt sich also auf $i = 3$. In Schritt 2a werden die Zeitpunkte bestimmt, zu denen ein Scheduling möglich ist. Für die einzelnen Prozesse ergeben sich die folgenden Möglichkeiten innerhalb der Zeitpunkte 0 bis 360 (letzteres entspricht demjenigen Prozeß mit der langsamsten Periode): der erste Prozeß mit der Periode 135 kann bei 0, 135 und 270 ausgeführt werden; der zweite Prozeß bei 0, 150 und 300; der dritte Prozeß bei 0 und 360. Die zu untersuchenden Zeitpunkte sind auf einer Zeitachse geordnet in Bild 7-12 dargestellt.

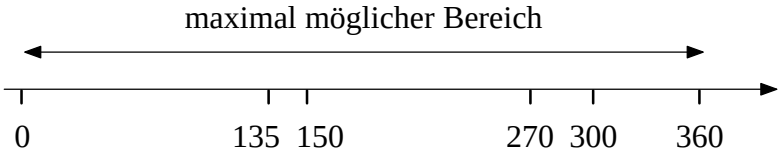


Bild 7-12: Schedulingzeitpunkte für Tabelle 7-2

Bei der Aufstellung der Ungleichungen nach Schritt 2b ergeben sich die folgenden Beziehungen:

$$C_1 + C_2 + C_3 \leq T_1$$
$$2 \cdot C_1 + C_2 + C_3 \leq T_2$$
$$2 \cdot C_1 + 2 \cdot C_2 + C_3 \leq 2 \cdot T_1$$
$$3 \cdot C_1 + 2 \cdot C_2 + C_3 \leq 2 \cdot T_2$$
$$3 \cdot C_1 + 3 \cdot C_2 + C_3 \leq T_3$$

Setzt man in Schritt 2c konkrete Werte für die obigen Variablen ein, so ergibt sich:

$$45 + 50 + 80 > 135$$
$$2 \cdot 45 + 50 + 80 > 150$$
$$2 \cdot 45 + 2 \cdot 50 + 80 = 2 \cdot 135$$
$$3 \cdot 45 + 2 \cdot 50 + 80 > 2 \cdot 150$$
$$3 \cdot 45 + 3 \cdot 50 + 80 > 360$$

Aus den obigen Ungleichungen läßt sich erkennen, daß ein ratenmonotones Scheduling für die Prozeßschar durchführbar ist, da eine Ungleichung erfüllt ist. Vergleicht man das Ergebnis mit der Berechnung nach Theorem 7.6, so kann man die Übereinstimmung erkennen.

Aus Theorem 7.6 lassen sich noch weitere Erkenntnisse gewinnen. Mit Hilfe dieses Theorems ist es möglich, die Berechnung der maximalen Ausführungszeit oder Periode eines Prozesses zu bestimmen. Anhand des Beispiels in Tabelle 7-3 wird die Berechnung der maximalen Ausführungszeit demonstriert.

Prozeß i	Ausführungszeit C_i	Periode T_i	Verhältnis C_i/T_i'	Prozessorausnutzung (1 .. N)	Obere Grenze nach Theorem 7.5
1	50	150	0,333	0,333	1,000
2	x	200	x/200	0,333 + x/200	0,828
3	192	400	0,48	0,813 + x/200	0,779

Tabelle 7-3: Berechnung der maximalen Ausführungszeit für Prozeß P_2

Nach Schritt 1 ergibt sich, daß die Prozessorausnutzung der beiden Prozesse P_1 und P_3 (ohne Betrachtung des unbekannten Wertes von P_2) oberhalb der oberen Grenze von Theorem 7.5 liegt und damit keine Aussage getroffen werden kann. Wendet man Schritt 2a an, so ergeben sich als Zeitpunkte, zu denen ein Scheduling möglich ist, die folgende Werte: 150, 200, 300 und 400. Bei der Aufstellung der Ungleichungen nach Schritt 2b ergeben sich die folgenden Beziehungen:

$$C_1 + C_2 + C_3 \leq T_1$$
$$2 \cdot C_1 + C_2 + C_3 \leq T_2$$
$$2 \cdot C_1 + 2 \cdot C_2 + C_3 \leq 2 \cdot T_1$$
$$3 \cdot C_1 + 2 \cdot C_2 + C_3 \leq T_3$$

Setzt man die entsprechenden Werte in Schritt 2c ein, so erhält man die folgenden Beziehungen:

$$50 + x + 192 \leq 150$$
$$2 \cdot 50 + x + 192 \leq 200$$
$$2 \cdot 50 + 2 \cdot x + 192 \leq 2 \cdot 150$$
$$3 \cdot 50 + 2 \cdot x + 192 \leq 400$$

Aus diesen Beziehungen muß nun der größte mögliche Wert für x bestimmt werden. Aus der letzten Gleichung erhält man $x = 29$ als die größte mögliche Ausführungszeit für Prozeß P_2.

Wichtiger als die Bestimmung der maximalen Ausführungszeit, die meist durch die Modellierung festgelegt ist, ist jedoch die Wahl der Periode. Die kleinste mögliche Periode bestimmt die maximal erreichbare Genauigkeit, mit der die Verarbeitung durchgeführt werden kann. Diese Bestimmung wird im nächsten Kapitel durchgeführt.

Pseudoraten-basiertes Scheduling

Als ein entscheidender Faktor für den Einsatz von Rapid Prototyping ist die Geschwindigkeit anzusehen, mit der eine Modellierung ausgeführt werden kann. Insbesondere im Bereich von Motorsteuerungen ist man mit Anforderungen konfrontiert, die im Bereich unterhalb von einer Millisekunde liegen. Das größte Optimierungspotential liegt dabei in den Bereichen des Schedulings, der optimalen Ausnutzung der Systemressourcen und dem Modellcode selbst. Nachdem im letzten Teilkapitel bereits das ratenmonotone Scheduling erläutert wurde, wird sich dieses Kapitel der Im-

plementierung des Schedulings unter den Randbedingungen des Rapid Prototyping widmen und eine Optimierung des ratenmonotonen Schedulings vornehmen.

Ein großer Teil der Rechenzeit des Schedulings unter Echtzeitbedingungen geht für die Ausführung und Überwachung von Timern verloren. Bei den heutigen Hardwareplattformen stehen oftmals nur software-implementierte Watchdog-Timer zur Verfügung und ein zusätzlicher hardware-implementierter Hilfstimer (engl. *auxiliary timer*). Dieser Auxiliary-Timer wird ohne weitere Systembelastung in Hardware ausgeführt und bekommt zu Beginn der Modellabarbeitung eine zeitliche Auflösung vorgegeben, nach deren Ablauf jeweils ein Interrupt an das System erzeugt wird. Dieser Interrupt kann benutzt werden, um eine Zeitbasis vorzugeben und die Einhaltung der Echtzeitbedingungen zu kontrollieren.

Heutige Hardwareplattformen stellen jedoch nur wenige, in den meisten Fällen nur einen Auxiliary-Timer zur Verfügung, der bei ratenmonotonem Scheduling auf die Periode eines einzigen Prozesses adaptiert werden kann. Die Anzahl der erzeugbaren Watchdog-Timern ist hingegen nicht limitiert. Jedoch bringt eine hohe Anzahl von Watchdog-Timern einen hohen Ressourcenverbrauch mit sich.

Eine weitere Möglichkeit besteht darin, den größten gemeinsamen Teiler aller Prozesse von einem Auxiliary-Timer erzeugen zu lassen und diesen als gemeinsame Zeitbasis allen Prozessen zur Verfügung zu stellen. Die benötigte Auflösung δ für eine Prozeßschar mit unterschiedlichen Perioden ist in Gleichung (7.45) wiedergegeben.

$$\forall\, i : \frac{T_i}{\delta} \in \mathbb{N}, \quad \forall\, j,\, k = 1 \ldots n,\, j \neq k : \quad \delta = gcd \left| T_j - T_k \right| \qquad (7.45)$$

In Gleichung (7.45) werden die zeitlichen Differenzen zwischen den Perioden aller Prozesse bestimmt. Die benötigte Auflösung δ ist dann mit dem größten gemeinsamen Teiler aller Differenzen gleichzusetzen. Schedulingstrategien, die ein einziges festes Zeitintervall als gemeinsame Zeitbasis für alle Prozesse benutzen, werden als pseudoraten-basiertes Scheduling (engl. *pseudo-rate scheduling*) bezeichnet.

Es kann geschehen, daß die errechneten Auflösungen zu klein sind, um erzeugt werden zu können. Wird beispielsweise die notwendige Auflösung δ für zwei Prozesse mit den Abtastfrequenzen 1 MHz und 800 kHz berechnet, ergibt sich als Auflösung δ = 0,25 µs (Bild 7-13).

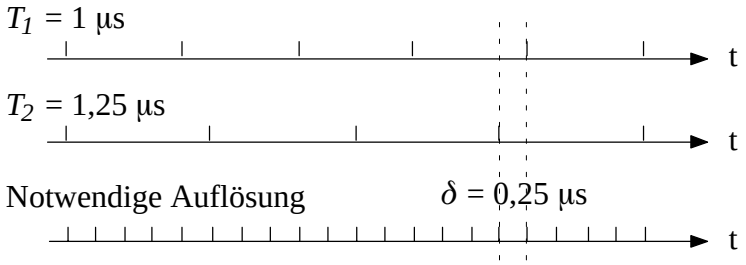


Bild 7-13: Notwendige Auflösung δ

Diese zeitliche Auflösung kann in manchen Fällen nicht erreicht werden. Wird die Auflösung des Auxiliary-Timers sehr klein, muß zusätzlich die Zeit für die Abarbeitung des ausgelösten Interrupts beachtet werden. Dies verringert die geringe Zeitspanne zusätzlich, die für die Abarbeitung des Modells zur Verfügung steht. Es muß also eine zeitliche Auflösung gefunden werden, die die Echtzeitbedingungen einhält, die andererseits aber nicht so klein gewählt ist, daß der Prozessor nur mit der Abarbeitung der Interrupts beschäftigt ist. Wird die Prozessorleistung nur zur Abarbeitung von Interrupts benötigt und die eigentliche Verarbeitung wird nicht mehr durchgeführt, so wird dies als

Trashing bezeichnet. Um diesen Fall zu vermeiden, muß man die Abtastraten der einzelnen Prozesse bereits im Vorfeld so adaptieren, daß eine Verarbeitung mit der zur Verfügung stehenden Hardware machbar ist. Liegt die Grenze der Abarbeitung der Hardware im Fall von Bild 7-13 im Bereich von einer Mikrosekunde, so müßte geprüft werden, ob eine Adaptierung der Periode des Prozesses P_2 auf 1 µs vorgenommen werden kann. Generell muß eine Adaptierung so vorgenommen werden, daß die Abtastrate kleiner oder gleich der ursprünglichen Abtastrate ist. Da auch dieses Vorgehen unter Umständen zu Problemen führen kann, wird die Problematik in Kapitel 7.4 gesondert betrachtet.

Die Durchführbarkeit eines pseudoraten-basierten Schedulings kann nicht mit den Theoremen berechnet werden, die im vorigen Teilkapitel für das ratenmonotone Scheduling aufgestellt wurden (Theorem 7.5 und Theorem 7.6), da die Deadline der Prozesse, die adaptiert werden, streng genommen gleichbleibt, obwohl ihre Periode verkürzt wird. Damit liegt ein Fall vor, bei dem die Deadline größer oder gleich der Periode ist. Die Annahme von Gleichung (7.4) ist verletzt. Für diesen Fall gibt es in der Literatur keine Lösung [Zale95]. Für den speziellen Fall im Bereich Rapid Prototyping kann jedoch die Deadline der Periode gleichgesetzt werden und die Theoreme des letzten Teilkapitels können verwendet werden. Statt der Periode T_i erhalten wir nun die verkürzte Periode T_i'.

$$T_i' = \left\lfloor \frac{T_i}{T_{aux}} \right\rfloor T_{aux} \qquad (7.46)$$

T_{aux} stellt dabei die Periode des Auxiliary-Timers dar. Mit dieser Periode können die Gleichungen von Theorem 7.5 und Theorem 7.6 angepaßt werden. Theorem 7.5 ergibt sich zu:

$$\sum_{i=1}^{n} \frac{C_i}{\left\lfloor \frac{T_i}{T_{aux}} \right\rfloor T_{aux}} \leq n\left(2^{\frac{1}{n}} - 1\right), \quad T_{aux} \leq T_i \qquad (7.47)$$

Auf diese Weise werden nur Werte für Perioden berücksichtigt, die Vielfache von T_{aux} sind. Das Theorem 7.6 kann folgendermaßen angepaßt werden:

$$\min_{(k,l)\in W_i} \left\{ \sum_{j=1}^{i} \frac{C_j}{l \left\lfloor \frac{T_k}{T_{aux}} \right\rfloor T_{aux}} \left\lceil \frac{l \left\lfloor \frac{T_k}{T_{aux}} \right\rfloor T_{aux}}{\left\lfloor \frac{T_j}{T_{aux}} \right\rfloor T_{aux}} \right\rceil \right\} \leq 1 \qquad (7.48)$$

$$W_i = \left\{ (k,l) \mid 1 \leq k \leq i,\ l = 1, \ldots, \left\lfloor \frac{\left\lfloor \frac{T_i}{T_{aux}} \right\rfloor T_{aux}}{\left\lfloor \frac{T_k}{T_{aux}} \right\rfloor T_{aux}} \right\rfloor \right\}$$

Wie bei Theorem 7.6 für ratenmonotones Scheduling werden Ungleichungen für alle Zeitpunkte bestimmt, zu denen ein Scheduling möglich ist. Durch die Festlegung der Perioden als Vielfache von T_{aux} sind die Zeitpunkte, zu denen ein Scheduling möglich ist, ebenfalls Vielfache von T_{aux}. Als Beispiel für Gleichung (7.48) wird das in Tabelle 7-2 aufgeführte Beispiel für ratenmonotones Scheduling abgewandelt. Wie aus dem vorigen Teilkapitel ersichtlich, war das ratenmonotone Scheduling durchführbar. Für dieses Beispiel sei angenommen, daß der Auxiliary-Timer alle 20 Zeiteinheiten einen Interrupt erzeugt. Tabelle 7-4 zeigt die aus Tabelle 7-2 bekannten Prozesse. Da die Perioden der Prozesse P_1 und P_2 keine Vielfachen von 20 Zeiteinheiten aufweisen, werden beide auf die nächste kleinere Grenze angepaßt. Somit ergeben sich statt einer ratenmonotonen Periode

von 135 Zeiteinheiten bei Prozeß P_1 eine pseudoraten-basierte Periode von 120 Zeiteinheiten und bei P_2 statt 150 Zeiteinheiten 140 Zeiteinheiten. Die pseudoraten-basierten Perioden sind in Tabelle 7-4 als T_i' wiedergegeben.

Prozeß i	**Ausführungs-zeit C_i**	**Ratenmono-tone Periode T_i**	**Pseudoraten Periode T_i'**	**Verhältnis C_i/T_i'**	**Auslastungs-faktor 1 .. N**	**Obere Grenze**
1	45	135	120	0,375	0,375	1,000
2	50	150	140	0,357	0,732	0,828
3	80	360	360	0,222	0,954	0,779

Tabelle 7-4: Beispiel für pseudoraten-basiertes Scheduling

Die Untersuchung auf Durchführbarkeit des Schedulings kann sich auf den Fall $i = 3$ von Gleichung (7.48) beschränken, da sich die Prozessorausnutzung der Prozesse P_1 und P_2 unterhalb der oberen Grenze nach Gleichung (7.47) befindet. Die weiteren Variablen lassen sich zu $j = 1, \ldots, 3$; $k = 1, \ldots, 3$ und $l = 1, \ldots, \left\lfloor \frac{\left\lfloor \frac{T_i}{T_{aux}} \right\rfloor T_{aux}}{\left\lfloor \frac{T_k}{T_{aux}} \right\rfloor T_{aux}} \right\rfloor$ festlegen. l ergibt sich bei variiertem k zu:

$k = 1$: $\left\lfloor \frac{360}{120} \right\rfloor = 3 \quad \Rightarrow \quad l = 1, \ldots, 3$

$k = 2$: $\left\lfloor \frac{360}{140} \right\rfloor = 2 \quad \Rightarrow \quad l = 1, 2$

$k = 3$: $\left\lfloor \frac{360}{360} \right\rfloor = 1 \quad \Rightarrow \quad l = 1$

Ausgehend von diesen Variablen lassen sich mittels Gleichung (7.48) für jede Periode und ihre Vielfache überprüfen, ob ein Scheduling möglich ist.

$k = 1, l = 1$:

$$\frac{C_1}{1 \cdot T_1'} \left\lceil \frac{1 \cdot T_1'}{T_1'} \right\rceil + \frac{C_2}{1 \cdot T_1'} \left\lceil \frac{1 \cdot T_1'}{T_2'} \right\rceil + \frac{C_3}{1 \cdot T_1'} \left\lceil \frac{1 \cdot T_1'}{T_3'} \right\rceil = \frac{45 + 50 + 80}{120} = 1,458 > 1$$

$k = 1, l = 2$:

$$\frac{C_1}{2 \cdot T_1'} \left\lceil \frac{2 \cdot T_1'}{T_1'} \right\rceil + \frac{C_2}{2 \cdot T_1'} \left\lceil \frac{2 \cdot T_1'}{T_2'} \right\rceil + \frac{C_3}{2 \cdot T_1'} \left\lceil \frac{2 \cdot T_1'}{T_3'} \right\rceil = \frac{2 \cdot 45 + 2 \cdot 50 + 80}{2 \cdot 120} = 1,125 > 1$$

$k = 1, l = 3$:

$$\frac{C_1}{3 \cdot T_1'} \left\lceil \frac{3 \cdot T_1'}{T_1'} \right\rceil + \frac{C_2}{3 \cdot T_1'} \left\lceil \frac{3 \cdot T_1'}{T_2'} \right\rceil + \frac{C_3}{3 \cdot T_1'} \left\lceil \frac{3 \cdot T_1'}{T_3'} \right\rceil = \frac{3 \cdot 45 + 3 \cdot 50 + 80}{3 \cdot 120} = 1,014 > 1$$

$k = 2, l = 1$:

$$\frac{C_1}{1 \cdot T_2'} \left\lceil \frac{1 \cdot T_2'}{T_1'} \right\rceil + \frac{C_2}{1 \cdot T_2'} \left\lceil \frac{1 \cdot T_2'}{T_2'} \right\rceil + \frac{C_3}{1 \cdot T_2'} \left\lceil \frac{1 \cdot T_2'}{T_3'} \right\rceil = \frac{2 \cdot 45 + 50 + 80}{150} = 1,467 > 1$$

$k = 2, l = 2$:

$$\frac{C_1}{2 \cdot T_2'} \left\lceil \frac{2 \cdot T_2'}{T_1'} \right\rceil + \frac{C_2}{2 \cdot T_2'} \left\lceil \frac{2 \cdot T_2'}{T_2'} \right\rceil + \frac{C_3}{2 \cdot T_2'} \left\lceil \frac{2 \cdot T_2'}{T_3'} \right\rceil = \frac{3 \cdot 45 + 2 \cdot 50 + 80}{2 \cdot 140} = 1,125 > 1$$

$k = 3, l = 1$:

$$\frac{C_1}{1 \cdot T_3'} \left\lceil \frac{1 \cdot T_3'}{T_1'} \right\rceil + \frac{C_2}{1 \cdot T_3'} \left\lceil \frac{1 \cdot T_3'}{T_2'} \right\rceil + \frac{C_3}{1 \cdot T_3'} \left\lceil \frac{1 \cdot T_3'}{T_3'} \right\rceil = \frac{3 \cdot 45 + 3 \cdot 50 + 80}{360} = 1{,}014 > 1$$

Da keine der Ungleichungen erfüllt ist, ist das pseudoraten-basierte Scheduling nicht durchführbar. Um ein Scheduling zu ermöglichen, müssen entweder die Perioden der Prozesse erhöht, die Ausführungszeiten verringert oder eine kleinere Auflösung gewählt werden. Nimmt man beispielsweise für das obige Beispiel eine geringere Ausführungszeit $C_3' = 75$ an, so ist das pseudoraten Scheduling durchführbar, wie der Fall für $k = 3$ und $l = 1$ zeigt:

$$\frac{C_1}{1 \cdot T_3'} \left\lceil \frac{1 \cdot T_3'}{T_1'} \right\rceil + \frac{C_2}{1 \cdot T_3'} \left\lceil \frac{1 \cdot T_3'}{T_2'} \right\rceil + \frac{C_3'}{1 \cdot T_3'} \left\lceil \frac{1 \cdot T_3'}{T_3'} \right\rceil = \frac{3 \cdot 45 + 3 \cdot 50 + 75}{360} \stackrel{!}{=} 1$$

Ein mögliches Scheduling für das Beispiel von Tabelle 7-4 zeigt Bild 7-14.

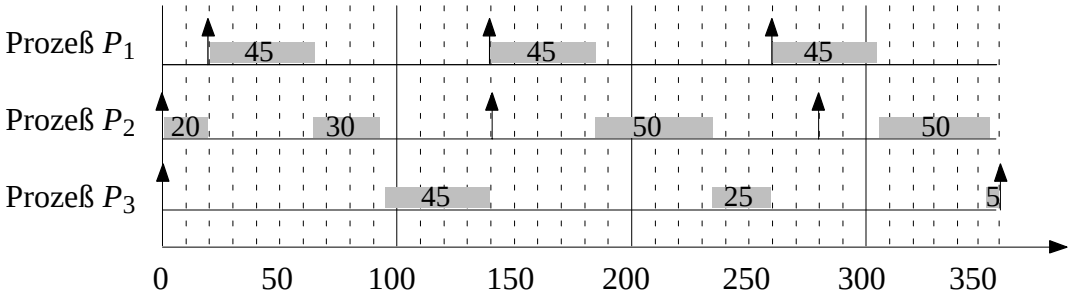


Bild 7-14: Pseudoraten-basiertes Scheduling für Tabelle 7-4

An dieser Stelle erkennt man, daß die Schedulingzeiten für pseudoraten-basiertes Scheduling schlechter ausfallen, als beim ratenmonotonen Scheduling. Dies ist auf die verkürzten Perioden zurückzuführen, die eingeführt werden mußten, um eine Abarbeitung mit nur einem Auxiliary-Timer zu ermöglichen. Um die bereits erwähnten Vorteile von pseudoraten-basiertem Scheduling für den Rapid Prototyping Prozeß auch rechnerisch nachweisen zu können, muß die Genauigkeit der Zeitabschätzungen erhöht werden [KaAS93]. Es ist insbesondere notwendig, auch diejenigen Zeiten in das Zeitmodell aufzunehmen, die beispielsweise von Interrupts, Watchdog-Timern und Auxiliary-Timern benötigt werden. Diese Zeiten sind im folgenden aufgeführt:

- C_{model} repräsentiert die Zeit, die für die Abarbeitung eines Modellschritts benötigt wird. Wie in Kapitel 4.2.2 bereits erläutert, versteht man unter einem Modellschritt das logische Verhalten des modellierten Systems, das im Berechnungszeitraum bei diskreten Systemen den Folgezustand festlegt oder bei kontinuierlichen Systemen die Berechnung eine definierte Zeitspanne voranschreiten läßt. Da in diesem Zusammenhang die Prozesse unabhängig voneinander sind, wird C_{model} durch C_1, ..., C_i repräsentiert.
- Die Variable C_{sched} gibt die Zeit wieder, die der Scheduler benötigt, um den nächsten aktiven Prozeß zu bestimmen.
- C_{int} repräsentiert die Zeit, die benötigt wird, um einen Interrupt zu verarbeiten. Dabei geht in diese Variable auch die Speicherung eines minimalen Registerkontext ein, der notwendig ist, um nach der Verarbeitung des Interrupts den Systemzustand wiederherstellen und den Scheduler aufrufen zu können.
- C_{store} repräsentiert die Zeit, die benötigt wird, um die Umgebung des aktuellen, unterbrochenen Prozesses zu speichern.

- C_{load} repräsentiert die Zeit, die benötigt wird, um die Umgebung des vor dem aktuellen Prozeß unterbrochenen Prozesses zu laden.
- C_{resume} stellt die Zeit dar, die bei der Wiederaufnahme eines Prozesses benötigt wird, wenn der aktuelle Prozeß zuvor durch einen Interrupt unterbrochen wurde. In diese Variable geht auch die Zeit ein, die benötigt wird, um den Registerkontext wiederherzustellen und zur Ausführung des Prozesses überzugehen.
- Die Variable C_{trap} repräsentiert die Zeit, die bei der normalen Beendigung eines Prozesses benötigt wird.
- Die Variable C_{timer} gibt die Zeit wieder, die benötigt wird, um einen Timer zu starten und neben der regulären Abarbeitung zu betreiben. C_{timer} kann auf zwei unterschiedliche Arten implementiert werden. Der Aufruf eines Auxiliary-Timers (siehe Beginn dieses Kapitels) erfordert nur die Abarbeitung des Interrupts C_{int} und der Zeit C_{resume}, die für die Wiederaufnahme des unterbrochenen Prozesses benötigt wird. Die benötigte Zeit wird in der Variablen C_{aux} zusammengefaßt. Die Variable C_{wd} gibt die Zeit wieder, die benötigt wird, um einen Watchdog–Timer zu benutzen und besteht aus der Abarbeitung des Interrupts C_{int}, der Zeit C_{resume}, die für die Wiederaufnahme des unterbrochenen Prozesses benötigt wird und einer zusätzlichen, kontinuierlichen Belastung C_{det}, die für die Bestimmung der Systemzeit erforderlich ist. Die Zeitspanne ist als wesentlich größer im Vergleich zu der Variablen C_{aux} einzustufen, da ein Watchdog-Timer einerseits in Software realisiert ist und somit eine größere Zeitspanne zur Abarbeitung benötigt, andererseits muß zur Überwachung des Watchdog-Timers ein zusätzlicher Prozeß verwendet werden.

Für die beiden Variablen $C_{timer,\,aux}$ und $C_{timer,\,wd}$ ergibt sich:

$$\begin{aligned} C_{aux} &= C_{int} + C_{resume} \\ C_{wd} &= C_{int} + C_{resume} + C_{det} \end{aligned} \qquad (7.49)$$

Wie in [KaAS93] vorgeschlagen, werden die vorgestellten Variablen als größere Gruppen zu $C_{preempt}$ und C_{exit} zusammengefaßt, da diese im weiteren Verlauf häufig benötigt werden.

$$\begin{aligned} C_{preempt} &= C_{store} + C_{load} \\ C_{exit} &= C_{trap} + C_{load} \end{aligned} \qquad (7.50)$$

In Bild 7-15 ist ein zeitgesteuertes Scheduling unter Einbeziehung der zusätzlich eingeführten Zeiten dargestellt. Zu Beginn befindet sich der Prozessor im ungenutzen Zustand (engl. *idle*). Jeder Timerinterrupt ruft eine Interrupt-Bearbeitung auf, die feststellt, ob die Prozesse Echtzeitbedingungen verletzt haben. Nach Beendigung der Interrupt-Bearbeitung wird der höchstpriore Prozeß aufgerufen, der gerade abgearbeitet werden muß (hier Prozeß P_1). Beim Aufruf des Prozesses wird die Zeit $C_{preempt}$ fällig. Wird der Prozeß komplett abgearbeitet, so wird beim Abschließen des Prozesses C_{exit} fällig. Ein Sonderfall tritt auf, wenn der Prozeß zwischen zwei Timerinterrupts nicht vollständig abgearbeitet wird und somit unterbrochen wird. In diesem Fall verursacht die Unterbrechung und Wiederaufnahme des Prozesses keine zusätzlichen Zeit, da das Laden und Speichern der Prozeßumgebung zum unterbrechenden Prozeß hinzugerechnet wird.

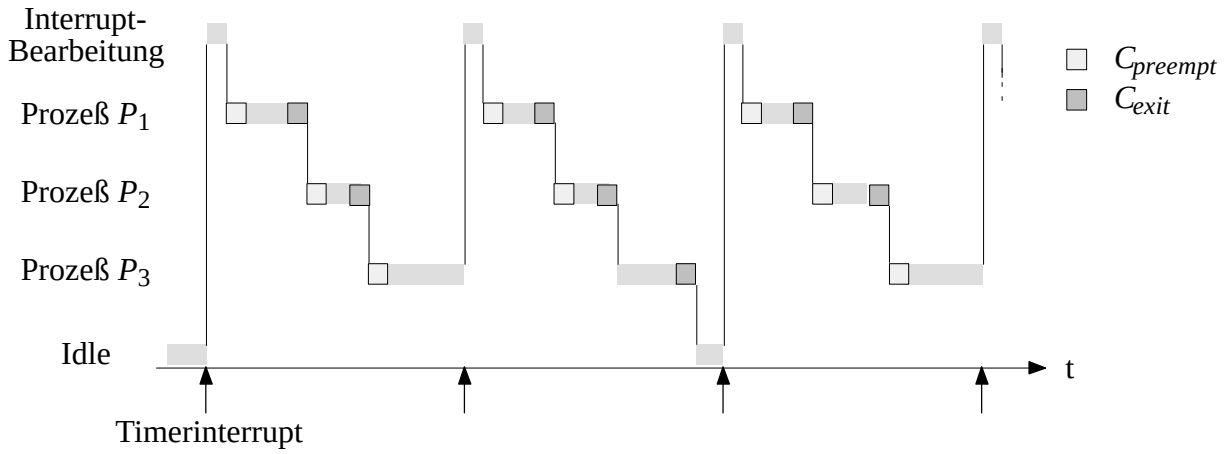


Bild 7-15: Zeitgesteuertes Scheduling

Theorem 7.7: Der größtmögliche Zusatzaufwand, der bei einer Unterbrechung für einen periodischen, mit festgelegter Priorität versehenen Prozeß aufgewendet werden muß, ist die Summe von $C_{preempt} + C_{exit}$.

Beweis: Ein Prozeß P_i werde zur Zeit $t = 0$ gestartet. Befindet sich ein höherpriorer Prozeß P_j bereits in der Ausführung, so wird die Abarbeitung des Interrupts des Prozesses P_i verzögert, bis alle höherprioren Prozesse abgearbeitet sind. Zu diesem Zeitpunkt bekommt der Prozeß P_i den Prozessor zugeteilt. Die Zuteilung des Prozesses zieht das Sichern des bis dahin aktiven Prozesses und das Laden der Prozeßumgebung nach sich (nach Gleichung (7.50) mit $C_{preempt}$ bezeichnet). Ist der Prozeß P_i komplett ausgeführt, so wird der Prozeß beendet (C_{trap}) und die Umgebung des davor aktiven Prozesses muß geladen werden (C_{load}). Insgesamt ergibt sich also die Zeitspanne $C_{preempt} + C_{exit}$. Ein niederpriorer Prozeß kann den Prozeß P_i nicht unterbrechen und wird deswegen keinen Zusatzaufwand verursachen. Falls ein höherpriorer Prozeß während der Abarbeitung von P_i eintrifft, so wird P_i während seiner regulären Abarbeitung unterbrochen. Der hierbei entstehende Zusatzaufwand wird komplett zum unterbrechenden Prozeß P_j hinzugerechnet. Somit ist der größtmögliche Zusatzaufwand, der zum Prozeß P_i hinzugerechnet wird, auf die Zeitspanne $C_{preempt} + C_{exit}$ begrenzt. □

Theorem 7.8: Der größtmögliche Zusatzaufwand für einen Timer im Intervall t mit einem zeitgesteuerten, periodischem Scheduling beträgt höchstens $\left\lceil \frac{lT_k}{T_{timer}} \right\rceil C_{timer}$.

Der Beweis zu Theorem 7.8 kann folgendermaßen geführt werden. In jedem Intervall $l\,T_k$ wird ein Timer höchstens $\left\lceil \frac{lT_k}{T_{timer}} \right\rceil$ Mal aufgerufen. Jedes Mal, wenn der Timer die Verarbeitung unterbricht, wird ein Zusatzaufwand von $C_{timer} = C_{int} + C_{resume} \left(+ C_{det} \right)$ gemäß Gleichung (7.50) benötigt. □

Theorem 7.9: Der größtmögliche Zusatzaufwand für das Blockieren eines Prozesses mit festgelegter Priorität mit einem zeitgesteuerten, periodischem Scheduling liegt maximal bei der zeitlichen Auflösung T_{timer} des Timers.

Beweis: Ein Timer erzeuge Interrupts zu periodischen Intervallen $\left[\, 0,\ T_{timer}\,,\ 2\,T_{timer}\,,\ ...\right)$. Der Prozeß P_j sei bereits zu einem Zeitpunkt $t < 0$ gestartet worden und wird zum Zeitpunkt $t = 0$ abgearbeitet. Ein höherpriorer Prozeß P_i unterbreche zu einer Zeit $t \in \left(0,\ T_{timer}\right)$ die Ausführung des Prozesses P_j. Der Prozeß P_i kann im ungünstigsten Fall erst zum nächsten Interrupt T_{timer} mit seiner Abarbeitung beginnen, da er von Prozeß P_j blockiert wird. □

Abgeleitet aus Theorem 7.6 ergibt sich für die Durchführbarkeit des Schedulings bei Verwendung eines genaueren Zeitmodells Theorem 7.10.

Theorem 7.10: Für eine Prozeßschar von n Prozessen mit einer festgelegten Prioritätsordnung ist ein Scheduling genau dann durchführbar, wenn die folgende Bedingung erfüllt ist:

$$\forall i : 1 \le i \le n$$

$$\min_{(k,l)\in W_i} \left(\sum_{j=1}^{i} \frac{C_j + C_{preempt} + C_{exit}}{l\,T_k} \left\lceil \frac{l\,T_k}{T_j} \right\rceil \right) + \left\lceil \frac{l\,T_k}{T_{timer}} \right\rceil \frac{C_{timer}}{l\,T_k} + \frac{T_{timer}}{l\,T_k} \le 1 \quad (7.51)$$

$$W_i = \left\{ (k,l) \mid 1 \le k \le i\,,\ \ l = 1, \ldots,\ \left\lfloor \frac{T_i}{T_k} \right\rfloor \right\}$$

Diese Grenze ist als pessimistisch anzusehen, da der Prozeß P_i nur die Zeit C_{trap} zur regulären Beendigung des Prozesses aufwenden muß, jedoch nicht zwingend einen Kontextwechsel zum nächsten Prozeß durchführen muß. Wird T_{timer} sehr klein gegen T_k, so wird die Auslastung der Abarbeitung durch die zeitliche Auflösung steigen. Für eine erfolgreiche Abarbeitung muß also mindestens

$$\left\lceil \frac{l\,T_k}{T_{timer}} \right\rceil \frac{C_{timer}}{l\,T_k} \le 1 \quad (7.52)$$

gelten, um überhaupt Zeit für die Abarbeitung der Prozesse zur Verfügung zu stellen. Im Grenzfall der Gleichheit wird die Prozessorleistung nur zur Abarbeitung von Interrupts benötigt (*Trashing*) und keine Verarbeitung des Modells durchgeführt.

Theorem 7.11: Die kleinstmögliche Timerauflösung T_{timer}, die bei einer zeitgesteuerten Abarbeitung mit ratenmonotoner oder pseudoraten-basierter Abarbeitung erreicht werden kann, wird durch den Prozeß mit der höchsten Priorität begrenzt. Die charakteristische Gleichung mit C_1 als Ausführungszeit und T_1 als Periode des Prozesses mit der höchsten Priorität lautet:

$$T_{timer} \le T_1 - \left(C_1 + C_{timer} \left\lceil \frac{T_1}{T_{timer}} \right\rceil + C_{preempt} + C_{exit} \right) \quad (7.53)$$

Beweis: Multipliziert man beide Seiten der in Theorem 7.10 erhaltenen Ungleichung mit $l\,T_k = T_1$, so ergibt sich mit $i = 1$, $C_j = C_1$ und $T_j = T_1$ die folgende Gleichung:

$$C_1 + C_{preempt} + C_{exit} + C_{timer} \left\lceil \frac{T_1}{T_{timer}} \right\rceil + T_{timer} \le T_1 \quad (7.54)$$

Diese Gleichung kann durch einfaches Umformen in Gleichung (7.53) überführt werden. □

Mit Hilfe von Theorem 7.11 kann der maximale Wert von T_{timer} bestimmt werden. Zu Beginn wird $T_{timer} = T_1$ gesetzt und iterativ der korrekte Wert von T_{timer} bestimmt. Dazu wird die rechte Seite

von Gleichung (7.53) mit den vorgegebenen Werten berechnet und der Wert als neuer Wert für T_{timer} erneut eingesetzt. So ergibt sich beispielsweise für die Werte $T_1 = 150$, $C_1 = 50$, $C_{preempt} + C_{exit} = 5$ und $C_{timer} = 10$ die folgende Bestimmung von T_{timer}:

$$T_{timer} = 150 : \quad T_{timer} \leq 150 - \left(50 + 10 \left\lceil \frac{150}{150} \right\rceil + 5\right) = 85$$

$$T_{timer} = 85 : \quad T_{timer} \leq 150 - \left(50 + 10 \left\lceil \frac{150}{85} \right\rceil + 5\right) = 75$$

$$T_{timer} = 75 : \quad T_{timer} \leq 150 - \left(50 + 10 \left\lceil \frac{150}{75} \right\rceil + 5\right) = 75$$

Für dieses Beispiel ergibt sich also als maximaler Wert für T_{timer} von 75. Aus Theorem 7.10 kann auch ein minimaler Wert für T_{timer} bestimmt werden.

Theorem 7.12: Kann eine zeitliche Auflösung T_{timer} bestimmt werden, für die ein erfolgreiches Scheduling der Prozeßschar durchgeführt wird, so existiert auch eine untere Schranke für die Auflösung, ab der kein erfolgreiches Scheduling der Prozeßschar durchgeführt werden kann. Die erhaltene zeitliche Auflösung wird als *minimale zeitliche Auflösung* $T_{timer,min}$ bezeichnet.

Beweis: Für eine Prozeßschar kann ein erfolgreiches Scheduling durchgeführt werden. Wird die zeitliche Auflösung T_{timer} immer weiter verkleinert, so folgt nach Gleichung (7.53), daß der Term $C_{timer} \left\lceil \frac{T_1}{T_{timer}} \right\rceil \rightarrow \infty$ gegen unendlich strebt und für die Prozeßschar kein erfolgreiches Scheduling gefunden werden kann. Der Punkt, an dem für die Prozeßschar gerade noch ein erfolgreiches Scheduling bestimmt werden kann, stellt das Minimum der Auflösung dar. □

Für das obige Beispiel kann das Minimum der zeitlichen Auflösung bestimmt werden, indem die Grenze bestimmt wird, an der T_{timer} negativ wird. Somit gilt folgende Gleichung:

$$\begin{aligned} &150 - \left(50 + 10 \left\lceil \frac{150}{T_{timer}} \right\rceil + 5\right) \leq 0 \\ &\left\lceil \frac{150}{T_{timer}} \right\rceil \geq 9,5 \quad \Rightarrow \quad T_{timer,min} > 14 \end{aligned} \qquad (7.55)$$

Definition 7.9: Die *optimale zeitliche Auflösung* $T_{timer,opt}$ liegt dann vor, wenn die Prozeßschar mit der größtmöglichen Auslastung (engl. *breakdown utilization*) abgearbeitet wird.

Nach Definition 7.9 bedeutet eine optimale Auflösung für Rapid Prototyping, daß die Berechnung der Prozeßschar mit der größtmöglichen Genauigkeit durchgeführt wird. Bild 7-16 zeigt qualitativ den Verlauf der maximal erreichbaren Prozessorausnutzung U über der zeitlichen Auflösung. Die pessimistisch geschätzte obere Grenze nach Theorem 7.4 liegt unterhalb der idealen oberen Grenze nach Theorem 7.6. Beide Grenzen verlaufen linear ohne Abhängigkeit von der zeitlichen Auflösung. Bezieht man die zeitliche Auflösung nach Theorem 7.10 ein, so erkennt man zwei Effekte, die die Prozessorausnutzung verringern. Auf der rechten Seite von Bild 7-16 ist ein starker linearer Abfall zu beobachten. Dieser Effekt hat seine Ursache in den Auswirkungen der Blockierung auf den höchstprioren Prozeß. Jedes weitere Ansteigen führt ab dieser zeitlichen Auflösung $T_{timer,block}$ zu einem direkten linearen Abfall. Auf der linken Seite von Bild 7-16 erkennt man eben-

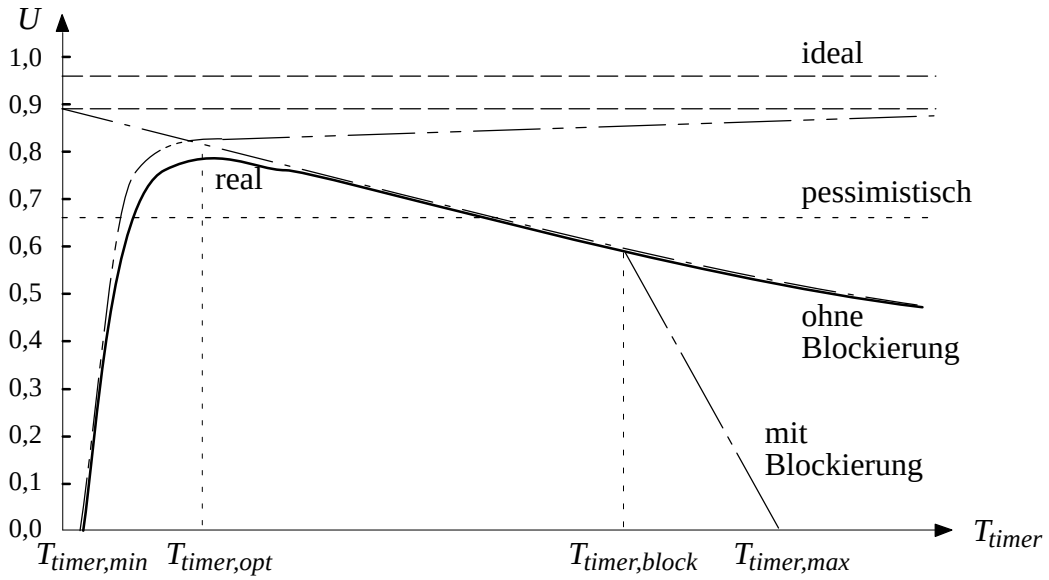


Bild 7-16: Verlauf der erreichbaren Prozessorausnutzung *U*

falls einen starken Abfall der Prozessorausnutzung bei einer stark verkleinerten zeitlichen Auflösung. Dieser Abfall hat seine Ursache im zusätzlichen Zeitaufwand, den die Interrupt-Abarbeitung der zeitlichen Auflösung mit sich bringt. Für $T_{timer} \ll l\,T_k$ kann der Term des zusätzlichen Timer-Aufwands aus Theorem 7.10 approximiert werden:

$$\left\lceil \frac{l\,T_k}{T_{timer}} \right\rceil \frac{C_{timer}}{l\,T_k} \approx \left(\frac{l\,T_k}{T_{timer}} \right) \frac{C_{timer}}{l\,T_k} = \frac{C_{timer}}{T_{timer}} \qquad (7.56)$$

Somit steigt der Aufwand für den zusätzlichen Timer-Aufwand für $T_{timer} \to 0$ stark an und die Prozessorausnutzung sinkt. Eine Grenze wird bei $C_{timer} = T_{timer}$ erreicht. Zu diesem Zeitpunkt wird die gesamte Zeit zwischen den Interrupts zur Interrupt-Abarbeitung verwendet.

Zwischen diesen beiden Effekten befindet sich die Prozessorausnutzung auf nahezu konstantem Niveau, die allerdings nach einer zeitlichen Auflösung $T_{timer,opt}$ leicht asymptotisch abfällt. In diesem Gebiet existiert mit $T_{timer,opt}$ eine zeitliche Auflösung, bei der der höchste Wert erreicht wird, der für die Abarbeitung der Prozeßschar unter völliger Prozessorausnutzung verwendet werden kann. Vergrößert man die zeitliche Auflösung, so wachsen die Zeiten, zu denen der Prozessor nicht beschäftigt ist und die Genauigkeit der Berechnungen sinkt. Aus diesem Grund fällt die Prozessorausnutzung leicht ab. Um die optimale zeitliche Auflösung $T_{timer,opt}$ zu berechnen, müssen die in Theorem 7.11 angegebenen Gleichungen für zeitgesteuertes Scheduling jeweils für die gesamte Prozeßschar mit unterschiedlichen zeitlichen Auflösungen iteriert werden.

Als Beispiel für Theorem 7.10 können die Prozesse von Tabelle 7-5 nach ratenmonotoner und pseudoraten-basierter Abarbeitung ausgeführt werden.

Prozeß i	Ausführungszeit C_i	Ratenmonotone Periode T_i	Pseudoraten Periode T_i'	Verhältnis C_i/T_i'	Ausnutzungsfaktor 1 .. N	Obere Grenze
1	70	250	240	0,292	0,292	1,000
2	90	290	280	0,321	0,613	0,828
3	140	720	720	0,194	0,807	0,779

Tabelle 7-5: Beispiel für pseudoraten-basiertes Scheduling mit genauerer Zeitabschätzung

Auch für dieses Beispiel wird angenommen, daß die ratenmonotone und pseudoraten-basierte Abarbeitung mit einer zeitlichen Auflösung von 20 geschieht. Die Untersuchung auf Durchführbarkeit des Schedulings kann sich auf den Fall $i = 3$ von Gleichung (7.51) beschränken, da sich die Prozessorausnutzung der Prozesse P_1 und P_2 unterhalb der oberen Grenze nach Gleichung (7.47) befindet. Die weiteren Variablen lassen sich für die pseudoraten-basierte Abarbeitung festlegen zu $j = 1, \ldots, 3$; $k = 1, \ldots, 3$ sowie

$$k = 1\text{:}\ \left\lfloor \frac{720}{240} \right\rfloor = 3 \quad \Rightarrow \quad l = 1, \ldots, 3$$

$$k = 2\text{:}\ \left\lfloor \frac{720}{280} \right\rfloor = 2 \quad \Rightarrow \quad l = 1, 2$$

$$k = 3\text{:}\ \left\lfloor \frac{720}{720} \right\rfloor = 1 \quad \Rightarrow \quad l = 1$$

Die weiteren Variablen sollen die folgenden Werte annehmen: $C_{aux} = 2$, $C_{wd} = 4$, $C_{preempt} + C_{exit} = 1$. Gemäß Gleichung (7.51) ergeben sich die folgenden Ungleichungen:

$k = 1, l = 1$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{1 \cdot T_1'} \left\lceil \frac{1 \cdot T_1'}{T_1'} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{1 \cdot T_1'} \left\lceil \frac{1 \cdot T_1'}{T_2'} \right\rceil + \frac{C_3 + C_{preempt} + C_{exit}}{1 \cdot T_1'} \left\lceil \frac{1 \cdot T_1'}{T_3'} \right\rceil + \left\lceil \frac{1 \cdot T_1'}{T_{aux}} \right\rceil \frac{C_{aux}}{1 \cdot T_1'} + \frac{T_{aux}}{1 \cdot T_1'} = \frac{71 + 91 + 141 + 24 + 20}{1 \cdot 240} = 1,4458 > 1$$

$k = 1, l = 2$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{2 \cdot T_1'} \left\lceil \frac{2 \cdot T_1'}{T_1'} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{2 \cdot T_1'} \left\lceil \frac{2 \cdot T_1'}{T_2'} \right\rceil + \frac{C_3 + C_{preempt} + C_{exit}}{2 \cdot T_1'} \left\lceil \frac{2 \cdot T_1'}{T_3'} \right\rceil + \left\lceil \frac{2 \cdot T_1'}{T_{aux}} \right\rceil \frac{C_{aux}}{2 \cdot T_1'} + \frac{T_{aux}}{2 \cdot T_1'} = \frac{2 \cdot 71 + 2 \cdot 91 + 141 + 48 + 20}{2 \cdot 240} = 1,1104 > 1$$

$k = 1, l = 3$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{3 \cdot T_1'} \left\lceil \frac{3 \cdot T_1'}{T_1'} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{3 \cdot T_1'} \left\lceil \frac{3 \cdot T_1'}{T_2'} \right\rceil +$$
$$\frac{C_3 + C_{preempt} + C_{exit}}{3 \cdot T_1'} \left\lceil \frac{3 \cdot T_1'}{T_3'} \right\rceil + \left\lceil \frac{3 \cdot T_1'}{T_{aux}} \right\rceil \frac{C_{aux}}{3 \cdot T_1'} + \frac{T_{aux}}{3 \cdot T_1'} =$$
$$\frac{3 \cdot 71 + 3 \cdot 91 + 141 + 72 + 20}{3 \cdot 240} = 0,9986 \overset{!}{<} 1$$

$k = 2, l = 1$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{1 \cdot T_2'} \left\lceil \frac{1 \cdot T_2'}{T_1'} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{1 \cdot T_2'} \left\lceil \frac{1 \cdot T_2'}{T_2'} \right\rceil +$$
$$\frac{C_3 + C_{preempt} + C_{exit}}{1 \cdot T_2'} \left\lceil \frac{1 \cdot T_2'}{T_3'} \right\rceil + \left\lceil \frac{1 \cdot T_2'}{T_{aux}} \right\rceil \frac{C_{aux}}{1 \cdot T_2'} + \frac{T_{aux}}{1 \cdot T_2'} =$$
$$\frac{2 \cdot 71 + 91 + 141 + 28 + 20}{1 \cdot 280} = 1,5071 > 1$$

$k = 2, l = 2$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{2 \cdot T_2'} \left\lceil \frac{2 \cdot T_2'}{T_1'} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{2 \cdot T_2'} \left\lceil \frac{2 \cdot T_2'}{T_2'} \right\rceil +$$
$$\frac{C_3 + C_{preempt} + C_{exit}}{2 \cdot T_2'} \left\lceil \frac{2 \cdot T_2'}{T_3'} \right\rceil + \left\lceil \frac{2 \cdot T_2'}{T_{aux}} \right\rceil \frac{C_{aux}}{2 \cdot T_2'} + \frac{T_{aux}}{2 \cdot T_2'} =$$
$$\frac{3 \cdot 71 + 2 \cdot 91 + 141 + 56 + 20}{2 \cdot 280} = 1,0929 > 1$$

$k = 3, l = 1$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{1 \cdot T_3'} \left\lceil \frac{1 \cdot T_3'}{T_1'} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{1 \cdot T_3'} \left\lceil \frac{1 \cdot T_3'}{T_2'} \right\rceil +$$
$$\frac{C_3 + C_{preempt} + C_{exit}}{1 \cdot T_3'} \left\lceil \frac{1 \cdot T_3'}{T_3'} \right\rceil + \left\lceil \frac{1 \cdot T_3'}{T_{aux}} \right\rceil \frac{C_{aux}}{1 \cdot T_3'} + \frac{T_{aux}}{1 \cdot T_3'} =$$
$$\frac{3 \cdot 71 + 3 \cdot 91 + 141 + 72 + 20}{1 \cdot 720} = 0,9986 \overset{!}{<} 1$$

Sowohl für $k = 1, l = 3$ als auch $k = 3, l = 1$ ist die entstandene Ungleichung erfüllt. Nach Theorem 7.10 ist ein Scheduling für die Prozeßschar machbar. Bild 7-17 zeigt die zu erfüllende Ungleichung $3 \cdot 71 + 3 \cdot 91 + 141 + \left\lceil \frac{720}{T_{aux}} \right\rceil 2 + T_{aux} \overset{!}{<} 720$. Auf diese Weise kann der Lösungsraum an zeitlichen Auflösungen T_{aux} bestimmt werden, für die ein machbares Scheduling der Prozeßschar gefunden werden kann.

Obwohl die Perioden für ratenmonotones Scheduling im Vergleich zu pseudoraten-basiertem Scheduling länger sind, zeigt die folgende Menge von Ungleichungen, daß für die Prozeßschar mit ratenmonotonem Scheduling keine Lösung existiert.

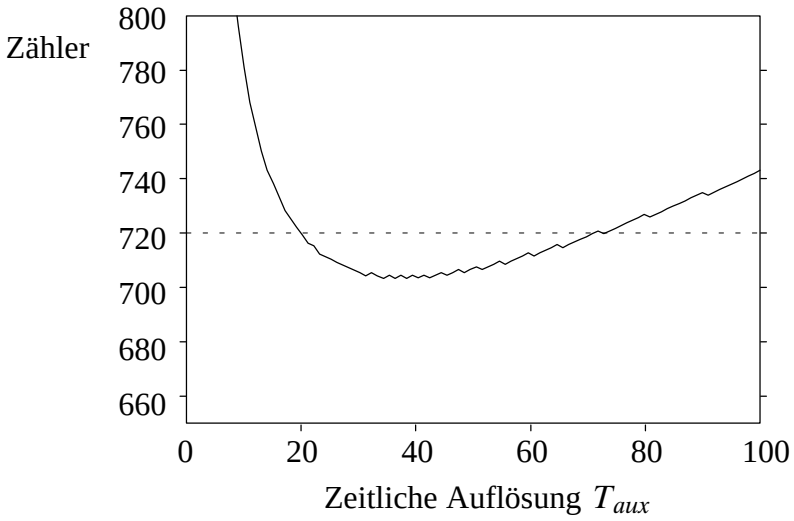


Bild 7-17: Grafische Darstellung der Ungleichung für k=1, l=3

$k = 1, l = 1$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{1 \cdot T_1} \left\lceil \frac{1 \cdot T_1}{T_1} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{1 \cdot T_1} \left\lceil \frac{1 \cdot T_1}{T_2} \right\rceil + \frac{C_3 + C_{preempt} + C_{exit}}{1 \cdot T_1} \left\lceil \frac{1 \cdot T_1}{T_3} \right\rceil + \left\lceil \frac{1 \cdot T_1}{T_{wd}} \right\rceil \frac{C_{wd}}{1 \cdot T_1} + \frac{T_{wd}}{1 \cdot T_1} = \frac{71 + 91 + 141 + 52 + 20}{1 \cdot 250} = 1,5 > 1$$

$k = 1, l = 2$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{2 \cdot T_1} \left\lceil \frac{2 \cdot T_1}{T_1} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{2 \cdot T_1} \left\lceil \frac{2 \cdot T_1}{T_2} \right\rceil + \frac{C_3 + C_{preempt} + C_{exit}}{2 \cdot T_1} \left\lceil \frac{2 \cdot T_1}{T_3} \right\rceil + \left\lceil \frac{2 \cdot T_1}{T_{wd}} \right\rceil \frac{C_{wd}}{2 \cdot T_1} + \frac{T_{wd}}{2 \cdot T_1} = \frac{2 \cdot 71 + 2 \cdot 91 + 141 + 100 + 20}{2 \cdot 250} = 1,17 > 1$$

$k = 2, l = 1$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{1 \cdot T_2} \left\lceil \frac{1 \cdot T_2}{T_1} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{1 \cdot T_2} \left\lceil \frac{1 \cdot T_2}{T_2} \right\rceil + \frac{C_3 + C_{preempt} + C_{exit}}{1 \cdot T_2} \left\lceil \frac{1 \cdot T_2}{T_3} \right\rceil + \left\lceil \frac{1 \cdot T_2}{T_{wd}} \right\rceil \frac{C_{wd}}{1 \cdot T_2} + \frac{T_{wd}}{1 \cdot T_2} = \frac{2 \cdot 71 + 91 + 141 + 60 + 20}{1 \cdot 290} = 1,5655 > 1$$

$k = 2, l = 2$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{2 \cdot T_2} \left\lceil \frac{2 \cdot T_2}{T_1} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{2 \cdot T_2} \left\lceil \frac{2 \cdot T_2}{T_2} \right\rceil + \frac{C_3 + C_{preempt} + C_{exit}}{2 \cdot T_2} \left\lceil \frac{2 \cdot T_2}{T_3} \right\rceil + \left\lceil \frac{2 \cdot T_2}{T_{wd}} \right\rceil \frac{C_{wd}}{2 \cdot T_2} + \frac{T_{wd}}{2 \cdot T_2} = \frac{3 \cdot 71 + 2 \cdot 91 + 141 + 116 + 20}{2 \cdot 290} = 1,1586 > 1$$

$k = 3, l = 1$:

$$\frac{C_1 + C_{preempt} + C_{exit}}{1 \cdot T_3} \left\lceil \frac{1 \cdot T_3}{T_1} \right\rceil + \frac{C_2 + C_{preempt} + C_{exit}}{1 \cdot T_3} \left\lceil \frac{1 \cdot T_3}{T_2} \right\rceil +$$
$$\frac{C_3 + C_{preempt} + C_{exit}}{1 \cdot T_3} \left\lceil \frac{1 \cdot T_3}{T_3} \right\rceil + \left\lceil \frac{1 \cdot T_3}{T_{wd}} \right\rceil \frac{C_{wd}}{1 \cdot T_3} + \frac{T_{wd}}{1 \cdot T_3} =$$
$$\frac{3 \cdot 71 + 3 \cdot 91 + 141 + 144 + 20}{1 \cdot 720} = 1{,}0986 > 1$$

Auch dieser Verlauf kann grafisch dargestellt werden. Bild 7-18 zeigt den Verlauf der zeitlichen Auflösung T_{wd} über dem Zählerpolynom für den Fall $k = 3$ und $l = 1$. Die entstehende Ungleichung unter Berücksichtigung von Blockade zeigt, daß für die Prozeßschar kein machbares Scheduling gefunden werden kann. Erst nach Vernachlässigung des Blockadeterms ist das Scheduling für die Ungleichung $3 \cdot 71 + 3 \cdot 91 + 141 + \left\lceil \frac{720}{T_{wd}} \right\rceil 4 \overset{!}{<} 720$ durchführbar. Die optimale zeitliche Auflösung wird für $T_{wd,opt} = 32$ gefunden.

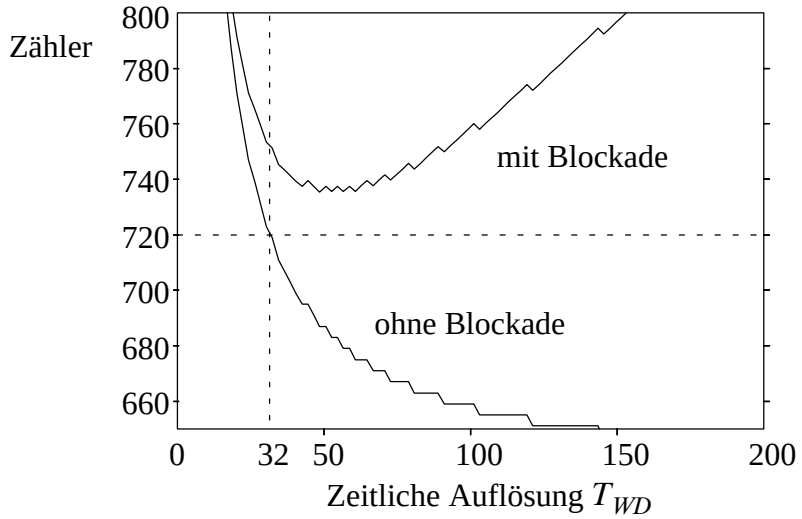


Bild 7-18: Grafische Darstellung der Ungleichung mit und ohne Berücksichtigung der Blockade

Die Ergebnisse und Messungen der beiden unterschiedlichen Schedulingarten ratenmonoton und pseudoraten-basiert werden in Kapitel 9 vorgestellt. Dabei wird insbesondere auf die Geschwindigkeitsvorteile des pseudoraten-basierten Schedulings bei einer großen Anzahl von Prozessen eingegangen.

7.3 Abschätzung der Ausführungszeit

Um Aussagen über die Durchführbarkeit des Schedulings machen zu können, benötigt man die Ausführungszeit, die für die Abarbeitung eines Prozesses notwendig ist. Die Ausführungszeit hängt von mehreren Faktoren ab, beispielsweise von dem verwendeten Prozessor, dem Prozessortakt, dem Cache, dem Echtzeitbetriebssystem, den Interruptlatenzzeiten des Echtzeitbetriebssystems auf dem verwendeten Prozessor, dem verwendeten Modell, Zeitpunkt und Vorgeschichte des Modells, dem verwendeten Compiler, etc. Dies erschwert die genaue Abschätzung beträchtlich.

Untersuchungen im Bereich von Ausführungszeiten von Software werden bereits seit einigen Jahren durchgeführt [PaSh91]. Dabei werden zwei Klassen von Untersuchungen unterschieden. Die *stochastischen Untersuchungen* betrachten die durchschnittliche Laufzeit von Programmteilen und liefern so nur ungenaue Abschätzungen. *Deterministische Untersuchungen* betrachten einzelne Programmteile analytisch und erreichen deswegen wesentlich genauere Ergebnisse. Ausgangspunkt dieser Betrachtungen ist das abzuarbeitende Programm, in das alle bekannten Randbedingungen eingearbeitet werden. Ein weiteres Unterscheidungsmerkmal besteht in der Ebene der Untersuchung. Während gängige Ansätze diese Untersuchung auf Maschinensprache-Ebene vornehmen [MACT89], gehen fortschrittlichere Ansätze bereits von höheren, komplexeren prozeduralen oder objekt-orientierten Sprachen wie Pascal, C oder C++ aus.

Eine Untersuchung in Maschinensprache erreicht die genauesten Abschätzungen, läßt jedoch Rückschlüsse auf das Modell nur sehr begrenzt zu und erfordert einen hohen Berechnungsaufwand. Außerdem muß für diese Art der Abschätzung bereits ein vollständiges Programm in Maschinensprache vorliegen. Eine Schwierigkeit bei der Abschätzung liegt auch in dem selten vorliegenden sequentiellen Verhalten von Programmen, das eine eindeutige Bestimmung der Ausführungszeiten erschwert.

Geht man von einer Hochsprache aus [PaSh91], so kann die Tatsache ausgenutzt werden, daß eine zusammengesetzte Folge aus Befehlen in Maschinensprache aus den Hochsprachenkonstrukten gebildet wird. Die Folge von Befehlen in Maschinensprache, die ein Hochsprachenkonstrukt repräsentieren, wird als *Segment* bezeichnet. Die Ausführungszeiten von Segmenten lassen sich aus der Summe der Ausführungszeiten der Befehle in Maschinensprache bestimmen. Vorteilhaft bei dieser Methode ist, daß die Ausführungszeiten von Befehlen, die mehrfach im Programm vorkommen, nur einmal bestimmt werden muß. Andere Ansätze [Stoy87] teilen das in Hochsprache vorliegende Programm in Segmente auf und ordnen diese in Form eines Baums an, der die Programmstruktur nachbildet.

Für das hier verwendete Rapid Prototyping müssen die Untersuchungen noch eine Abstraktionsebene oberhalb der Hochsprachenebene durchgeführt werden, da die Ausführungszeiten von Modellen bestimmt werden müssen. Die Modelle werden dann in eine Hochsprache und die Hochsprache in Maschinensprache übersetzt. Für die Betrachtungen im Bereich Rapid Prototyping ist nur der schlechteste mögliche Fall für die Abschätzung wichtig.

7.3.1 Abschätzung von diskreten Modellen

Im folgenden werden die in Kapitel 2.1.1 beschriebenen Statecharts, die sowohl in Zustandsautomaten als auch in Petri-Netze umformbar sind, als Grundlage der Abschätzung der Ausführungszeiten herangeführt. Die Betrachtungen können auch auf der Ebene des CASE Datenaustauschformats CDIF vorgenommen werden, da in der State/Event Subject Area von CDIF (siehe Kapitel 6.4) sämtliche Informationen über Labels an Zustandsübergängen, Hierarchieebene, Parallelitäten, etc. abgelegt sind.

Den Grundelementen Ereignis, Bedingung und Aktion eines Zustandsübergangs wird bei Ausführung auf einem Prozessor eine elementare Ausführungszeit zugeordnet. Die Ausführungszeit für Ereignisse wird mit C_E, für Bedingungen mit C_C und für Aktionen mit C_A bezeichnet. Dabei werden zur Vereinfachung jedem der möglichen Grundelemente eine gemeinsame Elementarzeit zugeordnet. Die Unterscheidung der Elementarzeiten ändert nichts am Prinzip der Zeituntersuchung und

kann jederzeit nachträglich eingeführt werden. Die Ausführungszeit ist in hohem Maß abhängig von der Zielarchitektur und muß durch Zeitmessungen bestimmt werden.

Ausgehend von diesen Elementarzeiten kann berechnet werden, wie lange ein Zustandsübergang für seine Ausführung benötigen wird. Dazu muß ermittelt werden, wieviele dieser Grundelemente der Zustandsübergang besitzt. Diese Zeit wird als Zustandsübergangszeit C_i bezeichnet. Die Variablen q_i, r_i und s_i repräsentieren die Anzahl der durchzuführenden Berechnungen während eines Modellschritts.

$$C_i = q_i C_E + r_i C_C + s_i C_A \qquad (7.57)$$

Um den Systemzustand zu bestimmen, muß die gesamte Zahl der durchzuführenden Zustandsübergänge ermittelt werden.

Systemzustand ohne Hierarchie und Parallelität

Besitzt ein Systemzustand keine Hierarchie und Parallelität, so kann nur ein Zustand gleichzeitig aktiv sein. Um den Folgezustand dieses aktiven Zustands zu ermitteln, müssen bei allen vom Zustand abgehenden Zustandsübergängen überprüft werden, ob die Ereignisse und Bedingungen erfüllt sind. Diese Zeit wird als Abfragezeit $C_{i,EC}$ bezeichnet, wobei $p_i = |P_i|$ die Mächtigkeit aller möglichen Zustandsübergänge P_i darstellt.

$$C_{i,EC} = \sum_{j=1}^{p_i} \left(q_j C_E + r_j C_C \right) \qquad (7.58)$$

Nach diesen Abfragen und der Nebenbedingung, daß keine Hierarchie und Parallelität verwendet werden, kann jedoch höchstens einer der Zustandsübergänge tatsächlich ausgeführt werden. Für Zeitabschätzungen in diesem Zusammenhang muß derjenige Zustandsübergang gefunden werden, der die größte Antwortzeit $C_{i,Amax}$ benötigt.

$$C_{i,Amax} = \max\left\{ s_j C_A \,|\, j = 1,\ldots,p_i \right\} \qquad (7.59)$$

Die Berechnung der maximalen Zustandsübergangszeit eines Systemzustands $C_{i,max}$ kann mit der folgende Summe erfolgen.

$$C_{i,max} = C_{i,EC} + C_{i,Amax} \qquad (7.60)$$

Systemzustand mit Hierarchie ohne Parallelität

Enthält der Systemzustand mindestens einen hierarchischen Zustand, so müssen für die Berechnung der maximalen Zustandsübergangszeit eines Systemzustands $C_{i,max}$ alle im Systemzustand vorkommenden Zustandsübergänge ermittelt werden.

$$C_{i,max} = \max\left\{ C_{1,max} \,;\, C_{k,max} + \sum_{j=1}^{k-1} C_{j,EB} \,|\, k = 2,\ldots,h \right\} \qquad (7.61)$$

Die Variable h stellt dabei die Hierarchietiefe dar. Der erste Term $C_{1,max}$ repräsentiert die Zeit, die bei einem Zustandsübergang auf höchster Ebene benötigt werden würde. Wird ein Zustandsübergang auf einer hierarchisch höheren Ebene durchgeführt, so müssen die hierarchisch tieferen Ebenen nicht mehr überprüft werden. Geschieht ein Zustandsübergang auf einer hierarchisch niedrigeren Ebene, so muß $C_{k,max}$ auf seiner Hierarchieebene zuzüglich der Abfragezeiten, die die Überprü-

fung der Übergangsbedingungen aller hierarchisch höher liegenden Ebenen gekostet haben, berücksichtigt werden.

Systemzustand mit Hierarchie und Parallelität

Durch die Einschränkung auf Einprozessorsysteme (siehe Kapitel 7.1) müssen Systemzustände mit parallelen Zuständen zeitlich nacheinander abgearbeitet werden. Die Zeit zur Bestimmung des nächsten Systemzustands wird in diesem Fall aus der Summe der Zustandsübergangszeiten der parallelen Zustände bestimmt.

$$C_{i,max} = \sum_{j=1}^{o_i} C_{j,max} \qquad (7.62)$$

Die Variable o_i repräsentiert dabei die Anzahl der parallelen Zustände. Eine Verringerung dieser maximalen Zustandsübergangszeit kann erzielt werden, wenn durch den Zustandsübergang eines parallelen Zustands der parallele Systemzustand (oder ein Teil von ihm) verlassen wird, da in diesem Fall die Zeiten der Zustandsübergänge der verlassenen nebenläufigen Zustände nicht mehr berücksichtigt werden müssen.

7.3.2 Abschätzung von kontinuierlichen Modellen

Bei kontinuierlichen Modellen wird zur Abarbeitung der zustandsabhängigen Funktionsblöcke eine Umformung in den Zustandsraum vorgenommen (Kapitel 2.1.2). Die darin enthaltenen Differentialgleichungen erster Ordnung werden als Differenzengleichungen berechnet. Die Zustandsvariablen werden dabei zu Beginn mit ihren Anfangswerten belegt. Bei einem Modellschritt wird zunächst die Ableitung der Zustände bestimmt, eine Integration durchgeführt und anhand der neu berechneten Zustände die Ausgangssignale berechnet. Die Zeit, die zur Berechnung eines solchen Funktionsblocks benötigt wird, kann folgendermaßen bestimmt werden:

$$C_{i,max} = C_{dot} + d_i \, c_{alg} \, C_{integ} + C_{out} \qquad (7.63)$$

$C_{i,max}$ ist also hauptsächlich abhängig von der Zeit C_{dot}, die für die Berechnung der Ableitung benötigt wird, von der Zeit C_{integ}, die zur Berechnung der Integration benötigt wird und von der Zeit C_{out}, die zur Berechnung der Ausgangsvariablen benötigt wird. Da eine Umwandlung einer Differentialgleichung d_i-ten Grades (Dimension des Zustandsraums) in d_i Differentialgleichungen ersten Grades vorgenommen wurde, muß die Integration für jede Dimension d_i des Zustandsraums einmal durchgeführt werden. Auch der gewählte Integrationsalgorithmus spielt eine wesentliche Rolle bei der Zeit, die für eine Integration benötigt wird. Beispielsweise benötigt ein Euler-Algorithmus in etwa die Hälfte der Zeit, die für einen Runge-Kutta-Algorithmus 2. Ordnung aufgewendet werden muß, da die Integrationsbildung beim Runge-Kutta Verfahren zweimal durchlaufen wird. Diese Abhängigkeit beeinflußt über den Faktor c_{alg} die Berechnungszeit.

Da ein System in der Regel aus Zustandsräumen und algebraischen Verknüpfungen besteht, wird algebraischen Verknüpfungen eine gemeinsame Ausführungszeit C_{exp} zugeordnet. Eine Unterscheidung der Ausführungszeiten ändert wie bei den diskreten Modellen nichts am Prinzip der Zeituntersuchung und kann jederzeit nachträglich eingeführt werden.

7.4 Auswirkungen des Schedulings

Um ein pseudoraten-basiertes Scheduling zu ermöglichen, kann es erforderlich sein, eine Reduktion der ursprünglich gewünschten Periode vornehmen zu müssen. Diese Reduktion hat auf die dis-

kreten und kontinuierlichen Modelle unterschiedliche Auswirkungen. Prozesse mit kontinuierlichen Modellen werden durch die kürzere Periode eine höhere Genauigkeit bei der Abarbeitung von Integralgleichungen erreichen, da Integrationsalgorithmen mit einer festen Schrittweite verwendet werden wie beispielsweise Algorithmen nach Euler oder Runge-Kutta. Die Genauigkeit dieser Algorithmen steigt mit kleineren Perioden, da eine verbesserte Annäherung (Untersummen- oder Obersummenbildung) an die abzuarbeitende Funktion erreicht werden kann. Die Erzeugung von Signalen wie beispielsweise PWM-Signale (engl. *pulse width modulation*) kann dagegen zu Problemen führen. Sollen Signale erzeugt werden, die sich nicht auf der Zeitbasis des Auxiliary-Timers befinden, wird sich das erzeugte Signal von dem gewünschten Signal unterscheiden. Nach dem Abtasttheorem beträgt die maximale Abweichung τ:

$$\tau = \frac{\delta}{2} \qquad (7.64)$$

Um dieses Problem zu lösen, muß die notwendige Auflösung δ nicht nur durch die Abtastraten der kontinuierlichen Prozesse, sondern auch durch die zeitlichen Randbedingungen der zu erzeugenden Signale festgelegt werden.

Ein ähnliches Problem tritt bei Prozessen mit diskreten Modellen auf, da auch die Modellierung absoluter Zeiten durch Einbeziehung der Zustandsübergangszeiten Probleme bereiten kann. Geht ein Entwickler bei der Aufstellung eines Zustandsautomaten implizit davon aus, daß ein Zustandsübergang innerhalb eines Zeitintervalls ausgeführt wird, so kann sich eine Änderung dieses Zeitintervalls auch auf das Systemverhalten auswirken. Unproblematisch dagegen ist die Abarbeitung beim Eintreffen zweier Ereignisse innerhalb einer Periode. Bild 7-19 zeigt ein Beispiel für diesen Fall.

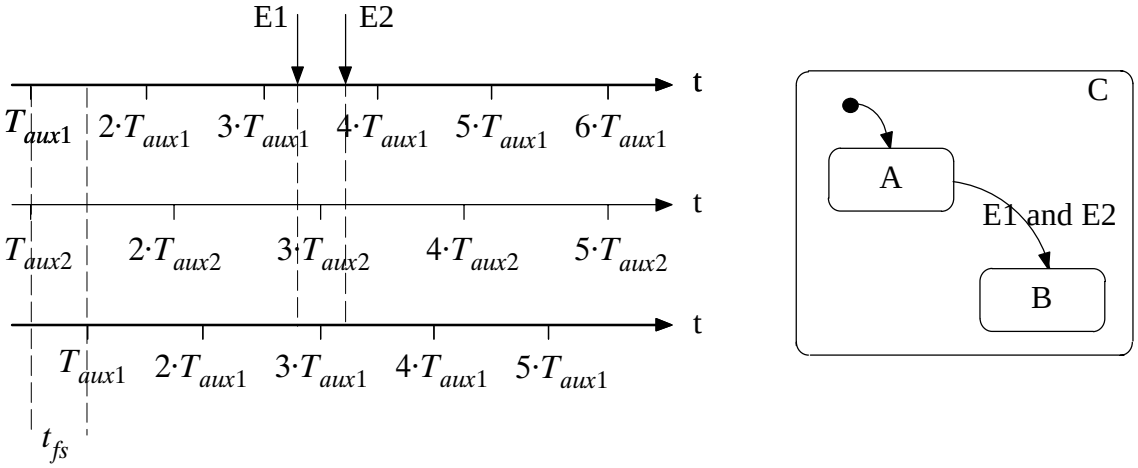


Bild 7-19: Diskretes Modellverhalten bei unterschiedlichen Perioden

Der Übergang von Zustand A nach Zustand B eines Statecharts findet nur dann statt, wenn E1 und E2 innerhalb des gleichen Modellschritts verarbeitet werden. Der Zeitstrahl in Bild 7-19 links oben zeigt die Periode des Auxiliary-Timers 1. Die beiden Ereignisse E1 und E2 treffen zwischen $3{\cdot}T_{aux1} \le t < 4{\cdot}T_{aux1}$ ein. Somit würde ein Zustandsübergang ausgelöst.

Wird das Modell mit dem Zeitstrahl in Bild 7-19 links in der Mitte abgearbeitet, so erkennt man, daß das Ereignis E1 zum Zeitpunkt $2{\cdot}T_{aux2} \le t < 3{\cdot}T_{aux2}$ eintrifft und das Ereignis E2 zum Zeitpunkt $3{\cdot}T_{aux2} \le t < 4{\cdot}T_{aux2}$. Ein Zustandsübergang wird nicht ausgelöst, da die beiden Ereignisse nicht

im gleichen Zeitintervall eingetroffen sind. Die Änderung der Periode des Auxiliary-Timers führt also in diesem Fall zu unterschiedlichem Verhalten.

Eine solche asynchrone Modellierung sollte vermieden werden, da das gleiche Fehlverhalten auch durch einen Frequenzverschiebung (engl. *frequence shift*) vorkommen kann. Da asynchrone Ereignisse wie im Beispiel von Bild 7-19 völlig unkorreliert zu der Abarbeitung der Modellierung geschehen, kann auch eine Verschiebung des Ablaufs um t_{fs} vorkommen. Auch in diesem Fall wird das Modell keinen Zustandsübergang durchführen, da die beiden Ereignisse E1 und E2 nicht im gleichen Zeitintervall eingegangen sind. Eine solche Modellierung sollte also in jedem Fall vermieden werden und führt nicht zwingend wegen einer unterschiedlichen Auxiliary-Timer Periode zu unterschiedlichem Verhalten.

Die obigen Effekte sind unerwünscht und sollten nach Möglichkeit vermieden werden. Eine Möglichkeit ist die Wahl einer Abtastrate in Vielfachen einer gewünschten notwendigen Auflösung δ. Diese kann bereits vor der eigentlichen Modellierung festgelegt werden und vermeidet Ungenauigkeiten bei einer Anpassung der Abtastrate oder die Überschreitung einer physikalischen Grenze der Auflösung.

8 Erzeugung von Echtzeitcode

Um eine Rapid Prototyping Applikation ausführen zu können, werden zwei Komponenten benötigt. Eine Komponente umfaßt das Echtzeitbetriebssystem, das für die Bereitstellung einer genormten, hardwareunabhängigen Schnittstelle für Systemaufrufe und für die Bereitstellung von Betriebsmitteln wie Timern, Scheduler, Netzwerkzugriffe, etc. zuständig ist. Die andere Komponente stellt die eigentliche Applikation in Form des Software-Prototypen dar.

8.1 Partitionierung des Echtzeitcodes

Zur Erstellung des Software–Prototypen ist es vorteilhaft, eine weitere Aufteilung vorzunehmen. Der *Modell-Code* repräsentiert die Verhaltensbeschreibung des zu entwickelnden Systems und besteht hauptsächlich aus einem C-Code, der auf Basis der Modelldaten, die in CDIF Datenformat abgelegt sind, mittels eines Codegenerators erzeugt wurde. Für den diskreten Bereich werden die Systemzustände und Zustandsübergänge durch entsprechende Konstrukte nachgebildet. Für den kontinuierlichen Bereich handelt es sich um algebraische Ausdrücke und Differentialgleichungen, die im Programmcode nachgebildet werden.

Der *Ein-/Ausgabe-Code* dient der Einbindung von Ein-/Ausgabe-Hardwarekomponenten in das System. Für jede Hardwarekomponente kann ein spezieller Programmcode eingebunden werden. Der Code besitzt für jede Komponente eine standardisierte Schnittstelle in Form von Funktionsaufrufen für die Initialisierung, Lesen, Schreiben und die Beendigung der Bearbeitung. Die Konfiguration des Codes geschieht über eine grafische Benutzerschnittstelle, die in Kapitel 9.1 näher erläutert wird.

Der *Scheduler-Code* stellt die wichtigste Komponente des Echtzeitcodes dar, da sie für die Ablaufsteuerung und Überwachung des Software-Prototypen verantwortlich ist. Der Aufbau des Schedulers ist hauptsächlich für die Abarbeitungsgeschwindigkeit des Systems verantwortlich. Die Aufgabe des Schedulers umfaßt u.a. die Kontrolle über den Modellablauf und Synchronisation der Modellkomponenten, Aufruf der Ein-/Ausgabe-Komponenten, die Kontrolle der Einhaltung der festgelegten Reaktionszeit des Echtzeitcodes und die Aufbereitung der Systemdaten für Prozeßvisualisierung und Fehlerdiagnose.

Um die drei Komponenten des Echtzeitcodes zu erzeugen, werden mehrere Schritte durchgeführt (Bild 8-1). Zunächst wird das System mit Hilfe von CASE-Werkzeugen (beispielsweise STATEMATE™ oder MATRIX$_X$™) modelliert. Die Modelle, die in werkzeugspezifischen Datenformaten vorliegen, werden über spezielle Compiler in das Datenformat CDIF überführt. Über den Compiler LINKCDIF werden die einzelnen CDIF-Modelle zu einem CDIF-Gesamtmodell zusammengefügt, aus dem der Modell-Code erzeugt wird. Zusätzlich zur Erzeugung des Modell-Codes werden die Modellein- und ausgänge einem Konfigurationswerkzeug zur Verfügung gestellt. Mit diesem Werkzeug können die Modellein- und ausgänge den physikalischen Signalen der Umgebung zuge-

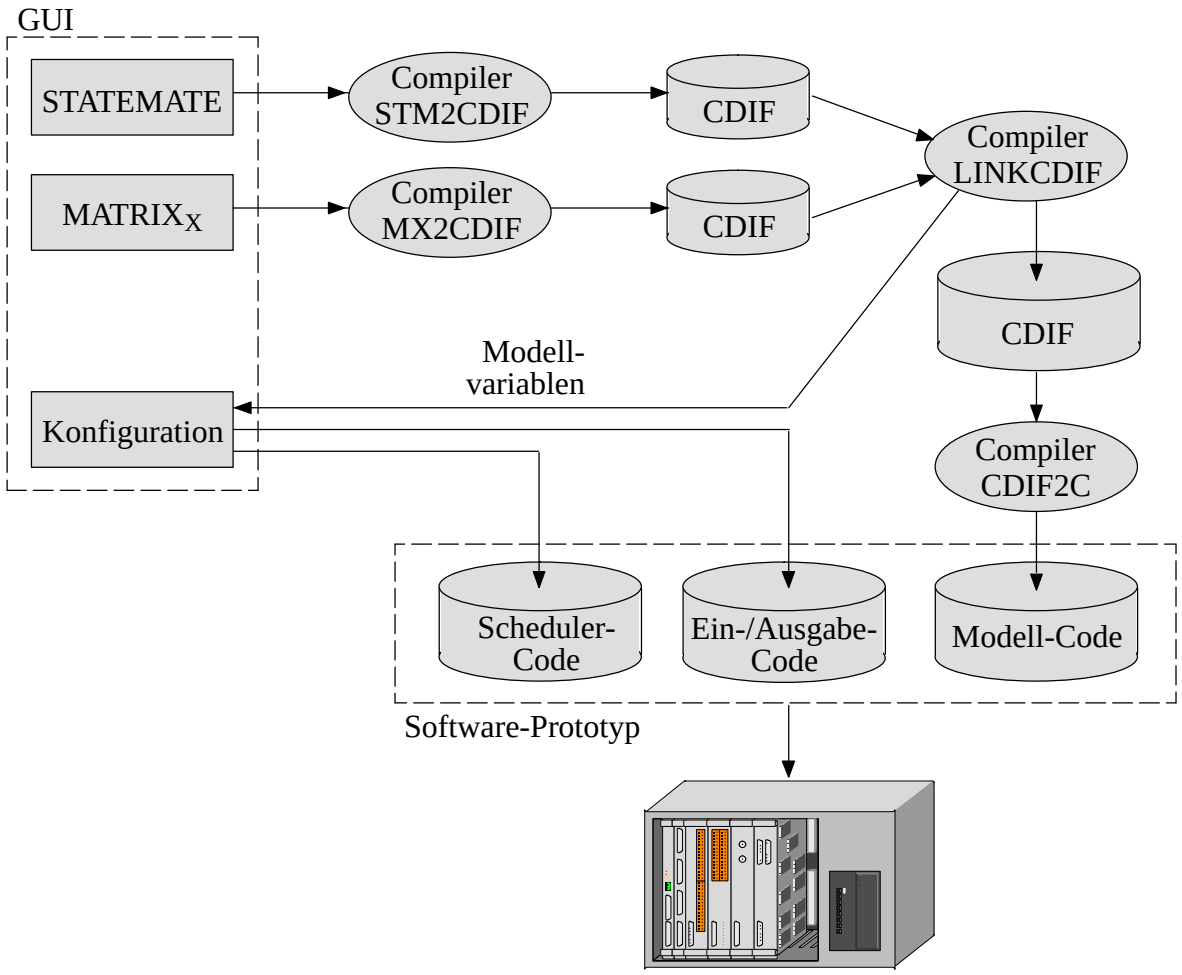


Bild 8-1: Aufbau der Codeerzeugung

ordnet und Einstellungen des Schedulers spezifiziert werden. Die drei Komponenten des Echtzeitcodes können dann als Software-Prototyp auf dem Rapid Prototyping Rechner ausgeführt werden. Bei der Ausführung werden die Funktionen eines Echtzeitbetriebssystems verwendet. Die drei Code-Komponenten werden im folgenden detailliert beschrieben.

8.2 Modell-Code

Für die automatische Erzeugung des Modell-Codes wird ein Compiler benutzt, der Daten einer Quellsprache in eine Zielsprache übersetzt. In dem vorliegenden Fall besteht die Quellsprache aus einer CDIF-Beschreibung, die in einen C-Code übersetzt wird. Die Überführung einer Quellsprache in eine Zielsprache läßt sich in zwei Teile untergliedern. Der erste Teil umfaßt die Analyse, bei der die Quellsprache zerlegt und in einer internen Datenstruktur abgelegt wird. Aus dieser Datenstruktur wird in einem zweiten Teil die Zielsprache erzeugt. Der zweite Teil wird als Synthese bezeichnet. Für beide Teile müssen mehrere Schritte durchgeführt werden, die in Bild 8-2 dargestellt sind.

Die lexikalische Analyse wird von einem *Scanner* durchgeführt. Hierbei wird die Quellsprache Zeichen für Zeichen eingelesen und anhand festgelegter Regeln zu Schlüsselwörtern, Operatoren, Zahlen, etc. zusammengefügt. Diese sogenannten *Tokens* werden für die syntaktische Analyse unter Verwendung einer Symboltabelle zur Verfügung gestellt. Jedes Token wird dazu mit einem eigenen

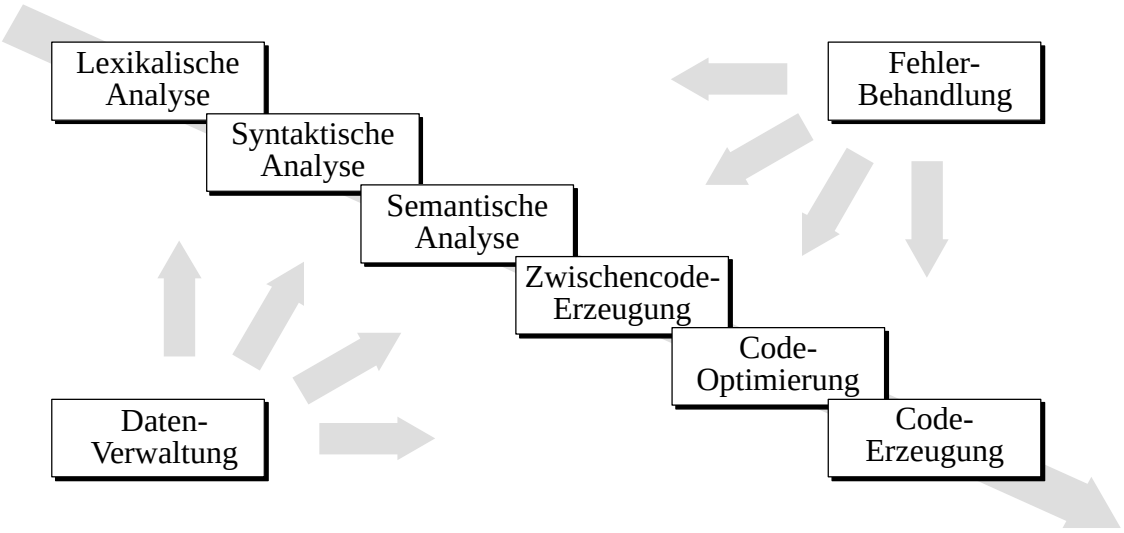


Bild 8-2: Struktur eines Compilers

Identifier versehen und in eine Symboltabelle eingetragen. Im weiteren Verlauf wird ausschließlich mit dem Identifier gearbeitet.

Mit Hilfe eines *Parsers* wird aufgrund der Art und Reihenfolge der Identifier festgestellt, welche Syntaxregel zutrifft und welches sprachliche Konstrukt somit gefunden wurde. Nach dieser syntaktischen Analyse wird eine Datenstruktur aufgebaut, die einem Abbild der Quellsprache entspricht. Die darauf folgende semantische Analyse untersucht den Inhalt der Datenstruktur im Hinblick auf seine Bedeutung. Mit den daraus gewonnenen Informationen kann ein Zwischencode erzeugt werden, der von Quell- und Zielsprache unabhängig ist. Da nach der semantischen Analyse alle notwendigen Informationen vorliegen, ist die Erzeugung eines Zwischencodes nicht zwingend notwendig. Allerdings erleichtert die Durchführung dieses Schritts eine Fehlersuche erheblich. Die Transformation der vorliegenden Informationen in eine optimierte Struktur wird zur Codeoptimierung durchgeführt. Nachdem die optimierte Struktur vorliegt, wird die Zielsprache ausgegeben.

Während der Durchführung aller Schritte wird die Datenverwaltung von einem eigenen Modul vorgenommen, das die Konsistenz der Daten überwacht. Ebenso erfolgt eine Fehlerbehandlung durch ein eigenes Modul.

In den nächsten beiden Teilkapiteln wird auf die Codeerzeugung für diskrete und kontinuierliche Modelle eingegangen. Gleichen sich Blöcke in ihrem Aufbau, werden sie gemeinsam behandelt. Zuerst wird bei jedem Block Aufbau und Funktionsweise erklärt, danach die Beschreibung des betreffenden Blocks in CDIF dargestellt und schließlich auf die Überführung des Blocks in C-Code eingegangen.

8.2.1 Echtzeitcode des diskreten Teilsystems

Um von einem diskreten Teilsystem (siehe Kapitel 2.1.1) Echtzeitcode erzeugen zu können, müssen die einzelnen Elemente der Modellierung in entsprechende Programmkonstrukte abgebildet werden. Für die Überführung von diskreten Modellen in Programmcode existieren verschiedene Strategien, die auf möglichst geringen Speicherbedarf, geringe Codegröße, hohe Ausführungsgeschwindigkeit oder einer Kombination aus den obigen Optimierungskriterien ausgerichtet sind.

Tabellarischer Ansatz

Wählt man als Optimierungskriterium einen möglichst geringen Speicherbedarf und Codegröße, so kann der *tabellarische Ansatz* nach Wietzke/Cochlovius [WiCo96] gewählt werden. Dieser Ansatz ist beispielsweise für Mikrocontroller-Anwendungen konzipiert, bei denen wegen der hohen Stückzahlen im Bereich der Unterhaltungselektronik die Kosten der Komponenten (z.B. Speicher) möglichst gering gehalten werden muß. Ausgehend von einer Modellierung mit Statecharts werden die Zustände in Zustandsübergangstabellen überführt, die von einem Interpreter abgearbeitet werden. Um einen Gewinn bei der Codegröße zu erreichen, wird der Interpreter in einigen Bereichen eingeschränkt. Die Einschränkungen umfassen beispielsweise den Ausschluß gleichzeitig auftretender Events sowie unbedingter Übergänge. Dies führt dazu, daß jeder Zustandsübergang genau von einem Event ausgelöst werden muß. Dadurch ist eine rein sequentielle Abarbeitung möglich und der Wettlauf von mehreren Events kann ausgeschlossen werden. Weitere Einschränkungen bestehen in der Beschränkung auf strukturell einfache Übergangsbedingungen, dem Ausschluß von History-Connectoren und Static Reactions.

Der Ansatz sieht für jedes Event eine Tabelle vor, in der alle für dieses Event möglichen Übergänge enthalten sind. Die Tabellen sind dabei in drei Spalten aufgeteilt (Bild 8-3). Die linke Spalte repräsentiert die Vergleichszustände, die aus Zustands- und Übergangsbedingungscodierung zusammengesetzt sind. Die mittlere Spalte gibt die Folgezustände wieder, die den gleichen Aufbau wie die Vergleichszustände aufweisen. Auf der rechten Seite stehen die Aktionen, die bei dem jeweiligen Übergang auszuführen sind. Die Zustandscodierung ist so aufgebaut, daß ein Zustand eine ’1’ oder eine ’0’ besitzt, je nachdem, ob der Zustand bei einem entsprechenden Übergang aktiv sein muß.

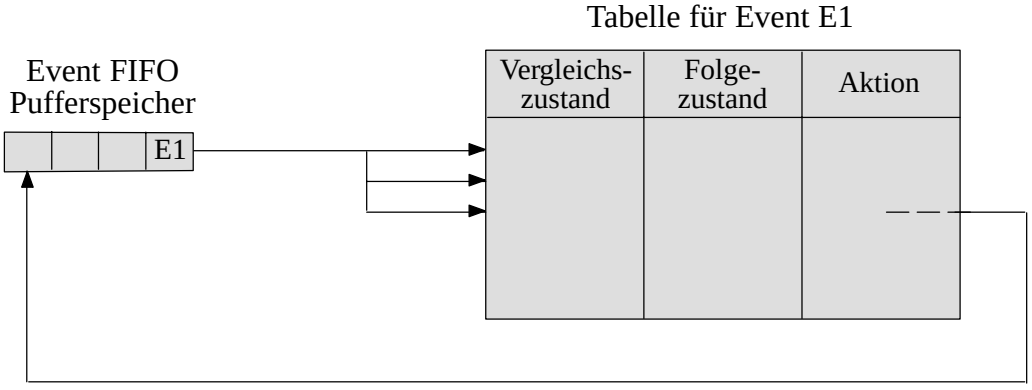


Bild 8-3: Aufbau des Interpreters beim tabellarischen Ansatz

Ausgelöste Events werden in einem Event FIFO Pufferspeicher eingetragen und sequentiell abgearbeitet. Dazu lädt der Interpreter die mit dem zu bearbeitenden Event verbundene Tabelle und vergleicht den aktuellen Systemzustand zeilenweise mit den Vergleichszuständen. Wird das erste Mal eine Übereinstimmung festgestellt, wird der Übergang ausgeführt. Danach wird das nächste Event aus dem FIFO Pufferspeicher abgearbeitet. Wird keine Übereinstimmung zwischen aktuellem Systemzustand und den Vergleichszuständen gefunden, verursacht das Event keinen Übergang und die Verarbeitung wird mit dem nächsten Event fortgesetzt.

Der Algorithmus, der dem tabellarischen Ansatz zugrunde liegt, kann also folgendermaßen dargestellt werden:

```
1     ComputeCurrentSystemState()
2     while (1) do
3       while (EventsInFIFO != 0) do
4         ComputeNumberOfRows()
5         while (NumberOfRows != 0) do
6           ComputeFIFOState()
7           if (CurrentSystemState == FIFOState) then
8             ComputeActions()
9             ComputeCurrentSystemState()
10            let NumberOfRows = 0
11          else
12            let NumberOfRows = NumberOfRows - 1
13          end if
14        end while
15        let EventsInFIFO = EventsInFIFO - 1
16      end while
17    end while
```

Um zu jedem Modellschritt jeweils nur einen Systemzustand verarbeiten zu müssen, werden alle Parallelitäten ausmultipliziert. Um unnötigen Zusatzaufwand zu vermeiden, werden nur diejenigen parallelen Zustände ausmultipliziert, die das gleiche Event in mehreren Zweigen aufweisen, da nur in diesem Fall zeitgleiche Übergänge stattfinden können.

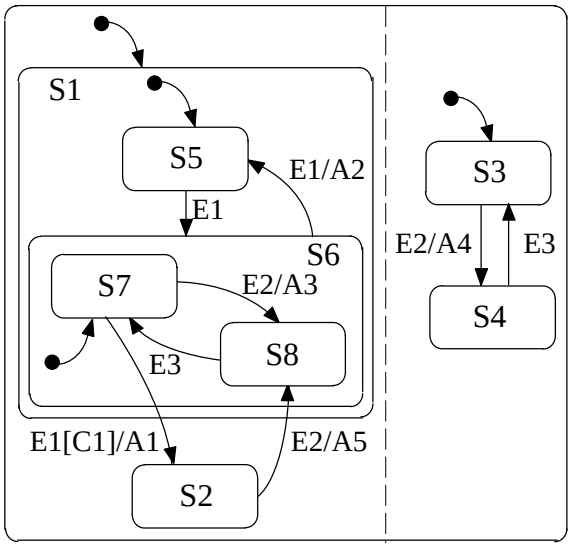


Tabelle für Event E1

Vergleichszustand					Folgezustand					Aktionen
S1	S5	S6	S7	C1	S1	S2	S5	S6	S7	
1	0	1	1	1	0	1	0	0	0	A1
1	1	0	0	0	1	0	0	1	1	-
1	0	1	0	0	1	0	1	0	0	A2

Tabelle für Event E2

Vergleichszustand					Folgezustand				Aktionen
S1	S2	S6	S7	S3	S1	S6	S8	S4	
1	0	1	1	1	1	1	1	1	A3&A4
1	0	1	1	0	1	1	1	0	A3
0	1	0	0	1	1	1	1	1	A4&A5
0	1	0	0	0	1	1	1	0	A5
0	0	0	0	1	0	0	0	1	A4

Tabelle für Event E3

Vergleichszustand				Folgezustand				Aktionen
S1	S6	S8	S4	S1	S6	S7	S3	
1	1	1	1	1	1	1	1	-
1	1	1	0	1	1	1	0	-
0	0	0	1	0	0	0	1	-

Bild 8-4: Aufbau der Tabellen für den tabellarischen Ansatz

Bild 8-4 zeigt ein Beispiel für den tabellarischen Ansatz. Die Priorität der Übergänge wird anhand der Reihenfolge der Einträge in die Tabelle festgelegt. Der Übergang von S7 nach S2 beispielsweise steht aufgrund seiner höheren Priorität in der Tabelle vor dem Übergang von S6 nach S5. Die Events

E2 und E3 werden in beiden Zweigen der Parallelität verwendet. Deswegen enthalten ihre Tabelleneinträge das Kreuzprodukt der relevanten, parallelen Zustände.

Für den Einsatz in einem Rapid Prototyping System eignet sich der tabellarische Ansatz weniger, da konzeptionelle Prototypen mit ausreichenden Ressourcen ausgestattet sind und meist hohe Anforderungen an das Zeitverhalten stellen. Erst für seriennahe Prototypen spielt der Ressourcenverbrauch eine wichtige Rolle. In diesem Bereich ist ein Einsatz des tabellarischen Ansatzes vorstellbar. Die Einschränkung des Funktionsumfangs ist zumindest in frühen Phasen nicht wünschenswert.

Bildung des Kreuzprodukts und Auflösung hierarchischer Komponenten

Wie bereits oben erwähnt, steht der Ressourcenverbrauch für Rapid Prototyping Systeme nicht im Vordergrund. Eine weitere Möglichkeit zur Codeerzeugung liegt in der Umwandlung der mit Hierarchie und Parallelität ausgestatteten Statecharts in flache Zustandsautomaten. Zur Entfernung werden alle parallelen Zustände miteinander multipliziert und die Übergangsbedingungen entsprechend angepaßt. Hierarchische Komponenten werden durch Erhöhung der Zustandsübergänge aufgelöst. Durch diese Transformation erhält man eine Darstellung mit nur einem aktiven Zustand und damit auch höchstens einem schaltbaren Zustandsübergang pro Modellschritt.

Der Vorteil dieses Ansatzes liegt darin, daß Kontrollen, die durch sich gegenseitig beeinflussende Zustandsübergänge (beispielsweise nach Bild 8-10) entstehen, nicht während der Codeabarbeitung ausgeführt werden. Allerdings steigt der Aufwand zur Identifikation des schaltenden Zustandsübergangs stark an, da von jedem Zustand i.a. mehrere Zustandsübergänge abgehen und komplexere Labels verwendet werden. Dies wirkt sich allerdings nicht wesentlich auf die Gesamtberechnungsdauer eines Modellschritts aus, so daß eine tatsächliche Beschleunigung im Ablaufverhalten erreicht wird. Auch die Abschätzung des ungünstigsten Zustandsübergangs (siehe Kapitel 7.3.1) wird vereinfacht, da die gesamte Abfrage in einem Konstrukt zusammengefaßt ist.

Der Ansatz weist jedoch auch erhebliche Nachteile auf. Insbesondere nimmt die Anzahl der Zustände bei Statecharts mit mehreren Parallelitäten exponentiell zu und führt somit zu einem starken Anwachsen des Speicherverbrauchs. Das Anwachsen kann bei großen Modellen so extrem ausfallen, daß eine Abarbeitung auch für die bei einem Rapid Prototyping System vorhandenen großen Ressourcen nicht mehr gewährleistet werden kann. Für ein Rapid Prototyping System ist es nicht wünschenswert, daß das erzeugte Modell nicht abbildbar ist. Zusätzlich nimmt die Lesbarkeit des erzeugten Programmcodes durch die fehlende Strukturierung ab und kann somit nicht für einen Übergang in realisierungsnähere Prototypen verwendet werden.

PROCORS

Von der Automobilindustrie wurde ebenfalls die Codeerzeugung aus einer Modellierung mit Statecharts untersucht. Beispielsweise wurde von der Firma BMW ein eigener Codegenerator entwickelt, der eine eigene Semantik für Statecharts benutzt und somit den Funktionsumfang der Statechart-Konstrukte einschränkt [Spre95] [Spre96]. Durch die Einschränkung wurde eine vereinfachte und somit schnellere Abarbeitung erreicht. Der Ansatz stellt eine Mischform aus einem Ansatz mit flachen Zustandsautomaten und einer direkten Abbildung der Statecharts dar.

Bei diesem Ansatz wird die Hierarchie des Modells aufgelöst, während parallele Zweige weiterhin getrennt voneinander abgearbeitet werden. Um eine gegenseitige Beeinflussung von parallelen Zweigen zu vermeiden, werden Transitionen nicht zugelassen, die Zustandsgrenzen von parallelen

Zweigen überschreiten. Somit können die parallelen Zweige sequentiell abgearbeitet werden. Der Ansatz zeichnet sich gegenüber flachen Zustandsautomaten durch die Verringerung der Zustandszahl und einer damit verbundenen Entflechtung der Übergangsbedingungen aus. Somit kann der Aufwand zur Identifikation des schaltenden Zustandsübergangs verringert werden und die Belegung von Speicherressourcen steigt nicht so stark wie bei flachen Zustandsautomaten an.

Eigener Ansatz mit direkter Abbildung

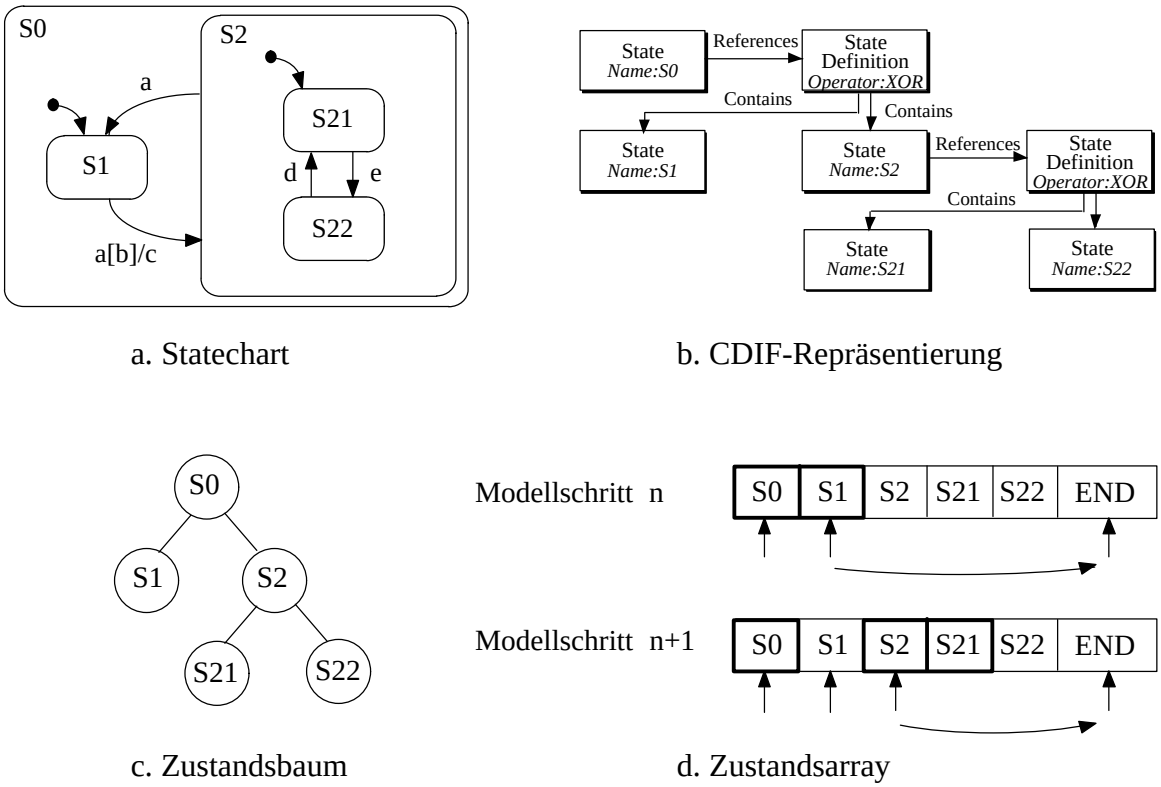
Einer der ersten Ansätze zur Codeerzeugung von Statecharts unter Beibehaltung der Struktur geht auf Harel und Gery zurück, die in [HaGe97] die C-Codeerzeugung untersuchten. Dieser Codegenerator wird in leicht veränderter Form mit dem CASE-Tool STATEMATE™ ausgeliefert. Auch die Firma ISI stellt mit der Codeerzeugung für das CASE-Tool BetterState™ eine Codeerzeugung unter Beibehaltung der Struktur zur Verfügung. Diese Ansätze stellen einen guten Ausgleich zwischen Speicherressourcen und Laufzeitverhalten dar. Zusätzlich ist der erzeugte Code gut lesbar. Die Codegeneratoren sind allerdings hauptsächlich auf die Beschleunigung der Simulation ausgerichtet und unterstützen daher auch keine Echtzeitanforderungen. Durch die häufige Benutzung von Bibliotheksfunktionen und mehrfache Kapselung von Funktionen werden allerdings keine optimalen Ergebnisse erzielt. Gery erweiterte den Codegenerator von STATEMATE™ für das Echtzeitbetriebssystem VRTX™ [Gery93]. Dazu wurden hauptsächlich Zeitbedingungen über das Echtzeitbetriebssystem verwaltet, jedoch die Struktur des Modellcodes nicht geändert.

Für die Codeerzeugung eines Rapid Prototyping Systems müssen mehrere Anforderungen aufgestellt werden. Diese umfassen kurze Modellschrittzeiten, effizienter Ressourcenverbrauch, vollständige Berücksichtigung aller Statechart-Konstrukte und Lesbarkeit des Modellcodes. Der eigene Ansatz, der diese Anforderungen berücksichtigt, wird im folgenden vorgestellt.

Ein zentrales Element der diskreten Modellierung stellen Zustände und deren Übergänge dar. Durch die jeweils aktiven Zustände der Modellierung ist festgelegt, welche Übergänge bei bestimmten Eingaben ausgeführt werden müssen. Aus diesem Grund wird im Code für jeden Zustand eine eigene Funktion verwendet. Diese Funktion wird als *Zustandsfunktion* bezeichnet. Die Übergänge werden innerhalb der Zustände in Form von if-then-else Konstrukten abgebildet.

Der Ausgangspunkt für die Codeerzeugung eines diskreten Teilsystems ist eine Datenstruktur, in der die Zustände in einer vorbestimmten Reihenfolge abgelegt werden. Ausgehend von der höchsten *StateDefinition* in CDIF wird die Struktur der Zustände untersucht. Dazu werden alle Elemente untersucht, die in der höchsten *StateDefinition* enthalten sind. Handelt es sich um einen Zustand, so wird dessen *StateDefinition* ermittelt und die Suche rekursiv fortgeführt. Jeder Zustand wird in dieser Reihenfolge in der Datenstruktur abgelegt. Sind alle Zustände durchsucht, ist eine Datenstruktur erzeugt worden, mit der sich die Funktionen der einzelnen Zustände aufbauen lassen. In der gleichen Art können globale und lokale Variablen erzeugt werden. Für die Codeerzeugung wird ein Zustandsbaum und daraus abgeleitet ein Zustandsarray wie folgt aufgebaut.

Ausgehend von einer Statechart-Modellierung (Bild 8-5a) wird eine Repräsentierung in CDIF (Bild 8-5b) erzeugt. In ihr besitzen alle Zustände auf einer Ebene dieselbe hierarchische Tiefe. Dies wird auch bei der Umwandlung in einen Zustandsbaum beibehalten (Bild 8-5c), der denselben hierarchischen Aufbau wie die CDIF Darstellung besitzt. Der höchste Zustand (im Beispiel der Zustand S0) stellt die Wurzel des Baums dar und wird als *Top-Level Zustand* bezeichnet. Ihm sind keine weiteren Zustände übergeordnet. Unterhalb dieses Zustands folgen seine untergeordneten Zustände. Ist der Zustand, der dem Top-Level Zustand folgt, ein Basic State (im Beispiel S1), so wird er nach dem



a. Statechart

b. CDIF-Repräsentierung

c. Zustandsbaum

d. Zustandsarray

Bild 8-5: Aufbau des Zustandsarrays

Top-Level Zustand in ein eindimensionales Zustandsarray aufgenommen (Bild 8-5d). Da ein Basic State keine weiteren Zustände beinhalten kann, endet an dieser Stelle die Untersuchung des Baums und die Suche wird beim nächsten, hierarchisch höchsten nicht untersuchten Element des Baums fortgesetzt. Wird ein hierarchischer Zustand erreicht (im Beispiel S2), so wird der Zustand in das Zustandsarray aufgenommen und alle weiteren untergeordneten Zustände (S21 und S22) untersucht. Bei parallelen Zweigen wird in derselben Art vorgegangen. Ist der gesamte Baum abgearbeitet, erhält man ein Zustandsarray, das die Struktur der Modellierung wiedergibt.

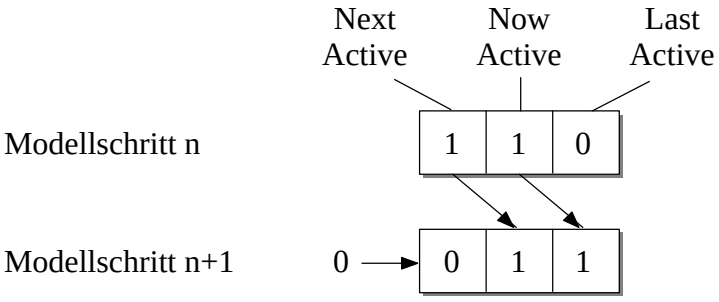


Bild 8-6: Aufbau der Zustandsbits

Jedem Zustand wird eine Zustandsfunktion und drei Zustandsbits (Bild 8-6) zugeordnet, die anzeigen, ob der Zustand im letzten Modellschritt aktiv war (*LastActive*), im aktuellen Modellschritt aktiv ist (*NowActive*) oder im nächsten Modellschritt aktiv sein wird (*NextActive*). Wird ein neuer Modellschritt begonnen, werden die Zustandsbits nach rechts geschoben und die Flag NextActive auf ’0’, d.h. nicht aktiv, zurückgesetzt. Ist ein Basic State aktiv, so werden auch alle hierarchisch übergeordneten Zustände aktiviert.

Die Ablaufsteuerung für die Ausführung eines Modellschritts besteht hauptsächlich aus einer for-Schleife (Zeile 1-4), die überprüft, ob die Flag eines Zustands innerhalb des Zustandsarrays aktiv ist. Ist das der Fall, so muß die Flag NowActive gesetzt sein und die zugehörige Zustandsfunktion wird ausgeführt.

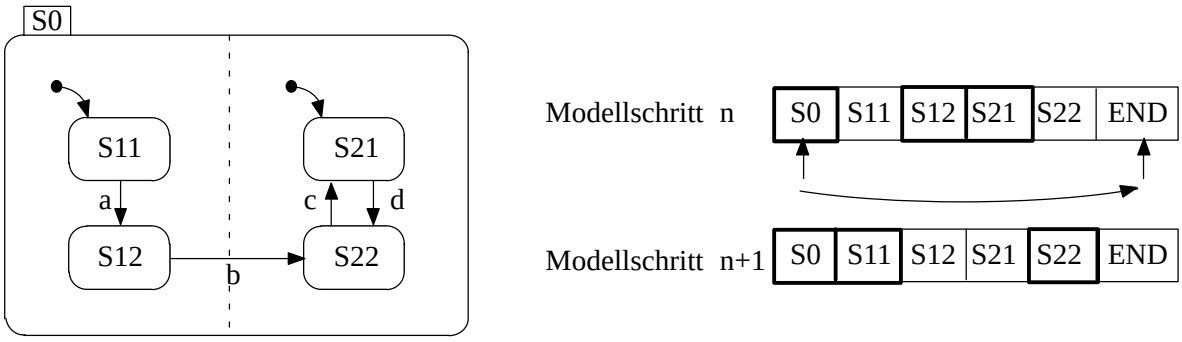
```
1    for (loop=1; loop<MaxNoOfStates; loop++) {
2      if (StateArray[loop].flags & NowActive)
3        (*StateArray[loop].state) ();
4      }
5    ModelStep++;
```

Wird während der Ausführung einer Zustandsfunktion (Zeile 3 der obigen for-Schleife) ein Zustandsübergang durchgeführt, werden die entsprechenden Aktionen durchgeführt (Zeile 5) und die nächsten aktiven Zustände durch Setzen der NextActive Flags bestimmt (Zeile 3-4). Die Zustandsfunktion, die dem Zustand S1 aus Bild 8-5a entspricht, ist im folgenden wiedergegeben. Zeile 2 zeigt die Überprüfung, ob das Event a aufgetreten ist und die Bedingung b erfüllt ist. Ist beides erfüllt, wird der Zustandsübergang durchgeführt.

```
1    void S1 (void) {
2      if ((a == ModelStep) && (b == TRUE)) {
3        StateArray[S21].flags |= NextActive;
4        StateArray[S2].flags |= NextActive;
5        c = ModelStep + 1;
6        loop += 3;
7      }
8    }
```

Bild 8-5d zeigt nicht nur den allgemeinen Aufbau eines Zustandsarrays, sondern zusätzlich die Abarbeitung eines Modellschritts. Gemäß der allgemeinen Ablaufsteuerung werden alle Zustände untersucht, ob ihre NowActive Flag gesetzt ist. Der Zustand S1 ist hervorgehoben, da er den einzigen aktiven Zustand darstellt, d.h. das Flag NowActive ist gesetzt. Als übergeordneter Zustand wird auch der Zustand S0 aktiviert. Wird die Zustandsfunktion von Zustand S1 ausgeführt, so kann es keine weiteren aktiven Zustände auf der gleichen Hierarchieebene oder darunter geben. Der Zeiger zur Abarbeitung des Zustandsarrays (die globale Variable loop) kann demnach auf das Ende des Zustandsarrays zeigen (Zeile 6). Durch die Ausführung des zur Zustandsfunktion von S1 gehörigen Programmcodes wurden die NextActive Flags der Zustände S21 und S2 gesetzt. Nach der Durchführung eines Modellschritts sind durch die Schiebeoperation die NowActive Flag beider Zustände gesetzt. Dies ist in Bild 8-5d unten dargestellt. Nach der Abarbeitung dieser Zustände muß keine Überprüfung des Zustands S22 mehr vorgenommen werden und der Zeiger zur Abarbeitung des Zustandsarrays kann auf das Ende zeigen.

Die Verarbeitung kann noch weiter beschleunigt werden, wenn hierarchische Zustände verlassen werden. Falls ein Zustandsübergang von einem übergeordneten Zustand ausgelöst wird, müssen die restlichen, hierarchisch tiefer liegenden Zustände nicht mehr ausgeführt werden. Im Beispiel von Bild 8-5 wäre das der Fall, wenn Zustand S21 aktiv ist und das Event a ausgelöst wird. Der Zeiger zur Abarbeitung des Zustandsarrays wird zu Beginn auf den Zustand S1 zeigen, dessen NowActive Flag nicht gesetzt ist. Danach wird der Zeiger auf den Zustand S2 zeigen, dessen NowActive Flag gesetzt ist. Somit wird die Zustandsfunktion von S2 ausgeführt und die NextActive Flag von Zustand S1 gesetzt werden. Der Zeiger kann direkt um die Anzahl der Zustände innerhalb des hierarchischen Zustands S2 erhöht werden, da die inneren Zustände nicht mehr ausgeführt werden müssen.



a. Statechart b. Zustandsarray

Bild 8-7: Aufbau eines Zustandsarrays für Statecharts mit parallelen Zweigen

Auch Zustände mit parallelen Zweigen können effizient ausgeführt werden. Bild 8-7a zeigt ein Statechart, das aus zwei parallelen Komponenten besteht. Beispielhaft sei angenommen, daß die Zustände S12 und S21 aktiv seien und das Event b im letzten Modellschritt ausgelöst wurde. Im aktuellen Modellschritt wird der Übergang von S12 nach S22 vorgenommen. Nachdem der Kontrollfluß den linken Teil des Statecharts verlassen hat, wird in diesem Teil über den Default-Connector der Zustand S11 eingenommen. Der erzeugte Modellcode wird den zugrunde liegenden Zustandsübergang von S12 nach S22 nicht als Teil des Zustandscodes von Zustand S12, sondern von Zustand S0 abarbeiten. Dies geschieht, da ein Zustandsübergang, der die Parallelitätslinie eines Zustands übertritt wie ein Zustandsübergang behandelt wird, der den Zustand verläßt und den anderen parallelen Zweig wieder betritt. Somit besitzt der Zustandscode von Zustand S0 den folgenden Aufbau:

```
1     void S0 (void) {
2       if (b == ModelStep) {
3         StateArray[S22].flags |= NextActive;
4         StateArray[S11].flags |= NextActive;
5         loop += 4;
6       }
7     }
```

Alle Zustände innerhalb von S0 werden nicht ausgeführt (Bild 8-7a) und damit wird der Zeiger zur Abarbeitung des Zustandsarrays auf das Ende des Arrays deuten.

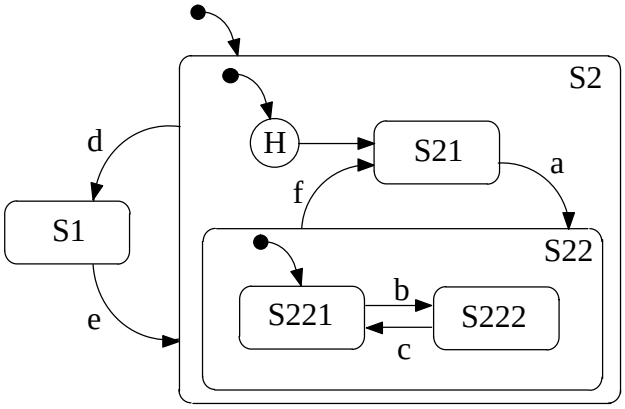


Bild 8-8: Statechart mit History-Connector

Der History-Connector eines Zustands speichert den letzten aktiven Unterzustand seiner Hierarchiestufe, wenn die Hierarchiestufe des Zustands verlassen wird. Das Beispiel in Bild 8-8 zeigt ein Statechart mit History-Connector. Der History-Connector wird bereits zu Beginn der Abarbeitung mit dem Wert S21 belegt (Zeile 1). Wird das Event d ausgelöst, so behält der History-Connector den Wert des Zustands, der gerade aktiv war. Dies bedeutet, daß jeder aktive Zustand innerhalb der Abarbeitung von Oberzustand S2 bei jedem Modellschritt seinen eigenen Wert als Variable HistoryState2 speichert. Im Beispiel entspricht dies den Zuständen S21 oder S22. Der Wert des History-Connectors wird benötigt, wenn ein Übergang von Zustand S1 nach S2 stattfindet. Im zugehörigen Modellcode des Zustands S1 wird im Übergangsfall (Event e wurde ausgelöst) der nächste Zustand dynamisch bestimmt (Zeile 5).

```
1    HistoryS2 = S21;
2    ...
3    void S1 (void) {
4      if (e == ModelStep) {
5        StateArray[HistoryS2].flags |= NextActive;
6      }
7    }
```

Die Aktion History Clear löscht den aktuell gespeicherten Wert des History-Connectors und setzt ihn auf seinen Default-Wert zurück (im Beispiel Zustand S21). Die zugehörige Programmzeile hat das folgende Aussehen:

```
HistoryS2 = S21;
```

Für die Abbildung von Deep History-Connectoren müssen Zustände auch auf mehreren Hierarchieebenen gespeichert werden. Hierfür wird für jede Hierarchieebene und Parallelität eine eigene Variable angelegt, die vom jeweils aktiven Zustand mit seinem eigenen Wert beschrieben wird.

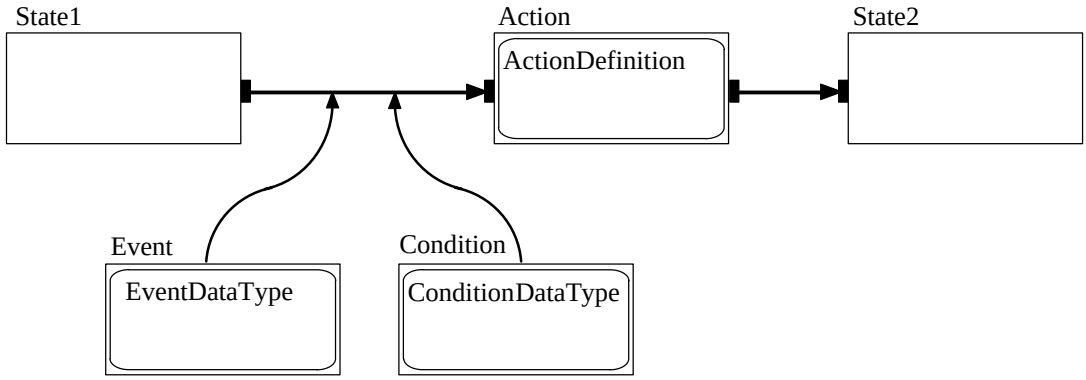


Bild 8-9: Modellierung eines Zustandswechsels

Zur Beschreibung eines Zustandsübergangs werden für jeden Zustand eigene Konstrukte erzeugt, in der die möglichen Übergänge beschrieben werden. Bild 8-9 zeigt die Modellierung eines allgemeinen Zustandsübergangs, der von einem Event und einer Condition ausgelöst wird und eine Action ausführt. Dabei können Event, Condition und Action auch weiter strukturiert sein, beispielsweise könnte ein Event ein Konstrukt (Event1 AND NOT Event2) enthalten.

Wichtig für die Auslösung eines Übergangs ist der Zeitpunkt, an dem ein Event erzeugt wird und der Wert, den eine Condition zu diesem Zeitpunkt besitzt. Ein Event wird im C-Code durch eine Integer-Variable dargestellt, deren Wert einem Zeitpunkt entspricht. Für die Abfrage eines Events wird diese Variable mit der Variablen ModelStep verglichen, der die Modellschritte zählt und somit ein

Maß für die Zeit darstellt. Das Konstrukt (Event1 AND NOT Event2) wird folgendermaßen abgebildet:

```
((pFixedData->E1 == ModelStep) && (!(pFixedData->E2 == ModelStep)))
```

Eine elementare Condition kann entweder den Wert TRUE oder den Wert FALSE beinhalten. Dies wird durch eine Variable repräsentiert, die den Wert ’0’ oder den Wert ’1’ enthält. Durch die Verwendung des Double Buffering Mechanismus (siehe Kapitel 7.1 und 8.2.3) werden die Daten in zwei Gruppen aufgeteilt. Diejenigen Daten, die für die Berechnungen während eines Modellschritts benutzt werden und sich nicht verändern, werden mit der Struktur pFixedData bezeichnet. Innerhalb der Struktur sind alle benötigten Variablen angelegt (in obigem Fall die Variablen E1 und E2). Diejenigen Daten, deren Werte innerhalb des Modellschritts geändert werden, werden mit der Struktur pDynamicData bezeichnet.

Conditions können ebenfalls durch den Vergleich von Variablen oder Konstanten beschrieben werden. Das Konstrukt (NOT Condition1 OR (X>(Y+5))) kann durch die folgenden Anweisungen abgebildet werden:

```
(( !(pFixedData->C1)) || (pFixedData->X > (pFixedData->Y + 5)))
```

Für jeden Zustand müssen alle Übergangsbedingungen ermittelt werden, die einen Zustandswechsel verursachen und nach Prioritäten geordnet werden. In einer if-Anweisung werden zuerst die Events und Conditions als Bedingung eingetragen und danach in den Rumpf der if-Anweisung die Aktion, die bei Eintreten der jeweiligen Bedingung ausgeführt werden muß. Eine wichtige Rolle für eine schnelle Verarbeitung kommt dabei der Anordnung der Übergangsbedingungen zu. Abfragen von Übergangsbedingungen mit einer hohen Wahrscheinlichkeit müssen möglichst eine hohe Priorität bekommen. Mit einer solchen Anordnung kann eine zeitraubende Abfrage von wenig wahrscheinlichen Übergangsbedingungen vor wahrscheinlichen Übergangsbedingungen vermieden und die Abarbeitung beschleunigt werden.

Aktionen setzen sich aus den Anweisungen, die an einen Zustandswechsel gebunden sind und den Anweisungen zur Aktivierung der Folgezustände zusammen. Nach Bild 8-9 lassen sich die Folgezustände durch Verfolgung der Transitionen bestimmen. Führt eine Transition in eine Aktion, so wird so lange die weiterführende Transition ermittelt, bis die Transition auf einen Zustand trifft. Dieser Zustand stellt den nächsten zu aktivierenden Zustand dar. Die Transition kann jedoch auch innerhalb des Zustands auf einen hierarchisch tiefer liegenden Zustand führen. In diesem Fall muß auch dieser Zustand aktiviert werden. Endet die Transition an einem hierarchischen Zustand, so muß innerhalb dieses Zustands der Default-Connector ermittelt werden und die von ihm abgehende Transition verfolgt werden. Zusätzlich muß beachtet werden, ob eine Transition in eine Parallelität führt und somit in den anderen Teilen der Parallelität zu einem Zustandswechsel führt. Dieser letzte Fall ist in Bild 8-10 dargestellt, bei dem durch Auslösung des Events a der Übergang von Zustand S1 auf Zustand S6 durchgeführt wird, der innerhalb des parallelen Zustands S2 liegt. Hierbei wird auch der linke Zweig der Parallelität aktiviert und ein Übergang in Zustand S3 durchgeführt.

Für das Beispiel von Bild 8-10 müssen also die drei Zustände S2, S6 und S3 im nächsten Schritt abgearbeitet werden. Die zugehörige Struktur im C-Code zur Aktivierung der Zustände hat das folgende Aussehen:

```
StateArray[S2].flags |= NextActive;
StateArray[S6].flags |= NextActive;
StateArray[S3].flags |= NextActive;
```

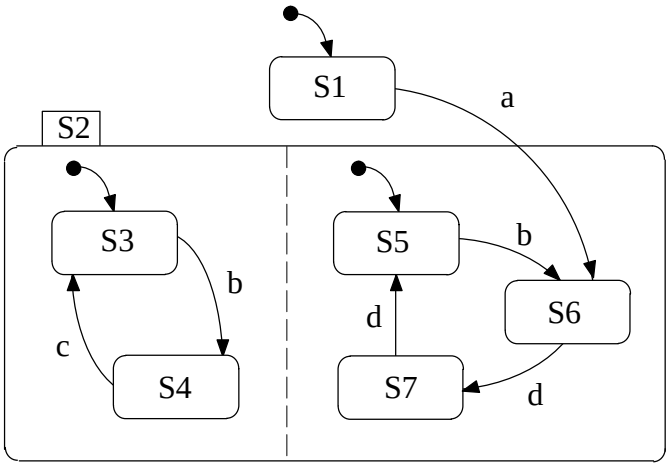


Bild 8-10: Aktivierung eines parallelen Zweigs

Die Anweisungen, die bei einem Zustandswechsel ausgeführt werden müssen, werden anhand einer *ActionDefinition* ermittelt, die den Inhalt der Aktionen wiedergibt. Dabei stehen unterschiedliche Aktionen zur Verfügung, die die Erzeugung von Events, das Setzen oder Löschen einer Condition, die Zuweisung von Werten an Variablen und das Löschen eines Zustandsspeichers umfassen. Wie bereits dargestellt, wird die Abfrage von Events über den Vergleich des Events mit einer Variablen ModelStep verglichen, der die Modellschritte zählt. Das Setzen von Events muß also in der folgenden Weise erfolgen:

```
pDynamicData->E1 = ModelStep + 1;
```

Beim nächsten Modellschritt wird die Variable ModelStep um Eins erhöht und nimmt somit den Wert an, der bereits im aktuellen Modellschritt der Variablen pDynamicData->E1 zugewiesen wurde. Eine Abfrage, ob beide Variablen den gleichen Wert aufweisen, liefert also den Nachweis, daß der Event für den aktiven Modellschritt erzeugt wurde. Für Conditions und Variablen werden die Wertzuweisungen direkt vorgenommen:

```
pDynamicData->C1 = 1;
pDynamicData->Flow1 = (pFixedData->X * (2 + pFixedDiscInt->V2));
```

Die einzelnen Elemente werden zu einer gemeinsamen Zustandsfunktion zusammengefaßt. Jede Transition wird von einer if-Anweisung eingefaßt, die alle Übergangsbedingungen und Anweisungen enthält. Diese if-Anweisungen können ineinander geschachtelt aufgebaut sein. Bei einem bedingungslosen Übergang wird die Erzeugung einer if-Anweisung nicht vorgenommen und die Anweisungen direkt ausgeführt. Für den Fall, daß keine Transition ausgelöst wurde, wird der aktive Zustand im nächsten Modellschritt erneut aktiviert.

Timeout-Events besitzen als Parameter ein Ereignis und eine Zahl N, die die gewünschte Verzögerung in Modellschritten vorgibt. Ausgelöst wird das Timeout-Event N Modellschritte nach dem letzten Auftreten des Ereignisses, an das es gekoppelt ist. Im Modellcode werden Timeout-Events wie gewöhnliche Ereignisse gesetzt. Allerdings wird das Ereignis nicht auf den Wert 'ModelStep + 1', sondern auf 'ModelStep + N + 1' gesetzt. Somit wird also eine zusätzliche Verzögerung addiert, um das Auslösen des Events zum korrekten Zeitpunkt zu gewährleisten. Timeout-Events müssen dann mit einem neuen Wert belegt werden, wenn das zugeordnete Ereignis neu erzeugt wird. Der alte Wert wird in jedem Fall überschrieben, da Timeout-Events nur das letztmalige Auftreten des zugeordneten Ereignisses berücksichtigen. Dies geschieht auch, wenn der alte Wert noch nicht ver-

arbeitet war. Für die Abfragen der Zustandsübergänge muß keine Unterscheidung zwischen gewöhnlichen Ereignissen und Timeout-Events vorgenommen werden, da die aktuellen Variablenwerte mit dem Modellschrittzähler verglichen werden.

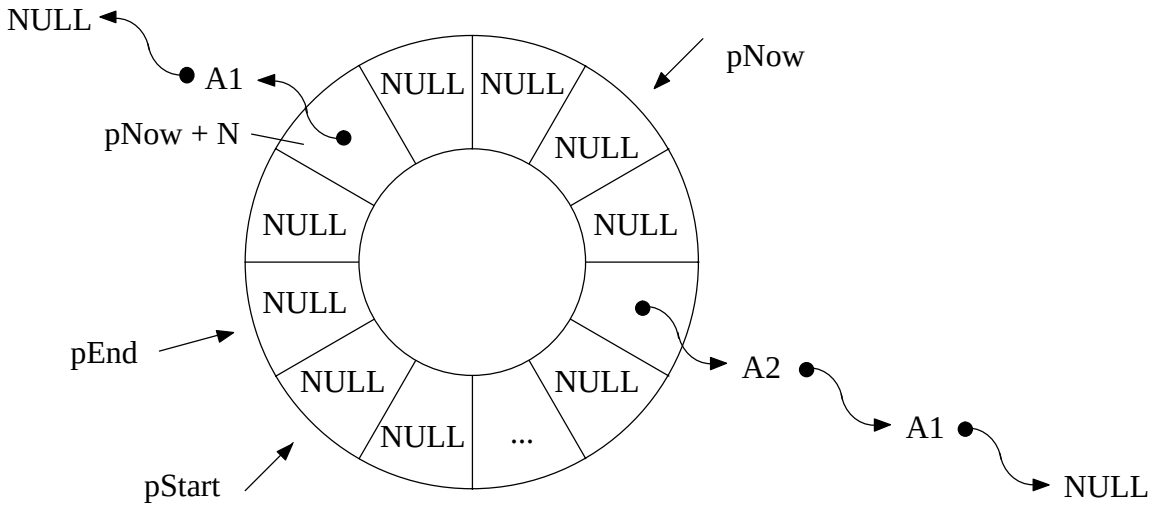


Bild 8-11: Aufbau eines Ringpuffers für Scheduled Actions

Scheduled Actions werden wie gewöhnliche Zustandsübergänge ausgelöst, besitzen allerdings eine bestimmte Verzögerung, die mit einer Anzahl von Modellschritten angegeben wird. Zur Abarbeitung von Scheduled Actions wird ein Ringpuffer verwendet, der sich aus einem Feld und zwei Zeigern zusammensetzt, die Anfang (pStart) und Ende (pEnd) des Feldes markieren (siehe Bild 8-11). Ein Zeiger pNow, der den aktuellen Modellschritt anzeigt, läuft schrittweise durch den Ringpuffer. Wird eine Scheduled Action ausgelöst, wird, ausgehend vom Zeiger pNow, N Felder weiter die zugehörige Aktion eingetragen.

In jedem Modellschritt wird geprüft, ob das Ringpufferfeld leer ist, auf das pNow aktuell zeigt. Ist das Feld leer (NULL), sind keine Scheduled Actions auszuführen. Andernfalls wird die Liste mit den Funktionszeigern abgearbeitet. Dabei wird jede Aktion ausgeführt, bis sämtliche Aktionen abgearbeitet sind und danach das Ringpufferelement gelöscht. Die Größe des Ringpufferelements wird nach der Verzögerung einer Scheduled Action festgelegt. Für jede Scheduled Action wird ein eigener Ringpuffer angelegt.

8.2.2 Echtzeitcode des kontinuierlichen Teilsystems

Der Ausgangspunkt für die Codeerzeugung eines kontinuierlichen Teilsystems ist die Darstellung in CDIF. Während in CDIF zustandsabhängige Funktionsblöcke bereits im Zustandsraum abgelegt sind (siehe Kapitel 6.5), besitzen statische Funktionsblöcke zumeist eigene Definitionen. Nicht für alle statischen Funktionsblöcke existieren jedoch eigene Definitionen, da sich einige statische Funktionsblöcke zu Gruppen zusammenschließen lassen. Die folgenden Teilkapitel geben den Aufbau des Codes für algebraische, trigonometrische, exponentielle und logarithmische Blöcken, Potenzblöcke, Interpolationsblöcke, stückweise lineare Blöcke und Signalgeneratoren wieder. Danach wird auf die Codeerzeugung für dynamische Blöcke eingegangen [Schn98].

Abbildung algebraischer Blöcke

Unter algebraischen Elementen versteht man bei der Modellierung kontinuierlicher Systeme beispielsweise die Produktbildung, Division oder Summation. Am Beispiel der Produktbildung (im

CASE-Werkzeug MATRIX$_X$™ mit 'Element by Element Product Block' bezeichnet) soll die Abbildung algebraischer Blöcke verdeutlicht werden. Bild 8-12 zeigt einen Product Block mit einem und zwei Ausgängen in der Darstellung des CASE-Tools MATRIX$_X$™.

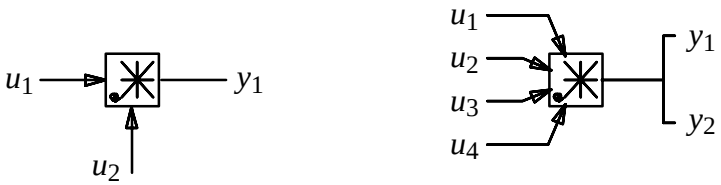


Bild 8-12: Product Block mit einem und zwei Ausgängen

Der Product Block multipliziert die Eingangssignale und stellt das Ergebnis am Ausgang zur Verfügung. Die Zahl der Ausgänge ist nicht fest vorgegeben und kann einen beliebigen Wert annehmen. Die Zahl der Eingänge wird an die Zahl der Ausgänge automatisch angepaßt und beträgt stets das Doppelte der Ausgangssignale. Existieren n Multiplikationen, so werden dem k-ten Ausgang die Eingänge k und $k + n$ zugeordnet. Allgemein ergibt sich daraus:

$$y_k = u_k \cdot u_{k+n}; \quad 0 \le k \le n, |y| = n, u \in \mathbb{R}, k \in \mathbb{N} \qquad (8.1)$$

Für die Abbildung des Blocks in CDIF werden Elemente aus der CACSD Subject Area, Expression Subject Area und der Data Flow Subject Area benötigt. Der Aufbau des Product Blocks in CDIF ist in Bild 8-13 wiedergegeben. Auf den genauen Aufbau einer algebraischen CDIF Beschreibung wurde bereits in Kapitel 6.5 eingegangen. Um mehr als die obigen zwei Eingänge und einen Ausgang beschreiben zu können, wird die *MultiplicationExpressionDefinition* mehrfach referenziert (in Bild 8-14 sind die Ausgangsvariablen betrachtet).

Den äußeren Rahmen des Product Blocks bildet eine *StaticCACSDProcessDefinition*, die in der CACSD Subject Area definiert ist. Die algebraischen Zusammenhänge der Multiplikation werden mittels einer *Expression* und einer *MultiplicationExpressionDefinition* im Inneren der StaticCACSDExpressionDefinition abgelegt (siehe Bild 8-13).

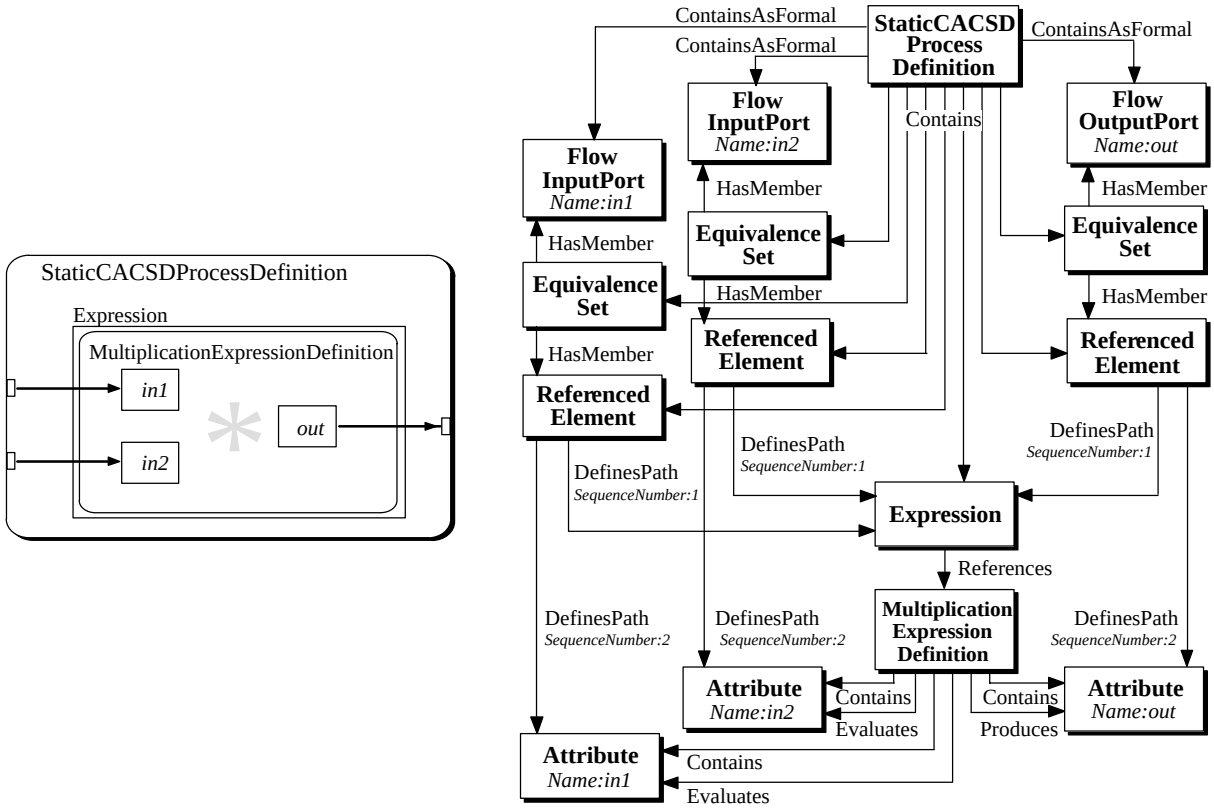


Bild 8-13: Aufbau eines Product Blocks mit einem Ausgang in CDIF

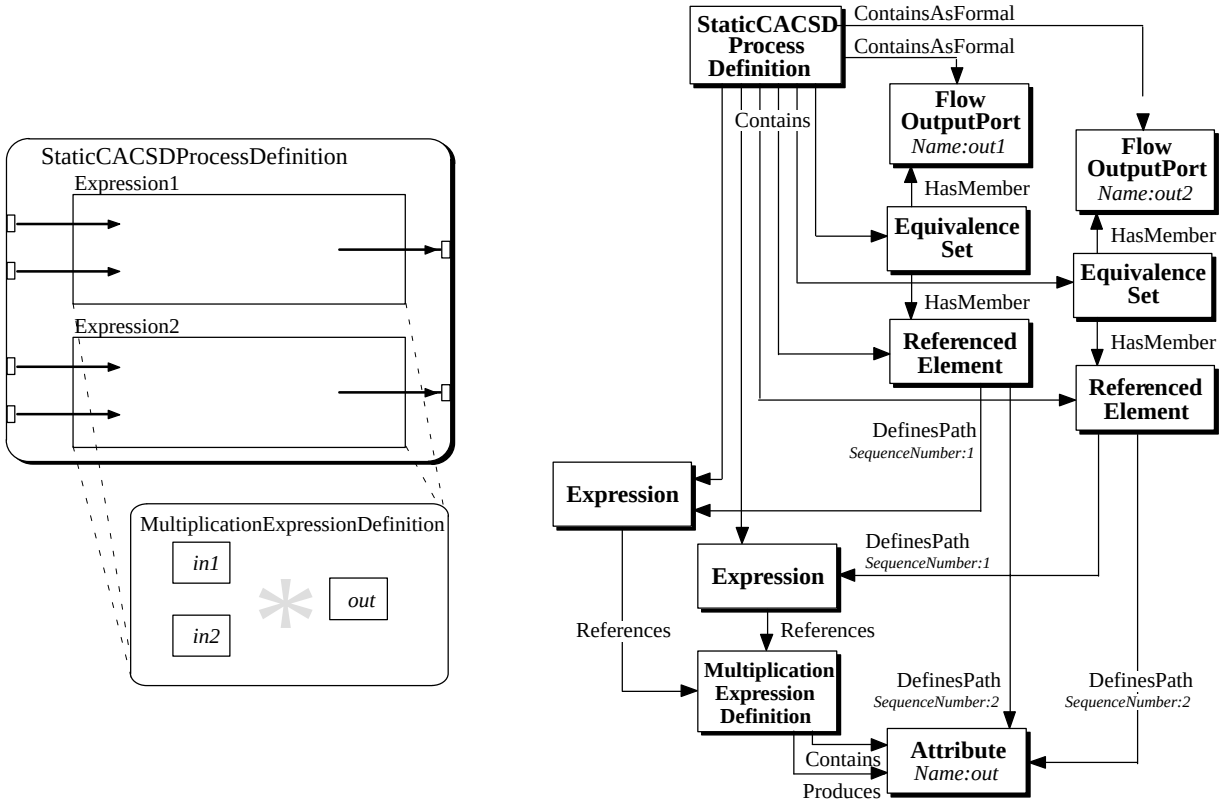


Bild 8-14: Aufbau eines Product Blocks mit zwei Ausgängen in CDIF

Bei zwei Multiplikationen wird für jede Multiplikation eine eigene *Expression* erzeugt. Der Aufbau dieser *Expressions* ist bis auf die unterschiedlichen Ein- und Ausgangsvariablen identisch und somit können die *Expressions* auf dieselbe *MultiplicationExpressionDefinition* zeigen. Um eine eindeutige Zuordnung zwischen einem Ausgang des Product Blocks, einem Ausgang des *StaticCACSDProcessDefinition* und einer *Expression* herstellen zu können, werden die Elemente *ReferencedElement* und *EquivalenceSet* benötigt. Für jede Multiplikation wird eine Entity *ReferencedElement* und zwei Relationen *DefinesPath* eingefügt. Die erste Relation verweist von *ReferencedElement* auf *Expression* und legt fest, für welche Multiplikation ein Objekt angesprochen wird. Durch die zweite Relation *DefinesPath* wird eine Entity *Attribute* mit dem *ReferencedElement* verbunden. Um die Reihenfolge der Relationen beschreiben zu können, wird jeder dieser Relationen ein Attribut *SequenceNumber* mit einer entsprechenden Ordnungszahl zugewiesen. Die erste Relation zwischen *ReferencedElement* und *Expression* hat die *SequenceNumber* mit dem Wert 1, die Relation zwischen *ReferencedElement* und *Attribute* mit dem Wert 2. Das *ReferencedElement* wird also jeder Multiplikation eineindeutig zugeordnet und kann über die Entity *EquivalenceSet* und zwei Relationen *HasMember* mit dem entsprechenden Ausgang, bzw. *FlowOutPort* verbunden werden. Die Eingänge des Product Blocks werden über *FlowInputPorts* aus der Data Modeling Subject Area beschrieben.

Die *MultiplicationExpressionDefinition* ermöglicht die Beschreibung von Multiplikationen in dem folgenden Format:

$$y = \prod_{i=1}^{n} u_i^{c_i} \qquad (8.2)$$

Für die Abbildung eines Produkt Blocks wird $c_i = 1$ gesetzt. Da bei einem MATRIX$_X$™ Product Block die Anzahl der Faktoren auf zwei beschränkt ist, werden für die Abbildung in CDIF zwei Entities *Attribute* und zwei Relationen *Evaluates* benötigt.

Die grafische CDIF-Darstellung kann mittels Syntax.1 und Encoding.1 (siehe Kapitel 6.3.4) auch im CDIF-Transferformat dargestellt werden:

```
( MultiplicationExpressionDefinition MED300 )
( Attribute AT301 )
( Attribute AT302 )
( Attribute AT303 )
( DefinitionObject.Contains.ComponentObject  REL304 MED300 AT301 )
( DefinitionObject.Contains.ComponentObject  REL305 MED300 AT302 )
( DefinitionObject.Contains.ComponentObject  REL306 MED300 AT303 )
( AlgebraicExpressionDefinition.Produces.Attribute  REL307 MED300 AT301)
( AlgebraicExpressionDefinition.Evaluates.Attribute REL308 MED300 AT302
        ( CoefficientValue  #D1))
( AlgebraicExpressionDefinition.Evaluates.Attribute REL309 MED300 AT302
        ( CoefficientValue  #D1))
```

Der Aufbau des C-Codes ist für algebraische Blöcke vergleichsweise einfach und besitzt das folgende Aussehen:

```
1    void calc_y( )
2    {
3       pDynamicData->OutMul = pFixedData->InMul1 * pFixedData->InMul2;
4    }
```

Die Variablen pDynamicData–>OutMul, pFixedData–>InMul1 und pFixedData–>InMul2 stellen dabei die Aus- und Eingänge des Product Blocks dar.

Abbildung trigonometrischer Blöcke

Trigonometrische Blöcke wie Sinus oder Cosinus Blöcke (Bild 8-15 zeigt die beiden Blöcke in der Darstellung des CASE-Werkzeugs MATRIX$_X$™) besitzen in der Grundeinstellung einen Eingang und einen Ausgang. Wie bei den algebraischen Blöcken kann die Anzahl der Ausgänge beliebig gewählt werden. Für jeden Ausgang wird von MATRIX$_X$™ automatisch ein Eingang erzeugt.

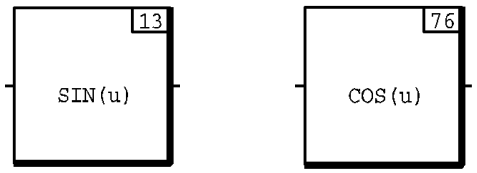


Bild 8-15: Darstellung der Sinus und Cosinus Blöcke von MATRIX$_X$™

Ein trigonometrischer Block mit *n* Eingängen und Ausgängen repräsentiert also *n* voneinander unabhängige trigonometrische Funktionen. Die Werte der Eingänge dürfen dabei beliebige reelle Zahlen sein und werden von MATRIX$_X$ automatisch einer Modulo 2π Division unterzogen und mit 2π multipliziert. Daraus ergeben sich für einen Sinus Block die folgende Übertragungsgleichung:

$$y_k = \sin\big((u_k \bmod 2\pi) \cdot 2\pi\big); \quad u_k \in \mathbb{R},\ k \in \mathbb{N} \qquad (8.3)$$

Da bei den trigonometrischen Blöcken der Ausgang direkt von den Eingängen abhängig ist, werden diese Blöcke mit einer *StaticCACSDProcessDefinition* beschrieben. CDIF besitzt in der Expression Subject Area zur Darstellung trigonometrischer Funktionen eine Entity *TrigonometricExpression-Definition*. Mit Hilfe dieser Entity können die folgenden trigonometrischen Funktionen dargestellt werden: Sinus, Cosinus, Tangens, Arcussinus, Arcuscosinus und Arcustangens. Die zu repräsentierende Funktion wird über ein Attribut *Typ* festgelegt, dessen Wert den obigen sechs Möglichkeiten entspricht. Für jede trigonometrische Funktion werden zwei Entities *Attribute* benötigt, die den Eingang und den Ausgang der Funktion repräsentieren. Die *TrigonometricExpressionDefinition* verweist auf die Entity, die dem Eingang entspricht, mit der Relation *Evaluates*, und auf die Entity, die dem Ausgang entspricht, mit der Relation *Produces*. Der weitere Aufbau der CDIF Repräsentation erfolgt ähnlich dem Product Block. Bild 8-16 zeigt die Darstellung eines Sinus Blocks unter CDIF.

Wird bei der Codeerzeugung eine *TrigonometricExpressionDefinition* erkannt, so wird die entsprechende Funktion durch die in der Standard “math.h” Bibliothek abgelegten trigonometrischen C-Funktionen repräsentiert. Der Modell-Code eines Sinus Blocks hat somit folgendes Aussehen:

```
1       void calc_y( )
2       {
3          pDynamicData->OutSin = sin(pFixedData->InSin);
4       }
```

Bei der Abbildung der Arcussinus und Arcuscosinus Blöcke ist zusätzlich die Beschränkung des Wertebereichs auf $(-1,\ 1)$ zu beachten. Liegen die Eingangswerte außerhalb dieses Bereichs muß eine definierte Fehlerbehandlung durchgeführt werden. Dies ist insbesodere im Bereich Rapid Prototyping von Wichtigkeit, um einen unkontrollierten Absturz des ausgeführten Programms zu vermeiden. Eine Implementierung der Arcussinus und Arcuscosinus Funktionen muß also um diese Fehlerbehandlung erweitert werden:

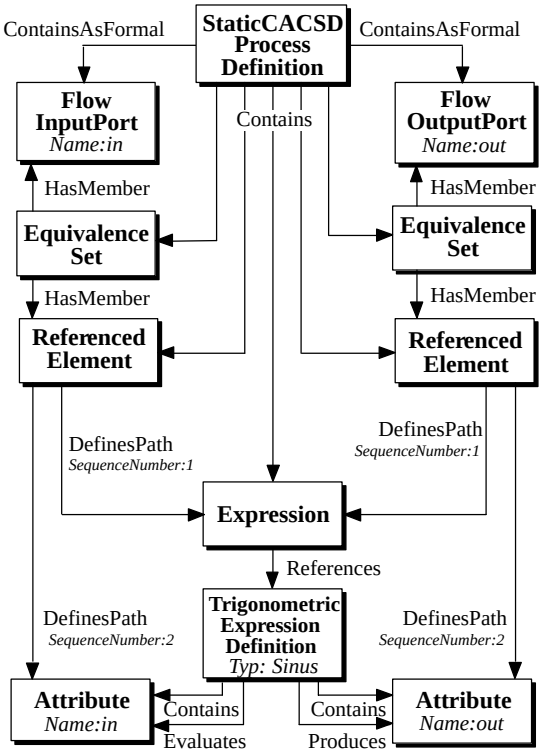


Bild 8-16: Abbildung eines Sinus Blocks in CDIF

```
1     void calc_y( )
2     {
3        pDynamicData->OutASin = m_asin(pFixedData->InASin);
4     }


5     float m_asin(float x)
6     {
7     float back;
8
9     if ((x>=-1) && (x<=1))
10       back = asin(x);
11    else
12       {
13       /* Fehlerbehandlung */
14       back = 0.0;
15       }
16    }
```

Der Aufruf der Arcussinus Funktion geschieht über eine eigengeschriebene Funktion *m_asin()*, die bei der Berechnung des Funktionswertes zuerst überprüft, ob der übergebene Eingangswert im erlaubten Zahlenbereich liegt. Ist dies der Fall, so wird die Berechnung über die in der Standard Bibliothek vorkommende Funktion *asin()* durchgeführt. Liegt der Eingangswert nicht im erlaubten Zahlenbereich, so muß eine Fehlerbehandlung durchgeführt werden. Diese Fehlerbehandlung kann je nach Anwendungsfall unterschiedlich ausfallen. Es ist beispielsweise denkbar, daß ein Fail Safe Zustand eingenommen wird, der die Anwendung in einen zuvor definierten ungefährlichen Zustand versetzt. Bei manchen Anwendungsfällen ist dies nicht möglich, da kein sicherer Zustand existiert (z.B. in Flugzeugen). Im obigen Fall wird als Ergebnis eines fehlerhaften Aufrufs der Wert 0.0 zurückgegeben.

Der Arcustangens Block unterscheidet sich in MATRIX$_X$™ von anderen trigonometrischen Blökken durch die Zahl der Eingänge. Jeder Ausgang besitzt zwei Eingänge, die zuerst dividiert werden. Das Zwischenergebnis wird dann als Eingangswert des Arcustangens verwendet. Bei diesem Block dürfen die Eingangswerte der beiden Zahlen jede reelle Zahl annehmen, wobei die Verwendung der Null im Nenner nicht zulässig ist. Die Übertragungsgleichung des Arcustangens Blocks kann also folgendermaßen angegeben werden:

$$y_k = \operatorname{atan}\left(\frac{u_k}{u_{k+n}}\right);\quad \begin{matrix} u_k \in \mathbb{R} \quad \textit{für}\ 1 \le k \le n\,; \\ u_k \in \mathbb{R}\backslash\{0\} \quad \textit{für}\ n \le k \le 2{\cdot}n \end{matrix} \quad ; k \in \mathbb{N} \tag{8.4}$$

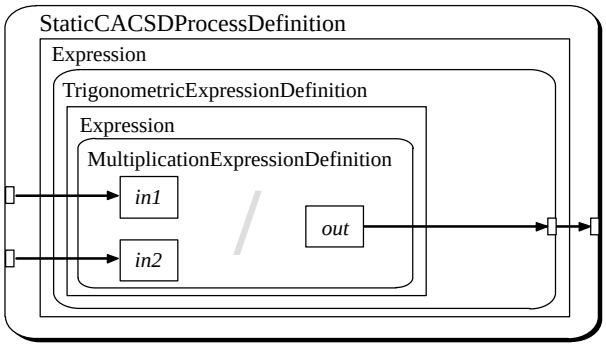


Bild 8-17: Struktureller Aufbau des Arcustangens Blocks in CDIF

In CDIF gleicht der äußere Aufbau dem Aufbau der übrigen trigonometrischen Blöcke. Zusätzlich muß jedoch der mathematische Zusammenhang des Blocks durch zwei ineinander geschachtelte Expressions abgebildet werden (siehe Bild 8-17). Die äußere Expression verweist zur Darstellung der trigonometrischen Arcustangens Funktion auf eine Entity *TrigonometricExpressionDefinition.* Die innere Expression, die das Argument des Arcustangens bildet, wird durch eine *MultiplicationExpressionDefinition* dargestellt. Der Eingänge des Arcustangens Blocks verweisen über die Relationen *EquivalenceSet, HasMember, ReferencedElement* und *DefinesPath* auf die *Attributes* der *MultiplicationExpressionDefininition* (siehe Bild 8-18). Diese sind über die Relation *Evaluates* mit der *MultiplicationExpressionDefinition* verbunden. Durch die Wahl der *CoefficientValues* zu 1 und –1 erfolgt eine Division der beiden Eingänge. Das Ergebnis wird dann in einer dritten Entity *Attribute* abgelegt. Die Verknüpfung des Ergebnisses mit dem Eingang der Arcustangens Funktion, d.h. der *TrigonometricExpressionDefinition* mit dem Typ Arcustangens, erfolgt wie bei den Eingängen über die Relationen *EquivalenceSet, HasMember, ReferencedElement* und *DefinesPath*. Das Ergebnis der *TrigonometricExpressionDefinition* wird ebenfalls über diese Elemente dem Ausgang des Arcustangens Blocks zugeordnet.

Der Aufbau des C-Codes für den Arcustangens Block erfolgt durch den Aufruf der Standard “math.h” Bibliothek für die Arcustangens Funktion. Die zugehörige Division wird durch den Aufruf der Funktion *mpow()* vorgenommen. In CDIF wird eine Division als inverse Multiplikation dargestellt, um eine möglichst breite Anwendung des Blocks zu ermöglichen. Dieses Konzept wurde auch als Grundlage bei der Abbildung in den C-Code übernommen.

```
1    void calc_y( )
2    {
3        pDynamicData->OutAtan=atan(pFixedData->InAtan1*m_pow(pFixedData->InAtan2,-1);
4    }
```

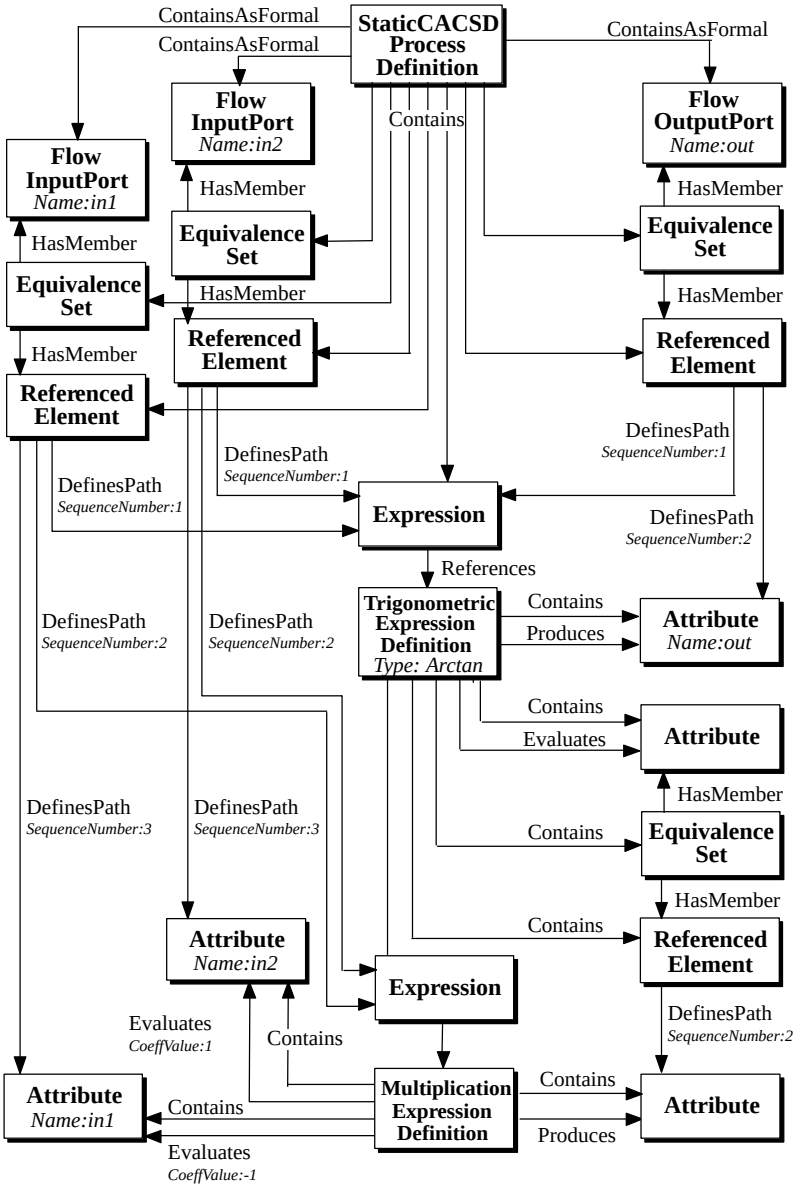


Bild 8-18: Abbildung eines Arcustangens Blocks in CDIF

```
 5    float m_pow(float x, float z)
 6      {
 7      double y, v;
 8      float back;
 9      back = 0.0;
10      intern = x;
11
12      if ((x == 0) && (z <= 0))
13        {
14        /* Fehlerbehandlung */
15        back = 0.0;
16        }
17      else
18        {
19        if ((x<0) && (modf(v,&y) <= 0.0))
20          {
21          /* Fehlerbehandlung */
22          back = 0.0;
23        }
```

```
24            else back = pow(x,z);
25          }
26          return back;
27      }
```

Abbildung von Potenzblöcken

Bild 8-19 zeigt den Quadratwurzel Block (engl. *squareroot*) in der Darstellung des CASE-Werkzeugs MATRIX$_X$™. Dieser Block berechnet die Quadratwurzel seines Eingangs und stellt das Ergebnis an seinem Ausgang zur Verfügung.

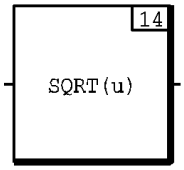


Bild 8-19: Darstellung des Squareroot Blocks von MATRIX$_X$

Der Wertebereich des Eingangs ist bei diesem Block auf positive, reelle Werte beschränkt. Eingangswerte kleiner als Null müssen gesondert als Fehlerfall behandelt werden. Wie bei algebraischen und trigonometrischen Blöcken kann die Anzahl der Ausgänge beliebig hoch gewählt werden. Die Anzahl der Eingänge wird automatisch an die Zahl der Ausgänge angepaßt. Ein Block mit *n* Ein- und Ausgängen repräsentiert *n* voneinander unabhängige Quadratwurzelfunktionen. Für die Übertragungsgleichung eines Quadratwurzel Blocks mit *n* Ein- und Ausgängen ergibt sich die folgende Gleichung:

$$y_k = \sqrt{u_k} = u_k^{\frac{1}{2}}; \quad u_k \geq 0\,,\ u_k \in \mathbb{R}\,,\ k \in \mathbb{N}\,,\ k \leq n \qquad (8.5)$$

Auch der Quadratwurzel Block ist ausschließlich von den Eingängen des Blocks ohne innere Zustände abhängig und wird deswegen innerhalb einer *StaticCACSDProcessDefinition* aufgebaut. Wie bei den Blöcken zuvor werden auch bei diesem Block die Ein- und Ausgänge durch *FlowInPorts* und *FlowOutPorts* dargestellt. Die Quadratwurzel wird als Potenz mit dem Exponenten $\frac{1}{2}$ dargestellt und kann deswegen nach Gleichung (8.2) durch eine *MultiplicationExpressionDefinition* abgebildet werden. Der Exponent wird dabei als Attribut der Relation *Evaluates* angegeben. Bild 8-20 zeigt die Abbildung eines Quadratwurzel Blocks in CDIF mit einem Ein- und Ausgang.

Der Aufbau des C-Codes für den Quadratwurzel Block erfolgt durch den Aufruf der bereits eingeführten Funktion *m_pow()*. Als Argument wird der Wert 0.5 übergeben, der der Bildung der Wurzelfunktion entspricht.

```
1     void calc_y( )
2     {
3        pDynamicData->OutSqrRoot = m_pow(pFixedData->InSqrRoot,0.5);
4     }
```

Der MATRIX$_X$™ Block UPowerConst (Bild 8-21) dient der Beschreibung von Potenzen mit konstantem Exponenten. Die Basis des Exponenten wird über den Eingang des Blocks dargestellt. Der frei wählbare, konstante Exponent der Potenz wird als Parameter eingetragen. Der Block besitzt in seiner Grundeinstellung einen Eingang und einen Ausgang. Die Anzahl kann jedoch auf *n* Ausgänge und *n* Eingänge erhöht werden. Ein solcher Block stellt *n* voneinander unabhängige Potenzen dar.

Der UPowerConst Block besitzt die folgende Übertragungsgleichung:

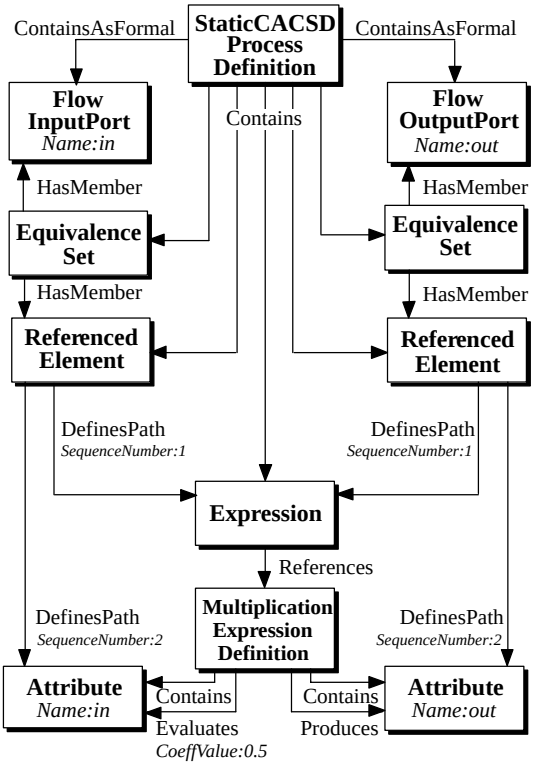


Bild 8-20: Abbildung eines Quadratwurzel Blocks in CDIF

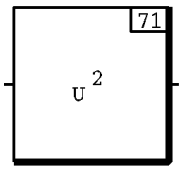


Bild 8-21: Darstellung des UPowerConst Blocks in MATRIX$_X$™

$$y_k = u_k^{c_k}; \quad k \le n,\ k \in \mathbb{N} \qquad \begin{array}{ll} \text{und} & u_k \in \mathbb{R}_+;\ c_k \in \mathbb{R} \\ \text{oder} & u_k = 0;\ c_k \in \mathbb{R}_+ \\ \text{oder} & u_k \in \mathbb{R}_-;\ c_k \in \mathbb{N} \end{array} \qquad (8.6)$$

Der Aufbau des UPowerConst Blocks in CDIF ähnelt dem Aufbau der Product und Quadratwurzel Blöcken, da auch beim UPowerConst Block der mathematische Zusammenhang mit den Entities *Expression* und *MultiplicationExpressionDefinition* hergestellt wird. Ein wichtiger Unterschied ergibt sich bei der Darstellung eines Blocks mit mehreren Ausgängen. Während bei der Darstellung der Product und Quadratwurzelblöcke mit *n* Ausgängen *n Expressions* verwendet werden können, die auf dieselbe *MultiplicationExpressionDefinition* verweisen, ist dies bei der Darstellung eines UPowerConst Blocks nur bedingt möglich. Der Exponent einer Potenz ist in CDIF durch Verwendung des Attributs *CoeffValue* ein fester Bestandteil der *MultiplicationExpressionDefinition*. Somit muß für jede Abbildung eines UPowerConst Blocks für jeden neuen Wert des Exponenten eine eigene *MultiplicationExpressionDefinition* benutzt werden. Dieselbe *MultiplicationExpressionDefinition* kann für diejenigen *Expressions* benutzt werden, deren Potenzen denselben Exponenten aufweisen. Bild 8-22 zeigt den strukturellen Aufbau eines UPowerConst Blocks mit drei Ausgängen. Die ersten beiden Potenzen besitzen dabei den gleichen Exponenten c_1, während die dritte Potenz einen unterschiedlichen Exponenten c_2 aufweist.

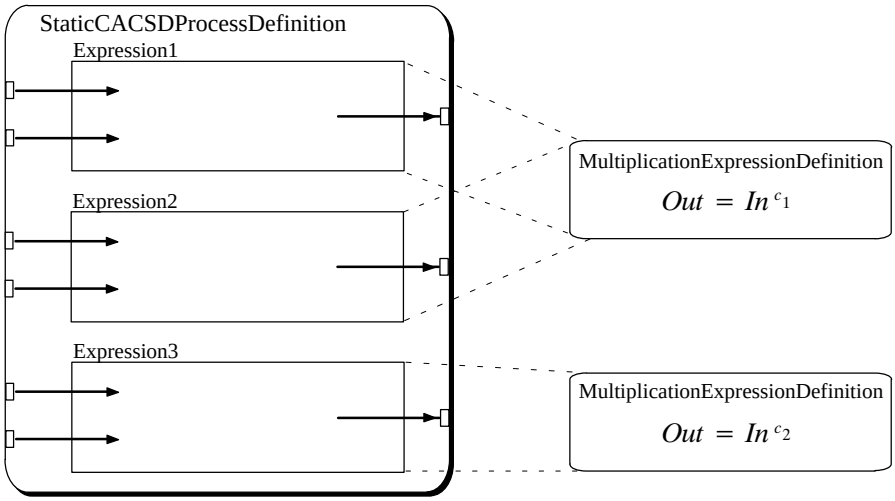


Bild 8-22: Struktureller Aufbau des UPowerConst Blocks mit drei Ausgängen in CDIF

Die Darstellung eines UPowerConst Blocks in CDIF mit einem Ausgang und einem Eingang ist in Bild 8-23 gezeigt.

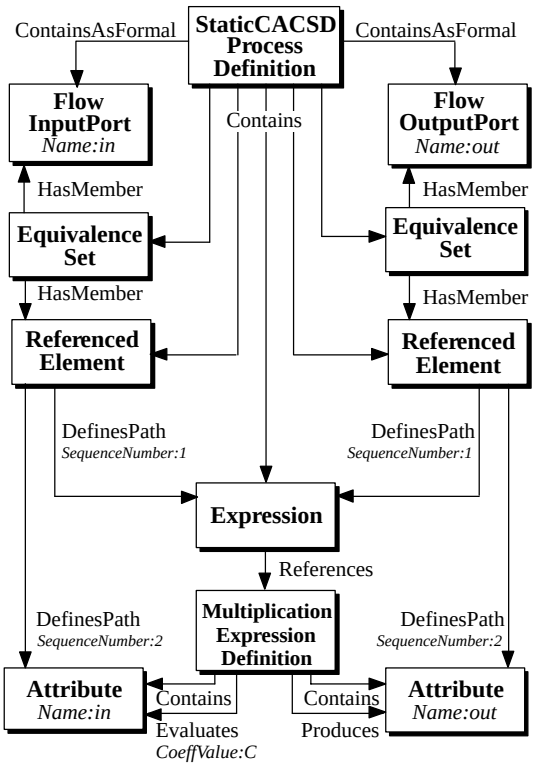


Bild 8-23: Abbildung eines UPowerConst Blocks in CDIF

Der Programmcode des UPowerConst Blocks kann mit Hilfe der Funktion *m_pow()* gebildet werden. Auch hierbei werden während der Ausführung des Codes auf einem Rapid Prototyping Rechners die durch unzulässige Eingabewerte verursachten Fehler abgefangen. Die Implementierung des UPowerConst Blocks hat das folgende Aussehen. Der Exponent wird dabei durch die Konstante *c* repräsentiert.

```
1    void calc_y( )
2    {
3       pDynamicData->OutUPowC = m_pow(pFixedData->InUPowC, c);
4    }
```

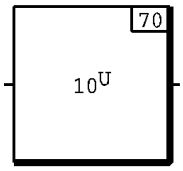


Bild 8-24: Darstellung des ConstPowerU Blocks in MATRIX$_X$™

Beim ConstPowerU Block von MATRIX$_X$™ (Bild 8-24) wird im Gegensatz zum UPowerConst Block die Basis durch einen konstanten Parameter und der Exponent durch den Eingang des Blocks festgelegt. Die Übertragungsgleichung des ConstPowerU Blocks ergibt sich also zu:

$$y_k = c_k^{u_k};\quad k \le n,\ k \in \mathbb{N} \qquad \begin{array}{ll} \text{und} & c_k \in \mathbb{R}_+;\ u_k \in \mathbb{R} \\ \text{oder} & c_k = 0;\ u_k \in \mathbb{R}_+ \\ \text{oder} & c_k \in \mathbb{R}_-;\ u_k \in \mathbb{N} \end{array} \qquad (8.7)$$

Die Darstellung dieses Blocks ist mit den Entities und Relationen der Expression Subject Area oder den übrigen Subject Areas nicht möglich. Dies liegt daran, daß eine Abbildung des Blocks mit der *MultiplicationExpressionDefinition* nicht vorgenommen werden kann, da der Exponent nur konstante Werte annehmen kann (siehe Abbildung des UPowerConst Blocks). Die Abbildung eines Exponenten, der von dem Eingangswert eines Blocks abhängig ist, ist mit der *MultiplicationExpressionDefinition* somit nicht möglich.

Um dennoch eine Abbildung dieses Blocks vornehmen zu können, ist eine individuelle Erweiterung des Meta-Modells notwendig. Die Expression Subject Area wurde um die folgenden drei Elemente erweitert:

- Entity *PowerExpressionDefinition*
- Relation *HasExponent*
- Relation *IsExponentFor*

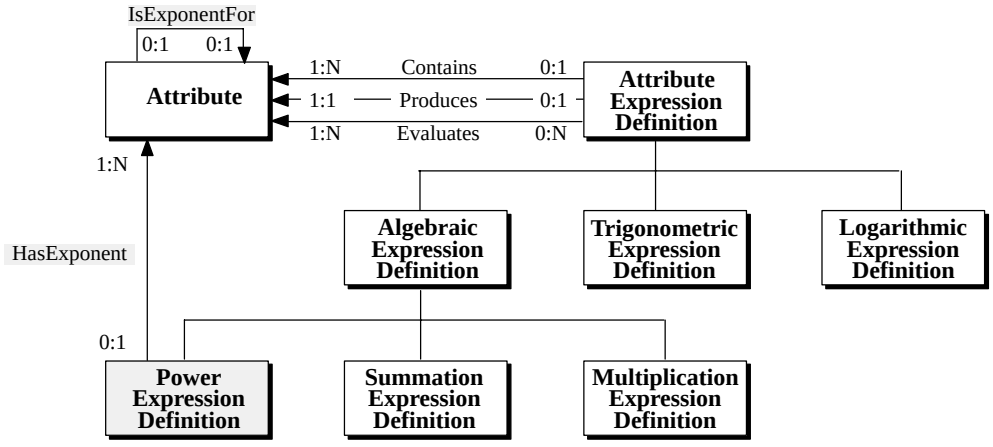


Bild 8-25: Ausschnitt der erweiterten Expression Subject Area

Mit Hilfe der neuen Elemente können Polynome mit folgendem Aufbau in CDIF abgebildet werden:

$$y = \sum_{i=1}^{j} c_i \cdot u_i^{k_i} \qquad (8.8)$$

Zur Darstellung eines Polynoms nach Gleichung (8.8) existiert ein Verweis der *PowerExpression-Definition* mit der Relation *Evaluates* auf die Basen des Polynoms. Der Faktor c_i wird durch das Attribut *CoeffValue* der Relation *Evaluates* zugeordnet. Diese Darstellung wurde in Anlehnung an die *MultiplicationExpressionDefinition* der Expression Subject Area gewählt. Die einzelnen Exponenten des Polynoms werden entsprechend den Basen durch die Entity *Attribute* abgebildet. Die *PowerExpressionDefinition* weist auf einen Exponenten mit der Relation *HasExponent* hin. Um Basen und Exponenten richtig zuordnen zu können, zeigt ein *Attribute* (Exponent) mit der Relation *IsExponentFor* auf ein *Attribute*, das die zugehörige Basis darstellt. Diese Zuordnung wurde in Anlehnung an die *InterpolationExpressionDefinition* aus der CACSD Subject Area gewählt.

In CDIF wird also zur Darstellung eines ConstPowerU Blocks eine Basis mit einem Faktor (*CoeffValue*) und einem Exponenten benötigt. Dem Faktor wird entsprechend dem MATRIX$_X$™ Block der Wert eins zugewiesen (Bild 8-26). Da der Exponent direkt vom Eingang des Blocks abhängt, wird das entsprechende *Attribute* der *PowerExpressionDefinition* über die Elemente *EquivalenceSet*, *HasMember*, *ReferencedElement* und *DefinesPath* mit dem *FlowInPort* verknüpft. Die Basis ist dabei nicht von einem Eingang, sondern einem konstanten Parameter des Blocks abhängig, weshalb das ihr entsprechende Attribute statt mit einem *FlowInPort* mit einem *ConstantAttribute* verbunden wird. Die Abbildung des ConstPowerU Blocks ist in Bild 8-26 für einen Ausgang dargestellt.

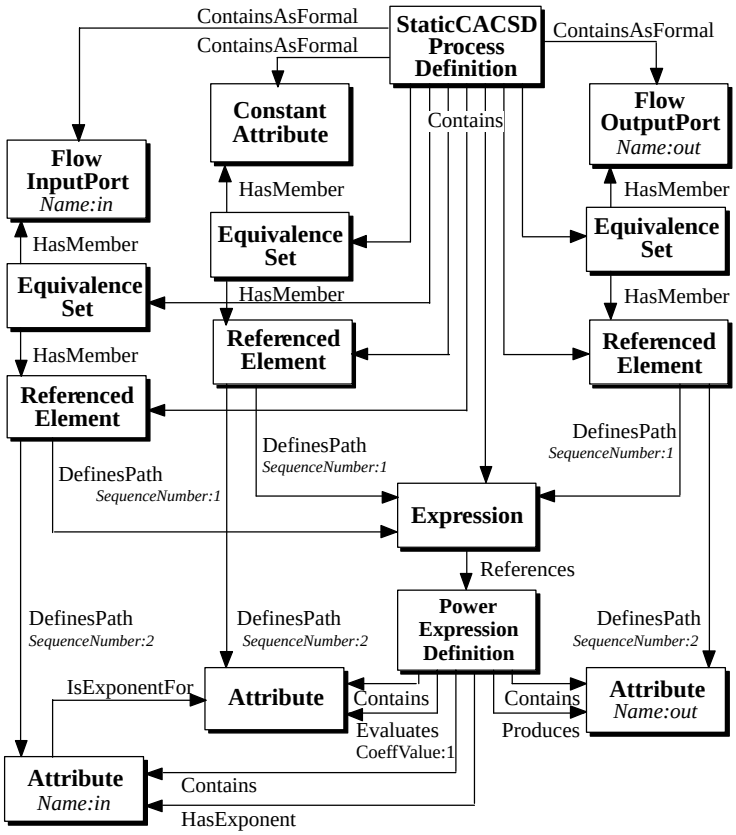


Bild 8-26: Abbildung eines ConstPowerU Blocks in CDIF

Hierbei sei angemerkt, daß die vorgenommene Erweiterung der Expression Subject Area nach Kapitel 6.3.3 erlaubt ist und in den Datentransfer aufgenommen werden muß. Somit schränkt diese Erweiterung den Datenaustausch zwischen CDIF verarbeitenden Werkzeugen in keiner Weise ein. Die vorgenommene Erweiterung kann wegen ihres allgemeinen Aufbaus für die Abbildung komplexerer Funktionen eingesetzt werden.

Bei der Implementierung in C-Code kann der allgemeine Aufbau der *m_pow()* Funktion auch für die Abbildung des ConstPowerU Blocks benutzt werden. Das Codefragment für diesen Block besitzt das folgende Aussehen. Die Basis wird dabei durch die Konstante *c* repräsentiert.

```
1    void calc_y( )
2    {
3        pDynamicData->OutCPowU = m_pow(c, pFixedData->InCPowU);
4    }
```

Abbildung exponentieller und logarithmischer Blöcke

Der Exponential Block in MATRIX$_X$™ (Bild 8-27) erlaubt die Darstellung von *n* unabhängigen Exponentialfunktionen. Der Block besitzt *n* Ausgänge und *n* Eingänge. Seine Übertragungsgleichung lautet:

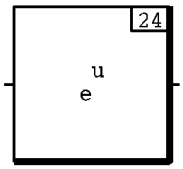


Bild 8-27: Darstellung des Exponential Blocks in MATRIX$_X$™

$$y_k = e^{u_k}; \quad u \in \mathbb{R}; \; k \le n, \; k \in \mathbb{N} \qquad (8.9)$$

Zur Abbildung der Exponentialfunktion in CDIF kann die *LogarithmicExpressionDefinition* der Expression Subject Area benutzt werden. Mit dieser Entity können Potenzfunktionen und Logarithmen zur Basis e, 2 und 10 abgebildet werden. Die Auswahl dieser Funktionen wird über ein Attribut *Type* festgelegt. Zur Darstellung des Eingangs und des Ergebnisses verwendet die *LogarithmicExpressionDefinition* zwei Entities vom Typ *Attribute*. Auf den Eingang verweist die *LogarithmicExpressionDefinition* mit der Relation *Evaluates*, während auf das Ergebnis mit der Relation *Produces* verwiesen wird. Bild 8-28 zeigt die Abbildung eines Exponential Blocks in CDIF.

Der Aufbau des C-Codes für den Exponential Block hat das folgende Aussehen. Für die Berechnung wird die ANSI-C Funktion *exp()* benutzt.

```
1    void calc_y( )
2    {
3       pDynamicData->OutExp = exp(pFixedData->InExp);
4    }
```

Der Logarithm (Logarithmus) Block in MATRIX$_X$™ (Bild 8-29) erlaubt die Darstellung der Logarithmusfunktion von *n* unabhängigen Ausgängen. Der Wertebereich ist dabei auf reelle Zahlen größer als Null beschränkt. Die Übertragungsgleichung des Logarithm Blocks für *n* Ein- und Ausgänge lautet:

$$y_k = \log\left(u_k\right); \quad u \in \mathbb{R}_+, \; k \le n, \; k \in \mathbb{N} \qquad (8.10)$$

Die Darstellung des Logarithm Blocks in CDIF kann über die bereits beim Exponential Block benutzte *LogarithmicExpressionDefinition* erfolgen. Hierbei wird der *LogarithmicExpressionDefini-*

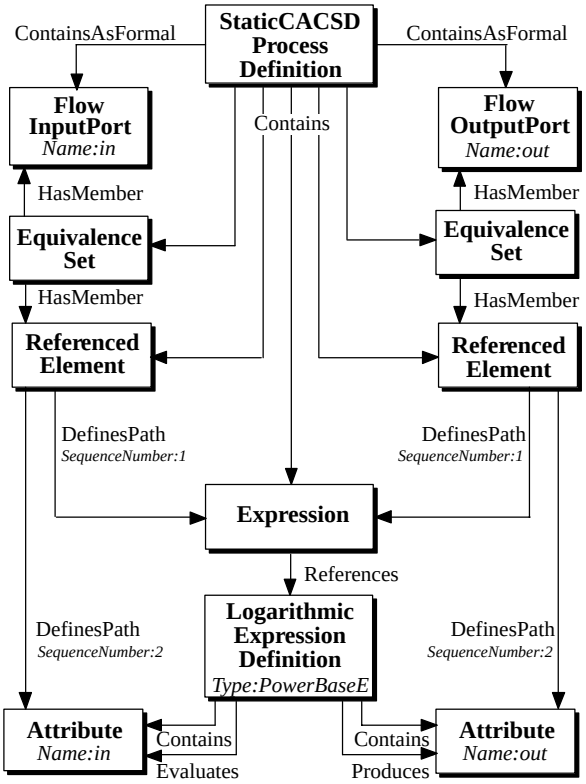


Bild 8-28: Abbildung eines Exponential Blocks in CDIF

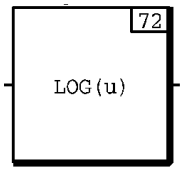


Bild 8-29: Darstellung des Logarithm Blocks in $MATRIX_X$ ™

tion das Attribut *Type* mit dem Wert *LogarithmBase10* zugewiesen. Der Aufbau des Blocks ist bis auf das Attribut identisch zum Exponential Block.

Der Aufbau des C-Codes erfolgt über die *m_log()* Funktion, die zunächst prüft, ob die Eingabewerte im erlaubten Wertebereich liegen und gegebenenfalls mit Hilfe der ANSI-C Funktion *log()* die Berechnung durchführt.

```
1    void calc_y( )
2      {
3       pDynamicData->OutLog = m_log(pFixedData->InLog);
4      }

5    float m_log(float x)
6      {
7      float back;
8
9      if (x > 0)
10       back = log(x);
11     else
12       {
13       if (x<=0)
14         {
15         /* Fehlerbehandlung */
16         ...
```

```
17            }
18          }
19        return back;
20        }
```

Abbildung von Interpolationsblöcken und stückweise linearen Blöcken

Für die Darstellung linearer Interpolationen wird von MATRIX$_X$™ der Linear Interpolation Block zur Verfügung gestellt (Bild 8-30). In der Grundeinstellung besitzt dieser Block einen Eingang, einen Ausgang und zwei Parameter-Vektoren zur Festlegung der Stützpunkte und Stützwerte. Der Ausgang des Linear Interpolation Blocks wird durch eine Interpolation der beiden Stützstellen berechnet, die den Eingangswert einschließen. Die Übertragungsgleichung des Linear Interpolation Blocks lautet:

$$y = \frac{v_{n+1} - v_n}{s_{n+1} - s_n}\,; \quad s_n \le u < s_{n+1};\ v,\ s,\ u \in \mathbb{R}\,,\ n \in \mathbb{N} \qquad (8.11)$$

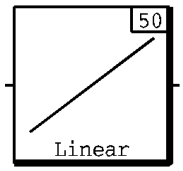


Bild 8-30: Darstellung des Linear Interpolation Blocks in MATRIX$_X$™

Der Term $s_{n+1} - s_n$ stellt dabei den Stützpunkte-Vektor und der Term $v_{n+1} - v_n$ den zugehörigen Stützwerte-Vektor dar. Werden dem Linear Interpolation Block mehrere Ausgänge zugewiesen, wird für jeden Ausgang je ein Stützpunkte-Vektor und ein Stützwerte-Vektor erzeugt. Bei der Wahl von *n* Ausgängen werden vom Linear Interpolation Block *n* unabhängige Interpolationen repräsentiert.

Zur Abbildung des Linear Interpolation Blocks in CDIF verfügt die CACSD Subject Area über die Entity *InterpolationExpressionDefinition*. Dieser Entity können die Ein- und Ausgangswerte als *Attributes* mit der Relation *HasInput* und *HasOutput* zugewiesen werden. Stützstellen und Stützwerte werden der *InterpolationExpressionDefinition* über die Relationen *HasBreakpoint* und *HasValue* zugewiesen. Die Zuordnung eines Stützwertes zu dem entsprechenden Stützpunkt erfolgt über die Relation *IsValueForBreakpoint*. Bild 8-31 zeigt die Abbildung eines Linear Interpolation Blocks mit einem Eingang, einem Ausgang und einer Stützstelle in CDIF. Hierbei sei angemerkt, daß eine *InterpolationExpressionDefinition* mindestens zwei Stützstellen besitzen muß, da die Relationen *HasBreakpoint* und *HasValue* die Kardinalität 2:N besitzen. Da der Aufbau weiterer Stützstellen in derselben Art erfolgt wie in Bild 8-31 dargestellt, wird aus Vereinfachungsgründen auf die Darstellung weiterer Stützstellen verzichtet.

Der Aufbau des C-Codes des Linear Interpolation Blocks erfolgt mit der *m_lip()* Funktion. Der Funktion wird der Eingangswert des Linear Interpolation Blocks als float Variable übergeben. Die Stützpunkte und Stützwerte werden als ein eindimensionales Array von float Werten übergeben. Ein weiterer Parameter, der die Anzahl der Stützstellen angibt, bildet den letzten Übergabewert.

```
1     float m_lip(float x, float breakpoint[], float breakvalue[],int size)
2       {
3       float back=0.0;
4       int i;
5
6       if (x < breakpoint[0])
```

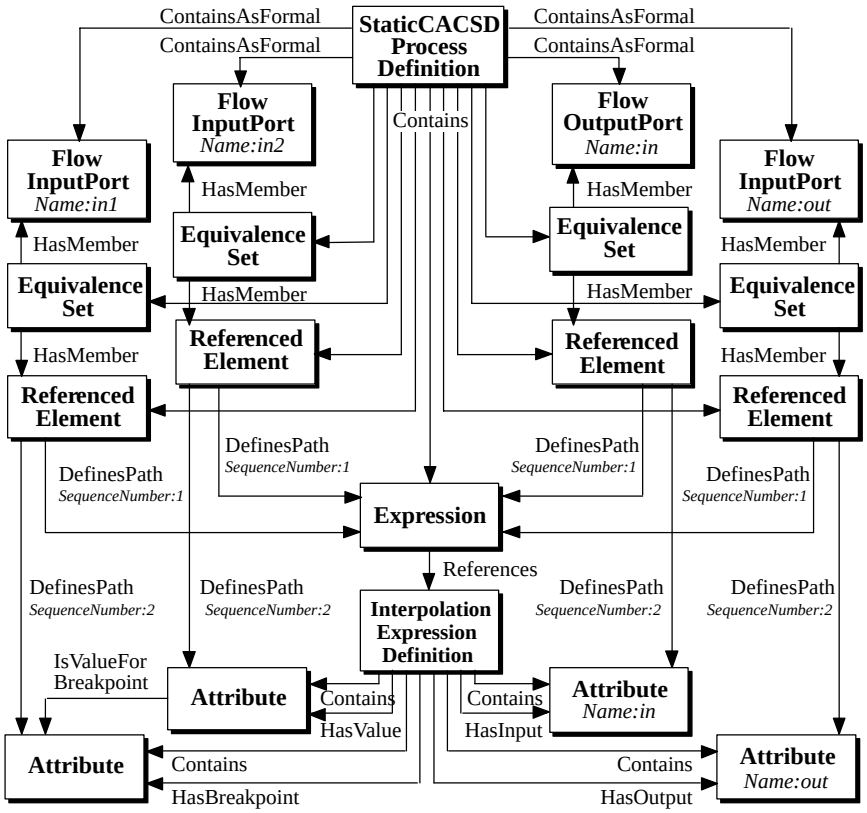


Bild 8-31: Abbildung eines Linear Interpolation Blocks in CDIF

```
 7          back = breakvalue[0]+(x-breakpoint[0])*(breakvalue[1]-breakvalue[0])/(breakpoint[1] -breakpoint[0]);
 8        else
 9          {
10          if (x >= breakpoint[size-1])
11            back = breakvalue[size-1]+(x-breakpoint[size-1])*(breakvalue[size-1]-
                        breakvalue[size-2])/(breakpoint[size-1]-breakpoint[size-2]);
12          else
13            {
14            i=0;
15            while (1)
16              {
17              if (i>=size)
18                {
19                /* Fehlerbehandlung */
20                ...
21                 break;
22                }
23              if ((stuetzstelle[i] <= x) && (stuetzstelle[i+1] > x))
24                {
25                  back =   breakvalue[i]+(x-stuetzstelle[i])*(breakvalue[i+1]-breakvalue[i])/(stuetzstelle[i+1]-stuetzstelle[i]);
26                 break;
27                 }
28               i++;
29               }
30            }
31          }
32          return  back;
33        }
```

Bei der Codeerzeugung wird zur Behandlung einer Interpolation die Funktion *m_lip()* sowie ein Aufruf dieser Funktion mit den jeweiligen Stützstellen und Stützwerten erzeugt. Um auch bei meh-

reren Interpolationen die einzelnen Vektoren identifizieren zu können, wird bei der Vergabe des Interpolationsvektors die eineindeutigen Werte der zugehörigen CDIF *Expression* angehängt. Für eine Interpolation mit den Stützstellen und Stützwerten (-1.0, 5.0), (1.0, 3.2) und (4.5, 7.0) wird der folgende Aufruf der Funktion *m_lip()* erzeugt:

```
1    /* Interpolation Variables */
2    float IP_Po263[3] = {-1.0, 1.0, 4.5};
3    float IP_Va263[3] = {5.0, 3.2, 7.0};
     ...
4    void calc_y( )
5    {
6    pDynamicData->OutInterp = m_lip(pFixedData->InInterp, IP_Po263, IP_Va263, 3);
7    }
```

Die nachfolgend vorgestellten Blöcke Absolute Value, Saturation, Limiter und Dead Band Blöcke gehören zur Gruppe der stückweise linearen Blöcke. Um den CDIF Standard so knapp und übersichtlich wie möglich zu halten, existieren keine direkten Abbildungen dieser Blöcke in einer eigenen Definition. Da sich die Übertragungsgleichungen dieser Blöcke jedoch aus stückweise linearen Funktionen zusammensetzen lassen, werden diese Blöcke als lineare Interpolation dargestellt.

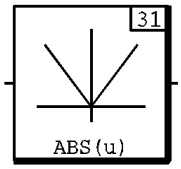


Bild 8-32: Darstellung des Absolute Value Blocks in MATRIX$_X$™

Der Absolute Value Block (Bild 8-32) bildet den Betrag des eingehenden Signals und stellt das Ergebnis an seinem Ausgang zur Verfügung. Auch bei diesem Block gleicht sich die Zahl der Eingänge der Zahl der Ausgänge an. Ein Absolute Value Block mit n Ausgängen besitzt die folgende Übertragungsgleichung:

$$y_k = |u_k|; \quad u \in \mathbb{R};\ 0 \le k \le n;\ k,n \in \mathbb{N} \qquad (8.12)$$

Da für diesen Block keine direkte Entsprechung in CDIF existiert, kann seine Übertragungsgleichung für Werte größer gleich Null als Gerade durch den Ursprung mit der Steigung ’1’ und für Werte kleiner Null als Gerade durch den Ursprung mit der Steigung ’-1’ dargestellt werden. Durch diese Darstellung ist es möglich, den Absolute Value Block in CDIF als Interpolation mit den Stützstellen (-1.0, 1.0), (0.0, 0.0) und (1.0, 1.0) abzubilden.

Der Aufbau des C-Codes des Absolute Value Blocks hat unter Verwendung der Funktion *m_lip()* das folgende Aussehen:

```
1    /* Interpolation Variables */
2    float IP_Po263[3] = {-1.0, 0.0, 1.0};
3    float IP_Va263[3] = {1.0, 0.0, 1.0};
     ...
4    void calc_y( )
5    {
6    pDynamicData->OutInterp = m_lip(pFixedData->InInterp, IP_Po263, IP_Va263, 3);
7    }
```

Um den C-Code mit Hilfe der ANSI C-Funktion *abs()* aufzubauen, kann eine zusätzliche Abfrage durchgeführt werden, die bei Verwendung der drei obigen Stützstellen eine Ersetzung des obigen Codes vornimmt.

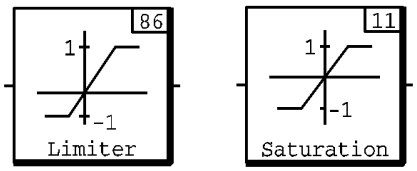


Bild 8-33: Darstellung des Limiter und Saturation Blocks in MATRIX$_X$™

Die beiden Blöcke Limiter und Saturation Block (Bild 8-33) dienen der Begrenzung eines Signals auf einen bestimmten Wertebereich. Liegt der Eingangswert der Blöcke innerhalb des durch Parameter festgelegten Intervalls, so wird der Eingangswert unverändert an den Ausgang weitergegeben. Bei Unter- oder Überschreitung des vorgegebenen Wertebereichs nimmt der Ausgang den Wert der oberen oder unteren Grenze des Intervalls an. Der Unterschied zwischen beiden Blöcken liegt in den Intervallgrenzen. Beim Limiter können die untere und die obere Grenze über zwei Parameter unabhängig voneinander gewählt werden. Beim Saturation Block wird der untere Wert dem negativen Wert der oberen Grenze angepaßt. Beide Blöcke besitzen einen Ausgang, einen Eingang und einen, bzw. zwei Parameter zur Einstellung des Wertebereichs. Bei einer Erhöhung der Ausgänge wird die Zahl der Eingänge und Parameter angepaßt. Für einen Limiter mit *n* Ausgängen ergibt sich die folgende Übertragungsgleichung:

$$y_k = \begin{cases} c_{max_k} & \text{für} \quad u_k > c_{max_k} \\ u_k & \text{für} \quad c_{min_k} \le u_k \le c_{max_k} \\ c_{min_k} & \text{für} \quad u_k > c_{min_k} \end{cases} \qquad (8.13)$$

$$\text{mit} \quad u_k,\, c_{min_k},\, c_{max_k} \in \mathbb{R};\; 0 \le k \le n\,; k,n \in \mathbb{N}$$

Hierbei stellen c_{max_k} und c_{min_k} die Intervallgrenzen des Wertebereichs der einzelnen Funktionen dar. Für die Übertragungsgleichung eines Saturation Blocks gilt Gleichung (8.13) mit der Einschränkung $c_{min_k} = -\,c_{max_k}$ ebenfalls.

Beide Blöcke können in CDIF als Interpolation dargestellt werden und besitzen die folgenden vier Stützstellen (c_{min_k}-1, c_{min_k}), (c_{min_k}, c_{min_k}), (c_{max_k}, c_{max_k}), (c_{max_k}+1, c_{max_k}). Mit diesen Parametern wird der C-Code in der gleichen Weise wie für Interpolationsblöcke erzeugt.

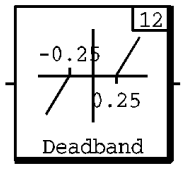


Bild 8-34: Darstellung des Dead Band Blocks in MATRIX$_X$™

Mit dem Dead Band Block (Bild 8-34) kann ein parametrierbares, zum Ursprung symmetrisches Null-Intervall gewählt werden. Befindet sich das Eingangssignal innerhalb dieses Intervalls, beträgt der Ausgangswert Null. Außerhalb des Intervalls stellt der Ausgang eine Gerade mit der Steigung ’1’ dar. Die Übertragungsgleichung des Dead Band Blocks ergibt sich also zu:

$$y_k = \begin{cases} u_k - \frac{d_k}{2} & \text{für} \quad u_k > \frac{d_k}{2} \\ 0 & \text{für} \quad |u_k| \le \frac{d_k}{2} \\ u_k + \frac{d_k}{2} & \text{für} \quad u_k < -\frac{d_k}{2} \end{cases} \qquad (8.14)$$

$$\text{mit} \quad u_k,\, d_k \in \mathbb{R};\; 0 \le k \le n;\; k,n \in \mathbb{N}$$

Der Parameter d_k stellt dabei den Parameter zur Einstellung der Bandbreite dar. Der Dead Band Block kann in CDIF als Interpolation mit den Stützstellen $(-\frac{d}{2} - 1, -1), (-\frac{d}{2}, 0), (\frac{d}{2}, 0), (\frac{d}{2} + 1, 1)$ dargestellt werden. Mit diesen Parametern wird der C-Code in gleicher Weise wie für Interpolationsblöcke erzeugt.

Abbildung von Signalgeneratoren

Im Gegensatz zu den bisher vorgestellten Blöcken besitzen Signalgeneratoren nur Ausgänge, die über Parameter eine Anpassung der erzeugten Signale vornehmen. Der Pulse Wave and Square Wave Block (Bild 8-35) ermöglichen die Bildung eines Rechtecksignals.

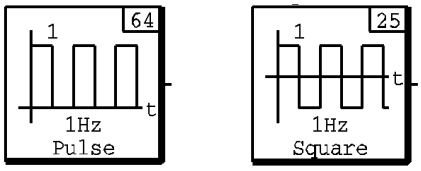


Bild 8-35: Darstellung des Pulse Wave und Square Wave Blocks in MATRIX$_X$™

Während der Pulse Wave Block Rechtecksignale zwischen ’0’ und einem frei wählbaren Maximalwert erzeugt, generiert der Square Wave Block abwechselnd Rechtecksignale mit positiver und negativer Pulshöhe. Für die Charakterisierung des Signalverlaufs können vier Parameter verwendet werden: Startzeit, Pulshöhe, Pulsweite und Signalfrequenz. Bild 8-36 stellt den Signalverlauf eines Pulse Wave und Square Wave Blocks dar.

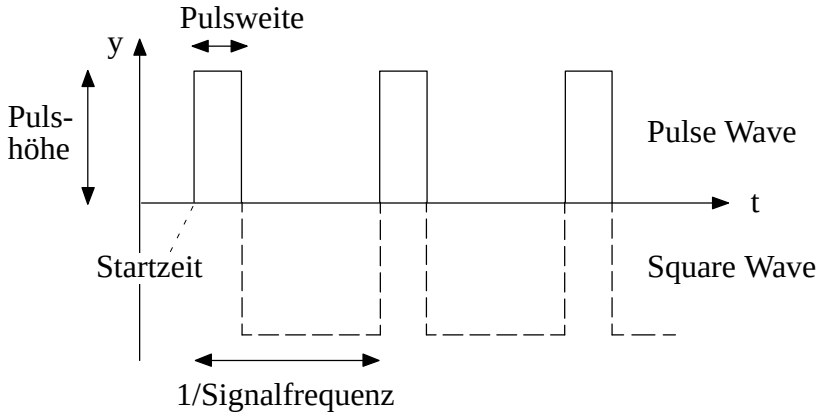


Bild 8-36: Signalverlauf eines Pulse Wave und Square Wave Blocks

Zur Abbildung eines Pulse Wave und Square Wave Blocks in CDIF verfügt die CACSD Subject Area über die Entity *SignalGeneratorDefinition*. Um eine Differenzierung zwischen den Signalgeneratortypen vornehmen zu können, gliedert sich die *SignalGeneratorDefinition* in die Entities *SinusoidalGeneratorDefinition, SquareGeneratorDefinition, RandomGeneratorDefinition* und *Ar-*

bitraryWaveformDefinition. Die Darstellung eines Pulse Wave und Square Wave Blocks kann mit einer *SquareGeneratorDefinition* vorgenommen werden. Die Parameter, die zur Charakterisierung des Signalverlaufs dienen, werden über neun Attribute festgelegt:

- *StartTime*: Dieses Attribut legt die Zeit fest, zu der das Generatorsignal zum ersten Mal erzeugt werden soll. Vor der Startzeit hat das Generatorsignal den Wert ’0’. Der Wert dieses Attributes entspricht dem Wert, der von dem MATRIX$_X$™ Parameter Startzeit festgelegt wird.
- *EndTime*: Dieses Attribut legt die Endzeit für das erzeugte Signal fest. Nach dem Erreichen dieser Zeit nimmt das Generatorsignal den Wert ’0’ an. Verfügt ein Signalgenerator über keine Endzeit, so wird dieses Attribut nicht gesetzt.
- *TimeUnit*: Dieses Attribut gibt die Zeiteinheit an, nach der alle weiteren Zeitangaben für den Signalverlauf festgelegt werden.
- *IsPeriodic*: Dieses Attribut vom Typ Boolean legt fest, ob ein periodisches Signal dargestellt werden soll. Beim Wert ’True’ wird das Ausgangssignal periodisch wiederholt. Liegt der Wert ’False’ vor, so wird das Signal nur ein einziges Mal generiert und nimmt anschließend den Wert ’0’ an. Beim Pulse Wave und Square Wave Block muß dieses Attribut den Wert ’True’ besitzen.
- *IsStartMax*: Mit diesem Attribut vom Typ Boolean wird festgelegt, ob das Generatorsignal mit seinem Maximalwert (’True’) oder seinem Minimalwert (’False’) beginnt. Sowohl der Pulse Wave als auch der Square Wave Block beginnen mit dem Maximalwert.
- *MaximumValue*: Dieses Attribut legt den Maximalwert des generierten Signals fest. Für den Pulse Wave und Square Wave Block entspricht dieses Attribut der Pulshöhe.
- *MinimumValue*: Dieses Attribut legt den Minmalwert des generierten Signals fest. Für den Pulse Wave Block beträgt der Wert dieses Attributs ’0’, während beim Square Wave Block das Attribut *MinimumValue* der negativen Pulshöhe entspricht.
- *MaxTime*: Dieses Attribut definiert die Pulsweite des Signals, d.h. die Zeitdauer, für das das Signal den Maximalwert annimmt. Damit entspricht das Attribut dem Parameter Pulsweite in MATRIX$_X$™.
- *Period*: Mit diesem Attribut wird die Periodendauer des Signal festgelegt. Im Gegensatz zu MATRIX$_X$™ wird dieser Wert in CDIF nicht als Frequenz, sondern als Zeit angegeben. Damit nimmt das Attribut Periode für den Pulse Wave und den Square Wave Block den Wert 1/Signalfrequenz an.

Bild 8-37 zeigt auf der linken Seite den Verlauf eines Square Wave Blocks, der nach ’1 s’ mit der Generierung des Signals beginnen soll, als Pulshöhe ’2’, als Pulsweite ’1 s’ und als Periode ’3 s’ besitzen soll und auf der rechten Seite die entsprechende Abbildung in CDIF. Bei einem Pulse Wave Block mit demselben Parametersatz müßte das Attribut *MinimumValue* auf ’0’ geändert werden.

Für den Aufbau des C-Codes des Pulse Wave und Square Wave Blocks ist es zweckmäßig, eine allgemeine Funktion *m_pulse()* zu erstellen, die zur Implementierung mehrerer Signalgeneratoren verwendet werden kann.

```
1    float m_pulse (float Time, float Max, float Min, float Width, float Period, float Start, int StartMax)
2      {
3      float Signal = 0.0;
4      float InternalTime;
5      float Rest;
6
```

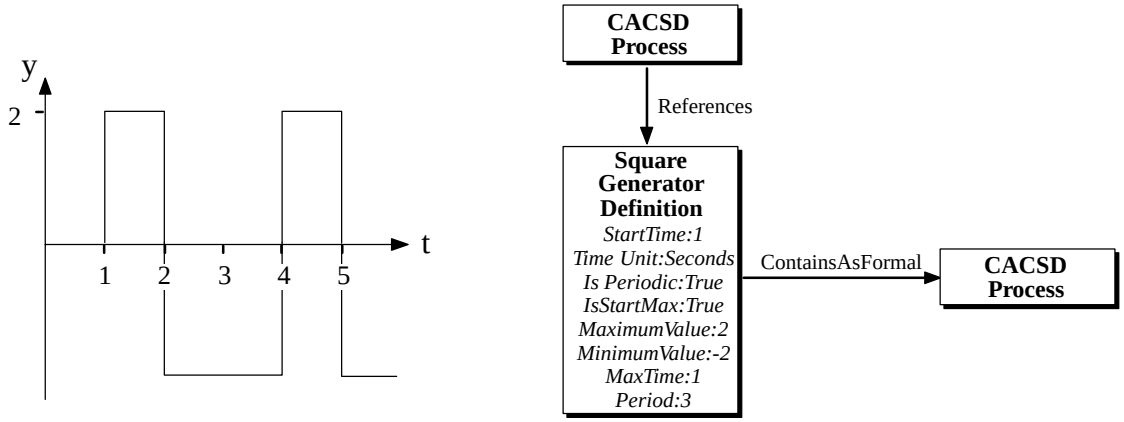


Bild 8-37: Abbildung eines Square Wave Blocks in CDIF

```
 7        if (Time >= Start)
 8          {
 9          InternalTime = Time - Start;
10          Rest = (float) fmod((double) InternalTime, (double) Period);
11          if (StartMax == 0)
12            {
13            if (Rest >= (Period - Width)) Signal = Max;
14            else Signal = Min;
15            }
16          else
17            {
18          if (Rest < Width) Signal = Max;
19            else Signal =Min;
20            }
21          }
22        return Signal;
23        }
```

Soll das Ausgangssignal eines Square Wave Blocks nach Bild 8-37 bestimmt werden, so sieht der zugehörige Modell-Code folgendermaßen aus:

```
1       void calc_y()
2         {
3         pDynamicData->OutPulse = m_pulse(SystemZeit, 2.0, -2.0, 1.0, 3.0, 1.0, true);
4         }
```

Die aktuelle Systemzeit, für die der Signalwert berechnet werden soll, wird der Funktion hier als Parameter SystemZeit übergeben. Die Bestimmung der aktuellen Systemzeit erfolgt im Scheduler, der mit der Periode T_{timer} aufgerufen wird und die Variable SystemZeit mit jeder Periode erhöht.

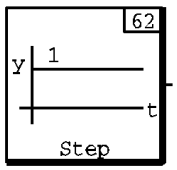


Bild 8-38: Darstellung des Step Blocks in MATRIX$_X$ ™

Mit dem Step Block (Bild 8-38) wird ein Signalgenerator zur Erzeugung von Sprungfunktionen zur Verfügung gestellt. Die für die Charakterisierung notwendigen Parameter sind die Startzeit T_{start} und die Sprunghöhe y_{max}. Für die Ausgangsfunktion des Step Blocks ergibt sich:

$$y_k(t) = \begin{cases} 0 & \text{für} \quad t < T_{start} \\ y_{k,max} & \text{für} \quad t \geq T_{start} \end{cases} \qquad 0 \leq k \leq n;\ n,k \in \mathbb{N};\ y_{k,max} \in \mathbb{R} \qquad (8.15)$$

Der Step Block stellt einen Sonderfall des Pulse Wave Blocks dar. Entsprechend kann auch die Darstellung in CDIF mit der *SquareGeneratorDefinition* vorgenommen werden. Die Attribute der *SquareGeneratorDefinition* besitzen für die Darstellung eines Step Blocks die folgenden Werte: *StartTime* = T_{start}, *TimeUnit* = Seconds, *IsPeriodic* = False, *IsStartMax* = True, *MaximumValue* = y_{max}, *MinimumValue* = 0. Den Attributen *MaxTime* und *Period* wird derselbe Wert (ohne Null) zugeordnet. Mit diesen Parametern nimmt das generierte Signal nach Erreichen des Startwerts permanent den Maximalwert an.

Auch der Aufbau des C-Codes erfolgt mit Hilfe der *m_pulse()* Funktion, die mit den obigen Parametern eine Sprungfunktion erzeugt. Für den Step Block ergibt sich der folgende Funktionsaufruf:

```
1    void calc_y()
2    {
3       pDynamicData->OutStep = m_pulse(SystemZeit, 2.0, 0.0, 5.0, 5.0, 1.0,true);
4    }
```

Mit den obigen Werten ergibt sich eine Sprungfunktion, die nach ’1 s’ den Maximalwert ’2’ annimmt.

Abbildung dynamischer Blöcke

Der Numerator-Denominator Block (Bild 8-39) dient der Darstellung von Differentialgleichungen und Differentialgleichungssystemen im Laplace Bereich. Im Gegensatz zu den bislang behandelten Blöcken, deren Ausgänge direkt von den Eingängen abhängig waren, verwendet dieser Block zur Berechnung des Ausgangs zusätzlich interne, aus dem Verlauf bereits vergangener Eingänge abhängiger Zustände.

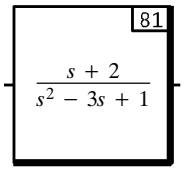


Bild 8-39: Darstellung des Numerator-Denominator Blocks in MATRIX$_X$ ™

Wie bereits bei der Einführung in CDIF gezeigt (Kapitel 6.5), werden rationale Übertragungsglieder der Form $Y(s) = G(s)\,U(s)$ in CDIF im Zustandsraum dargestellt:

$$\begin{aligned} \dot{\underline{x}} &= \underline{A}\,\underline{x} + \underline{B}\,u \\ y &= \underline{C}\,\underline{x} + \underline{D}\,u \end{aligned} \qquad (8.16)$$

Bei der Abbildung des Numerator-Denominator Blocks mit *h* Eingängen und *k* Ausgängen wird jede der Übertragungsfunktionen $G_{k,h} = \dfrac{Z_{k,h}}{N(s)}$ zunächst einzeln in den Zustandsraum transformiert und danach zu einem gesamten Zustandsraum mit den Vektoren $A_{k,h}$, $B_{k,h}$, $C_{k,h}$ und $D_{k,h}$ zusammengefügt. Da die Matrizen $A_{k,h}$ und $B_{k,h}$ aus den Koeffizienten des Nenners gebildet werden, die für alle Übertragungsfunktionen identisch sind, gilt:

$$A_{0,0} = A_{0,1} = ... = A_{1,0} = A_{1,1} = ... = A_{k-1,h-1} = A_{k,h} = A$$
$$B_{0,0} = B_{0,1} = ... = B_{1,0} = B_{1,1} = ... = B_{k-1,h-1} = B_{k,h} = B \qquad (8.17)$$

Für einen Numerator-Denominator Block ergibt sich also folgende Zustandsraumgleichung:

$$\dot{\underline{x}} = \begin{bmatrix} \underline{A} & 0 & ... & 0 \\ 0 & \underline{A} & ... & 0 \\ ... & ... & \ddots & ... \\ 0 & 0 & 0 & \underline{A} \end{bmatrix} \underline{x} + \begin{bmatrix} \underline{B} & 0 & ... & 0 \\ 0 & \underline{B} & ... & 0 \\ ... & ... & \ddots & ... \\ 0 & 0 & ... & \underline{B} \end{bmatrix} u$$
$$y = \begin{bmatrix} \underline{C}_{1,1} & \underline{C}_{1,2} & ... & \underline{C}_{1,h} \\ \underline{C}_{2,1} & \underline{C}_{2,2} & ... & \underline{C}_{2,h} \\ ... & ... & \ddots & ... \\ \underline{C}_{k,1} & \underline{C}_{k,2} & ... & \underline{C}_{k,h} \end{bmatrix} \underline{x} + \begin{bmatrix} \underline{D}_{1,1} & \underline{D}_{1,2} & ... & \underline{D}_{1,h} \\ \underline{D}_{2,1} & \underline{D}_{2,2} & ... & \underline{D}_{2,h} \\ ... & ... & \ddots & ... \\ \underline{D}_{k,1} & \underline{D}_{k,2} & ... & \underline{D}_{k,h} \end{bmatrix} u \qquad (8.18)$$

Der Aufbau des C-Codes zur Berechnung eines Zustandsraums erfolgt mittels numerischer Lösung der Zustandsvariablen. Für jede Zustandsvariable wird ein Integrationsalgorithmus angewendet. Da ein Zustandsraum ausschließlich aus Differentialgleichungen erster Ordnung besteht, können sich Integrationsalgorithmen auf die Lösung dieser Differentialgleichungsklasse beschränken. Bei der Klassifizierung dieser Algorithmen unterscheidet man zwischen Einschritt- und Mehrschrittverfahren sowie zwischen Verfahren mit fester oder variabler Schrittweite.

Mehrschrittverfahren benötigen neben einer bereits berechneten Näherung eines vorangegangenen Stützpunkts zusätzlich noch weitere Funktionswerte aus der Vergangenheit. Da zu Beginn der Verarbeitung bei $t = 0$ keine Werte aus der Vergangenheit bekannt sind, ist eine Anlaufrechnung mit einem geeigneten Einschrittverfahren notwendig. Der Rechenaufwand von Mehrschrittverfahren ist im Vergleich zu Einschrittverfahren hoch. Somit eignen sich diese Verfahren nicht für Algorithmen, bei denen eine feste, kurze Schrittweite vorgegeben ist und eine hohe Effizienz bei der Berechnung eines Einzelschritts benötigt wird. Für eine Abarbeitung im Rahmen von Rapid Prototyping eignen sich Einschrittverfahren besser.

Mehrschrittverfahren werden hauptsächlich bei Algorithmen mit einer variablen Schrittweite eingesetzt. Bei diesen Algorithmen wird eine Fehlerabschätzung eingesetzt, die aus einem durch Interpolation geschätzten Näherungswert und einem aus der Ableitung gewonnenen Korrekturwert besteht, um die Schrittweite bestmöglich anzupassen. Die Veränderung der Schrittweite ist jedoch hauptsächlich bei langsamveränderlichen Funktionen interessant, um eine mögliche Vergrößerung der Berechnungsintervalle durchzuführen. Wie bereits in Kapitel 7.2.2 dargestellt, ist auch der Zusatzaufwand beim Scheduling hoch, da keine einheitliche Timerauflösung benutzt werden kann.

Somit eignen sich Algorithmen mit fest vorgegebener Schrittweite für den Einsatz in einem Rapid Prototyping System am besten. Einschrittverfahren sind den Mehrschrittverfahren vorzuziehen, da eine größere Effizienz bei der Näherungsberechnung mit entsprechender Fehlerordnung vorliegt und somit kleinere Schrittweiten erreicht werden können [Kock97]. Für die Lösung einer Differentialgleichung mittels Einschrittverfahren mit fester Schrittweite existieren mehrere Algorithmen. Die drei wichtigsten Algorithmen werden im folgenden vorgestellt.

Der *Euler/Cauchy-Algorithmus* stellt das einfachste Prinzip zur numerischen Integration dar. Um die Ausgangsgleichung (8.16) zu lösen wird die Ableitung durch den Differenzenquotienten $\frac{\Delta x}{\Delta t}$

ersetzt. Die Lösung der Integration erfolgt nach einer geeigneten Umformung aus der folgenden Gleichung:

$$x(t + h) = x(t) + h\, f(x(t),\, u(t)) \qquad (8.19)$$

Die Variable $h = \Delta t$ repräsentiert dabei die angenommene feste Schrittweite. Der Funktionswert an der nächsten Stützstelle $x(t + h)$ ergibt sich also aus dem vorangegangenen Stützstelle $x(t)$ und der Ableitung an dieser Stelle $f(x(t),\, u(t)) = \dot{x}(t)$ multipliziert mit der Schrittweite h. Somit nähert sich das Verfahren durch die Annäherung der gesuchten Fläche unter die Ableitungskurve mit Hilfe der Untersumme an.

Der Euler/Cauchy-Algorithmus zeichnet sich durch einen geringen Rechenaufwand aus und stellt eine Näherung erster Ordnung der Integration dar. Die Fehlerordnung des Algorithmus beträgt $o(h^2)$ und benötigt deswegen eine kurze Schrittweite, um eine brauchbare Genauigkeit zu erreichen. Der Diskretisierungsfehler nimmt bei größeren Schrittweiten stark zu.

Der *Runge/Kutta-Algorithmus zweiter Ordnung* unterscheidet sich vom Euler/Cauchy-Algorithmus durch eine zweite Näherung und wird deswegen auch als verbesserter Euler/Cauchy-Algorithmus bezeichnet. Die aus dem ersten Schritt erhaltene Näherungslösung wird in die Gleichung $\dot{x}(t + h) = f(x(t + h),\, u(t + h))$ eingesetzt, um die Steigung an der nächsten Stützstelle zu berechnen. Die beiden Steigungen $\dot{x}(t)$ und $\dot{x}(t + h)$ werden gemittelt und dieser Wert mit dem Funktionswert des letzten Schritts summiert. Damit entspricht dieser Algorithmus der Mittelung von Ober- und Untersumme zur Annäherung des Flächeninhalts unter der Ableitungskurve.

Es ergeben sich die folgenden Rechnenschritte für den Algorithmus. Mit der Einführung der Variablen k_1 wird die Steigung zum Zeitpunkt t bezeichnet:

$$k_1 = f(x(t),\, u(t)) \qquad (8.20)$$

Für die erste Näherung unter Verwendung der Variablen k_1 ergibt sich:

$$x(t + h) = x(t) + h\, k_1 = \tilde{x}(t + h) \qquad (8.21)$$

Für die Steigung zum Zeitpunkt $t + h$ erhält man unter Einführung der Variablen k_2:

$$k_2 = f(\tilde{x}(t + h),\, u(t + h)) \qquad (8.22)$$

Für die zweite Näherung unter Verwendung der Variablen k_2 ergibt sich:

$$x(t + h) = x(t) + \frac{h}{2}(k_1 + k_2) \qquad (8.23)$$

Somit ergibt sich die folgende algorithmische Darstellung:

```
 1    CalculateDerivationFunctionsOfX()
 2    for each dimension do
 3      let SavedIntegrationValue = IntegrationValue
 4      let k1 = StateEquation
 5      let IntegrationValue = SavedIntegrationValue + k1 * ModelStep
 6    end for
 7    CalculateStateAndOutputEquations()
 8    CalculateDerivationFunctionsOfX()
 9    for each dimension do
10      let k2 = IntegrationValue
11      let IntegrationValue = SavedIntegrationValue + 0,5 * (k1 + k2) * ModelStep
12      end for
13    CalculateStateAndOutputEquations()
```

Ein Problem bei der Abarbeitung stellt der Eingangswert an der Stelle $t + h$ dar. Bei der Berechnung der zweiten Näherung ist dieser Wert noch nicht verfügbar und muß deswegen entweder interpoliert oder durch den Eingangswert an der Stelle t ersetzt werden.

Der Rechenaufwand des Runge/Kutta-Algorithmus zweiter Ordnung liegt im Vergleich zum Euler/Cauchy-Algorithmus etwa um den Faktor zwei höher, da die Funktion für die Ableitung zweimal berechnet werden muß. Für die zu integrierende Fläche erhält man allerdings einen Näherungswert zweiter Ordnung und damit die Fehlerordnung $o(h^3)$. Der Runge/Kutta-Algorithmus zweiter Ordnung stellt einen guten Kompromiß zwischen Abarbeitungsgeschwindigkeit und erzielter Genauigkeit dar.

Die Codegenerierung mit einem Runge/Kutta-Algorithmus zweiter Ordnung soll anhand eines Beispiels näher erläutert werden. Geht man von einer Übertragungsfunktion $G(s)$ nach Bild 8-39 aus, so erhält man für die Codegenerierung bereits eine Darstellung im Zustandsraum. Auf der linken Seite von Gleichung (8.24) ist die Übertragungsgleichung dargestellt, auf der rechten Seite die zugehörige Zustandsraumdarstellung.

$$G(s) = \frac{3 + s}{1 + 3s + 2s^2} \qquad \begin{aligned} \dot{\underline{x}} &= \begin{bmatrix} 0 & 1 \\ -0.5 & -1.5 \end{bmatrix} \underline{x} + \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} u \\ y &= \begin{bmatrix} 2.5 & 1 \end{bmatrix} \underline{x} + 0 \cdot u \end{aligned} \qquad (8.24)$$

Für die Berechnung des Zustandsraums wird die Funktion *integration()* (siehe untenstehender C-Code, Zeile 2-25) aufgerufen, die in der Funktion *calc_xdott()* (Zeile 26-30) die Ableitung der internen Zustände $\dot{\underline{x}}$ berechnet. Im untenstehenden C-Code sind diese Ableitungen im Array pXD[j] enthalten. Danach wird die erste Näherung durch ein Euler-Verfahren berechnet (Zeile 11-16), die einer Untersummenbildung entspricht. Die erste Näherungslösung wird dann in die Ausgangsgleichung (Zeile 17) und Zustandsgleichung (Zeile 18) eingesetzt. Es folgt die Berechnung der zweiten Näherungslösung (Zeile 19-23) und die Berechnung der Ausgangsgleichung mit der zweiten Näherungslösung (Zeile 24).

```
1    #define DIMENSION 2
     ...
2    /* Runge/Kutta-Algorithmus zweiter Ordnung */
3    void integration(void)
4      {
5      float k1[DIMENSION];
6      float k1[DIMENSION];
7      float X_saved[DIMENSION];
8      int j;
9
10     calc_xdott();
11     for (j=0; j<DIMENSION; j++)
12       {  /* Berechnung der ersten Naeherung */
13       X_saved[j] = *pX[j],
14       k1[j] = *pXD[j];
15       *pX[j] = X_saved[j] + k1[j] * delta_t;
16       }
17     calc_y();
18     calc_xdott();
19     for (j=0; j<DIMENSION; j++)
20       {  /* Berechnung der zweiten Naeherung */
21       k2[j] = *pXD[j];
22       *pX[j] = X_saved[j] + 0.5 * (k1[j] + k2[j]) * delta_t;
23       }
```

```
24        calc_y();
25        }
          ...
26        void calc_xdott(void)
27        {
28        *pXD[Z1] =   1.000000 * *pX[Z2];
29        *pXD[Z1] =   0.500000 * *pX[Z1] + 1.500000 * *pX[Z2] + 0.500000 * pFixedData->In;
30        }
          ...
31        void calc_y( )
32        {
33        pDynamicData->V1 =    2.500000 * *pX[Z1] + 1.000000 * *pX[Z2];
34        pDynamicData->Out = pDynamicData->V1;
35        }
```

Der *Runge/Kutta-Algorithmus vierter Ordnung* unterscheidet sich in der viermaligen Auswertung der Ableitungsfunktion vom Runge/Kutta-Algorithmus zweiter Ordnung. Der grundsätzliche Aufbau des Algorithmus vierter Ordnung entspricht dem Aufbau des Algorithmus zweiter Ordnung, unterscheidet sich jedoch in der Gewichtung der Zwischenergebnisse und der Auswertung von zwei Näherungen im Inneren des betrachteten Intervalls $(t,\ t + h)$.

Es ergeben sich die folgenden Rechenschritte für den Algorithmus. Mit der Einführung der Variablen k_1 wird die Steigung zum Zeitpunkt t bezeichnet:

$$k_1 = f(x(t),\ u(t)) \tag{8.25}$$

Für die erste Steigung zum Zeitpunkt $t + \frac{1}{2}h$ erhält man unter Einführung der Variablen k_2:

$$k_2 = h \cdot f\left(x(t) + \frac{k_1}{2},\ u(t + \frac{h}{2})\right) \tag{8.26}$$

Für die zweite Steigung zum Zeitpunkt $t + \frac{1}{2}h$ erhält man unter Einführung der Variablen k_3:

$$k_3 = h \cdot f\left(x(t) + \frac{k_2}{2},\ u(t + \frac{h}{2})\right) \tag{8.27}$$

Für die Steigung zum Zeitpunkt $t + h$ ergibt sich:

$$k_4 = h \cdot f\big(x(t) + k_3,\ u(t + h)\big) \tag{8.28}$$

Somit ergibt sich für die Näherung vierter Ordnung:

$$x(t + h) = \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{8.29}$$

Für den Runge/Kutta-Algorithmus vierter Ordnung existieren weitere Lösungen, die die Zeitpunkte und die Gewichtung der errechneten Näherungen variieren. Beispielsweise werden bei der Newton-Cotes-Quadraturformel die Funktionswerte jeweils nach einem Drittel des Intervalls ermittelt und die Stützpunkte im Inneren des Intervalls stärker gewichtet. Für diese Lösung ergibt sich die folgende Näherung vierter Ordnung:

$$x(t + h) = \frac{h}{8}\left(k_1 + 3k_2 + 3k_3 + k_4\right) \tag{8.30}$$

Der Runge/Kutta-Algorithmus vierter Ordnung erfordert im Vergleich zum Algorithmus zweiter Ordnung in etwa den doppelten Rechenaufwand. Die Fehlerordnung des Algorithmus vierter Ordnung besitzt die Fehlerordnung $o(h^5)$. Auch hier existieren Probleme bei der Berechnung, da die Eingangswerte zwischen den Abtastzeitpunkten nicht vorliegen. Verwendet man statt dessen die Eingangswerte vom Beginn des Intervalls, steigt die Ungenauigkeit des Verfahrens.

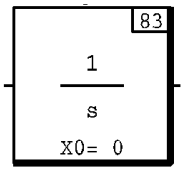


Bild 8-40: Darstellung des Integrator Blocks in MATRIX$_X$™

Ein Integrator Block besitzt für jeden Ausgang einen Eingang, einen Parameter Ordnung, Verstärkung und einen Vektor V für die Definition der Anfangszustände des Integrators. Die Übertragungsfunktion des Integrator Blocks lautet:

$$Y(s) = \frac{k}{s^n}\, U(s)\,; \quad \dot{y}(0) = V_1,\ \ddot{y}(0) = V_2,\ \ldots,\ \overset{(n)}{y}(0) = V_n;\ k \in \mathbb{R};\ n \in \mathbb{N} \qquad (8.31)$$

Neben diesen Parametern kann ein Integrator auch mit einem Reseteingang und n Zustandseingängen versehen werden. Ein weiterer Parameter bestimmt, ob ein Reset nur bei steigender oder bei steigender und fallender Flanke des Signals am Reseteingang ausgelöst werden soll. Wird ein Reset durchgeführt, so werden die internen Zustände des Integrator Blocks mit den an den Zustandseingängen anliegenden Werten belegt. Auch bei diesem Block ist es möglich, eine beliebige Anzahl an Ausgängen zu wählen. Diese besitzen allerdings dieselbe Ordnung und teilen sich den gleichen Reseteingang.

Auch der Integrator Block wird in CDIF im Zustandsraum repräsentiert. Ein Integrator mit der Verstärkung 3 und der Ordnung 2 besitzt somit die folgende Übertragungsfunktion, bzw. Darstellung im Zustandsraum:

$$G(s) = \frac{3}{s^2} \qquad \begin{aligned} \dot{\underline{x}} &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \underline{x} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \\ y &= \begin{bmatrix} 3 & 0 \end{bmatrix} \underline{x} + 0 \cdot u \end{aligned} \qquad (8.32)$$

Der Aufbau des C-Codes erfolgt nach dem im letzten Abschnitt vorgestellten Algorithmus für die Abbildung des Zustandsraums.

8.2.3 Code-Kopplung

Eine Kopplung des diskreten und kontinuierlichen Teilmodells besteht aus einem Datenaustausch zwischen den Systemen. Die Synchronisation dieses Datenaustauschs erfolgt zu festgelegten, äquidistanten Modellschritten (siehe Kapitel 7.2). Um die Datenunabhängigkeit, die für das Scheduling vorausgesetzt wurde, realisieren zu können, werden statische und dynamische Variablen für die gesamten Daten angelegt. Dieser als Double-Buffering bezeichnete Vorgang entspricht auf der Ebene des Modellcodes der Einführung von zwei Strukturen. Die statische Struktur beinhaltet die Werte des vorangegangenen Modellschritts und wird als pFixedData bezeichnet. Mit den in ihr enthaltenen Werten wird für den folgenden Modellschritt gerechnet. Die während eines Modellschritts neu berechneten Ergebnisse werden in einer dynamischen Struktur abgelegt und als pDynamicData bezeichnet. Am Ende des Modellschritts, d.h. nachdem alle Daten berechnet sind, wird diese Struktur als statische Struktur für den nächsten Modellschritt herangezogen.

Systembedingt werden die Daten anderer Teilsysteme also um einen Modellschritt verzögert übernommen. Würde man die Daten bereits während der Abarbeitung eines Modellschritts ändern, wäre

die Konsistenz der Daten nicht gewährleistet. Dieses Vorgehen weist Ähnlichkeiten zu abgetasteten Systemen auf, bei denen Ausgangswerte erst verzögert auf Eingangswerte reagieren können.

8.3 Ein-/Ausgabe-Code

Wie bereits in Kapitel 8.1 erwähnt, ist der Ein-/Ausgabe-Code sowohl vom Modell-Code als auch vom Scheduling-Code unabhängig. Der Ein-/Ausgabe-Code nimmt eine Umwandlung der logischen Modellvariablen in die Ansteuerung von physikalischen Signalen vor. Damit ist die Codekomponente für die Kommunikation mit der realen Umwelt verantwortlich. Der Benutzer konfiguriert den Ein-/Ausgabe-Code über eine grafische Oberfläche, die die logischen Modellvariablen und die physikalischen Signale in Verbindung bringen kann. Ein großer Teil des Ein-/Ausgabe-Codes ist statisch auf die jeweilige Hardwareumgebung angepaßt. Lediglich die aktuellen Variablen werden zusammen mit einem Verstärkungsfaktor oder einem zu verwendenden Protokoll in den Ein-/Ausgabe-Code eingefügt [Kock97].

Der Ein-/Ausgabe-Code wird als eigenständiges Modul zu jedem Modellschritt vom Scheduler aufgerufen. Nach der Umwandlung der dynamischen in die statischen Variablen (siehe Kapitel 8.2.3) werden auch die Werte der Ein-/Ausgabe-Variablen für den Modellschritt festgelegt. Eine dynamische Änderung während eines Modellschritts erfolgt aus den gleichen Gründen wie bei der Code-Kopplung nicht.

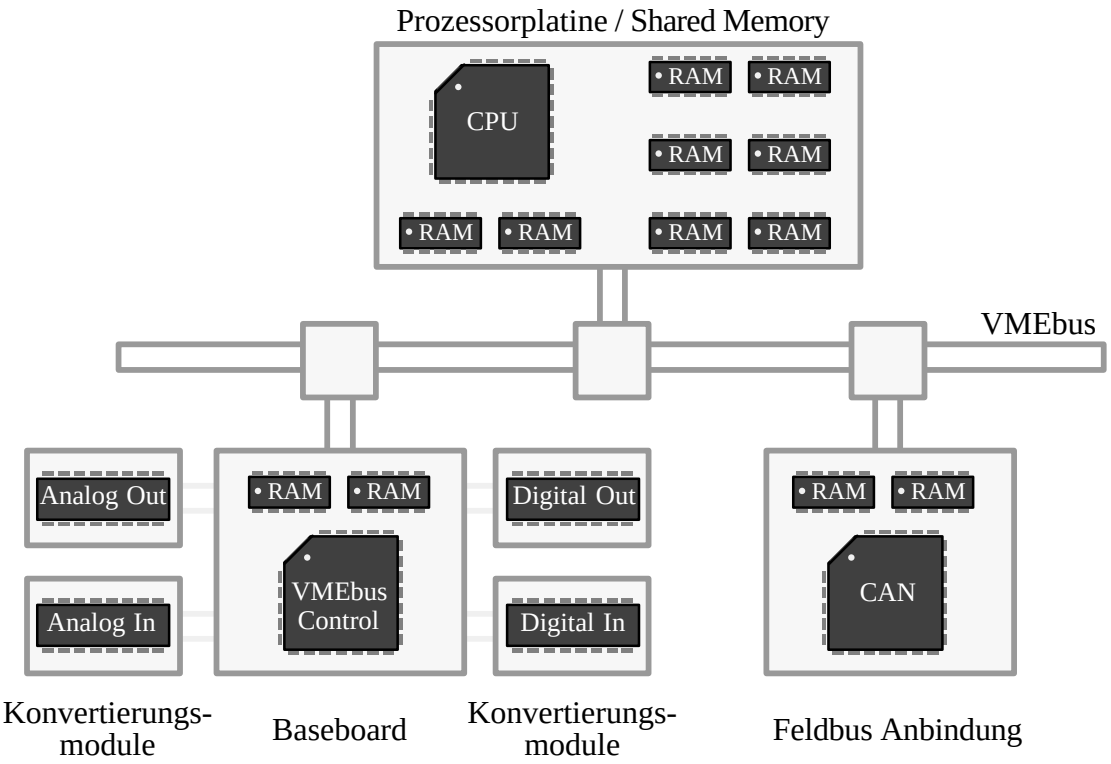


Bild 8-41: Schematischer Aufbau eines Rapid Prototyping Systems

Der gewählte modulare Aufbau erlaubt eine beliebige Kombination von Prozessorplatinen und unterschiedlicher Ein-/Ausgabekomponenten, die über einen zentralen Systembus (z.B. VMEbus) miteinander kommunizieren. Bild 8-41 zeigt den schematischen Aufbau eines Rapid Prototyping Systems. Die Abarbeitung des Modellcodes geschieht auf der Prozessorplatine, die einen Speicher-

bereich für die Zugriffe weiterer Komponenten freigeben kann. Die weiteren Systemkomponenten sind für die Anbindung an die Umgebung verantwortlich. Über ein Baseboard, das die Anbindung an den VMEbus vornimmt und TTL-Signale erzeugt oder verarbeitet, werden Konvertierungsmodule angesteuert. Die Konvertierungsmodule erzeugen die für die Umgebung erforderlichen Spannungen und Ströme und werden für den Anwendungsfall konfiguriert. Eine weitere Systemkomponente stellt die Feldbus Anbindung dar, die über konfigurierbare Mikrocontrollerplatinen vorgenommen wird [Kloe95]. Durch den modularen Aufbau des Systems können weitere Systemkomponenten wie beispielsweise Ethernet- oder SCSI-Anbindungen vorgenommen werden.

Wegen der hohen Anzahl an unterschiedlichen Konfigurationsmöglichkeiten für die Ein-/Ausgabe umfaßt der Programmcode die folgenden Komponenten:

- Konfiguration und Initialisierung der Ein-/Ausgabekomponenten
- Initialisierung aller Speicherstrukturen der Modellvariablen
- Einlesen der Systemeingänge
- Schreiben der Systemausgänge
- Zurücksetzen der Ein-/Ausgabekomponenten am Ende der Bearbeitung

Der Inhalt der einzelnen Komponenten beinhaltet spezielle Funktionsaufrufe der Ein-/Ausgabekomponenten, die für die jeweilige Hardware unterschiedlich aufgebaut sind und dynamisch konfiguriert werden. Anhand der Struktur des Programmcodes für ein Baseboard und die zugehörigen Konvertierungsmodule wird im folgenden beispielhaft die Einbindung vorgestellt.

Das Baseboard übernimmt die Anbindung an den VMEbus. Dabei wird ein konfigurierbarer Bereich ab der Adresse ADDR_BASEBOARD in den Speicherbereich des VMEbus eingeblendet. In diesen Speicherbereich werden die auszugebenden Signale als logische Werte geschrieben und von den Konvertierungsmodulen in Signale umgewandelt. Jedes der (im Fall des Baseboards BBA201N vier) Konvertierungsmodulen besitzt seinen eigenen Speicherbereich. Die Adresse von Konvertierungsmodul 0 beginnt ab der Adresse ADDR_BASEBOARD und besitzt eine Länge von 200 Bytes. Die kleinste Adresse der weiteren Konvertierungsmodule beginnt jeweils zum Ende der größten Adresse des vorangehenden Konvertierungsmoduls. Somit kann jedes Konvertierungsmodul über seinen Speicherbereich einzeln angesprochen werden.

```
1   #define ADDR_BASEBOARD  (0xfaaaa000)                          /* Adresse Baseboards Typ A201N */
2   #define ADDR_MMODUL0   (UINT8 *)(ADDR_BASEBOARD)              /* Adresse Konvertierungsmodul Nr. 0 */
3   #define ADDR_MMODUL1   (UINT8 *)(ADDR_BASEBOARD + 0x200)      /* Adresse Konvertierungsmodul Nr. 1 */
4   #define ADDR_MMODUL2   (UINT8 *)(ADDR_BASEBOARD + 0x400)      /* Adresse Konvertierungsmodul Nr. 2 */
5   #define ADDR_MMODUL3   (UINT8 *)(ADDR_BASEBOARD + 0x600)      /* Adresse Konvertierungsmodul Nr. 3 */
```

Die Initialisierung der Konvertierungsmodule wird über eine Folge von Bitwerten vorgenommen, die in den Speicherbereich des jeweiligen Konvertierungsmoduls geschrieben werden. Mit den Initialisierungsdaten wird beispielsweise festgelegt, ob ein Modul ein Ausgang oder ein Eingang darstellen soll und wie die Ein-/Ausgangssignale verarbeitet (buffered, latched, etc.) werden sollen. Die Initialisierungsdaten werden mit einer for-Schleife in die niedrigsten 20 Byte des Speichers eines Konvertierungsmoduls geschrieben. Der Typ UINT8 stellt einen Unsigned Integer von 8 Bit Länge unter dem Echtzeitbetriebssystem VxWorks™ dar.

```
1   /* Initialisierungs-Sequenzen der Konvertierungsmodule */
2   UINT8 InitMMod0[20] = { 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00,
                            0xff, 0x00, 0xff, 0x80, 0xff, 0x80, 0xff, 0x00, 0xff, 0x00 };
3
```

```
4       /* Initialisierung von Konvertierungsmodul0 */
5       for ( i = 0; i < 20; i++ ) {
6         vmeaddr8 = ( ADDR_MMODUL0 ) + i;
7         *vmeaddr8 = InitMMod0[i];
8         }
```

Zu jedem Modellschritt werden die Ein-/ und Ausgänge gelesen, bzw. geschrieben. Um eine solche Aktion durchzuführen, wird die Funktion UpdateBBA201NInput, bzw. UpdateBBA201NOutput aufgerufen. Das Einlesen von Daten geschieht über das Auslesen einer Speicheradresse, die im Speicherbereich des Konvertierungsmoduls liegt. Der Speicherbereich wird über die Adresse ADDR_MMODUL0 + PADR angesprochen (Zeile 7). Für die digitale Eingabe (in der folgenden Funktion mit Konvertierungsmodul 0 bezeichnet) wird das von der Hardware erzeugte Byte über Bitverknüpfungen ausgelesen. Jedes Bit entspricht einem logischen Wert ’0’ oder ’1’ für ein Signal. In der untenstehenden Funktion ist das Auslesen der Variablen UP, DOWN und AUTO wiedergegeben (Zeilen 10-12). Diese entsprechen den Bitwerten BIT0, BIT1 und BIT2 des ausgelesenen Bytes.

```
1       void UpdateBBA201NInput ( void )
2         {
3         UINT8 PortData;
4         UINT8 *vmeaddr;  /* Pointer auf Adresse im VMEBus-Bereich */
5
6         /* >>>> Update der Eingaenge von Konvertierungsmodul0 <<<< */
7         vmeaddr = ( ADDR_MMODUL0 + PADR );  /* Adresse Konvertierungsmodul0 Port A */
8         PortData = *vmeaddr;
9
10        pDynamicData->UP = ( PortData & BIT0 ) ? 1 : 0 ;
11        pDynamicData->DOWN = ( PortData & BIT1 ) ? 1 : 0 ;
12        pDynamicData->AUTO = ( PortData & BIT2 ) ? 1 : 0 ;
13        }
```

Die digitale Ausgabe von Daten geschieht über das Schreiben eines Bytes in eine Speicherstelle, die im Speicherbereich des Konvertierungsmoduls liegt (in der folgenden Funktion mit Konvertierungsmodul 1 bezeichnet). Der Speicherbereich wird über die Adresse ADDR_MMODUL1 + PADR angesprochen (Zeile 7). Um die Bitwerte der Variablen PortData zu setzen, wird eine bitweise ODER-Verknüpfung von den Bitwerten durchgeführt, die den Variablen MOTOR_U und MOTOR_D entsprechen (Zeilen 10-11). Das aus den einzelnen Bitwerten zusammengesetzte Byte wird dann in den Speicherbereich des Konvertierungsmoduls geschrieben (Zeile 13).

```
1       void UpdateBBA201NOutput ( void )
2         {
3         UINT8 *vmeaddr;  /* Pointer auf Adresse im VMEBus-Bereich */
4         UINT8 PortData;
5
6         /* >>>> Update der Ausgaenge von Konvertierungsmodul1 <<<< */
7         vmeaddr = ( ADDR_MMODUL1 + PADR );  /* Adresse Konvertierungsmodul1 Port A */
8         PortData = 0;
9
10        if ( pDynamicData->MOTOR_U )    PortData = PortData | BIT0;
11        if ( pDynamicData->MOTOR_D )    PortData = PortData | BIT1;
12
13        *vmeaddr = PortData;
14        }
```

Eine Anpassung an weitere Hardwarekomponenten gestaltet sich wegen der Unabhängigkeit der drei Codekomponenten als unproblematisch. In weiteren Arbeiten [Laie95] wurde bereits gezeigt,

daß eine Anpassung des Codes an einen Mikroprozessor C167 von Siemens problemlos durchführbar ist.

8.4 Scheduler-Code

8.4.1 Ablauf eines Modellschritts

Wird der Software-Prototyp ausgeführt, ist der zeitliche Ablauf der Abarbeitung von Modell-Code und Ein-/Ausgabe-Code entscheidend. Für die Ausführung stehen mehrere Möglichkeiten zur Verfügung:

- *So schnell wie möglich* (engl. *as fast as possible*): Bei dieser Abarbeitung wird jeder Schritt sofort nach Beendigung des vorherigen Schritts ausgeführt.
- *Äquidistante Zeitintervalle*: Bei dieser Abarbeitung wird jedem Schritt eine vorher definierte Zeitspanne zur Abarbeitung eingeräumt. Wird die Zeitspanne überschritten, liegt ein Abarbeitungsfehler vor und die Echtzeitbedingungen können nicht mehr gewährleistet werden. Diese Art der Abarbeitung wird als ratenmonotone oder pseudoraten-basierte Abarbeitung bezeichnet.
- *Adaptive Zeitintervalle*: Bei dieser Abarbeitung wird einem Schritt zu Beginn eine definierte Zeitspanne zur Abarbeitung eingeräumt. Wird festgestellt, daß die Zeitspanne zu groß oder zu klein ist, wird eine Vergrößerung bzw. Verkleinerung durchgeführt.

Wie bereits in Kapitel 7.2.2 untersucht, bieten die äquidistanten Zeitintervalle für Rapid Prototyping die beste Möglichkeit der Abarbeitung, da einerseits Echtzeitbedingungen ohne zusätzlichen Aufwand überwacht werden können und andererseits eine Zeitbasis für die Abarbeitung von diskreten und kontinuierlichen Modellen vorliegt. Eine Abarbeitung nach dem Prinzip 'So schnell wie möglich' kann keine Zeitspanne für die Ausführung eines Schrittes garantieren. Somit wird beispielsweise die Abarbeitung eines kontinuierlichen Blocks zur Erzeugung eines Pulse Wave Signals nach dieser Ausführungsmöglichkeit keine definierte Länge aufweisen können. Eine Abarbeitung nach dieser Art eignet sich besser für rein zustandsbasierte Systeme. Allerdings ist auch hier zu beachten, daß Übergänge keine definierte Zeitspanne benötigen. Die Ausführung mit adaptiven Zeitintervallen hat den Nachteil, daß eine Überschreitung der Modellschrittzeit in einem Rapid Prototyping System nicht toleriert werden kann. Bei dieser Abarbeitungsart wird jedoch versucht, eine möglichst hohe Ausnutzung der Modellschrittzeit zu erreichen. Die Berücksichtigung des schlechtesten anzunehmenden Falls führt zu einer Abarbeitung mit äquidistanten Zeitintervallen.

8.4.2 Zeitverhältnisse und Synchronisation

Der Scheduler ist eine statische Komponente, die eine geringe Konfiguration benötigt (Art der Ausführung, Zeitbasis und Modellschrittzeit). Arbeitet man den Software-Prototypen so schnell wie möglich ab, so ist der Scheduler eine triviale Komponente, die hauptsächlich aus einer unendlich oft ausgeführten Schleife besteht. Der Aufbau eines Schedulers für ratenmonotones oder pseudoratenbasiertes Scheduling besteht hingegen aus zwei Codekomponenten. Die erste Komponente ist im folgenden in algorithmischer Darstellung wiedergegeben.

```
1    let CheckSemaphore = 1
2    let ModelSemaphore = 0
3    let ExecutionError = 0
4    sysAuxClkRateSet(ModelStepTime)
```

```
 5    sysAuxClkConnect(AuxTimer() )
 6    sysAuxClkEnable()
 7    while true do
 8      if CheckSemaphore == 0 then
 9        let CheckSemaphore = 1
10        let ModelSemaphore = 1
11        UpdateInputs()
12        ModelStep()
13        UpdateOutputs()
14        let ModelSemaphore = 0
15      else
16        if ExecutionError == 1 then
17          break
18        end if
19      end if
20    end while
```

Diese Komponente dient zur Ausführung eines Modellschritts und der Ausführung des Ein-/Ausgabe-Codes. Zuerst wird der Auxiliary-Timer auf die Modellschrittzeit (ModelStepTime) gesetzt (Zeile 4). Der Auxiliary-Timer wird mit der Funktion *AuxTimer()* verbunden (Zeile 5). Jedes Mal, wenn der Auxiliary-Timer einen Interrupt auslöst wird die Funktion *AuxTimer()* ausgeführt. Nach dem Starten des Auxiliary-Timers (Zeile 6) wird eine while-Schleife ohne Abbruchbedingung ausgeführt. Da weder die Variable CheckSemaphore gleich ’0’ ist (Zeile 8), noch die Variable ExecutionError gleich ’1’ ist (Zeile 16), wird die while-Schleife ohne Ausführung eines Befehls durchlaufen. Wird die Funktion *AuxTimer()* wegen des Auslösens des Auxiliary-Timers durchlaufen (siehe unten), ist die Variable ModelSemaphore gleich ’0’ und die Variable CheckSemaphore wird auf ’0’ gesetzt (Zeile 9, unten). Nach dieser Variablenänderung von CheckSemaphore werden in der obigen Komponente CheckSemaphore und ModelSemaphore auf ’1’ gesetzt (Zeilen 9-10) um anzuzeigen, daß die Modellabarbeitung begonnen hat. Nach der Aktualisierung der Eingänge (Zeile 11) wird die Abarbeitung eines Modellschritt durchgeführt (Zeile 12). Nach der Ausführung des Modell-Codes werden die Ausgänge aktualisiert (Zeile 13).

Wird die Ausführung des Modellschritts während der Abarbeitung von einem erneuten Auslösen eines Interrupts des Auxiliary-Timers unterbrochen, ist die Variable ModelSemaphore noch zu ’1’ gesetzt. Diese Verletzung der Modellschrittzeit wird von der Funktion *AuxTimer()* abgefangen, da ein erneuter Aufruf des Auxiliary-Timers nicht mehr erlaubt wird und die Variable ExecutionError auf ’1’ gesetzt wird (Zeilen 5-7, unten). Nach Verlassen der Funktion *AuxTimer()* wird der Modellschritt beendet und die Verarbeitung unterbrochen (Zeile 16-17).

```
 1    if CheckSemaphore == 0 then
 2      sysAuxClkDisable()
 3      let ExecutionError = 1
 4    else
 5      if ModelSemaphore == 1 then
 6        sysAuxClkDisable()
 7        let ExecutionError = 1
 8      else
 9        let CheckSemaphore = 0
10      end if
11    end if
```

Falls die Variable CheckSemaphore beim Eintritt in die Funktion *AuxTimer()* den Wert ’0’ besitzt, so konnte Zeile 9 der while-Schleife nicht abgearbeitet werden, da anderenfalls die Variable sofort auf ’1’ gesetzt wird. Dieser Fall kann eintreten, wenn eine sehr große Überlastung des Systems vorliegt. Auch in diesem Fall wird die Verarbeitung beendet (Zeile 16-17).

9 Ergebnisse

9.1 Entwurfsumgebung

Zur Modellierung des Gesamtsystems wurde das Programm RPSTEP (Rapid Prototyping based on STEP and CDIF) entwickelt, das über einen eigenen grafischen Editor gemäß Kapitel 5.4 verfügt [Kühl97]. Mit dem Editor kann das Gesamtmodell in mehrere Teilmodelle zerlegt werden. Die Teilmodelle können die Modellierung eines CASE-Werkzeugs enthalten oder weitere, hierarchisch aufgebaute Teilmodelle beinhalten. Bild 9-1 zeigt die Oberfläche, auf der zwei heterogene Teilmodelle zu sehen sind.

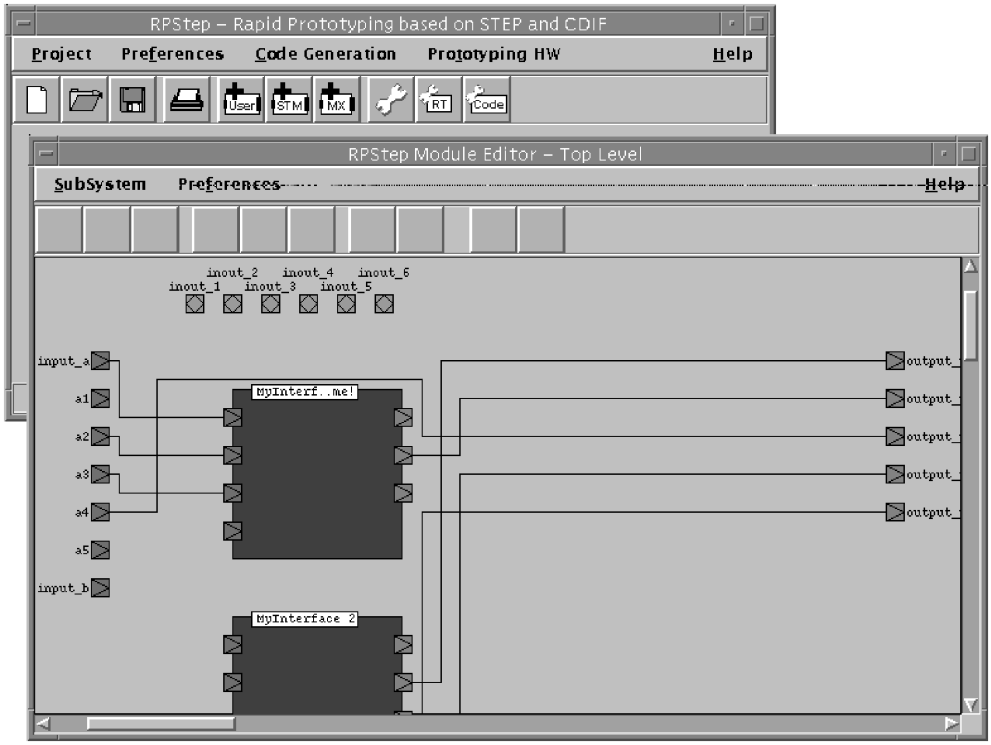


Bild 9-1: Rapid Prototyping Gesamtsystem-Modellierung RPSTEP

Das Programm MERPS (Multiprozessor Echtzeit Rapid Prototyping System) dient der Konfiguration des Software-Prototypen. Bild 9-2 zeigt die Hauptkomponente der grafischen Benutzerschnittstelle von MERPS (oben), mit der sämtliche Menüpunkte aufgerufen werden können. Über die Menüpunkte können Projekte angelegt, geladen und gespeichert werden, CDIF Modelle geladen, editiert und gespeichert werden, die Auswahl und Konfiguration der Ein-/Ausgabehardware vorgenommen werden und die Codekomponenten erzeugt sowie die zugehörigen Echtzeitbedingungen formuliert werden. Bild 9-2 zeigt im unteren Teil die Auswahl der zur Verfügung stehenden Hardwarekomponenten für ein M-Modul Baseboard der Firma men, die in Form einer Liste aufgeführt

sind. Ein M-Modul Baseboard stellt eine Trägerplatine für jeweils vier standardisierte Ein-/Ausgabeplatinen dar, die als M-Module bezeichnet werden und ist mit dem VMEbus verbunden. Dabei können mehrere M-Module, beispielsweise digitale Ein-/Ausgabe, analoge Ein-/Ausgabe, ein Frequenzgenerator, ein CAN-Bus Interface oder ein Centronics-Interface, eingebunden werden. Um mehrere Baseboards gleichzeitig nutzen zu können, wird eine unterschiedliche VMEbus Adresse für jedes Baseboard gewählt.

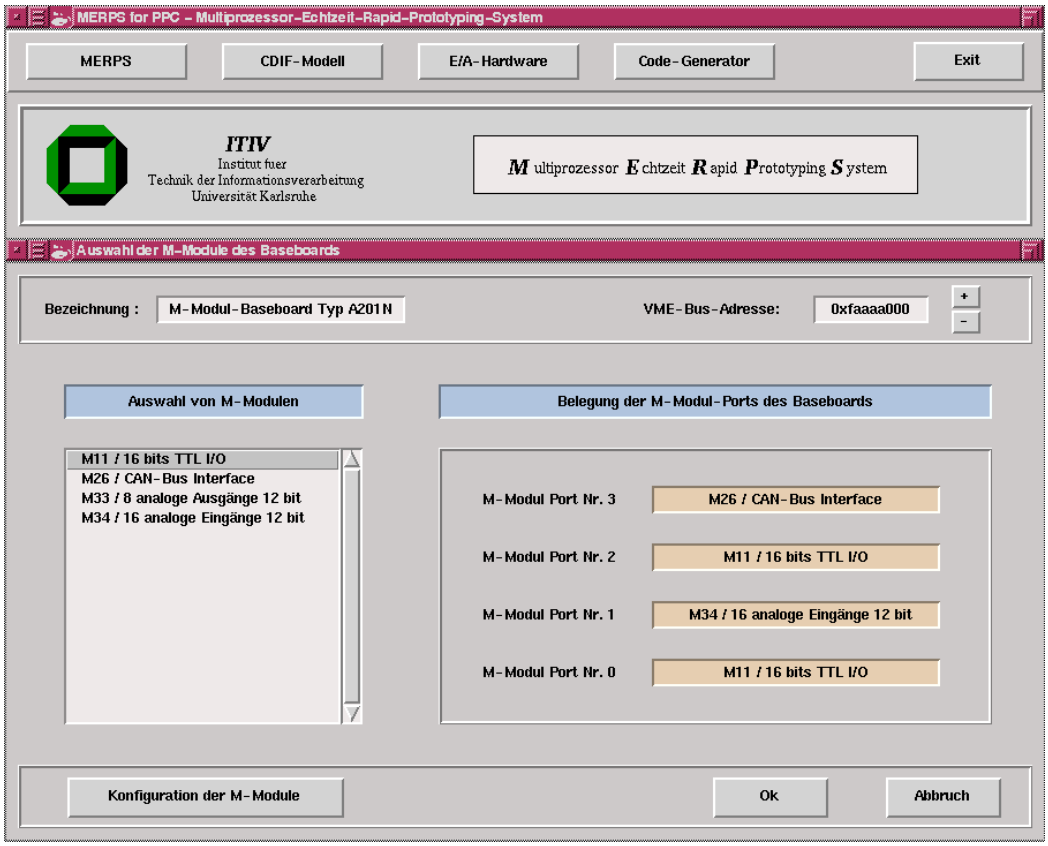


Bild 9-2: Rapid Prototyping Konfigurationsoberfläche MERPS

Die Menüpunkte zur Konfiguration der Ein-/Ausgabehardware sind dynamisch aufgebaut und stehen somit nur für Hardware zur Verfügung, die ausgewählt wurde. In Bild 9-3 ist die Konfiguration für ein digitales Ein-/Ausgabemodul M11 der Firma men gezeigt. Mit dieser Hardwarekomponente ist es möglich, 2 x 8 digitale Kanäle wahlweise als Ein- oder Ausgang zu benutzen. Die Zuordnung der Modellvariablen zu den physikalischen Signalen erfolgt durch Auswahl der Modellvariablen, die in Listenform wiedergegeben sind und den Ein-/Ausgängen frei zugeordnet werden können.

Zusätzlich zur Zuordnung der Modellvariablen zu den physikalischen Signalen kann die Auswahl der Betriebsart mit Hilfe einer Oberfläche (Bild 9-4) durchgeführt werden. Dabei ist es beispielsweise möglich, eine Verwendung als Ein-/Ausgabe vorzugeben oder die Ein-/Ausgabesignale in unterschiedlicher Weise zu puffern.

Für die Konfiguration der analogen Eingänge wird das M-Modul M34 verwendet. Auch bei dieser Oberfläche (Bild 9-5) findet die Zuordnung der Modellvariablen zu den physikalischen Signalen in derselben Art wie für die digitale Ein-/Ausgabe statt. Zusätzlich wird bei der Konfiguration der analogen Eingänge der Spannungsbereich eingegeben. Die erreichbare Auflösung wird in einem Feld hinter dem jeweiligen Signal angegeben.

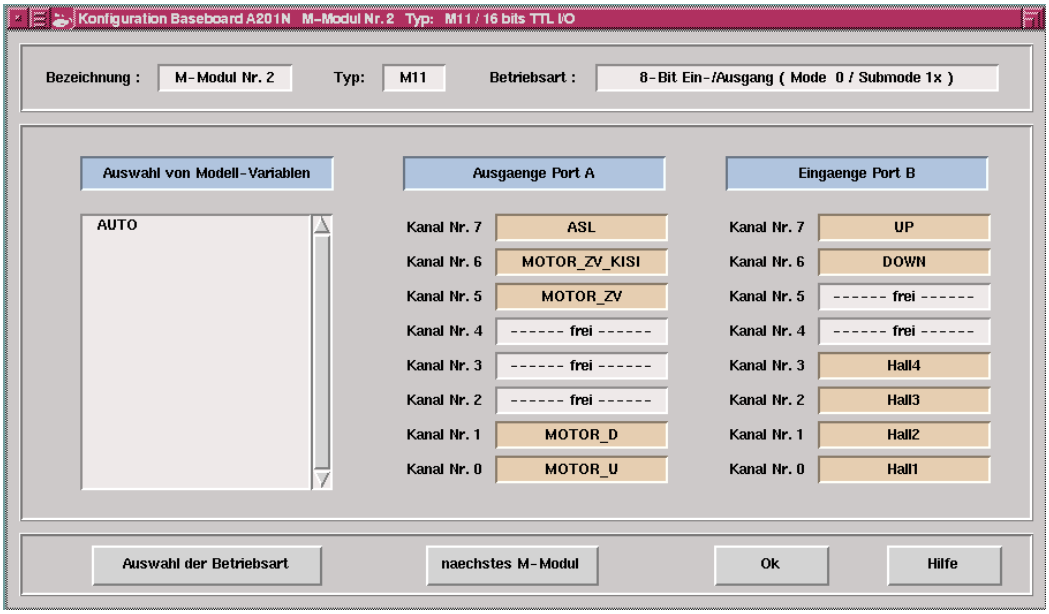


Bild 9-3: Konfiguration der Schnittstellenkarte M11

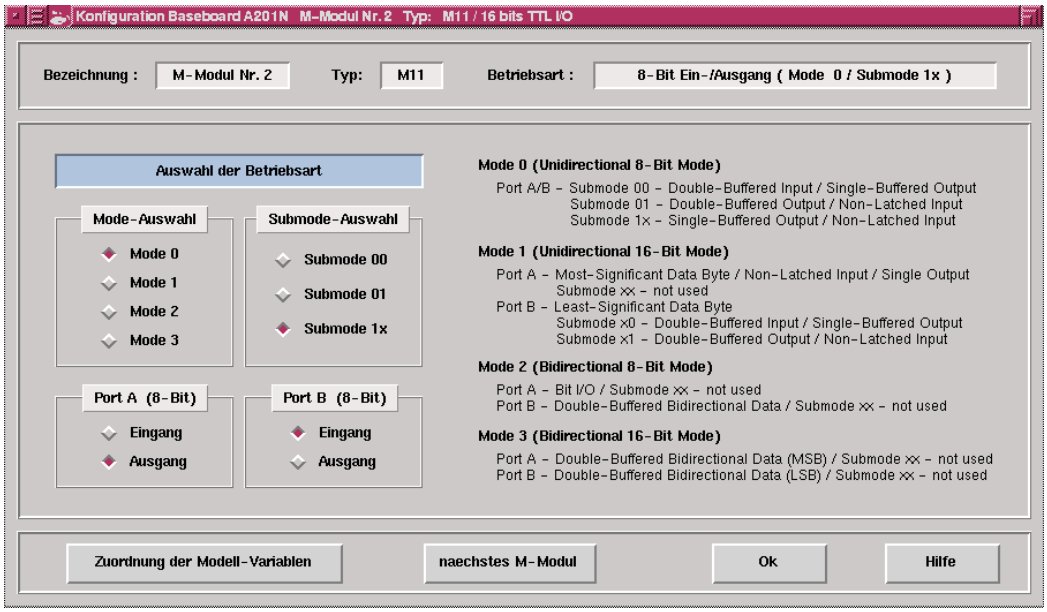


Bild 9-4: Konfiguration der Schnittstellenkarte M11

Die am Markt erhältlichen Hardwarekomponenten decken jedoch nur einen Teil derjenigen Anforderungen ab, die in der Industrie benötigt werden. Beispielsweise werden in industriellen Projekten mehrere Hundert Ein-/Ausgabesignale benötigt oder hohe Anforderungen an Zeitverhältnisse bzw. Signalstärken vorgegeben. Um diese speziellen Hardwarebedürfnisse ebenfalls zu unterstützen, wurde eine konfigurierbare AD/DA-Wandlerplatine entwickelt [Spit95], über die jeweils 32 analoge und digitale Ein-/Ausgänge angesteuert werden können. Auch für diese Hardwarekomponente wurden Konfigurationsoberflächen erstellt [AlHS95], die in derselben Art wie bei den oben erwähnten M-Modulen arbeiten. Im Gegensatz zu den M-Modulen können hierbei zusätzlich Schwellwerte (obere und untere Schranke), Eingangswiderstände, die Art der Messung (Strom-, Spannungs- oder Widerstandsmessung) und die Entprellung eingestellt werden. Die Ausführung von zeitkritischen Aufgaben kann direkt auf einem Siemens C167 Mikrocontroller erfolgen.

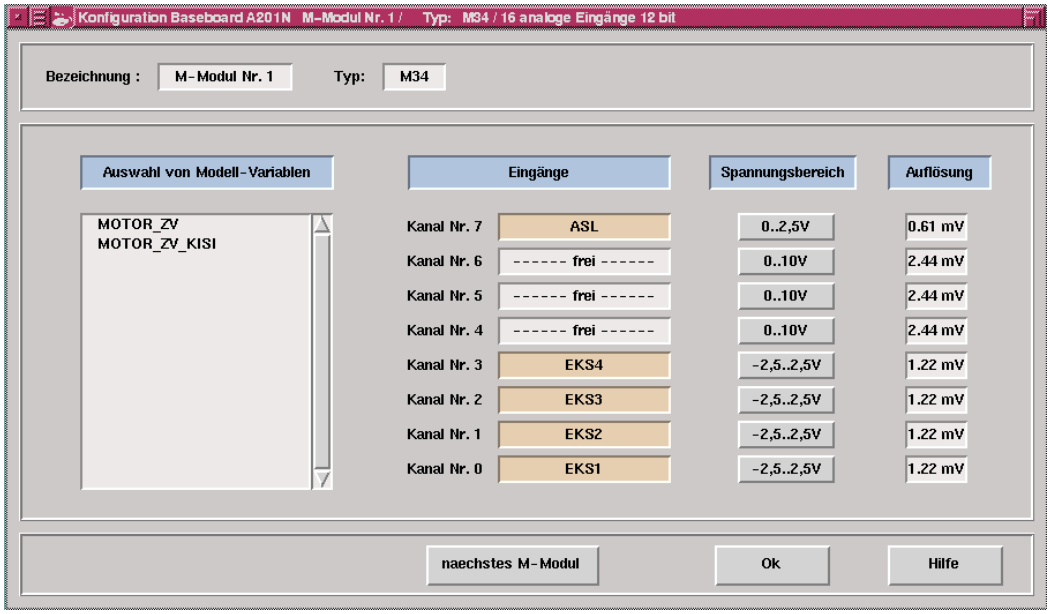


Bild 9-5: Konfiguration der Schnittstelle M34

Für die Konfiguration des Software-Prototypen bezüglich Schedulingkonfiguration, Zeitbasis und Integrationsalgorithmus dient die Oberfläche aus Bild 9-6. Mit ihr kann festgelegt werden, welche der drei Abarbeitungsarten aus Kapitel 7 eingesetzt werden soll. Die Zeitbasis wird in Ticks pro Sekunde eingestellt und repräsentiert die Frequenz, mit der der eingesetzte Auxiliary-Timer betrieben wird. Für eine Abarbeitung mit 100 µs muß also ein Wert von 10 000 Ticks pro Sekunde eingegeben werden. Die Modellschrittzeit kann dann als Vielfache der gewählten Zeitbasis eingegeben werden. Auch der Integrationsalgorithmus kann hier ausgewählt werden. Dazu stehen die Integrationsalgorithmen Euler, Runge-Kutta-2 und Runge-Kutta-4 zur Verfügung. Die Einbindung weiterer Integrationsalgorithmen ist modular möglich.

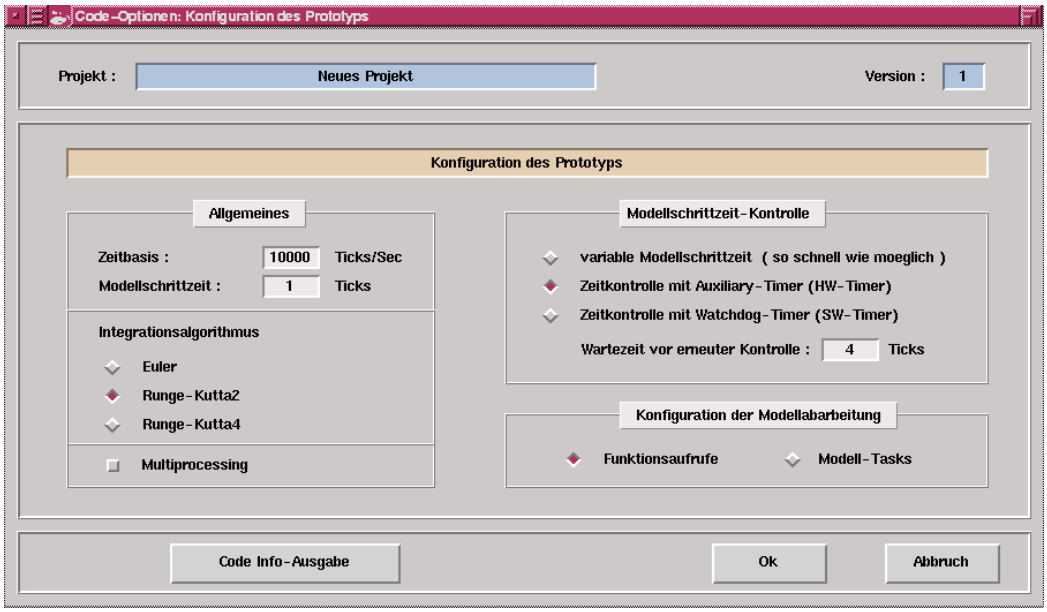


Bild 9-6: Schedulingkonfiguration des Prototypen

Die erstellten Oberflächen gewährleisten einen schnellen und komfortablen Modellierungs-, Konfigurations- und Rekonfigurationsprozeß. Durch den modularen Aufbau der Komponenten ist eine Verwendung auch mit zukünftigen Erweiterungen möglich und kann auf den Bereich Hardware-in-the-Loop erweitert werden.

9.2 Codegenerierung

Um eine Aussage über die Leistungsfähigkeit der entwickelten Codegeneratoren treffen zu können, wurden Tests sowohl mit diskreten und kontinuierlichen Modellen als auch mit heterogenen Modellen vorgenommen und mit der Codeerzeugung der Programme STATEMATE™ und MATRIX$_X$™ verglichen. Die Ergebnisse werden in den nächsten Teilkapiteln vorgestellt. Für die Übersetzung der Modelldaten in CDIF und von CDIF in Programmcode wurde jeweils eine SUN SPARCstation Ultra mit 512 MB Hauptspeicher unter Solaris 2.6 eingesetzt.

9.2.1 Diskreter Bereich

Die Untersuchung der Leistungsfähigkeit wurde im diskreten Bereich mit Hilfe eines kaskadierten Arbiters (dt. *Schiedsrichter*) vorgenommen. Bild 9-7 zeigt die Modellierung einer Arbiterzelle, die aus fünf Zuständen und sieben Zustandsübergängen besteht. An die Arbiterzelle können zwei Geräte angeschlossen werden, die mit Hilfe der Signale ReqL (Request Left) und ReqR (Request Right) einen Request anmelden können. Mit den Arbitersignalen GoL (Go Left) oder GoR (Go Right) bekommt genau eines der beiden Geräte den Zuschlag.

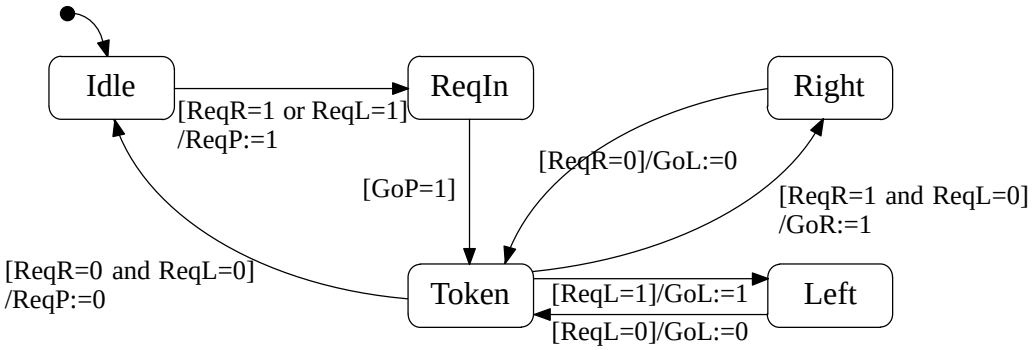


Bild 9-7: Statechart eines kaskadierbaren Arbiters

Um mehr als zwei Geräte an einen Arbiter anschließen zu können, wird eine entsprechende Anzahl von Arbitern in Form einer Kette kaskadiert (Bild 9-8). Hierbei wird ein Ausgang des Arbiters als Eingangssignal des nächsten Arbiters benutzt. Das eingehende Signal wird als GoP (Go Parent) bezeichnet. Mit diesem Aufbau kann eine beliebig hohe Anzahl von Geräten angesteuert werden.

Die Anzahl der Entities und Relationen in CDIF, die für die Darstellung eines Arbiters benötigt wird, ist linear von der Anzahl der Arbiter abhängig. Zehn Arbiter benötigen beispielsweise 3 385 Entities und 7 131 Relationen. Bild 9-9 zeigt die Anzahl der Entities und Relationen für die Abbildung einer Arbiterkette bestehend aus n Arbitern.

Durch die hohe Anzahl an Entities und Relationen sowie durch die Benutzung einer ASCII basierten Datenhaltung wächst die Filegröße stärker als bei aktuellen CASE-Werkzeugen. Bild 9-10 zeigt die Filegrößen des CASE-Werkzeugs STATEMATE™ im Vergleich zur Filegröße einer CDIF Beschreibung. Wird eine binäre Codierung verwendet, fällt die Filegröße etwas kleiner im Vergleich

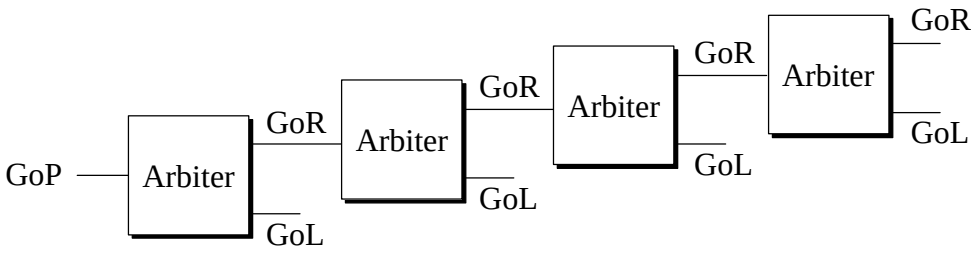


Bild 9-8: Kaskadierte Arbiterkette

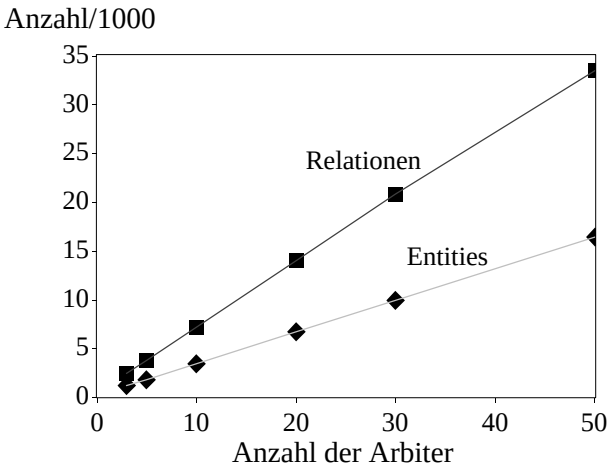


Bild 9-9: Anzahl der Entities und Relationen für die Abbildung einer Arbiterkette

zur Filegröße von STATEMATE ™ aus. Da augenblicklich keine binäre Codierung für CDIF existiert, mußte eine grobe Schätzung mit Hilfe eines Komprimierungswerkzeugs (”gzip“) durchgeführt werden. Es ist zu erwarten, daß die endgültigen Ergebnisse besser ausfallen werden, da eine binäre Codierung eine Komprimierung mit Kenntnis der Semantik ermöglicht.

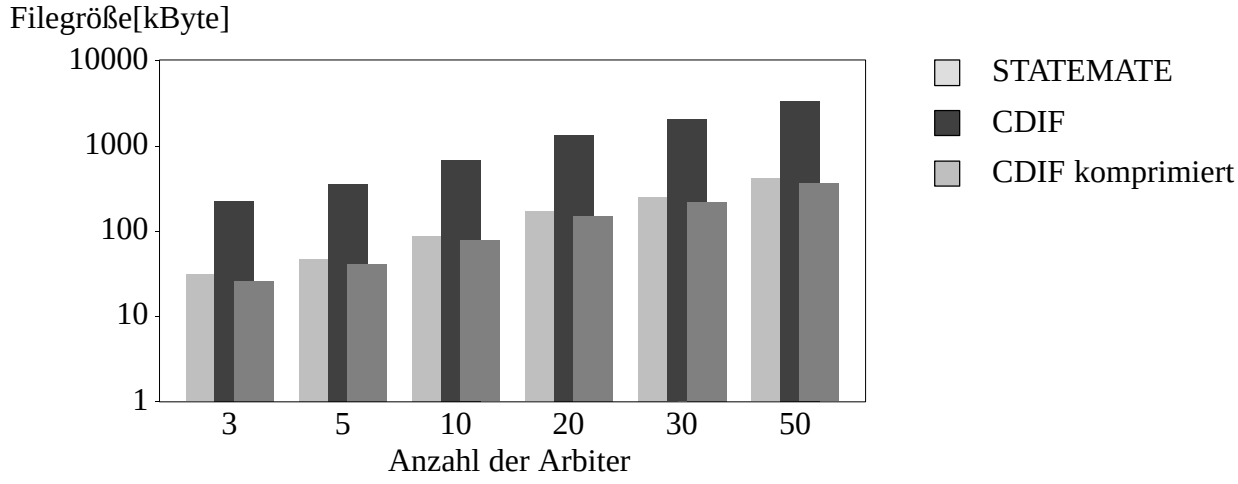


Bild 9-10: Filegröße einer Arbiterkette mit STATEMATE ™ und CDIF-Datenhaltung

Die Übersetzungszeiten des Compilers STM2CDIF, der Statecharts in eine CDIF-Beschreibung übersetzt, liegen bei sehr großen Modellen im Bereich von wenigen Minuten. Für die Übersetzung einer Arbiterkette bestehend aus 50 Arbitern wird beispielsweise weniger als drei Minuten benötigt. Beim praktischen Einsatz ist zu erwarten, daß eher eine Vielzahl von kleineren Modellen übersetzt

wird. Für diesen Fall kann mit Zeiten unter einer Minute gerechnet werden. Bild 9-11 zeigt die Übersetzungszeiten für die Arbiterkette.

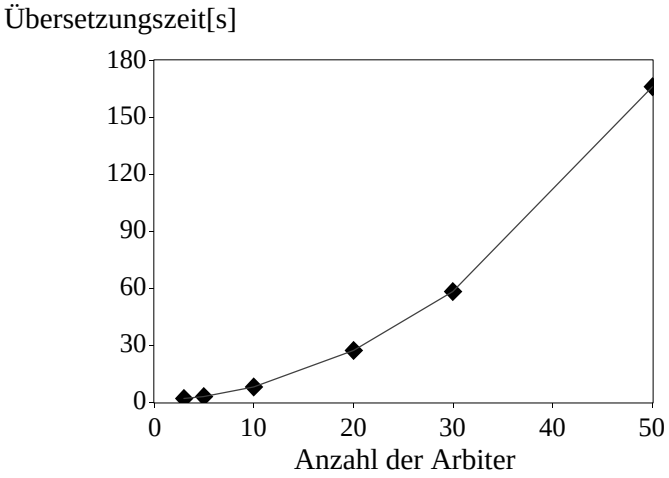


Bild 9-11: Übersetzungszeiten des Compilers STM2CDIF

Der benötigte Speicherplatz zur Übersetzung der Statecharts in CDIF bewegt sich bei allen Modellen unterhalb von 50 kByte und kann damit auch für große Modelle vernachlässigt werden.

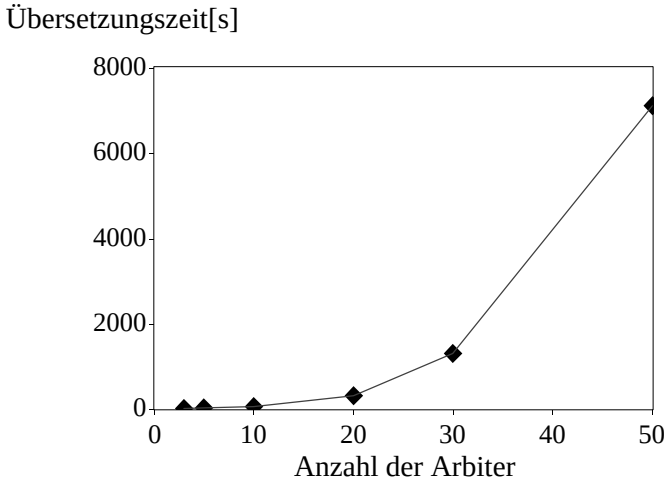


Bild 9-12: Übersetzungszeiten des Compilers CDIF2C

Die Übersetzungszeiten des Compilers CDIF2C, der die erzeugte CDIF-Beschreibung in C-Code umwandelt, wachsen für große Modelle stark an. Beispielsweise benötigt die Transformation einer Arbiterkette von 30 Arbitern bereits 23 Minuten. Bild 9-12 zeigt die Übersetzungszeiten für die Arbiterkette. Die hohen Übersetzungszeiten hängen hauptsächlich von der aufwendigen Analyse hierarchischer und paralleler Strukturen der Statecharts ab. Messungen haben gezeigt, daß die Zugriffszeiten auf die Hashtables der Compiler stark anwachsen und damit die Übersetzungszeiten ebenfalls ansteigen. Der benötigte Speicherplatz wächst wegen der Verwendung von Hashtables ebenfalls an. Selbst für eine Arbiterkette von 50 Arbitern wird jedoch nur ein Speicherplatz von 3,4 MByte benötigt und ist damit vernachlässigbar.

Für die Messung der Leistungsfähigkeit des erzeugten Echtzeitcodes wurde als Recheneinheit ein Motorola MVME-2604 Board eingesetzt, das mit einem PowerPC Prozessor 604 mit 200 MHz und 32 MByte Speicher bestückt ist. Als Echtzeitbetriebssystem wurde VxWorks™ der Firma WindRiver Inc. in der Version 5.3 eingesetzt. Diese Konfiguration wurde für alle Messungen verwendet.

Die Messungen der minimal möglichen Modellschrittzeiten wurde ohne Timerunterstützung (engl. *as fast as possible*, abgek. AFAP), ratenmonoton (abgek. RM) mit Hilfe eines Watchdog-Timers und pseudoraten-basiert (abgek. PR) mit Hilfe eines Auxiliary-Timers durchgeführt.

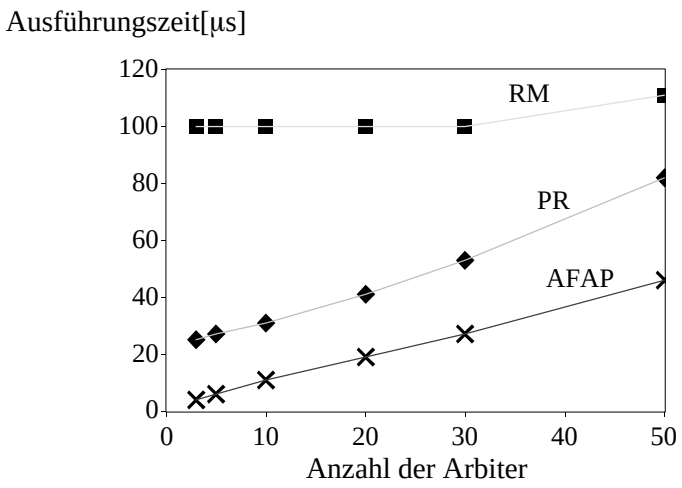


Bild 9-13: Minimale Modellschrittzeit des Echtzeitcodes für eine Arbiterkette

Da linear zur Anzahl der Arbiter auch die Anzahl der zu berechnenden Systemzustände steigt, steigt die minimale Modellschrittzeit ebenfalls linear an. Die Messungen, die in Bild 9-13 wiedergegeben sind, verdeutlichen dieses Verhalten. Die Modellschrittzeiten für den Echtzeitcode ohne Verwendung eines Timers fallen insbesondere bei kleinen Modellen ins Gewicht. Hierbei sei nochmals angemerkt, daß bei der Verwendung von diskreten und kontinuierlichen Modellen eine Zeitmessung erforderlich ist (siehe Kapitel 7.2.2). Ebenso ist zu sehen, daß die Modellschrittzeiten bei Verwendung einer ratenmonotonen Abarbeitung größer als die pseudoraten-basierte Abarbeitung ausfällt. Die ratenmonotone Abarbeitung wurde bei den Messungen durch die maximale Wiederholrate des eingesetzten Watchdog–Timers begrenzt, die bei 100 µs liegt.

Im Vergleich zu dem Code, der von STATEMATE ™ erzeugt wird, wird die Abarbeitung deutlich schneller durchgeführt. Dies ist hauptsächlich auf die starke Nutzung von Bibliotheken und den damit verbundenen Aufruf mehrerer Funktionen des STATEMATE ™-Codes zurückzuführen. Bei der Analyse des Codes fiel auf, daß einige Funktionen nur eine weitere Funktion aufriefen. Der durch diese Punkte entstehende Mehraufwand erklärt den deutlichen Performanz-Unterschied der beiden Echtzeitcodes. Für die Messungen wurde die Version STATEMATE ™ Magnum 1.2.4 eingesetzt. Da der erzeugte Code nicht echtzeitfähig ist, wurde der hochoptimierte Scheduler aus Kapitel 8.4 verwendet und an den von STATEMATE ™ erzeugten Code angepaßt.

Bei der Erzeugung des Echtzeitcodes werden die zusätzlichen Arbiter durch neue Zustände dargestellt, die als Funktionen mit if-then-else Abfragen realisiert sind. Wie aus Bild 9-15 zu erkennen ist, wächst der Code erwartungsgemäß linear mit steigenden Zustandszahlen. Die Größe der unterschiedlichen Schedulingkomponenten (AFAP, RM, PR) fallen gegenüber der Modellcodegröße sehr klein aus. Aus diesem Grund ist in Bild 9-15 nur eine Codegröße für alle Abarbeitungsarten dargestellt.

Im Vergleich zu dem erzeugten Code von STATEMATE ™ fällt die geringe Größe des eigenen Echtzeit-Codes auf. Auch hier ist als Hauptgrund die Verwendung großer Bibliotheken auszumachen, die statisch zu dem erzeugten Code hinzugelinkt werden.

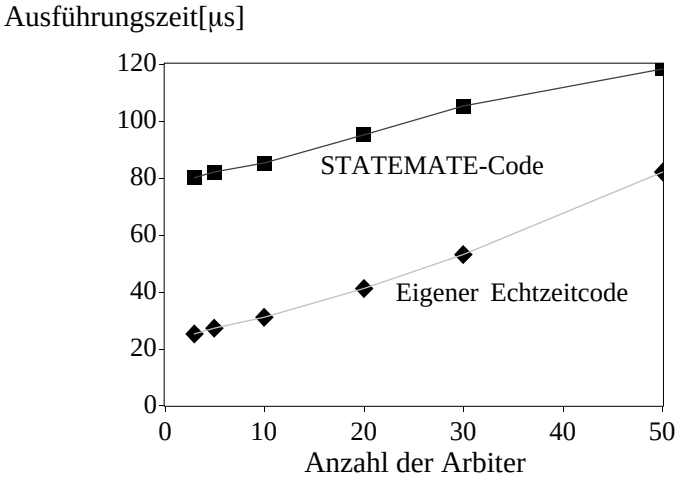


Bild 9-14: Modellschrittzeit im Vergleich zu STATEMATE ™

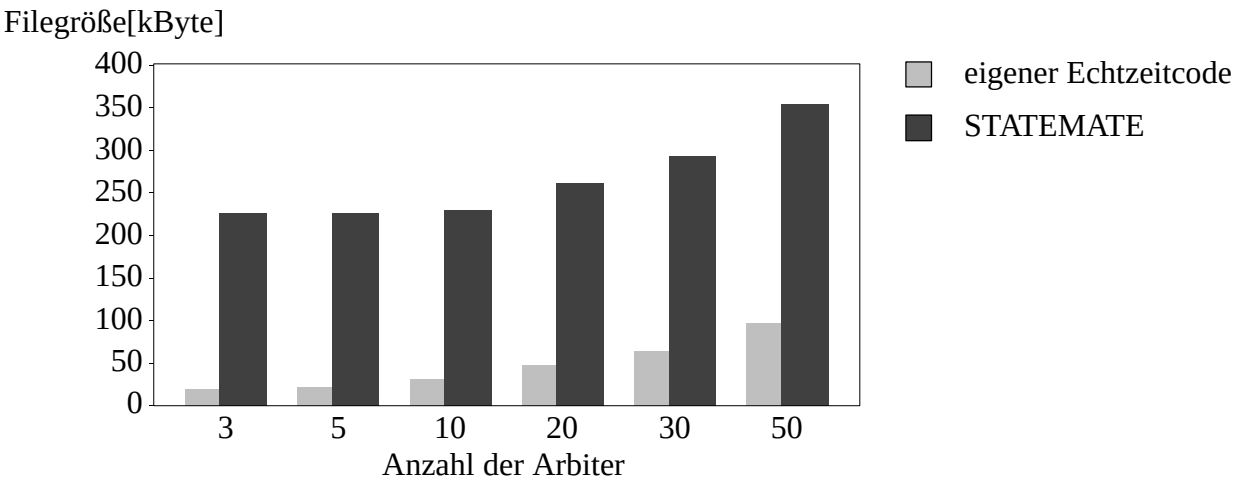


Bild 9-15: Codegröße im Vergleich zu STATEMATE ™

9.2.2 Kontinuierlicher Bereich

Die Untersuchungen im kontinuierlichen Bereich wurden anhand von FIR-Filtern unterschiedlicher Ordnung erstellt. Das mit $MATRIX_X$ ™ erstellte Modell wurde in CDIF abgebildet und aus der CDIF-Beschreibung C-Code erzeugt. Die Modellierung einer einzelnen Filterordnung (engl. *tap*) erfolgte unter Verwendung eines Multiplizierers, Addierers und eines Verzögerungsglieds. Durch die Aneinanderreihung von einzelnen Filterordnungen nach Bild 9-16 können FIR-Filter beliebiger Ordnung modelliert werden.

Die Anzahl der Entities und Relationen, die für die Darstellung eines FIR-Filters in CDIF benötigt wird, ist linear von der Ordnung des Filters abhängig. Für die Darstellung eines Filters 50. Ordnung werden beispielsweise 7 577 Entities und 17 212 Relationen benötigt. Bild 9-17 stellt die Anzahl der Entities und Relationen für einen FIR-Filter n-ter Ordnung dar.

Die hohe Anzahl an Entities und Relationen ist auch an der Größe der erzeugten CDIF-Files erkennbar. Bild 9-18 zeigt die Filegröße eines FIR-Filters des CASE-Werkzeugs $MATRIX_X$ ™ im Vergleich zur Filegröße einer CDIF Beschreibung. Durch die hohe Zahl der Entities und Relationen fällt auch die Filegröße höher aus. Der Größenunterschied resultiert aus der kompakten Darstellung

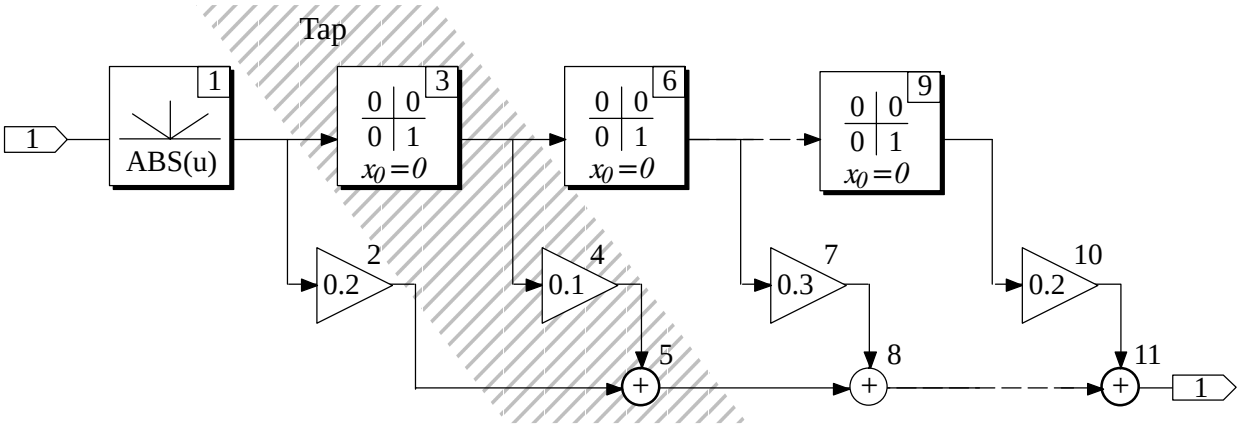


Bild 9-16: Modellierung eines FIR-Filters

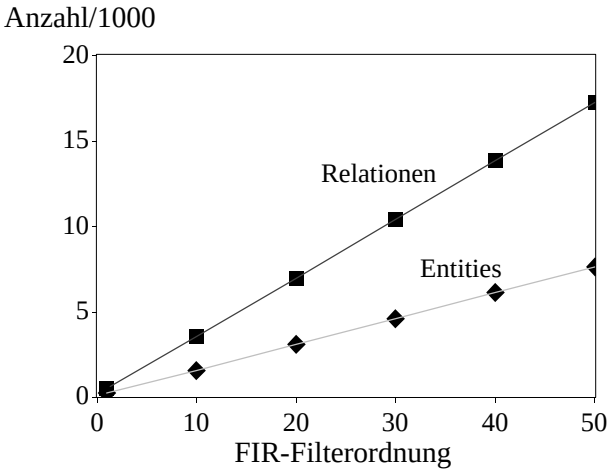


Bild 9-17: Anzahl der Entities und Relationen für die Abbildung eines FIR-Filters

der Blöcke in MATRIX$_X$™. Während in CDIF die komplette Semantik der Blöcke abgebildet werden muß, speichert die werkzeug-spezifische Datenhaltung von MATRIX$_X$™ lediglich Position, Blocktyp und Verbindungen. Im Vergleich zum diskreten Bereich fällt auf, daß auch die komprimierte Darstellung in CDIF wesentlich größer ausfällt. Der Größenunterschied liegt hauptsächlich darin begründet, daß die Semantik eines Zustands im diskreten Bereich wesentlich einfacher als die eines kontinuierlichen Blocks ist. Beispielsweise werden für die Darstellung eines Sinusblocks in CDIF 11 Entities und 20 Relationen benötigt. Für die Darstellung eines Zustands hingegen werden lediglich zwei Entities und zwei Relationen benötigt. Die absoluten Zahlen (Bild 9-18 im Vergleich zu Bild 9-9) fallen für den diskreten Bereich höher aus, da mehr Elemente für die Modellierung eines Arbiters als für die Modellierung eines FIR-Filters benötigt werden.

Die Übersetzungszeiten des Compilers MX2CDIF, der kontinuierliche Beschreibungen in MATRIX$_X$™ in eine CDIF-Beschreibung übersetzt, liegen bei sehr großen Modellen im Bereich von wenigen Sekunden. Für die Übersetzung eines FIR-Filters 50-ter Ordnung wird beispielsweise weniger als 25 Sekunden benötigt. Beim praktischen Einsatz ist auch hier zu erwarten, daß eher eine Vielzahl von kleineren Modellen übersetzt werden. Für diesen Fall kann mit vernachlässigbaren Übersetzungszeiten gerechnet werden. Bild 9-19 zeigt die Übersetzungszeiten für den FIR-Filter.

Der benötigte Speicherplatz ist für die Übersetzung jeder Filterordnung stets unterhalb von 200 kByte und kann deswegen ebenfalls vernachlässigt werden.

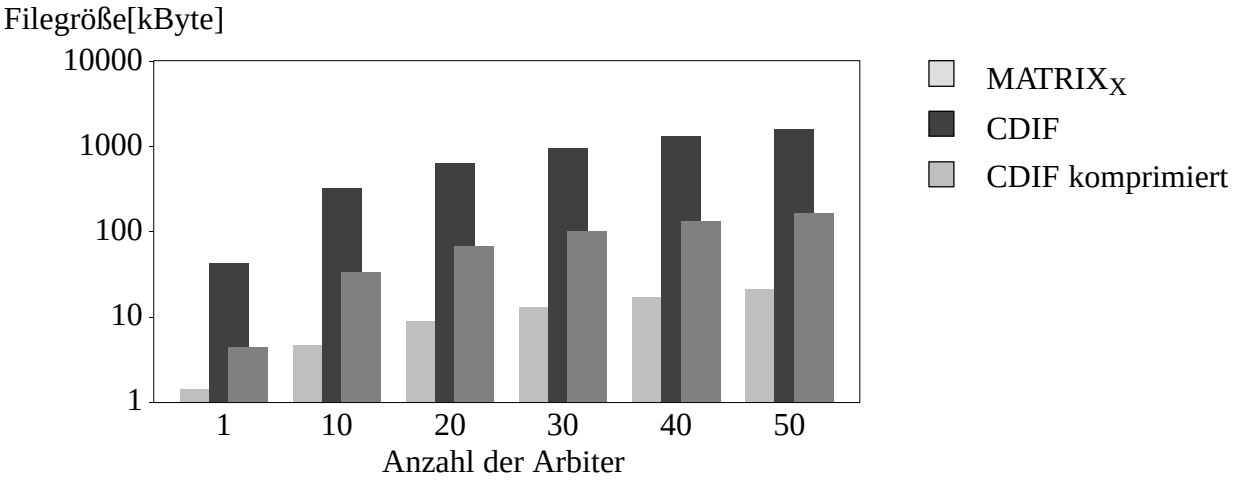


Bild 9-18: Filegröße einer FIR-Filter mit MATRIX$_X$™ und CDIF-Datenhaltung

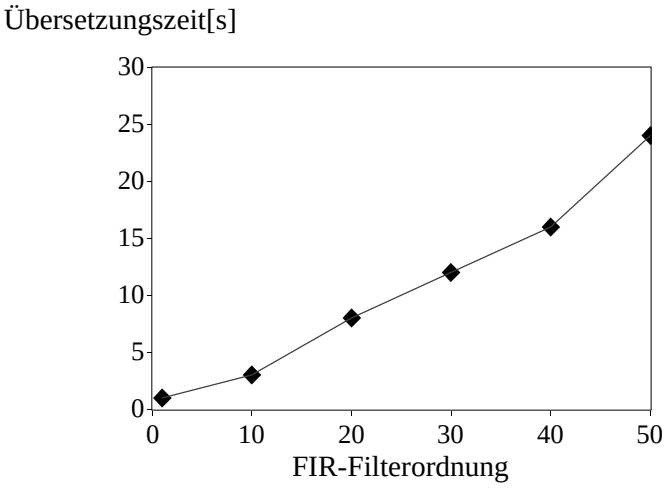


Bild 9-19: Übersetzungszeiten des Compilers MX2CDIF

Die Übersetzungszeiten des Compilers CDIF2C, der eine Übersetzung der CDIF-Beschreibung in C-Code vornimmt, steigt auch für den kontinuierlichen Bereich für große Modelle stark an. Die zur Erzeugung des C-Codes benötigte Analyse fällt aufwendiger aus als bei der Übersetzung von der kontinuierliche Modellierung in eine CDIF-Beschreibung. Bild 9-20 zeigt die Übersetzungszeiten des Compilers CDIF2C für einen FIR-Filter. Der benötigte Speicherplatz beträgt für einen FIR-Filter 50-ter Ordnung unter 3 MByte.

Die minimalen Modellschrittzeiten des erzeugten Echtzeitcodes sind sowohl von der Abarbeitungsart als auch vom verwendeten Integrationsalgorithmus abhängig. Bild 9-21 zeigt die minimalen Modellschrittzeiten für den Euler und den Runge-Kutta-2 Integrationsalgorithmus für pseudoraten-basierte Abarbeitung. Eine ratenmonotone Abarbeitung wird durch die maximale Wiederholrate des eingesetzten Watchdog–Timers begrenzt, die bei 100 µs liegt. Eine Abarbeitung nach der Abarbeitungsart AFAP ist nicht sinnvoll, da für die Integrationsalgorithmen eine Zeitbasis benötigt wird.

Die Abarbeitungszeit für den Runge-Kutta-2 Integrationsalgorithmus erfordert etwa den Faktor zwei über der Abarbeitungszeit des Euler Integrationsalgorithmus, wenn man den Aufbau der beiden Integrationsalgorithmen miteinander vergleicht (Kapitel 8.2.2). Dies resultiert hauptsächlich

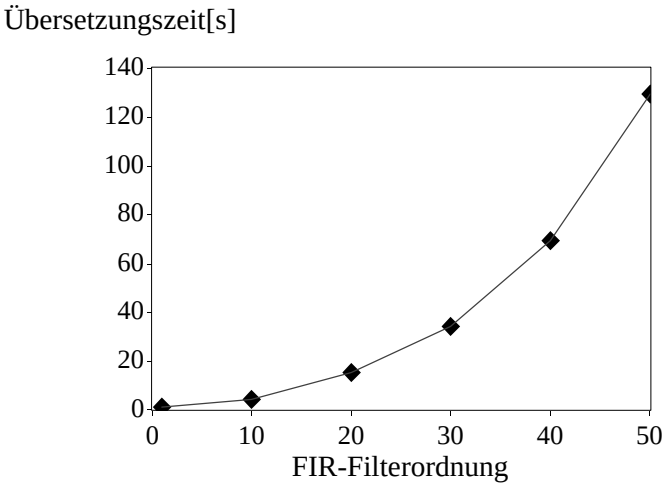


Bild 9-20: Übersetzungszeiten des Compilers CDIF2C

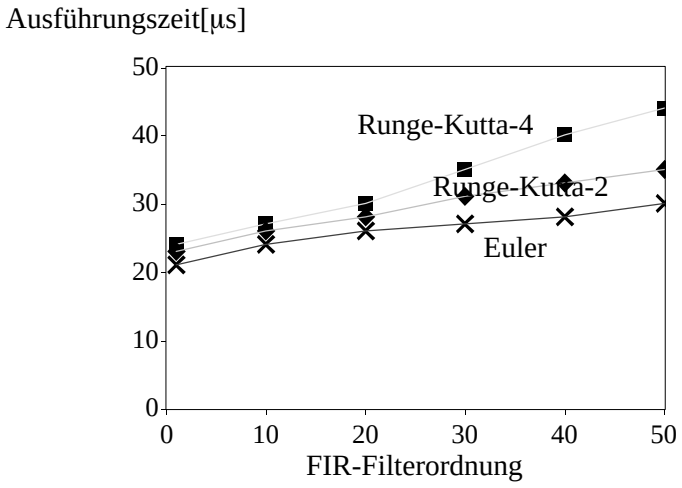


Bild 9-21: Minimale Modellschrittzeit des Echtzeitcodes für einen FIR-Filter

durch die zweifache Berechnung der Stützstellen. Rechnet man den Mehraufwand heraus, der für die Abarbeitung des pseudoraten-basierten Schedulings benötigt wird, ergibt sich der doppelte Aufwand. Durch die vierfache Berechnung der Stützstellen erfordert der Runge-Kutta-4 Integrationsalgorithmus nochmals den doppelten Berechnungsaufwand.

Im Vergleich zu dem von MATRIX$_X$™ erzeugten Echtzeit-Code ist der eigenerzeugte Code in etwa um den Faktor drei schneller (Bild 9-22). Dies ist insbesondere deswegen erstaunlich, weil der von MATRIX$_X$™ erzeugte Echtzeit-Code denselben Integrationsalgorithmus benutzt. Für die Codeerzeugung wurde die Version 6.01 von MATRIX$_X$™ verwendet. Der mitgelieferte Scheduler mußte für eine Verwendung mit dem Echtzeitbetriebssystem VxWorks™ unter Verwendung eines Templates umgeschrieben werden, eine optimierte Fassung lag nur für das Echtzeitbetriebssystem pSOS$^+$ vor. Die Zeitunterschiede resultieren hauptsächlich aus der zeitintensiven Speicherung der errechneten Ergebnisse zu jedem Modellschritt. Mit dieser Speicherung läßt sich auch der konstante Abstand zwischen den Ausführungszeiten erklären.

Auch die Größe des Echtzeitcodes liegt in etwa um den Faktor 1,5 über der des eigenerzeugten Echtzeitcodes. Während die Integrationsalgorithmen von Aufbau und Größe sehr dicht beieinander liegen, benötigen Scheduler und einige Verwaltungsfunktionen den Großteil der Codegröße. Betrach-

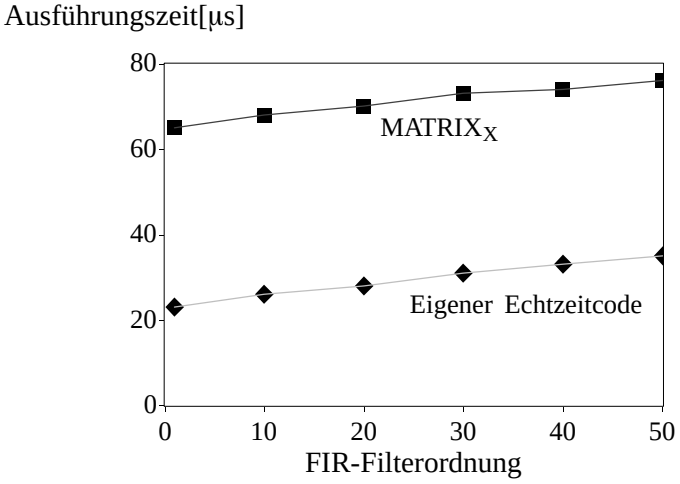


Bild 9-22: Modellschrittzeit im Vergleich zu MATRIX$_X$ ™

tet man die Größe der Applikation fällt die Größe beider Echtzeitcodes im Vergleich zum diskreten Teil sehr klein aus. Auch der Anstieg der Codegröße ist sehr gering, da zur Abarbeitung der zusätzlichen FIR-Filter hauptsächlich zusätzliche Datenfelder benötigt werden. Der Integrationsalgorithmus kann von allen FIR-Filtern benutzt werden.

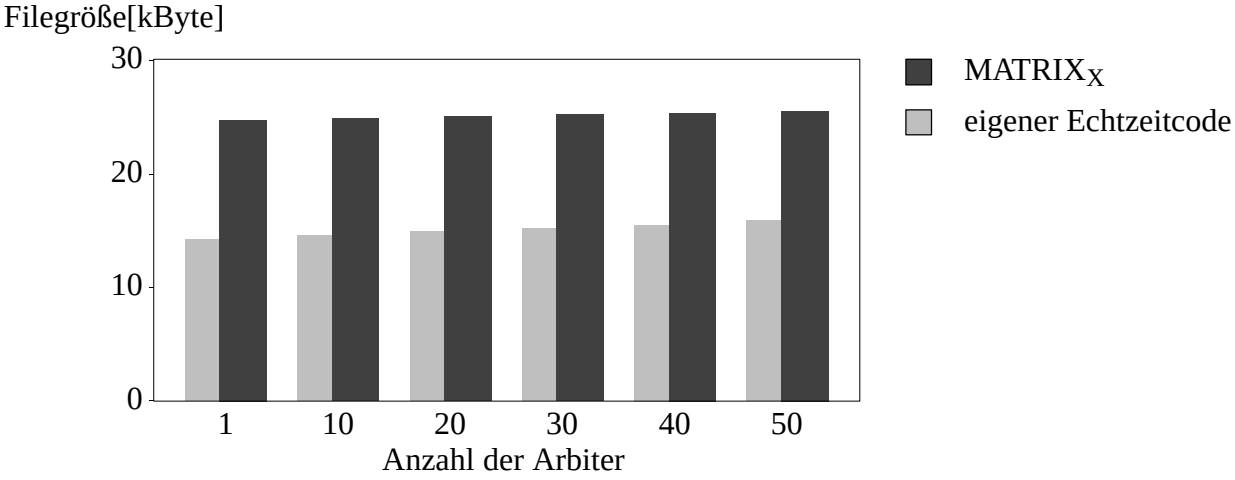


Bild 9-23: Codegröße im Vergleich zu MATRIX$_X$ ™

9.2.3 Heterogener Bereich

Um eine Aussage über die Kopplung diskreter und kontinuierlicher Modelle treffen zu können, wird die diskrete Arbiterkette mit den zeit–kontinuierlichen Signalen der FIR-Filter gekoppelt. Zu diesem Zweck werden die Ausgänge der Arbiter (GoL-Signal) einer Arbiterkette jeweils mit dem Eingang eines FIR-Filters 3. Ordnung verbunden. Bild 9-24 zeigt die Struktur des Gesamtsystems [Schn98].

Die CDIF-Beschreibungen der einzelnen Modelle sind identisch zu den in Kapitel 9.2.1 und 9.2.2 beschriebenen Modellen. Um die Kopplung durchzuführen, wird der Compiler LINKCDIF verwendet, der mehrere Modellbeschreibungen in CDIF einliest und zur Erzeugung des Gesamtsystems die entsprechenden Ein- und Ausgangssignale miteinander verbindet. In Bild 9-25 ist dargestellt, welche Zeit für die Kopplung der Modelle benötigt wird. Die hohen Übersetzungszeiten kön-

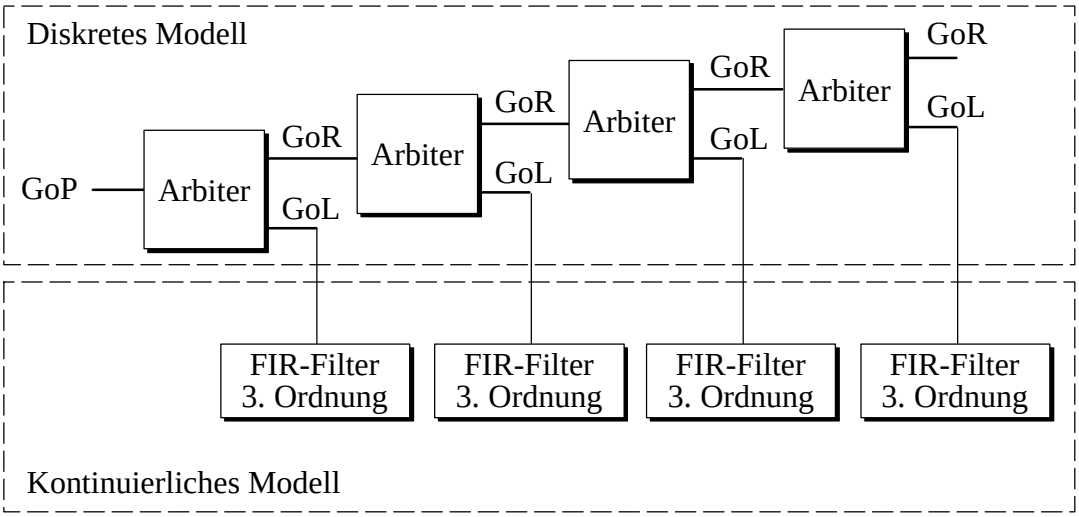


Bild 9-24: Struktur des Arbiter/FIR-Filter Systems

nen durch das mehrfache Kopieren der Datenbäume, die bei der Kopplung erzeugt werden, und den damit verbundene hohen Allokationsaufwand erklärt werden. Der benötigte Speicherplatz liegt unter 100 kByte für alle Modelle.

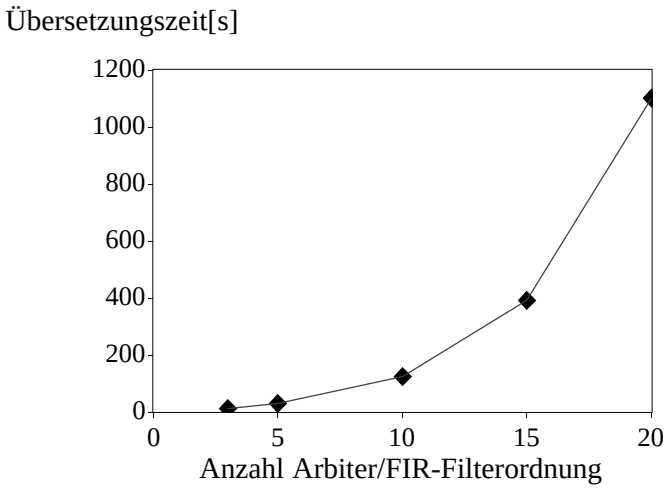


Bild 9-25: Übersetzungszeiten des Compilers LINKCDIF

Die Modellschrittzeiten des gekoppelten Systems wurden für den Euler, den Runge-Kutta-2 und den Runge-Kutta-4 Integrationsalgorithmus jeweils für ratenmonotone und pseudoraten-basierte Abarbeitung gemessen (Bild 9-26). Auch hier liegt die minimale Modellschrittzeit der ratenmonotonen Abarbeitung deutlich über der pseudoraten-basierten Abarbeitung. Insbesondere das starke Ansteigen der Ausführungszeit bei großen Modellen erfordert eine genauere Betrachtung, die in Kapitel 9.3 vorgenommen wird. Die Unterschiede bei der Verwendung unterschiedlicher Integrationsalgorithmen fallen weniger deutlich aus, da das kontinuierliche Modell nur einen Teil des gesamten Systems ausmacht. Vergleicht man die Summe der einzeln gemessenen Modelle von Arbiter und FIR-Filtern mit den gekoppelten Systemen gleicher Größe, fällt auf, daß das gekoppelte System schneller abgearbeitet wird. Dies ist mit der gemeinsamen Verwendung eines Auxiliary-Timers zu erklären, da im Gegensatz zu der Abarbeitung der Einzelmodelle der Mehraufwand des Timers nur einfach aufgewendet werden muß.

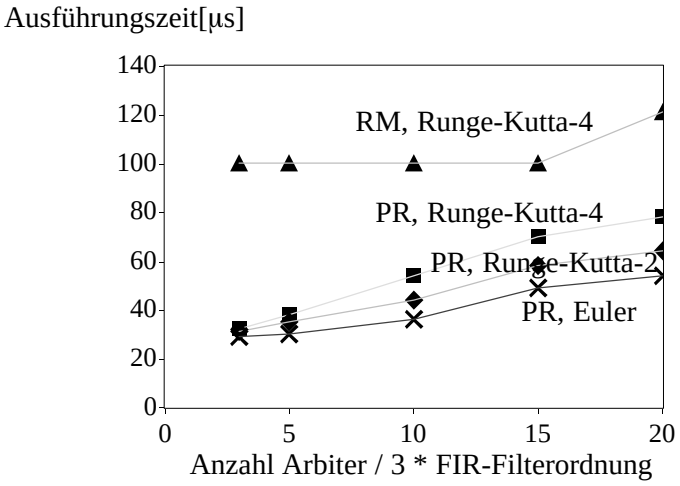


Bild 9-26: Minimale Modellschrittzeit des Echtzeitcodes für eine FIR-/Arbiter-Anordnung

Die Größe des erzeugten Echtzeitcodes (Bild 9-27) steigt nahezu linear mit der Größe der Anzahl der Arbiter und FIR-Filter an. Wie bereits aus den Ergebnissen von Kapitel 9.2.1 und 9.2.2 zu erkennen ist, ist die Codegröße hauptsächlich von der Anzahl der Arbiter abhängig, während die Codegröße auch für FIR-Filterordnungen nahezu konstant ist.

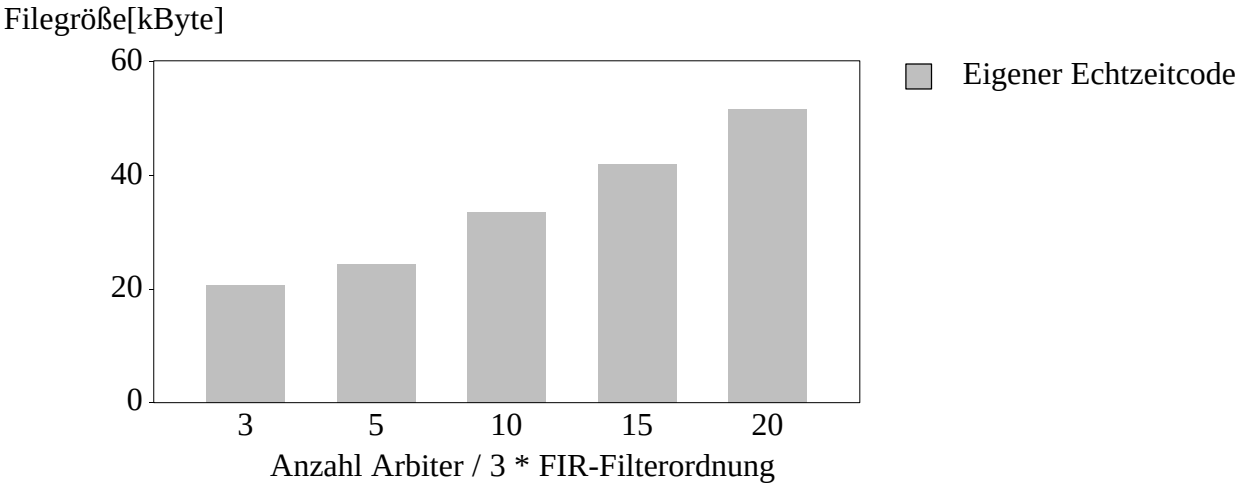


Bild 9-27: Codegröße des heterogenen Modells

Zusammenfassend läßt sich sagen, daß die Übersetzungszeiten durch die umfangreichen CDIF-Modelle insbesondere bei der Übersetzung einer CDIF-Beschreibung in Echtzeitcode noch zu groß sind, um mit sehr großen Modellen arbeiten zu können. Optimierungsarbeiten in diesem Bereich bergen allerdings noch großes Potential [Schn98], so daß davon ausgegangen werden kann, daß die Übersetzungszeiten bei weiterer Optimierung der Compiler stark verringert werden können. An dieser Stelle sollte auch erwähnt werden, daß der Umfang der verwendeten Modelle so groß war, daß die CASE-Werkzeuge teilweise bereits Darstellungsprobleme aufwiesen. Die minimalen Modellschrittzeiten und die Größe des Echtzeitcodes dieser umfangreichen Modelle eignen sich für konzept-orientiertes Rapid Prototyping bestens. Um eine Einschätzung für den industriellen Einsatz vornehmen zu können, wurde das Rapid Prototyping System anhand einer praktischen Aufgabenstellung aus dem Automobilbereich getestet, die im folgenden vorgestellt wird.

9.3 Fallbeispiel Fensterhebermodellierung

Für den Test des entwickelten Rapid Prototyping Systems an einer realen Aufgabenstellung wurde die Modellierung eines Fensterhebers mit Einklemm- und Diebstahlschutz implementiert. Die Modellierung entspricht der Spezifikation eines aktuellen Oberklasse-Automobils und wurde an einer Daimler-Benz C-Klasse und S-Klasse Tür erprobt, um die notwendigen Veränderungen zur Anpassung an unterschiedliche Umgebungen dokumentieren zu können.

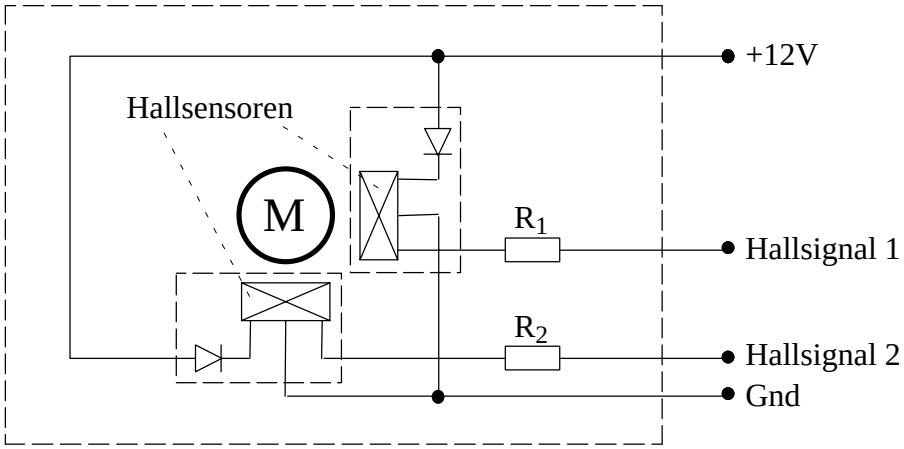


Bild 9-28: Blockschaltbild des Fensterhebermotors

Die Spezifikation unterscheidet grundsätzlich zwischen zwei Betriebsarten, die mit nicht-normiertem und normiertem Betrieb bezeichnet sind. Der nicht-normierte Betrieb ist zu Beginn aktiv und unterstützt die Tastenfunktion “manueller Hochlauf”, bei der das Fenster nach oben fährt, und die Tastenfunktion “manueller Tieflauf”, bei der das Fenster nach unten fährt. Ein automatischer Hochlauf oder Tieflauf des Fensters ist hierbei nicht aktiviert. Nach Erkennen eines Einklemmvorgangs beim manuellen Hochlauf wird der normierte Betrieb für den Bereich unterhalb der erreichten Fensterposition ermöglicht. Die Fensterposition kann durch einen Positionszähler bestimmt werden, der durch zwei Hallsensoren gespeist wird, die im Fensterhebermotor integriert sind (Bild 9-28). Die beiden Hallsignale sind um 90 Grad phasenverschoben, so daß sowohl die Bewegungsrichtung als auch die absolute Positionsveränderung bestimmt werden kann. Bild 9-29 zeigt die beiden phasenverschobenen Hallsignale bei einem Einklemmvorgang. Zu Beginn wird ein Hochlauf ohne Einklemmvorgang vorgenommen, bei dem die Hallsignale mit konstanter Länge um 90 Grad phasenverschoben erzeugt werden. Durch den Einklemmvorgang wird die Länge der Hallsignale immer weiter gestreckt, bis keine Änderung mehr detektiert werden kann.

Nach Erkennen eines Einklemmvorgangs wird der normierte Betrieb für den Bereich unterhalb der erreichten Fensterposition ermöglicht. Hierbei sei angemerkt, daß ein Einklemmvorgang durch die maximale Kraft F_{max} erkannt wird, die an einem eingeklemmten Körper anliegt. Die anliegende Kraft wird im nicht-normierten Betrieb auf 100 N begrenzt. Man spricht in diesem Zusammenhang von einer *Überschußkraftbegrenzung.*

Im normierten Betrieb werden die Tastenfunktionen manueller Hochlauf und manueller Tieflauf wie im nicht-normierten Betrieb unterstützt. Zusätzlich wird ein automatischer Lauf des Fensters nach oben und unten ermöglicht. Nach Erkennen eines Einklemmvorgangs wird bei einem automatischen Hochlauf des Fensters innerhalb von $t < 50\ ms$ der Antrieb in Richtung tief angesteuert. Diese Funktionalität entspricht dem Einklemmschutz. Wird ein Einklemmvorgang während des

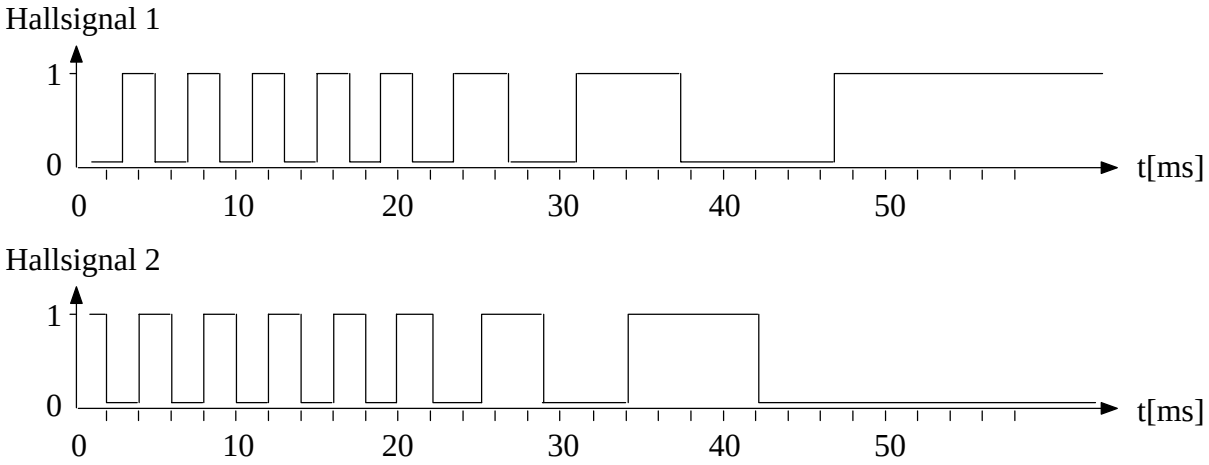


Bild 9-29: Verlauf der Hallsignale bei einem Einklemmvorgang

Tieflaufs erkannt, so wird der Antrieb gestoppt. Wird während dem Erkennen des Einklemmens beim Hochlauf eine aktive Ansteuerung durch eine Schalterbetätigung vorgenommen, so stoppt der Antrieb innerhalb von $t < 50\,ms$. Erst nach Rücknahme der Schalterbetätigung reversiert der Antrieb. Diese Funktionalität entspricht dem Diebstahlschutz (Eingriff von außen durch das Fenster).

Eine Entnormierung, d.h. der Übergang von normiertem in nicht-normierten Betrieb, muß erfolgen, wenn das Fenster beim manuellen oder automatischen Betrieb den höchsten möglichen Zählerstand überschreitet. Bei Überschreitung des höchsten möglichen Zählerstands bei gleichzeitiger Einklemmerkennung wird eine automatische Nachnormierung durchgeführt. Zusätzlich ist sicherzustellen, daß mechanische Schwingungen nicht als Einklemmen erkannt werden dürfen (”Schlechtweg–Erkennung”).

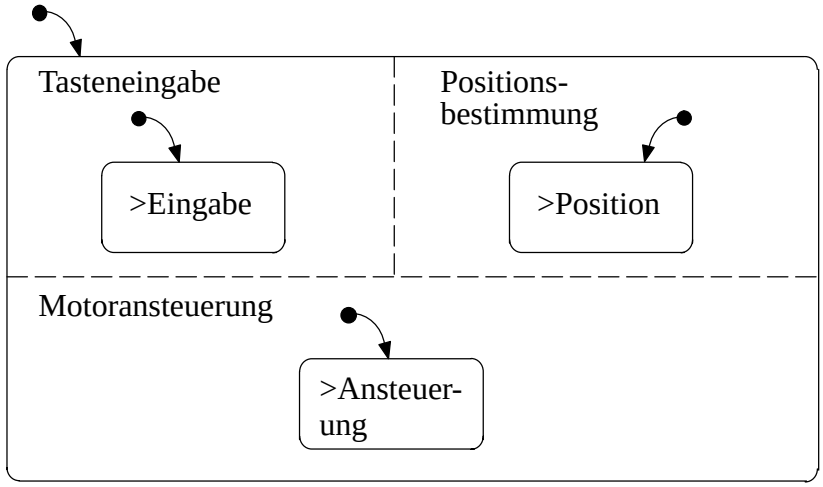


Bild 9-30: Statechart-Modellierung MainChart

Die Modellierung erfolgte mit den CASE-Werkzeugen STATEMATE™ und MATRIX$_X$™. Mit STATEMATE™ wurde die Eingabe der Fensterbewegung, die Positionsbestimmung und die Motoransteuerung implementiert. Bild 9-30 zeigt die drei hauptsächlichen parallelen Komponenten der Modellierung des Fensterhebersteuergeräts in hierarchischer Sichtweise. Die Modellierung der drei hierarchischen Zustände wird im folgenden näher erläutert.

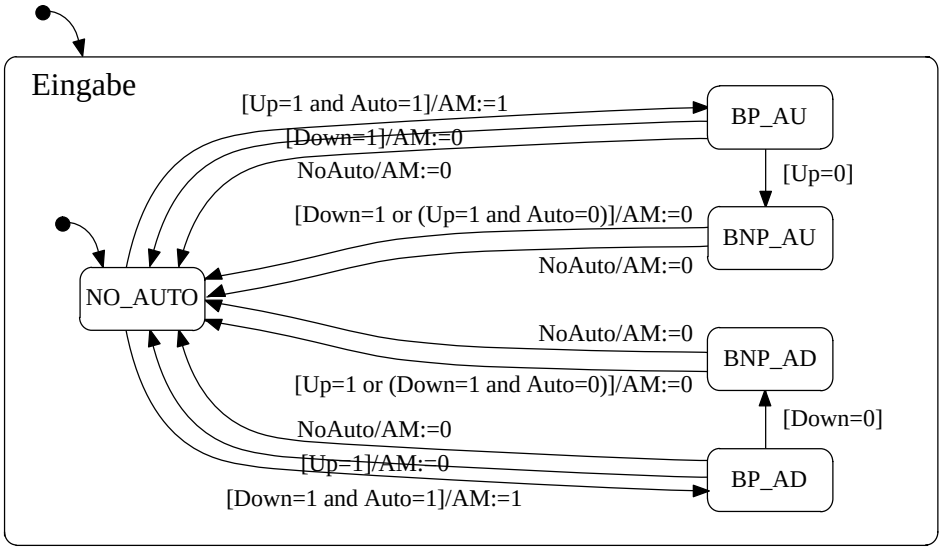


Bild 9-31: Statechart-Modellierung der Tasteneingabe

Die Modellierung der Tasteneingabe (Bild 9-31) erfolgt mit Hilfe von fünf Zuständen, die keine Tastenbetätigung, Tastenbetätigung für den automatischen Hoch- und Tieflauf sowie die Speicherung des automatischen Hoch- und Tieflaufs ohne Betätigung einer Taste repräsentieren. Der Zustand NO_AUTO wird nur verlassen, wenn eine Tastenbetätigung im normierten Betrieb vorliegt. Das gemeinsame Betätigen der Tasten für den Hoch- und Tieflauf ist durch den mechanischen Aufbau des Tasters ausgeschlossen. Wurde beispielsweise die Taste ’Up’ im normierten Betrieb betätigt, wird die Variable ’AM’ (AutoMode) zu ’1’ gesetzt und ein Zustandsübergang in den Zustand BP_AU (ButtonPress_AutoUp) erfolgt. Die Ansteuerung des Fensterhebermotors geht in den automatischen Hochlauf über (siehe Statechart Ansteuerung). Auch für den Fall, daß die Tastenbetätigung nicht mehr erfolgt, wird der Hochlauf beibehalten (Zustand BNP_AU). Erst wenn entweder die gegenläufige Taste ’Down’ betätigt wird oder ein Abbruch des Automatikbetriebs durch das Event ’NoAuto’ wegen eines Einklemmvorgangs ausgelöst wird, wird in den Zustand NO_AUTO zurückgekehrt.

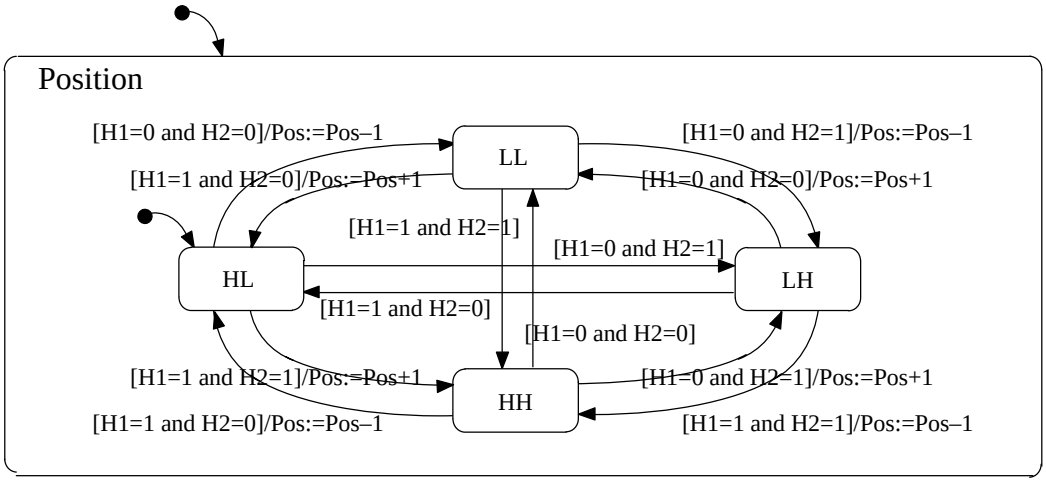


Bild 9-32: Statechart-Modellierung der Positionsbestimmung

Zur Positionsbestimmung werden vier Zustände benutzt, die die Auswertung der Hallsignale vornehmen (Bild 9-32). Entsprechend den eingegangenen Hallsignalen 1 und 2 geben die Namen der

Zustände jeweils den aktuellen Wert der Signale wieder (H für 'high', L für 'low'). Findet die Änderung eines Hallsignals statt, wird ein Zustandsübergang durchgeführt. Da die Hallsignale um 90 Grad phasenverschoben sind, kann ein gleichzeitiger Übergang beider Signale innerhalb eines Modellschritts nur auftreten, wenn die Modellschrittzeit größer als die Phasenverschiebung der beiden Hallsignale ist. Für diesen Fall, der im allgemeinen ausgeschlossen werden sollte, sind Zustandsübergänge beider Hallsignale in Bild 9-32 berücksichtigt.

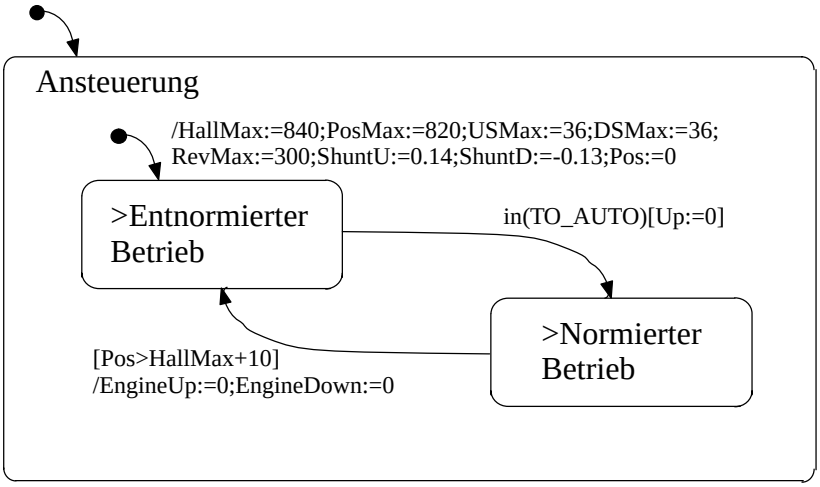


Bild 9-33: Statechart-Modellierung der Motoransteuerung

Die Ansteuerung des Fensterhebermotors ist in Bild 9-33 gezeigt. Die Ansteuerung unterteilt sich in zwei Bereiche, die dem entnormierten und dem normierten Betrieb entsprechen. Der Zustandsübergang zwischen entnormierten und normierten Betrieb findet statt, wenn ein Einklemmvorgang beim Hochlauf erkannt wird. In diesem Fall wird der Zustand TO_AUTO betreten, der im Statechart des entnormierten Betriebs zu finden ist und den Übergang in den normierten Betrieb auslöst, wenn die Taste 'Up' nicht mehr betätigt wird. Der Zustandsübergang von normierten zu entnormierten Bereich ist vorgesehen, wenn eine fehlerhafte Normierung stattgefunden hat. Dies ist dann der Fall, wenn das Fenster sich nicht in der höchsten Position bei der Normierung befand und somit den Normierungswert 'HallMax' überschreiten kann. Als Toleranz ist in dieser Modellierung der Wert von 10 Hallsignalen vorgesehen, was in etwa 5mm entspricht.

Das Statechart des entnormierten Betriebs ist in Bild 9-34 dargestellt. Bevor dieses Statechart betreten wird, werden einige für den Betrieb benötigte Variablen initialisiert (siehe Bild 9-33). Der Zustand IDLE, der zuerst aktiv ist, wird verlassen, wenn eine der Tasten 'Up' oder 'Down' betätigt wurde. Für den Tieflauf wird die Taste 'Down' betätigt und der Zustand DOWN betreten. Bei der Durchführung des Zustandsübergangs wird der Fensterhebermotor mit der Variablen 'EngineDown:=1' angesteuert. Der Tieflauf wird solange durchgeführt, bis die Variable 'EngineDown' auf den Wert '0' zurückgesetzt wird. Nach Betreten des Zustands DOWN wird eine Schleife durchlaufen, die die Variable 'Cnt' bis zum Wert von 'DSMax–1' erhöht. Dies geschieht, um bei der Anlaufphase des Fensters, bei der der Motorstrom Spitzenwerte aufweist, eine fälschlich erkannte Blokkade wegen eines Einklemmvorgangs zu verhindern. Der Motorstrom wird über einen Shunt-Widerstand ermittelt und über die Variable 'Shunt' zur Verfügung gestellt. Unterschreitet die Variable 'Shunt' einen vorher festgelegten Wert beim Tieflauf, liegt ein Einklemmzustand (BLOCKED) vor und der Motor kann abgeschaltet werden. Erst wenn eine eventuell vorhandene Tastenbetätigung zurückgenommen wird, wird ein Zustandsübergang in den Zustand IDLE durchgeführt.

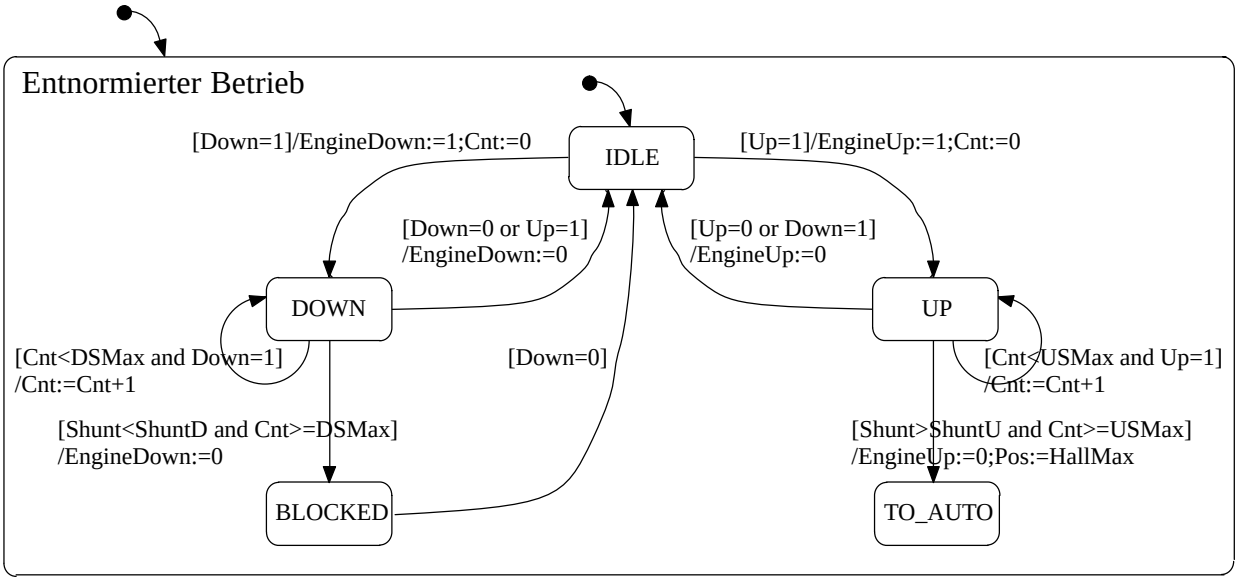


Bild 9-34: Statechart-Modellierung des entnormierten Betriebs

Ebenso wie beim Tieflauf verhält sich die Modellierung beim Hochlauf. Überschreitet die Variable ’Shunt’ einen festgelegten Wert, so ist ein Einklemmzustand erkannt worden. Im Gegensatz zum Tieflauf wird der Zustand TO_AUTO angenommen, der Hochlauf unterbrochen und die Variable ’Pos’ nimmt den Wert von ’HallMax’ an. Wird eine eventuell vorhandene Tastenbetätigung ’Up’ zurückgenommen, wird der normierte Teil des Statecharts betreten (siehe Bild 9-33).

Der normierte Betrieb ist in Bild 9-35 dargestellt. Auch im normierten Betrieb wird bei Betätigung der Taste ’Down’ der Zustand DOWN betreten. Der Einklemmschutz wird beim Anfahren außer Betrieb gesetzt, indem die Variable ’Cnt’ erst den Wert ’DSMax’ annehmen muß, ehe ein Zustandsübergang nach BLOCKED stattfinden kann. Um den automatischen Tieflauf zu unterbrechen, muß entweder die Taste ’Up’ betätigt werden oder der Automatikmodus AM zurückgenommen worden sein, ohne daß die Taste ’Down’ betätigt wird. Ist dies der Fall, so wird ein Zustandsübergang in den Zustand BLOCKED durchgeführt, der nach Rücknahme der Taste ’Up’ einen Zustandsübergang nach IDLE vornimmt.

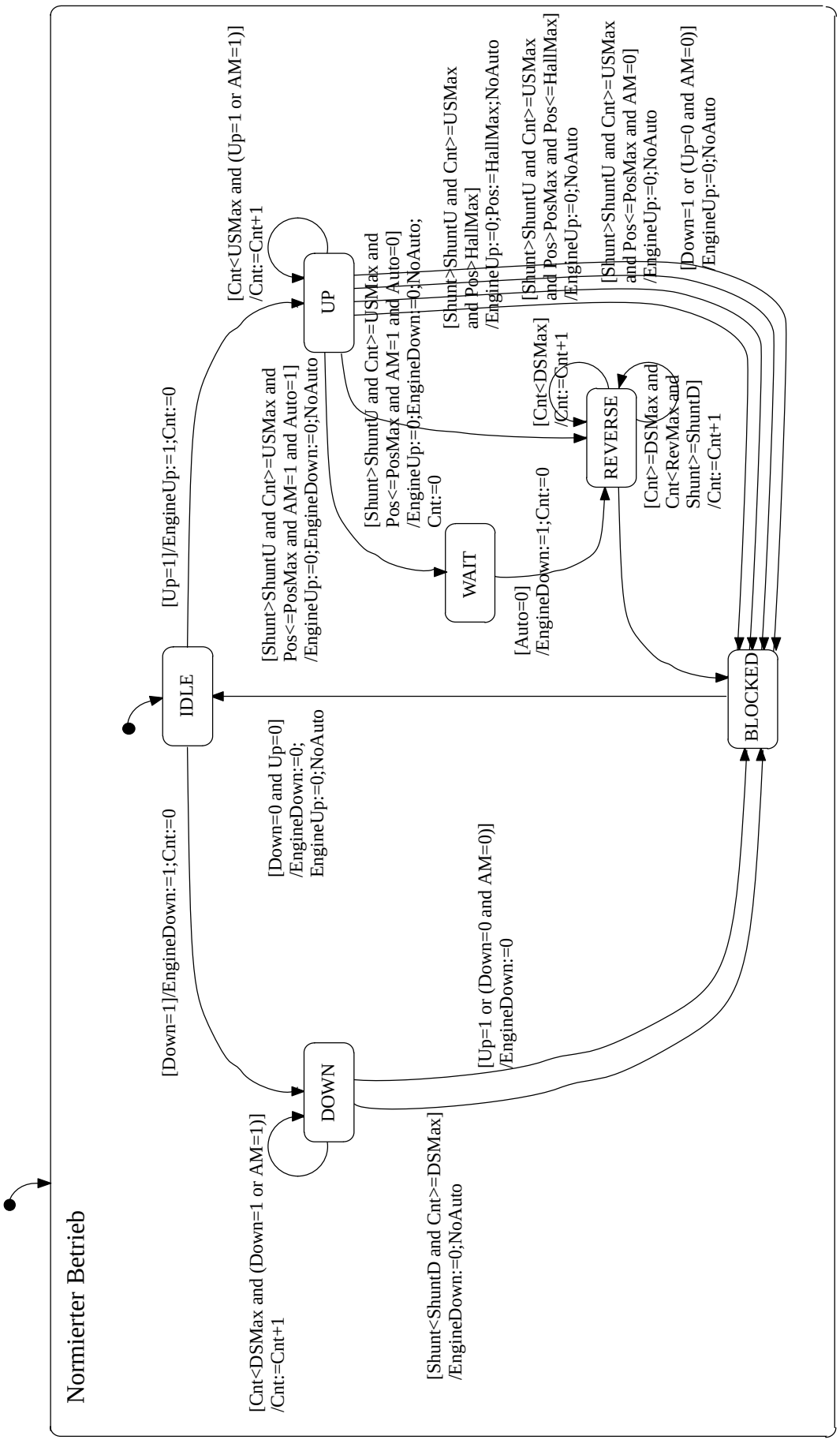


Bild 9-35: Statechart-Modellierung des normierten Betriebs

Die Modellierung des Hochlaufs im normierten Betrieb fällt umfangreicher aus. Bei Betätigung der Taste ’Up’ wird der Zustand UP betreten. Auch hier ist der Einklemmschutz beim Anfahren außer Betrieb gesetzt, da die Variable ’Cnt’ erst den Wert ’USMax’ annehmen muß, ehe ein Zustandsübergang nach WAIT, REVERSE oder BLOCKED vorgenommen werden kann. Ist die Anfahrphase vorbei, kann eine Einklemmerkennung durchgeführt werden. Überschreitet die Variable ’Shunt’ einen festgelegten Wert ’ShuntU’, wird der Hochlauf unterbrochen und der Tieflauf aktiviert. Wird beim Tieflauf ein Einklemmvorgang erkannt, der nicht in der Anlaufphase liegt, wird auch der Tieflauf unterbrochen und der Zustand BLOCKED betreten. Ansonsten wird der Tieflauf fortgesetzt, bis die Variable ’Cnt’ den Wert ’RevMax’ erreicht hat. Bei der Erkennung eines Einklemmvorgangs im Zustand UP wird zusätzlich eine Abfrage durchgeführt, ob die Taste ’Auto’ betätigt ist. Ist das der Fall, so wird erst bei Zurücknahme der Betätigung der automatische Tieflauf aktiviert. Diese Funktionalität entspricht dem oben beschriebenen Diebstahlschutz. Der manuelle oder automatische Hochlauf kann durch die Taste ’Down’ unterbrochen werden. Wird während des automatischen Hochlaufs ein Einklemmen an einer Position oberhalb der Variablen ’HallMax’ erkannt, findet eine Nachnormierung des Positionswerts statt, indem der Positionsvariablen ’Pos’ der Wert ’HallMax’ zugewiesen wird. Findet die Einklemmung zwischen einem Positionswert ’PosMax’ und ’HallMax’ statt, wird der Hochlauf ohne Reversieren beendet. Das Fenster wird in diesem Fall in seiner regulären Endposition geschlossen.

Der normierte Betrieb wird nur dann verlassen, wenn der Positionszähler ’Pos’ einen Wert größer als ’HallMax+10’ annimmt (Bild 9-33). In diesem Fall wurde eine fehlerhafte Normierung durchgeführt, da die Obergrenze deutlich überschritten wurde. Es findet ein Rücksprung zum entnormierten Betrieb statt.

Zur Modellierung der Schlechtweg–Erkennung wurde ein FIR-Filter (Bild 9-16) als Moving Average Filter verwendet, der mit MATRIX$_X$™ modelliert wurde. Der Moving Average Filter unterschiedet sich vom FIR-Filter durch die Verwendung gleicher Koeffizienten bei allen Multiplizierern. In der Summe betragen die Koeffizienten aller Multiplizierer ’1’. Der Moving Average Filter wurde verwendet, um die Schwankungen des eingehenden analogen Shunt-Signals zu glätten. Für die Modellierung des Fensterhebers wurde ein Moving Average Filter 4. Ordnung verwendet, der den Koeffizienten ’0.25’ für alle Multiplizierer aufwies. Der aktuell ermittelte Wert des Shunt-Signals geht somit ebenso wie die drei zuvor ermittelten Werte zu 25% in das Modell ein.

Zur Überprüfung der Funktionalität wurde eine Simulation im Einzelschrittmodus durchgeführt. Bei der vorliegenden Komplexität fällt diese Simulationsart zeitintensiv aus, da die Stimulieingabe für jedes Signal per Hand erfolgen muß. Um eine bessere Überprüfbarkeit zu erlangen, wurde eine grafische Oberfläche erstellt, die auch das Verhalten der Umgebung der Fensterheberansteuerung nachbilden konnte. Auf diese Art konnte die Funktionalität schnell überprüft werden.

Die Messungen wurden mit einem Motorola MVME-2604 Board, das mit einem PowerPC Prozessor 604 mit 200 MHz und 32 MByte Speicher bestückt war, und dem Echtzeitbetriebssystem VxWorks™ der Firma WindRiver Inc. durchgeführt. Das Blockschaltbild der verwendeten Hardware zur Ansteuerung des Fensterhebers ist in Bild 9-37 dargestellt. Die Tasten ’Up’, ’Down’ und ’Auto’ sowie die Hallsignale werden mit Hilfe des digitalen Ein-/Ausgabemoduls M11 verarbeitet. Zur Ansteuerung des Fensterhebermotors werden die digitalen Signale ’EngineUp’ und ’EngineDown’ erzeugt, die eine H-Brückenschaltung speisen. Die H-Brückenschaltung besteht aus vier Leistungstransistoren, die Durchlaßströme bis zu 40 A erlauben. Je nach anliegendem Signal läuft

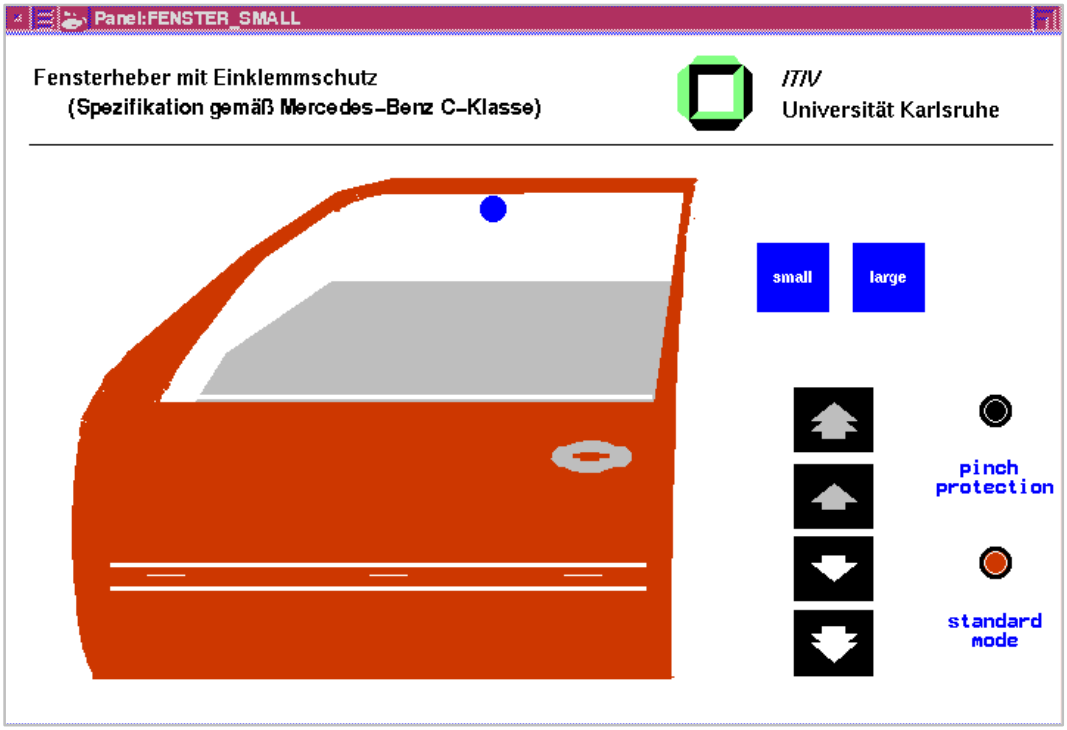


Bild 9-36: Interaktive Simulation der Fensterheberansteuerung

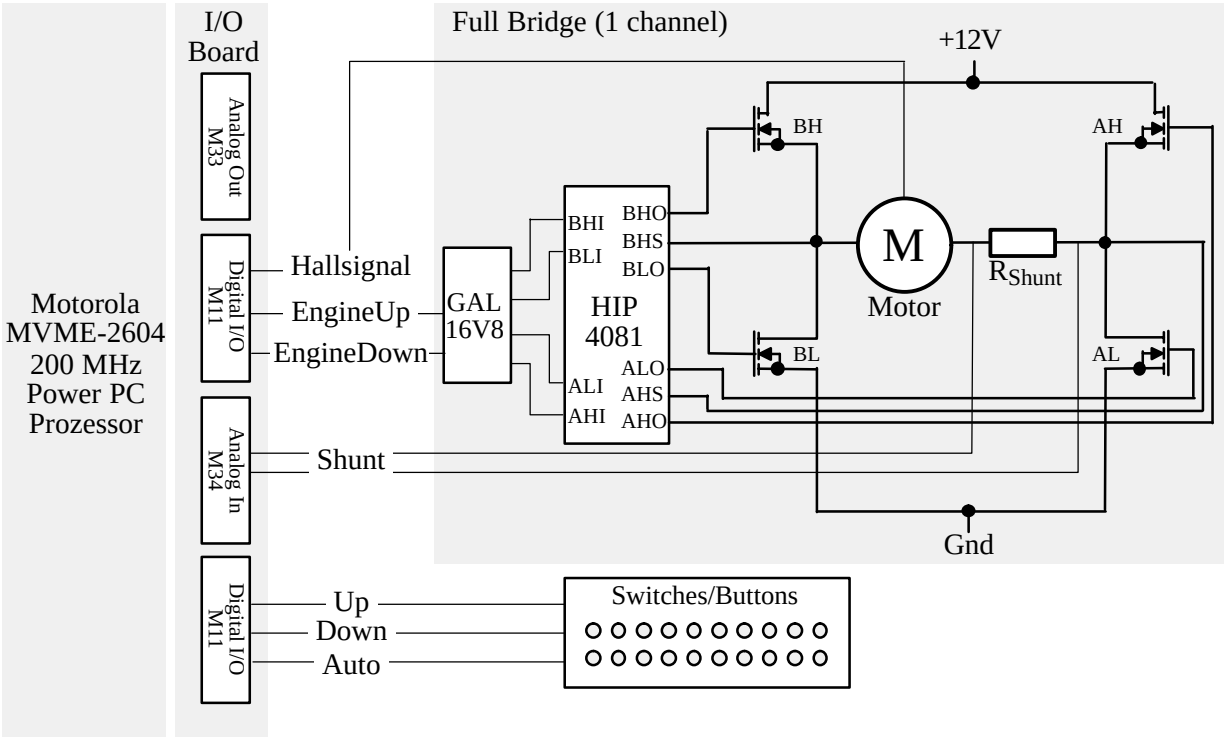


Bild 9-37: Blockschaltbild der Fensterheber-Umgebung

der Motor links oder rechts herum, blockiert oder wird im Leerlauf betrieben. Werden die Transistoren AH und BL geschaltet, so dreht sich der Motor in eine Richtung. Werden die Transistoren AL und BH geschaltet, dreht sich der Motor in die Gegenrichtung. Ein blockierender Motor wird durch die geschalteten Transistoren AL und BL erreicht. Bei der Ansteuerung muß insbesondere darauf geachtet werden, daß es zu keinen Kurzschlüssen der Leistungstransistoren kommt. Um eine kurz-

schlußfreie Ansteuerung unter Beachtung von Zeitbedingungen zu erreichen, wird ein H-Brückentreiberbaustein HIP 4081 verwendet. Der Baustein schleift die eingehenden Steuereingänge BHI, BLI, AHI und ALI an die Ausgänge BHO, BLO, AHO und ALO durch und verhindert dabei alle Eingangsbelegungen, die zu einem Kurzschluß führen können.

Bild 9-38: Rapid Prototyping einer Fensterheberansteuerung

Bild 9-38 zeigt den Versuchsaufbau im Automobil. Im Kofferraum ist der Rapid Prototyping Rechner befestigt, der mit den Türtastern verbunden ist und den Fensterhebermotor ansteuert. Führt man das Modell aus, können die Initialisierungswerte aus Bild 9-33 auf den vorliegenden Fall angepaßt werden. Die unterschiedliche Beschaffenheit der verwendeten Gummidichtungen beeinflußt beispielsweise die Größe der Shunt-Variablen. Auch die Form des verwendeten Fensters und die Übersetzung des Getriebes des Fensterhebers führt zu Anpassungen der Initialisierungswerte. Um einen korrekten Betrieb in unterschiedlichsten Einsatzbedingungen zu gewährleisten, nehmen industrielle Fensterheberansteuerungen auch eine nachträgliche Adaption der Shuntwerte während des Betriebs vor. Auf diese Art können Temperaturänderungen oder Änderungen der Gummibeschaffenheit ausgeglichen werden.

Die Übersetzungszeiten der benötigten Compiler STM2CDIF, MX2CDIF, LINKCDIF und CDIF2C liegen für dieses Beispiel insgesamt bei unter einer Minute. Anhand des Beispiels der Fensterheberansteuerung kann nochmals der Vorteil der pseudoraten-basierten Abarbeitung im Vergleich zur ratenmonotonen Abarbeitung aufgezeigt werden. Die minimale Modellschrittzeit einer pseudoraten-basierten Abarbeitung wird in 28 µs ausgeführt, während die ratenmonotone Abarbeitung 100 µs benötigt. Die Codegröße des Echtzeitsystems beträgt bei beiden Abarbeitungsarten 24 kByte. Sowohl die Abarbeitungszeit als auch die Codegröße zeigen eindrucksvoll, daß das entwikkelte Rapid Prototyping System auch für komplexere Modellierungen beispielsweise im Bereich der Motorsteuergeräte Verwendung finden kann. Obwohl die Erzeugung von serientauglichem Code nicht beabsichtigt wurde, liegt die Codegröße bereits nahe an einer Serienimplementierung. Erzeugt man den diskreten Teil des Echtzeitcodes mit dem Codegenerator von STATEMATE™, so

erhält man eine minimale Ausführungszeit bei pseudoraten-basierter Abarbeitung von 83 µs und eine Codegröße von 224 kByte.

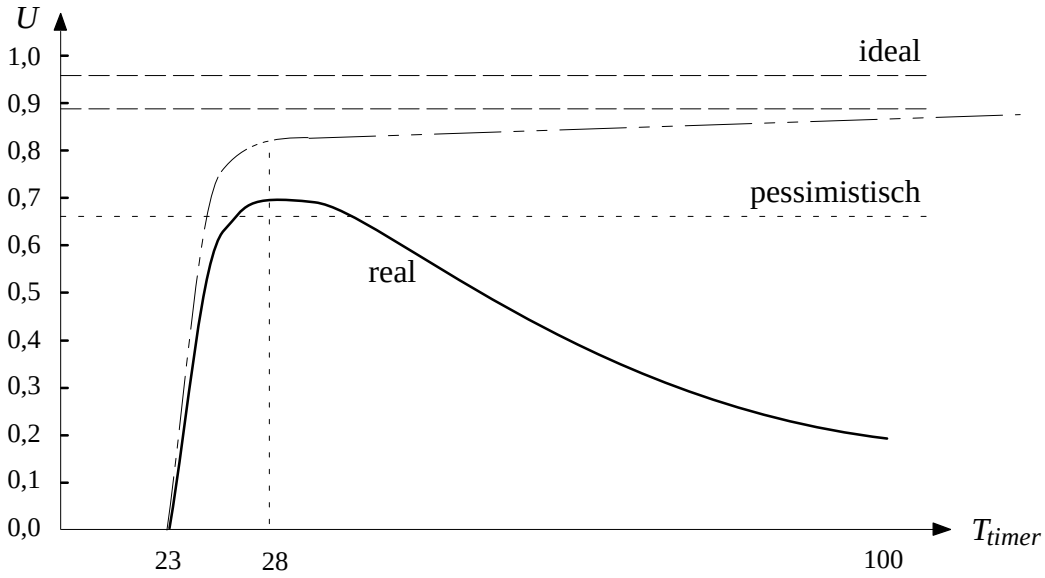


Bild 9-39: Prozessorausnutzung *U* für die Fensterheberansteuerung

Die Prozessorausnutzung *U* für die Fensterheberansteuerung mit pseudoraten-basierter Abarbeitung ist in Bild 9-39 dargestellt. Die Kurve entspricht dem in Kapitel 7.2.2 errechneten Verlauf. Die optimale zeitliche Auflösung $T_{timer,opt}$ wird bei 28 µs erreicht, während der minimale zeitliche Auflösung $T_{timer,min}$ bei 23 µs liegt. Als maximaler Wert der Utilization wird ein Wert von 71% erreicht. Bereits bei 100 µs ist die Utilization auf 0,19 gefallen.

Um den Vergleich zwischen ratenmonotonen und pseudoraten-basiertem Scheduling ohne Einschränkungen (Begrenzung des Watchdog-Timers auf 100 µs) durchführen zu können, wurde das komplette Modell mehrfach parallel berechnet. Bei der ratenmonotonen Abarbeitung wurde für jedes Modell ein eigener Watchdog-Timer erzeugt, während die pseudoraten-basierte Abarbeitung mit einem gemeinsam verwendeten Auxiliary-Timer betrieben wurde. Die minimale Ausführungszeit liegt bei wenigen parallelen Modellen noch dicht beieinander. Bei 50 parallelen Modellen liegt jedoch bereits ein Unterschied um den Faktor 4,7 vor (Bild 9-40).

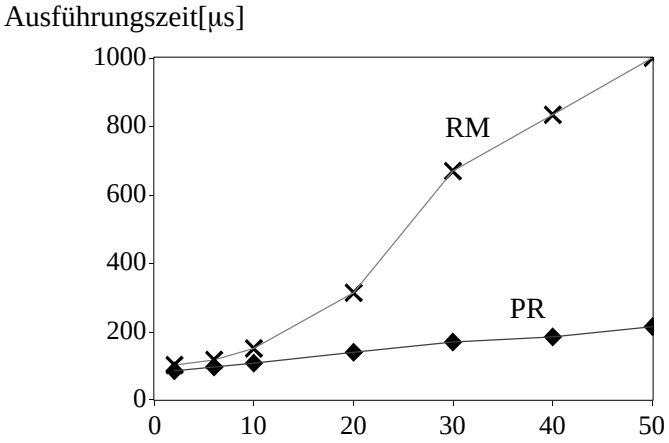


Bild 9-40: Vergleich von ratenmonotonen und pseudoraten-basiertem Scheduling

Durch die hohe Güte des erzeugten Echtzeitcodes kann auch architektur-orientiertes Rapid Prototyping unterstützt werden. Für den Einsatz als Seriencode müssen noch weitere Optimierungen vorgenommen werden. Dieser Einsatz stand bei den Entwicklungen im Rahmen dieser Arbeit nicht im Vordergrund.

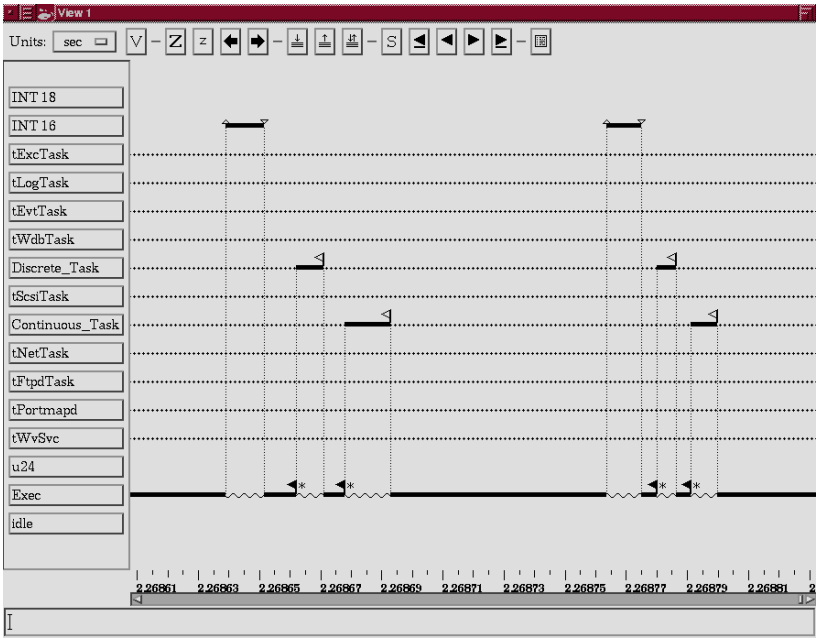


Bild 9-41: Screenshot des Echtzeitanalyse-Werkzeugs WindView

Durch die Verwendung des Echtzeitbetriebssystems VxWorks™ können weitere Analysewerkzeuge den erzeugten Code detailliert überprüfen. In Bild 9-41 ist ein Screenshot des Echtzeitanalyse-Werkzeugs WindView der Firma WindRiver Inc. dargestellt. Mit diesem Werkzeug ist es beispielsweise möglich, Echtzeitprozesse oder Semaphoren zu visualisieren und die Zeiten zu bestimmen, die für einzelne Echtzeitprozesse aufgewendet werden müssen. Die für die Analyse benötigte Zeit fällt gering aus und gefährdet damit die Abarbeitung in Echtzeit in den meisten Fällen nicht. Dargestellt ist hier die Abarbeitung des Fensterhebers mit pseudoraten-basierter Abarbeitung. Der vom Auxiliary-Timer erzeugte Interrupt ist hier als ’Int16’ bezeichnet und tritt periodisch auf. Der Prozeß ’Exec’ korrespondiert mit dem Scheduler, während die ’Discrete_Task’ und Continuous_Task’ mit dem aus STATEMATE™ und MATRIX$_X$™ erzeugten Code korrespondieren. Eine ausgefüllte Fahne zeigt an, daß der Scheduler eine Semaphore für jeden Modellprozeß freigibt. Die beiden Modellprozesse nehmen die Semaphoren und geben sie bei Beendigung ihrer Abarbeitung wieder frei. Nach der Freigabe kann der Scheduler die Verarbeitung wieder aufnehmen und wartet bis zum Ende des Modellschritts. Dort wird kontrolliert, ob alle Modellprozesse abgeschlossen sind. Sind die Modellprozesse noch nicht vollständig abgearbeitet, wird die Ausführung unterbrochen.

10 Zusammenfassung

10.1 Erzielte Ergebnisse

Die wachsende Komplexität elektronischer Systeme und der zunehmende internationale Wettbewerb erfordern ständige Verbesserungen des Entwurfsablaufs. Hierzu werden verstärkt formale Hilfsmittel zur Spezifikation und zur abstrakten Modellierung eingesetzt, von denen man sich eine Erhöhung der Entwurfsqualität und einen schnelleren Weg zum fertigen Produkt erhofft. Mit den formalen Hilfsmitteln finden auch die Simulation und formale Verifikation der Modellierung Einzug in den Entwurf elektronischer Systeme. Die Modellierungstechniken, Simulation und Verifikation sind jedoch oftmals auf einzelne CASE-Werkzeuge festgelegt und können somit nicht als “Best-of-Point” Werkzeuge genutzt werden. Simulation und formale Verifikation besitzen im Entwurfsverlauf Schwächen und Einschränkungen, die beispielsweise aus der Notwendigkeit der Modellierung der Umgebung eines elektronischen Systems und der eingeschränkten zeitlichen Analyse im Zusammenspiel mit Echtzeitbetriebssystemen bestehen. Die Einbeziehung von Prototypen bietet hier eine Lösungsalternative, die jedoch sehr aufwendig und nicht änderungsfreundlich ist. Konzept-orientiertes Rapid Prototyping als Weiterentwicklung der herkömmlichen Prototypen verbindet die Vorteile eines Prototypen mit einem schnellen Aufbau und Änderungsfreundlichkeit. Bestehende Rapid Prototyping Systeme sind jedoch meist werkzeug-abhängig, bieten eine eingeschränkte Hardwareunterstützung und fügen sich nicht nahtlos in den Gesamtsystementwurf ein. Im Rahmen dieser Arbeit wurde eine umfassende Untersuchung zum Stand der schnellen Prototypenentwicklung durchgeführt, die vorhandenen Engpässe bestehender konzept-orientierter Rapid Prototyping Systeme untersucht und ein Anforderungsprofil für zukünftige Rapid Prototyping Systeme aufgestellt. Die gestellten Anforderungen konnten mit dem realisierten Rapid Prototyping System erfüllt werden.

Als Ausgangspunkt des in dieser Arbeit vorgestellten Rapid Prototyping Systems wurde die Beschreibung und Kopplung von heterogenen elektronischen Systeme auf Basis des CASE Datenaustauschformats CDIF vorgenommen. Die technischen Eigenschaften wie die eindeutige Repräsentation, die Trennung zwischen Semantik und Präsentation, die Erweiterbarkeit des Meta-Modells, die Unterteilung in einzelne Subject Areas und die Trennung von Syntax und Encoding machen CDIF zu einer sehr gut geeigneten Basis für den Systementwurf. Das Datenformat wird derzeit noch weiterentwickelt und liegt im Moment zur endgültigen Standardisierung bei der ISO vor. Das in der Industrie vorhandene Interesse läßt auf eine breite Akzeptanz des endgültigen Standards schließen, nicht zuletzt weil andere Alternativen derzeit fehlen und die technische Qualität von CDIF sehr hoch ist.

Die in CDIF vorliegenden Daten können für die Analyse und Simulation eines heterogenen Gesamtsystems verwendet werden und wurden im Rahmen dieser Arbeit zur Code-Erzeugung des Ge-

samtsystems benutzt. Die damit gewonnene Unabhängigkeit von den verwendeten Modellierungswerkzeugen und die für alle Modellierungen einheitlichen Analyse-, Simulations- und Codeerzeugungsmöglichkeiten sind wesentliche Vorteile. Der erstellte Codegenerator kann für alle Modellierungswerkzeuge verwendet werden, die eine semantische Modellierung diskreter und kontinuierlicher Systeme erlauben.

Um einen hochoptimierten Echtzeit-Code aus der abgelegten CDIF-Beschreibung zu erhalten, wurde der Aufbau von Echtzeit-Code für diskrete und kontinuierliche Codeteile untersucht. Zusätzlich mußte die Synchronisation der Codeteile analysiert werden, da keine spezifischen Lösungen für das Scheduling bei gleichzeitiger Abarbeitung von diskreten und kontinuierlichen Anteilen vorliegen. Die daraus gewonnenen Erkenntnisse wurden für den Aufbau eines Compilers genutzt, der die Übersetzung in C-Code vornimmt.

Auch die Anbindung des Rapid Prototyping Systems an die Systemumgebung und die notwendige Hardware zur Unterstützung der Anbindung wurde untersucht. Exemplarisch wurden dazu die für die Automobilindustrie notwendigen Schnittstellen ermittelt und teilweise realisiert, um eine Beispielapplikation anbinden zu können. Um eine schnelle Konfiguration der Schnittstellen zu gewährleisten, wurde eine anwenderorientierte Schnittstellenvisualisierung erstellt, die die Konfiguration sämtlicher Hardwarekomponenten in Software zuläßt.

Insgesamt konnten aus der Arbeit die folgenden wissenschaftlichen Erkenntnisse und Ergebnisse erzielt und publiziert werden:

- Konzeption und Aufbau einer heterogenen Entwurfsumgebung unter Berücksichtigung der horizontalen und vertikalen Kooperation sowie der Verbindung von Modellen auf unterschiedlichen Abstraktionsebenen [BuWo98a]. Dabei wurde insbesondere auf die nahtlose Integration der Modellierung mit den Folgeschritten Analyse, Simulation, Verifikation, Rapid Prototyping und Produktion geachtet. Wiederverwendbarkeit und Bibliotheksbildung der Modellierung erleichtert den Entwurf erheblich.
- Entwicklung eines Scanners und Parsers für CASE Data Interchange Format (CDIF) Beschreibungen. Diese beiden Komponenten sind für alle existierenden und in der Zukunft erstellten CDIF-Beschreibungen verwendbar und können unverändert in alle Folgeprojekte eingebunden werden [BuWo98b].
- Beschreibung und Kopplung von Modellkomponenten auf Basis von CDIF. Dabei wurde die formale Abbildung von CDIF-Modellen für sämtliche Komponenten des diskreten und kontinuierlichen Bereichs vorgenommen. Erstmals ist es damit gelungen, ein heterogenes Gesamtsystem mit einer standardisierten Repräsentation abzubilden [BuWo98b].
- Untersuchung von Schedulingalgorithmen und deren Eignung für Rapid Prototyping. Die Vorteile von zeitgesteuertem Scheduling gegenüber ereignisgesteuertem Scheduling für heterogene Systeme mit diskreten und kontinuierlichen Komponenten wurden aufgezeigt und führt auf die weitere Untergliederung in ratenmonotones und pseudoraten-basiertes Scheduling. Nach formaler Untersuchung der beiden zeitgesteuerten Schedulingarten konnte die Eignung von pseudoraten-basiertem Scheduling vermutet werden. In Tests wurde diese Eignung für ein Rapid Prototyping System nachgewiesen [BuWo99].
- Entwicklung der Compiler für die Abbildung von STATEMATE ™ und MATRIX_X ™ Modellen in CDIF, deren Kopplung in CDIF und die Abbildung in Echtzeit-Code. Der Scheduling Algorithmus zur Kopplung der Modelle auf Ebene des Echtzeit-Codes wurde ebenfalls hergeleitet

[BuSW98]. Der Vergleich zwischen eigenem Echtzeitcode und dem Echtzeitcode der kommerziellen Werkzeuge ergab einen deutlichen Performanzvorteil des eigenen Codes.

- Aufteilung des erzeugten Echtzeit-Codes in drei Teile (Modellcode, Ein-/Ausgabecode und Schedulingcode), um einen modularen Aufbau zu erreichen [BuSW98]. Die dadurch gewonnene Unabhängigkeit führt zu einer verbesserten und vereinfachten Systemwartung.
- Entwicklung von softwarekonfigurierbarer Hardware auf Basis des VMEbus [SpBM99].

Erste Erfahrungen mit dem Rapid Prototyping System konnten bei der Entwicklung eines Fensterhebersteuergeräts mit Diebstahl– und Einklemmschutz gemäß eines industriellen Lastenhefts gewonnen werden. Durch die bei der Modellierung auftretenden diskreten und kontinuierlichen Aspekte, mußte die Applikation aus einem STATEMATE™ und einem MATRIX$_X$™ Modell aufgebaut werden. In dem Test kam ein Motorola MVME 2604 VMEbus basierendes System zum Einsatz, das mit einem 200 MHz PowerPC Prozessor 604, dem Echtzeitbetriebssystem VxWorks™ von WindRiver und eigenentwickelter Ein–/Ausgabehardware ausgestattet war. Die Ausführungszeiten liegen bei diesem Modell unterhalb von 100 µs und die Codegröße bei wenigen Kilobyte.

Die Modellbildung, Simulation und Codegenerierung für das Rapid Prototyping dieses Steuergerätes benötigte ca. 6 Personentage. Etwa zwei bis drei Iterationszyklen pro Tag können für Verbesserungen oder Erweiterungen durchgeführt und erprobt werden. Die Wirksamkeit des konzept–orientierten Rapid Prototyping zur frühzeitigen Klärung der funktionalen Ziele beim Steuergeräteentwurf konnte eindrucksvoll bestätigt werden.

10.2 Ausblick

Die vorliegende Arbeit stellt einen weiteren Schritt zur Realisierung von konzept-orientierten Rapid Prototyping Systemen dar. Mit dem bestehenden System wird die semantische Modellierung von diskreten und kontinuierlichen Systemen unterstützt. Eine Erweiterung um die Bereiche objekt-orientierte Analyse und Design (z.B. UML), Datenbankanbindung oder Projektmanagement kann vorgenommen werden und wird bereits von CDIF unterstützt. Zur Verkleinerung der Modellgrößen ist ein binäres Speicherformat für CDIF wünschenswert. Um einen zukünftigen Datenaustausch mit weiteren Systemen durchführen zu können, die CDIF unterstützen, ist ein Datenaustausch über XML sinnvoll. Die Abbildung von Syntax und Encoding zu XML ist bereits Bestandteil des CDIF Standards.

Neben der funktionalen Erweiterung des bestehenden Systems gestaltet sich auch die Einbindung und Integration weiterer CASE-Werkzeuge und die Abschätzung des Integrationsaufwands als äußerst interessant. Für die Einbindung des CASE-Werkzeugs MatLab wäre beispielsweise lediglich ein Compiler zu erstellen, der die Umsetzung in CDIF vornimmt. Alle weitere Funktionalität wie beispielsweise die Codeerzeugung wäre durch den werkzeug-unabhängigen Aufbau des Rapid Prototyping Systems bereits vorhanden.

Es hat sich gezeigt, daß konzept–orientiertes Rapid Prototyping bestens zur Entwicklung von Vorserien- oder A–Muster Steuergeräten geeignet ist und die Zusammenarbeit großer Projektgruppen nachhaltig verbessern kann. Das realisierte Rapid Prototyping System stellt eine gute Basis zur weiteren Erprobung dar und könnte in einem zukünftigen Projekt bei der Entwicklung von zeitkritischen Applikationen beispielsweise im Bereich Motormanagement eingesetzt werden.

Große Anstrengungen werden derzeit unternommen, um den Modellaustausch für ausführbare Spezifikationen unter den Gesichtspunkten Entwurf für und mit Wiederverwendung sowie ”Intellectual Property” voranzutreiben und die automatische Codegenerierung für B– und C–Muster ”serientauglich” zu machen. Die bereits erzielten Fortschritte sind beachtlich und zeigen den Weg zu einer weiteren Verkürzung der Entwicklungszeiten und einer Steigerung der Qualität beim Entwurf auf. Auf diese Weise kann den Herausforderungen begegnet werden, die neue sicherheitsrelevante Funktionen wie beispielsweise gear/brake/steer–by–wire im Automobilbereich mit sich bringen. Die noch immer bestehende Lücke zwischen der Codeerzeugung für Rapid Prototyping und der Seriencodeerzeugung erfordert eine weitere Optimierung der Codeerzeugung. Hochoptimierte Codegeneratoren versprechen zwischenzeitlich ein Faktor von 1,3 selbst bei komplexen Aufgabenstellungen im Motormanagementbereich [Etas99].

Aufstrebende Folgetechnologien wie formale Verifikation und modellbasierte Diagnose können ebenfalls in die Entwurfsumgebung integriert werden und erlauben auf diese Weise eine weitere Verbesserung des Entwurfsablaufs.

Literaturverzeichnis

[AlHS95] Ålemark, N.; Haeberlein, C.; Steinke, C.: *Programmierung und Inbetriebnahme einer Kopplung zum Datenaustausch zwischen einer grafischen Oberfläche und einem Rapid Prototyping System*. Team-Studienarbeit ILT-458, Universität Karlsruhe, ITIV, 1995.

[Anal99] Analogy, Inc.: *VHDL–AMS Industry Standards*. http://www.vhdl–ams.com, 1999.

[BiLa93] Biesenack, J.; Langmaier, A.; Pilsl, M.; Rumler, S.; Wehn, N.: *High-Level VHDL Optimization, Tranformation and Analysis.* Siemens AG, 1993.

[BöDK94] Böckle, K.; Doll, G.; Kloepfer, T.; Volz, M.: *Realisierung eines Rapid Prototyping Systems.* Team–Studienarbeit ILT-459, Universität Karlsruhe, ITIV, 1994.

[Boeh76] Boehm, B.: *Software Engineering.* IEEE Transactions on Computers, Vol. 25, 1976.

[Boeh86] Boehm, B.: *A Spiral Model of Software Development and Enhancement.* ACM Sigsoft Software Engineering Notes, Vol. 11, 1986.

[Booc94] Booch, G.: *Object–oriented analysis and design with applications.* Benjamin/Cummings, 1994.

[BoJa97] Booch, G.; Jacobson, I.; Rumbaugh, J.: *UML Summary.* Rational Software Corp., http://www.rational.com, 1997.

[BoHi96] Bortolazzi, J.; Hirth, T.; Raith, T.: *Specification and Design of Electronic Control Units.* Proc. EURO–DAC, 1996.

[Bort94] Bortolazzi, J.: *Untersuchungen zur rechnergestätzten Erfassung, Verwaltung und Prüfung von Anforderungsspezifikationen und Einsatzbedingungen elektronischer Steuerungs- und Regelungssysteme*. Dokotorarbeit, Universität Erlangen-Nürnberg, 1994.

[BuSc97] Burst, A.; Schmerler, S.; Spitzer, B.; Tanurhan, Y.: *Rapid Prototyping of Electronic Systems - Emulation of Electronic Systems for Fast Definition and Verification of System Requirements.* Studie, Institut für Technik der Informationsverarbeitung und FZI–ESM, Karlsruhe, Deutschland, September 1997.

[BuSW98] Burst, A.; Spitzer, B.; Wolff, M.; Müller-Glaser, K.: *On Code Generation for Rapid Prototyping Using CDIF.* Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA, Vancouver, Canada, 1998.

[BuWo98a] Burst, A.; Wolff, M.; Kühl, M.; Müller-Glaser, K.: *A Rapid Prototyping Environment for the Concurrent Development of Mechatronic Systems.* European Concurrent Engineering Conference ECEC, Erlangen, Germany, 1998.

[BuWo98b] Burst, A.; Wolff, M.; Kühl, M.; Müller-Glaser, K.: *Using CDIF for Concept-Oriented Rapid Prototyping of Electronic Systems.* Rapid System Prototyping Conference RSP, Leuven, Belgium, 1998.

[BuWo99] Burst, A.; Wolff, M.; Kühl, M.; Müller-Glaser, K.: *Scheduling Strategies and Estimations for Concept-Oriented Rapid Prototyping*. Rapid System Prototyping Conference RSP, Tampa, Florida, USA, 1999.

[CADF95] CAD Framework Initiative: *Tool Encapsulation Specification*. http://www.cfi.org/TES, 1995.

[Calv93] Calvez, J.: *Embedded Real-Time Systems.* John Wiley & Sons, New York, 1993.

[Chen76] Chen, P.: *The Entity-Relationship Model - Towards a Unified View of Data*. ACM Transactions on Database Systems, Vol.1, No.1, March 1976, p. 9-36.

[ChEr93] Chowanetz, M.; Ernst, J.: *Rapid Prototyping unter Einsatz von Werkzeugen zur Systemspezifikation und zur Systemsimulation wie i–Logix Statemate.* Universität Erlangen–Nürnberg, Lehrstuhl für Rechnergestützten Schaltungsentwurf, Studie, 1993.

[CoHu93] Cooling, J.; Hughes, T.: *Animation Prototyping of Real-Time Embedded Systems*. Microprocessors and Microsystems, No.6, 1993.

[DeMa79] DeMarco, T.: *Structured Analysis and System Specification.* Englewood Cliffs, Yourdon Press, 1979.

[Doll95] Doll, G.: *Partitionierung und Abbildung eines mittels CDIF beschriebenen reaktiven Modells auf ein Multiprozessor–Rapid Prototyping System*. Diplomarbeit ID-714, Universität Karlsruhe, ITIV, 1995.

[ECMA99] European Computer Manufacturers Association: *Standardizing Information and Communication Systems*. http://www.ecma.ch, 1999.

[EIA94] EIA/CDIF Technical Committee: *CDIF / CASE Data Interchange Format.* EIA Interim Standard EIA/IS– 106–112, 1994.

[ElMa97] Elmquist, H.; Mattsson, S.: *Modelica – The next Generation Modeling Language.* Proc. of the 1st world Congress on System Simulation, 1997.

[Eppi93] Eppinger, A.: *Computer–Integrated System Technology with ASCET.* Ph.D. thesis, University of Paderborn, 1993.

[Erns96] Ernst, J.: *Structured Analysis and Control Design: A Unified Approach.* Proc. ISATA, Florence, Italy, 1996.

[Erns97] Ernst, J.: *An Open Simulation Architecture for the Development of Complex Embedded Systems Using Distributed Objects.* Proc. SAE, Detroit, USA, 1997.

[Etas99] ETAS GmbH: *Toolkette im Einsatz*. Real Times 1, 1999.

[Fabe97] Faber, R.: *Entwicklung eines dynamischen Modells für einen Kraftfahrzeug–Fensterheber.* Diplomarbeit ID-810, Universität Karlsruhe, ITIV, 1997.

[Fisc88] Fischer, J.: *Softwaretechnologie 'Rapid Prototyping' bei der Entwicklung von Kommunikationssoftware verteilter sowie eingebetteter Systeme.* Dissertation, Humboldt-Universität Berlin, 1988.

[Flat98] Flatscher, R.: *Meta-Modellierung in EIA/CDIF.* ADV–Verlag, Wien 1998.

[Föll85] Föllinger, O.: *Regelungstechnik: Einführung in die Methoden und ihre Anwendung.* Hüthig, 1985.

[Friz97] Friz–Jung, M.: *Entwicklung und Realisierung einer automatischen Codegenerierung für Mikrocontroller basierend auf einem CASE Datenaustauschformat*. Diplomarbeit ID-802, Universität Karlsruhe, ITIV, 1997.

[GaKK95] Gack, D.; Knüppel, A.; Kock, J.; Meier, W.; Mews, M.: *Entwicklung und Aufbau einer Testumgebung für ein Rapid Prototyping System*. Team–Studienarbeit ILT-459, Universität Karlsruhe, ITIV, 1995.

[Gery93] Gery, E.: *Porting Statemate generated Code to Embedded Environments*. Internal Paper, I-logix, 1993.

[Grei96] Greiner, C.: *Untersuchungen zur Kopplung von diskreten und kontinuierlichen Modellen für ein Rapid Prototzping System*. Studienarbeit IL-470, Universität Karlsruhe, ITIV, 1996.

[Grei97] Greiner, C.: *Entwurf und Implementierung eines Konzepts dynamisch integrierbarer Komponenten einer Rapid Prototyping Entwicklungsplattform unter Berücksichtigung des kooperativen Systementwurfs*. Diplomarbeit ID-804, Universität Karlsruhe, ITIV, 1997.

[Hare87] Harel, D.: *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming, 8, 1987.

[HaGe97] Harel, D.; Gery, E.: *Executable Object Modeling with Statecharts.* IEEE Computer, Nr.7, 1997.

[Harr98] Harrison, J.: *Intergration of CAE Tools for Complete System Prototyping*. Society of Automotive Engineerings SAE, 1998.

[Hery95] Herynek, R.: *Gewinnung und Implementierung von Funktionen zur Auswertung des Druckverlaufs im Brennraum für ein Kfz–Motorsteuersystem*. Diplomarbeit ID-704, Universität Karlsruhe, ITIV, 1995.

[HLNP90] Harel, D.; Lachover, H.; Naamad, A.; Pnuelli, A.; et.al.: *STATEMATE: A Working Environment for the Development of Complex Reactive Systems*. IEEE Transactions on Software Engineering, Vol. 16, No. 4, 1990.

[Hart97] Hartmann, N.: *Konzeption und Implementierung des Systemkerns einer vollintegrierten, verteilten und dynamisch skalierbaren Rapid Prototyping Entwicklungsplattform.* Diplomarbeit ID-805, Universität Karlsruhe, ITIV, 1997.

[HeEd90] Henderson-Sellers, B.; Edwards, J.: *The Object_Oriented Systems Life Cycle.* Communications of the ACM, 33, 1990.

[Hodg93] Hodgson, R.: *Das X-Modell - Prozeß.* Systeme, 6, 1993.

[IABG99] IABG: *Das V–Modell*. http://www.v–modell.iabg.de, 1999.

[IsLa97] Isazadeh, H.; Lamb, D.: *CASE Environments and MetaCASE Tools*. Technical report 1997–403, Department of Computing and Information Science, Queen's University, Kingston, 1997.

[ISO99] International Organization for Standardization: *The STEP Project*. http://www.nist.gov/sc4/www/stepdocs.htm, 1999.

[Jaco92] Jacobson, I.: *Object–oriented Software Engineering : A Use Case Driven Approach.* Addison–Wesley, 1992.

[JäKl91] Jäker, K.–P.; Klingebiel, P.: *Tool Integration by way of a Computer–Aided Mechatronic Laboratory (CAMeL).* 5th IFAC/IMACS Symposium on Computer Aided Design in Control Systems, Swansea, Wales, 1991.

[KaAS93] Katcher, D.; Arakawa, H.; Strosnider, J.: *Engineering and Analysis of Fixed Priority Schedulers.* IEEE Transactions on Software Engineering, Vol. 19, No. 9, 1993.

[Kloe95] Kloepfer, T.: *Aufbau und Programmierung einer Mikrocontrollerplatine zur Emulation eines Busprotokolls.* Diplomarbeit ID-719, Universität Karlsruhe, ITIV, 1995.

[Kock97] Kock, J.: *Aufbau, Optimierung und Hardwareanbindung von Echtzeitcode zur Emulation von diskret–kontinuierlichen Modellen für ein Rapid Prototyping System*. Diplomarbeit ID-800, Universität Karlsruhe, ITIV, 1997.

[Kühl97] Kühl, M.: *Untersuchungen zur Integration eines CAD–Datenaustauschformats in ein Rapid Prototyping Entwicklungssystem für den Entwurf von mechatronischen Systemen.* Diplomarbeit ID-831, Universität Karlsruhe, ITIV, 1997.

[KuBo93] Kurpis, G.; Booth, C.: *The New IEEE Standard Dictionanry of Electrical and Electronics Terms.* New York, 1993.

[Laie95] Laier, J.: *Portierung von Statemate-Code auf den Mikrocontroller C167 unter Verwendung eines Echtzeitbetriebssystems*. Diplomarbeit ID-712, Universität Karlsruhe, ITIV, 1995.

[Lano90] Lano, R.: *The N^2-Chart*. In: Thayer, R.; Dorfman, M.: *System and Software Requirements Engineering*. IEEE Computer Society Press Tutorial, IEEE Computer Society Press, Los Alamitos, California, 1990.

[Lawl83] Lawler, E.: *Recent Results in the Theory of Machine Scheduling*. In: Bachen, A.: Mathematical Programming: The State of the Art; Springer, New York, 1983.

[LeSD89] Lehoczky, J.; Sha, L.; Ding, Y.: *The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior.* Proc. Real-Time Systems Symp., IEEE Computer Society Press, Los Alamitos, 1989.

[Lipp95] Lipp, H.M.: *Grundlagen der Digitaltechnik*. Oldenbourg Verlag, München, 1995.

[LiLa73] Liu, C.; Layland, J.: *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment.* Journal of the Association for Computer Machinery, Vol. 20, 1973.

[Meal55] Mealy, G.: *A Method of Synthesizing Sequential Circuits.* Bell System Techn. Journal, Bd. 34, 1955.

[Meie97] Meier, W.: *Kopplung von diskreten und kontinuierlichen Modellen auf Basis eines CASE–Datenaustauschformats und Codegenerierung für ein echtzeitfähiges Rapid Prototyping System*. Diplomarbeit ID-801, Universität Karlsruhe, ITIV, 1997.

[MACT89] Mok, A.; Amerasinghe, P.; Chen, M.; Tantisirivat, K.: *Evaluating Tight Execution Time Bounds of Programs by Annotation*. Proceedings of the 6th IEEE Workshop on Real-Time Operating Systems and Software, 1989.

[Moor56] Moore, E.: *Gedanken-Experiments on Sequential Machines*. Automata Studies (Annals of Mathematics Studies 34), Princeton, 1956.

[Müll94] Müller-Glaser, K.: *Die goldene Mitte. "Meet in the Middle"-Strategie - Kopplung von rechnergestützten Werkzeugen.* Elektronik 21/23, 1994.

[MBSS99] Müller-Glaser, K.; Burst, A.; Spitzer, B.; Schmerler, S.: *Rapid Prototyping von Informationssystemen für Kraftfahrzeuge*. Informationstechnik und technische Informatik it+ti, Oldenbourg, 05, 1999.

[MBSK00] Müller-Glaser, K.; Burst, A.; Spitzer, B.; Kühl, M.: *Rapid Prototyping von eingebetteten elektronischen Systemen*. Informationstechnik und technische Informatik it+ti, Oldenbourg, 02, 2000.

[Obje99] Object Management Group: *Welcome to the OMG's site for CORBA and UML Success Stories!*. http://www.corba.org, 1999.

[OSEK97] OSEK/VDX Technical Committee: *OSEK/VDX Operating System Specification*. http://www.osek–vdx.org, 1997.

[PaSh91] Park, C.; Shaw, A.: *Experiments with a Program Timing Tool Based on Source-Level Timing Schema.* Advances in Real-Time Systems, IEEE Computer Society Press, Los Alamitos, 1991.

[Petr62] Petri, C.: *Fundamentals of a Theory of Asynchronous Information Flow.* Proc. of IFIP Congress, 1962.

[Ratc88] Ratcliffe, B.: *Early and not-so early prototyping - rationale and support.* Proceedings COMPSAC88, IEEE Computer Society Press, 1988.

[ReLe94] Rembold, U.; Levi, P.: *Realzeitsysteme zur Prozeßautomatisierung*. Hanser, 1994.

[RBKG00] Renner, F., Becker, J.; Kirschbaum, A.; Glesner, M.: *Architekturgenaues Rapid Prototyping von eingebetteten Systemen mit harten Echtzeitbedingungen*. Informationstechnik und technische Informatik it+ti, Oldenbourg, 01, 2000.

[Royc70] Royce, W.: *Managing the Development of Large Software Systems.* IEEE Wescon, New York, 1970.

[RuBl91] Rumbaugh, J.; Blaha, M.: *Object–Oriented Modeling and Design.* Prentice Hall, 1991.

[Schn98] Schnellbacher, K.: *Optimierung der Kopplung von diskreten und kontinuierlichen Modellen auf Basis eines CASE Datenaustauschformats für ein echtzeitfähiges Rapid Prototyping System.* Diplomarbeit ID-838, Universität Karlsruhe, ITIV, 1998.

[Schm98] Schmerler, S.: *Prädikative Methoden für optimistische Synchronisationsprotokolle in der verteilten Simulation.* Dokotorarbeit, Forschungszentrum Informatik Publikation 3/98, 1998.

[Schu98] Schulz, S.: *The New System-Level Design Language.* Integrated System Design Magazine, http://www.isdmag.com/Editorial/1998/RTLFeature9807.html, 1998.

[ScTa95] Schmerler, S.; Tanurhan, Y.; Müller–Glaser, K.: *A Backplane Approach for Cosimulation in High–Level System Specification Environments.* Proc. EURO–DAC, 1995.

[Spit95] Spitzer, B.: *Planung und Realisierung von softwarekonfigurierbaren analogen und digitalen Ein-/Asugabekarten auf VMEbus- Basis für ein Rapid Prototyping System*. Diplomarbeit ID-700, Universität Karlsruhe, ITIV, 1995.

[SpBM99] Spitzer, B.; Burst, A.; Müller–Glaser, K.: *Interface Technologies for Versatile Rapid–Prototyping Systems*. Rapid System Prototyping Conference RSP, Tampa, Florida, USA, 1999.

[Spre95] Spreng, M.: *Rapid Prototyping for Automotive System Development.* Proc. SAE, Detroit, 1995.

[Spre96] Spreng, M.: *Rapid Prototyping elektronischer Steuerungssysteme in der Automobilentwicklung*. Dokotorarbeit, Universität Karlsruhe, 1996.

[SSDB95] Stankovic, J.; Spuri, M.; Di Natale, M.; Buttazzo, G.: *Implications of Calssical Scheduling Results for Real-Time Systems*. IEEE Computer, 6, 1995.

[Stoy87] Stoyenko, A.: *A Real-Time Language with a Schedulability Analyzer.* Dissertation, Computer Science Research Institute, University of Toronto, 1987.

[Teic97] Teich, J.: *Digitale Hardware/Software Systeme - Synthese und Optimierung.* Springer-Verlag, Berlin, 1997.

[Turn99] Turner, R.: *System-Level Verification - A Comparison of Approaches.* Rapid System Prototyping Conference RSP, Tampa, Florida, USA, 1999.

[Volz95] Volz, M.: *Konzeption und Realisierung eines Multiprozessor–Rapid Prototyping Systems auf Basis einer CDIF–Modellbeschreibung*. Diplomarbeit ID-713, Universität Karlsruhe, ITIV, 1995.

[WaTh85] Walker, R.; Thomas, D.: *A Model of Design Representation and Synthesis.* 22nd Design Automation Conference, Las Vegas, 1985.

[WaRW95] Wang, F.; Richards, B.; Wright, P.: *A Generic Framework for the Integration of Mechanical and Electrical CAD Tools for Concurrent Design of Consumer Electronic Products.* Proc. ASME 21th Design Automation Conference, 1995.

[WeRG95] Weisser, M.; Rüger, B.; Geisweis, H.: *Richtig in die Gänge kommen – Durch Rapid Prototyping schneller mit der Funktionsidee ins Auto*. Elektronik, 24, 1995.

[WiCo96] Wietzke, J.; Cochlovius, E.: *Kompakter Controller-Code aus Statemate-Modellen*. Elektronik, 20, 1996.

[Will97] Willis, J.: AIRE/CE: *Advanced Intermediate Representation with Extensibility / Common Environment. Internal Intermediate Representation Specification.* http://www.vhdl.org/aire, 1997.

[WUSL86] Wunsch, G; Unbehauen, R.; Schreiber, H.; Locke, M.; Goessel, M.; Bochmann, D.; Strobel, H.; Schwarze, G.: *Handbuch der Systemtheorie.* Akademie-Verlag Berlin, Hrsg. G. Wunsch, 1986.

[Your89] Yourdon, E.: *Modern Structured Analysis.* Prentice–Hall, 1989.

[Zale95] Zalewski, J.: *What Every Engineer Needs to Know about Rate-Monotonic Scheduling: A Tutorial.* In: Zalewski, J.: Advanced Multimicroprocessor Bus Architectures, IEEE Computer Society Press, Los Alamitos, 1995.

Formelbezeichnungen

α	:	zur Ausführung zur Verfügung stehende Maschine (nach Lawler)
β	:	charakteristische Prozeßmerkmale
c	:	Maschinenkonstante
c_{alg}	:	Faktor zur Einbeziehung des gewählte Integrationsalgorithmus
d_i	:	Grad einer Differentialgleichung (Dimension des Zustandsraums)
δ	:	Übertragunsfunktion eines endlichen Automaten
δ	:	Inkrement der Simulationszeit
δ	:	Auflösung einer Prozeßschar mit unterschiedlichen Perioden
γ	:	Optimierungsmethode (Metrik) der Abarbeitung
i	:	elektrischer Strom (allgemein)
indpt	:	Prozeßabarbeitung bei Unabhängigkeit der Daten einzelner Prozesse
λ	:	Ausgabefunktion eines endlichen Automaten
pmnt	:	Prozeßabarbeitung mit unterbrechbaren Prozessoren
prec	:	Durchführung einer prioritätsbasierten Abarbeitung
ϕ	:	magnetischer Fluß
p_i	:	Stelle eines Petri-Netzes
q_i	:	Anzahl der abzuarbeitenden Ereignisse während eines Modellschritts
resrc	:	Prozeßabarbeitung mit limitierten Ressourcen
r_i	:	Anzahl der abzuarbeitenden Bedingungen während eines Modellschritts
s_i	:	Anzahl der auszuführenden Aktionen während eines Modellschritts
t_i	:	Transition eines Petri-Netzes
τ	:	maximale Abweichung differierender Signale
u	:	elektrische Spannung (allgemein)
$\underline{u}$	:	Zustandsvektor eines Zustandsraums
$\underline{x}$	:	Eingangsvektor eines Zustandsraums
$\underline{y}$	:	Ausgangsvektor eines Zustandsraums
A	:	endliches Ausgabealphabet eines Automaten
A	:	Ankunftszeit eines Prozesses
$\underline{A}$	:	erste Matrix der Eingangsgleichung der Frobenius-Matrix
$\underline{B}$	:	zweite Matrix der Eingangsgleichung der Frobenius-Matrix
C	:	elektrische Kapazität (allgemein)
C	:	Ausführungszeit eines Prozesses
C_A	:	Ausführungszeit einer Aktion
C_C	:	Ausführungszeit einer Bedingung

C_{det} : zusammengefaßte Zeit innerhalb einer Periode der kontinuierlichen Belastung zur Bestimmung der aktuellen Systemzeit

C_{dot} : Zeit zur Berechnung aller Ableitungen eines kontinuierlichen Modellschritts

C_E : Ausführungszeit eines Ereignisses

C_{exit} : Zeit zur Beendigung eines Prozesses und Laden des nächster Prozesses

C_{exp} : gemeinsame Ausführungszeit aller algebraischen Verknüpfungen

C_i : Ausführungszeit eines Prozesses P_i

$C_{i,Amax}$: Antwortzeit zur Abarbeitung aller durchzuführenden Aktionen eines Modellschritts

$C_{i,EC}$: Abfragezeit zur Überprüfung, welche Ereignisse und Bedingungen innerhalb eines Modellschritts erfüllt sind

$C_{i,max}$: maximale Zustandsübergangszeit eines Systemzustands

C_{int} : Zeit für die Verarbeitung eines Interrupts

C_{integ} : Zeit zur Berechnung einer Integration

C_{load} : Zeit für das Laden der Umgebung des aktuellen Prozesses

C_{model} : Zeit für die Abarbeitung eines Modellschritts

C_{out} Zeit zur Berechnung der Ausgangsvariablen

$C_{preempt}$: Zeit für die Unterbrechung eines Prozesses

C_{resume} Zeit für die Wiederaufnahme eines Prozesses bei vorheriger Unterbrechung durch einen Interrupt

C_{sched} : Zeit des Schedulers zur Bestimmung des nächsten aktiven Prozesses

C_{store} : Zeit für die Speicherung der Umgebung des aktuell unterbrochenen Prozesses

C_{timer} : Zeit für den Start und das nebenläufige Betreiben eines Timers

$C_{timer,aux}$: Zeit für den Betrieb eines Auxiliary-Timers

$C_{timer,wd}$: Zeit für den Betrieb eines Watchdog-Timers

C_{trap} : Zeit zur Beendigung eines Prozesses

$\underline{C}$: erste Matrix der Ausgangsgleichung der Frobenius-Matrix

D : Frist eines Prozesses

D_i : spätestmögliche Endzeit eines Prozesses P_i

$\underline{D}$: zweite Matrix der Ausgangsgleichung der Frobenius-Matrix

Δ : infinitesimal kleine Zeitspanne

E : endliches Eingabeanphabet eines Automaten

F : Flußrelation eines Petri-Netzes

F : Abschlußzeit eines Prozesses

$G(s)$	:	Übertragungsfunktion eines kontinuierlichen Systems
I_A	:	Strom des Ankerkreis einer Gleichstrommaschine
I_f	:	Feldstrom einer Gleichstrommaschine
J	:	Trägheitsmoment (allgemein)
K	:	Menge der Kapazitäten eines Petri-Netzes
K_D	:	Verstärkungsfaktor der Übertragunsfunktion eines Differenzierglieds
K_I	:	Verstärkungsfaktor der Übertragunsfunktion eines Integrierglieds
K_P	:	Verstärkungsfaktor der Übertragunsfunktion eines Proportionalglieds
L_A	:	Induktivität der Ankerwicklung einer Gleichstrommaschine
L_{max}	:	maximale Verspätung eines Prozesses
M_0	:	Menge der Anfangsmarkierungen eines Petri-Netzes
M_B	:	Beschleunigungsmoment
M_i	:	von einer Gleichstrommaschine erzeugtes Drehmoment
M_L	:	Lastmoment
N	:	Anzahl von maximal benötigten Filter
$\mathbb{N}$	:	Menge der natürlichen Zahlen
$\mathbb{N}_0$	:	Menge der natürlichen Zahlen inklusive Null
P	:	Menge der Stellen eines Petri-Netzes
P	:	Prozeß (allgemein)
P_i	:	Prozeß
$Prio_i$	:	Priorität eines Prozesses P_i
Ω	:	Winkelgeschwindigkeit
R	:	elektrischer Widerstand (allgemein)
R	:	Bereitszeit eines Prozesses
R_A	:	ohmscher Widerstand der Ankerwicklung einer Gleichstrommaschine
R_f	:	Feldwiderstand einer Gleichstrommaschine
R_i	:	frühestmögliche Startzeit eines Prozesses P_i
$\mathbb{R}$	:	Menge der reellen Zahlen
$\mathbb{R}_+$	:	Menge der postitiven reellen Zahlen ohne Null
$\mathbb{R}_-$	:	Menge der negativen reellen Zahlen ohne Null
S	:	endlicher Speicher eines Automaten
S	:	Startzeit eines Prozesses
S_i	:	Schdulingzeitpunkte eines Prozesses P_i
S^ν	:	aktueller Speicherzustand eines Automaten
Sp_i	:	sporadisches Attribut eines Prozesses P_i
T	:	Menge der Transition eines Petri-Netzes

T	:	Periode eines Prozesses
T_{aux}	:	Periode eines Auxiliary-Timers
T_i	:	Periode eines Prozesses P_i
T_{timer}	:	Periode eines Timers
$T_{timer,block}$	:	Periode, an der sich die Blockierung des höchstprioren Prozesses auswirkt
$T_{timer,max}$	:	maximale zeitliche Auflösung der Periode eines Timers (bei Blockierung)
$T_{timer,min}$	:	minimale zeitliche Auflösung der Periode eines Timers
$T_{timer,opt}$	:	optimale zeitliche Auflösung der Periode eines Timers
T_{wd}	:	Periode eines Watchdog-Timers
U	:	Prozessorausnutzung
U_A	:	Spannung des Ankerkreis einer Gleichstrommaschine
U_i	:	induzierte Spannung im Ankerkreis einer Gleichstrommaschine
U_f	:	Feldspannung einer Gleichstrommaschine
V_i	:	Verdrängungsattribut eines Prozesses P_i
W	:	Menge der Kantengewichte eines Petri-Netzes
$W_i(t)$	:	gesamte Zeit, die zur Abarbeitung aller höherprioren Prozesse von P_i benötigt wird

Verzeichnis der Abkürzungen

ABS	:	Anti Blockier System
A/D	:	Analog/Digital
AFAP	:	As Fast as Possible (siehe Scheduling)
AIRE	:	Advanced Intermediate Representation with Extensibility/Common Environment
ASCII	:	American Standard Code for Information Interchange
ASIC	:	Application Specific Integrated Circuit
AUX	:	Auxiliary–Timer
BDL	:	Block Description Language (siehe BEACON)
BNF	:	Backus Naur Form
CACSD	:	Computer Aided Control System Design Subject Area (siehe CDIF)
CAD	:	Computer Aided Design
CAE	:	Computer Aided Engineering
CAMeL	:	Computer–Aided Mechatronic Laboratory
CAN	:	Controller Area Network (Datenbus)
CASE	:	Computer Aided Software/Systems Engineering
CDIF	:	CASE Data Interchange Format
CFI	:	CAD Framework Initiative
CORBA	:	Common Object Request Broker Architekture
CPU	:	Central Processing Unit
D/A	:	Digital/Analog
DDEF	:	Data Definition Subject Area (siehe CDIF)
DFG	:	Deutsche Forschungsgemeinschaft
DFM	:	Data Flow Modeling Subject Area (siehe CDIF)
DMOD	:	Data Modeling Subject Area (siehe CDIF)
DSC	:	Dynamic System Code (siehe CAMeL)
DSP	:	Digital Signal Processor
DUCADE	:	Domain Unified CAD Environment
EASY5	:	Engineering Analysis System Version 5 (CASE-Werkzeug)
EDA	:	Electronic Design Automation
EIA	:	Electronic Industry Association
ER	:	Entity Relationsship
ESP	:	Elektronisches Stabilitätsprogramm
FIR	:	File Intermediate Representation (siehe AIRE)
FIR Filter	:	Finite Impluse Response Filter

FPGA	:	Field Programmable Gate Array
FSM	:	Finite State Machines
GAL	:	Generic Array Logic
HA	:	Halbaddierer
HDL	:	Hardware Description Language
HIL	:	Hardware-in-the-Loop
HTML	:	Hypertext Markup Language
IC	:	Integrated Circuit
IEEE	:	Institute of Electrical and Electronics Engineers
IIR	:	Internal Intermediate Representation (siehe AIRE)
IRDS	:	Information Resource Dictionary System
ISO	:	International Organization for Standardization
LIP	:	Linear Interpolation
MERPS	:	Multiprozessor Echtzeit Rapid Prototyping System
ODSL	:	Objective-Dynamic System Language (siehe CAMeL)
ODSS	:	Objective-Dynamic System Structure (siehe CAMeL)
OMG	:	Object Management Group
OMT	:	Object Modeling Technique
OOAD	:	Object Oriented Analysis and Design Subject Area (siehe CDIF)
OSEK	:	Open Systems and the Corresponding Interfaces for Automotive Electronics
PCB	:	Printed Circuit Board
PCTE	:	Portable Common Tool Environment
PHS	:	Peripheral High-Speed (siehe dSPACE)
PLaC	:	Presentation Location and Connectivity Subject Area (siehe CDIF)
PR	:	Pseudoraten-basiert (siehe Scheduling)
PROCORS	:	Prototype Code for Real-Time Systems
RAM	:	Random Access Memory
RK	:	Runge-Kutta Algorithmus
RM	:	Ratenmonoton (siehe Scheduling)
RP	:	Rapid Prototyping
RSP	:	Rapid System Prototyping
RTOS	:	Real-Time Operating System
SA	:	Subject Area (siehe CDIF)
SEW	:	System Engineering Workbench
SLDL	:	System-Level Design Language
SPARC	:	Scalable Processor Architecture
SPICE	:	Simulation Program with Integrated Circuit Emphasis
STEP	:	Standard for the Computer-Interpretable Representation and Exchange of computer data (ISO 10303)

STEV	:	State Event Subject Area (siehe CDIF)
STM	:	STATEMATE (CASE-Werkzeug)
SUN	:	Stanford University Network (Unternehmen der Computer-Branche)
SUT	:	System under Test
Tcl	:	Tool Command Language
TES	:	Tool Encapsulation Specification
Tk	:	Toolkit
TTL	:	Transistor–Transistor Logic
UML	:	Unified Modelling Language
Unix	:	Betriebssystem für Workstations
VA	:	Volladdierer
VHDL	:	VHSIC Hardware Description Language
VHDL-AMS	:	VHDL Analog and Mixed Signal
VHSIC	:	Very High Speed IC
VME	:	Versa Module Europa (versatil = vielfältig)
WD	:	Watchdog-Timer
WWW	:	World Wide Web
XMI	:	XML Metadata Interchange Format
XML	:	Extended Markup Language
XNF	:	Xilinx Netlist Format

Bildverzeichnis

Tabellenverzeichnis

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Lebenslauf

Name	:	Alexander Burst
Geburtsdatum	:	21. Juni 1967
Geburtsort	:	Karlsruhe
Staatsangehörigkeit	:	deutsch
Familienstand	:	verheiratet
1973 - 1977	:	Werner-von-Siemens Schule, Karlsruhe
1977 - 1986	:	Humboldt-Gymnasium Karlsruhe
10. Juni 1986	:	Allgemeine Hochschulreife
Oktober 1986 - Oktober 1987	:	Grundwehrdienst 2./Fernmeldebattaillon 230, Dillingen a.d. Donau
Oktober 1987 - September 1993	:	Diplomstudiengang Elektrotechnik an der Universität Karlsruhe (TH)
7. September 1993	:	Abschluß: Dipl-Ing.
Januar 1994 - Mai 2000	:	Wissenschaftlicher Angestellter am Institut für Technik der Informationsverarbeitung, Universität Karlsruhe (TH)
15. Februar 2000	:	Promotion: Dr.-Ing.
ab Juni 2000	:	Mitarbeiter der Firma ETAS, Stuttgart Feuerbach, Deutschland

Karlsruhe, den 25. April 2000

Institut für Technik der Informationsverarbeitung
Universität Karlsruhe
Engesserstr. 5 · D - 76131 Karlsruhe