# Entwurf und Implementierung einer formalen Sprache zur Beschreibung dynamischer Modelle

Peter Eschenbacher

1989

Dissertation

erschienen bei:
Institut für Mathematische Maschinen
und Datenverarbeitung
Arbeitsberichte Band 23 Nummer 1
ISSN 0344-3515

## Vorwort

Am Institut für Mathematische Maschinen und Datenverarbeitung (IMMD IV) der Universität Erlangen-Nürnberg besteht seit den siebziger Jahren eine Arbeitsgruppe, die sich mit Systemanalyse, Modellbildung und Simulation befaßt, und von Prof. Dr. B. Schmidt geleitet wird. Die langjährige Beschäftigung mit Modellen aus den unterschiedlichen wissenschaftlichen Disziplinen und ihrer Simulation auf dem Digitalrechner hat gezeigt, daß die Vorgehensweise bei der Modellbildung und bei der Durchführung von Modelluntersuchungen trotz der Verschiedenheit der Fachgebiete weitgehend identisch ist.

Diese Erfahrung hat uns ermutigt, ein Projekt (SIMPLEX-II) ins Leben zu rufen, das sich zum Ziel gesetzt hat, einen Anwender bei Modelluntersuchungen durch geeignete Software optimal zu unterstützen. Basierend auf einer gemeinsamen methodologischen Grundlage ist diese Software fachrichtungsunabhängig. Es ist unser Anliegen, daß der Umgang mit dieser Software eine einheitliche Denkweise und damit die Zusammenarbeit zwischen unterschiedlichen Fachrichtungen fördert.

SIMPLEX-II ist ein Simulationssystem, das alle zur Durchführung von Simulationsstudien erforderlichen Funktionen unter einer gemeinsamen Bedienoberfläche vereint. Für das Anlegen von Modellen und die Ausführung von Experimenten realisierte mein Kollege K.-J. Langer eine Modellerstellungs- und Experimentierumgebung. Mein Kollege K. Dörnhöfer sorgte für die Funktionen zum Ablegen, zur Auswertung und zur Darstellung der Ergebnisdaten. Mein Beitrag war der Entwurf und die Implementierung einer formalen Sprache zur Beschreibung von Simulationsmodellen. Aus einer Modellbeschreibung wird durch automatische Generierung ein lauffähiges Simulationsprogramm erzeugt.

Jede Kommunikation über Erkenntnisse einer Wissenschaft geschieht in der Weise, die eigene modellhafte Vorstellung über einen Erkenntnisgegenstand einem anderen Wissenschaftler zugänglich zu machen. Es war daher mein Anliegen, die Sprache so abzufassen, daß beschriebene Modelle leicht verständlich sind und sich damit für eine Verständigung zwischen Wissenschaftlern eignen, insbesondere auch dann, wenn sie verschiedenen Disziplinen angehören. Um diesen Ansprüchen gerecht zu werden, ist eine solide methodologische

Grundlage, eine klare und eindeutige Spezifikation der Sprache sowie eine korrekte und effiziente Ausführung der generierten Simulationsmodelle erforderlich.

Die Grundkonzepte der dabei entstandenen Modellbeschreibungssprache SIMPLEX-MDL beruhen auf Überlegungen, die im Rahmen einer allgemeinen mathematischen Systemtheorie in den letzten 30 Jahren entstanden sind. Dies betrifft insbesondere die Darstellung und Auswertung der Modellgleichungen sowie die Zerlegung eines Modells in Komponenten.

Im Sinne der Informatik verkörpert diese Sprache keine algorithmische Sprache, sondern eine Spezifikationssprache. Eine Modellspezifikation besteht demnach aus einer Menge von deklarierten Objekten (Modellgrößen und Komponenten) und einer Menge funktionaler Zusammenhänge zwischen diesen Objekten.

Um aus einer in dieser Sprache abgefaßten Modellbeschreibung ein fertiges Simulationsprogramm zu erhalten, entstand ein Compiler, der die Modellspezifikation in ein sequentielles Programm überführt, und ein Laufzeitsystem, das für eine geeignete Abwicklung der Modelldynamik sorgt. Die Implementierungssprache war C unter UNIX System V.

Mein Dank gilt Herrn Prof. Dr. B. Schmidt für die Betreuung und Unterstützung dieses Vorhabens.

Herrn Prof. Dr. A. Reinhardt danke ich für die Übernahme des Zweitgutachtens und das große Interesse an dieser Arbeit.

Danken möchte ich insbesondere Herrn Prof. Dr. Schmidt sowie meinen Kollegen K.-J. Langer und K. Dörnhöfer für die hervorragende Zusammenarbeit und die vielen anregenden Diskussionen, die ein Gelingen dieser Arbeit erst ermöglichten.

Wichtige Beiträge lieferten auch Frl. S. Petsch und Herr J. Wittmann, die mit großem Fleiß in ihren Studienarbeiten die Implementierung des Compilers und der Modellerstellungsumgebung voranbrachten.

Eine große Hilfe war mir Frl. S. Wilfer, die mit viel Einsatz und Sorgfalt das Tippen der Texte und die Ausgestaltung mit Bildern übernahm.

# Inhaltsverzeichnis

# 1 Einführung

Es war das Ziel dieser Arbeit, eine formale Sprache zur Beschreibung von dynamischen Modellen zu entwerfen. Dieses Bestreben wurde von der Überzeugung motiviert, daß die bestehenden Simulationssprachen pragmatische Ansätze verkörpern, die sowohl methodologischen Grundlagen als auch einer hinreichend genauen Spezifikation entbehren.

Der Nutzen einer gut lesbaren, verständlichen Modellbeschreibung liegt einerseits darin, daß man daraus ein Simulationsprogramm ableiten kann, andererseits darin, daß es möglich wird, Forschungsergebnisse, die in Form eines Modells vorliegen, an andere Wissenschaftler weiterzugeben. Die formale Modellbeschreibung ist die ideale Dokumentation zu ausgeführten Simulationsläufen und die ideale Grundlage für Diskussionen über den betrachteten Forschungsgegenstand.

Im ersten Abschnitt dieses Kapitels wird ein kurzer Überblick gegeben, in welches Umfeld die vorliegende Arbeit einzuordnen ist. Die folgenden Abschnitte erläutern den bisherigen Stand der Technik in Bezug auf Simulations-Software, und es werden darüber hinausgehende Ziele formuliert.

Das zweite Kapitel stellt die Vorgehensweise bei der Modellbildung genauer dar. Es beschreibt, wie aus Beobachtungen formale Beziehungen entwickelt werden.

Das dritte Kapitel liefert die theoretischen Grundlagen für eine formale Modellspezifikation und deren algorithmische Behandlung.

Im vierten Kapitel wird die Sprachsyntax und -semantik der Modellbeschreibungssprache SIMPLEX-MDL vorgestellt als eine Möglichkeit, um formale Modelle zu spezifizieren.

Das fünfte Kapitel beschreibt die Erzeugung des Simulationsprogramms und stellt seine Beziehung zur Experimentierumgebung dar.

Im sechsten Kapitel wird der Aufbau des generierten Simulationsprogramms eingehend behandelt. Hier werden die Algorithmen besprochen, welche die Beziehungen im Modell auswerten.

Das siebte Kapitel umfaßt eine Sammlung von Modellen, die in SIMPLEX-MDL spezifiziert sind und den Einsatz der Sprache an Anwendungsbeispielen demonstrieren.

## 1.1 Modellbildung und Simulation

Die empirische Wissenschaft hat die Zielsetzung, Beobachtungen an der Realität in allgemein-gültigen Aussagen (Gesetzen) zusammenzufassen. Auf diese Art entsteht ein (formal-) sprachliches Modell eines untersuchten Objekts.

Ein solches Modell läßt Zusammenhänge zwischen den Beobachtungen erkennen, erlaubt Schlußfolgerungen, die wiederum auf die Realität übertragen werden können und dient der Weitergabe erworbenen Wissens. Ein Modell ist daher Ausdruck der Erkenntnisse empirischer Forschung.

Die methodische Vorgehensweise bei der Modellbildung ist stets dieselbe: Bestimmte Beobachtungen an der Realität erscheinen besonders interessant und man versucht, diese Beobachtungen mit anderen in Verbindung zu bringen, d. h. auf andere Beobachtungen zurückzuführen.

Um alle diese Beobachtungen vorzunehmen, werden eine Reihe von Beobachtungseinrichtungen (Meßgeräte) eingesetzt, die es gestatten, nebeneinander Beobachtungen durchzuführen und aufzuzeichnen. Alles übrige, was sich in der Realität sonst noch abspielt, interessiert vorläufig nicht. Durch diese Vorgehensweise wird also ein Ausschnitt aus der Wirklichkeit abgegrenzt. Dieser Ausschnitt enthält das zu untersuchende Objekt und die Beobachter. Man bezeichnet diesen Ausschnitt als Original, originales System oder kurz System.

Unter Variation verschiedener Bedingungen werden nun Experimente durchgeführt. Ein Experiment verschafft über einen bestimmten Zeitraum hinweg Sätze gleichzeitig ausgeführter Beobachtungen. Die gewonnene Information liegt in Form von Daten vor und wird daher als Originaldaten oder Systemdaten bezeichnet.

Auf der Basis der gewonnenen Originaldaten setzt nun die Modellbildung ein. Bei einem Modellentwurf für ein *geplantes* System setzt man statt der Originaldaten fiktive Daten an.

Die Systemanalyse (oder Modellbildung) sucht nach gleichbleibenden (zeitinvarianten) Beziehungen zwischen den gewonnenen Originaldaten. Diese Beziehungen werden sprachlich formuliert und beschreiben daher ein abstraktes Modell des zugrundeliegenden Originals. Bei Kenntnis der Beziehungen lassen sich die interessierenden Beobachtungen aus anderen Beobachtungen ableiten. Gelten die gefundenen Beziehungen auch unter anderen Bedingungen, dann erhalten sie den Charakter von Gesetzmäßigkeiten.

In den sogenannten exakten Wissenschaften werden diese Beziehungen formal-sprachlich, d. h. mathematisch beschrieben. Ein System formaler Zusammenhänge, das die Beziehungen zwischen den Originaldaten wiedergibt, bezeichnet man dann als formales oder mathematisches Modell des Originals.

Die Gültigkeit eines Modells ist im strengen Sinne nur für die zugrundegelegten Originaldaten gewährleistet. Durch gezielte Experimente läßt sich jedoch meist ein Bereich eingrenzen, innerhalb dessen ein Modell Gültigkeit besitzt. Dadurch gewinnt das Modell Allgemeingültigkeit und liefert einen echten Gewinn an Erkenntnis.

Anhand logischer Schlußfolgerungen kann man weitere allgemeine Aussagen über das Modell erlangen. Durch Kombination mehrerer kleiner Modelle lassen sich größere Modelle zu nicht speziell untersuchten Systemen aufbauen.

Die wichtigste Eigenschaft eines Modells ist aber, daß es in der Lage ist, gleiche Ergebnisse zu produzieren wie das reale System, wenn gleiche Anfangsbedingungen vorgegeben werden. Diese Modellexperimente beruhen auf zwei verschiedenen Methoden, um die Beziehungen des Modells auszuwerten.

1. Mit Hilfe logisch-deduktiver (analytischer) Methoden werden aus den gefundenen Zusammenhängen allgemeine Lösungen, d. h. Lösungen, die unter allen Anfangsbedingungen gelten, hergeleitet. Bei Vorgabe einer speziellen Anfangsbedingung liefert die allgemeine Lösung Modellbeobachtungen zu jedem gewünschten Zeitpunkt. Eine derartige analytische Lösung läßt sich jedoch nur in speziellen Fällen finden.

2. Man baut ein *reales Modell* auf, das die gegebenen Beziehungen mit Hilfe algorithmischer Verfahren oder physikalischer Vorgänge auswertet. In Modellexperimenten liefert dieses *reale Modell* aus einer gegebenen Anfangssituation fortlaufend weitere Beobachtungsdaten. Diese nennt man, da sie über das Modell gewonnen wurden, Modelldaten.

In diesem Fall spricht man von Simulation des Originals durch ein (reales) Modell. Das physikalische System, das die Simulation ausführt (hierzu zählt auch eine digitale Rechenanlage), heißt simulierendes System oder Simulator. Das Programm, das einen geeigneten Algorithmus enthält, um aus den gegebenen Beziehungen Modelldaten zu gewinnen, wird als Simulationsprogramm bezeichnet.

Bei korrekter Ausführung der beschriebenen Vorgehensweise müssen Modelldaten und Systemdaten übereinstimmen. Trifft dies für alle ausgeführten Experimente zu, sagt man, das Modell ist validiert.

Ergebnisse, die mit Hilfe eines validierten Modells gewonnen wurden, gelten dann umgekehrt auch für das originale System, das dem Modell zugrundeliegt. In vielen Fällen ist es vorteilhaft, wenn Untersuchungen am Modell statt am realen System ausgeführt werden können.

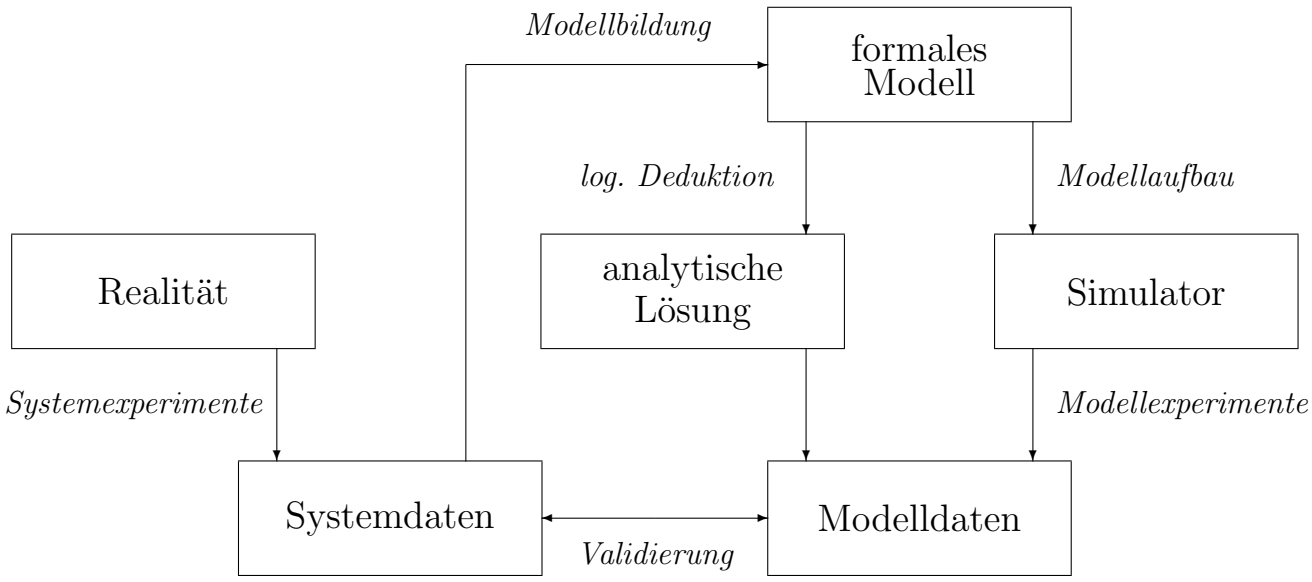



Abb. 1.1-1: Vorgehensweise bei Modelluntersuchungen

Die vorliegende Arbeit unterstützt die beschriebene Vorgehensweise durch

- den Entwurf einer formalen Sprache zur Beschreibung (Spezifikation) dynamischer Modelle, einer sog. Modellbeschreibungssprache
- die automatische Generierung eines Simulationsprogramms (Modellaufbau) nach Vorgabe einer formalen Modellspezifikation.

## 1.2 Realisierung formaler Modelle durch Simulationsprogramme

Um ein vorliegendes formales Modell auf einer digitalen Rechanlage simulieren zu können, benötigt man ein Simulationsprogramm, welches die angegebenen Beziehungen mit geeigneten Algorithmen auswertet. Zur Formulierung eines Simulationsprogramms eignet sich im Prinzip jede prozedurale höhere Programmiersprache. Dennoch erschienen bereits in den frühen sechziger Jahren Simulationssprachen und andere Software-Werkzeuge, die speziell zum Zweck der Simulation ausgerichtet waren.

Aus heutiger Sicht war hierfür in erster Linie ein Grund maßgebend:
In realen Systemen finden gleichzeitig und nebeneinander eine Vielzahl von Vorgängen statt, die sich auf verschiedene Art und Weise (gegenseitig) beeinflussen.

Die Abbildung eines solchen Systems auf ein prozedurales (sequentielles) Rechnerprogramm erfordert, daß der Programmierer sehr viel Aufwand treiben muß, um den Programmfluß in geeigneter Weise zu steuern.

Die wesentlichen Teile eines solchen Simulationsprogramms, welche die Beziehungen des formalen Modells repräsentieren, gehen dabei in einem Wirrwarr von Verzweigungen und Sprunganweisungen völlig unter. Ein solches Programm ist für Außenstehende nicht mehr nachvollziehbar, geschweige denn änderbar, und kann, sobald es einen größeren Umfang annimmt, nicht mehr mit vertretbarem Aufwand getestet werden. Den gewonnenen Ergebnissen kann man dann keine Zuverlässigkeit mehr zuschreiben.

## 1.3 Software-Werkzeuge zum Aufbau von Simulationsprogrammen

Es gibt nun verschiedene Ansätze, mit denen man versucht hat, diesem Problem zu begegnen:

- Höhere Programmiersprachen mit Nebenläufigkeit von Prozessen
- Simulationspakete
- Simulationssprachen
- Datengesteuerte Simulatoren

### Höhere Programmiersprachen mit Nebenläufigkeit

Verschiedene höhere Programmiersprachen wurden um die Fähigkeit erweitert, Programmabschnitte (Prozesse, Tasks) nebenläufig abzuarbeiten. Aber erst neuere Ansätze, welche die Konzepte der objekt-orientierten Programmierung verfolgen, sind in gewisser Weise eine Hilfe beim Schreiben von Simulationsprogrammen. Die wichtigsten Vertreter objekt-orientierter Programmiersprachen sind SIMULA, SMALLTALK, C++.

Zwischen einzelnen Prozessen (Objekten) ist ein Informationsaustausch möglich, der durch einen in der Sprache verankerten Kommunikationsmechanismus (meist durch Übergabe von Botschaften) ausgeführt wird. Durch Einbringen spezieller Anweisungen (signal, wait, etc.) wird geregelt, welcher Prozeß gerade aktiv ist.

Es ist die Bildung von Objekt-Klassen möglich, d.h. von abstrakten Datentypen mit mehrfacher Ausprägung. Jedes Objekt besteht aus seinen eigenen Variablen und einem Satz von Funktionen (Methoden), die nur auf den eigenen Datenbeständen operieren. Ein Objekt kann über Botschaften Methoden eines anderen Objekts anstoßen. Mehrere Objekte einer Klasse unterscheiden sich durch die aktuellen Werte ihrer Daten.

Diese Hilfen sind allerdings nur gut, um eine bestimmte Programmiermethodik zu unterstützen. Beim Aufbau seines Simulationsprogramms wird der Benutzer alleine gelassen. Er benötigt im Gegenteil ganz spezielle Programmierkenntnisse, die man von keinem Anwender erwarten kann. Der komponentenweise Aufbau kontinuierlicher Modelle ist zudem gänzlich unmöglich, da man hierfür andere Kommunikationsmechanismen benötigt.

**Simulationspakete**

Simulationspakete unterstützen den Anwender beim Aufbau eines Simulationsprogramms in einer höheren Programmiersprache. Sie geben einen Programmrahmen für das zu implementierende Modell, eine Ablaufsteuerung und eine Sammlung von Unterprogrammen vor, welche aus Algorithmen zur Auswertung der Modellbeziehungen (Integrationsverfahren etc.) und zur Auswertung und Darstellung der Ergebnisse bestehen.

Die wichtigsten Vertreter für Simulationspakete sind GASP IV, GPSS-FORTRAN Version 3 und BORIS.

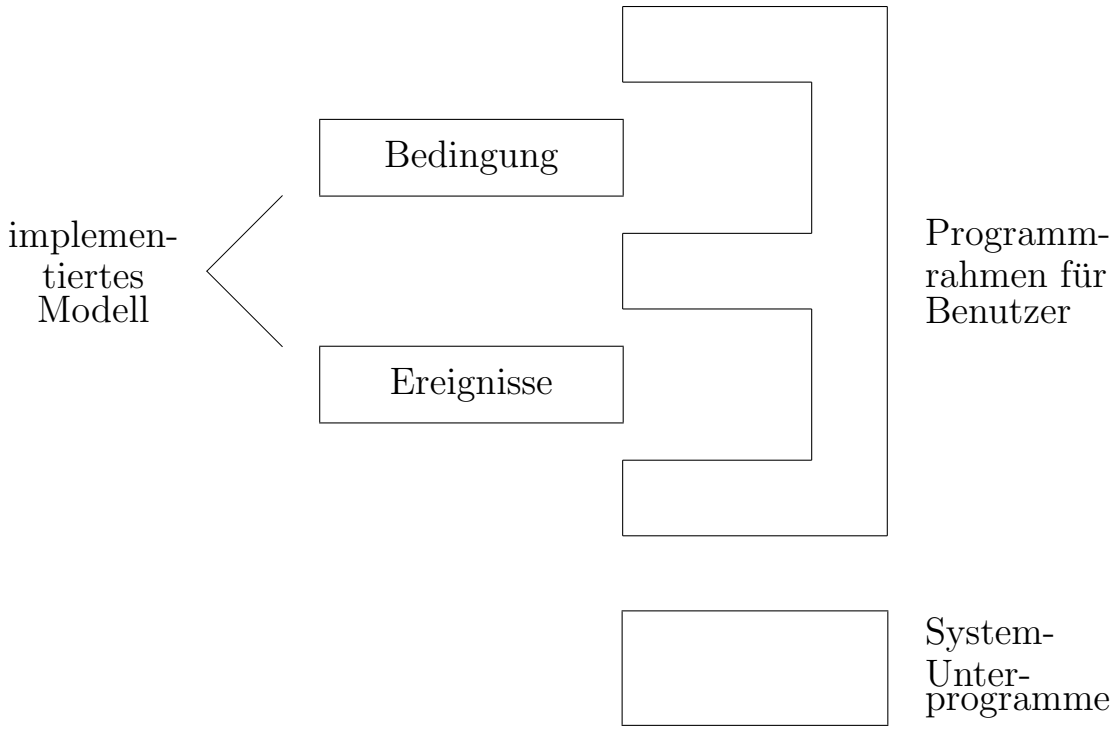



Abb. 1.2-1: Prinzipieller Aufbau eines Simulationspakets

Ein Simulationspaket gestattet dem Benutzer, sich auf die Implementierung seines Modells zu konzentrieren. Er ist dabei aber in seinen Modellvorstellungen nicht mehr völlig frei. Die möglichen Modellobjekte (Modellgrößen, Transactions etc.) sowie die Art der Beziehungen zwischen diesen Modellobjekten (Ereignisse, Differentialgleichungen, Transactionfluß) sind ihm vorgegeben und in den entsprechenden Abschnitt im Programmrahmen einzutragen.

Diese vorgegebene Systematik entbindet den Benutzer davon, eine eigene Programmstruktur zu entwerfen. Die Ablaufsteuerung und die Zeitführung werden vom Simulationspaket übernommen.

Trotz dieser Vorgaben benötigt man noch eine ganze Reihe von Kenntnissen, um ein gegebenes formales Modell zu implementieren. Es ist notwendig, die Programmiersprache zu beherrschen, in der das Simulationspaket geschrieben ist, und man muß wissen, wie die Datenbereiche der Modellobjekte aufgebaut sind. Die Modellbeziehungen müssen ebenfalls in geeigneter Form implementiert werden, und zu ihrer Auswertung bedarf es der Kenntnis zahlreicher Systemunterprogramme und deren Parameterlisten. Spielt die Reihenfolge der Abarbeitung bei zeitgleichen Ereignissen eine Rolle, ist eine sehr genaue Kenntnis der Ablaufsteuerung notwendig.

Diese Kritikpunkte betreffen noch nicht so sehr die erstmalige Erstellung eines Simulationsprogramms, sondern mehr noch seine spätere Verwendung. Da das implementierte Modell ohne genaue Kenntnis des Simulationspakets nicht verständlich ist, kann sich ein Außenstehender schlecht heineindenken, es ändern oder erweitern. Daher finden solche Simulationsprogramme meist nur einmal Verwendung.

Wegen des starren Schemas der algorithmischen Abarbeitung kann die Effizienz eines Simulators, der mit Hilfe eines Simulationspakets aufgebaut wurde, deutlich unter der eines Spezialsimulators liegen.

### Simulationssprachen

Die Simulationssprachen versuchen die Nachteile der Simulationspakete zu vermeiden. Das Modell wird in einer Art Spezifikationssprache abgefaßt und von einem Präprozessor oder Compiler in Programmabschnitte einer (meist höheren) Programmiersprache übersetzt. Als Laufzeitsystem dient ein Simulationspaket, das auf diese Art um das zu implementierende Modell ergänzt wird.

Die Verständlichkeit eines Modells läßt sich dadurch deutlich verbessern. Die Terminologie der Sprache kann sich an einer Begriffswelt orientieren, wie sie bei der Modellbildung üblich ist.

Semantische Überprüfungen der Modellspezifikation können garantieren, daß der erzeugte Code fehlerfrei und korrekt ablauffähig ist.

Die Modellobjekte können durch komplexere Datenstrukturen repräsentiert sein als man es in einem Simulationspaket dem durchschnittlichen Benutzer

zumuten könnte. Erst hierdurch wird eine Modellzerlegung in hierarchisch geschachtelte Komponenten möglich.

Andererseits tun sich bei diesem Konzept neue Probleme auf, da der Benutzer mit dem zur Verfügung gestellten Sprachumfang vorlieb nehmen muß. Es beschränkt den Anwender in der Formulierung seiner Modelle und in den Möglichkeiten sein Modell zu initialisieren, die Simulationsergebnisse fortlaufend auszuwerten, Daten mit der Umgebung auszutauschen oder den Zeitfortschritt mit der Echtzeit zu synchronisieren.

Aus diesem Grund gestatten viele Simulationssprachen Eingriffe in das darunterliegende Simulationspaket. Die erzielten Vorteile gehen dadurch natürlich wieder verloren, und so konnten Simulationssprachen nie eine breite Akzeptanz finden und werden nur in bestimmten Anwendungsgebieten eingesetzt.

Die wichtigsten Vertreter von Simulationssprachen für Warteschlangen-Modelle sind GPSS, SIMSCRIPT II.5, SLAM, SIMAN und RESQ.

Zur Formulierung kontinuierlicher und ereignis-orientierter Modelle haben die Sprachen DYNAMO, CSSL IV und ACSL Bedeutung erlangt. Neuere Entwicklungen auf diesem Gebiet, die auch einen modularen Modellaufbau gestatten, erlangten keine Produktreife. Hierzu zählen GEST, SYSMOD und COSMOS.

### Datengesteuerte Simulatoren

Beschränkt man sich auf ein sehr enges Anwendungsgebiet, kann man häufig Simulationsprogramme so gestalten, daß die Spezifikation des Modells in Form von Tabellen abgelegt werden kann. Ohne daß eine erneute Übersetzung erforderlich wäre, kann ein Modell verändert werden. In Verbindung mit einer komfortablen Benutzeroberfläche gestattet dieser Ansatz ein sehr zügiges Arbeiten und ermöglicht auch dem Nicht-Informatiker, Modelle aufzubauen und in ihrer Struktur zu variieren.

Datengesteuerte Simulatoren wurden in letzter Zeit zu Baustein-Simulatoren weiterentwickelt. Auf dem Gebiet der Regelungstechnik werden SIDAS II und PROSIGN angeboten. In der Fertigungstechnik stehen ISIS, DOSIMIS III und GRAFSIM für den Stand der Technik.

Trotz der hohen Benutzerfreundlichkeit datengesteuerter Simulatoren ist ihre Verbreitung sehr begrenzt. Die Ursache ist vor allem darin zu sehen, daß sie auf ein sehr enges Anwendungsgebiet zugeschnitten sind und keine Sonderwünsche des Benutzers zulassen.

Von den genannten Werkzeugen zur Erstellung von Simulationsprogrammen sehen wir für die Zukunft die besten Chancen im Bereich der Simulationssprachen. Die Nachteile, die Simulationssprachen heute noch besitzen, liegen nicht in der Natur der Sache, sondern darin, daß ihre Entwicklung nicht mit genügender Intensität vorangetrieben wurde.

## 1.4 Kritische Anmerkungen zu bestehenden Simulationssprachen

Dieser Abschnitt soll aufzeigen, welchen technischen Stand Simulationssprachen bislang erreicht haben und auf welchen Bereichen Verbesserungen denkbar und wünschenswert sind.

Wir wollen Simulationssprachen bezüglich der folgenden Gesichtspunkte näher betrachten:

- Sprachumfang
- modularer Modellaufbau
- Ablaufsteuerung:
  - Zeitführung
  - Ereignisausführung
  - Gleichzeitigkeit
- graphische Modelldarstellung
- Zielmaschine
- Anwender-Orientierung

### Sprachumfang

Die bestehenden Simulationssprachen orientierten sich bezüglich ihres *Sprachumfangs* meist an einer einzigen Modellklasse. Eine solche Modellklasse wird begründet durch eine Modellvorstellung, für die eine mathematische Theorie vorliegt. Diese Theorien liefern allgemeine Aussagen über eine solche Modellklasse und in speziellen Fällen analytische Lösungen.

Die wichtigsten Modellklassen sind:

- Systeme von Differentialgleichungen
- Systeme von Differenzengleichungen
- Systeme von diskreten Übergängen (Automaten)
- Systeme von Warteschlangen

Beim Aufbau größerer Modelle zeigt sich jedoch sehr schnell, daß man mit den Möglichkeiten einer einzigen Modellklasse starken Beschränkungen unterliegt. Daher wurden fast alle bestehenden Simulationssprachen um Fähigkeiten anderer Modellklassen erweitert.

Da man aber versäumt hat, diese Kombination wiederum auf eine saubere theoretische Basis zu stellen, benötigt man für eine korrekte Beschreibung des Modellverhaltens Kenntnisse über die interne Ablaufsteuerung. Dies betrifft insbesondere die Abarbeitung gleichzeitiger Ereignisse. Solche Sprachen können daher nicht mehr als echte Spezifikationssprachen verstanden werden.

Der Vollständigkeit halber soll noch erwähnt werden, daß die Forschung eine Vielzahl anderer Modellklassen hervorgebracht hat, um bestimmten Anwendungen gerecht zu werden oder bestimmte Sachverhalte zu ergründen. Zu diesen zählen beispielsweise lineare Netzwerke, Petri-Netze oder Bond-Graphen. Alle diese Modellklassen lassen sich jedoch auf die oben genannten zurückführen. So läßt sich ein lineares Netzwerk stets als System linearer Differentialgleichungen oder ein Petri-Netz stets als endlicher Automat darstellen. Aus diesem Grund wollen wir uns hier nur mit den eingangs erwähnten Modellklassen beschäftigen.

### Modularer Modellaufbau

Modularer Modellaufbau dient dazu, um ein größeres Modell aus bereits existierenden und ausgetesteten Bausteinen aufbauen zu können und es dadurch übersichtlicher und zuverlässiger zu machen. Bei sehr großen Modellen kann auch die Einsparung von Programmcode ein wichtiger Gesichtspunkt werden.

Obwohl ein modularer Modellaufbau für größere Modelle als beinahe zwingende Notwendigkeit erscheint, findet man diese Fähigkeit bei keiner der angebotenen Simulationssprachen. Lediglich datengesteuerte Simulatoren und verschiedene höhere (objekt-orientierte) Programmiersprachen ermöglichen eine modulare Zerlegung des Modells.

Stattdessen findet man bei einigen Simulationssprachen die Möglichkeit vor, parametrisierbare Makros zu definieren. Dies ist jedoch kein Ersatz für Module, die über einen eigenen Datenbereich verfügen, sich auf syntaktische und semantische Korrektheit prüfen und als Objektcode in einer Bibliothek ablegen lassen.

Ein Modell sollte aus einzelnen Komponenten aufgebaut werden können, die miteinander in geeigneter Weise kommunizieren. Jede Komponente besitzt wiederum eigene Modellobjekte und zeigt ein dynamisches Verhalten, das sie als Reaktion auf Einwirkungen von außen und auf momentane innere Zustände liefert.

Man kann mit der Beschreibung einer Komponente auch eine ganze Klasse von Komponenten vereinbaren. Die einzelnen Komponenten unterscheiden sich nur in den aktuellen Ausprägungen ihrer Zustände. Dieser Ansatz wurde erstmals in SIMULA realisiert und später von anderen objekt-orientierten Sprachen übernommen.

Sorgt man dafür, daß auch zusammengesetzte Komponenten miteinander kommunizieren können, dann läßt sich ein Modell auch auf mehreren Hierarchieebenen beschreiben.

Einen theoretischen Unterbau für eine solche Modellzerlegung liefert die Allgemeine Systemtheorie [Pich 75]. Diesen Ansatz verfolgte Ören mit seiner Sprache GEST [Ören 84].

### Ablaufsteuerung

Die Ablaufsteuerung eines Simulators regelt die Reihenfolge, in der die Beziehungen zwischen Modellgrößen ausgewertet werden.

Eine gute Simulationssprache sollte ein Modell so spezifizieren, daß der Benutzer keinerlei Kenntnis über die Implementation der Ablaufsteuerung benötigt. Leider ist das fast nie der Fall. Insbesondere bei gleichzeitig ausführbaren Geschehnissen geht die Modelldynamik nicht mehr aus dem Modelltext alleine hervor.

Für die Fortschaltung der Simulationszeit sowie die Ausführung von Folgeereignissen zum gleichen Zeitpunkt findet man zwei grundsätzlich verschiedene Konzepte vor:

1. *Time (Event) Scheduling:*
   Jedes Ereignis besitzt einen Ereigniskalender, in dem der nächste Ausführungszeitpunkt oder auch mehrere Zeitpunkte eingetragen sind. Die Simulationszeit wird zum nächst kleinsten Ereigniszeitpunkt fortgeschaltet und dann die betreffenden Ereignisse ausgeführt. Die Einträge in den Ereigniskalender nimmt der Anwender vor.

   Dieses Verfahren ist zwar effizient, bürdet aber dem Benutzer auf, Auswirkungen in der Zukunft vorherzusehen. Gegebenenfalls müssen bereits angemeldete Ereignisse umdisponiert werden. Dies ist für den Anwender sehr aufwendig und fehleranfällig.

2. *Activity Scanning*
   Die Simulationszeit wird um eine Zeiteinheit fortgeschaltet und anhand von Auslösebedingungen wird geprüft, ob irgend ein Ereignis auszuführen ist. Ist dies der Fall, wird diese Überprüfung zum gleichen Zeitpunkt nochmals durchgeführt. Die Zeit wird dann weitergeschaltet, wenn keine weiteren Ereignisse auszuführen sind.

   Dieses Verfahren ermöglicht eine Beschreibung des Modells, die sich immer nur auf den gegenwärtigen Zeitpunkt bezieht, und ist daher für den Anwender am angenehmsten. In der Ausführung ist es jedoch sehr ineffizient und wird daher nur selten eingesetzt.

Größere Bedeutung hat auch eine Mischung dieser beiden Konzepte erlangt, die zwischen Zeitereignissen und bedingten Ereignisse unterscheidet. Zeitereignisse werden in einen Ereigniskalender eingetragen und bedingte Ereignisse per Activity Scanning abgearbeitet. In der Ereignisbedingung darf selbstverständlich nicht die Zeit auftreten.

### Graphische Repräsentation

Für viele Anwender ist eine textuelle Repräsentation eines Modells noch zu unanschaulich. Deshalb gab es eine Reihe von Bestrebungen, Modelle graphisch darzustellen (GPSS, SLAM, SIMAN). Für Sprachen ohne Komponentenbildung führt dies jedoch zu Bildern, die Flußdiagrammen ähneln und bei größeren Modellen riesige Ausmaße annehmen. Läßt man Informationen weg, um das Bild zu verkleinern, ist die Beschreibung des Modells nicht mehr vollständig und damit meist nicht mehr brauchbar.

### Zielmaschine

Der Anwenderkreis für Simulationssoftware ist – verglichen mit Anwendungsgebieten wie Textverarbeitung, Buchhaltung, Datenbanken etc. – relativ klein, und daher sind mit solcher Software keine großen Umsätze zu erwarten. Das Team für Entwicklung, Wartung und Installation ist daher möglichst klein zu halten. Dies läßt sich nur durch ausdrückliche Beschränkung auf Standard-Software und - Hardware erreichen.

Fast alle Simulationswerkzeuge basieren daher auf ANSI-FORTRAN 66 bzw. ANSI-FORTRAN 77. Modelle, die in einer Simulationssprache formuliert sind, werden zunächst auf diese Zwischensprache abgebildet. Damit ist sichergestellt, daß die erzeugten Simulatoren auf allen Maschinen ablauffähig sind, die im technisch-wissenschaftlichen Bereich Verwendung finden.

Seit der Normierung des Betriebssystems UNIX und der Programmiersprache C, erscheint es jedoch fragwürdig, an FORTRAN als Basissprache festzuhalten. Auf Rechnern kleiner bis mittlerer Größenordnung erfährt FORTRAN nur noch geringe Beachtung. Die Zuverlässigkeit dieser Sprachimplementationen läßt oft zu wünschen übrig, und die bei größeren Rechenanlagen gewohnte hohe Effizienz bei numerischen Anwendungen wird nicht geboten.

### Anwender-Orientierung

Die Anwender für Simulationssoftware sind in allen wissenschaftlichen Disziplinen zu finden, deren Modelle sich mathematisch-formal beschreiben lassen. Hierzu zählen Ingenieure wie Elektrotechniker, Maschinenbauer, Informatiker und Chemie-Ingenieure, Naturwissenschaftler wie Physiker, Chemiker und Biologen, Gesellschaftswissenschaftler wie Wirtschaftswissenschaftler und Soziologen sowie gelegentlich auch Psychologen.

Bisherige Simulationssprachen entstanden meist aus den Anforderungen eines Anwendungsgebiets und verwenden dessen Terminologie und Notation. Trotzdem kommt ein in Programmiersprachen unerfahrener Benutzer nicht mit der angebotenen Software zurecht. Er braucht stets noch einen Programmierer, der ihm das gewünschte Modell aufbaut, austestet und evtl. sogar noch Experimente daran ausführt.

Besitzt der Anwender die nötigen Programmierkenntnisse und baut sein Modell selbst auf, so ist er dennoch nicht in der Lage, dieses implementierte Modell an Dritte zur Durchsicht oder zur Durchführung weiterer Simulationsstudien weiterzugeben.

Verantwortlich für diesen Zustand sind eine schlechte Lesbarkeit der Modelle, viele implizite Annahmen in der Sprache sowie ein Mangel an semantischen Prüfungen, die auf fehlerhafte Modellierung hinweisen würden.

## 1.5 Zielsetzungen für eine neue Simulationssprache

Es war das Ziel, eine formale Sprache zur Beschreibung dynamischer Modelle zu schaffen, die eine breite Akzeptanz in den verschiedensten wissenschaftlichen Disziplinen finden kann.

Diese Sprache soll sich nicht nur dazu eignen, ein Simulationsprogramm zu generieren, sondern auch den Austausch von erworbenem Wissen zwischen Fachkollegen und Wissenschaftlern anderer Disziplinen ermöglichen. Diese Arbeit leistet damit auch einen Beitrag zur Förderung interdisziplinärer Zusammenarbeit.

Um diese globale Zielsetzung zu erreichen, waren eine Vielzahl einzelner Aspekte zu berücksichtigen. Die wesentlichen sind:

1. *Verständlichkeit*
   Die Sprache muß eine Terminologie verwenden, die in allen Fachwissenschaften Akzeptanz findet. Sie darf keine völlig neuartige Terminologie etablieren wollen.

   Die Sprache muß gut lesbar sein. Es darf nicht notwendig sein, ein Handbuch zu benutzen, um ein Modell zu verstehen.

2. *Modularisierung eines Modells*
   Ein Modell muß in hierarchische Komponenten zerlegbar sein, wobei das Klassenkonzept zum Tragen kommt.

   Die Kommunikationsmechanismen zwischen Komponenten müssen so elementar sein, daß sie die Modellzerlegung nicht behindern und die Modellierung komplexerer Kommunikationsmechanismen gestatten.

3. *Graphische Modelldarstellung*
   Neben der textuellen Darstellung sollte sich die komponentenweise Zerlegung eines Modells durch eine Grafik repräsentieren lassen, die alle Informationen enthält, um das Modell vollständig zu verstehen.

4. *Mächtigkeit der Sprache*
   Die Sprache muß es zulassen, den größten Teil der Modelle aus den verschiedenen Anwendungsbereichen zu beschreiben. Nur so kann sie ihrem Anspruch auf ein universelles Werkzeug gerecht werden.

5. *Eindeutige Beschreibung des Modellverhaltens*
   Alle Informationen über die Reihenfolge des Modellablaufs müssen in der Beschreibung des Modells enthalten sein.

6. *Deklarative Sprache*
   Die Beziehungen zwischen Modellgrößen stellen Definitionen und kein Ablaufgeschehen dar. Dementsprechend darf die Reihenfolge, in der die Beziehungen angegeben werden, keine Rolle spielen.

7. *Kombination verschiedener Modellklassen*
   Um die Mächtigkeit der Sprache zu gewährleisten, muß auf mehrere Modellklassen zurückgegriffen werden. Dazu ist zunächst eine theoretische Vereinheitlichung zu leisten.

8. *Abarbeitung der Modellbeziehungen*
   Die Kombination mehrerer Modellklassen erfordert auch eine sorgfältige Implementierung der Ablaufsteuerung. Insbesondere die Unterbrechung kontinuierlicher Vorgänge durch Unstetigkeitsstellen bedarf besonderer Beachtung.

9. *Trennung von Modell- und Experimentbeschreibung*
   Nach der Erstellung des Simulationsprogramms aus der Modellspezifikation muß der Benutzer noch in der Lage sein, beliebige Experimente mit dem Modell durchzuführen. Beschreibungen von Experimenten dürfen deshalb nicht Bestandteil einer Modellspezifikation sein.

10. *Zielmaschine*
    Als Zielmaschine für Simulationsanwendungen kommen in Zukunft in erster Linie Workstations und schnelle PC's in Betracht. Da diese Rechner fast alle mit UNIX betrieben werden, bietet sich die Programmiersprache C als ideale Implementierungssprache an.

## 1.6 Das SIMPLEX-II-Projekt

Die Realisierung einer Sprache mit den genannten Zielsetzungen ist nicht als losgelöstes Produkt möglich. Ein generiertes Simulationsprogramm benötigt eine entsprechende Umgebung, in der Experimente ausgeführt und Simulationsergebnisse ausgewertet und dargestellt werden können. Hierbei fallen eine Vielzahl von Dateien an, die in geeigneter Weise verwaltet werden müssen.

Deshalb wurde Ende 1984 das Projekt SIMPLEX-II begonnen, mit dem Ziel, ein Simulationssystem zu erstellen, das den Benutzer in allen Belangen bei der Durchführung einer Simulationsstudie unterstützt.

SIMPLEX-II ist ein Softwaresystem mit eigener Benutzeroberfläche auf der Basis von UNIX System V und in der Programmiersprache C implementiert.

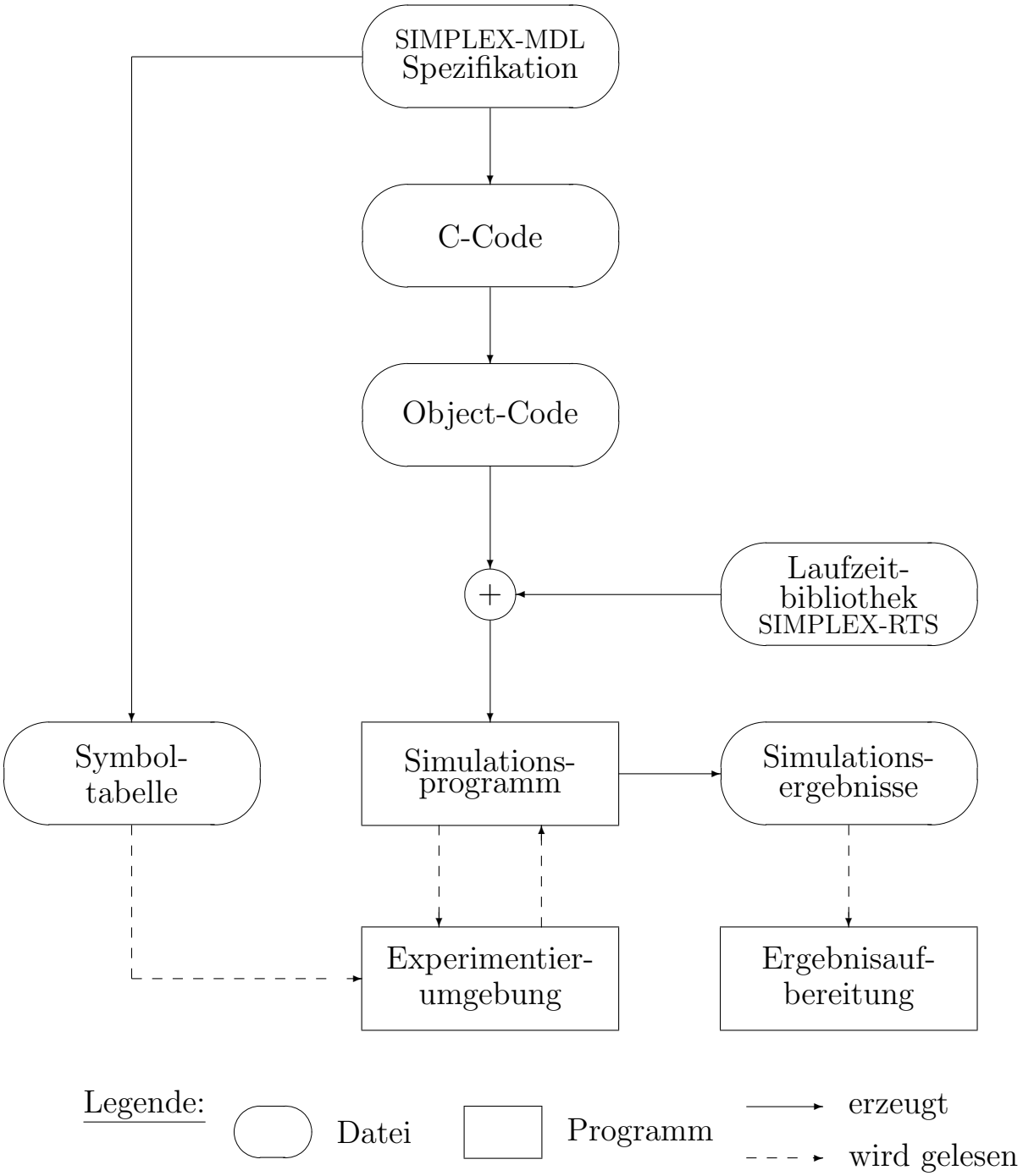



Abb. 1.3: Erstellung des Simulationsprogramms und Anbindung an das Simulationssystem in SIMPLEX-II

Das Gesamtsystem ist in der Dissertation von K.-J. Langer [Lang 90] vollständig dokumentiert. Darstellungen, die einen Überblick vermitteln, finden sich auch in [LaSch 87].

In dieser Arbeit wird das Simulationssystem nur insoweit behandelt, als es die Modellbeschreibung, die Erzeugung des Simulationsprogramms und die Kommunikation mit der Experimentierumgebung betrifft.

Die Modellbeschreibungssprache SIMPLEX-MDL dient der Formulierung von Modellen. Der Benutzer spezifiziert sein Modell in dieser Sprache und überläßt die Generierung des Simulationsprogramms der Modellerstellungsumgebung. Diese überprüft die Modellspezifikation auf syntaktische und semantische Korrektheit, erzeugt aus jeder Modellkomponente prozeduralen C-Code sowie eine Symboltabelle, veranlaßt die Übersetzung in Objektcode und bindet alle Modellobjekte und eine Laufzeitbibliothek zum fertigen Simulationsprogramm zusammen.

Das Simulationsprogramm wird von einer Experimentierumgebung gestartet. Diese besorgt sich die Namen der Modellobjekte und deren Adressen. Während des Experimentierens kann der Benutzer jederzeit lesend wie schreibend auf die Modellobjekte zugreifen. Er kann sich Beobachter einrichten, die alle Veränderungen von Modellgrößen für eine spätere Aufbereitung aufzeichnen.

# 2 Methodologie der Modellbildung

Die angestrebte Modellbeschreibungssprache soll die vollständige formale Spezifikation eines dynamischen Modells erlauben. Um festzustellen, welche sprachlichen Mittel hierzu benötigt werden, wird in diesem Kapitel die Vorgehensweise bei der Modellbildung näher beleuchtet.

## 2.1 Beobachtungen an der Wirklichkeit

Unter einer Beobachtung wollen wir jede Art der sinnlichen Wahrnehmung verstehen. Unser gesamtes Wissen über ein bestimmtes reales System beruht auf Beobachtungen an diesem System.

Indem wir uns in der Zahl der Beobachtungen beschränken, grenzen wir das interessierende System von seiner Umgebung ab. Wir betrachten daher lediglich einen Ausschnitt der uns umgebenden Wirklichkeit.

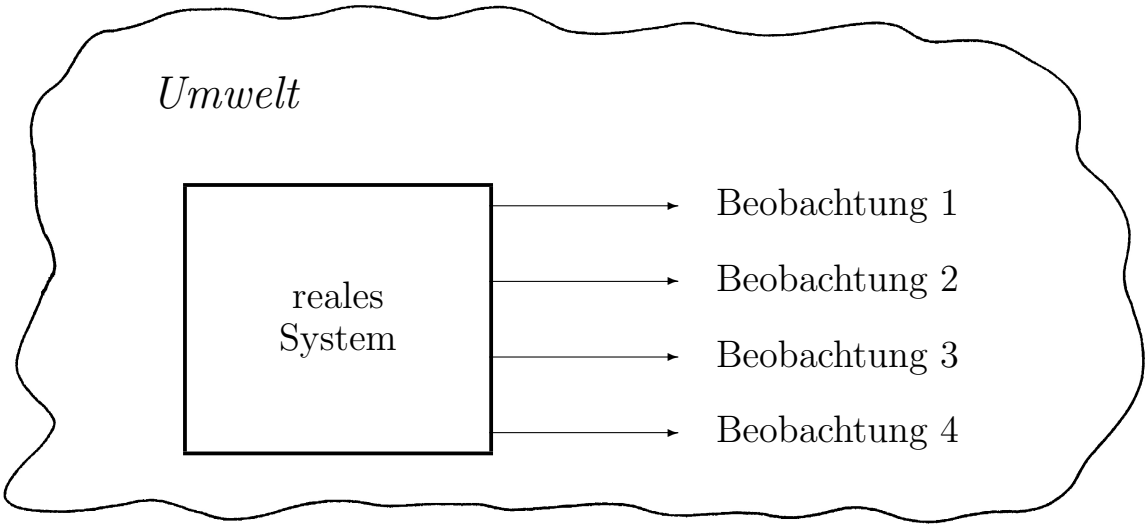



Abb. 2.1-1: Beobachtungen am realen System

Die in Abb. 2.1-1 gezeigte Darstellung bedarf jedoch der Verfeinerung, falls der Beobachter seinerseits Wirkungen auf das reale System ausübt. Das reale System ist in solchen Fällen um die Beobachter zu erweitern.

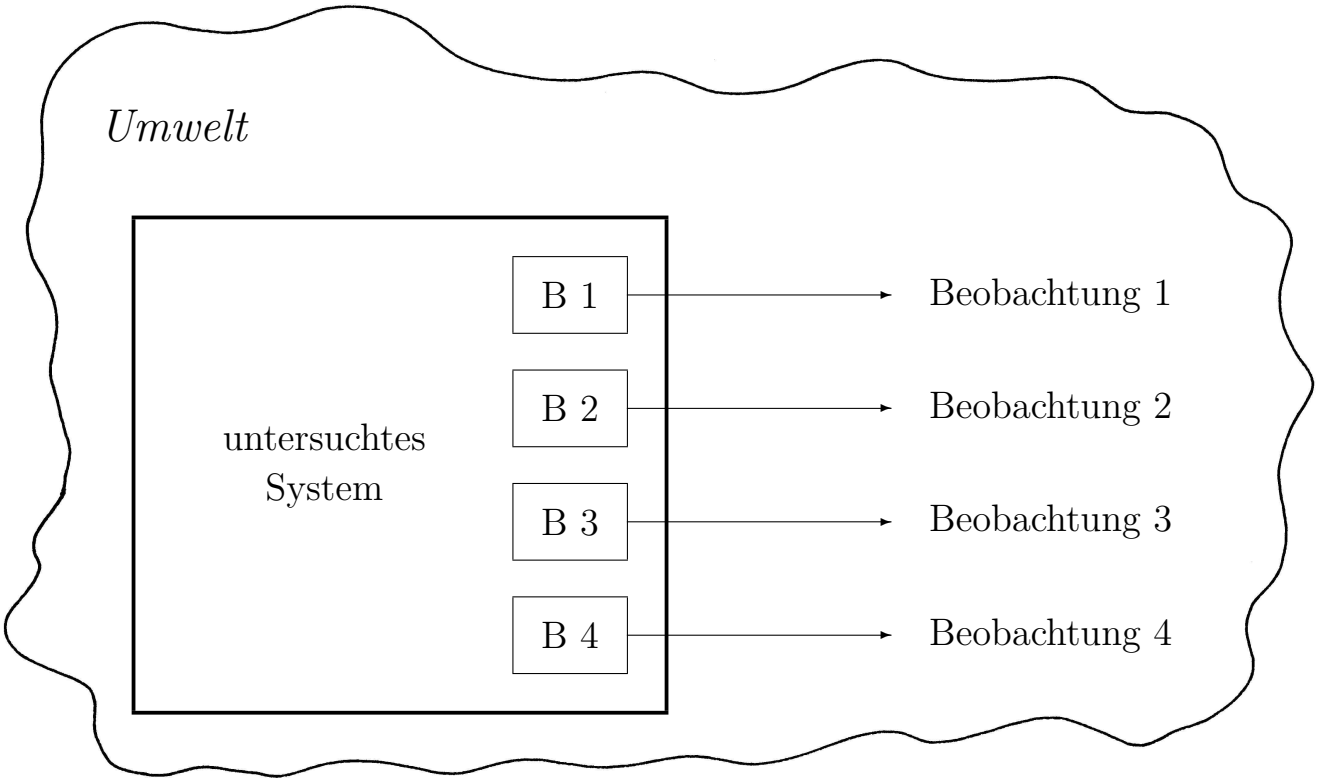



Abb. 2.1-2: Das reale System mit integrierten Beobachtern

Die integrierten Beobachter üben eine zunächst unbekannte Wirkung auf das untersuchte System aus. Unser Wirklichkeitsausschnitt umfaßt jetzt das interessierende System zuzüglich der Beobachter.

Beobachtungen sind Aussagen über den Ort der Beobachtung im realen System. Der Beobachtungsort kann als Punkt, als Fläche oder als Raum verstanden werden.

Wollen wir diese Aussagen formalisieren, müssen wir herausfinden, welche Gestalt diese Aussagen typischerweiser annehmen. Es sind vor allem zwei Arten von Beobachtungen, die für die Systemanalyse von Bedeutung sind:

1. Beobachten einer Eigenschaft
2. Beobachten einer Inhaltsgröße
   (a) mit gleichen Bestandteilen
   (b) mit individuellen Bestandteilen

Zur Bestimmung einer Eigenschaft verwendet man ein Normal (Maßskala), das die Menge aller möglichen Ausprägungen dieser Eigenschaft enthält. Durch Vergleich wird diejenige Ausprägung (Wert) des Normals ausgewählt (gemessen), welche der Beobachtung am ähnlichsten ist.

*Beispiel 1:* Durch Vergleich der Farbe eines Gegenstandes mit einer Farbskala wird diejenige Farbe ausgesucht, die der Farbe des Gegenstandes am nächsten kommt.

*Beispiel 2:* Durch Anlegen eines Maßbands an einem Baumstamm, wird dessen Umfang gemessen.

Eine Inhaltsgröße gibt an, wieviele Bestandteile eines Typs am Beobachtungsort anzutreffen sind. Vermindert sich die Inhaltsgröße an einem Beobachtungsort, geht dieser Bestand auf eine Inhaltsgröße an einem anderen Beobachtungsort über.

*Beispiel:* Die Masse eines Gegenstands läßt sich auf einer Balkenwaage bestimmen, indem genormte Gewichte solange auf die andere Waagschale gelegt werden, bis die Waage ausbalanciert ist.
Wird ein Teil des Gegenstands abgetrennt und auf eine andere Balkenwaage gebracht, lassen sich beide Waagen wieder ausbalancieren, wenn man einen bestimmten Teil der Gewichte von einer Waage auf die andere bringt.

Beispiele für Inhaltsgrößen sind in der Physik die Energie, in der Hydraulik Flüssigkeitsvolumen, in den Wirtschaftswissenschaften Kapitalbestände, in den Sozialwissenschaften Bevölkerungszahlen.

Eine Unterscheidung zwischen Inhaltsgrößen *(Levels)* und Eigenschaften *(Rates)* zum Zweck einer anschaulicheren Modellbildung wurde erstmals in dem von J. W. Forrester entwickelten Konzept des *System Dynamics* [Forr 73] vorgenommen.

Eine verfeinerte Form der Beobachtung eines Inhalts ergibt sich, wenn man sich nicht nur für die Anzahl von Objekten an einem Beobachtungsort interessiert, sondern feststellt, welche Objekte mit welchen Eigenschaften an einem Beobachtungsort anzutreffen sind. Solche Beobachtungen werden beispielsweise an Systemen vorgenommen, in denen individuelle Aufträge (bzw. Kunden) in Warteschlangen und Bedienstationen verweilen.

*Beispiel:* In einer Warteschlange vor einem Bankschalter warten 3 Kunden mit unterschiedlichen Aufträgen. In diesem Fall reicht nicht mehr die Feststellung, daß 3 Kunden warten, sondern es müssen für jeden Kunden individuell die Wünsche notiert werden.

Eine Inhaltsgröße, die nur die Anzahl an Bestandteilen angibt (Masse, Energie, ...), kann auch als Eigenschaft des Beobachtungsortes verstanden werden. Eine Unterscheidung zum Zweck der Modellbildung ist daher nicht unbedingt erforderlich.

Es ist deshalb möglich, daß wir uns in dieser Arbeit auf die Beschreibung von Modellen beschränken, die auf der Beobachtung von Eigenschaften beruhen, ohne die Anwendungsbandbreite zu sehr einzuschränken.

Inhaltsgrößen allgemeiner Art bleiben einer späteren Arbeit vorbehalten. Insbesondere Warteschlangensysteme, die auf der Verfolgung individueller Bestandteile beruhen, sind daher von den folgenden Betrachtungen ausgenommen.

Für unsere weiteren Untersuchungen liegt uns ein reales System mit integrierten Beobachtern vor. Jeder Beobachter liefert zu jedem Beobachtungszeitpunkt eine Aussage über die gegenwärtige Ausprägung der beobachteten Eigenschaft.

Die beobachteten Eigenschaften eines Systems (Systemgrößen) fassen wir im Beobachtungsvektor

$$\mathbf{V}(t) = \left[ \; v_1(t) \;\; v_2(t) \;\; \ldots \;\; v_{N_V}(t) \; \right]$$

zusammen, der eine Funktion der Zeit ist.

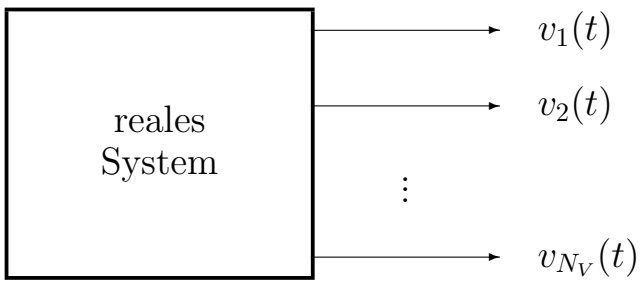



Abb. 2.1-3: Beobachtungen am realen System

Zu jedem Zeitpunkt nimmt eine Systemgröße einen Wert ihrer Ausprägungsmenge (Wertemenge) an.

Die Ausführung von Beobachtungen über einen bestimmten Zeitraum bezeichnet man als Experiment. Es liefert eine zeitlich geordnete Folge von N Beobachtungsvektoren.

$$\mathbf{V}(t_0), \mathbf{V}(t_1), \ldots, \mathbf{V}(t_N)$$

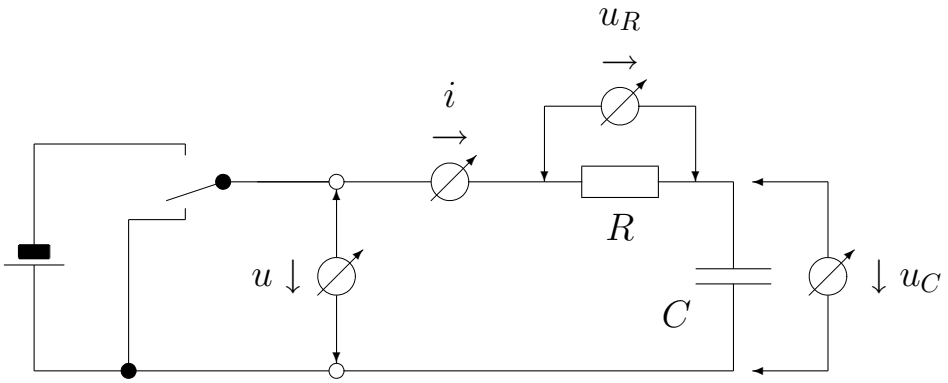
Experimente dienen immer dem Ziel, solche Daten zu beschaffen, die Beziehungen zwischen den Beobachtungen erkennen lassen. Daher werden Experimente selten willkürlich ausgeführt. Meist ist man am Zustandekommen bestimmter Beobachtungen besonders interessiert und versucht diese auf andere Beobachtungen am realen System zurückzuführen.

Die Beobachtungen am realen System werden daher auf diejenigen Größen reduziert, von denen man einen Einfluß auf die interessierenden Größen vermutet. Durch die starke Einschränkung der Zahl der Beobachtungen wird deutlich, daß ein aus solchen Experimenten abgeleitetes Modell nur noch wenige Aspekte des ursprünglichen realen Systems wiedergibt.

## 2.2 Beziehungen zwischen Beobachtungen

Die eigentliche Aufgabe der Modellbildung besteht darin, Beziehungen zwischen beobachteten Größen zu beschreiben. Diese Beziehungen lassen sich umgekehrt wiederum dazu nutzen, die beobachteten Daten unter variierten Bedingungen zu erzeugen.

Diese Vorgehensweise soll durch die Systemanalyse eines RC-Glieds erläutert werden. Der Experimentaufbau ist durch folgendes Schaltbild gegeben:




$$\mathbf{V}(t) = \begin{bmatrix} u(t) \\ u_R(t) \\ u_C(t) \\ i(t) \end{bmatrix}$$

Eine Batterie sei über einen Schalter an ein RC-Glied angeschlossen, das mit Meßinstrumenten versehen ist. Wir nehmen an, daß diese keine Rückwirkung auf das System haben. Es werden die Spannungen am RC-Glied $u(t)$, die Spannung am Widerstand $u_R(t)$ und am Kondensator $u_C(t)$ sowie der Strom durch das RC-Glied $i(t)$ beobachtet.

Mit den Kenntnissen der heutigen Physik lassen sich die Beziehungen zwischen den Systemgrößen selbstverständlich auch theoretisch ableiten. Wir wollen daher annehmen, daß uns weder die Kirchhoffschen Gesetze noch das Verhalten der beiden Bauteile Widerstand und Kondensator bekannt sind .

Wir führen nun ein Experiment aus: In der Anfangssituation zeigen alle Instrumente den Wert Null an. Zum Zeitpunkt $t = 0$ wird der Schalter umgelegt, und die Meßgeräte liefern uns eine Folge von Beobachtungsvektoren

$$\mathbf{V}(t_0), \mathbf{V}(t_1), \mathbf{V}(t_2), \cdots \quad ,$$

wie sie in der nachstehenden Tabelle abgedruckt sind:

| $t_k[sec]$ | $u[V]$ | $u_R[V]$ | $u_C[V]$ | $i[mA]$ |
|---|---|---|---|---|
| 0,00 | 9,40 | 9,40 | 0,00 | 20,00 |
| 0,025 | 9,43 | 8,94 | 0,49 | 19,00 |
| 0,05 | 9,46 | 8,51 | 0,95 | 18,10 |
| 0,075 | 9,48 | 8,09 | 1,39 | 17,20 |
| 0,1 | 9,51 | 7,70 | 1,81 | 16,40 |
| | | | | |
| 0,2 | 9,60 | 6,30 | 3,30 | 13,40 |
| 0,3 | 9,67 | 5,16 | 4,51 | 11,00 |
| 0,4 | 9,73 | 4,22 | 5,51 | 9,00 |
| 0,5 | 9,78 | 3,46 | 6,32 | 7,36 |
| 0,6 | 9,82 | 2,83 | 6,99 | 6,02 |
| 0,7 | 9,85 | 2,32 | 7,53 | 4,93 |
| 0,8 | 9,88 | 1,90 | 7,98 | 4,04 |
| 0,9 | 9,90 | 1,55 | 8,35 | 3,31 |
| 1,0 | 9,92 | 1,27 | 8,65 | 2,71 |

Im Regelfall sind eine Vielzahl von Experimenten auszuführen, um ein gegebenes System vollständig zu analysieren. Nur bei sehr einfachen Systemen, die ein lineares Verhalten zeigen, und bei einer hohen Meßgenauigkeit ist ein einziges Experiment ausreichend.

Die Auswertung des Zahlenmaterials in der Horizontalen liefert uns zunächst zwei *statische Beziehungen*, d. h. Beziehungen, die für jeden Beobachtungsvektor zu beliebigen Zeitpunkten $t_k$ gelten.

$$\left.\begin{array}{rcl} u_R(t_k) & = & R \cdot i(t_k) \\ u(t_k) & = & u_R(t_k) + u_C(t_k) \end{array}\right\} \text{für alle } t_k$$

Eine dritte Beziehung erhalten wir, indem wir je zwei zeitlich benachbarte Beobachtungen miteinander in Beziehung bringen. Diese Beziehung wollen wir als *dynamische Beziehung* bezeichnen.

$$u_C(t_k + \Delta t) = u_C(t_k) + \frac{1}{C} \cdot i(t_k)\Delta t$$

Die Konstante $C$ variiert mit dem zeitlichen Abstand $\Delta t$. Für hinreichend kleines $\Delta t$ strebt $C$ gegen einen Grenzwert.

Wir haben nun unser Modell durch die Angabe von drei Gleichungen beschrieben. Da wir jedoch vier Größen beobachtet haben, ist das Modellverhalten nocht nicht vollständig formuliert. Die vierte Größe läßt sich allerdings nicht als Funktion der anderen Größen angeben. Durch äußere Einwirkung wird ihr Zeitverlauf dem System aufgezwungen. Sie ist daher als explizite Funktion der Zeit anzugeben, z.B.

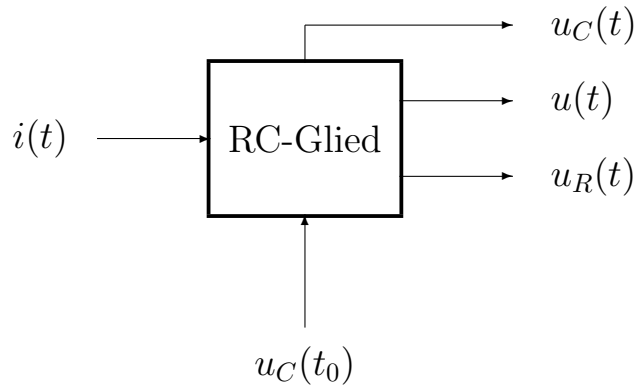
$$u(t) = \begin{cases} 10\ V & \text{für } t \geq 0 \\ 0\ V & \text{sonst} \end{cases}$$

Zur Durchführung von Modellexperimenten müssen wir festlegen, welche Größen uns interessieren und welche Größen vorgegeben werden. Wir erhalten dadurch unterschiedliche Darstellungen des gleichen Modells. Es ist zu entscheiden, welche Größe als Eingangsgröße, Zustandsgröße oder Ausgangsgröße dienen soll.

Als *Eingangsgröße* wird diejenige Systemgröße bestimmt, deren Zeitverlauf explizit vorgegeben wird. *Zustandsgrößen* sind diejenigen Größen, für welche dynamische Beziehungen angegeben werden. Da ihr Wert zum Zeipunkt $t_0$ bekannt sein muß, eignen sich vor allem Größen, deren Wert sich bereits vorher bestimmen läßt und sich zum Anfangszeitpunkt nicht verändert. Die übrigen Größen werden durch statische Beziehungen festgelegt und heißen *Ausgangsgrößen.*

Nehmen wir in unserem Beispiel den Strom $i(t)$ als Eingangsgröße, die Kondensatorspannung $u_C(t)$ als Zustandsgröße und die Größen $u(t)$ und $u_R(t)$ als Ausgangsgrößen an, dann erhalten wir eine spezielle Form, die Modellbeziehungen darzustellen.

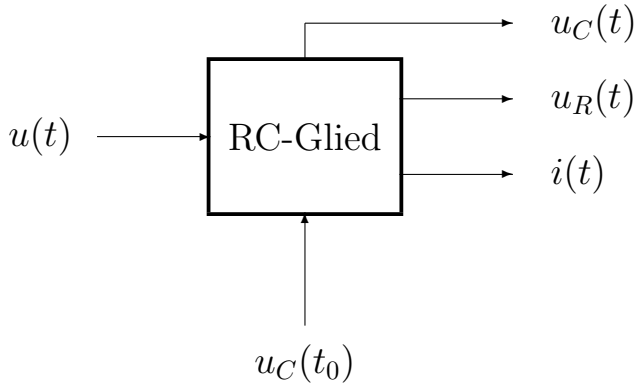
$$\begin{aligned} u_R(t_k) &:= R \cdot i(t_k) \\ u(t_k) &:= u_R(t_k) + u_C(t_k) \\ u_C(t_k + \Delta t) &:= u_C(t_k) + \frac{1}{C} \cdot i(t_k)\Delta t \end{aligned}$$




Wählen wir $u(t)$ als Eingangsgröße und $i(t)$ als Ausgangsgröße, wird das Modell durch die Beziehungen

$$\begin{aligned} i(t_k) &:= \frac{1}{R} \cdot u_R(t_k) \\ u_R(t_k) &:= u(t_k) - u_C(t_k) \\ u_C(t_k + \Delta t) &:= u_C(t_k) + \frac{1}{C} \cdot i(t_k)\Delta t \end{aligned}$$

dargestellt.




Aus dem Zeitverlauf der Eingangsgröße $u(t)$ und dem Anfangswert $u_C(t_0)$ lassen sich nun die Zeitverläufe aller beobachteten Größen erzeugen.

Die Zustandsgleichung läßt sich unter Verwendung der statischen Beziehungen stets so umformen, daß die Zustandsgröße nicht mehr von Ausgangsgrößen abhängig ist.

$$u_C(t_k + \Delta t) := u_C(t_k) + \frac{1}{RC}\left(u(t_k) - u_C(t_k)\right)\Delta t$$

Weitere Darstellungsmöglichkeiten des Modells ergeben sich, wenn man die Zustandsgrößen substituiert.

Wir wollen hierzu als weitere Systemgröße die Ladung des Kondensators $q_C$ einführen. Es gilt der Zusammenhang

$$q_C(t_k) = C \cdot u_C(t_k)$$

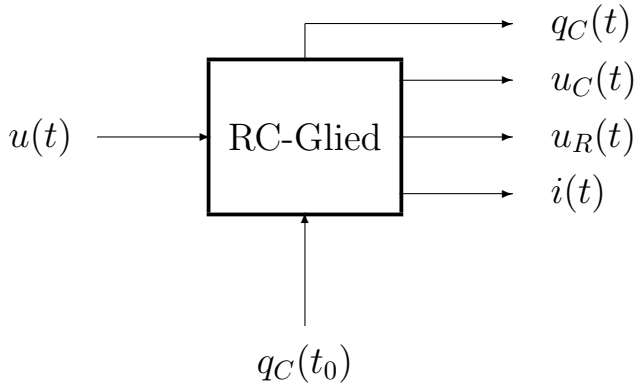
und damit

$$\begin{aligned} q_C(t_k + \Delta t) &= q_C(t_k) + i(t_k)\Delta t \\ &= q_C(t_k) + \frac{1}{R}\left(u(t_k) - \frac{1}{C} \cdot q_C(t_k)\right)\Delta t \end{aligned}$$

Die Spannung am Kondensator $u_C$ läßt sich nun als Ausgangsgröße angeben:

$$u_C(t_k) = \frac{1}{C} \cdot q_C(t_k)$$

Der Anfangszustand wird jetzt selbstverständlich über die Ladung angegeben.




Eine solche Zustandstransformation kann durchaus sinnvoll sein und zu einfacheren Modellbeziehungen führen.

Findet man bei der Systemanalyse mehrere Zustandsgrößen, dann muß man prüfen, ob zwischen ihnen nicht statische Abhängigkeiten bestehen. Ist dies der Fall, dann sollte man eine geeignete minimale Zahl von Zustandsgrößen auswählen, für die man die dynamischen Beziehungen angibt.

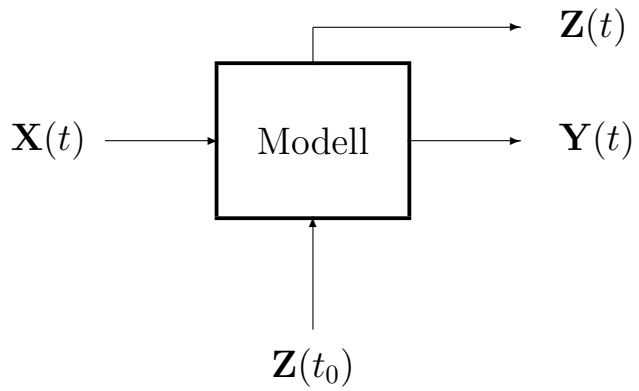
## 2.3 Vollständige Beschreibung des Modellverhaltens

Die im letzten Abschnitt anhand eines Beispiels erläuterte Vorgehensweise wird abschließend zusammengefaßt und formalisiert.

Es ist unser Anliegen, das Modell so zu beschreiben, daß sich aus den formalen Beziehungen Modelldaten erzeugen lassen. Zu diesem Zweck unterteilen wir den Beobachtungsvektor $\mathbf{V}$ in einen Eingangsvektor $\mathbf{X}$, einen Ausgangsvektor $\mathbf{Y}$ und einen Zustandsvektor $\mathbf{Z}$.

$$\mathbf{V} = \begin{bmatrix} \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \end{bmatrix}$$

Das zeitliche Verhalten des Ausgangsvektors $\mathbf{Y}(t)$ und des Zustandsvektors $\mathbf{Z}(t)$ läßt sich bei einer vollständigen Beschreibung des Modells aus dem zeitlichen Verhalten des Eingangsvektors $\mathbf{X}(t)$ und aus dem Anfangszustand $\mathbf{Z}(t_0)$ ableiten.



Hierzu benötigt man 3 Arten von Beziehungen:

1. Verhalten der Eingangsgrößen $\mathbf{X}(t)$
2. Verhalten der Zustandsgrößen $\mathbf{Z}(t)$
3. Verhalten der Ausgangsgrößen $\mathbf{Y}(t)$

Das zeitliche Verhalten der Eingangsgrößen wird nicht aus anderen Beobachtungen abgeleitet, sondern als vorgegeben angenommen. Eine Eingangsgröße ist daher eine reine Funktion der Zeit.

$$x_i(t_k) := e_i(t_k) \tag{2.3.0-1}$$

Das zeitliche Verhalten der Zustandsgrößen leitet sich aus Beziehungen ab, die zwischen aufeinanderfolgenden Beobachtungen bestehen.

$$\begin{aligned} z_i(t_{k+1}) &:= f_i(\mathbf{X}(t_k), \mathbf{Y}(t_k), \mathbf{Z}(t_k), t_k) && (2.3.0\text{-}2) \\ &:= f_i(\mathbf{X}(t_k), \mathbf{Z}(t_k), t_k) && (2.3.0\text{-}3) \end{aligned}$$

Das zeitliche Verhalten der Ausgangsgrößen erhält man durch Auswertung der Beziehungen, die auf Beobachtungen eines Zeitpunkts beruhen.

$$\begin{aligned} y_i(t_k) &:= g_i(\mathbf{X}(t_k), \mathbf{Y}(t_k), \mathbf{Z}(t_k), t_k) && (2.3.0\text{-}4) \\ &:= g_i(\mathbf{X}(t_k), \mathbf{Z}(t_k), t_k) && (2.3.0\text{-}5) \end{aligned}$$

Beziehungen, in denen die Funktionen $f_i$ bzw. $g_i$ auch auf $\mathbf{Y}(t_k)$ beruhen, lassen sich stets so umformen, daß diese Abhängigkeiten verschwinden. Jedoch werden die Beziehungen dadurch komplexer und unübersichtlicher, und deshalb macht man in der Praxis hiervon ungern Gebrauch.

# 3 Theoretische Grundlagen

Dieses Kapitel liefert zunächst eine mathematische Spezifikation für eine Modellbeschreibung auf der Grundlage der Systemtheorie. Die systemtheoretische Betrachtungsweise wird hierzu modifiziert, an die Bedürfnisse einer Modellbeschreibungssprache angepaßt und für kombinierte Modelle verallgemeinert.

Die Bedeutung dieser Spezifikation für diskrete Modelle wird anhand mehrerer Fallstudien verdeutlicht. Nach einigen Betrachtungen zur Darstellung funktionaler Zusammenhänge wird ein Algorithmus abgeleitet, welcher die angegebene Spezifikation erfüllt und insbesondere auch eine komponentenweise Modellzerlegung erlaubt. Der abschließende Abschnitt behandelt die numerischen Aspekte dieses Algorithmus.

Die formalen Darstellungen weisen eine einheitliche Notation auf:
Variablen sind durch kleine Buchstaben $(x, y, z)$ dargestellt, Vektoren durch große Buchstaben in Fettdruck $(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$, Mengen durch kalligraphische Schriftzeichen $(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$. Die Komponenten eines Vektors $(\mathbf{X})$ sind von 1 bis $N_X$ indiziert, ebenso die Wertemengen der Komponenten. Es gilt demnach:

$$\mathbf{X} = [x_1 x_2 \ldots x_{N_X}] \qquad \text{mit} \quad \mathbf{X} \in \mathcal{X} \quad \text{bzw.} \quad x_i \in \mathcal{X}_i$$

$$\text{und} \quad \mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_{N_X}$$

Ebenso werden Funktionen durch kleine Buchstaben $(f, g)$ dargestellt sowie Vektoren von Funktionen durch große Buchstaben in Fettdruck $(\mathbf{F}, \mathbf{G})$.

## 3.1 Die Modelldarstellung der Systemtheorie

Im letzten Kapitel wurde ausgeführt, daß wir die Beobachtungen am realen System auf meßbare Größen beschränken wollen, d. h. auf Beobachtungen, die ein Attribut durch Vergleich mit einem Normal bestimmen. Weiterhin wurde gezeigt, welche Gestalt ein formales Modell typischerweise annimmt, wenn man die zulässigen Beobachtungen in der genannten Art einschränkt.

Eine Theorie, die genau auf diesen Modellvorstellungen beruht, ist die mathematische Systemtheorie. Das Bestreben, Modellgleichungen mit physikalischen Komponenten auszuwerten, hat in der Nachrichtentechnik und Analogrechentechnik zur Begründung dieser Theorie geführt. Ihre hauptsächliche Bedeutung hat sie dadurch erlangt, daß sie bei Beschränkung auf lineare Systeme eine Vielzahl von Theoremen liefert, die man zur Herleitung analytischer Ergebnisse heranziehen kann [Unbe 80, Schü 84].

Da wir für die Simulation Systeme allgemeiner Art zulassen möchten, machen wir uns zum Entwurf einer Modellbeschreibungssprache lediglich die zugrundeliegende Betrachtungsweise zu eigen. Es ist dies die Beschreibung von Modellen mit Hilfe der sogenannten Zustandsdarstellung und die daraus resultierende Möglichkeit, Modelle modular aus einzelnen Komponenten aufzubauen. Auf diese beiden Aspekte wollen wir uns in dieser Arbeit beschränken. Wir werden sie so weiterentwickeln, daß sie als Grundlage einer leistungsfähigen Modellbeschreibungssprache dienen können. Der an einer einführenden Darstellung in die Systemtheorie interessierte Leser sei auf die Literatur verwiesen [Pich 75, Wun 86, Göld 87].

### 3.1.1 Zustandsdarstellung eines Systems

In der Systemtheorie versteht man unter einem *System* ein System mathematischer Beziehungen, die das dynamische Verhalten eines zugeordneten realen Systems wiedergeben. Ein System stellt in diesem Sinne ein formales Modell eines realen Systems dar.

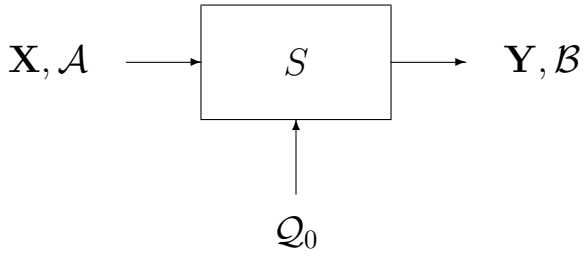
Die Zustandsdarstellung eines Systems $S$ ist ein 11-Tupel $(\mathcal{A}, \mathcal{B}, \mathbf{X}, \mathbf{Y}, \mathcal{Q}_0, \mathcal{Q}, \mathbf{Z}, \mathbf{F}, \mathbf{G}, \mathcal{T}, t_0)$ mit

- der Eingangszeitfunktion $\mathbf{X}(t)$, der Ausgangszeitfunktion $\mathbf{Y}(t)$, der Zustandszeitfunktion $\mathbf{Z}(t)$ und den dazugehörigen Wertemengen $\mathcal{A}, \mathcal{B}$ bzw. $\mathcal{Q}$
- der Zeitmenge $\mathcal{T} = \{t_0, t_1, \ldots\}$, deren Elemente in aufsteigender Ordnung vorliegen
- dem Anfangszustand $\mathcal{Q}_0$ zur Zeit $t_0$, d. h. $\mathbf{Z}(t_0) \equiv \mathcal{Q}_0$
- der lokalen Überführungsfunktion $\mathbf{F}$, wobei gilt

  $$\mathbf{Z}(t_{k+1}) := \mathbf{F}(\mathbf{X}(t_k), \mathbf{Z}(t_k), t_k)$$

- der Ausgabefunktion $\mathbf{G}$, wobei gilt

  $$\mathbf{Y}(t_k) := \mathbf{G}(\mathbf{X}(t_k), \mathbf{Z}(t_k), t_k)$$




Die *lokale Überführungsfunktion* beschreibt, wie der Zustand des Systems in den Zustand zum unmittelbar folgenden Zeitpunkt übergeht.

Beginnend beim Anfangszustand $\mathcal{Q}_0$ zum Zeitpunkt $t_0$ ergibt sich bei gegebener Eingangszeitfunktion $\mathbf{X}(t)$ durch sukzessives Ausführen der lokalen Überführungsfunktion der Verlauf der *Zustandszeitfunktion* $\mathbf{Z}(t)$ für alle Zeitpunkte der Zeitmenge $\mathcal{T}$. Diese resultierende Zeitfunktion heißt auch globale Überführungsfunktion. In einfach gearteten Fällen kann sie deduktiv aus der lokalen Überführungsfunktion hergeleitet und in geschlossener Form angegeben werden.

Die *Ausgangszeitfunktion* $\mathbf{Y}(t)$ resultiert zu jedem Zeitpunkt unmittelbar aus der Eingangszeitfunktion und der Zustandszeitfunktion.

### 3.1.2 Zerlegen und Zusammenfügen von Systemen

Das Verhalten eines Systems wird neben dem Anfangszustand insbesondere durch die Eingangszeitfunktion bestimmt. Als Eingangszeitfunktion eines Systems kann man wiederum eine Ausgangszeitfunktion eines anderen Systems heranziehen.

Durch diese Betrachtungsweise kann man größere Systeme durch Zusammenfügen von Teilsystemen aufbauen (Komposition) oder größere Systeme in Teilsysteme zerlegen (Dekomposition). Größen und Funktionen, die sich auf ein Teilsystem beziehen, werden durch einen hochgestellten Index gekennzeichnet.

Als Eingangsgröße $x_j^{(S)}$ eines Systems $S$ kann jede Ausganganngsgröße $y_i^{(S')}$ eines anderen (oder auch des gleichen) Systems $S'$ dienen, wenn beide in ihrer Wertemenge übereinstimmen.

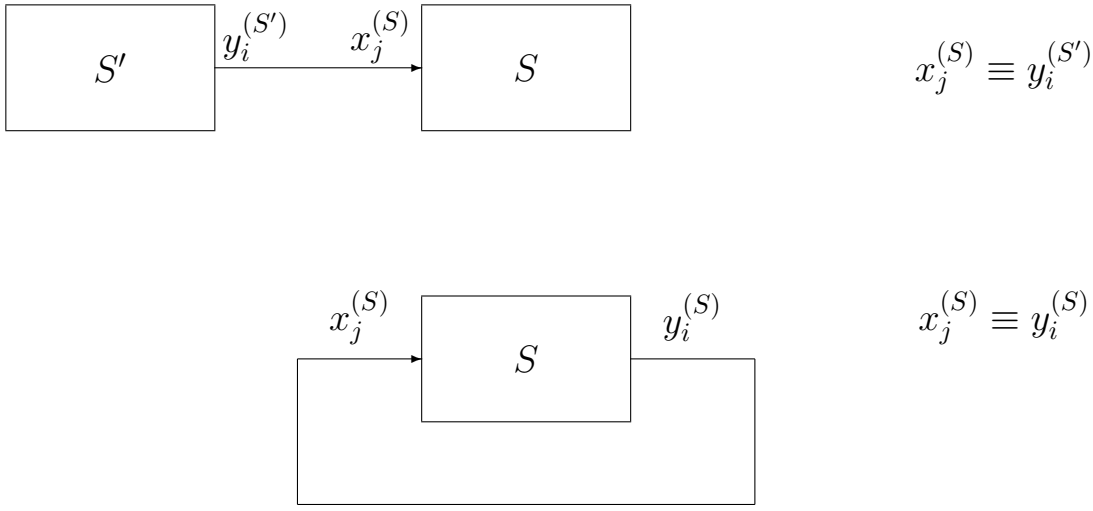



Abb. 3.1-1: Erzeugung einer Eingangszeitfunktion $x_j^{(S)}$

Der Pfeil stellt die Kopplung einer Ausgangsgröße an eine Eingangsgröße dar. Dadurch wird die Eingangsgröße $x_j^{(S)}$ mit der Ausgangsgröße $y_i^{(S')}$ gleichgesetzt. Wir sprechen auch von einer *Verbindung* oder *Connection* zwischen beiden Größen. Der Pfeil symbolisiert auch die Wirkung einer Komponente auf eine andere. Während eine Eingangsgröße nur mit genau einer Ausgangsgröße gleichgesetzt werden kann, ist die Verbindung einer

Ausgangsgröße mit mehreren Eingangsgrößen (der gleichen oder verschiedener Systeme) möglich.

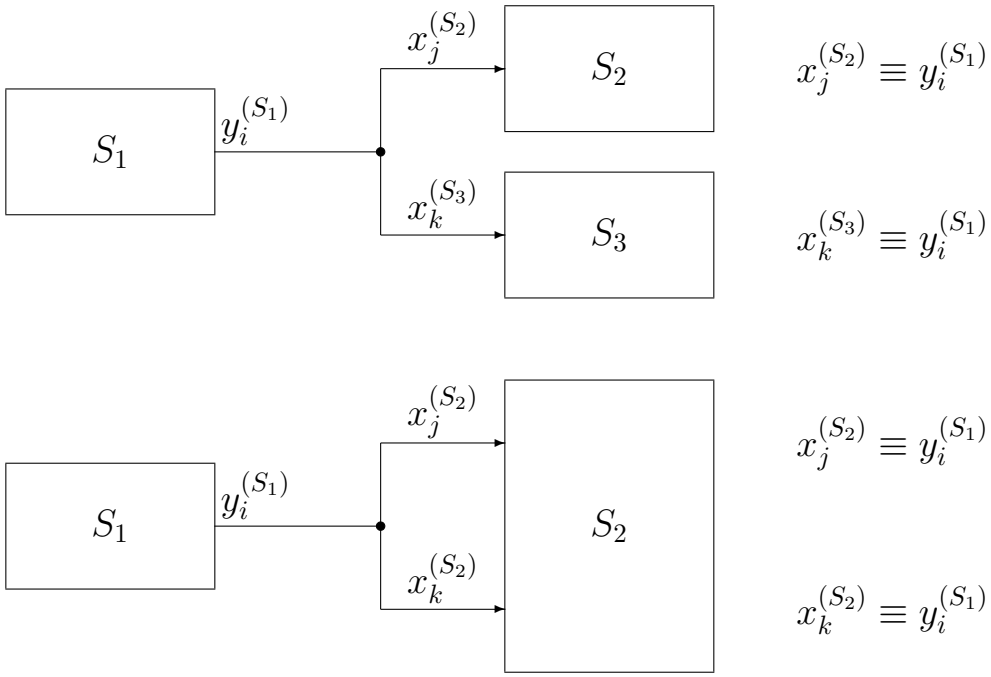



Abb. 3.1-2: Wirkung einer Ausgangsgröße auf mehrere Eingangsgrößen

Ein *abgeschlossenes Modell* liegt vor, wenn alle Eingangsgrößen der Teilsysteme durch Gleichsetzung mit Ausgangsgrößen abgedeckt sind. Aus den Modellgleichungen fallen dann alle Eingangsgrößen heraus, so daß ein abgeschlossenes Modell durch die dynamischen und statischen Beziehungen

$$\begin{aligned} \mathbf{Z}(t_{k+1}) &:= \mathbf{F}(\mathbf{Z}(t_k), t_k) \\ \mathbf{Y}(t_k) &:= \mathbf{G}(\mathbf{Z}(t_k), t_k) \end{aligned}$$

beschrieben wird.

Zur *Zerlegung* eines Systems oder eines abgeschlossenen Modells in $N$ Teilsysteme faßt man die Zustandsgrößen und Ausgangsgrößen in Gruppen zusammen und verteilt sie auf mehrere Systeme.

$$\mathbf{Z} = \begin{bmatrix} \mathbf{Z}^{(1)} \\ \mathbf{Z}^{(2)} \\ \vdots \\ \mathbf{Z}^{(N)} \end{bmatrix} \qquad \mathbf{Y} = \begin{bmatrix} \mathbf{Y}^{(1)} \\ \mathbf{Y}^{(2)} \\ \vdots \\ \mathbf{Y}^{(N)} \end{bmatrix}$$

Ebenso werden die Zustands- und Ausgangsgleichungen, welche das Verhalten der Zustands- und Ausgangsgrößen definieren, auf die Teilsysteme verteilt.

$$\begin{aligned} \mathbf{Z}^{(j)}(t_{k+1}) &:= \mathbf{F}^{(j)}(\mathbf{Z}(t_k), t_k) \\ \mathbf{Y}^{(j)}(t_k) &:= \mathbf{G}^{(j)}(\mathbf{Z}(t_k), t_k) \end{aligned} \qquad \text{für alle} \quad j \in \{1 \ldots N\}$$

Die Funktionen $\mathbf{F}^{(j)}$ und $\mathbf{G}^{(j)}$ greifen nun noch auf den Zustandsvektor des Gesamtmodells und damit auf Zustandsgrößen fremder Teilsysteme zurück.

Um diesen Übergriff transparent zu machen, werden Eingangsgrößen eingeführt, die dem jeweiligen Teilsystem zugeordnet sind und über die sich das Teilsystem Informationen über Zustandsgrößen anderer Systeme beschafft. Da die Systemtheorie nur Verbindungen zu Ausgangsgrößen vorsieht, sind andere Komponenten ggf. um weitere Ausgangsgrößen und Ausgangsgleichungen zu ergänzen.

$$\begin{aligned} \mathbf{Z}^{(j)}(t_{k+1}) &:= \mathbf{F}^{(j)}(\mathbf{X}^{(j)}(t_k), \mathbf{Z}^{(j)}(t_k), t_k) \\ \mathbf{Y}^{(j)}(t_k) &:= \mathbf{G}^{(j)}(\mathbf{X}^{(j)}(t_k), \mathbf{Z}^{(j)}(t_k), t_k) \end{aligned}$$

Der Wert einer Zustandsgröße bzw. einer Ausgangsgröße wird durch genau eine Gleichung bestimmt. Diese Beziehung gehört stets zu dem System, das die entsprechende Größe enthält. Ein System bestimmt daher einzig und allein über den Zeitverlauf seiner Zustands- und Ausgangsgrößen. Wir bezeichnen daher ein System auch als autonom. Die Zerlegung eines Systems genügt damit einem Aspekt, der dem Konzept der objekt-orientierten Programmierung als wesentlich zugrunde liegt.

### 3.1.3 Spezielle Überführungsfunktionen

Je nach gewählter Zeitmenge $\mathcal{T}$ und dem Typ der lokalen Überführungsfunktion $\mathbf{F}$ bzw. ihrer Komponenten $f_i$ ergeben sich spezielle Varianten von Modellbeschreibungen.

1. *Der allgemeine Automat*

$$\begin{aligned} \text{Zeitmenge} \qquad \mathcal{T} &= \{t_k \mid t_k = k, k \in N_0\} \\ z_i(t+1) &:= f_i(\mathbf{X}(t), \mathbf{Z}(t), t) \end{aligned} \tag{3.1.3-1}$$

2. *System von Differenzengleichungen*

$$\begin{aligned} \text{Zeitmenge} \qquad \mathcal{T} &= \{t_k \mid t_k = k \cdot \Delta t, k \in N_0, \Delta t \in R^+\} \\ z_i(t+\Delta t) &:= z_i(t) + \Delta t \cdot r_i^{(\Delta t)}(\mathbf{X}(t), \mathbf{Z}(t), t) \end{aligned} \tag{3.1.3-2}$$

   Die Funktion $r_i^{(\Delta t)}$ wird als nderungsrate der Größe $z_i$ im Zeitschritt $\Delta t$ bezeichnet.

3. *System von Differentialgleichungen*

$$\begin{aligned} \text{Zeitmenge} \qquad \mathcal{T} &= R \\ \lim_{\Delta t \to 0} z_i(t+\Delta t) &:= z_i(t) + \lim_{\Delta t \to 0} (\Delta t \cdot f_i^{'}(\mathbf{X}(t), \mathbf{Z}(t), t)) \end{aligned} \tag{3.1.3-3}$$

   Die Funktion $f_i^{'}$ wird als Differentialquotient oder Ableitung der Größe $z_i$ nach der Zeit bezeichnet.

Um die genannten Systeme analytisch behandelbar zu machen, schränkt man die Funktionen $f_i, g_i, r_i$ und $f_i^{'}$ meist auf lineare Funktionen ein. Neben geschlossenen analytischen Lösungen ist es der Automatentheorie bzw. Systemtheorie gelungen, eine Vielzahl interessanter Sätze herzuleiten, die den Umgang mit solchen linearen Systemen erheblich vereinfachen. Daher versucht man in ingenieur-wissenschaftlichen Anwendungen mit linearen Modellansätzen auszukommen.

Ist dies möglich, dann ist sicherlich der analytische Lösungsweg der geeignete und eine Simulation dient ggf. nur noch der Überprüfung der Ergebnisse.

Da wir den Nutzen der Simulation darin begründet sehen, möglichst allgemeine Aufgabenstellungen zu bewältigen, war es unser Anliegen, insbesondere diskrete Zustandsübergänge, wie sie durch den Automaten beschrieben werden, mit kontinuierlichen Zustandsübergängen, wie sie ein Differentialgleichungssystem darstellt, zu vereinen. Im folgenden Abschnitt wird gezeigt, wie dies durch eine geeignete kombinierte Überführungsfunktion erreicht werden kann.

## 3.2 Modifizierte Modelldarstellung

Die Welt der Systemtheorie ist stark an die Arbeitsweise mit physikalischen Komponenten angelehnt. Der Benutzer kann daher mit Modellbausteinen ebenso umgehen wie mit realen Systemen. Dieser Gesichtspunkt erscheint uns ganz entscheidend, um Simulationstechnik, die sich auf formale Modellbeschreibung stützt, beim praxis-orientierten Anwender einführen zu können.

Andererseits ist es notwendig, den systemtheoretischen Ansatz so zu erweitern und umzugestalten, daß eine Modellbeschreibung möglich ist, die gut lesbar ist, keine unnötige Redundanz aufweist und leistungsfähig genug ist.

### 3.2.1 Verwendete Terminologie

Mit dem Begriff System wird in der Systemtheorie sowohl eine physikalische Komponente als auch ein System mathematischer Beziehungen assoziiert.

Stattdessen wollen wir in Zukunft den Begriff *Modellkomponente* verwenden, der uns daran erinnern soll, daß wir uns eine *gedankliche Vorstellung* von einem realen System machen.

Ein reales System ist wie ein schwarzer Kasten anzusehen, dessen Vorgänge im Inneren unbekannt sind. Es tritt uns lediglich in Form einiger beobachtbarer Größen in Erscheinung. Die Geschehnisse in einem *Modell* hingegen sind vollständig bekannt und daher möchten wir sie während eines Simulationsexperiments auch beobachten. Ein Modell ist deshalb als durchsichtiger Kasten zu betrachten.

Dementsprechend sind alle Größen eines Modells bzw. einer Modellkomponente von außen zugänglich, d.h. nicht nur die von der Systemtheorie als Ausgangsgrößen bezeichneten Größen. Um Mißverständnisse zu vermeiden bezeichnen wir diese Größen im folgenden als *abhängige Größen*. Dieser Begriff soll deutlich machen, daß ihre Werte unmittelbar von Zustands- und Eingangsgrößen abhängig sind. *Ausgangsgrößen* umfassen in Zukunft sowohl Zustandsgrößen wie abhängige Größen.

In der systemtheoretischen Modelldarstellung sind lediglich Verbindungen von abhängigen Größen zu Eingangsgrößen vorgesehen. Wir wollen diese unnötige Einschränkung fallen lassen und daher auch Verbindungen von Zustandsgrößen zu Eingangsgrößen zulassen.

Eingangsgrößen dienen dazu, einer Modellkomponente Informationen über ihre Umgebung, d.h. über Größen aus benachbarten Modellkomponenten, zugänglich zu machen. Darum wollen wir künftig von *Sensorgrößen* sprechen.

### 3.2.2 Darstellung einer Modellkomponente

Eine Modellkomponente besteht aus

- dem Vektor der Zustandsgrößen $\mathbf{Z}$ mit Wertemenge $\mathcal{Z}$
- dem Vektor der abhängigen Größen $\mathbf{Y}$ mit Wertemenge $\mathcal{Y}$
- dem Vektor der Sensorgrößen $\mathbf{X}$ mit Wertemenge $\mathcal{X}$

Der Zeitverlauf der Zustandsgrößen und abhängigen Größen, d. h. die Dynamik der Modellkomponente, wird bestimmt durch

- die dynamischen Beziehungen (Zustandsüberführungsfunktion)

$$\begin{aligned} \mathbf{Z}(t_{k+1}) &:= \mathbf{F}(\mathbf{X}(t_k), \mathbf{Z}(t_k), t_k) && \text{(3.2.2-1)} \\ &:= \mathbf{F}(\mathbf{X}(t_k), \mathbf{Z}(t_k), \mathbf{Y}(t_k), t_k) && \text{(3.2.2-2)} \end{aligned}$$

- die statischen Beziehungen

$$\begin{aligned} \mathbf{Y}(t_k) &:= \mathbf{G}(\mathbf{X}(t_k), \mathbf{Z}(t_k), t_k) && \text{(3.2.2-3)} \\ &:= \mathbf{G}(\mathbf{X}(t_k), \mathbf{Z}(t_k), \mathbf{Y}(t_k), t_k) && \text{(3.2.2-4)} \end{aligned}$$

Für den Zustandsvektor ist ein Anfangszustand

$$\mathbf{Z}(t_0) := \mathbf{Z}_o$$

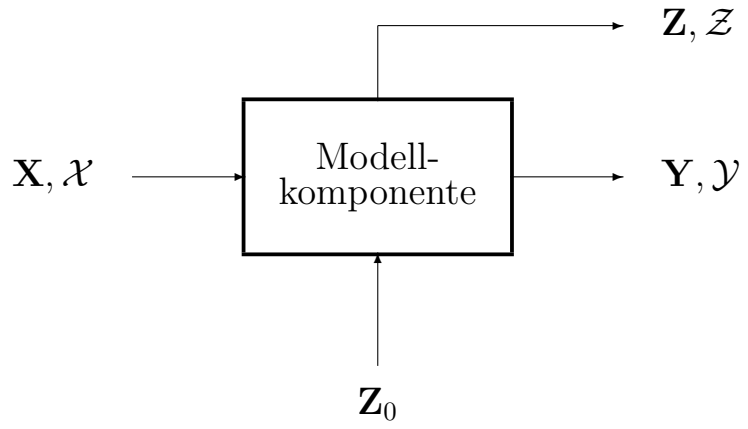
angebbar und für den Vektor der Sensorgrößen eine konstante Vorbelegung

$$\mathbf{X}(t_k) := \mathbf{X}_0 \qquad \forall \quad t_k \quad ,$$

die jedoch nur für solche Sensorgrößen $x_i$ gilt, die nicht angeschlossen sind.

Diese Belegung der Sensorgrößen läßt in beschränktem Umfang ein Arbeiten mit Modellkomponenten zu, deren Sensorgrößen nicht oder nur zum Teil angeschlossen sind.

Als Zeitmenge liegt die Menge der reellen Zahlen zugrunde.




Die dynamischen und statischen Beziehungen lassen wir auch in Abhängigkeit der abhängigen Größen $\mathbf{Y}$ zu. Dies führt zu kürzeren und übersichtlicheren Beziehungen bei der Modellbeschreibung und daher zu einer besseren Akzeptanz beim Anwender.

Die Darstellungen (3.2.2-1) und (3.2.2-3) werden wir hingegen für theoretische Betrachtungen bevorzugen.

### 3.2.3 Die Zustandsüberführungsfunktion

Die Leistungsfähigkeit einer Modellbeschreibung hängt in erster Linie von der zulässigen Zeitmenge und der möglichen Überführungsfunktion ab.

Wir wählen daher die reellen Zahlen als Zeitmenge und eine zusammengesetzte Überführungsfunktion, die kontinuierliche und diskrete Zustandsübergänge mit der gleichen Zustandsgröße gestattet.

Die Zustandsüberführungsfunktion hängt vom Sensorvektor $\mathbf{X}$, dem Zustandsvektor $\mathbf{Z}$ und der Zeit $t$ ab. Abkürzend fassen wir diese Größen im

*erweiterten Zustandsvektor*

$$\mathbf{W}(t) := \begin{bmatrix} \mathbf{X}(t) \\ \mathbf{Z}(t) \\ t \end{bmatrix}$$

zusammen. Seine Wertemenge ist das karthesische Produkt

$$\mathcal{W} = \mathcal{X} \times \mathcal{Z} \times \mathcal{T}.$$

Durch eine Differentialgleichung 1. Ordnung beschreiben wir einen *kontinuierlichen Zustandsübergang*, der für reellwertige Zustandsgrößen zulässig ist.

$$\mathcal{T} = R, \qquad z_i \in R$$

$$\lim_{\Delta t \to 0} z_i(t + \Delta t) \;:=\; z_i(t) + \lim_{\Delta t \to 0} \Delta t \cdot f_i'(\mathbf{W}(t)) \tag{3.2.3-1}$$

Die reellwertige Funktion $f_i'$ heißt Differentialquotient der Zustandsgröße $z_i$. Eine Differentialgleichung bewirkt eine stetige Veränderung einer Zustandsgröße.

Durch ein sogenanntes *Ereignis* wird ein *diskreter Zustandsübergang* beschrieben, der für Zustandsgrößen mit beliebiger Wertemenge zulässig ist.

$$\mathcal{T} = R$$

$$\lim_{\Delta t \to 0} z_i(t + \Delta t) \;:=\; \begin{cases} \hat{f}_i(\mathbf{W}(t)) & \text{falls} \quad \mathbf{W}(t) \in \mathcal{A}_i \\ \\ z_i(t) & \text{sonst} \end{cases} \tag{3.2.3-2}$$

Die Menge $\mathcal{A}_i$ ist eine Teilmenge der erweiterten Zustandsmenge $\mathcal{W}$

$$\mathcal{A}_i \subset \mathcal{W}$$

und heißt Auslösemenge des Ereignisses. Die Bedingung $(\mathbf{W}(t) \in \mathcal{A}_i)$ bezeichnen wir als Ereignisbedingung, die Funktion $\hat{f}_i$ als Ereignisfunktion. Sie ist nur für $\mathbf{W}(t) \in \mathcal{A}_i$ definiert.

Ein Ereignis bewirkt eine sprunghafte nderung einer Zustandsgröße, die quasi ohne Zeitverzug ausgeführt wird.

Für die Gesamtheit der in einem Modell enthaltenen Ereignisse

$$\lim_{\Delta t \to 0} \mathbf{Z}(t + \Delta t) \;:=\; \begin{cases} \hat{\mathbf{F}}(\mathbf{W}(t)) & \text{falls} \quad \mathbf{W}(t) \in \mathcal{A} \\ \mathbf{Z}(t) & \text{sonst} \end{cases} \tag{3.2.3-3}$$

$$\text{mit} \quad \mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \cup \ldots \mathcal{A}_{N_Z}$$

ist eine Konvergenzbedingung zu fordern:

Jeder Zustand $\mathbf{W}(t) \in \mathcal{A}$ muß nach endlich vielen Zeitinkrementen $\Delta t$ in einen Zustand $\mathbf{W}(t) \notin \mathcal{A}$ übergehen.

Nur wenn diese Konvergenzbedingung erfüllt ist, erhält man ein brauchbares Modellverhalten.

Für reellwertige Zustandsgrößen lassen sich die beiden Typen von Zustandsübergängen durch additive Überlagerung zusammenfassen:

$$\begin{aligned} \lim_{\Delta t \to 0} z_i(t + \Delta t) \;:=\; & \lim_{\Delta t \to 0} \Delta t \cdot f_i'(\mathbf{W}(t)) \\ & + \begin{cases} \hat{f}_i(\mathbf{W}(t)) & \text{falls} \quad \mathbf{W}(t) \in \mathcal{A}_i \\ z_i(t) & \text{sonst} \end{cases} \end{aligned} \tag{3.2.3-4}$$

Der erste Summand beschreibt den kontinuierlichen Zustandsübergang, der zweite Summand den diskreten Zustandsübergang der Zustandsgröße $z_i$.

Solange $\mathbf{W}(t) \notin \mathcal{A}_i$ gilt, wird nur der kontinuierliche Zustandsübergang ausgeführt. Sobald $\mathbf{W}(t) \in \mathcal{A}_i$ ist, findet ein Ereignis statt. Die Zustandsgröße $z_i$ verändert – quasi ohne Zeitverzug – ihren Wert auf $\hat{f}_i(\mathbf{W}(t))$, ohne daß die differentielle Zustandsänderung darauf Einfluß hat.

Setzt man die Konvergenz der Ereignisse voraus, kann man diesen kombinierten Zustandsübergang auch durch

$$\lim_{\Delta t \to 0} z_i(t+\Delta t) \;:=\; \begin{cases} \hat{f}_i(\mathbf{W}(t)) & \text{falls} \quad \mathbf{W}(t) \in \mathcal{A}_i \\ \\ z_i(t) + \lim\limits_{\Delta t \to 0} \Delta t \cdot f_i'(\mathbf{W}(t)) & \text{sonst} \end{cases} \tag{3.2.3-5}$$

darstellen.

In bisherigen Darstellungen (vgl. GEST) ordnet man die Zeitmenge und den zulässigen Typ von Zustandsübergängen der Modellkomponente zu. Man unterscheidet zwischen diskreten und kontinuierlichen Komponenten. Dies hat zur Folge, daß hiervon alle Größen einer Komponente betroffen sind. Die hier vorgenommene Vereinheitlichung der Zustandsübergangsfunktion bietet viel größere Freiheiten bei der Gestaltung der Modellkomponenten.

Zur Beschreibung des Zustandsübergangs einer Zustandsgröße $z_i$ werden wir zukünftig auch die folgende, abkürzende Schreibweise verwenden.

Für kontinuierliche Zustandsübergänge:

$$z_i' \;:=\; f_i'(\mathbf{W}(t)) \tag{3.2.3-6}$$

Für diskrete Zustandsübergänge:

$$\hat{z}_i \;:=\; \hat{f}_i(\mathbf{W}(t)) \quad \text{immer wenn} \quad \mathbf{W}(t) \in \mathcal{A}_i \tag{3.2.3-7}$$

Ein kombinierter Zustandsübergang liegt vor, wenn beide Arten von Zustandsübergängen für die gleiche Zustandsgröße angegeben sind.

Ist die Ereignisfunktion $\hat{f}_i$ eine abschnittsweise zusammengesetzte Funktion

$$\hat{f}_i(\mathbf{W}(t)) = \begin{cases} \hat{f}_i^{(1)}(\mathbf{W}(t)) & \text{für} \quad \mathbf{W}(t) \in \mathcal{B}_i^{(1)} \\ \vdots & \\ \hat{f}_i^{(n)}(\mathbf{W}(t)) & \text{für} \quad \mathbf{W}(t) \in \mathcal{B}_i^{(n)} \end{cases}$$

$$\text{mit} \qquad B_i^{(j)} \cap B_i^{(k)} = \emptyset \qquad \forall \quad j \neq k \quad ,$$

dann ergibt sich für die Formulierung eines Ereignisses eine interessante Variante, wenn man die Bedingungen der Fallunterscheidung in die Ereignisbedingung mit aufnimmt.

$$\hat{z}_i := \begin{cases} \hat{f}_i^{(1)}(\mathbf{W}(t)) & \text{immer wenn } \mathbf{W}(t) \in \mathcal{A}_i \cap \mathcal{B}_i^{(1)} \\ \vdots & \\ \hat{f}_i^{(n)}(\mathbf{W}(t)) & \text{immer wenn } \mathbf{W}(t) \in \mathcal{A}_i \cap \mathcal{B}_i^{(n)} \end{cases} \tag{3.2.3-8}$$

Die Ereignisbedingung $\mathcal{A}_i \cap \mathcal{B}_i^{(j)}$ ist häufig wesentlich einfacher als die einzelnen Bedingungen und daher erscheint es sinnvoll, auch die Form (3.2.3-8) zur Beschreibung eines diskreten Zustandsübergangs zuzulassen.

### 3.2.4 Modularer, hierarchischer Modellaufbau

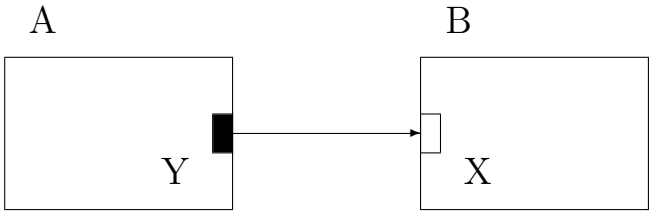
Wir haben bislang einzelne Modellkomponenten beschrieben und untersuchen nun, wie man zusammengesetzte Modellkomponenten bzw. ganze Modelle aufbaut.

Zukünftig wollen wir zwei Typen von Modellkomponenten unterscheiden:

Komponenten, die statische und dynamische Beziehungen enthalten, werden *Grundkomponenten* oder *Basiskomponenten* genannt.

Größere Einheiten entstehen, indem Sonsorgrößen mit Zustandsgrößen oder abhängigen Größen gleichgesetzt werden. Eine Komponente, die Verknüpfungen zwischen untergeordneten Komponenten beschreibt, wird als *höhere Komponente* bezeichnet.

Verknüpfungen zwischen Komponenten nennen wir auch *Component Connections*. Sie lassen sich als bildhafte Darstellung wiedergeben.




$B.X \equiv A.Y;$

Abb. 3.2.4-1: Component Connection

Die Sensorgröße $X$ der Komponente $B$ wird mit der abhängigen Größe (oder Zustandsgröße) $Y$ der Komponente $A$ gleichgesetzt.

Wir stellen Sensorgrößen durch ein leeres Kästchen, Zustandsgrößen oder abhängige Größen durch ein ausgefülltes Kästchen dar. Die Gleichsetzung zweier Größen wird durch einen Pfeil symbolisiert, der auf die Sensorgröße gerichtet ist.

**Beispiel:** Modell eines Dreiecksgenerators

Das folgende Beispielmodell besteht aus 3 Basiskomponenten und einer höheren Komponente.

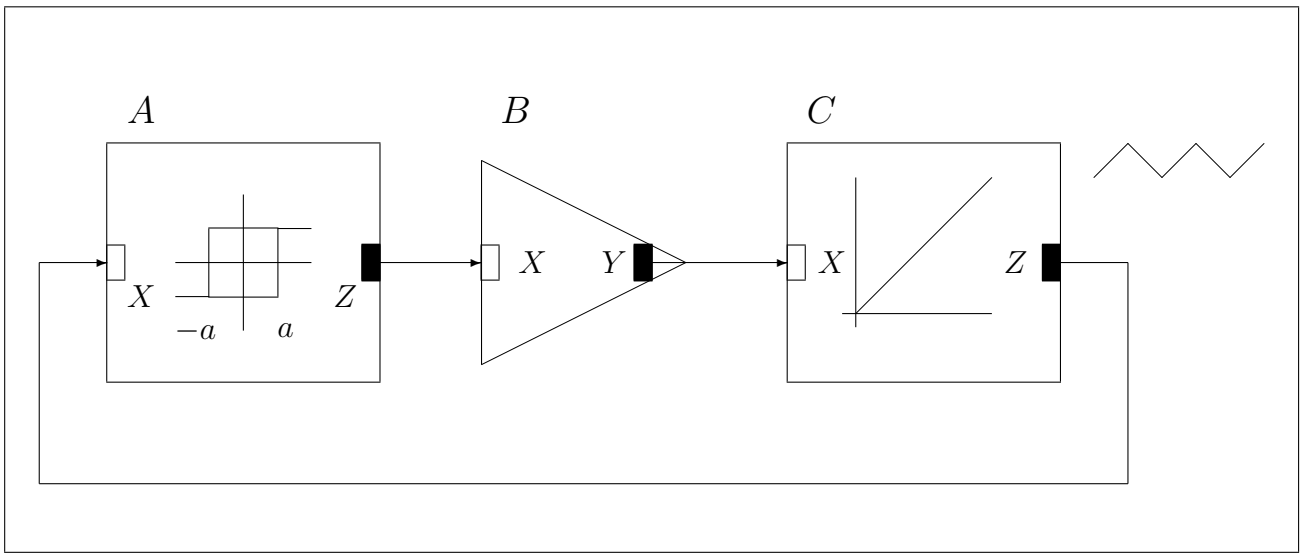



Abb. 3.2.4-2: Modell eines Dreiecksgenerators

Das dynamische Verhalten des Hysteresegliedes $A$ wird durch das Ereignis

$$\hat{Z} := \begin{cases} 1 & \text{immer wenn } X > a \quad \wedge \quad Z \leq 0 \\ -1 & \text{immer wenn } X < -a \quad \wedge \quad Z > 0 \end{cases}$$

beschrieben. Die Zustandsgröße $Z$ ist mit $+1$ oder $-1$ vorzubesetzen.

Der Inverter $B$ ist durch die statische Beziehung

$$Y := -X;$$

definiert.

Das Verhalten des Integrierers $C$ ist durch die Differentialgleichung

$$Z' := X;$$

gegeben.

Diese drei Basiskomponenten sind im Gesamtmodell durch die Gleichsetzungen

$$B.X \equiv A.Z; \qquad C.X \equiv B.Y; \qquad A.X \equiv C.Z;$$

gekoppelt, welche den Inhalt einer höheren Komponente ergeben. Die Schreibweise *Subkomponente.Modellgröße* identifiziert die Modellgröße einer bestimmten Subkomponente.

Um Modelle hierarchisch aufbauen zu können, müssen auch für höhere Komponenten Anschlußstellen zur Umgebung geschaffen werden. Das bedeutet, daß man Modellgrößen untergeordneter Komponenten für eine Komponente der nächst höheren Hierarchiestufe zugänglich machen muß.

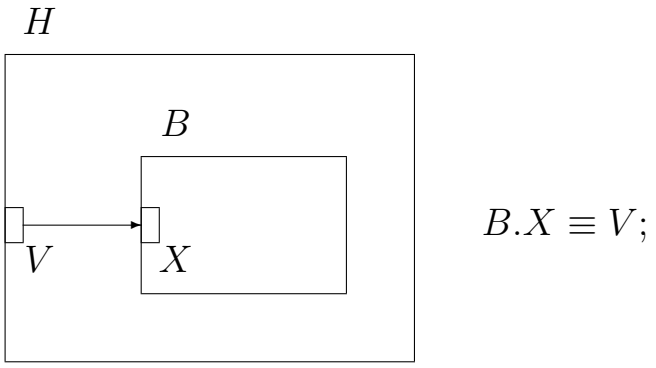



Abb. 3.2.4-3: Input Connection

Die Sensorgröße $X$ der Komponente $B$ setzt man der Sensorgröße $V$ gleich, die man zu diesem Zweck in der Komponente $H$ einführt. Diese Art der Verbindung wollen wir als *Input Connection* bezeichnen.

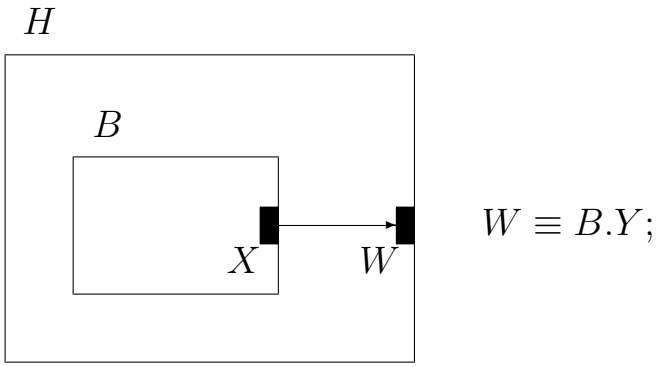



Abb. 3.2.4-5: Output Equivalence

In der Komponente $H$ wird eine Ausgangsgröße $W$ eingeführt, die man einer Zustandsgröße oder abhängigen Größe $Y$ der untergeordneten Komponente $B$ gleichsetzt. Diese Art der Verbindung wollen wir *Output Equivalence* nennen.

Eine Sensorgröße darf selbstverständlich nur mit genau einer Zustandsgröße oder abhängigen Größe gleichgesetzt werden. Daraus ergibt sich die Regel, daß sich in Pfeilrichtung Verbindungen aufteilen dürfen, während entgegen der Pfeilrichtung stets eine eindeutige Zuordnung hergestellt sein muß.

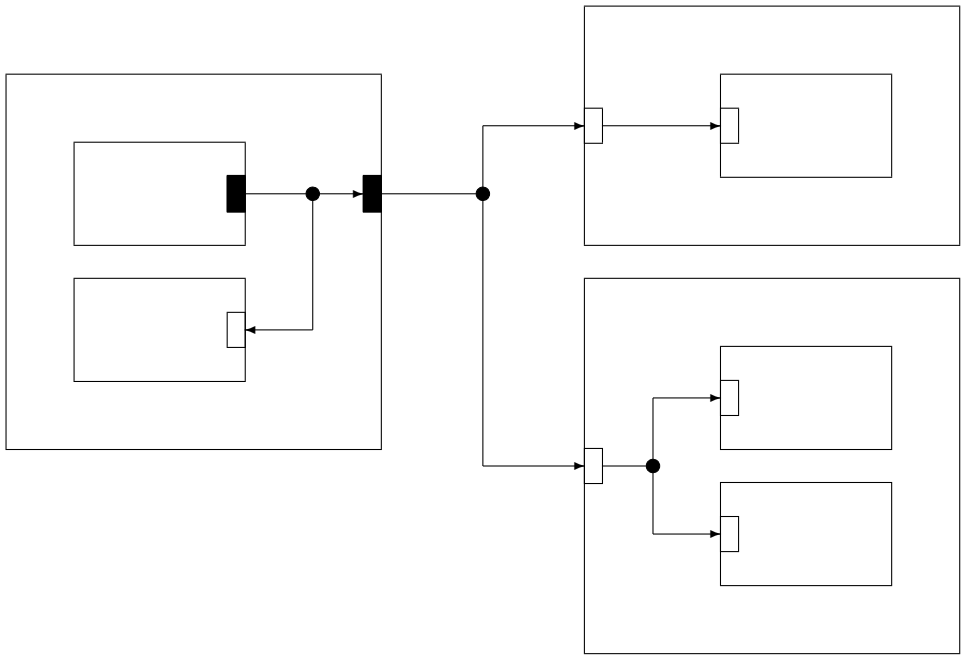


Abb. 3.2.4-6: Mögliche Auffächerung der Verbindungen

Damit verschaltete Komponenten ein abgeschlossenes Modell verkörpern, müssen alle Sensorgrößen einer Zustandsgröße oder abhängigen Größe zugeordnet sein. Diese starke Bedingung verhindert jedoch, daß einzelne Modellkomponenten betrieben oder auch nur getestet werden können, wenn sie offene Eingänge besitzen.

Aus diesem Grund erscheint es sinnvoll, eine Vorbesetzung für Sensorgrößen einzuführen, die genau dann verwendet wird, wenn eine Sensorgröße nicht angeschlossen ist.

## 3.3 Fallstudien zur Ereignisbearbeitung

Während die Formulierung von kontinuierlichen Zustandsübergängen durch Differentialgleichungen seit langem praktiziert wird, hat die Beschreibung diskreter Zustandsübergänge erst seit dem Aufkommen digitaler Rechenautomaten Bedeutung erlangt. Die Definition diskreter Zustandsübergänge, wie sie in der Automatentheorie vorgenommen wird, schafft eine gute Grundlage zur Spezifikation diskreter Modelle. Insbesondere Gleichzeitigkeiten werden eindeutig behandelt.

Betrachtet man die existierenden Simulationssprachen, stellt man fest, daß Ereignisse, also diskrete Zustandsübergänge, den Wert der Zustandsvariablen sofort ändern und in der Reihenfolge ausgeführt werden, in der sie beim Programmdurchlauf erreicht werden. Ereignisse beeinflussen sich daher gegenseitig. Dies hat zur Folge, daß der Modellablauf von der Abarbeitungsreihenfolge der Ereignisse abhängt. Dieses Konzept ist für den Benutzer schwer zu handhaben und spätestens dann unbrauchbar, wenn ein Modell in mehrere Komponenten zerlegt wird und die Reihenfolge der Bearbeitung dem Benutzer nicht mehr bekannt ist.

Daher haben wir das Konzept der Automatentheorie aufgegriffen und mit kontinuierlichen Zustandsübergängen kombiniert. Ein Ereignis beschreibt den Wert einer Zustandsvariablen für einen künftigen Zeitpunkt.

$$\hat{z}_i := \hat{f}_i(\mathbf{W}(t)) \qquad \text{immer wenn} \quad \mathbf{W}(t) \in \mathcal{A}_i$$

Dieser Zeitpunkt liegt zwar nur einen infinitesimal kleinen zeitlichen Abstand entfernt, besitzt aber einen eigenen Modellzustand.

$$\lim_{\Delta t \to 0} z_i(t + \Delta t) := \begin{cases} \hat{f}_i(\mathbf{W}(t)) & \text{falls} \quad \mathbf{W}(t) \in \mathcal{A}_i \\ z_i(t) & \text{sonst} \end{cases}$$

Solange die Auslösebedingung $\mathbf{W}(t) \in \mathcal{A}_i$ gilt, rückt die Zeit um infinitesimale Einheiten $\Delta t$ weiter, und es ist quasi kein Zeitfortschritt vorhanden. Das Modell wird nach dem Schema der Automatentheorie bearbeitet. Wird $\mathbf{W}(t) \notin \mathcal{A}_i$, bleibt der Wert der Zustandsgröße $z_i$ solange erhalten, bis die Auslösebedingung wieder erfüllt ist. Das kann nur zu einem späteren Zeitpunkt der Fall sein. Auf diese Art findet ein ständiger Wechsel zwischen Ereignisausführung und Zeitfortschaltung statt.

Anhand einiger Fallstudien wird die Wirkung von Ereignissen eingehend studiert.

### a) Ereignisse mit identischer Auslösemenge

Beispiel 1:

$$\left.\begin{array}{lcl} \hat{z}_1 & := & 5 \\ \hat{z}_2 & := & z_1 + z_2 \end{array}\right\} \text{ immer wenn } \quad z_1 \leq 1$$

Zustand $\mathbf{Z}(t)$ : $\quad z_1 = 1 \quad z_2 = 2$
Zustand $\mathbf{Z}(t + \Delta t)$ : $\quad z_1 = 5 \quad z_2 = 3$

Beispiel 2:

$$\left.\begin{array}{lcl} \hat{z}_1 & := & z_2 \\ \hat{z}_2 & := & z_1 \end{array}\right\} \text{ immer wenn } \quad z_1 > z_2$$

Zustand $\mathbf{Z}(t)$ : $\quad z_1 = 5 \quad z_2 = 3$
Zustand $\mathbf{Z}(t + \Delta t)$ : $\quad z_1 = 3 \quad z_2 = 5$

### b) Gleichzeitiges Eintreffen verschiedener Ereignisbedingungen

Beispiel 3:

$\hat{z_1} := 0 \quad$ immer wenn $\quad (x_1 \geq 0) \wedge (z_1 \geq 5)$

$\hat{z_2} := z_1 \quad$ immer wenn $\quad (x_1 \leq 2) \wedge (z_2 \leq 5)$

Zustand $\mathbf{Z}(t)$ : $\quad x_1 = 1 \quad z_1 = 7 \quad z_2 = 3$
Zustand $\mathbf{Z}(t + \Delta t)$ : $\quad x_1 = 1 \quad z_1 = 0 \quad z_2 = 7$

### c) Ereignisfortpflanzung ohne Zeitverzug

Wenn durch die Ausführung eines Ereignisses ein weiteres Ereignis ausgelöst wird, sprechen wir von Ereignisfortpflanzung. Wir unterscheiden Ereignisfortpflanzungen mit und ohne Zeitverzug. Kein Zeitverzug liegt vor, wenn das Folgeereignis zum unmittelbar folgenden Zeitpunkt stattfindet.

Beispiel 4:

$$\begin{array}{ll} \hat{z}_1 := 0 & \text{immer wenn } (z_1 \geq 5) \\ \hat{z}_2 := 2 & \text{immer wenn } (z_1 = 0) \wedge (z_2 \geq 3) \end{array}$$

$$\begin{array}{lll} \text{Zustand } \mathbf{Z}(t): & z_1 = 10 & z_2 = 5 \\ \text{Zustand } \mathbf{Z}(t+\Delta t): & z_1 = 0 & z_2 = 5 \\ \text{Zustand } \mathbf{Z}(t+2\Delta t): & z_1 = 0 & z_2 = 0 \end{array}$$

Beispiel 5:

$$\hat{z} := z - 1 \quad \text{immer wenn } (z \geq 3)$$

$$\begin{array}{ll} \text{Zustand } \mathbf{Z}(t): & z = 3 \\ \text{Zustand } \mathbf{Z}(t+\Delta t): & z = 2 \\ \text{Zustand } \mathbf{Z}(t+2\Delta t): & z = 1 \\ \text{Zustand } \mathbf{Z}(t+3\Delta t): & z = 0 \end{array}$$

Die Ereignisfunktionen $\hat{f}_i(\mathbf{W}(t))$ müssen so beschaffen sein, daß die Ereignisfortpflanzungen ohne Zeitverzug für jeden zulässigen Zustand $\mathbf{W}(t)$ terminieren (Konvergenzbedingung). Ist dies nicht der Fall, liegt keine korrekte Modellbildung vor.

Beispiel 6:

$$\hat{z} := \begin{cases} -z & \text{immer wenn } z < 0; \\ -1 & \text{immer wenn } z \geq 1 \end{cases}$$

$$\begin{array}{ll} \text{Zustand } \mathbf{Z}(t): & z = -5 \\ \text{Zustand } \mathbf{Z}(t + \Delta t): & z = +5 \\ \text{Zustand } \mathbf{Z}(t + 2\Delta t): & z = -1 \\ \text{Zustand } \mathbf{Z}(t + 3\Delta t): & z = +1 \\ \text{Zustand } \mathbf{Z}(t + 4\Delta t): & z = -1 \\ \text{Zustand } \mathbf{Z}(t + 5\Delta t): & usw. \end{array}$$

Sobald $z$ den Bereich $|z| < 1$ verläßt, ist die Konvergenz der Ereignisse nicht mehr gegeben.

### d) Ereignisfortpflanzung mit Zeitverzug

Auslösebedingungen von Ereignissen dürfen die Zeit $t$ enthalten. Dadurch kann die Auslösung an einen bestimmten Zeitpunkt gebunden werden.

Beispiel 7:

$$\hat{z} := 1 \quad \text{immer wenn} \quad (t = 10) \wedge (z \neq 1)$$

$$\begin{array}{ll} \text{Zustand } \mathbf{Z}(t = 0): & z = 0 \\ \text{Zustand } \mathbf{Z}(t = 10): & z = 0 \\ \text{Zustand } \mathbf{Z}(t = 10 + \Delta t): & z = 1 \end{array}$$

Beispiel 8:

$$\hat{z} := t + 1 \quad \text{immer wenn} \quad (t = z)$$

$$\begin{array}{ll} \text{Zustand } \mathbf{Z}(t = 0): & z = 0 \\ \text{Zustand } \mathbf{Z}(t = 0 + \Delta t): & z = 1 \\ \text{Zustand } \mathbf{Z}(t = 1): & z = 1 \\ \text{Zustand } \mathbf{Z}(t = 1 + \Delta t): & z = 2 \quad usw. \end{array}$$

## 3.4 Formale Darstellung funktionaler Zusammenhänge

Eine Modellbeschreibung ordnet jeder einzelnen Modellgröße einen Wert zu, der abhängig ist von anderen Modellgrößen des gleichen oder vorangegangenen Zustands. Diese Abhängigkeit wird durch einen funktionalen, also eindeutigen Zusammenhang beschrieben.

$$f: \quad \mathcal{X} \to \mathcal{Y} \qquad \text{bzw.} \qquad y = f(x) \qquad \text{mit} \quad x \in \mathcal{X} \quad \text{und} \quad y \in \mathcal{Y}$$

Durch die Funktion $f$ wird jedem Element $x^i$ einer Urmenge $\mathcal{X}$ genau ein Element $y^j$ einer Bildmenge $\mathcal{Y}$ zugeordnet. Eine Funktion besteht daher aus ebensovielen Zuordnungen wie Elementen ihrer Urmenge:

$$\begin{aligned} x^1 &\to y(x^1) \\ x^2 &\to y(x^2) \\ &\vdots \end{aligned}$$

Ist die Urmenge das kartesische Produkt mehrerer Teilmengen

$$\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_n \quad ,$$

wird die Funktion in Abhängigkeit mehrerer Variablen angegeben

$$y = f(x_1, x_2, \ldots, x_n)$$

und als n-stellig bezeichnet. Jedes Element der Urmenge ist ein n-Tupel $(x_1^i, x_2^j, \ldots, x_n^k)$ aus je einem Element der Teilmengen.
Zur Darstellung einer Funktion sind verschiedene Möglichkeiten denkbar:

- tabellarische Darstellung
- algebraische Darstellung
  (explizit, implizit, evtl. mit Fallunterscheidung)
- algorithmische Darstellung

Diese Darstellungen sind auch miteinander kombinierbar. So kann ein algebraischer Ausdruck wiederum Funktionen enthalten.

### 3.4.1 Tabellarische Darstellung von Funktionen

Die tabellarische Darstellung einer Funktion beruht auf einer Aufzählung der einzelnen Zuordnungen zwischen Urmenge und Bildmenge.

Dies ist vollständig realisierbar, solange die Urmenge klein genug ist. Ist dies nicht der Fall, vor allem bei einer reellwertigen Urmenge, muß man sich Zwischenwerte beispielsweise durch Interpolation verschaffen.

Wegen der geringfügigen Bedeutung mehrstelliger Tabellenfunktionen wollen wir uns auf tabellarische Funktionen mit einer Variablen beschränken.

Die Definition einer Tabellenfunktion umfaßt die folgende Information

- Name der Tabellenfunktion
- Urmenge und Bildmenge
- Liste von Zuordnungen, die jedem Element der Urmenge genau ein Element der Bildmenge zuordnet.
- Angaben zum Interpolationsverfahren bei reellwertiger Urmenge

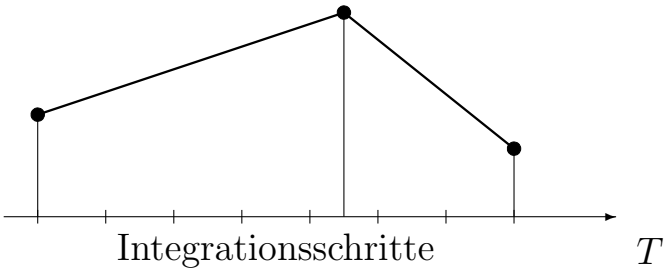
Als Beispiel diene die Definition einer Parabel in SIMPLEX-MDL

```
TABULAR FUNCTION  Parabel (REAL --> REAL)
CONTINUOUS BY CUBIC BESSEL INTERPOLATION
ON  (-2, -1, 0, 1, 2) -->
    ( 4,  1, 0, 1, 4)
```

Statt einzelner Zuordnungen wurden die Stützstellen und die Stützwerte in je einer Liste zusammengefaßt.

Die Wahl des Interpolationsverfahrens ist nicht nur für die Genauigkeit bedeutsam, mit der Zwischenwerte konstruiert werden, sondern hat auch Einfluß auf den Integrationsfehler, wenn eine Stützstelle im Integrationsintervall liegt.

**Beispiel:**




An einer Stützstelle macht die interpolierte Funktion einen mehr - oder weniger starken Knick, der einen größeren Integrationsfehler verursacht. Deshalb sollten die zur Verfügung gestellten Interpolationsverfahren so gewählt sein, daß der Benutzer auf die Glättung des Kurvenverlaufs Einfluß nehmen kann.

Es erscheinen daher die folgenden 3 Verfahren sinnvoll:

1. lineare Interpolation mit Unstetigkeit in der 1. Ableitung
2. kubische Bessel-Interpolation mit Unstetigkeit in der 2. Ableitung
3. kubische Spline-Interpolation mit Unstetigkeit in der 3. Ableitung

Die kubischen Interpolationsverfahren verbinden die Stützstellen durch ein Polynom 3. Grades. Die Koeffizienten dieses Polynoms errechnen sich aus den Werten der beiden benachbarten Stützstellen sowie aus Schätzungen für die Steigung des Funktionsverlaufs an diesen Stützstellen.

Die kubische Bessel-Interpolation schätzt die Steigung an einer Stützstelle, indem eine Parabel durch die beiden Nachbarpunkte gelegt wird. Für die Steigungen der Randpunkte wird die Parabel durch die nächsten zwei Stützstellen zur Mitte hin gelegt.

Die kubische Spline-Interpolation schätzt die Steigungen an den Randpunkten ebenso wie die kubische Bessel-Interpolation, falls diese nicht vorgegeben wurden. Der Spline-Algorithmus liefert dann die übrigen Steigerungen so, daß bei Interpolation mit einem kubischen Polynom auch die 2. Ableitungen stetig werden.

### 3.4.2 Algebraische Darstellung von Funktionen

Ein algebraischer Ausdruck verknüpft die Größen $x_1, \ldots, x_n$ der Urmenge und die Größe $y$ der Bildmenge mittels Operatoren und Funktionen zu einer Gleichung.

**Beispiele:**

| | |
|---|---|
| $y = ax;$ | $y = f(x)$ |
| $y = \sqrt{x_1^2 + x_2^2};$ | $y = f(x_1, x_2)$ |
| $x^2 + xy + y^2 = 0;$ | $f(x, y) = 0$ |

Läßt sich eine solche Gleichung nach $y$ auflösen, sagt man, $y$ ist explizit darstellbar. Ist dies nicht möglich, spricht man von einer impliziten algebraischen Gleichung.

Die Formulierung einer expliziten algebraischen Gleichung entspricht einer Wertzuweisung, wie sie in jeder höheren prozeduralen Porgrammiersprache zu finden ist.

Für den Fall, daß eine Beziehung zwischen Urmenge und Bildmenge nur implizit darstellbar ist, müssen besondere Vorkehrungen getroffen werden.

#### Implizite algebraische Gleichungen

Implizite algebraische Gleichungen liefern in vielen Fällen nicht genau eine Lösung, sondern häufig auch gar keine oder mehrere Lösungen. Bei einer korrekten Erstellung des Modells besitzt eine implizite Gleichung aber wenigstens eine Lösung. Existieren mehrere Lösungen, müssen die überzähligen durch geeignete Maßnahmen ausgeschlossen werden, da die Auswertung der Modellgleichungen ein eindeutiges Verhalten des Modells verlangt.

Implizite algebraische Gleichungen lassen sich numerisch nur durch ein iteratives Verfahren lösen. Die gesuchte Größe $y$ wird solange variiert, bis die Gleichung mit genügender Genauigkeit erfüllt ist. Es ist stets das Ziel, mit möglichst wenigen Iterationsschritten die Lösung mit der erforderlichen Genauigkeit zu finden.

Aus diesem Grund wird man die implizite algebraische Gleichung

$$f(y, x_1, x_2, \ldots, x_n) = 0$$

durch eine Interationsformel ersetzen, die schnell und sicher gegen die Lösung konvergiert.

$$y^{(i+1)} := g(y^{(i)}, x_1, x_2, \ldots, x_n)$$

Als Abbruchkriterium kommt eine relative Abweichung

$$\mid y^{(i+1)} - y^{(i)} \mid \quad < \quad \rho \cdot \mid y^{(i+1)} \mid$$

ober – falls $y^{(i+1)}$ zu klein werden sollte – eine absolute Abweichung

$$\mid y^{(i+1)} - y^{(i)} \mid \quad < \quad \varepsilon$$

in Betracht.

Falls mehrere Lösungen existieren, kann man durch eine geeignete Vorgabe des Startwertes $y^{(0)}$ erreichen, daß die Iteration gegen die richtige Lösung konvergiert.

Die Umformung einer impliziten Gleichung in eine explizite Iterationsformel ist nicht automatisierbar und daher vom Anwender zu leisten. Insbesondere muß die Konvergenz für alle möglichen $y, x_1, \ldots, x_n$ sichergestellt sein. Es gibt aber einige Standardverfahren, die solche Umformungen leisten und sich in der Praxis bewährt haben.

Am meisten Bedeutung hat das Newtonsche Iterationsverfahren erlangt, das sich durch eine hohe Konvergenzgeschwindigkeit auszeichnet. Es ist dabei allerdings erforderlich, die Funktion symbolisch zu differenzieren, was nicht immer ohne weiteres möglich ist.

$$y^{(i+1)} := y^{(i)} - \frac{f(y^{(i)}, x_1, \ldots, x_n)}{\frac{d}{dy} f(y^{(i)}, x_1, \ldots, x_n)}$$

Eine Modellbeschreibungssprache muß daher Vorkehrungen treffen, damit der Benutzer solche Iterationsformeln angeben kann und diese entsprechend ausgewertet werden.

Die größte Bedeutung haben implizite algebraische Gleichungen bei der Formulierung statischer Beziehungen. Da diese, wie wir noch sehen werden, ohnehin iterativ ausgewertet werden, kommt man in diesem Fall sogar ohne zusätzliche sprachliche Konstruktionen aus.

**Fallunterscheidungen**

Manchmal läßt sich eine funktionale Beziehung nicht in einem einzigen algebraischen Ausdruck darstellen, sondern man benötigt dazu mehrere Ausdrücke, von denen jeweils nur einer Gültigkeit besitzt.

$$y = f(x) = \begin{cases} f^{(1)}(x) & \text{für} \quad x \in \mathcal{X}^{(1)} \\ f^{(2)}(x) & \text{für} \quad x \in \mathcal{X}^{(2)} \\ \vdots & \\ f^{(n)}(x) & \text{für} \quad x \in \mathcal{X}^{(n)} \end{cases}$$

Die dazu notwendige Fallunterscheidung zerlegt die Urmenge $\mathcal{X}$ in einzelne Gebiete $\mathcal{X}^{(1)}, \mathcal{X}^{(2)}, \ldots, \mathcal{X}^{(n)}$. Damit nur einer der algebraischen Ausdrücke $f^{(1}(x), f^{(2)}(x), \ldots, f^{(n)}(x)$ Gültigkeit besitzt, dürfen sich die einzelnen Gebiete nicht überschneiden, d. h. sie müssen disjunkt sein.

$$\mathcal{X}^{(i)} \cap \mathcal{X}^{(j)} = \phi \qquad \forall \quad i \neq j$$

Weiterhin müssen die genannten Gebiete die gesamte Urmenge beinhalten, d. h.

$$\bigcup_{i=1}^{n} \mathcal{X}^{(i)} = \mathcal{X}.$$

Diese beiden Forderungen für eine korrekte Formulierung einer Fallunterscheidung können von einem Compiler nur sehr schwer überprüft werden. Deshalb erscheint uns eine Formulierung, die zwar in der Mathematik unüblich ist, in höheren prozeduralen Programmiersprachen sich aber als Standard eingebürgert hat, für geeigneter.

Hierbei werden die Bedingungen der Fallunterscheidung sukzessive überprüft. Die erste Bedingung, die erfüllt ist, weist auf den gültigen Ausdruck.

$$y := f(x) = \begin{cases} & f^{(1)}(x) & \text{für} & x \in \mathcal{X}^{(1)} \\ \text{sonst} & f^{(2)}(x) & \text{für} & x \in \mathcal{X}^{(2)} \\ \vdots & & & \\ \text{sonst} & f^{(n)}(x) & & \end{cases}$$

Auf diese Art ist sichergestellt, daß immer genau ein Ausdruck gültig ist. Auch die Eingrenzung der Gebiete $\mathcal{X}^{(1)}, \ldots, \mathcal{X}^{(n-1)}$ ist nun einfacher, da ein Gebiet $\mathcal{X}^{(i)}$ nur noch aus der Menge $\mathcal{X} \backslash \mathcal{X}^{(1)} \backslash \ldots \backslash \mathcal{X}^{(i-1)}$ ausgegrenzt werden muß.

Aus den genannten Gründen empfiehlt sich auch für eine deklarative Modellbeschreibungssprache der Einsatz eines IF-THEN-ELSIF-ELSE-Konstruktes. Es bietet zudem den Vorteil, daß mehrere Funktionen, die nach der gleichen Fallunterscheidung zerlegt werden müssen, im gleichen Konstrukt untergebracht werden können.

**Beispiel:**

| | | | | |
|---|---|---|---|---|
| IF | $X > 0$ | THEN | $Y := X + 1;$ | |
| ELSIF | $X < 0$ | THEN | $Y := X - 1;$ | |
| | | ELSE | $Y := 0;$ | ENDIF |
| | Bedingungsteil | | Definitionsteil | |

Da dieses Konstrukt aber die Bedingungen an den Anfang stellt und im Definitionsteil beliebigen Inhalt zuläßt, handelt man sich wiederum andere semantische Probleme ein.

Es muß nämlich sichergestellt werden, daß im Definitionsteil jeweils dieselbe Größe (hier: $Y$) definiert wird. Diese Prüfung kann jedoch von einem Compiler - wenn auch mit einigem Aufwand - ausgeführt werden.

### 3.4.3 Algorithmische Darstellung von Funktionen

In manchen Fällen kann ein funktionaler Zusammenhang weder durch Tabellenfunktionen noch durch algebraische Gleichungen geeignet beschrieben werden.

Dies trifft beispielsweise auf mathematische Standardfunktionen wie $sin(x)$, $ln(x)$ oder $\sqrt{x}$ zu. Während solche Standardfunktionen immer wieder benötigt und daher zweckmäßigerweise von der Sprache zur Verfügung gestellt werden, gibt es aber auch Anwendungen, insbesondere die Beschreibung von Strategien, bei denen der Benutzer selbst eine Funktion durch einen Algorithmus formulieren möchte.

Eine Modellbeschreibungssprache muß daher auch Sprachkonstrukte anbieten, welche, vergleichbar einer höheren prozeduralen Sprache, die Formulierung von Algorithmen erlauben. Wie die Methodik der *strukturierten Programmierung* lehrt, sind hierfür Schleifenkonstruktionen und Verzweigungskonstruktionen ausreichend. Da aber die übrige Modellbeschreibung deklarativen Charakter hat, ist es nicht mit der Einführung zweier neuer Sprachkonstrukte getan.

Es stellt sich vielmehr das Problem, wie prozedurale Formulierungen in eine ansonsten deklarative Umgebung eingepaßt werden können. Hierfür gibt es prinzipiell zwei Mögichkeiten:

1. Definition eigenständiger Funktionen mit lokaler oder globaler Gültigkeit
2. Einbringung prozeduraler Abschnitte in die Modellbeschreibung

Die Definition einer eigenständigen Funktion bedeutet, daß

- ein geeigneter Mechanismus zur Parameterübergabe
- die Definition eigener Variablen
- Wertzuweisungen
- Schleifen und Verzweigungskonstruktionen

zur Verfügung gestellt werden müssen. Die Modellbeschreibung muß eine Deklaration der Funktion enthalten. Die Übergabeparameter sind auf ihre Kompatibilität zu prüfen.

Der Vorteil einer Funktionsdefinition liegt darin, daß die Funktion mehrfach eingesetzt werden kann und die Modellbeschreibung nicht unnötig aufgebläht wird und dadurch ihre Prägnanz verloren geht.

Der Vorteil prozeduraler Abschnitte ist darin zu sehen, daß problemlos auf alle Größen der Modellkomponente zugegriffen werden kann und diese nicht in einer Parameterliste übergeben werden müssen. Ist der Algorithmus nur sehr kurz, wird die Modellbeschreibung nicht unnötig verstreut.
Allerdings ist es erforderlich, daß der prozedurale Abschnitt durch geeignete Schlüsselwörter von der deklarativen Modellbeschreibung abgegrenzt wird.

Aus diesen Gründen erscheint die Einführung eigenständiger Funktionen in jedem Fall sinnvoll, die Einführung prozeduraler Abschnitte ist nur innerhalb von Ereignissen notwendig und ratsam.

## 3.5 Algorithmische Auswertung der Modellspezifikation

Die simulative Auswertung einer Modellspezifikation ist durch den Einsatz geeigneter physikalischer Komponenten oder die Abarbeitung eines geeigneten Algorithmus auf einer digitalen Rechenanlage möglich.

Während die Analogrechentechnik nur noch bei speziellen Anwendungen Verwendung findet, erfreut sich der Einsatz des Digitalrechners zunehmender Beliebtheit. Die wesentlichen Gründe hierfür sind:

- größere Rechengenauigkeit
- größerer Zahlenbereich
- einfachere Aufbereitung der Ergebnisse
- beliebige Operationen ausführbar, daher auch für Nichtlinearitäten geeignet
- keine Transformation des Zahlen- und Frequenzbereichs notwendig
- geringe Kosten

Der Analogrechner besitzt allerdings den Vorteil der höheren Rechengeschwindigkeit, da eine echte Parallelverarbeitung der einzelnen Komponenten gegeben ist. Aber auch in diesem Bereich ist man bestrebt, Analogrechner durch Multiprozessorsysteme zu verdrängen.

Wir entwicklen im folgenden einen Algorithmus für einen Monoprozessor, der geeignet ist, die in Abschnitt 3.2 vorgestellte Modellspezifikation auszuwerten und damit das dynamische Verhalten eines Modells zu simulieren.

### 3.5.1 Auswertung der Beziehungen für eine einzelne Modellkomponente

Wir nehmen an, wir haben eine einzelne Modellkomponente vorliegen, die durch $\mathbf{F}$ und $\mathbf{G}$ (mit Komponenten $f_i$ bzw. $g_i$) spezifiziert wurde. Der Anfangszustand $\mathbf{Z}(t_0) = \mathbf{Z}_0$ sowie der Verlauf der Eingangsfunktion $\mathbf{X}(t)$ seien vorgegeben.

Dann simuliert der folgende Algorithmus das dynamische Verhalten der Komponente sukzessive für jeden Zeitpunkt $t_k$ der Zeitmenge $\mathcal{T} = \{t_0, t_1, \ldots, t_N\}$:

1. Anfangszeit setzen: $k = 0 \quad \Rightarrow \quad t_k = t_0 \quad$ mit $k \in \mathcal{N}_0$
   Anfangszustand belegen: $\mathbf{Z}(t_0) = \mathbf{Z}_0$

2. statische Beziehungen für alle $\quad i \in \{1 \ldots N_Y\} \quad$ auswerten

   $$y_i(t_k) := g_i(\mathbf{X}(t_k), \mathbf{Z}(t_k), t_k)$$

3. dynamische Beziehungen für alle $\quad i \in \{1 \ldots N_Z\} \quad$ auswerten

   $$z_i(t_{k+1}) := f_i(\mathbf{X}(t_k), \mathbf{Z}(t_k), t_k)$$

4. Zeit fortschalten

   $$k := k + 1;$$

5. Mit Schritt 2 fortfahren

Die Spezifikation erfordert, daß die Zustandsvektoren $\mathbf{Z}(t_k)$ und $\mathbf{Z}(t_{k+1})$ in getrennten Speicherplätzen abgelegt werden. Zustandsgrößen, die für den Zeitpunkt $t_{k+1}$ errechnet wurden, wirken sich nicht auf den aktuellen Zustand $t_k$ aus. Daraus ergibt sich die Konsequenz, daß die Reihenfolge, in der die Zustandsgrößen ermittelt werden, keinen Einfluß auf das Ergebnis hat.

Für die statischen Beziehungen gilt das gleiche, solange das Modell nicht in mehrere Komponenten zerlegt wird. Dieser Fall wird im nächsten Abschnitt behandelt.

Die Tatsache, daß die Reihenfolge, in der die Modellgleichungen eines Typs ausgewertet werden, ohne Belang ist, erleichtert es ganz entscheidend, einer Modellbeschreibungssprache deklarativen Charakter zu verleihen. Wir werden daher versuchen, diese Eigenschaft auch für modular aufgebaute Modelle zu erhalten.

### 3.5.2 Auswertung der Beziehungen für zusammengesetzte Modelle

Bei zusammengesetzten Modellen sind die Modellgleichungen über mehrere Modellkomponenten verstreut und die Kopplung der Komponenten durch Gleichsetzen der Sensorgrößen mit Zustandsgrößen oder abhängigen Größen realisiert.

Wir wollen nun diskutieren, welche Auswirkungen auf die algorithmische Auswertung die Verknüpfung zweier Komponenten hat. Hierzu seien zwei Komponenten $S_1$ und $S_2$ gegeben, die beide je eine statische und dynamische Modellbeziehung aufweisen.

$$
\begin{array}{lrcl}
S_1\text{:} & z_1(t_{k+1}) & := & f_1(x_1(t_k), z_1(t_k)) \\
 & y_1(t_k) & := & g_1(x_1(t_k), z_1(t_k)) \\
 & & & \\
S_2\text{:} & z_2(t_{k+1}) & := & f_2(x_2(t_k), z_2(t_k)) \\
 & y_2(t_k) & := & g_2(x_2(t_k), z_2(t_k))
\end{array}
$$

Verknüpfen wir die beiden Komponenten über eine Verbindung $x_2 \equiv z_1$,


erhalten wir für das Gesamtmodell $S$ die folgenden Beziehungen:

$$
\begin{array}{lrcl}
S\text{:} & z_1(t_{k+1}) & := & f_1(x_1(t_k), z_1(t_k)) \\
 & z_2(t_{k+1}) & := & f_2(z_1(t_k), z_2(t_k)) \\
 & & & \\
 & y_1(t_k) & := & g_1(x_1(t_k), z_1(t_k)) \\
 & y_2(t_k) & := & g_2(z_1(t_k), z_2(t_k))
\end{array}
$$

In diesem Fall hat sich an der Unabhängigkeit des Ergebnisses von der Ausführungsreihenfolge nichts geändert.

Dies wird im folgenden Fall anders, wenn wir die beiden Komponenten durch die Verbindung $x_2 \equiv y_1$ miteinander verknüpfen.

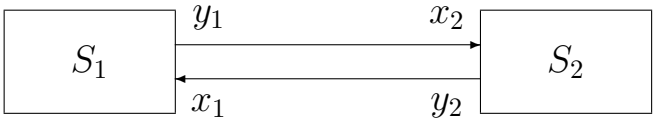
Für das Gesamtmodell $S$ erhalten wir nun die folgenden Beziehungen:

$$
\begin{array}{lrcl}
S\text{:} & z_1(t_{k+1}) & := & f_1(x_1(t_k), z_1(t_k)) \\
 & z_2(t_{k+1}) & := & f_2(y_1(t_k), z_2(t_k)) \\
 & & & \\
 & y_1(t_k) & := & g_1(x_1(t_k), z_1(t_k)) \\
 & y_2(t_k) & := & g_2(y_1(t_k), z_2(t_k))
\end{array}
$$

Die statischen Beziehungen dürfen nun nicht mehr in beliebiger Reihenfolge ausgewertet werden, sondern es muß zunächst die abhängige Größe $y_1$ und danach $y_2$ bestimmt werden. Für die Zustandsgrößen ist die Auswertungsreihenfolge weiterhin beliebig.

Auf Maßnahmen, die dieser Schwierigkeit begegnen, wird im Abschnitt 3.4.4 eingegangen. An dieser Stelle sei zunächst noch auf eine weitere Problematik zusammengesetzter Modelle eingegangen.

Hierzu verknüpfen wir die beiden Komponenten $S_1$ und $S_2$ durch zwei Verbindungen $x_2 \equiv y_1$ und $x_1 \equiv y_2$.



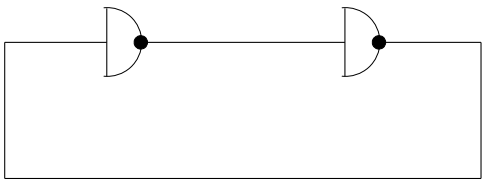

Für das Gesamtmodell $S$ ergeben sich die Beziehungen:

$$
\begin{array}{lrcl}
S\text{:} & z_1(t_{k+1}) & := & f_1(y_2(t_k), z_1(t_k)) \\
 & z_2(t_{k+1}) & := & f_2(y_1(t_k), z_2(t_k)) \\
 & & & \\
 & y_1(t_k) & := & g_1(y_2(t_k), z_1(t_k)) \\
 & y_2(t_k) & := & g_2(y_1(t_k), z_2(t_k))
\end{array}
$$

In diesem Fall läßt sich zur Ermittlung der statischen Beziehungen überhaupt keine geeignete Reihenfolge angeben. Zur Berechnung von $y_1(t_k)$ benötigt man bereits $y_2(t_k)$ und umgekehrt.

Die statischen Beziehungen des Gesamtmodells $S$ beschreiben kein definiertes dynamisches Verhalten mehr. Deshalb resultiert dieses Problem nicht aus der algorithmischen Auswertung, sondern tritt vielmehr auch bei physikalischen Komponenten auf.

Ein einfaches System, das im Falle einer solchen Zusammenschaltung kein aus den Modellgleichungen ableitbares Verhalten zeigt, sondern wilde Schwingungen vollführt, sind zwei rückgekoppelte NOR-Gatter.



Die Ursache für dieses undefinierte Verhalten ist nicht etwa nur in der Rückführung der Verbindungen zu sehen, sondern auch in der Gestalt der Funktionen $g_1$ und $g_2$. Hängt nur eine der beiden Funktionen nicht von $x_1$ bzw. $x_2$ ab, ist der Kreislauf durchbrochen, und es resultiert ein Modellverhalten gemäß der Spezifikation.

Während die Spezifikation einer einzelnen Modellkomponente stets ein wohldefiniertes Verhalten beschreibt, kann eine ungünstige Verknüpfung zweier oder mehrerer Komponenten bewirken, daß die statischen Beziehungen kein stationäres Verhalten aufweisen.

Allgemein gesehen gilt:

Eine statische Beziehung einer einzelnen Modellkomponente $j$

$$y_i^{(j)}(t_k) \;:=\; g_i^{(j)}(\mathbf{X}^{(j)}(t_k), \mathbf{Z}^{(j)}(t_k), t_k) \tag{3.5.2-1}$$

geht in die Form

$$y_i^{(j)}(t_k) \;:=\; g_i^{(j)}(\mathbf{Y}(t_k), \mathbf{Z}(t_k), t_k) \tag{3.5.2-2}$$

über, wenn die Sensorgrößen an Zustandsgrößen bzw. abhängige Größen angeschlossen werden.

Bei der Auswertung statischer Beziehungen muß demnach berücksichtigt werden, daß diese selbst auf abhänigen Größen beruhen können. Das ist der Grund, warum wir für statische Beziehungen prinzipiell auch Abhängigkeiten von anderen abhängigen Größen zulassen können (siehe 3.2.2-4).

Die Beziehung (3.5.2-2) kann nun in drei verschiedenen Varianten auftreten.

1. $y_i^{(j)}(t_k) := g_i^{(j)}(y_1, y_2, \ldots, y_{i-1}, \mathbf{Z}, t_k)$

   Alle benötigten $y_1, \ldots, y_{i-1}$ können vorher berechnet werden. Die Gleichung ist daher stabil.

2. $y_i^{(j)}(t_k) := g_i^{(j)}(y_1, y_2, \ldots, y_i, \mathbf{Z}, t_k)$

3. $y_i^{(j)}(t_k) := g_i^{(j)}(y_1, y_2, \ldots, y_{N_Y}, \mathbf{Z}, t_k)$

   Zur Berechnung von $y_i^{(j)}$ ist die Größe $y_i$ selbst bzw. weitere Größen $y_{i+1}, \ldots, y_{N_Y}$ notwendig, deren Werte noch nicht bekannt sind. Hierbei handelt es sich um eine implizite algebraische Gleichung bzw. ein implizites algebraisches Gleichungssystem. Ob sich durch Iteration eine stabile Lösung finden läßt, hängt von der Konvergenzeigenschaft des Gleichungssystems ab.

Aus den Betrachtungen dieses Abschnitts ergibt sich:
Die Auswertung der Modellbeziehungen bei zusammengesetzten Modellen kann nach dem gleichen Algorithmus erfolgen wie für eine einzelne Komponente mit dem Unterschied, daß für die Auswertung der statischen Beziehungen besondere Vorkehrungen zu treffen sind.

### 3.5.3 Auswertung statischer Beziehungen

Wir untersuchen nun Möglichkeiten, statische Beziehungen der Form

$$y_i(t_k) := g_i(\mathbf{Y}(t_k), \mathbf{Z}(t_k), t_k)$$

so auszuwerten, daß sie in beliebiger Reihenfolge aufteten dürfen und die Verbindungen zwischen Modellkomponenten beliebig angelegt werden können.

Es bieten sich zwei grundsätzlich verschiedene Verfahren an:

1. *Sortieren der statischen Beziehungen*
   Die Verbindungen zwischen den Modellkomponenten werden aufgelöst, alle im gesamten Modell vorkommenden statischen Beziehungen zu einem Block zusammengelegt und entsprechend ihren Abhängigkeiten sortiert. Dieses Verfahren arbeitet zur Laufzeit ohne Effizienzverlust, verhindert aber, daß die Modellkomponenten getrennt übersetzt werden können.

2. *Iterative Auswertung*
   Man beläßt die vorgefundene Reihenfolge der Beziehungen oder sortiert nur innerhalb einer Modellkomponente. Zur Ausführungszeit wiederholt man die Berechnung sämtlicher statischer Beziehungen solange, bis keine von den abhängigen Größen mehr ihren Wert verändert.

   Dieses Verfahren ermöglicht eine getrennte Übersetzung der Modellkomponenten und ist äußerst einfach zu implementieren, kostet aber sehr viel Rechenzeit. Selbst im günstigsten Fall, sind wenigstens zwei Durchläufe erforderlich. Einer um die neuen Größen zu ermitteln und ein zweiter um festzustellen, daß keine weiteren nderungen stattfanden.

Beide Verfahren können noch erheblich beschleunigt werden, wenn man nur diejenigen Beziehungen auswertet, von denen sich eines ihrer Argumente verändert hat.

Zu diesem Zweck wird bei der Übersetzung des Modells ein Abhängigkeitsgraph aufgestellt und im Simulationsprogramm abgelegt. Verändert beim Ablauf der Simulation irgendeine Modellgröße ihren Wert, dann wird dem Abhängigkeitsgraphen entnommen, welche Beziehungen davon betroffen sind und diese markiert. Es werden dann immer nur markierte Beziehungen ausgewertet.

Wendet man diese Methode zur Beschleunigung an, dann dürfte das zweite Verfahren schlimmstenfalls um einen Faktor zwei langsamer sein als das erste. Aus Gründen der Modularität des generierten Simulationsprogramms haben wir uns daher für das zweite Verfahren entschieden.

Dieses Verfahren liefert obendrein einen angenehmen Nebeneffekt. Wegen der iterativen Abarbeitung ermöglicht es die Formulierung impliziter algebraischer Gleichungen als Iterationsformel.

Besitzen die abhängigen Variablen die anfängliche Belegung

$$\mathbf{Y}(t_k, 0) \qquad \text{bzw.} \qquad y_i(t_k, 0) \quad \forall \quad i$$

dann wird die statische Beziehung

$$y_i(t_k, l+1) := g_i(\mathbf{Y}(t_k, l), \mathbf{Z}(t_k), t_k)$$

solange iteriert, bis die Bedingung

$$\mid y_i(t_k, l+1) - y_i(t_k, l) \mid \quad < \quad \rho \cdot \mid y_i(t_k, l+1) \mid + \varepsilon$$

für vorgegebene $\rho$ und $\varepsilon$ erfüllt ist.

Es liegt in der Verantwortung des Benutzers, dafür zu sorgen, daß die Iteration abbricht.

**Beispiel:**

aktuelle Zustandsgrößen: $z_1 = 2 \qquad z_2 = 5$
aktuelle abh. Größen: $y_1 = 2 \qquad y_2 = 10 \qquad y_3 = 12$

Statische Beziehungen in angegebener Reihenfolge:

$$y_2 := y_1 \cdot z_2$$
$$y_1 := z_1 + 1$$
$$y_3 := y_1 + y_2$$

1. *Auswertung nach Sortieren*

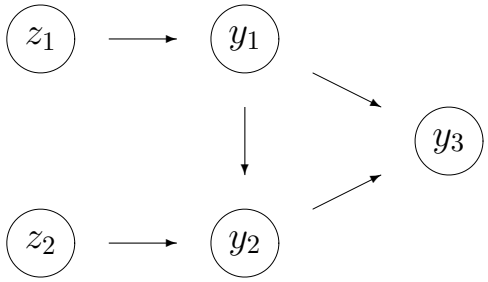
$$\begin{array}{lcl} y_1 := z_1 + 1 & & y_1 := 3 \\ y_2 := y_1 \cdot z_2 & \longrightarrow & y_2 := 15 \\ y_3 := y_1 + y_2 & & y_3 := 18 \end{array}$$

2. *Iterative Auswertung*

| | 1. Schritt | 2. Schritt | 3. Schritt |
|---|---|---|---|
| $y_2 := y_1 \cdot z_2$ | $y_2 := 10$ | $y_2 := 15$ | $y_2 := 15$ |
| $y_1 := z_1 + 1$ | $y_1 := 3$ | $y_1 := 3$ | $y_1 := 3$ |
| $y_3 := y_1 + y_2$ | $y_3 := 13$ | $y_3 := 18$ | $y_3 := 18$ |

Der zweite und dritte Schritt liefert für alle abhängigen Größen das gleiche Ergebnis. Damit ist die Iteration beendet.

3. *Aufstellen eines Abhängigkeitsgraphen und Auswertung in vorgegebener Reihenfolge*




Beim vorausgegangenen Zustandsübergang wurde nur $z_1$ verändert. Die erste Markierung (1) wird daher bei $y_1$ angebracht.

1. Zyklus:

| | Markierung | Ergebnis | |
|---|---|---|---|
| $y_2 := y_1 \cdot z_2$ | | | |
| $y_1 := z_1 + 1$ | (1) | $y_1 := 3$ | $\rightarrow$ setze Marke (2), (3) |
| $y_3 := y_1 + y_2$ | (2) | $y_3 := 13$ | |

2. Zyklus:

| | | | |
|---|---|---|---|
| $y_2 := y_1 \cdot z_2$ | (3) | $y_2 := 15$ | $\rightarrow$ setze Marke (4) |
| $y_1 := z_1 + 1$ | | | |
| $y_3 := y_1 + y_2$ | (4) | $y_3 := 18$ | |

Da nach der letzten Berechnung von $y_3$ keine offenen Abhängigkeiten mehr bestehen, kann die Iteration beendet werden. In diesem Beispiel konnte der Aufwand von 9 ausgewerteten Gleichungen auf 4 reduziert werden.

## 3.6 Numerische Behandlung der Modellspezifikation

Die Definition der Zustandsübergänge für die Zustandsgrößen eines Modells setzt als Zeitmenge die Menge der reellen Zahlen und infinitesimal kleine Zeitschritte voraus.

Die hier angestrebte Simulation eines mathematisch-formalen Modells auf einem Digitalrechner muß jedoch mit dessen Beschränkungen vorlieb nehmen. Zahlenwerte sind nur mit einer endlichen Stellenzahl darstellbar und die mögliche Anzahl von ausführbaren Zustandsübergängen wird durch die hierfür erforderliche Rechenzeit begrenzt.

Wir müssen daher nach einer Implementierung des Modells suchen, die ein Modellverhalten gewährleistet, das dem theoretischen Verhalten der vorgegebenen Spezifikation hinreichend gut angenähert ist und in einer vertretbaren Ausführungszeit abläuft.

### 3.6.1 Die Darstellung der Zeitmenge

Zur Darstellung reeller Zahlen stellt die Hardware von Digitalrechnern normalerweise das Datenformat *Gleitkommazahl* zur Verfügung. Dieses Datenformat stellt eine Zahl $a$ durch das Produkt einer Mantisse $m$ (mit fester Stellenzahl $N_m$) und einem Skalierungsfaktor mit einem Exponenten $e$ (mit fester Stellenzahl $N_e$) zur Potenz 2 (gelegentlich auch $2^3$ oder $2^4$) dar.

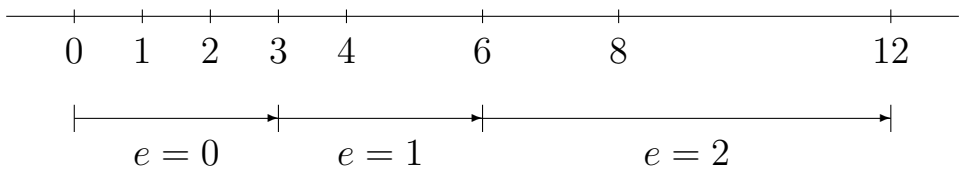
$$a = m \cdot 2^e$$

Die Menge aller in diesem Format darstellbaren Zahlen wollen wir mit $\mathcal{F}$ (floating point numbers) bezeichnen.

Dieses Datenformat liefert stets die gleiche relative Darstellungsgenauigkeit von $N_m$ Binärstellen. Die absolute Darstellungsgenauigkeit, d. h. die kleinste darstellbare Differenz zweier benachbarter Zahlen, nimmt jedoch mit wachsenden Exponenten ab.

**Beispiel:**

$$N_m = 2, \qquad m \in \{0,1,2,3\}, \qquad e \in \{0,1,2\}$$




Dieses Datenformat liefert keine geeignete Zeitmenge. Das ungleichmäßige Auflösungsvermögen bewirkt, daß sich das Verhalten des simulierten Modells mit wachsender Simulationszeit verändert. Auch hat der Benutzer keine Möglichkeit, das Auflösungsvermögen zu beeinflussen und dadurch ganz gezielt Probleme der Gleichzeitigkeit zu untersuchen.

Wir wählen daher als Zeitmenge $\mathcal{T}$ das kartesische Produkt zweier Mengen $\mathcal{T}_C$ und $\mathcal{T}_D$ und stellen einen Zeitpunkt $t_k$ als Zahlenpaar $(T_k, J_k)$ dar.

$$\mathcal{T} = \mathcal{T}_C \times \mathcal{T}_D$$

mit

$$\mathcal{T}_C = \{T \mid T = k \cdot R, \quad k \in Z, \quad R \in \mathcal{F}\} \tag{3.6.1-1}$$

$$\mathcal{T}_D = \{J \mid J \in N_0\} \tag{3.6.1-2}$$

Die Menge $\mathcal{T}_C$ unterscheidet die Zustände, die durch kontinuierliche Zustandsübergänge erzeugt werden und gibt einen realen Zeitfortschritt an. Ihre einzelnen Zeitpunkte $T$ sind ganzzahlige Vielfache des Auflösungsvermögens $R$ (resolution).

Die Menge $\mathcal{T}_D$ unterscheidet die Zustände, die durch diskrete Zustandsübergänge erzeugt werden, ohne daß ein realer Zeitfortschritt stattfindet. Ihre Werte werden fortlaufend durchnumeriert.

Der Index $k$ unterscheidet wie bisher einzelne Zustände. Nach jedem Zustandsübergang wird $k$ um 1 erhöht. Entsprechend (3.2.3-5) werden wir für die numerische Behandlung entweder einen kontinuierlichen oder einen diskreten Zustandsübergang durchführen.

Für die Zeitfortschaltung eines diskreten Zustandsübergangs ergibt sich nun

$$\begin{aligned} T_{k+1} &= T_k \\ J_{k+1} &= J_k + 1 \end{aligned}$$

und bei einem kontinuierlichen Zustandsübergang gilt:

$$\begin{aligned} T_{k+1} &= T_k + R \\ J_{k+1} &= J_k \quad . \end{aligned}$$

Unter Zugrundelegung der zweidimensionalen Zeitmenge ergibt sich für einen diskreten Zustandsübergang

$$\mathbf{Z}(T_{k+1}, J_{k+1}) := \begin{cases} \hat{\mathbf{F}}(\mathbf{W}(T_k, J_k)) & \text{falls} \quad \mathbf{W}(T_k, J_k) \in \mathcal{A} \\ \mathbf{Z}(T_k, J_k) & \text{sonst} \end{cases} \tag{3.6.1-3}$$

und für einen kontinuierlichen Zustandsübergang

$$\mathbf{Z}(T_{k+1}, J_{k+1}) := \mathbf{Z}(T_k, J_k) + \int_{T_k}^{T_k+R} \mathbf{F}'(W(\tau, J_k))d\tau \qquad \text{mit} \quad \tau \in \mathcal{F} \quad . \tag{3.6.1-4}$$

Der infinitesimale Schritt $\Delta t \to 0$ wird demnach beim diskreten Zustandsübergang auf den gleichen, realen Zeitpunkt projeziert und beim kontinuierlichen Zustandsübergang durch ein Integral ersetzt. Durch ein geeignetes Verfahren der numerischen Integration ist das Intervall $[T_k, T_k + R]$ zu überbrücken. Dazu benötigt man auch Werte der Funktion $\mathbf{F}'$ zu Zeitpunkten $\tau$, die im Inneren des Intervalls liegen und nicht der Zeitmenge $\mathcal{T}_C$ angehören. Hierauf wird an späterer Stelle eingegangen.

Trägt man die Zeitmenge $\mathcal{T}$ in einem Diagramm zweidimensional auf und markiert darin diejenigen Zeitpunkte $t_k = (T_k, J_k)$, zu denen ein Simulationslauf Zustände $\mathbf{Z}(t_k)$ erzeugt hat, erkennt man gut die Abfolge kontinuierlicher und diskreter Zustandsübergänge. Wir wollen dieses Diagramm daher Zustandsfolgediagramm nennen.

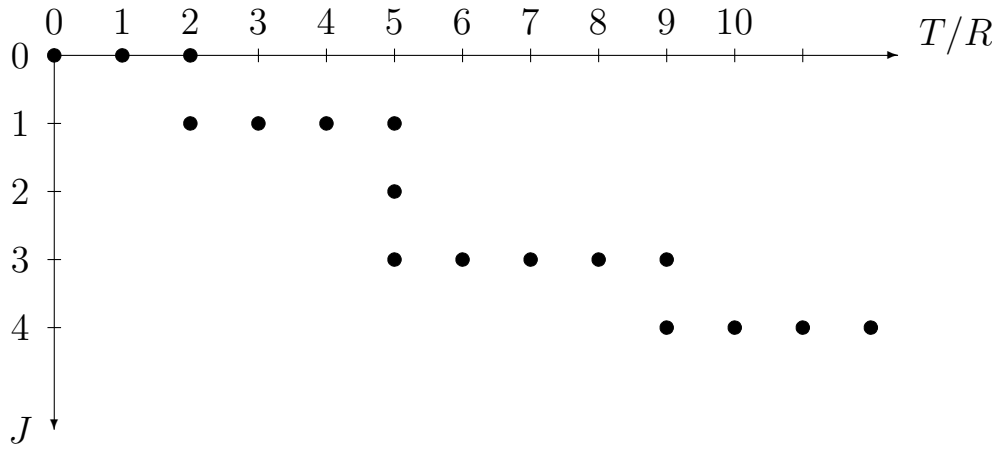



Abb. 3.5.1-1: Beispiel eines Zustandsfolgediagramms

### 3.6.2 Auswertung kombinierter Zustandsübergänge

Ereignisse und Differentialgleichungen werden unter Zugrundelegung der zweidimensionalen Zeitmenge wechselweise ausgewertet.

Es werden zu einem Zeitpunkt $T$ solange Ereignisse ausgeführt, bis keine Auslösebedingung mehr erfüllt ist. Dann wird durch Integration zum nächsten Zeitpunkt $T + R$ übergegangen.

Für eine Simulation im Zeitintervall $[T_{Beg}, T_{End}]$ läßt sich der folgende Algorithmus angeben:

A1) Anfangszeit setzen: $k = 0, \quad T_k = T_{Beg}, \quad J_k = 0$

Anfangszustand belegen: $\mathbf{Z}(T_{Beg}, 0) := \mathbf{Z}_0$

A2) Statische Beziehungen für $i \in \{1 \ldots N_Y\}$ auswerten:

$y_i(T_k, J_k) := g_i(\mathbf{W}(T_k, J_k))$

A3) Prüfen, ob aktueller (erweiterter) Zustand Ereignisse bedingt:

$$\mathbf{W}(T_k, J_k) \in \mathcal{A} \qquad ?$$

Falls ja $\rightarrow$ A4
Falls nein $\rightarrow$ A5

A4) Ereignisse für $i \in \{1 \ldots N_Z\}$ ausführen:

$$z_i(T_{k+1}, J_{k+1}) \;=\; \begin{cases} \hat{f}_i(\mathbf{W}(T_k, J_k)) & \text{falls} \quad \mathbf{W}(T_k, J_k) \in \mathcal{A}_i \\ z_i(T_k, J_k) & \text{sonst} \end{cases}$$

Zeit fortschalten:

$$T_{k+1} = T_k, \qquad J_{k+1} = J_k + 1$$
$$k \leftarrow k + 1$$

Mit A2 fortfahren

A5) Prüfen, ob Simulationsende erreicht wurde:

$$T_k = T_{End} \qquad ?$$

Falls ja $\rightarrow$ Endezustand erreicht, Simulation beenden

A6) Integration für $i \in \{1 \ldots N_Z\}$ ausführen:

$$z_i(T_{k+1}, J_{k+1}) \;:=\; z_i(T_k, J_k) + \int_{T_k}^{T_k+R} f'(\mathbf{W}(\tau, J_k))d\tau$$

Zeit fortschalten:

$$T_{k+1} = T_k + R, \qquad J_{k+1} = J_k$$
$$k \leftarrow k + 1$$

A7) Mit A2 fortfahren

*Anmerkung:* Führt das Integrationsverfahren Zwischenschritte aus, dann sind nach jedem Zwischenschritt die abhängigen Größen neu zu bestimmen.

### 3.6.3 Numerische Ungenauigkeiten bei der Auswertung zeitkontinuierlicher Zustandsübergänge

Für die folgenden Betrachtungen spielt die diskrete Zeit $J_k$ meist keine Rolle. Der Übersichlichkeit halber lassen wir in solchen Fällen die diskrete Zeit wegfallen.

Die Auswertung eines kontinuierlichen Zustandsübergangs

$$z_i(T_k + R) = z_i(T_k) + \int_{T_k}^{T_k+R} f_i^{'}(\mathbf{W}(\tau))d\tau \qquad \text{mit} \quad \tau \in \mathcal{F}$$

erfordert die numerische Lösung eines Integrals.

Ein numerisches Integrationsverfahren liefert jedoch nur einen Schätzwert $\tilde{z}_i(T_{k+1})$ für $z_i(T_{k+1})$. Stimmen $\tilde{z}_i(T_k)$ und $z_i(T_k)$ zum Zeitpunkt $T_k$ noch überein, bezeichnet man die betragsmäßige Abweichung

$$\alpha_i(T_{k+1}) = \mid \tilde{z}_i(T_{k+1}) - z_i(T_{k+1}) \mid$$

als absoluten, lokalen Fehler im Integrationsschritt $k + 1$.

Von größerer Bedeutung ist der relative, lokale Fehler

$$\rho_i(T_{k+1}) = \frac{\alpha_i(T_{k+1})}{z_i(T_{k+1})}.$$

Dieser wird gerne als Gütekriterium für die Integration herangezogen. Liegt der relative, lokale Fehler während eines ganzen Simulationslaufs unter einer vorgegebenen Schranke $\delta_i$, dann schließt man daraus auf die Güte des erzielten globalen Verlaufs der Größe $z_i$. Dieser Vorgehensweise liegen rein heuristische Erfahrungen zugrunde, die wir an dieser Stelle akzeptieren wollen. Näheres hierüber findet sich in der Literatur, z.B. bei [WerArn 86, ConBo 72].

Für den lokalen Fehler sind verantwortlich:

- das Verfahren zur numerischen Integration
- die Konsistenz- bzw. Konvergenzordnung des Verfahrens
- die Integrationsschrittweite $T_{k+1} - T_k$

Für ein bestimmtes Verfahren gilt: Je höher die Ordnung des Verfahrens und je kleiner die Schrittweite, desto kleiner wird der lokale relative Fehler. Die Tatsache, daß bei extrem kleinen Schrittweiten Rundungsfehler eine Rolle spielen, soll hier nicht betrachtet werden.

Gibt man sich einen bestimmten lokalen relativen Fehler vor, so muß man abwägen zwischen der Ordnung des Verfahrens und der Integrationsschrittweite. Mit wachsender Ordnung steigt der Rechenaufwand für einen Integrationsschritt, dafür nimmt die Anzahl der Schritte ab.

Es hat sich herausgestellt, daß Verfahren der Ordnung 4 - 6 meist recht gute Ergebnisse liefern. Diese Verfahren beschaffen sich jedoch auch Stützstellen, die innerhalb des Intervalls $[T_k, T_{k+1}]$, daß heißt zwischen den Zeitpunkten $T = k \cdot R$ und $T = (k+1) \cdot R$ liegen. Zu diesem Zweck – und nur hierfür – ist es notwendig, die Zeitmenge $\mathcal{T}_C$ durch Werte aus der Menge der Gleitkommazahlen $\mathcal{F}$ zu ergänzen.

Damit dies möglich ist, muß sichergestellt sein, daß $\mathcal{F}$ die Zeitmenge $\mathcal{T}_C$ noch genügend fein unterteilt. Dies läßt sich gewährleisten, wenn man $k$ aus der Menge der 32-Bit-Integerzahlen nimmt und für die Darstellung der Menge $\mathcal{F}$ eine Mantisse von 48 Bit wählt. Auf UNIX-Maschinen, läßt sich dies durch geeignete Wahl der Datentypen (meist: *long* für $k$, *double* für $T$ und $R$) erreichen.

Die Wahl des Auflösungsvermögens $R$ richtet sich nach zwei Gesichtspunkten:

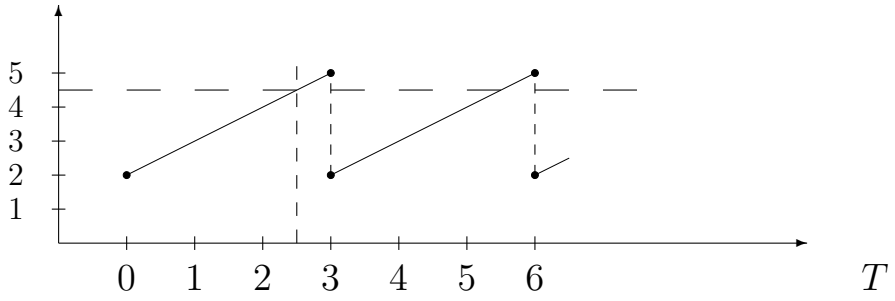
1. Das Integrationsverfahren muß bei Schrittweite $R$ zu jedem Zeitpunkt $T_k$ noch die vorgegebene Schranke für den lokalen, relativen Fehler einhalten können.
2. Der Zeitverzug, der im Wechselspiel mit diskreten Zustandsübergängen oder an Unstetigkeitsstellen der Ableitungen auftritt, darf sich nicht nachteilig auf den weiteren Simulationslauf auswirken.

Der zweite Aspekt soll an einem Beispiel erläutert werden:

Modellgleichungen: $z' := 0.5;$ $\hat{z} := 1$ immer wenn $z \geq 4.5$

Anfangszustand: $z(0) := 2;$

Auflösungsvermögen: $R = 1$




Die Simulation liefert eine Abweichung vom spezifizierten Modellverhalten. Erst bei Erreichen des Zustands $\mathbf{Z}(T = 3)$ wird der diskrete Zustandsübergang ausgeführt. Die Größe $z$ überschreitet deshalb die vorgegebene Schranke von 4.5 um 0.5. Dies bewirkt mit fortschreitender Zeit eine zunehmende Verschiebung zum spezifizierten Verlauf der Größe $z$. Durch ein besseres Auflösungsvermögen, d. h. eine Verkleinerung von $R$, können diese Fehler reduziert werden.

Die Wahl des Auflösungsvermögens $R$ orientiert sich also letztlich an der Genauigkeit der Angaben zu den zeitlichen Vorgängen im Modell.

### 3.6.4 Differentialquotienten mit Unstetigkeitsstellen

Ein besonderes Problem stellen Differentialquotienten mit Unstetigkeitsstellen dar. Ein Integrationsschritt, der über eine Unstetigkeitsstelle reicht, produziert einen ungewöhnlich großen Fehler. Alle Integrationsverfahren höherer Ordnung erwarten einen stetigen Verlauf der Ableitungen.

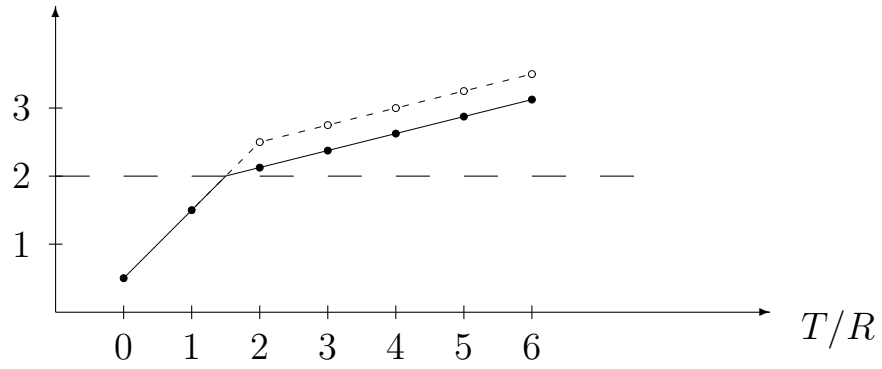
**Beispiel:**

Modellgleichung:

$$z' := \begin{cases} +1 & \text{für} \quad z < 2 \\ +0.25 & \text{für} \quad z \geq 2 \end{cases}$$

Anfangszustand: $z(0) = 0.5$

Auflösungsvermögen: $R = 1$




Die durchgezogene Linie zeigt den theoretischen Verlauf der Größe $z$. Der korrekte Wert für den Zeitpunkt $T = 2$ wäre $z(2) = 2.125$. Bei Anwendung verschiedener Integrationsverfahren erhält man Werte wie

2.5, 2.125, 2.248, 2.5, 2.136, 2.261.

Die auftretenden Fehler sind so beträchtlich, daß eine Fortsetzung der Integration nicht sinnvoll erscheint. Ein genaueres Ergebnis kann nur durch eine Verminderung der Integrationsschrittweite, d.h. durch ein besseres Auflösungsvermögen $R$ erzielt werden.

Es fällt auf, daß auch oder gerade Verfahren höherer Ordnung einen ungewohnt großen Integrationsfehler verursachen. Die Ursache ist darin zu sehen, daß sich Verfahren höherer Ordnung mehrere Stützwerte im gesamten Integrationsintervall beschaffen. Ein Teil der Stützstellen befindet sich damit auf der einen Seite der Unstetigkeitsstelle, ein anderer Teil auf der anderen Seite. Je nach der Lage der Unstetigkeitsstelle zu diesen Stützstellen fällt der Integrationsfehler größer oder kleiner aus.

Wegen des ungewissen Ergebnisses an Unstetigkeitsstellen setzen wir hier das Eulersche Polygonzugverfahren ein, das mit einem einzigen Extrapolationsschritt einen Wert auf der anderen Seite der Unstetigkeitsstelle liefert. Es berechnet den Zustand $z(T + R)$ alleine aus $z(T)$:

$$z(T + R) := z(T) + R \cdot z'(T)$$

Die weiteren Integrationsschritte können wiederum mit jedem anderen Verfahren ausgeführt werden. Die gestrichtelte Linie zeigt den neuen Verlauf der Größe $z$. Wir erhalten für den Zeitpunkt 2 den Wert $z(2) = 2.5$. Durch ein höheres Auflösungsvermögen läßt sich die Abweichung vom theoretischen Wert beliebig verbessern.

Gelegentlich findet man eine andere Lösung des Problems vor, die jedoch nicht korrekt ist. Hierbei wird das Integrationsverfahren beibehalten und die Umschaltung von einer Ableitungsfunktion auf eine andere erst dann vorgenommen, wenn ein neuer Zeitpunkt erreicht ist. Diese Möglichkeit scheidet aber aus, da dabei eventuell Argumente von Standardfunktionen außerhalb ihres zulässigen Definitionsbereichs zu liegen kommen. Dies zeigt das folgende Beispiel:

$$z' := \begin{cases} \sqrt{x} & \text{für } x > 0 \\ \sqrt{-x} & \text{für } x < 0 \end{cases}$$

Wechselt $x$ innerhalb eines Integrationsschritts das Vorzeichen, wird das Argument unter der Wurzel negativ.

Unstetigkeitsstellen erfordern daher die folgende Vorgehensweise:
Wird während eines Integrationsschritts eine Unstetigkeitsstelle erkannt, muß der Integrationsschritt unter Verwendung des Euler-Verfahrens wiederholt werden.

### 3.6.5 Integration mit vergrößerter Schrittweite

Wir sind bisher immer davon ausgegangen, daß die Integrationsschrittweite dem Auflösungsvermögen $R$ entspricht. Da sich aber die Wahl von $R$ nicht nur nach der gesetzten Schranke für den Integrationsfehler richtet, ist häufig auch eine Schrittweite möglich, die ein ganzzahliges Vielfaches $l$ von $R$ beträgt. Wir führen dann den Zustandsübergang

$$z_i(T_k + l \cdot R) \;:=\; z_i(T_k) + \int_{T_k}^{T_k+l\cdot R} f_i^{'}(\mathbf{W}(\tau))d\tau \quad \text{mit} \quad \tau \in \mathcal{F}$$

durch.

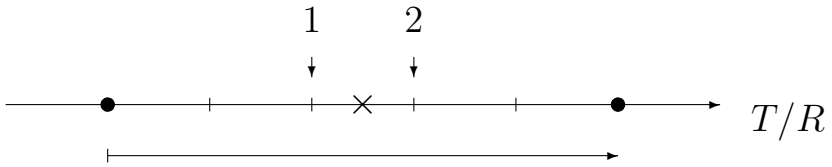
Die dazwischenliegenden Zustände $z_i(T_{k+1}) \ldots z_i(T_{k+l-1})$ werden bei dieser Vorgehensweise nicht mehr erzeugt.

Dies wirft Probleme auf, wenn zwischenzeitlich Differentialquotienten Unstetigkeitsstellen aufweisen oder Ereignisbedingungen wahr werden.

Damit das vorgegebene Auflösungsvermögen eingehalten wird, muß in einem solchen Fall der Integrationsschritt reduziert werden und zwar

1. im Falle einer unstetigen Ableitung bis zu dem Zeitpunkt, zu dem noch keine Unstetigkeit vorliegt. Anschließend kann ein Integrationsschritt nach Euler mit Schrittweite $R$ ausgeführt werden, um die Unstetigkeitsstelle zu überwinden.

2. bei Bedingungen, die Ereignisse auslösen, bis zu dem Zeitpunkt, zu dem die Bedingung erstmalig wahr wird. Nun kann das Ereignis ausgeführt werden.



1. Integrationsschritt

Das Kreuz markiert den Bereich zwischen den beiden Zuständen, in dem die Bedingung wahr wird.

Wir müssen daher in der Lage sein, unstetige Ableitungen und eintretende Ereignisbedingungen während eines Integrationsschritts zu erkennen und die Integrationsschrittweite zu vermindern.

Ein weiteres Problem tritt dadurch auf, daß der Integrationsfehler mit wachsender Schrittweite sehr stark zunimmt. Es ist daher erforderlich, den Fehler zu schätzen und die Schrittweite so zu steuern, daß der Fehler unterhalb einer gegebenen Schranke bleibt.

### 3.6.6 Reduzierung des Integrationsschritts auf stetigen Bereich

Im Kapitel 6 wird beschrieben, wie das Verlassen des stetigen Bereichs erkannt wird. Wir gehen davon aus, daß uns diese Information vorliegt. Die einfachste, aber auch sicherste Art den Zeitpunkt zu finden, an dem der stetige Bereich endet, ist das Intervallhalbierungsverfahren. Die Unstetigkeitsstelle wird hierbei in ein Intervall eingeschlossen, das solange halbiert wird, bis sich die Intervallgrenzen nur noch um das Auflösungsvermögen unterscheiden.

Ist die Anfangsschrittweite $l_0 = l$, dann gilt für die weiteren Suchschritte die Schrittweiten:

$$l_{i+1} = \begin{cases} l_i + \Delta l_i & \text{falls stetig} \\ l_i - \Delta l_i & \text{falls unstetig} \end{cases} \qquad \text{mit} \quad \Delta l_i = \frac{l}{2^i}, \quad \Delta l_i \geq 1$$

### 3.6.7 Reduzierung des Integrationsschritts bis Ereignis eintritt

Das Eintreten eines Ereignisses wird durch die Ereignisbedingung festgestellt. Ereignisbedingungen enthalten in der Regel Vergleichsausdrücke der Form

$$x(T) \quad \textit{comp\_op} \quad y(T)$$

mit dem Vergleichsoperator *comp_op*. Besitzt wenigstens eine von beiden Seiten einen kontinuierlichen Verlauf, kann diese Bedingung während der Ausführung eines Integrationsschritts wechseln. Dieser Wechsel muß angezeigt werden, und die Schrittweite ist bis zu dem Zeitpunkt zu reduzieren, zu dem die Bedingung erstmalig wahr wird.

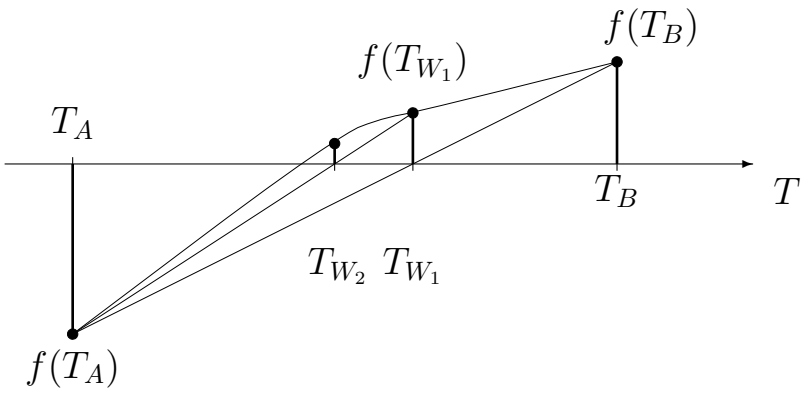
Dieses Problem ist gleichbedeutend mit einer Nullstellensuche

$$f(T) = x(T) - y(T) = 0$$

für das die numerische Mathematik zahlreiche Verfahren zur Verfügung stellt [ConBo 72].

Am geeignetsten erscheint uns die *modifizierte regula falsi*, da sie das Suchintervall um die Nullstelle $[T_A, T_B]$ von beiden Seiten eingrenzt, sehr effizient arbeitet und leicht zu implementieren ist.

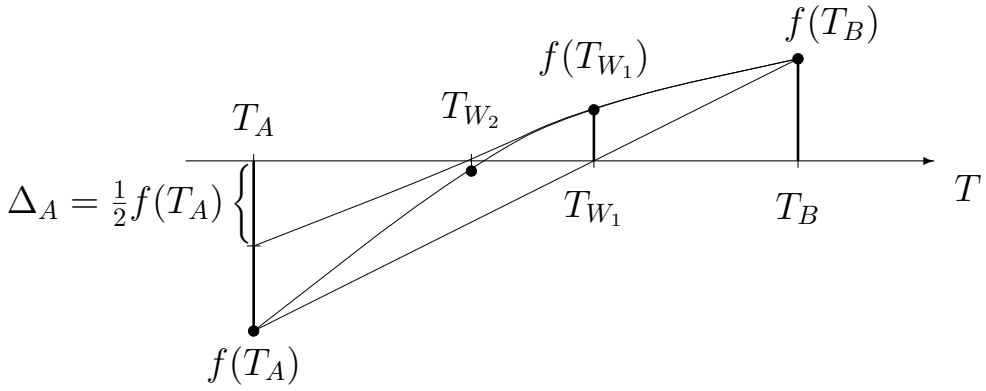
Das Prinzip der *regula falsi* ist es, die beiden Punkte $f(T_A)$ und $f(T_B)$ durch eine Sekante zu verbinden und ihren Schnittpunkt mit der Nullinie $T_W$ zu bestimmen. Zu diesem Zeitpunkt wird der Funktionswert $f(T_W)$ erneut bestimmt und geprüft, ob $f(T_W)$ auf der Seite von $T_A$ oder $T_B$ liegt. Das Suchintervall wird dann so verkleinert, daß es wiederum die Nullstellen einschließt.




Setzt man das Verfahren in gleicher Weise mit dem verkleinerten Intervall fort, nähert man sich der Nullstelle von einer Seite her an.

Durch eine einfache Modifikation läßt sich die Nullstelle von beiden Seiten einschließen. Das Suchintervall wird dadurch ebenfalls reduziert, und somit die Konvergenzgeschwindigkeit erhöht.

Stellt man fest, daß $T_W$ bereits beim letzten Suchschritt auf der gleichen Seite lag, versucht man durch eine Halbierung der Senkrechten bei $T_A$ bzw. $T_B$ die Nullstelle einzuschließen.




Konnte die Nullstelle eingeschlossen werden, wird wieder wie eingangs verfahren. Andernfalls wird die betreffende Senkrechte nochmals halbiert.

In einer formalen Darstellung besitzt der Algorithmus folgende Gestalt:

1. $T_0 = T_B; \quad T_{-1} = T_A; \quad i = 0;$

2. **Falls** $f(T_A) \cdot f(T_i) < 0$
   **dann** $T_B = T_i;$
   $\Delta_B = f(T_i);$
   **Falls** $f(T_i) \cdot f(T_{i-1}) > 0$
   **dann** $\Delta_A = \Delta_A/2;$
   **sonst** $\Delta_A = f(T_A);$
   **sonst** $T_A = T_i;$
   $\Delta_A = f(T_i);$
   **Falls** $f(T_i) \cdot f(T_{i-1}) > 0$
   **dann** $\Delta_B = \Delta_B/2;$
   **sonst** $\Delta_B = f(T_B);$

3. $T_{i+1} = \dfrac{\Delta_A \cdot T_B - \Delta_B \cdot T_A}{T_B - T_A}$

4. $i = i + 1;$

5. Mit Schritt 2 fortfahren, bis $T_B - T_A \leq R$

Da wir nicht mit einer kontinuierlichen, sondern mit einer gerasterten Zeitachse arbeiten, muß $T_i$, der Intention des Algorithmus entsprechend, nach oben oder unten gerundet werden.

### 3.6.8 Schätzung des lokalen, relativen Fehlers

Da man zur Bestimmung des lokalen, relativen Fehlers den exakten Wert für die Zustandsgröße $z_i(T_{k+l})$ nicht kennt, muß man sich numerisch einen Wert beschaffen, der deutlich genauer ist als der Wert $\tilde{z}_i(T_{k+l})$, den das Integrationsverfahren liefert.

Es gibt Integrationsverfahren, die diesen Wert durch Erhöhen der Ordnung liefern. Hierzu zählen die Runge-Kutta-Fehlberg-Verfahren und das Extrapolationsverfahren nach Bulirsch-Stoer. Diese Methoden sind sehr effektiv, da sie die gewünschte Information mit geringem Mehraufwand liefern.

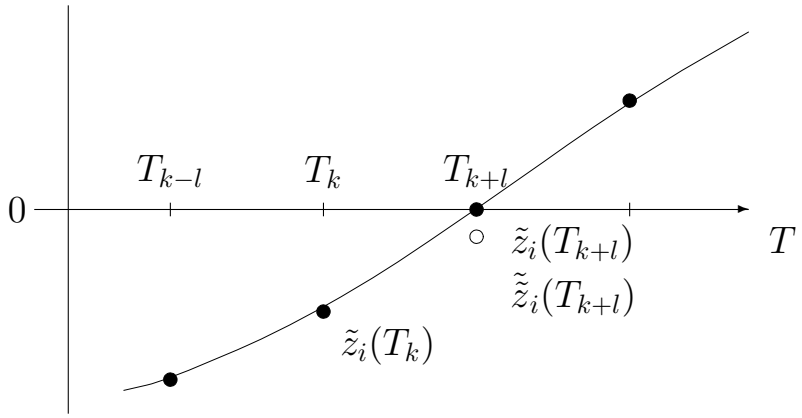
Beim Einsatz anderer Integrationsverfahren bedient man sich der Methode, im Integrationsintervall zwei aufeinanderfolgende Integrationsschritte mit halber Schrittweite auszuführen.

Den genaueren Wert wollen wir mit $\tilde{\tilde{z}}_i(T_{k+l})$ bezeichnen. Wenn wir annehmen, daß dieser Wert viel genauer ist als der erste, dann können wir mit

$$\tilde{\alpha}_i(T_{k+l}) = \mid \tilde{z}_i(T_{k+l}) - \tilde{\tilde{z}}_i(T_{k+l}) \mid$$

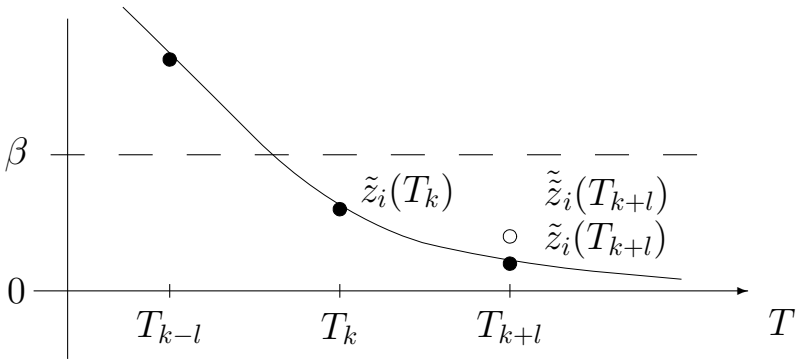
eine gute Schätzung des lokalen, absoluten Fehlers erzielen.

Zur Bestimmung des relativen Fehlers müssen wir $\tilde{\alpha}_i(T_{k+l})$ in Bezug setzen zur Größenordnung, in der sich die Zustandsgröße $z_i$ bewegt. Will man keine weiteren Werte aus der Vergangenheit mit berücksichtigen, bietet es sich an, das betragsmäßige Maximum der Größen $\tilde{z}_i(T_k)$, $\tilde{z}_i(T_{k+l})$ und $\tilde{\tilde{z}}_i(T_{k+l})$ heranzuziehen. Schneidet der Funktionsverlauf die Null-Linie, dann vermeidet diese Vorgehensweise das Auftreten unrealistisch hoher relativer Fehler.




Nähert sich der Funktionverlauf allmählich der Null-Linie an, ist zusätzlich zu bedenken, daß Rundungsfehler die erreichbare Genauigkeit beeinträchtigen. Es empfiehlt sich dann, den Dividenden zur Bestimmung des relativen Fehlers nicht kleiner als eine vorgegebene Schranke $\beta$ werden zu lassen. Zusammenfassend ermitteln wir den lokalen relativen Fehler nach

$$\rho_i(T_{k+1}) = \frac{\tilde{\alpha}_i(T_{k+1})}{max\{\tilde{\tilde{z}}_i(T_{k+1}), \tilde{z}_i(T_{k+1}), \tilde{z}_i(T_k), \beta\}}$$



### 3.6.9 Anpassung der Integrationsschrittweite

Wenn wir in der Lage sind, den lokalen, relativen Fehler eines Integrationsschritts zu schätzen, können wir diesen Wert auch dazu heranziehen, die Schrittweite so zu steuern, daß der Fehler unter einer angegebenen Schranke $\rho_{max}$, aber auch über einer Schranke $\rho_{min}$ bleibt.

Wir gehen dabei folgendermaßen vor:
Eine Anfangsschrittweite $l_0 \cdot R$ sei vorgegeben. Nach Ausführung eines Integrationsschritts vergleicht man das Maximum der für alle Zustandsgrößen geschätzten relativen Fehler

$$\rho_{tot}(T_{k+l}) = \max_i\{\tilde{\rho}_i(T_{k+l})\}$$

mit $\rho_{max}$ und $\rho_{min}$.

Liegt $\rho_{tot}$ zwischen diesen beiden Werten, wird die Schrittweite beibehalten. Liegt $\rho_{tot}$ unter $\rho_{min}$, wird im nächsten Integrationsschritt versucht, durch Verdopplung der Schrittweite den Fehler in das angestrebte Intervall zu bringen.

Liegt $\rho_{tot}$ über $\rho_{max}$, wiederholt man den Integrationsschritt entsprechend mit halber Schrittweite. Liefert dies einen Fehler, der noch immer zu hoch ist, halbiert man die Schrittweite erneut.

Ist der Fehler nach einer Schrittweitenhalbierung kleiner als $\rho_{min}$, würde eine Verdopplung im nächsten Integrationsschritt voraussichtlich wieder zu einem Fehler größer als $\rho_{max}$ führen. Daher vergrößert man im nächsten Integrationsschritt die Schrittweite lediglich um den Faktor 1.5.

# 4 Die Modellbeschreibungssprache

Basierend auf den im Kapitel 3 beschriebenen theoretischen Grundlagen entstand die Modellbeschreibungssprache des Simulationssystems SIMPLEX II. Sie trägt den Namen SIMPLEX-MDL, wobei MDL für Model Description Language steht.

Auch mit der in Kapitel 3 verwendeten mathematischen Notation lassen sich bereits Modelle spezifizieren. Soweit es möglich war, lehnt sich SIMPLEX-MDL auch daran an. Darüber hinaus jedoch bietet eine geschlossene formale Sprache

- eine einheitliche Darstellung von Modellen
- eine gute Lesbarkeit sowohl für Mensch wie auch Maschine
- die Möglichkeit, Modelle auf Vollständigkeit und Korrektheit zu prüfen
- die Möglichkeit, ein Simulationsprogramm zu generieren

Da Maschinenlesbarkeit eine unumgängliche Voraussetzung ist, müssen an die Notation einige Zugeständnisse gemacht werden, wie dies auch bei anderen Programmiersprachen der Fall ist. Zweidimensionale Notationen (Potenzen, Indizes, Fallunterscheidungen) müssen linearisiert werden und der Zeichenvorrat ist auf die darstellbaren ASCII-Zeichen zu beschränken. Um den Benutzer in dieser Beziehung nicht unnötig mit Neuem zu belasten, wurde in der Regel auf Schreibweisen zurückgegriffen, wie sie in gängigen Programmiersprachen (PASCAL, FORTRAN etc.) zu finden sind.

Dieses Kapitel stellt den lexikalischen und syntaktischen Aufbau der Sprache dar und erläutert die semantische Bedeutung der Symbole. Die exakte Spezifikation des definierten Modellverhaltens ergibt sich durch Vergleich mit Kapitel 3. Zum besseren Verständnis sind zahlreiche Beispiele eingestreut. Der letzte Abschnitt faßt die Sprachsyntax nochmals zusammen. Dort finden sich auch Hinweise zur verwendeten Backus-Naur-Notation.

## 4.1 Überblick

Die Modellbeschreibungssprache gibt dem Modellersteller eine bestimmte Methodologie vor, um sein Modell zu beschreiben.

Ein Modell kann in einer einzigen Komponente beschrieben oder in mehrere Komponenten zerlegt werden, welche Informationen austauschen. Auch diese Komponenten sind wiederum zerlegbar, so daß ein Modell hierarchisch strukturiert werden kann.

Komponenten, die nicht weiter zerlegt werden, heißen Basiskomponenten; Komponenten, die wiederum andere Komponenten beinhalten, nennen wir höhere Komponenten.

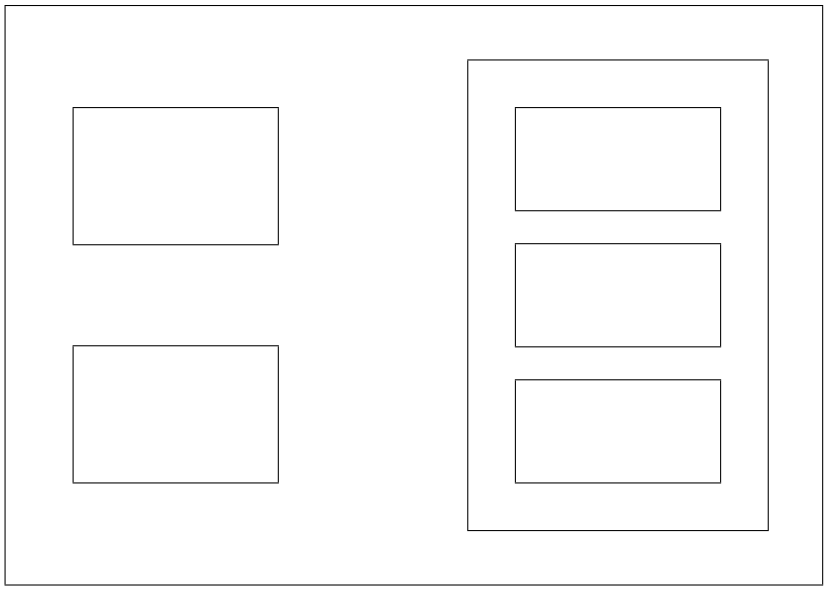


Abb 4.1-1: Hierarchische Zerlegung eines Modells

Ein Modell unterscheidet sich von einer Modellkomponente dadurch, daß es abgeschlossen ist, d. h. keine Kommunikation nach außen unterhält.

Die Modellbeschreibungssprache unterscheidet nicht zwischen Modellen und Modellkomponenten. Der Benutzer formuliert nur Modellkomponenten. Dadurch, daß alle Sensorgrößen vorbelegt werden, ist dafür gesorgt, daß jede Komponente auch als Modell betrieben werden kann. Dies hat den Vorteil, daß einzelne Komponenten getestet oder in größeren Modellen eingesetzt werden können, ohne daß an der Modellbeschreibung nderungen vorgenommen werden müssen.

Durch das Anlegen einer Komponente wird eine Klasse definiert. Wird ein Komponententyp in einer höheren Komponente mehrfach benötigt, können mehrere Ausprägungen dieser Klasse angelegt werden.

In den Basiskomponenten werden die Modellgrößen angelegt und deren dynamisches Verhalten beschrieben. Das Verhalten der Zustandsgrößen wird durch Ereignisse und Differentialgleichungen, das Verhalten der abhängigen Größen durch algebraische Gleichungen beschrieben.

Die Spezifikation der Modelldynamik wird auf ihre Eindeutigkeit geprüft. Der Wert einer Modellgröße darf zu jedem Zeitpunkt selbstverständlich nur durch eine einzige Beziehung definiert sein.

Höhere Komponenten dienen der Strukturierung des Modells. Sie setzen sich aus untergeordneten Komponenten zusammen und beschreiben den Informationsaustausch, der zwischen ihnen stattfindet.

Die Zerlegung eines Modells in einzelne Komponenten hat keinerlei Einfluß auf das Modellverhalten und kann nach beliebigen Gesichtspunkten (logische, physikalische Trennung) vorgenommen werden.

Die Kommunikation erfolgt über Verbindungen, die einer Komponente Einblick in andere Komponenten gewähren. Eine Verbindung kann auch als Wirkung einer Komponente auf eine andere Komponente angesehen werden.

Mit einer Modellkomponente verbindet sich die Vorstellung eines Glaskastens. Zwar kann man in ihn hineinschauen und alle Vorgänge beobachten, er ist von außen aber nur beeinflußbar, wenn ausdrücklich Reaktionen auf Einflüsse von außen vorgesehen sind.

Jede Modellkomponente kann nur ihre eigenen Größen verändern und ist daher gegenüber ihrer Umgebung autonom. Es existiert also kein direkter schreibender Zugriff auf Variablen fremder Komponenten. Hingegen sind alle Komponenten transparent, d. h. fremde Komponenten können auf alle Variablen lesend zugreifen, ohne eine Berechtigung zu benötigen.

Modellentwicklung wird stets von unten nach oben (buttom-up) betrieben. Die niederen Komponenten bilden die Grundlage für die darüberliegenden.

Als einführendes Beispiel diene der Dreiecksgenerator, der bereits in Kapitel 3 besprochen wurde. Er wird zunächst in einer einzigen Basiskomponente beschrieben und anschließend als zusammengesetzes Modell mit einer höheren Komponente und drei Basiskomponenten dargestellt.

```
BASIC COMPONENT   Dreieck            # Basiskomponente Dreiecksgenerator

DECLARATION OF ELEMENTS

   CONSTANT     a (REAL) := 1        # halbe Amplitude der Schwingung

   STATE VARIABLES
   DISCRETE     Z1 (REAL) := 1       # Schalterposition
   CONTINUOUS   Z2 (REAL) := 0       # Funktionsverlauf

   DEPENDENT VARIABLE
   DISCRETE     Y (REAL)             # aktuelle Steigung


DYNAMIC BEHAVIOUR

   Y := - Z1;                        # algebraische Gleichung

   DIFFERENTIAL EQUATION             # Differentialgleichung
       Z2' := Y;
   END

   WHENEVER  (Z2 > a)  AND  (Z1 <= 0)        # Ereignis
   DO
       Z1^ := 1;
   END

   WHENEVER  (Z2 < -a)  AND  (Z1 > 0)        # Ereignis
   DO
       Z1^ := -1;
   END

END OF   Dreieck
```

Die beschriebene Basiskomponente enthält alle drei möglichen Sprachkonstrukte zur Beschreibung der Modelldynamik. Wir zerlegen nun diese Basiskomponente so, daß sich die Zustandsvariablen `Z1` und `Z2` sowie die abhängige Variable `Y` auf drei Komponenten verteilen.

```
BASIC COMPONENT   Integ              # Basiskomponente Integrierer

DECLARATION OF ELEMENTS

   STATE VARIABLES
   CONTINUOUS   Z (REAL) := 0        # Ausgangsgroesse

   SENSOR VARIABLE
   CONTINUOUS   Y (REAL)             # Eingangsgroesse


DYNAMIC BEHAVIOUR

   DIFFERENTIAL EQUATION
       Z' := Y;                      # Integration
   END

END OF   Integ

-------------------------------------------------------------------

BASIC COMPONENT   Invert             # Basiskomponente Invertierer

DECLARATION OF ELEMENTS

   DEPENDENT VARIABLE
   CONTINUOUS  Y (REAL)              # Ausgangsgroesse

   SENSOR VARIABLE
   CONTINUOUS  X (REAL)              # Eingangsgroesse


DYNAMIC BEHAVIOUR

   Y := -X;                          # Invertierung

END OF   Invert

-------------------------------------------------------------------

BASIC COMPONENT   Hyst               # Basiskomponente Hystereseschalter

DECLARATION OF ELEMENTS

   CONSTANT     a (REAL) := 1        # halbe Hysterese
```

```
   STATE VARIABLES
   DISCRETE     Z (REAL) := 1        # Ausgangsgroesse

   SENSOR VARIABLE
   CONTINUOUS   X (REAL)             # Eingangsgroesse


DYNAMIC BEHAVIOUR

   WHENEVER  (X > a)  AND  (Z <= 0)       # Erreichen der oberen Schwelle
   DO
       Z^ := 1;
   END

   WHENEVER  (X < -a)  AND  (Z > 0)       # Erreichen der unteren Schwelle
   DO
       Z^ := -1;
   END

END OF   Hyst
```

Die höhere Komponente, welche diese drei Basiskomponenten zu einem Dreiecksgenerator zusammenschaltet, hat folgende Gestalt:

```
HIGH LEVEL COMPONENT   Dreieck

SUBCOMPONENTS

   Integ, Hyst, Invert            # beteiligte Komponenten

OUTPUT EQUIVALENCE

   Output := Integ.Z;             # Ausgangsgroesse der Komponente Dreieck

COMPONENT CONNECTIONS             # Verbindungen zwischen den Subkomponenten

   Integ.Z   -->  Hyst.X;
   Hyst.Z    -->  Invert.X;
   Invert.Y  -->  Integ.X;

END OF   Dreieck
```

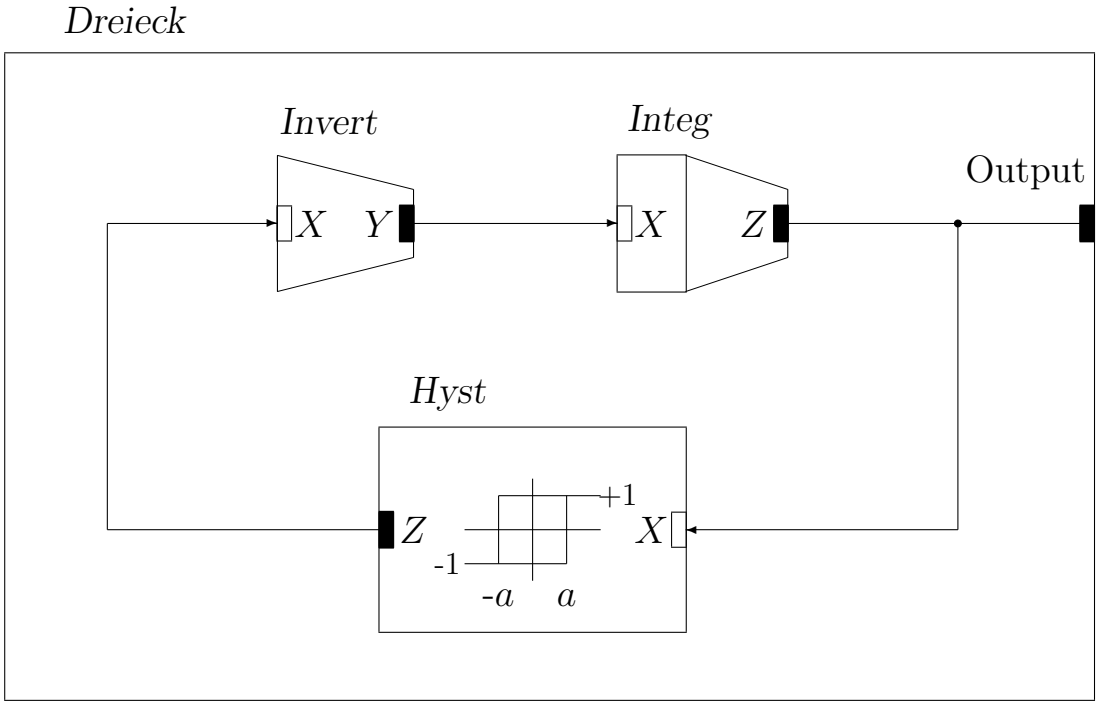



Abb. 4.1.-2: Dreiecks-Generator

Die höhere Komponente *Dreieck* zeigt exakt das gleiche Verhalten wie die Basiskomponente *Dreieck*. Um das dreieckförmige Signal nach außen weitergeben zu können, wird die Ausgangsvariable *Output* vereinbart.

Der Vorteil der komponentenweisen Zerlegung wird dann besonders deutlich, wenn sich aus bereits existierenden Teilkomponenten neue Modelle aufbauen lassen. Wir zeigen dies am Beispiel eines Sinus-/Cosinus-Generators, der auf die Komponenten *Integ* und *Invert* zurückgreift. Darüber hinaus wird ein Proportionalglied *Prop* benötigt. Es wird vom Klassenkonzept Gebrauch gemacht, da die Komponenten *Integ* und *Prop* je zweifach benötigt werden.

```
BASIC COMPONENT   Prop                 # Basiskomponente Proportionalglied

DECLARATION OF ELEMENTS

   CONSTANT    KP (REAL) := 1          # Proportionalitaetskonstante

   DEPENDENT VARIABLE
   CONTINUOUS  Y (REAL)                # Ausgangsgroesse


   SENSOR VARIABLE
   CONTINUOUS  X (REAL)                # Eingangsgroesse
```

```
DYNAMIC BEHAVIOUR

   Y := KP * X;

END OF   Prop

-------------------------------------------------------------------------

HIGH LEVEL COMPONENT   SinCos            # Sinus-/Cosinus-Generator

SUBCOMPONENTS                            # beteiligte Komponenten

   Integ1  OF CLASS  Integ,
   Integ2  OF CLASS  Integ,
   Prop1   OF CLASS  Prop,
   Prop2   OF CLASS  Prop,
   Invert

OUTPUT EQUIVALENCES

   OutSin  :=  Integ1.Z;                 # Ausgangssignal Sinus
   OutCos  :=  Integ2.Z;                 # Ausgangssignal Cosinus

COMPONENT CONNECTIONS

   Integ1.Z  -->  Prop1.X;
   Prop1.Y   -->  Invert.X;
   Invert.Y  -->  Integ2.X;
   Integ2.Z  -->  Prop2.X;
   Prop2.Y   -->  Integ1.X;

INITIALIZE

   Prop1.KP  :=  6.28;
   Prop2.KP  :=  6.28;

END OF   SinCos
```

Das folgende Bild zeigt die Verschaltung der einzelnen Komponenten des Sinus-/Cosinus-Generators.

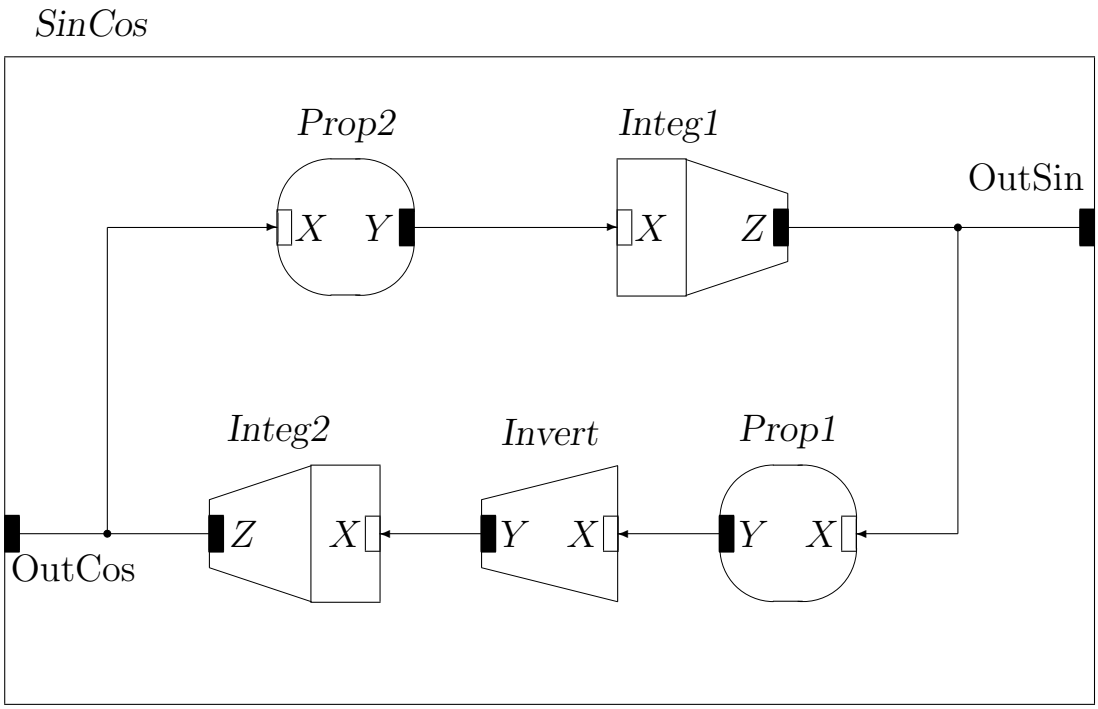



Abb. 4.1.-3: Sinus-/Cosinus-Generator

Diese Verschaltung realisiert das Differentialgleichungssystem

$$\begin{array}{lcl} z_1' & := & a \cdot z_2; \\ z_2' & := & -a \cdot z_1; \end{array} \qquad \begin{array}{lcl} z_1 & \hat{=} & \text{Integ1.Z} \\ z_2 & \hat{=} & \text{Integ2.Z} \\ a & \hat{=} & \text{Prop1.KP} = \text{Prop.KP} \end{array}$$

bzw. die Differentialgleichung

$$z_1'' + a^2 z_1 = 0$$

welche eine ungedämpfte harmonische Schwingung beschreibt.

Die genannten Beispiele wurden der Analogrechentechnik entnommen, da die dort verwendeten Komponenten einen sehr einfachen Aufbau besitzen und sich die komponentenweise Modellzerlegung dort schon seit langem eingebürgert hat.

Alle aufgeführten Komponenten stehen miteinander in engem Zusammenhang und bilden eine sogenannte Modellbank. Zum Aufbau von Modellen anderer Anwendungsgebiete können selbstverständlich andere Grundbausteine definiert werden, die in eine neue Modellbank eingegliedert werden.

## 4.2 Lexikalische Einheiten

Jede Modellkomponente wird in einer Datei abgelegt. Die Zeilen der Datei dürfen eine Länge von 80 Zeichen nicht überschreiten.

Aufeinander folgende Zeichen werden zu lexikalischen Einheiten zusammengefaßt.

Es wird nach lexikalischen Einheiten unterschieden, die der formalen Modellbeschreibung dienen

- Schlüsselwörter
- Sonderzeichen
- Bezeichner
- Zahlen
- Zeichenketten

und nach lexikalischen Einheiten, die den formal-sprachlichen Text ergänzen und nur informellen und dokumentarischen Charakter besitzen:

- Kommentare
- Meldungen

Die lexikalischen Einheiten, die der formalen Modellbeschreibung dienen, sind die terminalen Symbole der Sprachsyntax. Ihre Anordnung muß nach den Regeln erfolgen, welche die Sprachsyntax vorschreibt.

Ein terminales Symbol ist eine Folge einzelner Zeichen und darf nicht durch Trennzeichen unterbrochen sein.

Trennzeichen sind die Zeichen *Leerstelle*, *Tabulator* und *Zeilenende* und dienen der Abgrenzung terminaler Symbole gegeneinander. Nur zur Abgrenzung eines Sonderzeichens gegenüber einem anderen terminalen Symbol ist kein Trennzeichen erforderlich.

### Schlüsselwörter

Schlüsselwörter dienen zur Strukturierung der Modellbeschreibung. Alle Schlüsselwörter sind mit Großbuchstaben zu schreiben, um sie vom übrigen Text abzuheben.

Beispiel:

DYNAMIC BEHAVIOUR
leitet den Abschnitt der Modellbeschreibung ein, in dem das dynamische Verhalten beschrieben wird.

### Sonderzeichen

Sonderzeichen sind ebenso wie Schlüsselwörter reservierte Zeichen oder Zeichenketten, die in der Sprachsyntax vorkommen und eine bestimmte Bedeutung aufweisen.

Beispiele:

Operatoren wie `'+'`, `'-'`, `'*'`, `'/'`, `'>'`, `'>='`, `':='`
Markierung des Endes einer Wertzuordnung durch `';'`
Trennung von Listenelementen durch `','`
Einräumen einer Vorrangstellung durch Klammern: `'('`, `')'`

### Bezeichner

Bezeichner sind Namen für konkrete Objekte, die der Benutzer neu deklariert, oder welche die Sprache standardmäßig zur Verfügung stellt.

Notation:

```
identifier ::= letter { [ '_' ] letter_or_digit }

letter_or_digit ::= letter | digit

letter ::=   A | B | ... | Z
           | a | b | ... | z

digit  ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Bezeichner beginnen stets mit einem Buchstaben, gefolgt von weiteren Buchstaben oder Ziffern und dürfen Unterstriche enthalten. Zwei Unterstriche in Folge sowie ein Unterstrich am Anfang oder am Ende eines Bezeichners sind nicht gestattet.

Bezeichner dürfen bis zu 8 Zeichen lang sein.

Bezeichner dürfen Groß- und Kleinbuchstaben enthalten. Sie sind jedoch kein Unterscheidungsmerkmal (Speed = SPEED = speed). Um eine gut lesbare Dokumentation zu erhalten, muß in der Modellbeschreibung die bei der Deklaration gewählte Schreibweise beibehalten werden, beim Experimentieren dürfen Bezeichner beliebig mit großen und kleinen Buchstaben geschrieben werden.

Schlüsselwörter in Großschreibweise dürfen nicht als Bezeichner verwendet werden. Jede andere Schreibweise ist als Bezeichner erlaubt, da dann keine Verwechslungen möglich sind.

Benutzereigene Bezeichner müssen sich von Standardbezeichnern (siehe Abschnitt 4.5, 4.6) unterscheiden.

Beispiele:

```
erlaubte Bezeichner:
a  A  ABC  Abc  abc  ABC1   ABC_1   ABC_D   End

verbotene Bezeichner:
ABC_   _ABC   AB__C   1AB   5  5a  A12345678  END
```

### Zahlen

Zahlen sind die Elemente der numerischen Wertmengen INTEGER und REAL.

Notation:

```
number           ::=  [sign]  unsigned_number
unsigned_number  ::=  decimal [ [blank] exponent ]
```

```
decimal     ::=  digit_list
                 digit_list '.' digit_list

exponent    ::=  'E' sign digit_list
sign        ::=  '+' | '-'
digit_list  ::=  digit {digit}
blank       ::=  ' '
```

Zahlen werden dargestellt als Dezimalzahlen oder Dezimalbrüche mit oder ohne Exponent zur Basis 10. Aus Gründen der Deutlichkeit ist mindestens eine Ziffer vor und nach dem Dezimalpunkt sowie das Vorzeichen des Exponenten obligatorisch. Mantisse und Exponent dürfen durch eine Leerstelle getrennt sein.

Beispiele:

```
3, 3.5, 3 E+2, 3E+2, 3.5 E+2, 0.5, 0.5 E-6
```

Eine Zahl gilt als reell, wenn ihr Wert nicht als ganze Zahl darstellbar ist.

Beispiel:

```
3.5 E+2     ist INTEGER
3.519 E+2   ist REAL
```

Die zulässigen Wertebereiche orientieren sich an der Zahlendarstellung, wie sie auf 32 Bit-Rechnern üblich ist.

```
INTEGER  - 2 E+9   ...   +  2 E+9
REAL     - 1 E+38   ...  +  1 E+38, kleinste Zahl nach der
                                    Null: 1 E-38,
                                    Anzahl signifikanter
                                    Dezimalstellen: 16
```

### Zeichenketten

Eine Zeichenkette ist eine Folge von Zeichen, die in Anführungszeichen ( ” ) eingeschlossen ist. Soll die Zeichenkette selbst ein Anführungszeichen enthalten, so ist diesem ein nach hinten liegender Schrägstrich ( \ ) voranzustellen.

Eine Zeichenkette darf alle darstellbaren Zeichen enthalten sowie die Zeichen `'\n'` für Zeilenvorschub und `'\t'` für Tabulator.

Beispiele:

```
"Ein Modell"
"Das Modell \"Springender Ball\""
"Koeffizient  a = %f \n"
```

### Kommentare

Kommentare werden durch das Zeichen `'#'` eingeleitet und durch das Zeilenende abgeschlossen. Der Inhalt eines Kommentars wird überlesen und ist für die Erzeugung des Simulationsprogramms ohne Bedeutung.

Beispiel:

```
CONSTANT   g(REAL)  := 9.81   # Erdbeschleunigung [m/sec^2]
```

### Meldungen

Werden beim Überprüfen der Modellbeschreibung syntaktische oder semantische Fehler festgestellt, wird im Originaltext an der entsprechenden Stelle eine Fehlermeldung ausgegeben. Diese wird durch das Sonderzeichen `'$'` in der ersten Spalte einer Zeile eingeleitet. Ist der Fehler behoben, wird bei der nächsten Überprüfung die Fehlermeldung wieder aus dem Originaltext entfernt.

Beispiel:

```
   A := B + C;
$       |
$ [Fehler 1]    Bezeichner ist nicht deklariert
$ ----------
$
```

## 4.3 Basiskomponenten

Basiskomponenten enthalten Modellgrößen und die Vereinbarung ihres dynamischen Verhaltens. Durch benutzereigene Definitionen können Wertemengen durch Aufzählung vereinbart und tabellarische Funktionen angelegt werden.

Basiskomponenten sind in drei Abschnitte gegliedert:

1. Die Vereinbarung lokal gültiger Wertemengen und tabellarischer Funktionen
2. Die Aufzählung der Modellgrößen
3. Die Beschreibung des dynamischen Verhaltens der Modellgrößen

Alle Abschnitte außer dem zweiten sind optional.

Auf die Beschreibung der Dynamik kann jedoch nur verzichtet werden, wenn die Komponente keine abhängigen Größen enthält. Deren Verhalten ist in jedem Fall zu definieren. Zustandsgrößen behalten ihren Initialwert konstant bei.

**Syntax:**

```
basic_component ::=   BASIC COMPONENT   identifier
                      [local_definitions]
                      declaration_of_elements
                      [dynamic_behaviour]
                      END OF   identifier
```

Die Bezeichner in der ersten und letzten Zeile geben der Komponente einen Namen. Sie müssen identisch sein und mit dem Bezeichner übereinstimmen, der beim Anlegen der Komponente in der Modellerstellungsumgebung vergeben wurde.

### 4.3.1 Lokale Definitionen

Definitionen in diesem Abschnitt besitzen nur innerhalb der betreffenden Basiskomponente Gültigkeit. Hier kann der Benutzer Wertemengen durch Aufzählung sowie tabellarische Funktionen vereinbaren.

**Syntax:**

```
local_definitions ::=    LOCAL DEFINITION[S]
                       { enumeration_set  }
                       { tabular_function }
```

#### 4.3.1.1 Definition von Wertemengen durch Aufzählung

Für Modellgrößen, mit denen keine Rechenoperationen durchgeführt werden und die nur eine geringe Zahl von Werten annehmen können, empfiehlt es sich, deren Wertemengen durch Aufzählung der Elemente zu vereinbaren.

**Syntax:**

```
enumeration_set  ::=   VALUE SET  identifier  ':'
                       '(' enum_value_list ')'

enum_value_list  ::=   identifier ',' identifier
                                 { ',' identifier }
                     | identifier '<' identifier
                                 { '<' identifier }
```

Die Wertemenge ist in aufsteigender Folge sortiert, wenn ihre aufgezählten Elemente durch das Kleiner-Zeichen `'<'` getrennt werden. Sie ist unsortiert, wenn ihre Elemente durch Kommata getrennt sind.

Beispiele:

```
VALUE  SET  Colour:  (red, yellow, green, blue)
VALUE  SET  Size:    (small < medium < large)
```

Zwischen Modellgrößen mit gleichen Wertemengen sind Vergleichsoperationen erlaubt. Ist die Wertemenge unsortiert, ist nur ein Vergleich auf Gleichheit oder Ungleichheit zulässig. Bei sortierten Wertemengen sind auch die übrigen Vergleichsoperatoren anwendbar.

#### 4.3.1.2 Definition von Tabellenfunktionen

Eine Tabellenfunktion ist eine Abbildungsvorschrift, die zwischen den möglichen Werten der Argumente und dem Funktionswert tabellarisch die Zuordnung herstellt.

Beim derzeitigen Stand ist nur die Abbildung eines einzigen reellen Arguments auf einen reellen Funktionswert zulässig.

**Syntax:**

```
tabular_function  ::=   TABULAR FUNCTION   identifier
                        '(' REAL '-->' REAL ')'
                        CONTINUOUS
                        BY  interpolation_method  INTERPOLATION
                        ON  '('number_list')' '-->'
                            '('number_list')'

interpolation_method  ::=  LINEAR
                         | CUBIC BESSEL
                         | CUBIC SPLINE

number_list  ::=  number {',' number}
```

**Semantik:**

Die erste Zahlenreihe gibt die Lage der Stützstellen in aufsteigender Folge an, die zweite Zahlenreihe die Funktionswerte an diesen Stützstellen. Die Anzahl der Funktionswerte muß mit der Anzahl der Stützstellen übereinstimmen. Für eine lineare Interpolation sind mindestens zwei, für eine kubische Interpolation mindestens drei Stützstellen erforderlich.

Wird die Tabellenfunktion mit einem Argument aufgerufen, das zwischen zwei Stützstellen liegt, dann wird der Funktionswert durch Interpolation ermittelt. Neben der linearen Interpolation, die zwei benachbarte Stützwerte durch einen Geradenzug verbindet, stehen zwei kubische Interpolationsverfahren zur Verfügung, die je zwei Stützwerte links und rechts vom Argumentwert berücksichtigen und ein Polynom 3. Grades durch diese Punkte legen.

Die kubische Bessel-Interpolation sorgt dafür, daß die 1. Ableitung der interpolierten Kurve stetig ist, die kubische Spline-Interpolation liefert sogar eine stetige 2. Ableitung.

Beispiel:

```
TABULAR FUNCTION  Parabel  (REAL --> REAL)
CONTINUOUS
BY  CUBIC BESSEL INTERPOLATION
ON  (-1.5,  -1, -0.5,  0, 0.5,  1, 1.5)      # Stuetzstellen
--> (-2.25, -1, -0.25, 0, 0.25, 1, 2.25)     # Stuetzwerte
```

Die Tabellenfunktion `Parabel` beschreibt einen punktsymmetrischen Kurvenverlauf, der sich aus zwei Parabelästen zusammensetzt.

Wird die Tabellenfunktion mit einem Argument aufgerufen, das außerhalb des Stützstellenbereichs liegt (im Beispiel: kleiner $-1.5$ oder größer $+1.5$), dann wird der Simulationslauf mit einer Fehlermeldung beendet.

### 4.3.2 Deklaration der Modellgrößen

Modellgrößen sind die Elemente einer Modellkomponente. Eine Modellgröße wird vereinbart, indem man ihr einen Namen verleiht und ihre Eigenschaften spezifiziert.

Modellgrößen werden gemäß ihrer *Verwendung* unterteilt in:

- Konstanten
- Zustandsvariablen
- abhängige Variablen
- Sensorvariablen

**Syntax:**

```
declaration_of_elements ::=   DECLARATION OF ELEMENTS
                            [ list_of_constants ]

                            [ list_of_state_variables ]

                            [ list_of_dependent_variables ]

                            [ list_of_sensor_variables ]
```

Weitere Eigenschaften einer Modellgröße sind:

- *Zeitverlauf*
  `DISCRETE, CONTINUOUS`
- *Wertemenge*
  `INTEGER, REAL, LOGICAL,` Aufzählungsmenge

Eine Initialisierung vervollständigt die Deklaration einer Modellgröße.

**Syntax:**

```
list_of_constants ::=              CONSTANT[S]
                                   quantity {',' quantity }

list_of_state_variables ::=        STATE VARIABLE[S]
                                 [[ DISCRETE ]
                                   quantity { ',' quantity } ]
                                  [ CONTINUOUS
                                   quantity { ',' quantity } ]

list_of_dependent_variables ::=    DEPENDENT VARIABLE[S]
                                 [[ DISCRETE ]
                                   quantity { ',' quantity } ]
                                  [ CONTINUOUS
                                   quantity { ',' quantity } ]

list_of_sensor_variables ::=       SENSOR VARIABLE[S]
                                 [[ DISCRETE ]
                                   quantity { ',' quantity } ]
                                  [ CONTINUOUS
                                   quantity { ',' quantity } ]

quantity ::=   identifier '(' range ')'  [ ':='value ]

range ::=    INT[EGER]
           | REAL
           | LOGICAL
           | identifier

value ::=    number
           | logic_value
           | identifier

logic_value ::=  TRUE | FALSE
```

Beispiel:

```
DECLARATION OF ELEMENTS

CONSTANT  m (INT) := 10

STATE VARIABLES
DISCRETE     Light (Colour) := red,
             Z     (REAL)   := -1.5
CONTINUOUS   Y     (REAL)   := 0

DEPENDENT VARIABLES
DISCRETE     valid (LOGICAL)
             State (INTEGER)

SENSOR VARIABLES
CONTINUOUS   u (REAL),
             v (REAL) := 1
```

Die Konstante `m` wird mit der Wertemenge `INTEGER` versehen und mit 10 initialisiert. Die diskrete Zustandsvariable `Light` besitzt die Wertemenge `Colour` und daraus den Anfangswert `red`.

### Verwendung der Modellgrößen

Modellgrößen besitzen zu jedem Zeitpunkt des Modellablaufs einen bestimmten Wert aus ihrer Wertemenge.

Eine Konstante ist eine Modellgröße, die ihren Wert während des dynamischen Modellablaufs nicht verändert. Im Abschnitt "Dynamic Behaviour" darf ihr deshalb kein Wert zugewiesen werden.

Eine Zustandsvariable ist eine unabhängige Modellgröße. Sie kann zu jedem Zeitpunkt jeden beliebigen Wert aus ihrer Wertemenge annehmen. Ihre Dynamik wird durch Differentialgleichungen oder Ereignisse beschrieben.

Eine abhängige Variable kann zu jedem Zeitpunkt aus den Werten der Zustandsvariablen, Sensorvariablen und Konstanten errechnet werden und wird durch eine algebraische Gleichung definiert.

Eine Sensorvariable besitzt während des gesamten Modellablaufs den Wert einer ihr zugeordneten Modellgröße aus einer anderen Komponente. Ihr darf im Abschnitt "Dynamic Behaviour" kein Wert zugewiesen werden.

### Zeitverlauf der Modellgrößen

Eine diskrete Modellgröße verändert ihren Wert sprunghaft zu diskreten Zeitpunkten, während eine kontinuierliche Modellgröße ihren Wert stetig in der Zeit verändert.

Eine *Zustandsvariable* ist als kontinuierlich zu deklarieren, wenn ihre Dynamik durch eine Differentialgleichung beschrieben wird. Wird der Wert einer Zustandsvariablen nur durch Ereignisse verändert, ist ihr Zeitverlauf diskret.

Eine *abhängige Variable* ist als kontinuierlich zu deklarieren, wenn die algebraische Gleichung, durch die sie definiert wird, kontinuierliche Operanden enthält.

Eine *Sensorvariable* ist als kontinuierlich zu deklarieren, wenn es möglich sein soll, ihr eine kontinuierliche Variable zuzuordnen.

Eine Variable kann nur dann als kontinuierlich deklariert werden, wenn sie die Wertemenge REAL besitzt.

### Wertemengen der Modellgrößen

Standardmäßig werden die numerischen Wertemengen INTEGER und REAL sowie die boolsche Wertemenge LOGICAL zur Verfügung gestellt. Darüber hinaus kann sich der Benutzer neue Wertemengen durch Aufzählung einzelner Werte vereinbaren (siehe Abschnitt 4.3.1.1).

*INTEGER:* ganze Zahlen mit Wertebereich $-2 \cdot 10^9 \ldots +2 \cdot 10^9$
Teilmenge der Wertemenge REAL
interne Darstellung: C-Datentyp "long"

*REAL:* reelle Zahlen in Gleitkommadarstellung mit Wertebereich $-10^{38} \ldots +10^{38}$
16 signifikante Dezimalstellen
kleinste darstellbare Zahl nach der Null: $10^{-38}$
interne Darstellung: C-Datentyp "double"

*LOGICAL:* boolsche Werte TRUE und FALSE

*Aufzählungsmenge:* vom Benutzer definierte Werte

Die mit diesen Wertemengen möglichen Operationen sind im Abschnitt 4.3.3.1 *Die Bildung algebraischer Ausdrücke* beschrieben.

### Vorbesetzung der Modellgrößen

Konstanten und Zustandsvariablen müssen mit einem Anfangswert belegt werden. Die Vorbelegung von abhängigen Variablen und Sensorvariablen ist optional. Ist keine Initialisierung angegeben, werden diese Variablen mit Null vorbesetzt.

Der Anfangswert muß selbstverständlich in der Wertemenge der deklarierten Modellgröße enthalten sein.

Diese sogenannte Urinitialisierung stellt sicher, daß ein Modell nicht ohne Vorbesetzung des Modellzustands gestartet werden kann. Die Initialisierung, die in einer Basiskomponente vorgenommen wird, kann durch eine übergeordnete höhere Komponente überschrieben werden. Diese wiederum ist nur gültig, wenn sie nicht vom Experimentiersystem verändert wird.

Die Vorbelegung von abhängigen Variablen ist dann sinnvoll, wenn bei der ersten Ausführung der statischen Beziehungen die Gefahr eines Gleitkommaüberlaufs besteht.

Die Initialisierung von Sensorvariablen besitzt dann Gültigkeit, wenn die Sensorvariable nicht angeschlossen ist. Sie behält dann ihren Initialwert während des gesamten Simulationslaufs bei.

**Die Simulationszeit T**

Die Simulationszeit $T$ ist eine Größe, die von der Sprache standardmäßig zur Verfügung gestellt wird und daher nicht eigens deklariert werden muß.

Wir hatten im Kapitel 3 die Zeit als Zahlenpaar $(T, J)$ kennengelernt. Die Standardgröße $T$ entspricht der Größe $T$ des Zahlenpaares zuzüglich eventuell notwendiger Zwischenzeitpunkte zur Ausführung der numerischen Integration.

Die Simulationszeit $T$ ist zu Beginn eines Simulationslaufs standardmäßig mit Null belegt. Über die Experimentierumgebung kann jedoch auch ein beliebiger anderer Startzeitpunkt eingestellt werden. Der vorgegebene Wert wird auf einen Wert aus der Zeitmenge $\mathcal{T}_c$ gerundet. Während des Simulationslaufs wird die Zeit selbsttätig fortgeschaltet.

Die Simulationszeit $T$ darf in algebraischen Ausdrücken auftreten und wird semantisch wie eine Sensorgröße mit kontinuierlichem Zeitverlauf und reeller Wertemenge behandelt.

Die Zeitmenge $\mathcal{T}_c$ kann vom Benutzer durch die Angabe des Steuerparameters `'TScal'` über die Experimentierumgebung vorgegeben werden. `'TScal'` entspricht dem Auflöungsvermögen `'R'` und damit dem kleinsten zeitlichen Abstand zweier Ereignisse.

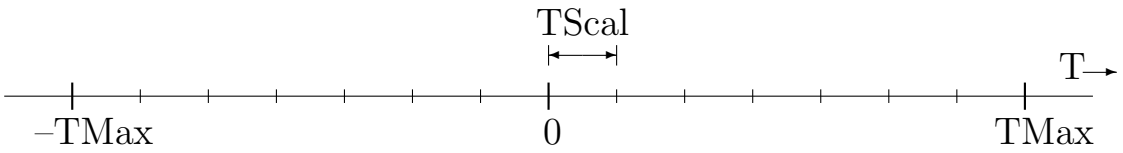



Abb. 4.3.1-1: Vorgabe der Zeitmenge $\mathcal{T}_c$

Das obere und untere Ende der Zeitachse ergibt sich als Produkt der größten darstellbaren ganzen Zahl *MaxInt* und *TScal*.

$TMax = MaxInt * TScal$

Für die Größe *MaxInt* wird der Wertebereich des Datentyps *long* zugrunde gelegt, der für gewöhnlich mit 32 Bit dargestellt wird und damit den Wert $2 \cdot 10^9$ besitzt.

Beispiel:

Für $TScal = 0.001$ und $MaxInt = 2 \cdot 10^9$ (32 Bit) ist $TMax = 2 \cdot 10^6$

Die Werte $J$ der Zeitmenge $\mathcal{T}_D$ sind dem Benutzer zur Formulierung von Modellen nicht zugänglich. Da $J$ für das Vielfache einer infinitesimalen Einheit steht, ist dies auch nicht sinnvoll.

Bei der Protokollierung der Ereignisse jedoch, gibt diese Größe dem Benutzer die Möglichkeit, die einzelnen Zustände, die während eines Zeitpunktes $T$ entstehen, auseinanderzuhalten und die Geschehnisse im Modell nachzuvollziehen.

Zu diesem Zweck wird allerdings nicht $J$, sondern der sogenannte Simulationstakt `'Takt'` ausgegeben. Der Takt unterscheidet sich von $J$ dadurch, daß er mit jedem neuen Zeitpunkt auf Null zurückgesetzt wird.

Trägt man statt $J$ den Takt auf, ergibt sich ein etwas modifiziertes Zustandsfolgediagramm.

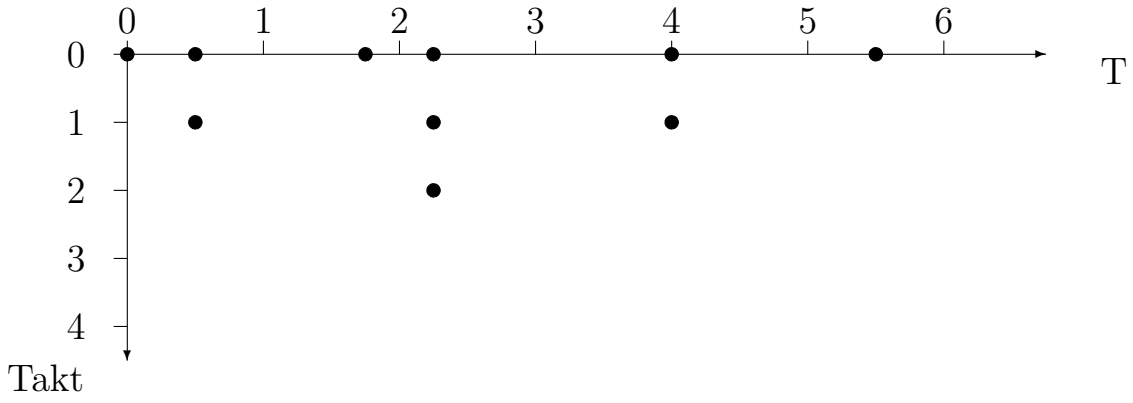



Abb. 4.3.1-2: Zustandsfolgediagramm mit Takten

### 4.3.3 Beschreibung des dynamischen Verhaltens

Für Zustandsgrößen und abhängige Größen ist das zeitliche Verhalten zu definieren. Hierfür stehen Sprachkonstrukte zur Formulierung von

- Algebraischen Gleichungen
- Differentialgleichungen
- Ereignissen

zur Verfügung.

Ein weiteres Sprachkonstrukt gestattet die abschnittsweise Definition dieser Beziehungen durch die Angabe von Bedingungen.

Die funktionalen Zusammenhänge werden durch algebraische Ausdrücke hergestellt.

**Syntax:**

```
dynamic_behaviour ::=    DYNAMIC BEHAVIOUR
                         statement_sequence

statement_sequence ::=   statement  { statement }

statement ::=   algebraic_equation
              | differential_equations
              | region_defining_statement
              | event_defining_statement
```

#### 4.3.3.1 Die Bildung algebraischer Ausdrücke

Ein algebraischer Ausdruck wird gebildet, indem Operanden durch Operatoren miteinander verknüpft werden.

**Syntax:**

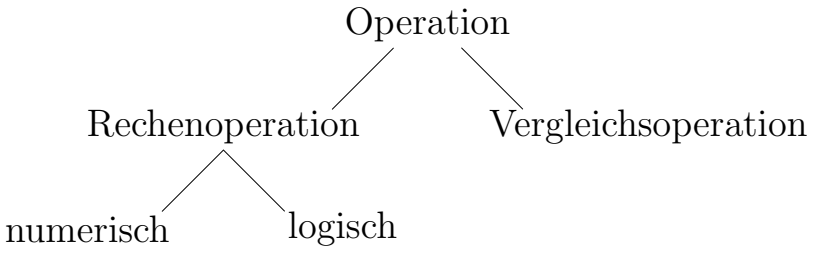
```
expression  ::=   simple_expression
                | comparison

comparison  ::=   simple_expression  compare_op
                  simple_expression

compare_op ::=   '<'  |  '>'  |  '=' |  '<>' |
                 '<=' |  '>='

simple_expression ::=   [ sign ]  term  { add_op term }

sign ::=     '+' | '-'

term ::=    factor
          | term mul_op factor

add_op ::=  '+' |  '-' |  OR

mul_op ::=  '+' |  '/' | AND

factor ::=    identifier
            | unsigned_number
            | logic_value
            | T
            | function_execution
            | '(' expression ')'
            | NOT factor
```

Zu den Operanden (factors) zählen:

- Größen (Variablen, Konstanten, selektierte Größen)
- Werte aus Aufzählungsmengen
- Zahlenwerte
- logische Werte ( `TRUE` und `FALSE` )
- Zeit `T`
- Funktionsaufrufe
- Ausdrücke in runden Klammern.

Ein Operator steht für eine einstellige oder zweistellige, elementare Funktion. Diese elementaren Funktionen bezeichnet man als Operationen und ihre Argumente als Operanden.

Operationen lassen sich in numerische und logische Rechenoperationen sowie Vergleichsoperationen unterteilen.




Ebenso wie die beteiligten Operanden besitzt der Wert eines Ausdruckes die Attribute *Wertemenge* und *Zeitverlauf*. Diese Attribute hängen ab von den beteiligten Operanden und Operatoren.

### Numerische Rechenoperationen

Numerische Rechenoperationen sind mit Operanden der Wertemengen INTEGER und REAL möglich. Als Operatoren sind zugelassen:

```
" + "   :   Addition
" - "   :   Subtraktion
" * "   :   Multiplikation
" / "   :   Division
```

In Ausdrücken, die mehrere numerische Operatoren enthalten, besitzen die Multiplikation und Division Vorrang vor Addition und Subtraktion d.h. es gilt die „Punkt-vor-Strich-Regel“.

Das Ergebnis eines numerischen Ausdrucks besitzt die Wertemenge `INTEGER`, wenn alle Operanden dieses Ausdrucks die Wertemenge `INTEGER` haben. Besitzen einer oder mehrere Operanden eines numerischen Ausdrucks die Wertemenge `REAL` oder kommt in diesem Ausdruck der Divisionsoperator '/' vor, werden alle `INTEGER`-Operanden nach `REAL` konvertiert und das Ergebnis ist selbstverständlich auch `REAL`.

Der Zeitverlauf des Ergebnisses richtet sich nach den beteiligten Operanden. Ist einer der Operanden kontinuierlich, besitzt auch das Ergebnis kontinuierlichen Zeitverlauf. Sind alle Operanden konstant, ist auch das Ergebnis konstant. In den übrigen Fällen erhält man ein zeitdiskretes Ergebnis.

| | INT | REAL |
|---|---|---|
| INT | Int | Real |
| REAL | Real | Real |

| | CONSTANT | DISCRETE | CONTINUOUS |
|---|---|---|---|
| CONSTANT | Constant | Discrete | Continuous |
| DISCRETE | Discrete | Discrete | Continuous |
| CONTINUOUS | Continuous | Continuous | Continuous |

Abb 4.3.3-1: Wertebereich und Zeitverlauf des Ergebnisses einer numerischen Rechenoperation

Beispiele:

x, y seien reelle Größen ( x = 3.5, y = 7 )
i, j seien ganzzahlige Größen ( i = 5, j = 4 )

```
1.  i + 2*j + 5.0      Ergebnis:   18, ganzzahlig
2.  i/j + 1/4          Ergebnis:  1.5, reell
3.  x - y/2            Ergebnis:    0, reell
4.  i*x - j            Ergebnis: 13.5, reell
```

### Logische Rechenoperationen

Logische Rechenoperationen sind nur mit Operanden der Wertemenge LOGICAL möglich. Als Operatoren sind zugelassen:

```
" OR "   :   logisches ODER      |
" AND "  :   logisches UND       |   zunehmender Vorrang
" NOT "  :   Negation            V
```

In Ausdrücken, die mehrere logische Operatoren enthalten, besitzt "NOT" Vorrang vor "AND" und "AND" Vorrang vor "OR".

Beispiele:

a, b, c seien logische Größen ( a = TRUE, b = FALSE, c = FALSE )

```
1.  a  OR  b  AND  c    Ergebnis:  TRUE
2.  a  AND  NOT b       Ergebnis:  TRUE
```

### Vergleichsoperationen

Vergleichsoperationen sind zwischen je zwei Operanden mit numerischen Wertemengen (INTEGER oder REAL), der Wertemenge LOGICAL oder gleichen Aufzählungsmengen möglich.

Als Operatoren sind

```
" = "   :   gleich
" <> "  :   ungleich
```

für alle Operanden außer für solche mit kontinuierlichem Zeitverlauf zugelassen.

Die Operatoren

```
" < "   :   kleiner
" <= "  :   kleiner gleich
" > "   :   groesser
" >= "  :   groesser gleich
```

sind für alle Operanden außer für solche mit Wertemenge LOGICAL oder unsortierten Aufzählungsmengen zugelassen.

Das Ergebnis einer Vergleichsoperation hat stets die Wertemenge LOGICAL. Ausdrücke, die mehrere Vergleichsoperatoren enthalten, müssen der Bedeutung gemäß geklammert werden, da für Vergleichsoperatoren keine vorrangige Bearbeitung festgelegt ist.

Ist eine Seite des Vergleichsausdrucks vom Wertebereich INTEGER und die andere vom Wertebereich REAL, werden alle beteiligten Operanden nach REAL konvertiert.

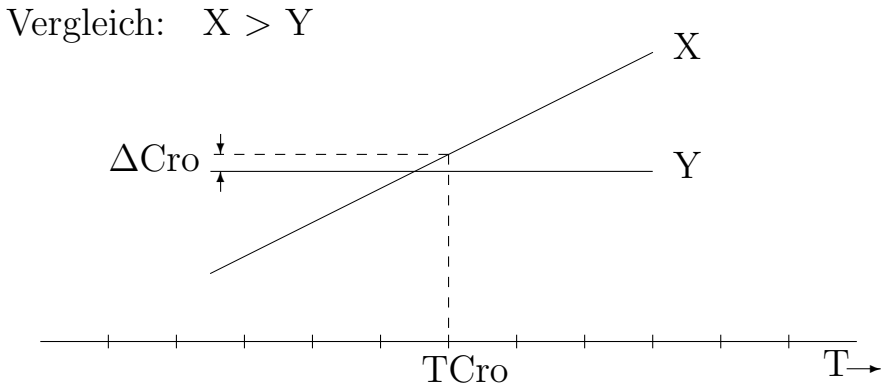
Beispiele:

```
1.  x + y  >= 5
2. (x + y  >= 5)  OR  (z < 0)
```

**Vergleiche mit kontinuierlichen Operanden (Crossings)**

Sind in einem Vergleichsausdruck Operanden enthalten, deren aktueller Wert während der numerischen Integration verändert wird, ist zu prüfen, ob dieser Vergleichsausdruck innerhalb des Integrationsschritts seinen Wahrheitswert gewechselt hat, d.h. ein sogenanntes *Crossing* eingetreten ist.

Ist dies der Fall, wird der Integrationsschritt lediglich bis zu dem Zeitpunkt ausgeführt, an dem der Vergleichsausdruck seinen logischen Wert wechselt. Der Crossingzeitpunkt *TCro* wird so genau bestimmt, wie es die diskretisierte Zeitachse zuläßt.




Dadurch bleibt eine Restabweichung $\Delta Cro$, um die sich die zu vergleichenden Operanden unterscheiden.

Zum Auffinden des Zeitpunkts *TCro* ist ein iteratives Suchverfahren auszuführen. Die Anzahl der maximal erlaubten Iterationsschritte kann durch den Steuerparameter

*CroMax*

vorgegeben werden. Beim Überschreiten dieser Anzahl wird der Simulationslauf mit Fehlermeldung abgebrochen. Die durchschnittliche Anzahl der benötigten Suchschritte wird in der Datei *Statistics.out* protokolliert.

**Zeitvergleiche**

Die Simulationszeit $T$ darf in einem Vergleichsausdruck

- nur auf der rechten oder linken Seite und
- nur als alleinstehender Operand

auftreten. Vergleiche auf Gleichheit oder Ungleichheit sind nicht zugelassen.

Beispiele:

```
T  >=  15
T  <   MIN (X,Y)
D + E  >  T
```

Diese Einschränkungen sind erforderlich, um ein effizientes Fortschalten der Simulationszeit gewährleisten zu können. Liegt ein Vergleichsausdruck vor, bei dem es nicht möglich ist, nach der Zeit aufzulösen, wie z.B.

```
SIN (T)  >  0.5
```

ist eine algebraische Gleichung zu vereinbaren, die den Teilausdruck, der die Zeit enthält, auswertet.

Beispiel:

```
SinT  :=  SIN (T) ;

IF  SinT  >  0.5    DO   ...   END
```

### Funktionsaufrufe

Ein Funktionsaufruf besteht aus dem Namen der Funktion mit angehängter Argumentenliste in runden Klammern.

**Syntax:**

```
function_execution ::=   identifier '(' argument_list ')'

argument_list      ::=   expression {',' expression }
```

**Semantik:**

Die Funktion bildet die übergebenen Argumente auf den Funktionswert ab. Die Anzahl der Argumente und ihre Wertemengen müssen mit der Deklaration der Funktion übereinstimmen (für Standardfunktionen siehe Anhang). Für den Zeitverlauf der Argumente gibt es keine Einschränkungen.

Die Wertemenge des Funktionswertes ist durch die Deklaration der Funktion festgelegt.

Der Zeitverlauf des Funktionswertes wird innerhalb eines Ereignisses stets als zeitdiskret betrachtet.

Außerhalb richtet sich der Zeitverlauf des Funktionswertes nach dem Zeitverlauf der Argumente. Es gelten dieselben Regeln wie bei der Verknüpfung durch Rechenoperationen. Ist eines der Argumente zeitkontinuierlich, so ist es auch der Funktionswert. Ansonsten ist der Funktionswert zeitdiskret, es sei denn, alle Argumente sind konstant. Dann ist auch der Funktionswert konstant.

### Bearbeitungsreihenfolge eines algebraischen Ausdrucks

Die Bearbeitungsreihenfolge eines algebraischen Ausdrucks kann durch das Einschließen von Teilausdrücken in runde Klammern gesteuert werden. Teilausdrücke dürfen wiederum Klammerungen enthalten.

Bei einem Ausdruck, der mehrere runde Klammern enthält, werden zuerst die Werte der innersten Teilausdrücke errechnet und zur Auswertung des nächst äußeren Teilausdrucks herangezogen.

In einem Ausdruck oder Teilausdruck ohne Klammerung entscheidet die Vorrangstufe der Operationen über die Bearbeitungsreihenfolge. Die Abstufung ist:

1. Stufe: `NOT`
2. Stufe: `AND, '*', '/'`
3. Stufe: `OR, '+', '-'`
4. Stufe: `'=', '<>', '>', '>=', '<', '<='`

Es ist zu beachten, daß die Vergleichsoperatoren den niedrigsten Rang haben. Ausdrücke, die mehrere Vergleichsoperatoren enthalten, sollten daher prinzipiell geklammert werden.

Stehen Operatoren gleicher Vorrangstufe hintereinander, wird zuerst die linke und dann die rechte Operation ausgeführt.

Beispiele:

```
1.   3 + 4 * 5                                  Ergebnis:   23
2.   (3 + 4) * 5                                Ergebnis:   35
3.   (( 2 * 10 + 3) * 10 + 4) * 10 + 5          Ergebnis: 2345
4.   2 + 7  >  10                               Ergebnis: FALSE
5.   (2 + 7  >  10)  OR  (3 = 6/2)              Ergebnis: TRUE
```

#### 4.3.3.2 Algebraische Gleichungen

Eine algebraische Gleichung bringt eine statische Beziehung zum Ausdruck. Sie definiert den Wert einer abhängigen Variablen zu jedem Zeitpunkt, indem sie ihre funktionelle Abhängigkeit von anderen Modellgrößen beschreibt. Dieser funktionelle Zusammenhang kann durch einen algebraischen Ausdruck beschrieben werden.

**Syntax:**

```
algebraic_equation ::=   identifier  ':=' expression ';'
```

**Semantik:**

1. Der Bezeichner vor dem Zuordnungsoperator steht für eine abhängige Variable.
2. Die Wertemenge des Ausdrucks muß mit der Wertemenge der algebraischen Variable übereinstimmen oder eine Teilmenge davon bilden, d. h. ganzzahlige Ausdrücke dürfen einer reellen Größe zugeordnet werden. Ist der Ausdruck ganzzahlig und die Variable reell, wird jeder einzelne Operand des Ausdrucks als reeller Wert behandelt.
3. Ist der Ausdruck im Zeitverhalten kontinuierlich, dann muß auch die abhängige Variable als kontinuierlich deklariert sein.
4. Der Wert der abhängigen Variablen wird ohne Zeitverzug, d.h. zum selben Zeitpunkt und Takt, dem Wert des algebraischen Ausdrucks angepaßt, sobald sich dieser ändert.
5. Auf eine (im gesamten Zustandsraum gültige) Definition einer abhängigen Größe durch eine statische Beziehung darf nicht verzichtet werden. Ist der Zustandsraum durch eine Fallunterscheidung unterteilt, muß die abhängige Variable in jedem Bereich des Zustandsraums durch genau eine statische Beziehung definiert sein (siehe auch *Unterteilung des Zustandsraums*).

**Anmerkungen:**

1. In algebraischen Ausdrücken ist die Angabe von Differentialquotienten nicht möglich.
2. Die Reihenfolge der statischen Beziehungen hat keinen Einfluß auf das Modellverhalten.

**Beispiele:**

```
1.   A := 2 * X;
2.   B := X + Y;
3.   L := X > Y;
```

Sobald sich die Variable `X` oder `Y` ändert, wird der Wert der abhängigen Variablen `A` bzw. `B` bzw. `L` automatisch aktualisiert.

**Hinweis zur Effizienz**

Die statischen Beziehungen eines Modells werden in mehreren Zyklen solange ausgewertet, bis keine Auswirkungen mehr auf andere abhängige Größen festgestellt werden.

Zum gegenwärtigen Stand werden statische Beziehungen innerhalb einer Modellkomponente nicht sortiert. Aus Gründen der Effizienz ist daher zu empfehlen, abhängige Variablen in der Reihenfolge zu definieren, in der sie voneinander abhängen.

Zur Auswertung der Anweisungsfolge

```
B := A + 1;
A := 2 * X;
```

ist ein Berechnungszyklus mehr notwendig als es für die Anweisungsfolge

```
A := 2 * X;
B := A + 1;
```

erforderlich ist.

### Lösung impliziter algebraischer Gleichungen durch Iteration

Abhängige Variablen, welche durch algebraische Gleichungen gegeben sind, die sich nicht explizit auflösen lassen, können häufig durch geeignete Iteration gelöst werden.

Beispiel:

Gegeben sei die Größe $x$.

Die abhängige Größe $y$ sei durch die implizite Gleichung $y^2 - x = 0$ definiert.

Diese läßt sich in die Iterationsformel $y := y - \frac{1}{2}(y - x/y)$

überführen, die für $x > 0$ gegen $y = \sqrt{x}$ konvergiert.

Damit die Iteration nicht beim ersten Durchlauf wegen Gleitkommaüberlauf abbricht, ist die Variable $y$ mit einem Wert verschieden von Null vorzubelegen.

Wegen des iterativen Auswerteverfahrens für algebraische Gleichungen ist es daher auch möglich, eine abhängige Variable über ihre Iterationsformel zu ermitteln.

Die Iteration wird beendet, wenn die relative Abweichung aus altem und neuen Variablenwert kleiner ist als die mit dem Steuerparameter

```
EPS
```

vorgebbare Schranke.

Der Simulationslauf wird mit Fehler abgebrochen, wenn mehr als

```
MaxCyc
```

Iterationszyklen durchlaufen wurden. In einem solchen Fall ist `MaxCyc` entweder zu klein gewählt, oder die algebraischen Gleichungen konvergieren nicht gegen einen festen Wert.

#### 4.3.3.3 Differentialgleichungen

Differentialgleichungen definieren die Differentialquotienten (Ableitungen nach der Zeit) von Zustandsvariablen und beschreiben dadurch kontinuierliche Zustandsübergänge.

SIMPLEX-MDL gestattet die Formulierung von expliziten Differentialgleichungen 1. Ordnung.

Differentialgleichungen mit Ableitungen höherer Ordnung sind in ein System 1. Ordnung umzuformen.

**Syntax:**

```
differential_equations  ::=  DIFFERENTIAL EQUATION[S]
                                 derivative_assignment
                              { derivative_assignment }
                             END

derivative_assignment   ::=  identifier  " ' "  ':='
                             expression ';'
```

Differentialgleichungen werden durch die Einbettung in einen Block von Schlüsselwörtern hervorgehoben. Ein Differentialquotient wird durch ein einfaches Hochkomma unmittelbar hinter dem Bezeichner dargestellt. Jede Differentialgleichung ist mit einem Strichpunkt abzuschließen.

**Semantik:**

1. Der Bezeichner vor dem Zuordnungsoperator steht für den Differentialquotienten einer Zustandsvariablen mit reeller Wertemenge und kontinuierlichem Zeitverlauf.

2. Der Ausdruck hinter dem Zuordnungsoperator muß eine numerische Wertemenge besitzen, d. h. ganzzahlig oder reell sein.
   Ist der Ausdruck ganzzahlig, dann wird jeder einzelne Operand des Ausdrucks als reeller Wert behandelt.

3. Ist für eine kontinuierliche Zustandsvariable (abschnittsweise) kein Differentialquotient definiert, so wird dieser mit Null angenommen. Der Wert der Variablen bleibt dann unverändert (siehe auch *Unterteilung des Zustandsraums*).

4. Für den Differentialquotienten einer Zustandsvariablen darf höchstens eine Wertzuweisung angegeben werden. Ist der Zustandsraum durch eine Fallunterscheidung unterteilt, darf in jedem Abschnitt des Zustandsraums höchstens eine Wertzuweisung gültig sein (siehe auch *Unterteilung des Zustandsraums*).

5. Ein Differentialquotient, der von Null verschieden ist, führt zu einer quasi-stetigen Veränderung der Zustandsgröße durch Ausführung eines numerischen Integrationsverfahrens. Durch verschiedene Steuerparameter kann die Genauigkeit des Ergebnisses eingestellt werden.

**Anmerkungen:**

1. In algebraischen Ausdrücken ist die Angabe von Differentialquotienten nicht möglich.

2. Die Reihenfolge, in der die Differentialgleichungen angegeben werden, hat keinen Einfluß auf das Modellverhalten.

Beispiele:

1. In der Ökologie werden Räuber-Beute Beziehungen üblicherweise durch die Lotka-Volterra Gleichungen beschrieben.

```
DIFFERENTIAL EQUATIONS
X' :=   a * X  -  c * X * Y;     # Aenderungsrate der
                                 # Beutetiere
Y' :=  -b * Y  +  d * X * Y;     # Aenderungsrate der
END                              # Raeuber
```

2. Die Schwingungslehre beschreibt gedämpfte harmonische Schwingungen durch die Differentialgleichung 2. Ordnung:
   $mx'' + kx' + Dx = 0$
   Nach Umformung in ein explizites System 1. Ordnung erhält man die Formulierung für SIMPLEX-MDL:

```
DIFFERENTIAL EQUATIONS
x' :=  v;
v' :=  -(1/m) * ( k*v + D*x );
END
```

### Steuerung der Integrationsausführung

Die Integration der Differentialgleichungen wird durch numerische Algorithmen ausgeführt. Da jedes Integrationsverfahren Abweichungen von der exakten Lösung liefert, stehen dem Benutzer zahlreiche Steuerparameter zur Verfügung, durch die er auf die Integration Einfluß nehmen kann.

*IntMeth:* Integrationsverfahren (1-6)

*StepSize:* Anfangsschrittweite

*ErrEval:* 0 : keine Fehlerabschätzung
1 : Fehlerabschätzung
2 : Fehlerabschätzung und Schrittweitenanpassung

Die Schrittweitenanpassung wird durch die folgenden Steuerparameter noch näher bestimmt.

*HighRErr:* größter erlaubter relativer Fehler

*LowRErr:* kleinster erlaubter relativer Fehler

*StepMax:* größte erlaubte Schrittweite

**Integrationsverfahren**

Zur numerischen Lösung der Differentialgleichungssysteme stehen 6 verschiedene Verfahren zur Verfügung:

1. Eulersches Polygonzugverfahren
   nur geeignet bei Differentialquotienten, die sich nicht kontinuierlich verändern

2. Runge-Kutta-Verfahren 4. Ordnung
   am häufigsten eingesetztes Standardverfahren für Integration mit fester Schrittweite

3. Runge-Kutta-Fehlberg-Verfahren 6. Ordnung
   am häufigsten eingesetztes Standardverfahren für Integration mit Schrittweitenanpassung

4. implizites Runge-Kutta-Verfahren 6. Ordnung vom Gauß-Typ
   Verfahren mit besonders guten Stabilitätseigenschaften

5. Extrapolationsverfahren 6. Ordnung nach Bulirsch-Stoer
   besonders effizientes Integrationsverfahren

6. Runge-Kutta-Verfahren 4. Ordnung mit varianten Koeffizienten nach Rosenbrock-Wanner
   Verfahren zur Integration steifer Differentialgleichungen

#### 4.3.3.4 Ereignisse

Ein Ereignis beschreibt diskrete Zustandsübergänge und deren Auslösung. Durch einen diskreten Zustandsübergang verändert eine Zustandsvariable sprunghaft ihren Wert. Der neue Wert wird in dem Takt gültig, der auf die Auslösung des Ereignisses folgt.

Da die Zustandsübergänge nicht unmittelbar, sondern erst mit einer Verzögerung um einen Takt gültig werden, tritt keine gegenseitige Beeinflussung der Ereignisse auf. Die Reihenfolge, in der die Ereignisse innerhalb einer Modellkomponente angeordnet werden, ist daher sowohl für das Modellverhalten wie auch für die Effizienz der Ausführung unerheblich.

Ein Ereignis besteht aus der Auslösebedingung und den auszuführenden Zustandsübergängen.

**Syntax:**

```
event_defining_statement ::=  WHENEVER  expression
                              DO
                                 transition_statement_sequence
                              END

transition_statement_sequence ::=   transition_statement
                                  { transition_statement }

transition_statement ::=   state_variable_assignment
                         | event_region_defining_statement
                         | display_statement
```

Neben Wertzuweisungen an Zustandsvariablen gibt es eine Anweisung zur Formulierung von Fallunterscheidungen und eine Anweisung, die Meldungen auf dem Bildschirm und auf Protokolldatei ausgibt.

**Die Auslösung eines Ereignisses**

**Semantik:**

1. Der Ausdruck hinter dem Schlüsselwort WHENEVER muß einen logischen Wert liefern.
2. Zu jedem Zustand wird geprüft, ob die Auslösebedingung den Wert *wahr* aufweist. Ist das der Fall, wird das Ereignis ausgelöst, d. h. die angegebenen Zustandsübergänge eingeleitet.

**Anmerkung:**
Die Bedingung ist so anzulegen, daß sie nicht ständig wahr bleibt und somit das Ereignis nicht zum gleichen Zeitpunkt ständig aufs Neue ausgeführt wird.

**Steuerparameter:**

Die Anzahl der Ereignisfortpflanzungen, die zu einem Zeitpunkt maximal ausgeführt werden, läßt sich mit dem Steuerparameter

```
MaxTakt
```

einstellen. Werden mehr Ereignistakte benötigt, wird die Simulation mit Fehlermeldung abgebrochen. Dadurch werden endlose Ereignisfortpflanzungen, wie sie durch fehlerhafte Modellierung auftreten können, erkannt.

Beispiele:

```
1. WHENEVER   N = 0           # Ereignis wird ausgefuehrt,
                              # solange die Bedingung
                              # "N = 0" wahr bleibt

2. WHENEVER   full            # Ereignis wird ausgefuehrt,
                              # solange die logische Variable
                              # "full" wahr bleibt
```

```
3. WHENEVER  T = 10              # Ereignis wird zum Zeitpunkt
                                 # 10 fortdauernd wiederholt

4. WHENEVER (T = 10)  AND        # Ereignis wird ausgefuehrt,
            (Gate = open)        # wenn zum Zeitpunkt 10 die
   DO                            # Bedingung "Gate = open" gilt
      Gate^ := closed;
   END
```

### Wertzuweisungen an Zustandsvariablen

Eine Wertzuweisung bestimmt den Wert einer Zustandsvariablen für den nächsten Takt zum gleichen Zeitpunkt, d. h. für den Folgezustand.

**Syntax:**

```
state_variable_assignment  ::=  identifier '^' ':='
                                expression ';'
```

Das Häckchen `'^'` soll deutlich machen, daß die Zustandsvariable erst im Folgezustand den Wert des ermittelten Ausdrucks annimmt.

**Semantik**

1. Der Bezeichner vor dem Zuordnungsoperator steht für eine Zustandsvariable.
2. Die Wertemenge des Ausdrucks muß mit der Wertemenge der Zustandsvariablen übereinstimmen oder eine Teilmenge davon bilden, d.h. ganzzahlige Ausdrucke dürfen einer reellen Größe zugeordnet werden. Ist der Ausdruck ganzzahlig und die Variable reell, wird jeder einzelne Operand als reeller Wert behandelt.
3. Der Ausdruck liefert den Wert der Zustandsvariablen für den folgenden Zustand. Dieser Zustand wird noch zum selben Zeitpunkt gültig. Die einzelnen Zustände eines Zeitpunkts werden durch den *Takt* unterschieden.

4. Innerhalb eines Ereignisses, d.h. für eine bestimmte Auslösebedingung, darf für eine Zustandsvariable höchstens eine Wertzuweisung angegeben werden. Ist der Zustandsraum durch eine Fallunterscheidung unterteilt, darf in jedem Abschnitt des Zustandsraums höchstens eine Wertzuweisung gültig sein.

5. Sind Wertzuweisungen an eine Zustandsvariable über mehrere Ereignisse verteilt, so ist darauf zu achten, daß sich die Auslösemengen nicht überschneiden. Dieser Fall kann vom Compiler nicht festgestellt werden und führt erst zur Laufzeit zu einem Fehler, wenn beide Ereignisse gleichzeitig ausgelöst werden.

Beispiele:

```
1.  X^ :=  5;                    # Die Zustandsvariable X
                                 # nimmt im Folgezustand
                                 # den Wert 5 an

2.  I^ :=  J;                    # Die Werte der Zustands-
    J^ :=  I;                    # variablen I und J sind im
                                 # Folgezustand vertauscht

3.  WHENEVER  X > 0              # X wird Takt fuer Takt
    DO                           # auf den Wert Null
       X^ :=  X - 1;             # heruntergezaehlt
    END

4. WHENEVER  T = TTakt           # Alle zehn Zeiteinheiten
   DO                            # wird der Wert von X
      TTakt^  :=  T + 10;        # aktualisiert
      X^      :=  A + B;
   END

5. WHENEVER  X > A + B           # Die Ausloesemengen sind
   DO   Z^  := ... END           # nicht disjunkt.
                                 # ==> Laufzeitfehler bei zeit-
   WHENEVER  X > A - B           #     gleicher Ausloesung
   DO   Z^  := ... END
```

### Fallunterscheidungen für diskrete Zustandsübergänge

Durch die Angabe von Bedingungen kann im deklarativen Ereignisabschnitt unterschieden werden, welcher von mehreren möglichen Zustandsübergängen im aktuellen Zustand auszuführen ist.

**Syntax:**

```
event_region_defining_statement  ::=

            IF  expression
              DO transition_statement_sequence  END
           { ELSIF  expression
              DO transition_statement_sequence  END }
           [ ELSE
              DO transition_statement_sequence  END ]
```

**Semantik:**

1. Den Schlüsselwörtern `IF` und `ELSIF` müssen logische Ausdrücke folgen.
2. Die logischen Ausdrücke werden in der angegebenen Reihenfolge ausgewertet. Es ist diejenige Anweisungsfolge gültig, welche der Bedingung folgt, die als erste für `TRUE` erkannt wird.

Beispiele:

```
1.  IF  X < 0  DO  Y^ := -X; END
        ELSE   DO  Y^ :=  X; END

2.  Y^ := X;                        # Zweideutige Festlegung des
    IF  X < 0  DO  Y^ := -X;        # Folgezustands Y^ !
```

### Die Display-Anweisung

Die Display-Anweisung ermöglicht die Ausgabe von Texten und Werten auf den Bildschirm und auf die Protokolldatei.

**Syntax:**

```
display_statement  ::=   DISPLAY  '(' string
                                  { ',' expression } ')' ';'
```

Die anzugebende Zeichenkette entspricht dem Steuerstring einer `printf`-Anweisung in der Programmiersprache C. Die Anzahl und die Wertemengen der nachfolgenden Ausdrücke müssen zum Steuerstring passen.

**Achtung:** An dieser Stelle wird keine Prüfung der Semantik vorgenommen. Bei falschem Gebrauch der Anweisung kann es daher zu einem Laufzeitfehler *segmentation violation*, *bus error* oder *Gleitkommaüberlauf* kommen.

Für die verschiedenen Wertemengen sind folgende Formate zu verwenden:

| | |
|---|---|
| INTEGER: | `%ld` |
| REAL: | `%f  %e  %g` |
| LOGICAL: | `%d` |
| Aufzählungsmenge: | `%d` |
| Zeit `T`: | `%f  %e  %g` |

Die Werte von logischen Größen und von Aufzählungstypen können bislang nur in ihrer internen Darstellung *(short int)* ausgegeben werden.

Beispiele:

```
1. DISPLAY  ( " Ankunft eines Kunden \n " );

2. DISPLAY  ( " Ankunft zur Zeit T = %f \n ", T );

3. DISPLAY  ( " I = %ld  X = %f  L = %d  E = %d  \n ",
                I, X, L, E );

   #  I: INTEGER, X: REAL, L: LOGICAL, E: Aufzaehlungsmenge
```

#### 4.3.3.5 Zerlegung des Zustandsraums

Das dynamische Verhalten, das durch Differentialgleichungen, Ereignisse und algebraische Gleichungen beschrieben wird, läßt sich für verschiedene Bereiche des Zustandsraums unterschiedlich definieren.

**Syntax:**

```
region_defining_statement ::=  IF expression
                                  DO statement_sequence  END
                               { ELSIF  expression
                                  DO statement_sequence  END }
                               [ ELSE
                                  DO statement_sequence  END ]
```

**Semantik:**

1. Hinter den Schlüsselwörtern IF und ELSIF müssen logische Ausdrücke folgen.
2. Die logischen Ausdrücke werden in der angegebenen Reihenfolge ausgewertet. Es ist diejenige Anweisungsfolge gültig, welche der Bedingung folgt, welche als erste für `TRUE` erkannt wird.
3. Die Ableitung einer kontinuierlichen Größe darf in jedem Teilbereich nur durch höchstens eine Gleichung definiert sein. Ist ein Differentialquotient in einem Teilbereich nicht definiert, wird sein Wert mit Null angenommen.
4. Ein Ereignis, das innerhalb eines IF-Statements vereinbart ist, wird nur dann ausgelöst, wenn der Modellzustand innerhalb des definierten Teilbereichs liegt.
5. Eine abhängige Variable muß in jedem Teilbereich des Zustandsraumes durch genau eine Gleichung definiert sein. Das bedeutet insbesondere, daß der ELSE-Teil nicht fehlen darf, wenn abhängige Variablen innerhalb des IF-Statements definiert werden.

Beispiele:

```
1.  IF   X > 0   DO  DIFFERENTIAL EQUATION
                     Y' := a + b;
                     END
                 END

    # Im Bereich des Zustandsraumes mit X <= 0 ist der
    # Differentialquotient Y' := 0.


2.  IF  X > 0   DO   WHENEVER  Gate = closed
                     DO
                       Gate^ = open;
                     END

    # Das Ereignis wird nur dann ausgefuehrt, wenn X > 0.


3.  A :=  X + Y;
    IF  Z >= 0    DO   B :=  X;   END
          ELSE    DO   B := -X;   END

    # Die abhaengige Variable A ist im gesamten Zustandsraum
    # durch dieselbe algebraische Gleichung definiert.
    # Die abhaengige Variable B besitzt fuer die Teilbereiche
    # Z >= 0 und Z < 0 verschiedene Definitionsgleichungen.


4.  IF   Z > 0   DO   A := X + Y;
                      IF  Y > 0   DO   A:= 0;   END
                 END

    # Die abhaengige Variable A ist im Bereich des
    # Zustandsraums Z > 0 und Y > 0 zweideutig definiert.
    # Fuer den Bereich Z <= 0 fehlt die Definition ganz.
```

## 4.4 Höhere Komponenten

Höhere Komponenten setzen sich aus untergeordneten Komponenten (Subkomponenten) zusammen. Gegenseitige Einflüsse werden durch Verbindungen zwischen Subkomponenten dargestellt. Um mit der Umwelt in Kontakt treten zu können, werden Verbindungen nach außen hergestellt. Die Initialisierung von Modellgrößen aus Subkomponenten kann überschrieben werden.

Höhere Komponenten sind in 5 Abschnitte gegliedert:

1. Aufzählung der beinhalteten Komponenten (Subkomponenten)
2. Vereinbarung von Eingangselementen (Input Connections)
3. Vereinbarung von Ausgangselementen (Output Equivalences)
4. Vereinbarung der Verbindungen zwischen den Komponenten (Component Connections)
5. Vorbesetzung der Modellelemente in Subkomponenten

Die Abschnitte 2-5 sind optional.

**Syntax:**

```
high_level_component ::=   HIGH LEVEL COMPONENT  identifier
                           subcomponent_declaration
                         [ input_connection_part ]
                         [ output_equivalence_part ]
                         [ component_connection_part ]
                         [ initialization_part ]
                           END OF  identifier
```

Die Bezeichner in der ersten und letzten Zeile geben der Komponente einen Namen. Sie müssen identisch sein und mit dem Bezeichner übereinstimmen, der beim Anlegen der Komponente in der Modellerstellungsumgebung vergeben wurde.

### 4.4.1 Deklaration der Subkomponenten

Mit der Definition einer Komponente wird eine Klasse von Modellbausteinen mit gleichem dynamischen Verhalten vereinbart.

Die Subkomponenten einer höheren Komponente werden aus bereits definierten Klassen entnommen und dabei Ausprägungen (Instanzen, Inkarnationen) dieser Klasse erzeugt. Jede Ausprägung erhält ihren eigenen Namen.

**Syntax:**

```
subcomponent_declaration ::=   SUBCOMPONENT[S]
                               subcomponent
                             { ',' subcomponent }

subcomponent ::=   identifier
                 | identifier  OF CLASS  identifier
```

Die Liste der Subkomponenten wird durch Kommata getrennt.

**Semantik:**

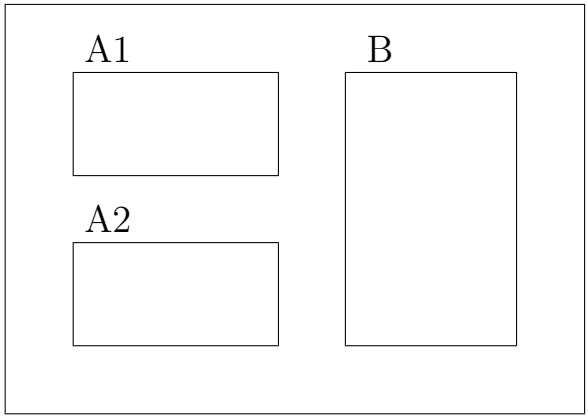
1. Ist eine Komponentenklasse nur mit einer einzigen Ausprägung in der höheren Komponente enthalten, dann genügt es, diese Komponente mit ihrem Klassennamen zu vereinbaren.
   Bei mehreren Ausprägungen einer Komponentenklasse muß für jede Ausprägung ein individueller Name vergeben werden.

2. Als Subkomponenten sind sowohl Basiskomponenten wie auch höhere Komponenten erlaubt. Dadurch wird eine Zerlegung des Modells über mehrere Hierarchiestufen möglich.

3. Die Existenz der Subkomponenten wird geprüft. Sie sollten daher bereits vor der darüberliegenden Komponente angelegt werden.

Beispiel:

```
SUBCOMPONENTS
   A1  OF  CLASS  A,
   A2  OF  CLASS  A,
   B
```




Die höhere Komponente enthält 3 Subkomponenten mit den Namen `A1`, `A2` und `B`. Die Subkomponenten `A1` und `A2` gehören derselben Komponentenklasse `A` an.

### 4.4.2 Input Connections

Eine *Input Connection* vereinbart eine Eingangsgröße einer höheren Komponente. Über diese Eingangsgröße werden Einflüsse, die von außen auf die höhere Komponente wirken, an den Eingang einer oder mehrerer Subkomponente weitergeleitet.

**Syntax:**

```
input_connection_part ::=   INPUT CONNECTION[S]
                              input_connection
                            { input_connection }

input_connection ::=   identifier '-->' input_ident ';'
                     | identifier '-->' '(' input_ident
                                          {,input_ident} ')' ';'

input_ident  ::=   identifier '.' identifier
```

Input Connections sind durch einen Strichpunkt abzuschließen.

Eine Input Connection kann die neu deklarierte Eingangsgröße mit einer oder (durch Angabe einer Liste) auch mit mehreren Eingangsgrößen aus Subkomponenten verbinden.

**Semantik:**

1. Vor dem Zuordnungspfeil `'-->'` steht der Bezeichner der Eingangsgröße, die durch diese Anweisung vereinbart wird.
2. Hinter dem Zuordnungspfeil steht der Bezeichner einer Subkomponente und (getrennt durch einen Punkt) der Bezeichner einer Eingangs- bzw. Sensorgröße aus dieser Subkomponente.
3. Die durch die Anweisung deklarierte Eingangsgröße übernimmt die Eigenschaften der zugehörigen Eingangsgröße aus der Subkomponente. Sie besitzt demnach gleichen Typ, gleiches Zeitverhalten und gleiche Wertemenge.

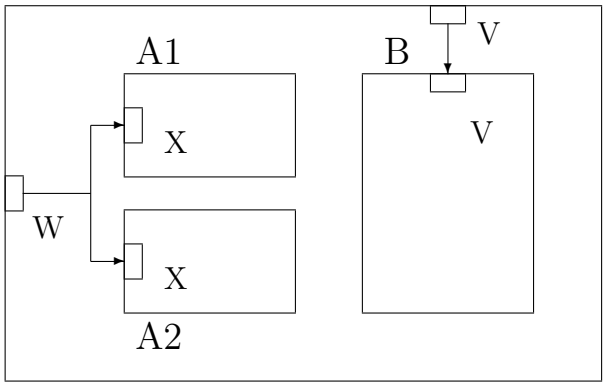
4. Verweist die deklarierte Eingangsgröße auf eine Liste von Eingangsgrößen untergeordneter Komponenten, dann müssen alle in der Liste enthaltenen Eingangsgrößen gleiche Eigenschaften besitzen.

5. Die deklarierte Eingangsgröße und alle zugeordneten Eingangsgrößen aus Subkomponenten sind in ihrem Wert zu jeder Zeit identisch.

6. Auf eine Eingangsgröße einer Subkomponente darf nur eine einzige Connection (Input oder Component Connection) gelegt werden.

**Anmerkung:**
Als Bezeichner für Eingangsgrößen dürfen auch Namen aus Subkomponenten vergeben werden.

Beispiele:




```
1.      V  -->  B.V;

2. a)   W  -->  (A1.X, A2.X);

   b)   W  -->  A1.X;
        W  -->  A2.X;
```

### 4.4.3 Output Equivalences

Eine *Output Equivalence* vereinbart eine Größe einer höheren Komponente, indem sie mit einer Größe einer Subkomponente gleichgesetzt wird. Eine solche Größe ist auch darüberliegenden Komponenten zugänglich und wird daher im folgenden als Ausgangsgröße bezeichnet.

**Syntax:**

```
output_equivalence_part ::=   OUTPUT EQUIVALENCE[S]
                                output_equivalence
                              { output_equivalence }

output_equivalence ::=  identifier ':='
                        identifier '.' identifier ';'
```

Output Equivalences sind durch einen Strichpunkt abzuschließen.

**Semantik:**

1. Vor dem Zuordnungsoperator `':='` steht der Bezeichner der Ausgangsgröße, die durch diese Anweisung vereinbart wird.

2. Hinter dem Zuordnungsoperator steht der Bezeichner einer Subkomponente und (getrennt durch einen Punkt) der Bezeichner einer Ausgangsgröße aus dieser Subkomponente. Die Angabe einer Liste ist in diesem Fall nicht erlaubt.

3. Ausgangsgrößen aus Subkomponenten können Konstanten, Zustandsvariablen und abhängige Variablen sein.

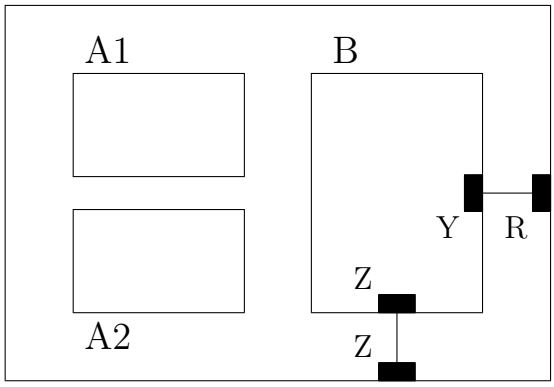
4. Die durch die Anweisung deklarierte Ausgangsgröße übernimmt alle Eigenschaften der zugehörigen Ausgangsgröße aus der Subkomponente. Sie besitzt demnach gleichen Typ, gleiches Zeitverhalten und gleiche Wertemenge.

5. Die deklarierte Ausgangsgröße und die Ausgangsgröße der Subkomponente sind in ihrem Wert zu jeder Zeit identisch. Da die Zuordnung bei einer Output Equivalence umkehrbar eindeutig ist, ist die deklarierte Größe ein Synonym für die Ausgangsgröße der Subkomponente.

6. Ausgangsgrößen dürfen nur in einer einzigen Output Equivalence auftreten.

**Anmerkung:**
Als Bezeichner für Ausgangsgrößen dürfen auch Namen aus Subkomponenten vergeben werden.

Beispiele:




```
1.)   R  :=  B.Y ;

2.)   Z  :=  B.Z ;
```

### 4.4.4 Component Connections

Eine *Component Connection* verbindet eine Ausgangsgröße einer Subkomponente mit einer Eingangsgröße einer anderen oder auch der gleichen Subkomponente.

**Syntax:**

```
component_connection_part ::=   COMPONENT CONNECTION[S]
                                  comp_connection
                                { comp_connection }

comp_connection ::=

            output_element  '-->'  input_element ';'

          | output_element  '-->'  '(' input_element
                                   {',' input_element} ')' ';'



output_element ::=    identifier

                    | identifier '.' identifier

input_element  ::=    identifier '.' identifier
```

Component Connections sind durch einen Strichpunkt abzuschließen.

Eine Component Connection kann eine Ausgangsgröße durch Angabe einer Liste auch mit mehreren Eingangsgrößen verbinden.

**Semantik:**

1. Vor dem Zuordnungspfeil `'-->'` steht die Ausgangsgröße einer Subkomponente. Diese Größe kann

   (a) durch einen einzelnen Bezeichner angegeben werden, wenn für sie bereits eine Output Equivalence vereinbart wurde, oder

   (b) durch einen Selektionspfad, der sich aus dem Namen der Subkomponente und (getrennt durch einen Punkt) dem Namen der Ausgangsgröße zusammensetzt.

2. Ausgangsgrößen können Konstanten, Zustandsgrößen oder abhängige Größen sein.

3. Hinter dem Zuordnungspfeil steht die Eingangsgröße einer Subkomponente oder eine Liste von Eingangsgrößen aus Subkomponenten. Sie werden durch einen Selektionspfad angegeben, der sich aus dem Namen der Subkomponente und (getrennt durch einen Punkt) dem Namen der Eingangsgröße zusammensetzt.

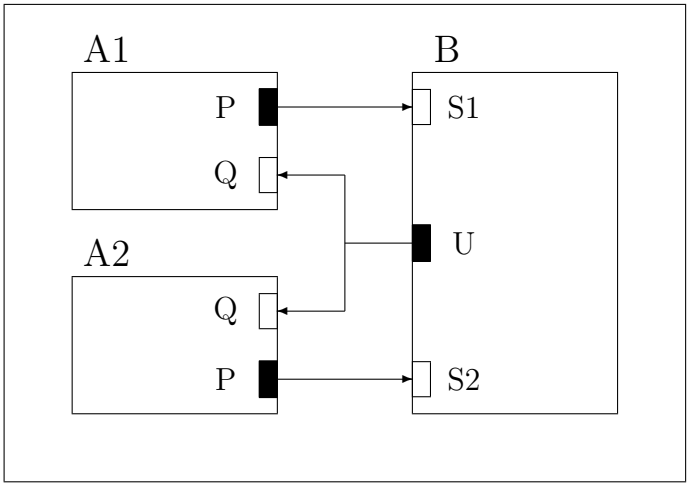
4. Die miteinander verbundenen Größen (Eingangs- und Ausgangsgröße) müssen gleichen Zeitverlauf und gleiche Wertemenge besitzen. Ist der Zeitverlauf der Eingangsgröße kontinuierlich, darf der Zeitverlauf der Ausgangsgröße auch diskret sein.

5. Die Eingangsgröße(n) besitzen zu jeder Zeit den gleichen Wert wie die zugeordnete Ausgangsgröße.

6. Auf eine Eingangsgröße darf nur eine einzige Connection (Input Connection oder Component Connection) gelegt werden.

**Anmerkung:**
Von einer Ausgangsgröße dürfen beliebig viele Verbindungen wegführen.

Beispiele:




```
1.      A1.P  -->  B.S1;
        A2.P  -->  B.S2;

2. a)   B.U  -->  A1.Q;
        B.U  -->  A2.Q;

   b)   B.U  -->  (A1.Q, A2.Q);
```

### 4.4.5 Initialisierung

Die Initialisierung, die in einer Basiskomponente vorgenommen wird, gilt für alle Ausprägungen dieser Klasse. In der höheren Komponente ist es nun möglich, diese Vorbesetzung zu überschreiben und jeder Ausprägung ihre individuellen Anfangswerte mitzugeben.

**Syntax:**

```
initialization_part ::=    INITIALIZE
                           initialization { initialization }

initialization ::=   output_element ':=' value ';'

output_element   ::=   identifier
                     | identifier '.' identifier
```

**Semantik:**

1. Vor dem Zuordnungsoperator `':='` steht die Ausgangsgröße einer Subkomponente. Diese Größe kann
   (a) durch einen einzelnen Bezeichner angegeben werden, wenn für sie bereits eine Output Equivalence vereinbart wurde, oder
   (b) durch einen Selektionspfad, der sich aus dem Namen der Subkomponente und (getrennt durch einen Punkt) dem Namen der Ausgangsgröße zusammensetzt.
2. Ausgangsgrößen können Konstanten, Zustandsgrößen oder abhängige Größen sein.
3. Hinter dem Zuordnungsoperator steht ein Wert, der in der Wertemenge der Ausgangsgröße enthalten sein muß.
4. Jeder Ausgangsgröße darf nur einmal ein Wert zugeordnet werden.
5. Der zugeordnete Wert überschreibt die Initialisierung der darunterliegenden Komponente.

Beispiele:

```
1.    A.I  :=  15;

2.    A.Y  :=  7.5 E-2;

3.    A.L  :=  TRUE;

4.    A.Status  :=  busy;        # Aufzaehlungstyp

5.    Out       :=  3.2;         # Output Equivalence
```

## 4.5 Standardfunktionen

### 4.5.1 Reellwertige Funktionen mit reellen Argumenten

x und y seien reelle Argumente.

**a) Stetig differenzierbare (kontinuierliche) Funktionen**

| | | |
|---|---|---|
| SQE (x) | Quadrat | $x^2$ |
| SQRT (x) | Quadratwurzel | $\sqrt{x}$ |
| POW (x,y) | Potenz | $x^y$ |
| ROOT (x,y) | Wurzel | $\sqrt[y]{x}$ |
| EXP (x) | Exponentialfunktion | $e^x$ |
| LN (x) | natürlicher Logarithmus | $\ln x$ |
| LOG10 (x) | Logarithmus zur Basis 10 | $\lg x$ |
| LOG (x,y) | Logarithmus zur Basis y | ${}^y\log x$ |
| SIN (x) | Sinus | $\sin x$ |
| COS (x) | Cosinus | $\cos x$ |
| TAN (x) | Tangens | $\tan x$ |
| ASIN (x) | Arcussinus | $\arcsin x$ |
| ACOS (x) | Arcuscosinus | $\arccos x$ |
| ATAN (x) | Arcustangens | $\arctan x$ |
| ATAN2 (x,y) | erweiterter Arcustangens | $\arctan(x/y)$ für $y \geq 0$<br>$\arctan(x/y) + \pi$ für $y < 0$ |
| SINH (x) | Sinus hyperbolicus | $\sinh x = (e^x - e^{-x})/2$ |
| COSH (x) | Cosinus hyperbolicus | $\cosh x = (e^x + e^{-x})/2$ |
| TANH (x) | Tangens hyperbolicus | $\tanh x = \frac{\sinh x}{\cosh x}$ |

### b) Nicht stetig differenzierbare (diskontinuierliche) Funktionen

| | | |
|---|---|---|
| ABS (x) | Betrag | $\|x\|$ |
| MOD (x,y) | Divisionsrest bei ganzzahliger Division x/y (ungerade Funktion) | $x \bmod y = x - y$ DIV (x,y) |
| MOD2(x,y) | Divisionsrest bei ganzzahliger Division x/y (gerade Funktion) | $x$ mod2 $y = x - y$ DIV2 (x,y) $= \begin{cases} +(x \bmod y) & \text{für } x \geq 0 \\ -(x \bmod y) & \text{für } x < 0 \end{cases}$ |
| MAX (x,y) | Maximum von x und y | |
| MIN (x,y) | Minimum von x und y | |

### c) Stufenförmige (diskrete) Funktionen

| | | |
|---|---|---|
| SIGN (x) | Signum | $+1$ für $x \geq 0$<br>$-1$ für $x < 0$ |
| TRUNC (x) | Abschneiden der Nachkommastellen | |
| RND (x) | Runden auf die nächstgelegene ganze Zahl | TRUNC $(x + 0.5)$ für $x \geq 0$<br>TRUNC $(x - 0.5)$ für $x < 0$ |
| FLOOR (x) | Abrunden auf nächst niedrigere ganze Zahl | |
| CEIL (x) | Aufrunden auf nächst höhere ganze Zahl | |
| DIV (x,y) | Division x/y mit abgeschnittenem ganzzahligem Ergebnis | TRUNC $(x/y)$ |
| DIV2(x,y) | Division x/y mit abgerundetem ganzzahligen Ergebnis | FLOOR $(x/y)$ |

### 4.5.2 Funktionen mit ganzzahliger Wertemenge und reellen Argumenten (Konvertierungsfunktionen)

| | |
|---|---|
| ITRUNC (x) | Abschneiden der Nachkommastellen |
| IRND (x) | Runden auf die nächstgelegene ganze Zahl |
| IFLOOR (x) | Abrunden auf nächst niedrigere ganze Zahl |
| ICEIL (x) | Aufrunden auf nächst höhere ganze Zahl |

### 4.5.3 Funktionen mit ganzzahliger Wertemenge und ganzzahligen Argumenten

k und n seien ganzzahlige Argumente.

| | | |
|---|---|---|
| IABS (k) | Betrag | $\|k\|$ |
| ISIGN (k) | Signum | $+1$ für $k \geq 0$<br>$-1$ für $k < 0$ |
| IDIV (k,n) | Division k/n mit ganzzahligem Ergebnis | |
| IMOD (k,n) | Divisionsrest bei ganzzahliger Division k/n | |
| IMAX (k,n) | Maximum von k und n | |
| IMIN (k,n) | Minimum von k und n | |

## 4.6 Zusammenfassung der Syntax

### (a) Erläuterungen zur erweiterten Backus-Naur-Form (BNF)

- Schlüsselwörter sind groß, Produktionen klein geschrieben.
- Sonderzeichen und Folgen von Sonderzeichen sind in einfache Anführungszeichen eingeschlossen.
- Produktionen in eckigen Klammern brauchen gar nicht auftreten oder erscheinen genau einmal.
- Produktionen in geschweiften Klammern brauchen gar nicht auftreten oder können beliebig häufig aneinander gereiht werden.
- Das Startsymbol ist "model".

### (b) Liste der Schlüsselwörter

```
AND,
BASIC, BEHAVIOUR, BESSEL, BY,
CLASS, CONNECTION, CONSTANT, CONTINUOUS, COMPONENT, CUBIC,
DECLARATION, DEFINITION, DEPENDENT, DIFFERENTIAL, DISCRETE,
DISPLAY, DO, DYNAMIC,
ELEMENTS, ELSE, ELSIF, EQUATION, EQUIVALENCE,
FALSE, FUNCTION,
HIGH,
IF, INITIALIZE, INPUT, INT, INTEGER, INTERPOLATION,
LEVEL, LINEAR, LOGICAL, LOCAL,
NOT,
OF, ON, OR, OUTPUT,
REAL,
SENSOR, SET, SPLINE, STATE, SUBCOMPONENT,
T, TABULAR, TRUE,
VALUE, VARIABLE,
WHENEVER
```

## (c) Liste der Sonderzeichen

| | |
|---|---|
| Komma: | ’,’ |
| Strichpunkt: | ’;’ |
| runde Klammern: | ’(’, ’)’ |
| Verbindungsoperator: | ’-->’ |
| Zuordnungsoperator: | ’:=’ |
| Doppelpunkt: | ’:’ |
| Gleichheitsrelationen: | ’=’, ’<>’ |
| Ordnungsrelationen: | ’>’, ’>=’, ’<’, ’<=’ |
| Ableitungssymbol: | " ’ " |
| Ereignisoperator: | ’^’ |
| Rechenoperatoren: | ’+’, ’-’, ’*’, ’/’ |
| Selektor: | ’.’ |

## (d) Notation für die lexikalischen Einheiten Bezeichner, Zahlen und Zeichenketten

```
identifier ::=   letter { [ '_' ] letter_or_digit }

letter_or_digit ::=   letter | digit

letter ::=       A | B | .. | Z
               | a | b | .. | z

digit ::=        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

unsigned_number ::=   decimal [[ blank ] exponent ]

decimal ::=    digit_list
             | digit_list '.' digit_list

exponent ::=     'E' sign digit_list

digit_list ::=   digit { digit }

blank ::=    ' '

string ::=  ' " '  { ascii_character } ' " '
```

## (e) Sprachsyntax

```
model  ::=   high_level_component
           | basic_component

high_level_component ::=     HIGH LEVEL COMPONENT   identifier
                             subcomponent_declaration
                           [ input_connection_part ]
                           [ output_equivalence_part ]
                           [ component_connection_part ]
                           [ initialization_part ]
                             END OF   identifier

subcomponent_declaration ::=  SUBCOMPONENT[S]
                               subcomponent  {',' subcomponent }

subcomponent ::=    identifier
                  | identifier   OF CLASS   identifier

input_connection_part  ::=   INPUT CONNECTION[S]
                             input_connection { input_connection }

input_connection  ::=    identifier  '-->'  input_ident ';'
                       | identifier  '-->'  '(' input_ident
                                          { ',' input_ident } ')' ';'

input_ident   ::=   identifier '.' identifier

output_equivalence_part  ::=   OUTPUT EQUIVALENCE[S]
                               output_equivalence
                             { output_equivalence }

output_equivalence  ::=    identifier ':='
                           identifier '.' identifier ';'

component_connection_part ::=    COMPONENT CONNECTION[S]
                                 comp_connection
                               { comp_connection }

comp_connection ::=  output_element '-->' input_element ';'
                   | output_element '-->' '('input_element
                                        { ','input_element }')' ';'

initialization_part ::=    INITIALIZE
                           initialization { initialization }

initialization ::=   output_element ':=' value ';'
```

```
output_element ::=    identifier
                    | identifier '.' identifier

input_element  ::=    identifier '.' identifier

================================================================

basic_component ::=    BASIC COMPONENT   identifier
                     [ local_definitions ]
                       declaration_of_elements
                     [ dynamic_behaviour ]
                       END OF   identifier

local_definitions ::=    LOCAL DEFINITION[S]
                       { enumeration_set  }
                       { tabular_function }

enumeration_set ::=    VALUE SET  identifier  ':'
                       '(' enum_value_list ')'

enum_value_list ::=    identifier ',' identifier { ',' identifier }
                     | identifier '<' identifier { '<' identifier }

tabular_function ::=   TABULAR FUNCTION
                       identifier '(' REAL '-->'REAL ')'
                       CONTINUOUS
                       BY  interpolation_method  INTERPOLATION
                       ON '(' number_list ')' '-->'
                       '(' number_list ')'

interpolation_method ::=   LINEAR
                         | CUBIC BESSEL
                         | CUBIC SPLINE

number_list ::=   number ',' number {',' number }

----------------------------------------------------------------

declaration_of_elements ::=   DECLARATION OF ELEMENTS
                            [ list_of_constants ]
                            [ list_of_state_variables ]
                            [ list_of_dependent_variables ]
                            [ list_of_sensor_variables ]
```

```
list_of_constants ::=             CONSTANT[S]
                                  quantity {',' quantity }

list_of_state_variables ::=       STATE VARIABLE[S]
                                 [[DISCRETE]
                                  quantity { ',' quantity } ]
                                 [ CONTINUOUS
                                  quantity { ',' quantity } ]

list_of_dependent_variables ::=   DEPENDENT VARIABLE[S]
                                 [[DISCRETE]
                                  quantity { ',' quantity } ]
                                 [ CONTINUOUS
                                  quantity { ',' quantity } ]

list_of_sensor_variables ::=      SENSOR VARIABLE[S]
                                 [[DISCRETE]
                                  quantity { ',' quantity } ]
                                  [CONTINUOUS
                                  quantity { ',' quantity } ]

quantity ::=   identifier '(' range ')'  [ ':=' value ]

range ::=    INT[EGER]
           | REAL
           | LOGICAL
           | identifier

value ::=    number
           | logic_value
           | identifier

-----------------------------------------------------------------

dynamic_behaviour ::=    DYNAMIC BEHAVIOUR
                         statement_sequence

statement_sequence ::=     statement  { statement }

statement ::=   algebraic_equation
              | differential_equations
              | event_defining_statement
              | region_defining_statement
```

```
algebraic_equation ::=   identifier  ':=' expression ';'

differential_equations ::=  DIFFERENTIAL EQUATION[S]
                              derivative_assignment
                            { derivative_assignment }
                              END

derivative_assignment ::=   indexed_identifier " ' "
                            ':=' expression ';'

region_defining_statement ::=  IF  expression
                                 DO  statement_sequence   END
                               { ELSIF  expression
                                 DO  statement_sequence   END }
                               [ ELSE
                                 DO  statement_sequence   END ]

event_defining_statement ::=   WHENEVER   expression
                               DO
                                  transition_statement_sequence
                               END

transition_statement_sequence  ::=  transition_statement
                                    { transition_statement}

transition_statement  ::=   state_variable_assignment
                          | event_region_defining_statement
                          | display_statement

state_variable_assignment ::=   identifier '^' ':=' expression ';'

event_region_defining_statement  ::=
            IF  expression
              DO  transition_statement_sequence  END
          { ELSIF  expression
              DO  transition_statement_sequence  END }
          [ ELSE
              DO  transition_statement_sequence  END ]

display_statement  ::=   DISPLAY '(' string { ',' expression } ')' ';'

-------------------------------------------------------------------------

expression ::=    simple_expression
                | comparison
```

```
comparison ::=   simple_expression  compare_op  simple_expression

compare_op ::=   '<' |  '>' |  '=' |  '<>' |  '<=' |  '>='

simple_expression ::=   [ sign ]  term  { add_op term }

sign ::=     '+' | '-'

term ::=    factor
          | term mul_op factor

add_op ::=  '+' |  '-' |  OR

mul_op ::=  '+' |  '/' | AND

factor ::=    identifier
            | unsigned_number
            | logic_value
            | T
            | function_execution
            | '(' expression ')'
            | NOT factor

function_execution ::=   identifier '(' argument_list ')'

argument_list ::=   expression  {',' expression }

number ::=    unsigned_number
            | sign unsigned_number

logic_value ::=    TRUE
                 | FALSE
```

# 5 Die Simulationsumgebung

Wir haben eine formale Sprache kennengelernt, die es uns ermöglicht, einzelne Modellkomponenten zu beschreiben.

Die Quelltexte dieser Komponenten müssen in Dateien abgelegt und verwaltet werden. Übersetzungsläufe müssen angestoßen werden, um aus der Modellbeschreibung ein fertiges Simulationsprogramm zu erzeugen. Diese Aufgaben übernimmt die Modellerstellungsumgebung des Simulationssystems.

Zur Durchführung von Experimenten steht eine Experimentierumgebung zur Verfügung, welche die nötigen Vorbesetzungen von Modellgrößen vornimmt, einzelne Simulationsläufe ausführt und die Ergebnisse in Dateien ablegt und verwaltet.

Für die Darstellung und Auswertung der Ergebnisse stellt die Simulationsumgebung weitere Funktionen bereit.

Alle diese Funktionen wurden in das Simulationssystem SIMPLEX-II integriert, das sich dem Benutzer mit einer einheitlichen Bedienoberfläche präsentiert. Die umseitige Abbildung zeigt einen groben Überblick über das Zusammenwirken der einzelnen Bestandteile des Systems.

Während der Durchführung der Experimente unterhält das Simulationsprogramm eine enge Kopplung an die Experimentierumgebung über Datenkanäle. Es wird dadurch Bestandteil eines Prozeßsystems und wir sprechen daher im folgenden besser von Simulationsprozeß und Experimentierprozeß.

Im Verlauf dieses Kapitels wird der MDL-Compiler, die Modellerstellungsumgebung, die Experimentierumgebung und die Kommunikation zwischen Experimentierumgebung und Simulationsprozeß behandelt. Es wird dabei der Weg vom spezifizierten Modell bis zum Ablauf des Simulationsprozesses verfolgt.

Die Umgebung des Simulationsprozesses wird nur insoweit beschrieben, als es zum Verständnis der Arbeitsweise des Simulationsprozesses erforderlich ist. Eine detaillierte Beschreibung findet sich in der Dissertation von K.-J. Langer [Lang 90].

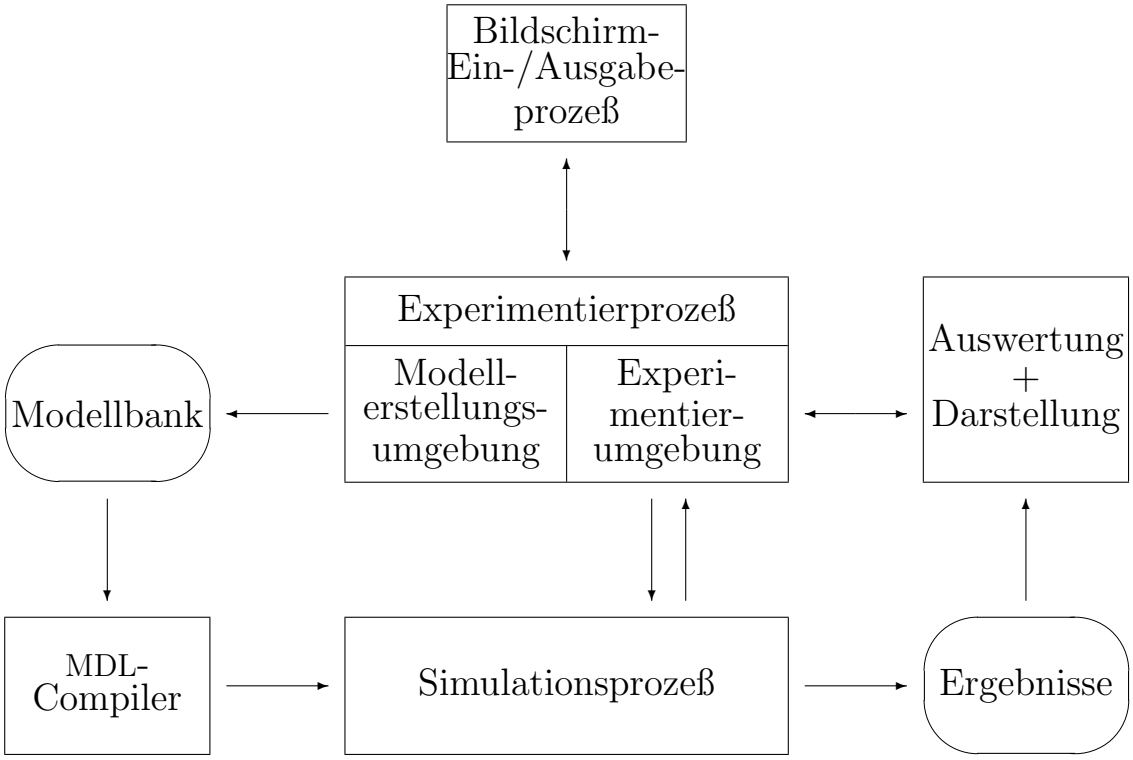



Abb. 5-1: Der Aufbau des Simulationssystems SIMPLEX-II

## 5.1 Der MDL-Compiler

Der MDL-Compiler erzeugt aus dem Quelltext einer Modellkomponente eine C-Funktion sowie eine Reihe von Dateien mit Symbolinformation.

Er wird spezifiziert durch die Sprachbeschreibung aus Kapitel 4 (lexikalische, syntaktische und semantische Analyse) und den zu generierenden C-Code, der im Kapitel 6 bei der Abhandlung des Simulationsprozesses besprochen wird.

Die Implementierung des MDL-Compilers wurde unter Zuhilfenahme der UNIX-Werkzeuge *lex* und *yacc* durchgeführt. *Lex* erzeugt einen endlichen Automaten zur lexikalischen Analyse und *yacc* generiert einen Parser, der einen Wertestack verwaltet, dessen Information für die semantische Analyse ausgewertet wird. Die semantische Analyse und Codegenerierung beruht auf Standardtechniken, wie sie im Compilerbau seit langem üblich sind.

Die Implementierung des MDL-Compilers weist daher keinen neuen wissenschaftlichen Gehalt auf. Alles wesentliche ist in der Literatur [SchFr 85, Joh 78, LeSch 78, Petsch 86, TreSor 85, WaiGo 85] nachzulesen.

Wir beschränken uns daher an dieser Stelle auf die Beschreibung der Aufgaben, die der MDL-Compiler zu erfüllen hat.

Der Quelltext einer Modellkomponente wird in einer Datei erwartet, welche die Endung `'.mdl'` besitzt. Der Name dieser Datei bezeichnet die Version der Komponente. Für jede Komponente existiert ein Directory, das den Namen der Komponente trägt. Dieses wiederum ist im Directory der Modellbank untergebracht. Die Quelltextdatei einer Modellkomponente besitzt daher den Pfadnamen

```
<modelbank> / <component> / <version>.mdl  .
```

Die Bedeutung dieser Dateistruktur wird im nächsten Abschnitt erläutert.

Der MDL-Compiler zerfällt in die beiden Programme MDL_C und MDL_SYM. Der Aufruf der Programme erfolgt aus dem Directory der Modellbank und besitzt folgende Syntax:

```
MDL_C    <component>  <version>  { <option> }

         <option> ::=  -sub | -prt | -sym | -noerr | -debug

MDL_SYM  <component>  <version>
```

Die erzeugten Dateien werden im Directory der Komponente abgelegt.

### Das Programm MDL_C

Ohne die Angabe von Optionen erledigt MDL_C die folgenden Aufgaben:

- lexikalische, syntaktische und semantische Analyse der Quelldatei `<component>/<version>.mdl`
- Erzeugung des C-Codes auf Datei `<version>.c`
- Erzeugung von Symbolinformation auf Datei `<version>.sym`
- Erstellung eines Listings mit Fehlermeldungen, das auf die Quelldatei `<version>.mdl` kopiert wird.

Durch die Angabe von Optionen kann dieser Leistungsumfang verändert bzw. erweitert werden:

`-sub:` liefert Anzahl und Namen der Subkomponentenklassen auf Datei `<version>.sub`.

(bei höheren Komponenten entfällt die semantische Analyse, Codeerzeugung und Symbolinformation)

`-prt:` Ausgabe des Listings auf Datei `<version>.prt` statt auf `<version>.mdl`

`-sym:` Ausgabe einer Symboltabelle am Ende des Listings

`-ln:` Ausgabe von Zeilennummern im Listing (wird nur in Verbindung mit der Option `-prt` ausgeführt)

`-noerr:` Unterdrückung der Fehlermeldungen im Listing

`-debug:` Ausgabe eines Debug-Protokolls auf Datei `yydebug`

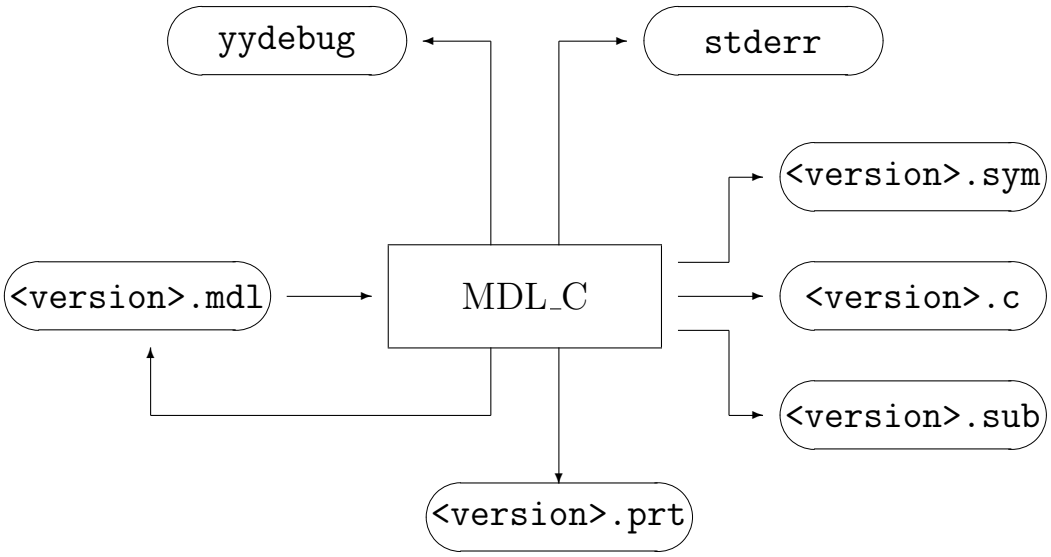



Abb. 5.1-1: Ein-/Ausgabedateien von MDL_C

*Die Fehlerausgabe*

Alle Fehlermeldungen werden auf dem UNIX-Standardkanal *stderr* ausgegeben. Dies hat den Vorteil, daß keine Datei geöffnet und beschrieben werden muß und daher die Ausgabe immer gelingt. Bei Bedarf kann diese Ausgabe auf andere Medien umgelenkt werden.

Fehler, die im Quelltext gefunden werden, werden darüber hinaus auch im Listing protokolliert. Um Compilermeldungen kenntlich zu machen, werchen sie durch das Zeichen `'$'` in der ersten Spalte gekennzeichnet.

Eine erneute Übersetzung entfernt die Fehlermeldung wieder aus dem Quelltext. Auf diese Art ist es möglich, daß der Benutzer direkt das Listing mit den Fehlermeldungen editiert.

Fehlermeldungen haben das folgende Format:

auf Standardausgabe:

```
[Fehler <Nr>]  Zeile  <Nr>  bei  <Symbol>
               <Fehlertext>
```

im Listing:

```
$  <Markierung der fehlerhaften Stelle(n) durch '|'>
$  [Fehler <Nr>]  <Fehlertext>
$  -------------
$
```

Um nach einem Fehler im Quelltext Kaskaden von Fehlermeldungen zu vermeiden, werden verschiedene Maßnahmen ergriffen:

1. Syntaxfehler werden erst dann wieder protokolliert, nachdem 3 Eingabesymbole korrekt erkannt wurden.
2. Nach einem Syntaxfehler wird die Prüfung auf semantische Fehler abgeschaltet.
3. Nach einem semantischen Fehler im Deklarationsteil wird die Prüfung auf semantische Fehler im Dynamikteil abgeschaltet.
4. Es werden maximal 10 Fehler protokolliert.

Diese Vorgehensweise führt gelegentlich dazu, daß nicht alle erkennbaren Fehler protokolliert werden. Andererseits zeigt die Erfahrung, daß selten ein Benutzer in der Lage ist, zwischen Folgefehlern und neuen Fehlern zu unterscheiden. Wenn der Quelltext mehrere Fehler enthält, benötigen die meisten Benutzer in aller Regel mehrere Versuche, bis alle Fehler beseitigt sind.

*Die externe Symboltabelle*

Die Datei `<version>.sym` enthält einen Auszug der internen Symboltabelle. Sie enthält einige Dimensionierungsangaben

- Anzahl Crossings
- Anzahl Regions
- Anzahl Zeitvergleiche

sowie Informationen über jeden Bezeichner der analysierten Komponente:

- Name des Bezeichners
- interne Schreibweise des Bezeichners
- Name der Wertemenge
- Nummer des Typs
- Nummer der Wertemenge
- Nummer eines Aufzählungswertes bzw.
  Anzahl der Aufzählungswerte

### Das Programm MDL_SYM

Für die aktuelle Version einer Modellkomponente bereitet das Programm MDL_SYM die Symbolinformation aus der Datei `<version>.sym` für verschiedene Anwendungen auf. Es erstellt die Dateien

| | |
|---|---|
| `symtab:` | Kopie der Datei `<version>.sym` |
| `mod_struct.h:` | Struktur der Modelldaten |
| `offset.dat:` | Symboltabelle für Experimentierprozeß |
| `offset.incl:` | C-Code zur Ermittlung der Offset-Adressen |
| `dimensions.h:` | Dimensionierung globaler Modellgrößen |

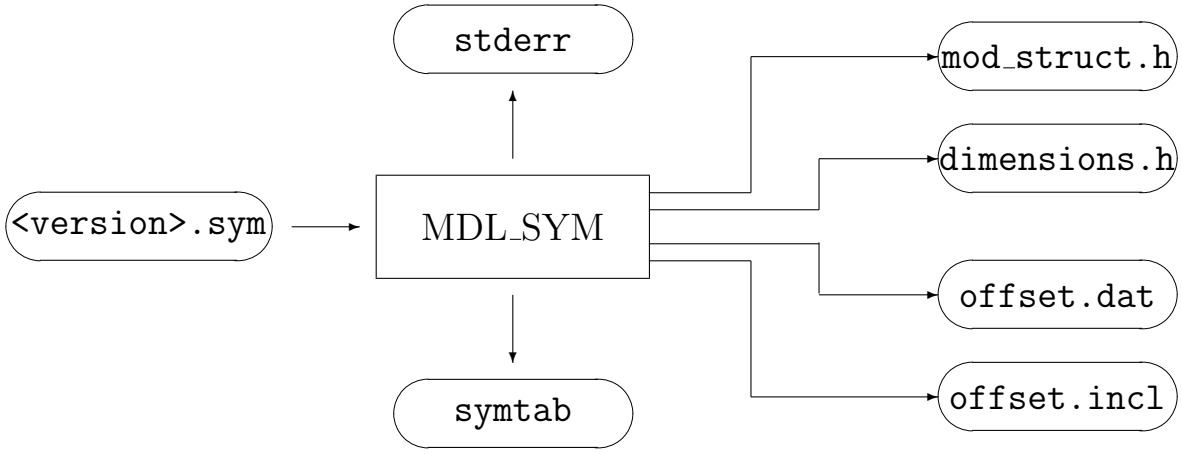



Abb. 5.1-2: Ein-/Ausgabe von MDL_SYM

Die Datei `symtab` macht die Symboltabelle für eine Konsistenzprüfung mit darüberliegenden Komponenten zugänglich. Die Datei `mod_struct.h` enthält die C-Datenstruktur der Modellkomponente. Ihr Inhalt wird im Kapitel 6 besprochen.

Die Datei `dimensions.h` enthält Präprozessor-Makros, die bei der Übersetzung des Koppelprogramms `Model.c` ausgewertet werden. Diese Makros liefern den Namen der Modellkomponente in verschiedenen Schreibweisen sowie zur Dimensionierung die

- Anzahl der Komponentenklassen
- Anzahl der Namen
- Anzahl der Aufzählungstypen
- Anzahl der Aufzählungswerte
- Anzahl der Basiskomponenten
- Anzahl der kontinuierlichen Zustandsvariablen
- Anzahl der Crossings
- Anzahl der Regions
- Anzahl der Zeitvergleiche

Diese Dimensionierungen beziehen sich nicht nur auf die betreffende Modellkomponente, sondern schließen alle in ihr enthaltenen Subkomponenten ein.

Die Datei `offset.incl` enthält den C-Code, der dem Experimentierprozeß die Adressen der Modellgrößen und Modellkomponenten mitteilt.

Die Datei `offset.dat` enthält die Informationen, die der Experimentierprozeß benötigt, um eine Verwaltung der Modelldaten anzulegen. Es sind darin Angaben zur Dimensionierung sowie Informationen über die einzelnen Symbole der Komponente enthalten. Sie umfassen:

- Anzahl der Komponentenklassen des Modells
- Anzahl der im Modell vorkommenden Namen
- Anzahl der Aufzählungsmengen
- Anzahl der Aufzählungswerte

Für jeden Aufzählungstyp:

- Typkennung
- Name
- Anzahl der Werte
- Namen der Werte

Für jede Subkomponente:

- Typkennung
- Namen der Subkomponente
- zugehörige Komponentenklasse

Für jede Modellgröße:

- Typkennung
- Name

Für kontinuierliche Zustandsgrößen wird ein zweiter Eintrag vorgenommen. Dieser enthält den Namen, der um das Ableitungssymbol (z.B. `X'`) ergänzt ist.

Als Typkennung wird eine dreistellige Zahl vergeben, die in der Header-Datei `rts_struct.h` definiert ist und sich zusammensetzt aus:

`<1/0: kontinuierlich ja/nein> <Verwendung> <Wertemenge>`

## 5.2 Die Modellerstellungsumgebung

Die Modellerstellungsumgebung gibt dem Anwender ein hierarchisches Ordnungsschema vor, in das angelegte Modellkomponenten eingegliedert werden:

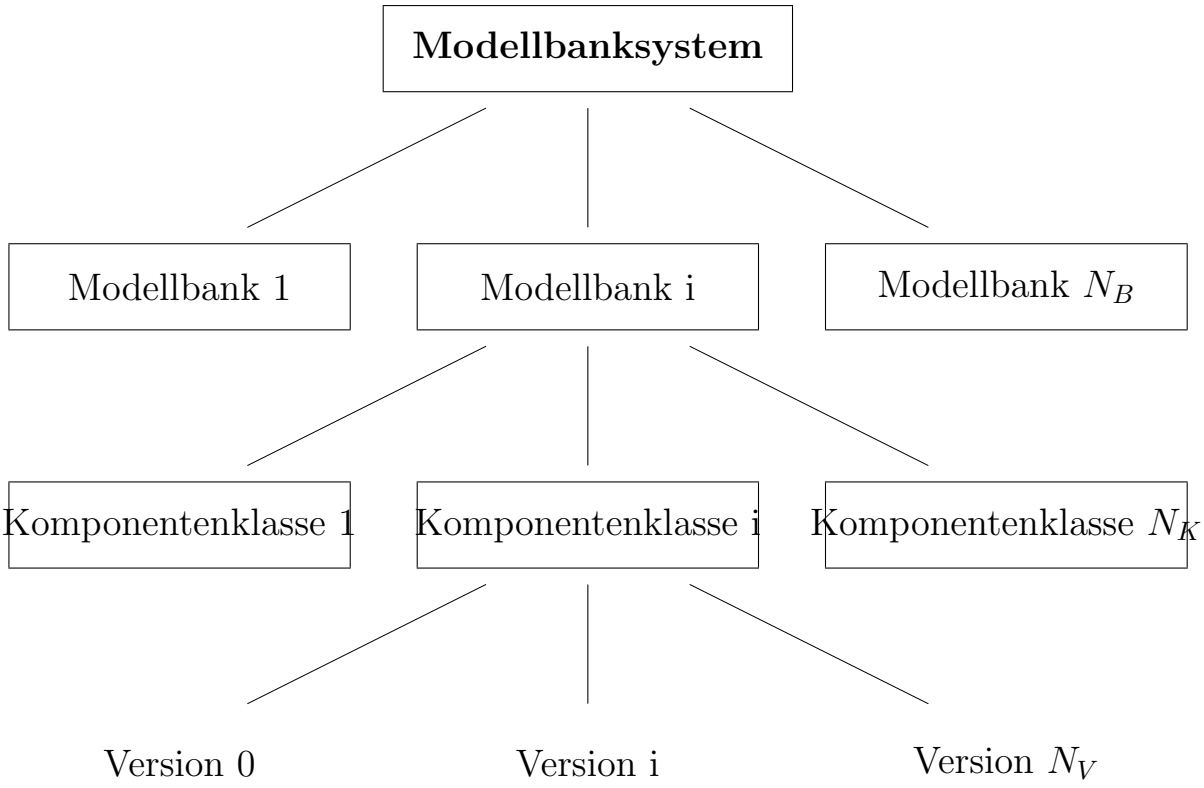



Abb. 5.2-1: Ordnungsschema der Modellerstellungsumgebung

Ein Modellbanksystem gliedert sich in mehrere Modellbanken. Eine Modellbank besteht aus Klassen von Modellkomponenten. Jede Komponentenklasse kann mit mehreren Versionen vertreten sein. Diese unterste Ebene enthält die Dateien, in denen Modellkomponenten in SIMPLEX-MDL definiert sind. Die Dateien tragen die Bezeichnung `<version>.mdl`.

Versionen, die zu einer Komponentenklasse gehören, zeichnen sich dadurch aus, daß sie den gleichen Namen tragen und die gleichen Schnittstellen zur Umgebung besitzen. Diese Eigenschaften ermöglichen jederzeit einen Austausch der Modellkomponente durch eine andere Version derselben Klasse. Für die erste Version einer Komponentenkasse kann auf die Angabe eines Versionsnamens verzichtet werden. Es wird dann standardmäßig der Name `Version0.mdl` vergeben.

Klassen von Modellkomponenten, die Bestandteil des gleichen Modells sind, müssen in einer gemeinsamen Modellbank abgelegt werden.

Modellbanken werden schließlich zu einem Modellbanksystem zusammengefaßt. Während der Benutzer Modellbanken, Komponentenklassen und Versionen selbst anlegen und mit Namen versehen kann, werden auf der obersten Ebene die beiden Modellbanksysteme *priv* und *pub* vom System zur Verfügung gestellt.

Zum Aufbau dieser Dateistruktur werden dem Benutzer Kommandos zum

- Anlegen
- ndern
- Kopieren
- Umbenennen
- Löschen

von Modellbanken und Modellkomponenten angeboten.

Die einzelnen Klassen von Modellkomponenten, die in einer Modellbank abgelegt sind, stehen in Abhängigkeit zueinander. Jede höhere Komponente enthält Subkomponenten, die einer bestimmten Komponentenklasse angehören, und wiederum Subkomponenten beinhalten können.

Die folgende Abbildung zeigt beispielhaft Abhängigkeiten zwischen Komponentenklassen einer Modellbank.

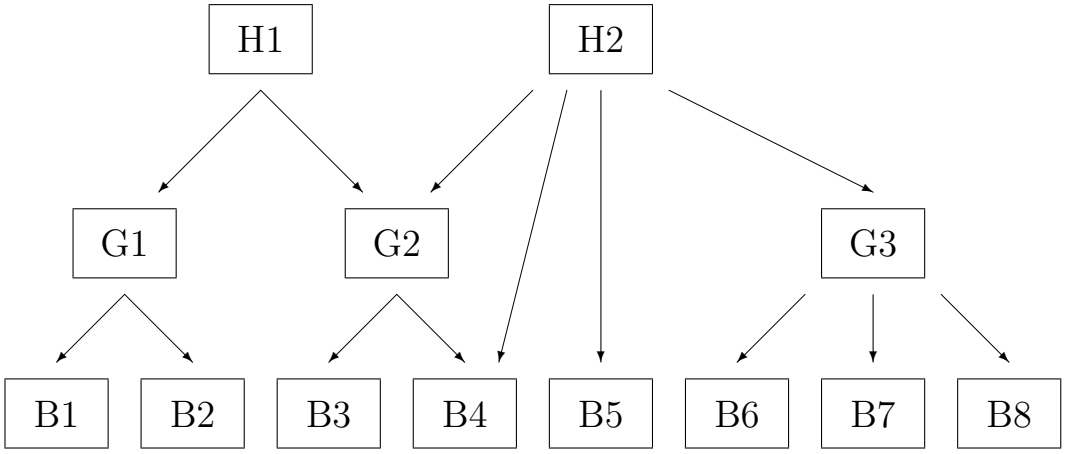



Abb. 5.2-2: Abhängigkeiten zwischen Komponentenklassen

Die mit *B* beginnenden Komponenten sind Basiskomponenten, die Komponenten *G* und *H* sind höhere Komponenten. Der Graph zeigt, daß beispielsweise die Komponente *H1* die Klassen *G1* und *G2*, und diese wiederum die Klassen *B1*, *B2*, *B3* und *B4* benötigen.

Der Graph muß zyklenfrei sein. Eine Komponente darf sich nicht selbst beinhalten. Diese Prüfung übernimmt die Modellerstellungsumgebung.

Wurden zu einer Komponentenklasse mehrere Versionen angelegt, wird für den Abhängigkeitsgraphen die aktuell gültige Version herangezogen. Diese läßt sich durch ein spezielles Kommando (*actvers*) benennen.

Liegen alle Komponentenklassen vor, die für ein Simulationsmodell benötigt werden, kann die Erzeugung des Simulationsprozesses erfolgen. Hierfür werden die Kommandos *check* und *install* zur Verfügung gestellt. Diese Kommandos umfassen im einzelnen folgende Aufgaben:

### Kommando *check*

1a) Prüfung der Komponenten auf syntaktische Korrektheit

1b) Aufstellen des Abhängigkeitsgraphen

2a) Prüfung der Komponenten auf semantische Korrektheit und Konsistenz

2b) Erzeugung einer C-Funktion pro Komponentenklasse mit Datenstruktur, Symboltabelle und Dimensionierung

Kommando *install*

1) Übersetzen der C-Funktion und Ablegen in der Objektbibliothek *libmodel.a*

2a) Übersetzen des Koppelprogramms *Model.c*

2b) Binden von Hauptprogramm, Koppelprogramm, Modellkomponenten und Laufzeitsystem zum Simulationsprozeß

Die Aufgaben des Kommandos *check* übernimmt der MDL-Compiler. Er umfaßt die beiden Programme MDL_C und MDL_SYM. Beide Programme liefern Dateien, die ins Directory der Komponentenklasse geschrieben werden.

Veranlaßt der Benutzer für eine Komponentenklasse die Prüfung auf Korrektheit (Kommando: *check*), dann wird zuerst der Abhängigkeitsgraph unterhalb dieser Komponentenklasse erzeugt. Ein Lauf des MDL-Compilers unter der Option *-sub* liefert in der Datei `<version>.sub` die Anzahl und Namen der benötigten Komponentenklassen.

Hiermit ist eine Syntaxprüfung verbunden, da eine korrekte Syntax Voraussetzung für das Auffinden der Klassennamen ist. Fehlermeldungen werden auf dem UNIX-Standardkanal für Fehlermeldungen *stderr* ausgegeben und vom Experimentierprozeß auf den Bildschirm umgeleitet.

Durch weitere Compilerläufe wird der Graph nach unten zu den Basiskomponenten entwickelt.

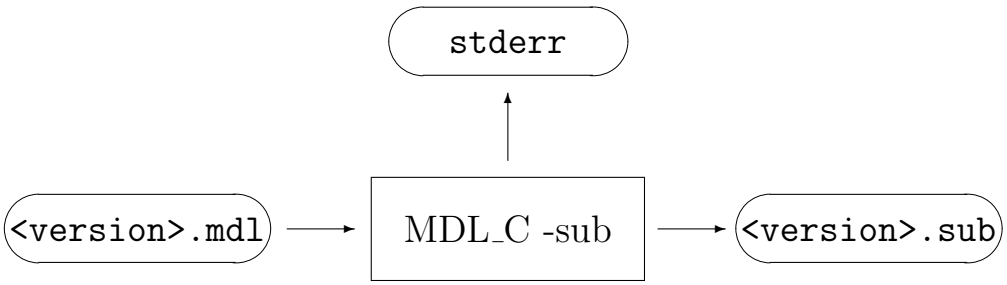



Abb. 5.2-3: Ermittlung der Subklassen

Nach dem Aufstellen des Abhängigkeitsgraphen, wird dieser nun in umgekehrter Richtung – von unten nach oben – durchlaufen und dabei alle Komponentenklassen erneut vom MDL-Compiler bearbeitet.

Das Programm MDL_C prüft jede Komponentenklasse auf semantische Korrektheit und insbesondere auf Korrektheit der Schnittstellen zu untergeordneten Komponenten (Konsistenzprüfung). Es generiert die beiden Dateien:

| | |
|---|---|
| `<version>.c:` | C-Code der Modellkomponente |
| `<version>.sym:` | Symboltabelle |

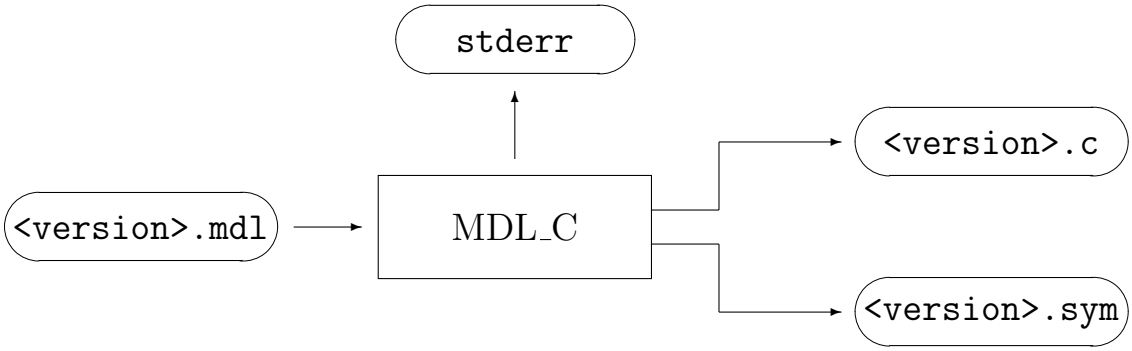



Abb. 5.2-4: Erzeugung von C-Funktion und Symboltabelle

Falls das Kommando *check* für die aktuelle Version einer Komponentenklasse ausgeführt wird, werden aus der Symboltabelle (`<version.sym>`) vom Programm MDL_SYM weitere Dateien für verschiedene Zwecke abgeleitet.

| | |
|---|---|
| `symtab:` | Symboltabelle zur Konsistenzprüfung |
| `mod_struct.h:` | Datenstruktur der Modellkomponente |
| `dimension.h:` | Dimensionierung verschiedener globaler Datenbereiche |
| `offset.dat:` | Angaben über Modellkomponenten und Modellgrößen für Experimentiersystem |
| `offset.incl:` | C-Code zum Erzeugen der Adressen für Modellkomponenten und Modellgrößen |

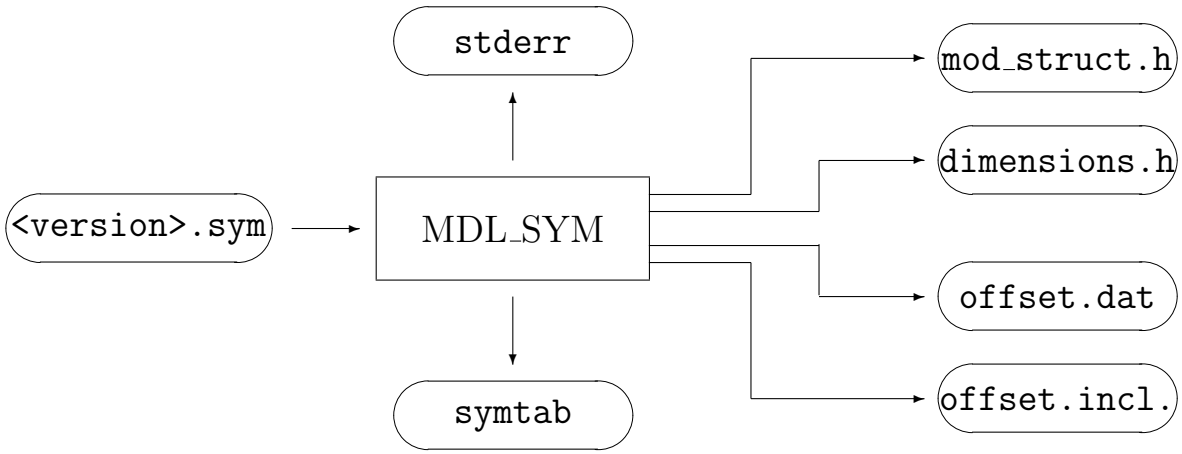



Abb. 5.2-5: Weiterverarbeitung der Symboltabelle

Die Dateien, welche das Programm MDL_C erzeugt, bleiben beim Wechsel der aktuell gültigen Komponentenversion (Kommando: *actvers*) erhalten, während die Ausgabedateien des Programms MDL_SYM neu erzeugt werden.

Die Datei `symtab` enthält diejenigen Symbole, die von übergeordneten Komponenten zugänglich sind, und ist daher die Grundlage einer Konsistenzprüfung.

Nachdem das Modell für korrekt befunden wurde, kann der Benutzer die Erzeugung des Simulationsprozesses veranlassen (Kommando: *install*). Hierbei wird zunächst für sämtliche beteiligten Komponentenklassen aus den C-Funktionen Objektcode (`<version>.o`) erzeugt und in einer Bibliothek (`libmodel.a`) abgelegt. Diese Bibliothek wird im Directory der Modellbank angelegt. Neben den Dateien, die der MDL-Compiler erzeugt hat, werden drei weitere Header-Dateien benötigt, um Symbole des Laufzeitsystems zu deklarieren.

`rts_struct.h`: Datenstrukturen für Modellgrößen

`rts_extern.h`: Externe Variablen des Laufzeitsystems

`rts_func.h`: Funktionen des Laufzeitsystems

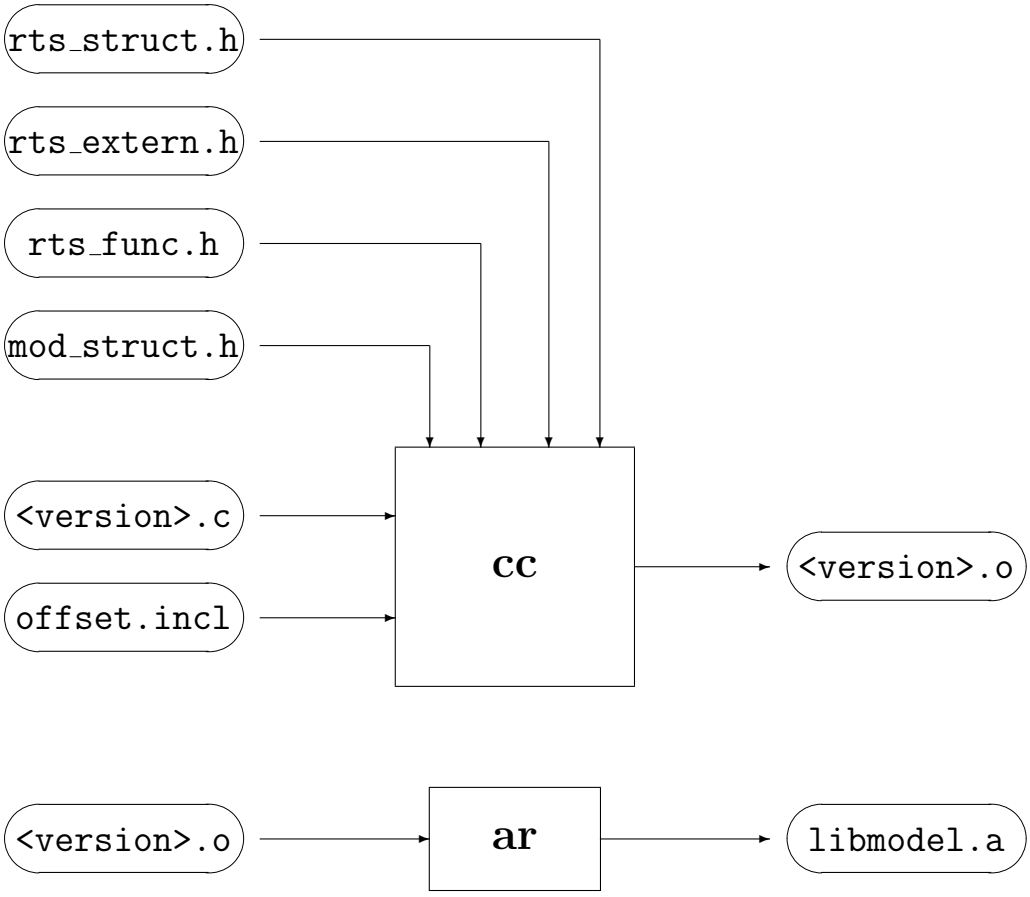



Abb. 5.2-6: Übersetzung der C-Komponente und Erzeugung der Objektcodebibliothek

Zur Kopplung von Laufzeitsystem und höchster Modellkomponente enthält die Funktion `Model.c` den Funktionsaufruf

```
mODEL (ModAdr)
```

der höchsten Modellkomponente. Das Pseudonym `mODEL` wird vom Präprozessor durch Auflösen eines Makros aus der Datei `dimensions.h` , z.B.

```
# define   mODEL   iNTEG
```

ersetzt. Mit `ModAdr` wird die Adresse des Modelldatenbereichs übergeben.

Die weiteren Aufgaben dieses Unterprogramms werden im Zusammenhang mit der Aktivierungsphase erläutert.

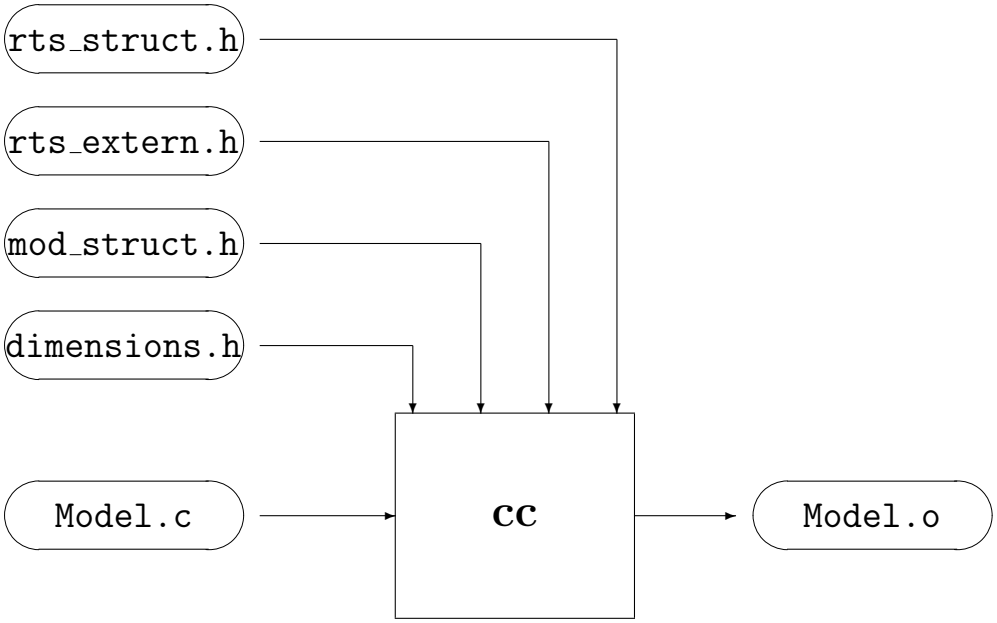



Abb. 5.2-7: Erzeugung des Koppelprogramms *Model.c*

Zum Hauptprogramm des Laufzeitsystems (`main.o`) werden schließlich das Koppelprogramm `Model.o`, die betreffenden Objekte der Komponentenbibliothek `libmodel.a` und Objekte des Laufzeitsystems `librts.a` gebunden. Das Resultat ist der Simulationsprozeß `model.out`.

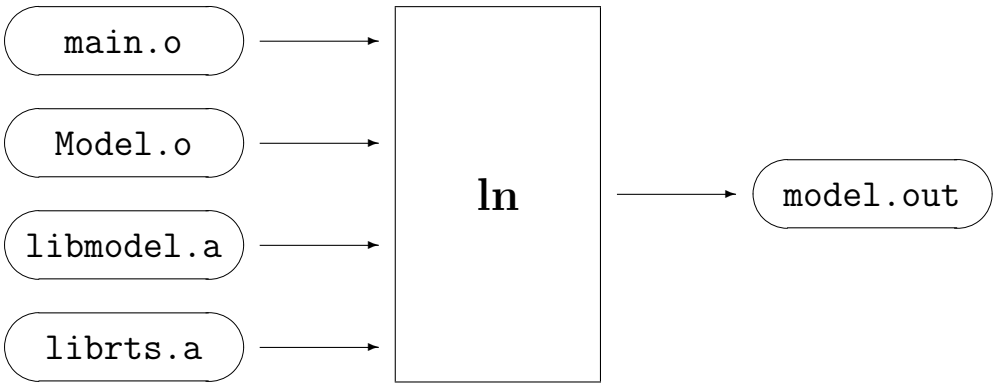



Abb. 5.2-8: Binden des Simulationsprozesses

## 5.3 Die Experimentierumgebung

Der von der Modellerstellungsumgebung aus der Modellspezifikation erzeugte Simulationsprozeß (Kommando: *install*) ist kein alleinstehend ablauffähiges Programm. Um den Simulationsprozeß so kompakt und effizient wie möglich zu halten, wurden alle Funktionen zur Bedienung in den Experimentierprozeß, die sogenannte Experimentierumgebung, verlagert. Diese beiden Prozesse kommunizieren über Datenkanäle des UNIX-Betriebssystems (*Pipes*).

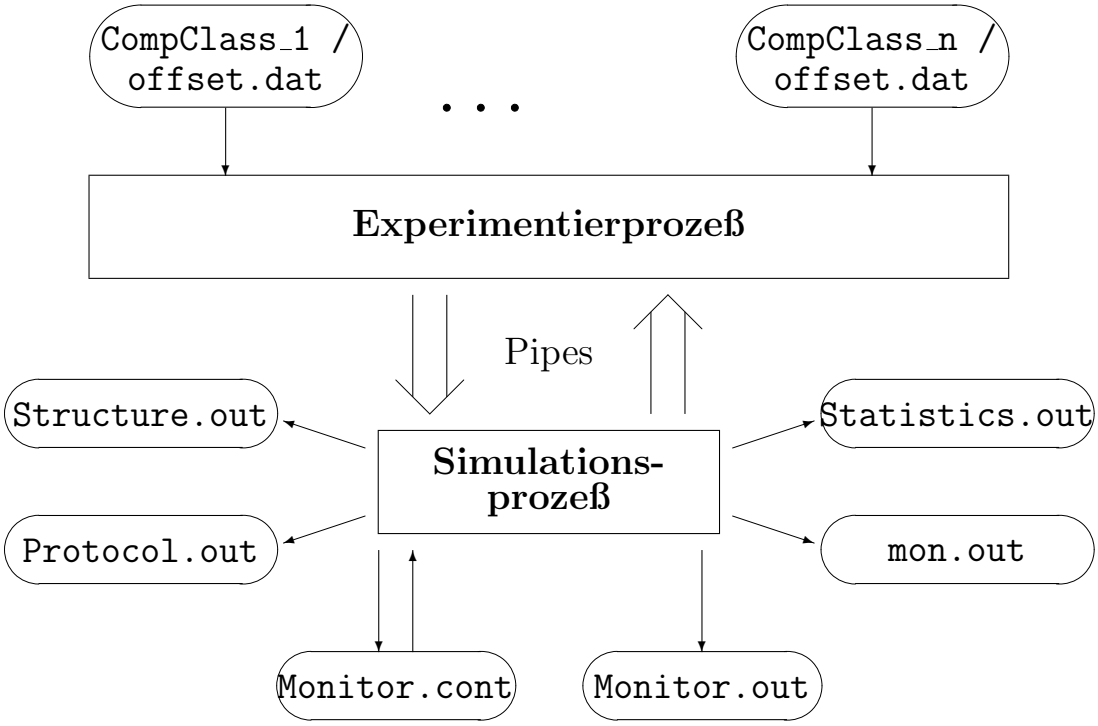



Abb. 5.3-1: Umgebung des Simulationsprozesses

Die Bedienfunktionen, die den Simulationsprozeß unmittelbar betreffen, sind:

1. Das Aktivieren des Simulationsprozesses
2. Die Besetzung des Anfangszustands
3. Das Setzen der Steuerparameter
4. Das Anlegen von Beobachtern
5. Die Ausführung einzelner Simulationsläufe

Die vielen weiteren Funktionen, welche die Experimentierumgebung anbietet, dienen dem Sichern, der Auswertung und Darstellung der Ergebnisse, die während eines oder mehrerer Simulationsläufe gewonnen werden.

### Aktivieren des Simulationsprozesses

Während der gesamten Zeit, in der ein Benutzer mit einem Simulationsmodell experimentiert, bleibt der Simulationsprozeß aktiv und die Kommunikation mit dem Experimentierprozeß bestehen.

Mit dem Aktivieren eines Simulationsmodells (Kommando: *activate*) wird der Prozeß gestartet, die Kommunikation aufgebaut und der Prozeß betriebsbereit gemacht. Am Ende dieser Aktivierungsphase wartet der Simulationsprozeß auf die Anweisung, Simulationsläufe auszuführen. Eine Protokollierung wird auf die Datei `Structure.out` vorgenommen.

Während der Experimentierprozeß von allen beteiligten Komponentenklassen die Namen der Modellgrößen und Subkomponenten aus den Dateien `offset.dat` liest, schickt der Simulationsprozeß die entsprechenden Adressen über die Pipe zum Experimentierprozeß (C-Code: `offset.incl`).

### Besetzen des Anfangszustands

Mit den Informationen über die Modellkomponenten und Modellgrößen, die der Experimentierprozeß während der Aktivierungsphase erhält, ist es diesem möglich, jede vom Benutzer definierte Modellgröße mit ihrem Pfadnamen anzusprechen und ihren Wert zu verändern.

**Setzen der Steuerparameter**

Durch das Setzen von Steuerparametern kann der Benutzer auf die Ausführung der Zustandsübergänge Einfluß nehmen. Die folgende Tabelle listet die Steuerparameter mit ihren Standardvorbesetzungen auf. Ihre Bedeutung wurde bereits in Kapitel 4 behandelt.

| | | | |
|---|---|---|---|
| Auflösungsvermögen: | `TScal` | = | `1 E-3` |
| größte darstellbare Zeit: | `TMax` | = | `1 E+6` |
| max. Anzahl Ereignistakte: | `MaxTkt` | = | `50` |
| max. Anzahl Crossingschritte: | `MaxCro` | = | `50` |
| max. Anzahl Zyklen: | `MaxCyc` | = | `50` |
| Abbruchschranke der Iteration: | `Eps` | = | `1 E-10` |
| Integrationsverfahren: | `IntMeth` | = | `3` |
| Integrationsschrittweite: | `IntStep` | = | `0.1` |
| Fehlerabschätzung: | `ErrEval` | = | `2` |
| größter zulässiger rel. Fehler: | `HighRErr` | = | `1 E-6` |
| kleinster zulässiger rel. Fehler: | `LowRErr` | = | `1 E-7` |
| größte zulässige Schrittweite: | `StepMax` | = | `10` |
| Protokollstufe: | `PrtDepth` | = | `0` |
| Zeitpunkt Protokollanfang: | `TPrtBeg` | = | `0` |
| Zeitpunkt Protokollende: | `TPrtEnd` | = | `1 E+6` |

Tabelle 5.2-1: Steuerparameter mit Standardvorbelegungen

### Anlegen von Beobachtern

Vor der Ausführung eines Simulationslaufs ist noch zu überlegen, welche Modellgrößen beobachtet werden sollen und in welcher Form dies geschehen soll. Es werden mehrere Typen von Beobachtern (*observers*) unterschieden:

1. Aufzeichnung von Zeitreihen in festem Abtastabstand
   Angabe von Anfangs- und Endezeit sowie Schrittweite
2. Aufzeichnung von Zeitreihen bei nderung eines Wertes (nur für diskrete Größen)
   Angabe von Anfangs- und Endezeit
3. Aufzeichnung von komprimierten Zeitreihen (Mittelwert, Maximum, Minimum eines Intervalls)
   Angabe von Anfangs- und Endezeit sowie Intervallbreite

Zu jedem angelegten Beobachter ist noch anzugeben, welche Modellgrößen aufgezeichnet werden sollen.

Die Informationen über die Beobachter werden auf einer Datei `Monitor.cont` abgelegt. Während des Simulationslaufs werden die gewünschten Daten auf der Datei `Monitor.out` aufgezeichnet. Am Ende des Simulationslaufs wird auf der Inhaltsverzeichnis-Datei `Monitor.cont` eingetragen, an welchen Stellen die Aufzeichnung der einzelnen Modellgrößen beginnt.

### Ausführung von Simulationsläufen

Der Simulationslauf kann nun unter Vorgabe einer Endezeit gestartet werden (Kommando: *simulate*). Der Simulationsprozeß durchläuft dabei die sogenannte Simulationsphase. Der Modellzustand und die Steuerparameter werden über die Pipe an den Simulationsprozeß geschickt, der Simulationslauf ausgeführt und die Ergebnisse auf Datei festgehalten. Abschließend wird der Endezustand zurückgeschickt.

Während des Simulationslaufs werden die von den Beobachtern gesammelten Daten auf die Datei `Monitor.out` geschrieben. Die ausgeführten Zustandsübergänge werden auf Wunsch des Benutzers auf der Datei `Protocol.out` aufgelistet.

Am Ende des Simulationslaufs wird auf die Datei `Statistics.out` statistische Information über die Ausführung der Zustandsübergänge geschrieben.

Die Datei `mon.out` wird mit Information beschrieben, aus der das UNIX-Kommando *profile* eine Statistik über die Ausführungszeit jeder einzelnen Funktion des Simulationsprozesses erzeugt.

Die Unterbrechung des Simulationslaufs ist jederzeit über Tastatur durch Eingabe von `CTRL \` möglich. Der Simulationslauf wird in gleicher Weise beendet wie beim Erreichen der Endezeit. Er kann deshalb wieder fortgesetzt werden.

Tritt während des Simulationslaufs ein Fehler auf, wird die entsprechende Fehlermeldung an den Experimentierprozeß geschickt und der Lauf beendet.

## 5.4 Die Kommunikation zwischen Simulations- und Experimentierprozeß

Beim Starten des Simulationsprozesses durch den Experimentierprozeß werden zwischen beiden Prozessen zwei Datenkanäle (sogenannte *Pipes*) eingerichtet, die eine schnelle Kommunikation in beiden Richtungen ermöglichen.

Diese Interprozeßkommunikation dient zur Erfüllung folgender Aufgaben:

- Übergabe der Adressen von Modellgrößen und Modellkomponenten
- Übergabe der Modelldaten
- Steuerung des Simulationslaufs
- Übergabe der Steuerparameter
- Übergabe des Namens der Beschreibungsdatei für Observer
- Ausgabe von Fehlermeldungen
- Ausgabe von DISPLAY-Meldungen
- Ausgabe von Zeitfortschrittsmeldungen

Während der Aktivierungsphase werden dem Experimentierprozeß die Adressen aller Modellgrößen und Modellkomponenten überspielt.

Es folgen die Modelldaten mit den Vorbelegungen, die der Benutzer bei der Modellbeschreibung vorgesehen hat. Mit Hilfe der Adressen kann der Experimentierprozeß sowohl lesend wie schreibend auf die Modelldaten zugreifen.

Die Ausführung eines Simulationslaufs wird vom Experimentierprozeß gestartet. Die initialisierten Modelldaten werden an den Simulationsprozeß übertragen und nach Beendigung des Simulationslaufs zurückgeschickt. Vor dem eigentlichen Modellablauf erhält der Simulationsprozeß auch die Belegung der Steuerparameter sowie den Namen derjenigen Datei, in der die Beobachter spezifiziert sind.

Während des Modellablaufs ist der Experimentierprozeß empfangsbereit für verschiedene Meldungen, um diese zur Ausgabe am Bildschirm weiterzuleiten. Dies ist notwendig, da der Simulationsprozeß nicht direkt auf den Bildschirm ausgeben darf.

Das Simulationssystem wickelt alle Ein-/Ausgabeoperationen mit dem Bildschrim über einen eigenen Prozeß ab, der die auszugebende Information in verschiedenen Fenstern darstellt. Fehlermeldungen und Meldungen, die das DISPLAY-Kommando erzeugt, werden im sogenannten Arbeitsfenster ausgegeben. Meldungen, welche die aktuelle Simulationszeit liefern (Zeitfortschrittsmeldungen) erscheinen in dem Fenster, das die laufenden Aktivitäten anzeigt.

# 6 Der Simulationsprozeß

## 6.1 Der Aufbau des Simulationsprozesses

Der Simulationsprozeß umfaßt vier Bestandteile:

- das Hauptprogramm *main.c*
- das Koppelprogramm *Model.c*
- die Funktionen der im Modell vorhandenen Komponentenklassen
- das Laufzeitsystem

Mit der Ausführung des Kommandos *activate* wird das Hauptprogramm betreten und damit der Simulationsprozeß gestartet. Der daraufhin veranlaßte Programmablauf (Aktivierungsphase) macht den Simulationsprozeß bereit für die Ausführung von Simulationsläufen (Simulationsphase).

In diesem Zustand wartet der Simulationsprozeß auf eine Nachricht vom Experimentierprozeß. Diese Nachricht kann die Ausführung eines Simulationslaufs (Kommando: *simulate*) oder die Beendigung des Simulationsprozesses veranlassen. Nach Ausführung eines Simulationslaufs ist der Prozeß bereit für weitere Läufe.

Das Hauptprogramm hat dementsprechend den folgenden Aufbau:

1. Aktivierungsphase ausführen
2. Simulationsphase ausführen

   ——————— Schleifenanfang ———————

   2.1 Nachricht vom Experimentierprozeß entgegennehmen
   2.2 Verlassen des Simulationsprozesses bei Ende-Meldung
   2.3
   ⋮ Ausführen eines Simulationslaufs
   2.8

   ——————— Schleifenende ———————

Jede im Modell enthaltene Komponentenklasse ist mit einer C-Funktion vertreten. Bei jedem Aufruf repräsentiert eine solche Funktion eine bestimmte Ausprägung einer Komponentenklasse. Während der Aktivierungsphase wird jeder Ausprägung ein eigener Datenbereich zugewiesen. Beim Aufruf der Komponentenfunktion während der Simulationsphase wird ihr als Parameter mitgegeben, auf welchem Datenbereich sie gerade zu operieren hat.

Das Programm *Model* bindet die höchste Komponente des Modells an das Hauptprogramm an. Es wird während der Aktivierungsphase aufgerufen, legt die Datenbereiche des Modells an und ruft seinerseits zweimal die höchste Modellkomponente auf.

Dies veranlaßt einen zweimaligen Abstieg im Modellbaum, bei dem sämtliche Komponenten des Modells durchlaufen werden. Zu diesem Zweck enthalten die Komponentenfunktionen Aufrufe ihrer Subkomponenten, so daß der gesamte Modellbaum traversiert wird. Jede Komponentenfunktion wird bei einem Abstieg genau sooft durchlaufen, wie sie Ausprägungen besitzt.

Dabei werden die Datenbereiche aller Komponenten initialisiert. Insbesondere wird das Modell so verzeigert, daß während der Simulationsphase nur noch diejenigen Komponentenfunktionen auszuführen sind, welche Basiskomponenten repräsentieren.

Die folgenden Abbildungen zeigen beispielhaft die Struktur eines Modells und die Aufrufe der Komponentenfunktionen während der Aktivierungsphase (durchgezogene Linien) und während der Simulationsphase (gestrichelte Linien).

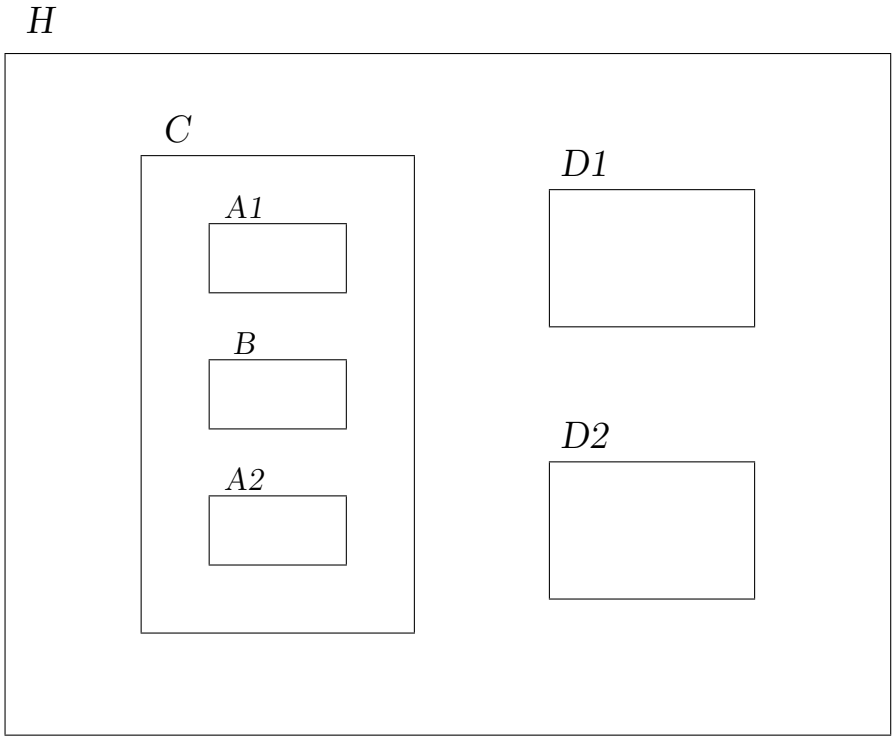



Abb. 6.1-1a: Struktur eines Modells

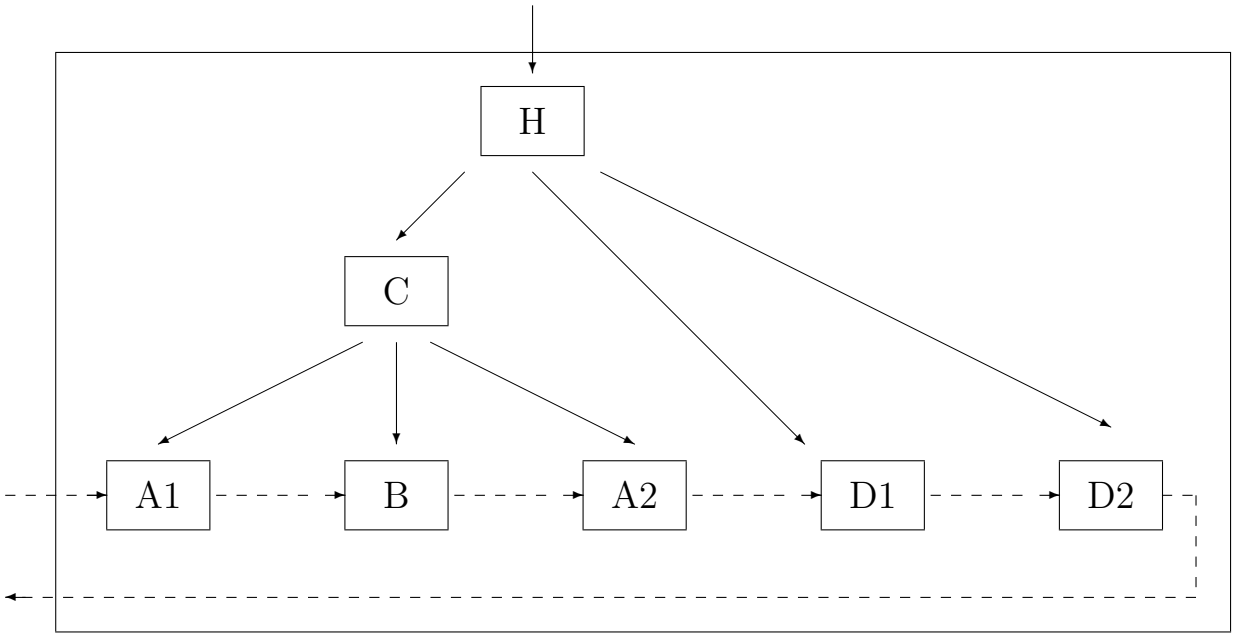



Abb. 6.1-1b: Aufrufe der Modellkomponenten

Der Modellablauf während der Simulationsphase findet im Wechselspiel zwischen dem Laufzeitsystem und den Funktionen der Basiskomponenten statt.

Das Laufzeitsystem umfaßt eine Sammlung von C-Funktionen zur

- Ablaufsteuerung
- Integration von Differentialgleichungen
- Vorbereitung der Zustandsübergänge
- Auswertung statischer Beziehungen
- Beobachtung der Modellgrößen (Monitor)

sowie eine Reihe von Hilfsfunktionen.

Während der Simulationsphase übernimmt die Ablaufsteuerung die Kontrolle über den Programmfluß. Sie sorgt dafür, daß das Verhalten des Modells der Spezifikation entspricht. Zur Auswertung der Modellbeziehungen ruft sie die Funktionen der Basiskomponenten auf mit der Vorgabe, Beziehungen eines bestimmten Typs auszuführen (algebraische Gleichungen, Differentialgleichungen, Ereignisse).

Die Funktionen der Basiskomponenten bedienen sich hierzu wiederum einiger kleinerer Funktionen zur Vorbereitung der Zustandsübergänge oder zum Auswerten statischer Beziehungen. Zur Ausführung eines Integrationsschritts gibt die Ablaufsteuerung die Kontrolle an ein Integrationsverfahren ab, das ebenfalls Modellbeziehungen in den Basiskomponenten auswerten läßt.

Bei Erreichen eines neuen Zustands veranlaßt die Ablaufsteuerung die Aufzeichnung der zu beobachtenden Modellgrößen.

Die folgende Abbildung zeigt den Aufbau des Simulationsprozesses und die gegenseitigen Aufrufe in einem Überblick:

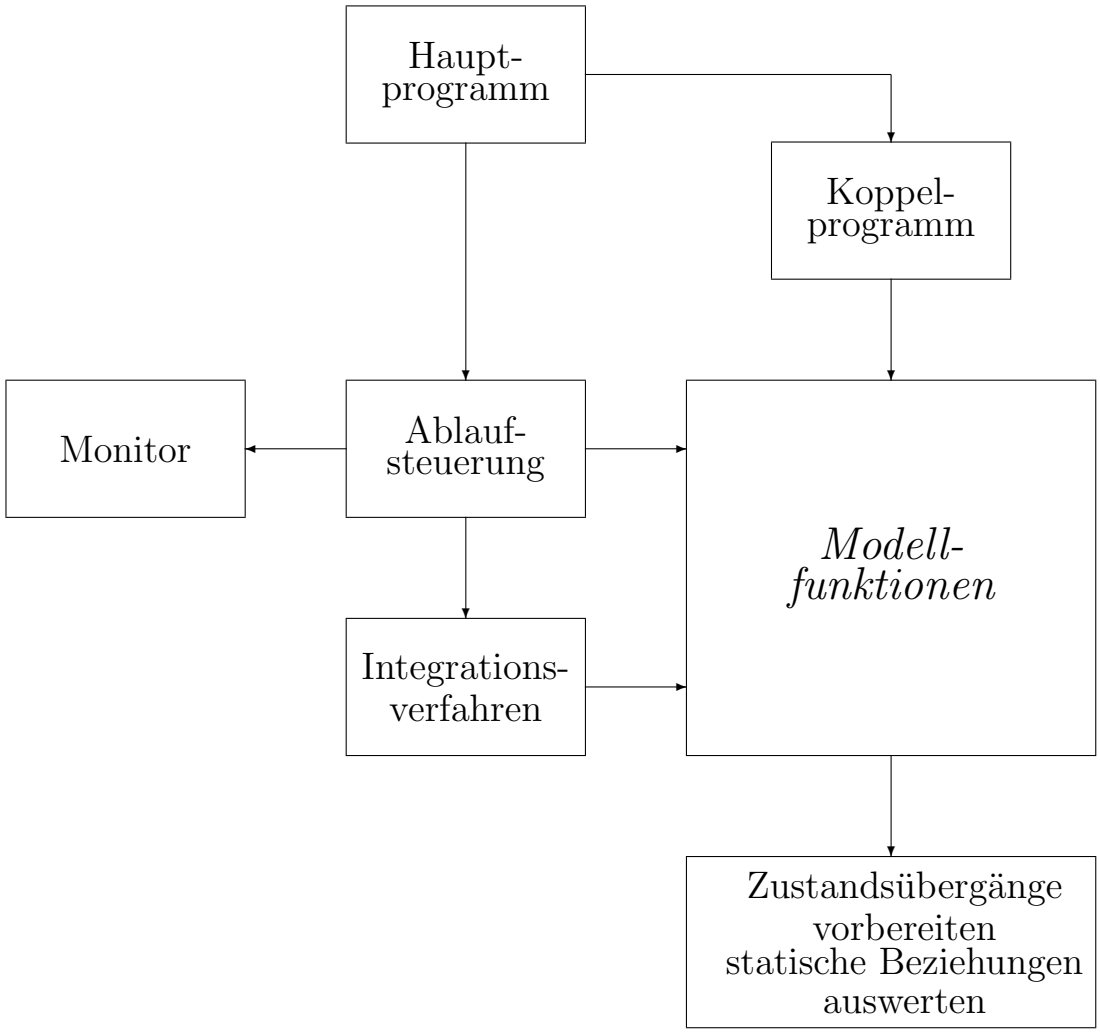



Abb. 6.1-2: Aufbau des Simulationsprozesses

## 6.2 Die Datenbereiche

Die globalen Datenbereiche des Simulationsprozesses lassen sich unterscheiden in

- Systemdaten und
- Modelldaten.

Die Systemdaten gehören fest zum Simulationsprozeß, während die Modelldaten aus der Modellbeschreibung des Benutzers abgeleitet werden.

Die verwendeten Datentypen (insbesondere Strukturen) für System- und Modelldaten sind in der Header-Datei

```
rts_struct.h
```

abgelegt.

Sämtliche globale Systemdaten werden im Hauptprogramm `main.c` definiert. Den Funktionen des Simulationsprozesses wird die Header-Datei

```
rts_extern.h
```

mitgegeben, in der alle globalen Systemdaten als *extern* deklariert werden.

Die Deklaration des Modelldatenbereichs wird vom MDL-Compiler in der Header-Datei

```
mod_struct.h
```

hinterlegt. Den erforderlichen Speicherplatz für die Modelldaten reserviert das Koppelprogramm *Model.c*.

Einige Systemdaten sind Felder, deren Dimensionen sich aus dem Modell ergeben. Ihre Dimensionierung wird ebenfalls im Koppelprogramm *Model.c* vorgenommen.

### 6.2.1 Konventionen zur Namensgebung

Namen für Variablen, Funktionen und Makros sollten gut lesbar und einprägsam sein. Sie dürfen nicht zu lang sein, damit der Programmtext kurz und prägnant bleibt. Und es darf keinen Konflikt mit Bezeichnern aus dem Modell geben.

Bezeichner, die der Benutzer in sein Modell einbringt, können

- mit Schlüsselwörtern und Standardfunktionen der Programmiersprache C
- mit globalen Systemgrößen und Makros

in Konflikt geraten.

Für die interne Schreibweise von benutzereigenen Bezeichnern treffen wir daher folgende Vereinbarung:

1. Bezeichner für Modelldaten (Modellgrößen und Modellkomponenten) werden in Großschreibweise umgewandelt. (z.B. `Kraft` → `KRAFT`)
2. Der Bezeichner für die C-Funktion einer Komponente und für die Datenstruktur einer Komponente wird in Großschreibweise umgewandelt, wobei das erste Zeichen in einen kleinen Buchstaben konvertiert wird. (z.B. `Feder` → `fEDER`)

Da in der Programmiersprache C ausschließlich die Kleinschreibweise gebräuchlich ist und alle Bezeichner länger als ein einzelner Buchstabe sind, kann kein Konflikt entstehen. Durch die unterschiedliche Schreibweise von Datenbereich und Funktion einer Komponente, läßt sich der gleiche Bezeichner mit unterschiedlicher Bedeutung einsetzen.

Für Systemvariablen wird auf die für Modelldaten vereinbarte Schreibweise verzichtet und Namen mit folgenden Eigenschaften bevorzugt:

- Namen sollen nicht oder kaum länger als 8 Zeichen sein.
- Sie sollen mit einem Großbuchstaben beginnen und mit Kleinbuchstaben fortgesetzt werden (z.B. `Mode`)
- Zusammengesetzte Namen enthalten weitere Großbuchstaben (z.B. `StepSize`). Ein Unterstrich wird nicht verwendet.
- Zeiger beginnen mit einem kleinen `'p'`, gefolgt von dem Objekt auf das sie zeigen.
  Der Name muß außer dem `'p'` mindestens einen weiteren Kleinbuchstaben enthalten. (z.B. `pCState, pData`)

### 6.2.2 Darstellung der Modellgrößen

Den in SIMPLEX-MDL angebbaren Wertemengen entsprechen in der C-Implementierung die folgenden Typdefinitionen:

| | | | |
|---|---|---|---|
| INTEGER | typedef | long | integ; |
| REAL | typedef | double | real; |
| LOGICAL | typedef | short | logic; |
| Enumeration Set | typedef | short | enumera; |

Die Zeit wird sowohl als diskrete als auch quasi-kontinuierliche Zeitmenge geführt.

| | | | | |
|---|---|---|---|---|
| diskrete Zeitmenge | typedef | long | time; | Variable `'T'` |
| quasi-kontinuierliche Zeitmenge | typedef | double | time_continuous; | Variable `'Tc'` |

Die Modellgrößen werden nach ihrem Typ, d.h. Verwendung, Zeitverlauf und Wertemenge, durch unterschiedliche Datenstrukturen repräsentiert.

Die nachfolgende Übersicht zeigt die verwendeten Datentypen:

| | INTEGER | REAL diskret | REAL kontinuierlich | LOGICAL | Enumeration |
|---|---|---|---|---|---|
| Konstante | integ | real | — | logic | enumera |
| Zustandsvariable | IState | DState | CState | LState | EState |
| abh. Variable | IDep | RDep | RDep | LDep | EDep |
| Sensorvariable | ISensor | RSensor | RSensor | LSensor | ESensor |
| Ausgangsgröße | IOutput | ROutput | COutput | LOutput | EOutput |
| Eingangsgröße | IInput | RInput | RInput | LInput | EInput |

Tabelle 6.2.2-1: Datentypen für Modellgrößen

Konstanten werden durch einfache Variablen dargestellt, während für alle anderen Modellgrößen eine Struktur (Verbund, Record) vereinbart wird. Zur Beschreibung dieser Strukturen verwenden wir die Notation der Programmiersprache C.

**Datenstruktur für Zustandsvariablen:**

```
struct   XState
{
  <value_set>      Value;       Wert im aktuellen Zustand
  <value_set>      NxtValue;    Wert im Folgezustand
  struct XState *  NxtVar;      Listenzeiger auf veraenderte ZV
  struct EFFECT    Effect;      Struktur mit Auswirkungen
}                               einer Aenderung
```

`XState` steht für `IState`, `RState`, `LState` oder `EState`.

`<value_set>` steht für `integ, real, logic` oder `enumera`.

Mit `struct XState *` wird ein Zeiger auf eine Struktur des Typs `XState` angelegt.

Mit `struct EFFECT` wird eine Struktur des Typs `EFFECT` angelegt.

Die Datenstruktur *XState* enthält in erster Linie den Wert der Zustandsvariablen im aktuellen Zustand. Wird sie – bedingt durch ein Ereignis – im Folgezustand einen anderen Wert aufweisen, wird dieser Wert vorgemerkt und die Datenstruktur in eine Liste eingehängt. Während der Aktivierungsphase wird für alle Modellgrößen festgestellt, welche Auswirkungen nderungen ihres Wertes hervorrufen und dies in der Datenstruktur `EFFECT` festgehalten.

Die Datenstruktur für kontinuierliche Zustandsgrößen `CState` beinhaltet ferner den Wert der Variablen zu Beginn des aktuellen Integrationsschritts sowie den Wert der Ableitung.

```
struct   CState
{
  real  Value;                  Wert im aktuellen Zustand
  real  NxtValue;               Wert im Folgezustand
  real  LstValue;               Wert zu Beginn des Inte-
                                grationsschrittes
  real  Deriv;                  Wert der Ableitung
  struct CState *  NxtVar;      Listenzeiger auf veraenderte ZV
  struct EFFECT    Effect;      Struktur mit Auswirkungen
}                               einer Aenderung
```

**Datenstruktur für abhängige Variablen:**

```
struct   XDep
{
  <value_set>    Value;          Wert
  struct EFFECT  Effect;         Struktur mit Auswirkungen
}                                einer Aenderung
```

`XDep` steht für `IDep, RDep, LDep` oder `EDep`.

Die Datenstruktur `XDep` beinhaltet den Wert der abhängigen Größe sowie die Auswirkungen, die eine Veränderung dieser Größe nach sich ziehen.

**Datenstruktur für Sensorvariablen:**

```
struct   XSensor
{
  <value_set> *  pValue;         Zeiger auf Wert der ange-
                                 schlossenen Groesse
  <value_set>    Value;          Wert der Vorbesetzung
  struct EFFECT  Effect;         Struktur mit Auswirkungen
}                                einer Aenderung
```

`XSensor` steht für `ISensor, RSensor, LSensor` oder `ESensor`.

`<value_set> *` steht für einen Zeiger auf einen Wert mit der Wertemenge `<value_set>`.

Die Datenstruktur `XSensor` enthält in erster Linie einen Zeiger auf den Wert derjenigen Modellgröße, zu der eine Verbindung besteht. Weiterhin beinhaltet sie den Wert, mit dem die Sensorgröße vorbesetzt wurde sowie die Auswirkungen einer Veränderung der Sensorvariablen.

**Datenstruktur für Ausgangsgrößen:**

```
struct   XOutput
{
  <value_set> *   pValue;       Zeiger auf Wert der aequi-
                                valenzierten Groesse
  struct EFFECT * pEffect;      Zeiger auf EFFECT-Struktur
}                               der aequivalenzierten Groesse
```

`XOutput` steht für `IOutput, ROutput, LOutput` bzw. `EOutput`.

Die Datenstruktur `XOutput` enthält einen Zeiger auf den Wert derjenigen Modellgröße in einer Basiskomponente, zu der eine ausgangsseitige Verbindung (quivalenzierung) besteht, sowie einen Zeiger auf die EFFECT-Struktur dieser Größe.

Eine Ausgangsgröße, die auf eine kontinuierliche Zustandsvariable verweist, beinhaltet darüber hinaus auch einen Zeiger auf den Wert der Ableitung der äquivalenzierten Größe.

```
struct   COutput
{
  real *           pValue;      Zeiger auf Wert der aequi-
                                valenzierten Groesse
  real *           pDeriv;      Zeiger auf Ableitung der
                                aequivalenzierten Groesse
  struct EFFECT *  pEffect;     Zeiger auf EFFECT-Struktur
}                               der aequivalenzierten Groesse
```

**Datenstruktur für Eingangsgrößen:**

```
struct   XInput
{
  <value_set> *    pValue;      Zeiger auf Wert der ange-
                                schlossenen Groesse
  <value_set>      Value;       Wert der Vorbesetzung
  struct EFFECT *  pEffect;     Zeiger auf EFFECT-Struktur
}                               der angeschlossenen Groesse
```

`XInput` steht für `IInput, RInput, LInput` bzw. `EInput`.

Die Datenstruktur `XInput` enthält einen Zeiger auf den Wert derjenigen Modellgröße, zu der eine Verbindung besteht. Weiterhin beinhaltet sie den Wert, mit dem die Eingangsgröße vorbesetzt wurde sowie einen Zeiger auf die EFFECT-Struktur der angeschlossenen Größe.

Die beschriebenen Typdefinitionen werden in der Header-Datei

```
rts_struct.h
```

zur Verfügung gestellt.

Diese Datei enthält auch verschiedene global verwendete Makrodefinitionen. Unter anderen gehören hierzu die Makros

```
# define   No   0
# define   Yes  1
```

zur Belegung von Variablen mit Datentyp `logic`.

### 6.2.3 Die Modelldatenstruktur

Für jede Modellkomponente wird eine Datenstruktur angelegt, welche ihre eigenen Modellgrößen sowie die Datenstrukturen aller Subkomponenten enthält. Diese Datenstruktur wird in der Datei `mod_struct.h` im Directory der Modellkomponente abgelegt.

Als Bezeichner für die Modellgrößen werden die Namen herangezogen, die der Benutzer vergeben hat, mit dem Unterschied, daß alle Namen in Großschreibweise konvertiert werden. Dies ist möglich, da SIMPLEX-MDL Bezeichner mit unterschiedlicher Groß-/Kleinschreibung nicht zuläßt.

Als Bezeichner für die Datenstruktur einer Komponente wird der Komponentenname in Großschreibweise mit dem ersten Buchstaben als kleines Schriftzeichen verwendet.

Der Aufbau der Modelldatenstruktur entspricht genau dem hierarchischen Aufbau des Modells. Alle Modellgrößen lassen sich damit über ihre natürlichen Zugriffspfade erreichen.

Die gesamten Daten eines Modells sind in einer einzigen Struktur und damit in einem zusammenhängenden Datenbereich abgelegt. Dies ist dazu erforderlich, um einen einfachen und schnellen Datenaustausch mit der Experimentierumgebung herstellen zu können.

**Beispiel für die Modelldatenstruktur einer Basiskomponente:**

```
                                    # ifndef Comp_BCOMP

BASIC COMPONENT   BComp               struct   bCOMP
DECLARATION OF ELEMENTS               {
                                         struct NAME_ELEM  NameElem;

CONSTANT   ICon (INT) := 0               integ  ICON;

STATE VARIABLES
DISCRETE   Var1 (REAL) := 0              struct RState  VAR1;
CONTINUOUS Var2 (REAL) := 0              struct CState  VAR2;

DEPENDENT VARIABLES
DISCRETE   Var3 (LOGICAL)                struct LDEP    VAR3;

SENSOR VARIABLE
CONTINUOUS Var4 (Colour)                 struct ESensor VAR4;

                                         long  CroBase;
                                         long  RegBase;
                                         long  TCmpBase;
                                      };

                                    # define  Comp_BCOMP
                                    # endif
```

`Colour` ist der Name einer Wertemenge, die durch Aufzählung vereinbart wurde.

Neben den Bezeichnern, die der Benutzer vereinbart hat, enthält die Datenstruktur einer Modellkomponente eine weitere Datenstruktur NAME_ELEM, aus welcher der Name der Komponente hervorgeht.

Basiskomponenten enthalten darüber hinaus die Basisindizes für verschiedene globale Felder.

**Beispiel für die Modelldatenstruktur einer höheren Komponente:**

```
                                    # ifndef Comp_HCOMP

                                    # include "Sub/mod_struct.h"
                                    # include "SubClass/mod_struct.h

HIGH LEVEL COMPONENT   HComp          struct   hCOMP
                                      {
                                         struct NAME_ELEM  NameElem;
SUBCOMPONENTS
   Sub                                   struct sUB   SUB;
   SubComp2 OF CLASS SubClass,           struct sUBCLASS SUBCOMP2;
   SubComp3 OF CLASS SubClass            struct sUBCLASS SUBCOMP3;

INPUT CONNECTION
   InVar --> SubComp2.Var1;              struct RInput   INVAR;

OUTPUT EQUIVALENCE
   OutVar := SubComp3.Var4;              struct EOutput  OUTVAR;
                                      };

                                    # define Comp_HComp
                                    # endif
```

Es wird davon ausgegangen, daß `Var1` eine reelle Größe und `Var4` eine Größe mit Aufzählungsmenge ist.

Die Strukturen `sUB` und `sUBCLASS` werden als bereits bekannt vorausgesetzt. Über die Preprozessor-Anweisung `# include` werden diese Strukturen zuvor vereinbart. Verschiedene Compiler lassen es nicht zu, daß eine Struktur mehrfach, wenn auch identisch, vereinbart wird. Dieser Fall würde beispielsweise dann eintreten, wenn die Struktur `sUB` ebenfalls die Struktur `sUBCLASS` enthielte.

Um die Mehrfachdeklaration einer Modelldatenstruktur zu verhindern, wird nach der Deklaration einer Modelldatenstruktur die Präprozessor-Variable `Comp_<NAME>` definiert und vor einer Deklaration geprüft, ob diese Variable existiert, d.h. die Datenstruktur bereits vereinbart wurde.

### Kritische Anmerkungen zur gewählten Darstellung der Modelldaten

Große Modelle erfordern den Aufbau großer Strukturen, die verschiedenen C-Compilern Probleme bereitet. Mögliche Beschränkungen liegen in

- der Schachtelungstiefe der Strukturen
- der Anzahl der Elemente pro Struktur
- der Gesamtgröße einer Struktur in Byte

Diese Dimensionierungen liegen häufig fest und lassen sich nur bei wenigen Compilern durch die Anwahl von Optionen anpassen.

Dennoch sind im praktischen Betrieb bislang kaum Fälle aufgetreten, bei denen die zulässige Schachtelungstiefe oder die Anzahl der Elemente pro Struktur nicht ausgereicht hätte.

Die mögliche Gesamtgröße einer Struktur stellt dagegen auf Maschinen mit 16 Bit-Architektur eher eine Beschränkung dar. Auf solchen Maschinen ist die Strukturgröße auf 64 KByte begrenzt. Andererseits benötigen Rechner dieser Klasse für solch große Modelle meist unerträglich lange Rechenzeiten, so daß sich dann sowieso der Umstieg auf Rechner mit 32-Bit Architektur anbietet. Für diese Rechnerklasse gilt prinzipiell keine Beschränkung der Strukturgröße. Leider gibt es aber auch hier Compiler, die nur den Aufbau von 64 KByte großen Strukturen erlauben (Microsoft C-Compiler auf SCO Xenix 386). Es bleibt zu hoffen, daß dieser Mangel bald beseitigt wird.

Andere Implementierungen der Modellstruktur wurden angedacht und als aufwendig, weniger übersichtlich und weniger effizient befunden. Daher wurde trotz der bekannten Probleme an der vorgestellten Lösung festgehalten.

### 6.2.4 Globale Systemdaten

Von den zahlreichen globalen Größen des Laufzeitsystems werden an dieser Stelle diejenigen näher beschrieben, die zum Verständnis der später dargestellten Algorithmen von Bedeutung sind.

Die Variable

```
enum  Mode
```

steuert, welche Abschnitte beim Aufruf einer Modellkomponente zu durchlaufen sind. Sie kann die Werte

`Activ:` Ausführung der Aktivierungsphase

`AlgEqu:` Auswertung der algebraischen Gleichungen

`DtlEqu:` Bestimmung der Differentialquotienten

`Events:` Ausführung der Ereignisse

`Detect:` Erkennen von Unstetigkeiten

annehmen.

Die Variable

```
short  ActPhase;
```

kann die Werte '1' oder '2' annehmen und zeigt damit an, welcher Abschnitt der Aktivierungsphase gerade durchlaufen wird.

Die Dimensionierung einiger Systemdaten hängt von der Größe des Modells ab. In den Variablen

`NBComp:` Anzahl der Basiskomponenten

`NCState:` Anzahl der kontinuierlichen Zustandsvariablen

`NTimeCmp:` Anzahl der Zeitvergleiche

`NRegion:` Anzahl der Bereichsprüfungen

`NCross:` Anzahl der Crossings

wird die Dimensionierung dieser Felder geführt.

In der Variablen

```
FBComp  [NBComp+1]
```

werden die Funktionenzeiger auf die Basiskomponenten und in

```
DBComp  [NBComp+1]
```

die Zeiger auf deren Datenbereiche abgelegt.

Die Variablen

```
pCState  [NCState+1]
```

verweisen auf die kontinuierlichen Zustandsvariablen des Modells.

Für Zeitvergleiche, Bereichsprüfungen und Crossings werden die Datenstrukturen

```
TimeCmp   [NTimeCmp+1]
Region    [NRegion+1]
Cross     [NCrossing+1]
```

angelegt.

Um festzuhalten, welche Modellkomponenten mit welchen Aktivitäten auszuführen sind, werden die logischen Variablen

`AlgEffect [NBComp+1]` : Auswirkungen auf algebraische Gleichungen

`EvtEffect [NBComp+1]` : Auswirkungen auf Ereignisse

`DtlEffect [NBComp+1]` : Auswirkungen auf Differentialgleichungen

belegt. Diese bezeichnen wir auch als komponentenbezogene Auswirkungs-Flags.

Im Gegensatz dazu heißen die Flags

```
ExecAlg,  ExecEvt,  ExecDtl
```

modellbezogene Auswirkungs-Flags, da sie anzeigen an, ob überhaupt Aktivitäten eines bestimmten Typs auszuführen sind.

Die Flags

```
logic  AnyDtl;
logic  TimeDep;
```

zeigen an, ob das Modell Differentialgleichungen bzw. zeitabhängige algebraische Ausdrücke enthält.

Zur Steuerung des zeitlichen Ablaufs dienen die Steuerparameter

`time_continuous TScal;` Auflösungsvermögen

`time TBegin;` Anfangszeit des Simulationslaufs

`time TEnd;` Endezeit des Simulationslaufs

Eine Unterbrechung des Simulationslaufs über Tastatur wird in der Variablen

```
logic  Intrpt;
```

festgehalten.

Zur Ausführung der Ereignisse werden

`short Takt;` Taktzähler

`short MaxTkt;` maximale Anzahl Takte

verwendet sowie die Zeiger

```
struct  XState *   HeadXSta,  TailXSta;
```

welche auf den Anfang und das Ende der verketteten Zustandsvariablen verweisen.

Zur Auswertung der algebraischen Gleichungen werden die Variablen

`short Cycle;` Zykluszähler

`short MaxCyc;` maximale Anzahl Zyklen

`double Eps;` zulässige relative Abweichung

benötigt.

Zur Behandlung von Laufzeitfehlern werden die Variablen

| | |
|---|---|
| `short ErrType;` | Fehlertyp |
| `int ActLine;` | aktuelle Zeile in Modellbeschreibung |
| `struct NAME_ELEM * ActComp;` | aktuelle Modellkomponente |

belegt.

Die vorgestellten Variablen reichen aus, um sowohl den größten Teil der Funktionen des Laufzeitsystems als auch den C-Code der Modellkomponenten zu verstehen.

## 6.3 Die Implementierung der Komponentenklassen

Der MDL-Compiler erzeugt für jede Klasse einer Modellkomponente eine C-Funktion. Nach der Übersetzung durch den C-Compiler wird der Objectcode in der Bibliothek der Modellbank `libmodel.a` abgelegt.

### 6.3.1 Aufbau der C-Funktion einer Komponente

Der prinzipielle Aufbau der C-Funktion ist für Basiskomponenten und höhere Komponenten ähnlich. Für höhere Komponenten entfällt allerdings der Code für die Simulationsphase und für Tabellenfunktionen (Punkte 3,5 und 6).

1. (a) Definition von Makros (`c_macros.h`)
   (b) Typdefinitionen des Laufzeitsystems (`rts_struct.h`)
   (c) Deklaration externer Variablen des Laufzeitsystems (`rts_extern.h`)
   (d) Deklaration externer Funktionen des Laufzeitsystems (`rts_func.h`)
   (e) Definition der Datenstruktur der Komponentenklasse (`mod_struct.h`)
2. Definition des Funktionsnamens und der Parameter
3. Deklaration der Funktionsnamen für Tabellenfunktionen
4. Code der Aktivierungsphase
5. Code der Simulationsphase
6. Definition und Implementierung der Tabellenfunktionen

Die Definitionen und Deklarationen unter Punkt 1 erfolgen durch das Einbringen von Include-Dateien.

**Definition der Funktion:**

Die C-Funktion wird als `integer` definiert und erhält als Parameter den Zeiger `pData`, der auf den Datenbereich der jeweiligen Ausprägung der Komponentenklasse verweist. Alle Operationen, welche die Funktion ausführt, werden auf diesem Datenbereich vorgenommen. Auf diese Art wird das Klassenkonzept implementiert.

```
BASIC COMPONENT  CompName      int cOMPNAME  (pData)

                               register struct cOMPNAME * pData;
                               {
                                  ...
                               }
```

Die Zugriffe auf die Modellgrößen der Komponenten erfolgen über den Zeiger `pData`. Um die Lesbarkeit des Programmtextes zu verbessern, werden die Zugriffspfade durch geeignete Makros beschrieben, die in der Datei `c_macros.h` hinterlegt sind.

Steht `var` für den Namen einer Modellgröße, ergeben sich beispielsweise folgende Zugriffe auf den Wert der Modellgröße:

| | | | |
|---|---|---|---|
| Für Konstanten: | `# define` | `Con(var)` | `pData -> var` |
| Für Zustandsgrößen: | `# define` | `Sta(var)` | `pData -> var.Value` |
| Für abh. Größen: | `# define` | `Dep(var)` | `pData -> var.Value` |
| Für Sensorgrößen: | `# define` | `Sen(var)` | `* (pData -> var.pValue)` |

### Deklaration der Tabellenfunktionen

Enthält die Komponente Tabellenfunktionen, werden diese Funktionen mit ihrem Typ vorab deklariert, da ihre Definitionen ans Ende der Datei gestellt werden.

### Codesegment für die Aktivierungsphase

Das Codesegment, das während der Aktivierungsphase durchlaufen wird, hat für Basiskomponenten folgenden Aufbau:

B-4.1 Definition lokaler Variablen

B-4.2 Rücksprung, falls zweiter Abstieg ins Modell

B-4.3 Übergabe der Offset-Adressen von Modellgrößen an den Experimentierprozeß
(nur beim 1. Aufruf der Komponentenklasse)

B-4.4 Speichern des Komponentennamens

B-4.5 Speichern der Funktions- und Datenbereichsadresse der Modellkomponente

B-4.6 Für alle Modellgrößen:
Durchführung der Urinitialisierung
Belegen der Zeiger
Belegen der Effect-Einträge

B-4.7 Belegen der Basisindices für Crossings, Regions und Zeitvergleiche

Der zweite Abstieg ins Modell ist für Basiskomponenten nicht relevant und deshalb erfolgt eine sofortige Rückkehr.

Für höhere Komponenten besitzt das Codesegment für die Aktivierungsphase folgenden Aufbau:

H-4.1 Definition lokaler Variablen

H-4.2 Übergabe der Offset-Adressen von Subkomponenten und Modellgrößen an Experimentierprozeß
(nur beim 1. Aufruf der Komponentenklasse)

H-4.3 Speichern des Komponentennamens (1. Abstieg)

H-4.4 Aufruf der Subkomponenten (1. Abstieg)

H-4.5 Eintragen der Ausprägungsnamen für Subkomponenten
(1. Abstieg)

H-4.6 Zeiger auf aktuellen Komponentennamen setzen

H-4.7 Code für Input Connections (1.+2. Abstieg)

H-4.8 Aufruf der Subkomponenten (2. Abstieg)

H-4.9 Code für Output Equivalences (1. Abstieg)

H-4.10 Code für Component Connections (1. Abstieg)

H-4.11 Code für Initialisierungen (1. Abstieg)

Für jeden der beiden Abstiege ins Modell werden unterschiedliche Abschnitte des Codes bearbeitet. Der Programmfluß ist so angelegt, daß von der untersten zur obersten Hierarchieebene zunächst *Output Equivalences* und *Component Connections* bearbeitet werden. Beim zweiten Abstieg ins Modell werden dann von der obersten zur untersten Ebene die *Input Connections* angelegt.

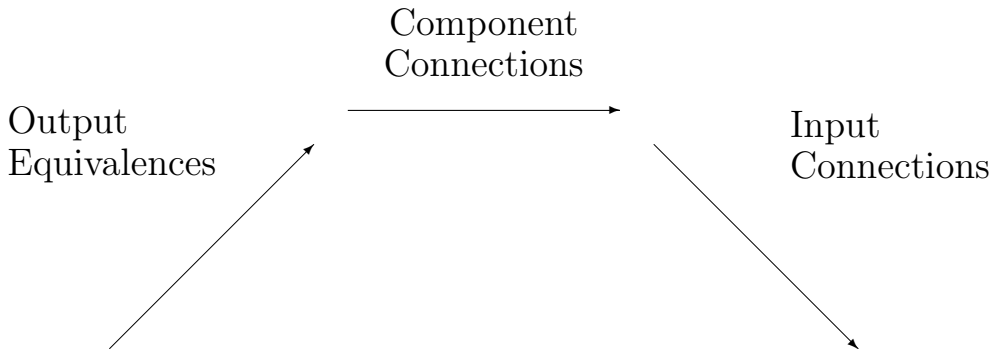



Abb. 6.3.1-1: Ausführungsreihenfolge der Codesegmente

Im Abschnitt über die Aktivierungsphase wird erläutert, warum diese Ausführungsreihenfolge benötigt wird. Auch die anderen Punkte werden dort ausführlich dargelegt.

Während der Simulationsphase werden nur Basiskomponenten durchlaufen. Das hierfür angelegte Codesegment hat folgenden Aufbau:

5.1 Merken des aktuellen Komponentennamens
5.2 Code für Crossings
5.3 Code für Modelldynamik

Im folgenden wird exakt spezifiziert, welcher Code aus einer gegebenen Modellbeschreibung zu generieren ist. Für jede Produktion der Syntax wird der zu erzeugende Code angegeben. Codeabschnitte, die noch weiter aufzulösen sind, werden in spitze Klammern eingeschlossen.

Für ein erstmaliges Lesen der Arbeit genügt es, die weiteren Unterabschnitte zu überfliegen, um sich eine grobe Vorstellung vom Inhalt einer C-Komponente zu verschaffen. Die Erklärungen zum erzeugten Code und das Zusammenspiel mit dem Laufzeitsystem sind der Inhalt der darauf folgenden Abschnitte.

### 6.3.2 Code für die Aktivierungsphase

## a) Basiskomponenten:

```
/*  Aktivierung und Urinitialisierung  */
/*  =================================  */
    if (Mode == Activ)
    {
       static  logic  Call1 = Yes;

       if  (ActPhase == 2)  return;

/*     Uebergabe der Adressen  */
/*     ----------------------  */
       if  (Call1  &&  !ReStart)
       {
#         include  "offset.incl"

          Call1 = No;
       }

/*     Ablegen des Komponentennamens  */
/*     -----------------------------  */
       SaveCompName ("CompName");

       BCount ++;

       FBComp [BCount]  =  cOMPNAME;
       DBComp [BCount]  =  (char *) pData;

       < Initialisierung der Modellgroessen >

       < Setzen der Effizienz-Vermerke >

       pData -> CroBase  =  CroCount;     CroCount  +=  <Anzahl Crossings>;
       pData -> RegBase  =  RegCount;     RegCount  +=  <Anzahl Regions>;
       pData -> TCmpBase =  TCmpCount;    TCmpCount +=  <Anzahl Zeitvergleiche>;

       return;
    }
```

< Initialisierung der Modellgrößen > : `ident '(' range ')' [:= value]`

| | |
|---|---|
| für Konstanten: | `Con(IDENT) = <value>;` |
| für Zustandsvariablen: | `Sta(IDENT) = <value>;`<br>`StNxt(IDENT) = 0;` |
| zusätzlich für kontinuierliche ZV: | `StCCount ++; pCState [StCCount] = Adr(IDENT);`<br>`StD(IDENT) = 0.0;` |
| für abh. Variablen: | `Dep(IDENT) = <value>;` |
| für Sensorvariablen: | `SenVal(IDENT) = <value>;`<br>`SenPtr(IDENT) = & SenVal(IDENT);` |

*mit den Makros:*

```
Con(IDENT)        pData -> IDENT
Sta(IDENT)        pData -> IDENT.Value
StNxt(IDENT)      pData -> IDENT.NxtVar
StD(IDENT)        pData -> IDENT.Deriv
Dep(IDENT)        pData -> IDENT.Value
SenVal(IDENT)     pData -> IDENT.Value
SenPtr(IDENT)     pData -> IDENT.pValue

Adr(IDENT)        & (pData -> IDENT)
```

< Setzen der Effizienz-Vermerke > :

*für jeden Variablentyp:*

```
VarEff (IDENT, CompNo)   =  BCount;
VarEFF (IDENT, EffFlag)  =  < EffFlag >;
VarEff (IDENT, eff_next) =  NULL;
```

*für jeden Zeitvergleich:*

```
TimeCmp [i]. Effect. CompNo   =  BCount;
TimeCmp [i]. Effect. EffFlag  =  < EffFlag >;
TimeCmp [i]. Effect. eff_next =  NULL;
```

mit dem Makro

```
VarEff (IDENT, selector)      pData -> IDENT.Effect.selector
```

Das Effect-Flag besagt, ob die betreffende Variable Auswirkungen

- auf algebraische Gleichungen `EFA = 1`
- auf Ereignisse `EFE = 2`
- auf Differentialquotienten `EFD = 4`

hat und ergibt sich zu

```
< EffFlag >  =  EFA | EFE | EFD;
```

(Das Symbol `'|'` bedeutet bitweise ODER-Verknüpfung)

## b) Höhere Komponenten

```
/*  Aktivierung und Urinitialisierung  */
/*  ================================  */
    static  logic  Call1 = Yes;

/*  Uebergabe der Adressen  */
/*  ----------------------  */
    if (Call1  &&  !ReStart)
    {
#      include "offset.incl"

       Call1 = No;
    }

/*  Ablegen des Komponentennamens  */
/*  -----------------------------  */
    if (ActPhase == 1)
    {
       SaveCompName ("HComp");
    }

/*  Aufruf der Subkomponenten (1. Abstieg)  */
/*  --------------------------------------  */
    if (ActPhase == 1)
    {
       < Aufruf jeder Subkomponente >

       < Ablegen des Namens jeder Subkomponente >
    }

/*  Input Connections  */
/*  -----------------  */
    < Code fuer jede Input Connection >

/*  Aufruf der Subkomponenten (2. Abstieg)  */
/*  --------------------------------------  */
    if (ActPhase == 2)
    {
       < Aufruf jeder Subkomponente >
    }

/*  Output Equivalences  */
/*  -------------------  */
    if (ActPhase == 1)
    {
       < Code fuer jede Output Equivalence >
    }
```

```
/*  Component Connections  */
/*  ---------------------  */
    if (ActPhase == 1)
    {
       < Code fuer jede Component Connection >
    }

/*  Initialisierungen  */
/*  -----------------  */
    if (ActPhase == 1)
    {
       < Code fuer jede Initialisierung >
    }

    return;
```

< Aufruf jeder Subkomponente > :

Syntax: Code:

```
subcomponent  ::=   SubComp            -->  SUBCOMP (Adr(SUBCOMP));
                  | SubComp
                    OF CLASS SubClass  -->  SUBCLASS (Adr(SUBCOMP));
```

mit dem Makro:

```
Adr(SUBCOMP)     & (pData -> SUBCOMP)
```

< Ablegen des Namens jeder Subkomponente > :

Code: `SaveSubCompName (SUBCOMP, "SubComp");`

Makro mit der Bedeutung:

```
strcpy (pData -> NameElem.CompName, "SubComp");

pData -> SUBCOMP.NameElem.SuperComp =  & (pData -> NameElem);
```

< Code für jede Input Connection > :

Fall a) Verbindung zu einer Basiskomponente: `V --> B.X;`
Fall b) Verbindung zu einer höheren Komponente: `V --> H.X;`

```
/*  Im 1. Abstieg der Aktivierungsphase  */
/*  -----------------------------------  */
    if (ActPhase == 1)
    {
/*     Vorbelegung der Zeiger und der Wertezelle  */
/*     -----------------------------------------  */
       InElem (V.pValue)  =  Adr (V.Value);
       InElem (V.pEffect) =  NULL;
       InElem (V.Value)   =  < Standardvorbelegung der Wertezelle > ;
    }

/*  Im 2. Abstieg der Aktivierungsphase  */
/*  -----------------------------------  */
    else
    {
/*     1) Pruefung auf Mehrfachbelegung der Sensor/Input-Groesse  */
/*     ---------------------------------------------------------  */
a:     if (SenElem (B.X.pValue) != Adr (B.X.Value) )  Error (40);
b:     if (InElem  (H.X.pValue) != Adr (H.X.Value) )  Error (40);

/*     2) Zuweisung des Zeigers auf die Wertezelle  */
/*     -------------------------------------------  */
a:     SenElem (B.X.pValue) = InElem (V.pValue);
b:     InElem  (H.X.pValue) = InElem (V.pValue);

/*     3a) Aufbau der Effect-Verzeigerung (falls Basiskomponente)  */
/*     ---------------------------------------------------------  */
a:     AddChain (InElem (V.pEffect),  Adr (B.X.Effect);

/*     3b) Weitergabe des Effect-Zeigers (falls hoehere Komponente)  */
/*     -----------------------------------------------------------  */
b:     InElem (H.X.pEffect) = InElem (V.pEffect);
    }
```

mit den Makros:

```
InElem(VAR)          pData -> VAR
SenElem(VAR)         pData -> VAR
Adr(VAR)           & (pData -> VAR)
```

< Code für jede Output Equivalence > : `Z := A.Y;`

*a) Falls* `A` *Basiskomponente:*

```
/*  Zuweisung des Zeigers auf die Wertezelle  */
/*  ----------------------------------------  */
    OutElem (Z.pValue) = Adr (A.Y);              falls Y Konstante
    OutElem (Z.pValue) = Adr (A.Y.Value);        sonst

/*  Zuweisung des Zeigers auf Ableitung (bei kontinuierlichen Variablen)  */
/*  ---------------------------------------------------------------------  */
    OutElem (Z.pDeriv) = Adr (A.Y.Deriv);        falls Y Zustandsvariable
    OutElem (Z.pDeriv) = NULL;                   falls Y abh. Variable

/*  Adresse des Effect-Eintrags uebernehmen  */
/*  ---------------------------------------  */
    OutElem (Z.pEffect) = NULL;                  falls Y Konstante
    OutElem (Z.pEffect) = Adr (A.Y.Effect);      sonst
```

*b) Falls* `A` *höhere Komponente:*

```
/*  Zeiger auf die Wertezelle weitergeben  */
/*  -------------------------------------  */
    OutElem (Z.pValue) = OutElem (A.Y.pValue);

/*  Zeiger auf Ableitung weitergeben (bei kontinuierlichen Variablen)  */
/*  -----------------------------------------------------------------  */
    OutElem (Z.pDeriv) = OutElem (A.Y.pDeriv);

/*  Zeiger auf Effect-Eintrag weitergeben  */
/*  -------------------------------------  */
    OutElem (Z.pEffect) = OutElem (A.Y.pEffect);
```

mit den Makros:

```
OutElem(VAR)          pData -> VAR
Adr(VAR)           & (pData -> VAR)
```

< Code für jede Component Connection > :

Fall a) Verbindung zu einer Basiskomponente: `A.Y --> B.X` bzw. `Z --> B.X;`

Fall b) Verbindung zu einer höheren Komponente: `A.Y --> H.X` bzw. `Z --> H.X;`

Adresse der Wertezelle von `Y` bzw. `Z`: `<valaddress>`
Adresse des Effect-Eintrags von `Y` bzw. `Z`: `<effaddress>`

```
/*  1) Pruefung auf Mehrfachbelegung der Sensor/Input-Groessen  */
/*  ----------------------------------------------------------  */
a:  if ( SenElem (B.X.pValue) != Adr (B.X.Value) )  Error (40);
b:  if ( InElem  (H.X.pValue) != Adr (H.X.Value) )  Error (40);

/*  2) Belegung des Zeigers auf Wertezelle  */
/*  --------------------------------------  */
a:  if ( SenElem (B.X.pValue) = <valaddress>;
b:  if ( InElem  (H.X.pValue) = <valaddress>;

/*  3a) Aufbau der Verkettung  */
/*  -------------------------  */
a:  AddChain ( <effaddress>, Adr (B.X.Effect) );

/*  3b) Weitergabe des Effect-Zeigers  */
/*  ---------------------------------  */
b:  InElem (H.X.pEffect) = <effaddress>;
```

mit den Makros

```
InElem(VAR)          pData -> VAR
SenElem(VAR)         pData -> VAR
OutElem(VAR)         pData -> VAR
Adr(VAR)           & (pData -> VAR)
```

und den Belegungen für `<valaddress>` und `<effaddress>`:

| <**valaddress**> | Y Konstante | sonst |
|---|---|---|
| `AB.Y --> B.X;` | `Adr(A.Y)` | `Adr(A.Y.Value)` |
| `AH.Y --> B.X;` | `OutElem(A.Y.pValue)` | |
| `Z --> B.X;` | `OutElem(Z.pValue)` | |

| <**effaddress**> | Y Konstante | sonst |
|---|---|---|
| `AB.Y --> B.X;` | `NULL` | `Adr(A.Y.Effect)` |
| `AH.Y --> B.X;` | `OutElem(A.Y.pEffect)` | |
| `Z --> B.X;` | `OutElem(Z.pEffect)` | |

`AB` ist Basiskomponente
`AH` ist höhere Komponente

< Code für jede Initialisierung > :

a) `B.Y := value;` B sei Basiskomponente

```
Con (B.Y)    :=  <value>;        falls Y Konstante
Sta (B.Y)    :=  <value>;        falls Y Zustandsvariable
Dep (B.Y)    :=  <value>;        falls Y abh. Variable
SenVal (B.Y) :=  <value>;        falls Y Sensorvariable
```

b) `H.Y := value;` H sei höhere Komponente

```
OutCon (H.Y)  :=  <value>;        falls Y Konstante
OutSta (H.Y)  :=  <value>;        falls Y Zustandsvariable
OutDep (H.Y)  :=  <value>;        falls Y abh. Variable
InVal  (H.Y)  :=  <value>;        falls Y Sensorvariable
```

c) `Z := value;`

```
OutCon (Z)  :=  <value>;        falls Z Konstante
OutSta (Z)  :=  <value>;        falls Z Zustandsvariable
OutDep (Z)  :=  <value>;        falls Z abh. Variable
InVal  (Z)  :=  <value>;        falls Z Sensorvariable
```

mit den Makros

```
OutCon(VAR)        * (pData -> VAR.pValue)
OutSta(VAR)        * (pData -> VAR.pValue)
OutDep(VAR)        * (pData -> VAR.pValue)

InVal(VAR)           pData -> VAR.Value
```

### 6.3.3 Code für die Simulationsphase

Das Codesegment, das in den Basiskomponenten für die Simulationsphase angelegt wird, besitzt den Aufbau

```
/*  5.1  Merken des aktuellen Komponentennamens  */
/*  ------------------------------------------  */
    ActComp =  & (pDB -> NameElem);

/*  5.2  Erkennen von Crossings  */
/*  ---------------------------  */
    if (Mode == Detect)
    {
       < Code fuer jedes Crossing >
    }

/*  5.3  Modelldynamik  */
/*  ------------------  */
    < Code fuer Modellgleichungen >

    return;
```

Wird in der Modellbeschreibung eine Bedingung erkannt, an der kontinuierliche Variablen beteiligt sind, wird für dieses sogenannte Crossing eine extra Codezeile generiert (siehe *comparison*).

Der Code für die Modelldynamik enthält die Beziehungen zwischen den Modellgrößen, wie sie im Abschnitt DYNAMIC BEHAVIOUR vorgefunden werden. Es wird beim Aufruf der Basiskomponente immer nur ein Typ von Beziehung ausgewertet und dazu mit Hilfe der globalen Variablen *Mode* unterschieden, welche Abschnitte zu durchlaufen sind.

Die folgende Aufstellung zeigt, welcher Code für die verschiedenen Sprachkonstrukte angelegt wird.

algebraische Gleichungen:

```
Var :=  expression;          if (Mode == AlgEqu)
                             {
                                ActLine = n;
                                DepAsgnX ( Adr (VAR), <expression> );
                             }
```

Differentialgleichungen:

```
DIFFERENTIAL EQUATIONS      if (Mode == DtlEqu)
DO                          {
                               ActLine = n;
    Var' :=  expression;       StD (VAR) =  <expression>;
END                         }
```

Ereignis:

```
WHENEVER  expression        if (Mode == Events)
DO                          {
                               ActLine = n;
                               if ( <expression> )
                               {
                                  ActLine = n;
    Var^ :=  expression;          StaAsgnX ( Adr (VAR), <expression> );
                                  ActLine = n;
    DISPLAY ("string",            sprintf ( Message, "string", <expression> );
             expression);         Display ( Message );

                                  EvtCond = Yes;
                               }
END                         }
```

event_region_defining:

```
IF   expression         if ( ActLine = n  &&  SetRegion(i) = <expression> )
DO   ...   END          {  ...  }
ELSIF   expression      else if ( ActLine = n  &&  SetRegion(i) = <expression> )
DO   ...   END          {  ...  }
ELSE                    else
DO   ...   END          {  ...  }
```

mit der Makrodefinition:

```
SetRegion(i)   Region[RegNo(i)].ActRegion
```

Der Index `i` wird beginnend bei 1 innerhalb der Komponente durchgezählt.

algebraische Ausdrücke ( <expression> ) :

```
expression ::=  simple_expression   -->   <simple_expression>
              | comparison          -->   <comparison>
```

Vergleichsoperationen ( <comparison> ) :

```
comparison ::=  simple_expression   comp_op   simple_expression
                       (S1)                          (S2)
```

Falls innerhalb Ereignis oder (S1 und S2 diskret)

```
-->   ( <S1>  <comp_op>  <S2> )
```

sonst falls S1 die Zeit $T$ und S2 diskret

```
-->   TCompare  ( <k_comp_op> , <S2> )
```

sonst falls S2 die Zeit $T$ und S1 diskret

```
-->   TCompare  ( <k_inverse_comp_op> , <S1> )
```

sonst S1 kontinuierlich oder S2 kontinuierlich

```
-->   ( <S1>  <comp_op>  <S2> )
```

```
ausserdem:   < Anlegen eines Crossings >
```

*Anlegen eines Crossings:*

```
CroCheck ( <S1> , <k_comp_op> , <S2> ) ;
```

| *Vergleichsoperator* | <comp_op> | <k_comp_op> | <k_inverse_comp_op> |
|---|---|---|---|
| comp_op ::= '=' | '=' | | |
| '<>' | '!=' | | |
| '>' | '>' | 'gt' | 'le' |
| '>=' | '>=' | 'ge' | 'lt' |
| '<' | '<' | 'lt' | 'ge' |
| '<=' | '<=' | 'le' | 'gt' |

Die Bezeichnungen 'gt', 'ge', 'lt', 'le' sind Makros und stehen für ganze Zahlen, die als Kennung für die Vergleichsoperatoren dienen.

<simple_expression> :

```
simple_expression ::=  [ sign ]  addlist      <sign>  <addlist>

add_list ::=  term                            <term>
            | addlist add_op term             <add_list>  <add_op>  <term>

term ::=  factor                              <factor>
        | term mul_op factor                  <term>  <mul_op>  <factor>

add_op ::=  '+'                     -->       '+'
          | '-'                     -->       '-'
          | OR                      -->       '||'

mul_op ::=  '*'                     -->       '*'
          | '/'                     -->       '/'
          | AND                     -->       '&&'

factor :=  identifier                         <identifier>
         | unsigned_number                    <unsigned_number>
         | logic_value                        <logic_value>
         | function_call                      <function_call>
         | NOT factor               -->       '!' <factor>
         | '(' expression ')'       -->       '(' <expression> ')'
         | T                        -->       'Tc', falls kein Zeitvergleich
                                                    angelegt wird

logic_value ::=  TRUE               -->       'true'
               | FALSE              -->       'false'

funcion_call ::=   identifier       -->       IDENTIFIER
                 '('expression                '(' <expression>
                   {, expression }                { ',' <expression> }
                 ')'                          ')'

unsigned_number                     -->       C-Notation als
                                              long bzw. double Zahl
                                              je nach Typ des gesamten Ausdrucks

identifier     -->    Con(IDENT)              falls Konstante
                      Sta(IDENT)              falls Zustandsvariable
                      Dep(IDENT)              falls abh. Variable
                      Sen(IDENT)              falls Sensorvariable

                      Val('no')               falls Aufzaehlungswert
```

### 6.3.4 Code für Tabellenfunktionen

*Syntax:*

```
TABULAR FUNCTION   identifier   '(' REAL --> REAL ')'
CONTINUOUS
BY   interpolation_method   INTERPOLATION
ON   '(' number_list_1 ')'  -->  '(' number_list_2 ')'
```

*Code zur Deklaration der Tabellenfunktion:*

```
double IDENT () ;
```

*Code zur Definition der Tabellenfunktion:*

```
static  double  IDENT  (x)
double  x;
{
   static  double  Breaks []  =  { <number_list_1> };
   static  double  Values []  =  { <number_list_2> };

   static  short  N       =  < Anzahl Stuetzstellen >;
   static  short  Intpol  =  < interpolation_method >;

   return  ( TabFunc (N, Breaks, Values, Interpol) ) ;
}
```

*Anlegen der Zahlenfolge:*

```
number_list ::=  number {, number }     -->   <number>  {, <number> }

number ::=  [ +|- ]  unsigned_number    -->   [+|-]  DOUBLE_NUMBER
```

*Kennung für Interpolationsverfahren:*

```
interpolation_method ::=  LINEAR        -->   '1'
                        | CUBIC BESSEL  -->   '2'
                        | CUBIC SPLINE  -->   '3'
```

## 6.4 Die Aktivierungsphase

### 6.4.1 Die Aufgaben des Hauptprogramms

Der Simulationsprozeß wird vom Experimentierprozeß durch Ausführung der Systemprogramme

*fork* und *exec*

gestartet, womit gleichzeitig eine Pipe-Kommunikation über je einen Eingabe- und einen Ausgabekanal eröffnet wird.

Mit dem Aufruf des Hauptprogramms

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

werden die folgenden Argumente übergeben:

`argv[1]`: *Kanalnummer des Eingabekanals*
`argv[2]`: *Kanalnummer des Ausgabekanals*
`argv[3]`: *Restart* = wiederholter Start nach Fehler
*Normal* = gewöhnlicher Start des Simulators

Im Falle eines Neustarts, der unmittelbar auf einen eingetretenen Fehler folgt, ist eine erneute Initialisierung unerwünscht und wird durch ein gesetztes Flag unterdrückt.

Zur Aktivierung des Simulationsprozesses führt das Hauptprogramm die folgenden Schritte aus:

1.1 Festlegen der Reaktionen auf System-Signale

1.2 Öffnen der Protokolldatei `Structure.out`

1.3 Argumente des Programmaufrufs auswerten

1.3.1 Kanalnummern für Pipe-Kommunikation übernehmen

1.3.2 `ReStart`-Flag setzen

1.4 Speicherbereich für Ausführungsprofil anlegen

1.5 Speicherbereich für Ausführungsprofil initialisieren

1.6 Erlaubnis zum Senden von Meldungen erteilen

1.7 Prozeßzustand sichern:
Wiederaufsetzpunkt im Fehlerfall

1.8 Aufruf des Koppelprogramms *Model* falls Fehlerfreiheit

1.9 Erlaubnis zum Senden von Meldungen entziehen

1.10 Erfolgs- bzw. Fehlermeldung an Experimentierprozeß schicken

1.11 Abbruch der Aktivierungsphase im Fehlerfall

1.12 Speicherplatzbedarf der angelegten Datenbereiche protokollieren

1.13 Speicherplatzbedarf des Modells und Basisadresse an Experimentierprozeß schicken

1.14 Urinitialisierung an Experimentierprozeß schicken

1.15 Schließen der Protokolldatei *Structure.out*

1.16 Daten für Ausführungsprofil auf Datei *mon.out* ausgeben

Während der Simulationsprozeß abläuft, können (aus sehr unterschiedlichen Gründe) die im UNIX-Betriebssystem vereinbarten Systemsignale eintreffen. Falls man für das Eintreffen eines Signals keine Vorkehrungen trifft, wird der laufende Prozeß abrupt beendet. Die Funktion

```
SigHdl ()
```

vereinbart daher die Reaktionen für verschiedene eintreffende Signale:
Das Tastatur-Signal *CTRL C* beendet den Prozeß sofort, das Tastatur-Signal *CTRL* \ wird nur während der Simulationsphase freigegeben und dient zum kontrollierten Beenden eines Simulationslaufs. Alle übrigen Signale führen zu einem Aufruf der Fehlerbehandlungsroutine

```
Error (),
```

die eine Fehlermeldung ausgibt und den Simulationsprozeß kontrolliert beendet (siehe *Behandlung von Laufzeitfehlern*).

Auf der Datei *Structure.out* werden wichtige Angaben zum Speicherplatzbedarf sowie Fehlermeldungen während der Aktivierungsphase ausgegeben.

Um festzustellen, wie häufig jede Funktion eines Prozesses aufgerufen wurde und wieviel Rechenzeit sie beansprucht hat, stellt UNIX einen Überwacher zur Verfügung, der die Daten für das Ausführungsprofil sammelt. Die benötigten Datenbereiche werden nach dem Start eines Programms angelegt. Vor den zu kontrollierenden Programmabschnitten (Aktivierungsphase, Ausführung eines Simulationslaufs) werden diese Datenbereiche initialisiert und danach ihre Ausgabe auf die Datei *mon.out* veranlaßt.

Eventuelle Fehlermeldungen im weiteren Verlauf der Aktivierungsphase sollen auch im Arbeitsfenster des Bildschirms erscheinen. Zu diesem Zweck wird die Erlaubnis zum Senden von Meldungen über Pipe erteilt.

Vor dem Abstieg ins Modell wird ein Wiederaufsetzpunkt vereinbart. Im Fehlerfall wird der Programmfluß an dieser Stelle fortgesetzt.

Mit dem Aufruf des Koppelprogramms *Model* werden Datenbereiche mit modellspezifischer Dimensionierung angelegt und ein zweifacher Abstieg ins Modell veranlaßt. Es werden sämtliche Modellkomponenten durchlaufen und dabei die Zeiger der Sensorgrößen belegt und die Modellgrößen mit den Initialwerten des MDL-Quelltextes vorbelegt. Diesen ursprünglichen Modellzustand bezeichnen wir als *Urinitialisierung*.

Dem Experimentierprozeß wird mitgeteilt, wieviel Speicherplatz die Datenstruktur des Modells in Anspruch nimmt und anschließend diese Datenstruktur mit der Urinitialisierung übertragen.

### 6.4.2 Die Aufgaben des Koppelprogramms „Model“

Das Koppelprogramm *Model* legt die Datenbereiche mit modellspezifischer Dimensionierung an und stellt die Anbindung des Hauptprogramms an die höchste Modellkomponente her.

Um diese Aufgaben zu erfüllen, ist das Koppelprogramm für jedes Modell neu zu übersetzen. Das Programm hat den folgenden Aufbau:

1. Belegung der Präprozessor-Konstanten mit dem Namen des Modells und den Dimensionen verschiedener Datenbereiche (*dimensions.h*)
2. Deklaration der Modelldatenstruktur (*mod_struct.h*)
3. Funktionsdefinition

4. Übernehmen der Dimensionierung in globale Variablen
5. Speicherplatz für Felder mit modellspezifischer Dimensionierung anlegen
6. Speicherplatzbedarf für Modelldatenstruktur bestimmen und Speicherplatz anlegen
7. Aufruf der höchsten Modellkomponente (1. Abstieg ins Modell)
8. Aufruf der höchsten Modellkomponente (2. Abstieg ins Modell)

Die Header-Datei *dimensions.h* enthält den Namen der höchsten Modellkomponente in unterschiedlichen Schreibweisen:

```
# define  MODEL     COMPNAME
# define  mODEL     cOMPNAME
```

Nach Ersetzung durch den Präprozessor wird der Speicherplatzbedarf vom C-Compiler durch die Anweisung

```
sizeof (struct  mODEL)
```

geliefert. Die Datenstruktur `cOMPNAME` wurde in *mod_struct.h* vereinbart.

Die globale Variable

```
char * ModAdr
```

zeigt auf den Anfang des Modelldatenbereichs, nachdem dieser angelegt wurde.

Dem Aufruf der höchsten Modellkomponente

```
mODEL (ModAdr)
```

wird die Adresse der Modelldatenstruktur als Parameter mitgegeben.

Daneben enthält die Header-Datei *dimension.h* folgende modellspezifische Dimensionierungen als Präprozessor-Konstanten:

| | |
|---|---|
| `XBComp:` | Anzahl der Basiskomponenten im Modell |
| `XCState:` | Anzahl der kontinuierlichen Zustandsvariablen |
| `XCross:` | Anzahl der Crossings |
| `XRegion:` | Anzahl der Regions |
| `XTimeCmp:` | Anzahl der Zeitvergleiche |

Damit werden für die globalen Felder

| | |
|---|---|
| `FBComp[XBComp+1]:` | Funktionenzeiger der Basiskomponenten |
| `DBComp[XBComp+1]:` | Datenbereichszeiger der Basiskomponenten |
| `pCState[XCState+1]:` | Zeiger auf kontinuierliche Zustandsvariablen |
| `Cross[XCross+1]:` | Feld mit Crossingvermerken |
| `Region[XRegion+1]:` | Feld mit Region-Vermerken |
| `TimeCmp[XTimeCmp+1]:` | Feld mit Zeitvergleichen |
| `AlgEffect[XBComp+1]:` | Anzeige bei Auswirkungen auf alg. Gleichungen |
| `EvtEffect[XBComp+1]:` | Anzeige bei Auswirkungen auf Ereignisse |
| `DtlEffect[XBComp+1]:` | Anzeige bei Auswirkungen auf Differentialgleichungen |
| `TimeEffect[XBComp+1]:` | Anzeige der Auswirkungen einer Zeitfortschaltung |

die entsprechenden Datenbereiche angelegt.

### 6.4.3 Der zweimalige Abstieg ins Modell

Es erfolgt ein zweimaliger Abstieg ins Modell, bei dem jede Modellkomponente mit dem ihr zugeordneten Datenbereichszeiger aufgerufen wird. Dabei sind die folgenden Aufgaben auszuführen:

1. Mitteilung der Offset-Adressen von Subkomponenten und Modellgrößen an Experimentierprozeß
2. Abspeichern der Komponentennamen
3. Abspeichern der Funktions- und Datenbereichsadressen aller Basiskomponenten

4. Urinitialisierung der Modellgrößen
5. Belegung der Zeiger für Sensorgrößen
6. Belegung und Verzeigerung der Effect-Einträge
7. Festlegen der Basisindizes für Crossings, Regions und Zeitvergleiche

Die im einzelnen auszuführenden Aufgaben werden im folgenden näher beschrieben:

**a) Mitteilung der Offsetadressen von Subkomponenten und Modellgrößen an Experimentierprozeß**

Der Modelldatenbereich wird zum Zweck der Vorbesetzung und der Ergebnisaufbereitung zum Experimentierprozeß übertragen und umgekehrt. Damit der Experimentierprozeß die Modellgrößen unter den Bezeichnungen auffinden kann, wie sie der Benutzer im MDL-Text vergeben hat, ist es notwendig, dem Experimentierprozeß die Namen aller Modellkomponenten und aller Modellgrößen sowie deren Adressen mitzuteilen.

Während die Namen mit ihren Datentypen von einer eigens dafür erzeugten Datei (`offset.dat`) gelesen werden können, müssen die Adressen durch ein C-Programm generiert werden. Diese Aufgabe übernimmt ein Programmabschnitt, der ebenfalls auf Datei (`offset.incl`) ausgelagert ist und während der Aktivierungsphase ausgeführt wird.

Beim erstmaligen Aufruf einer Komponentenklasse wird zum Experimentierprozeß übertragen (Unterprogramm `SendAdr`):

1. Info-Code: „Address“
2. Name der Modellkomponente
3. Offset-Adressen aller Subkomponenten und Modellgrößen (Offset-Adressen beziehen sich auf den Beginn einer Modellkomponente, d.h. die erste Variable hat die Adresse Null)
4. Endekennung (-1)

Parallel hierzu liest der Experimentierprozeß die für jede Modellkomponente betreffende Datei `offset.dat` mit den Informationen:

1. Name der Modellkomponente
2. Anzahl zu erwartender Einträge
3. Namen und Werte von Aufzählungsmengen
4. Typen und Namen der Subkomponenten zuzüglich des Klassennamens
5. Typen und Namen der Modellgrößen, ggf. zuzüglich der Wertemenge

Aus diesen Informationen kann der Experimentierprozeß umkehrbar eindeutige Zuordnungen zwischen dem Pfadnamen einer Modellgröße und ihrer Adresse innerhalb des Modelldatenbereichs herstellen. Dadurch ist gewährleistet, daß bei allen Ein- und Ausgabefunktionen, welche die Experimentierumgebung bereitstellt, mit den Bezeichnern des Benutzers gearbeitet werden kann.

### b) Abspeichern der Komponentennamen

Um Laufzeitfehler gut lokalisieren zu können, sollte die Fehlermeldung den vollständigen Namen der Komponente sowie die Zeilennummer enthalten.

Im Datenbereich jeder Modellkomponente wird daher der Name der Komponente sowie ein Zeiger auf den Namen der übergeordneten Komponente abgelegt. Hierzu wird die Struktur `NAME_ELEM` eingeführt

```
struct   NAME_ELEM
{
  char               CompName[9];     Name der Komponente
  struct NAME_ELEM * SuperComp;       Zeiger auf Namen der
                                      Superkomponente
}
```

und während der Aktivierungsphase, beginnend mit den Basiskomponenten, belegt. Jede Komponente trägt ihren eigenen Klassennamen ein und initialisiert den Zeiger mit NULL. Für ihre Subkomponenten überschreibt sie den Klassennamen mit dem Ausprägungsnamen und trägt den Zeiger auf ihre eigene Namens-Struktur ein.

```
SUBCOMPONENTS   B1  OF CLASS  B,
                B2  OF CLASS  B
```

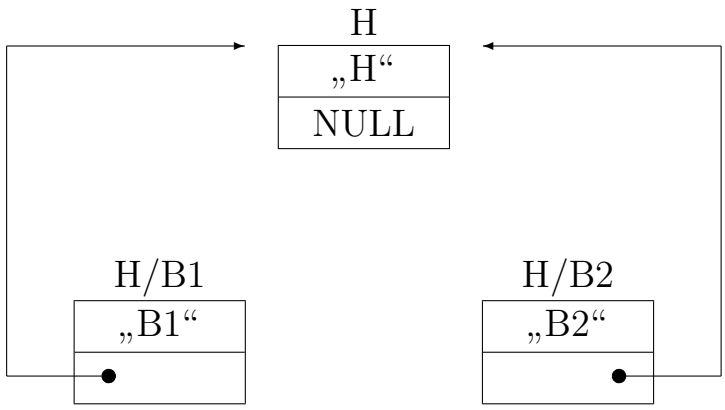



Abb. 6.4.3-1: Verzeigerung der Namenseinträge

Während des Modellablaufs verfügt jede Basiskomponente über ihre eigene Namens-Struktur und kann sich über die Verzeigerung ihren kompletten Pfadnamen konstruieren.

### c) Abspeichern der Funktions- und Datenbereichsadressen aller Basiskomponenten

Während der Simulationsphase werden nur noch die Basiskomponenten eines Modells durchlaufen. Hierzu werden die Zeiger auf die Funktionen sowie auf die Datenbereiche der Basiskomponenten in den Feldern `FBComp` bzw. `DBComp` abgelegt.

### d) Urinitialisierung der Modellgrößen

Die Initialisierung wird von unten nach oben vorgenommen, d.h. zuerst bei den Basiskomponenten, danach bei den übergeordneten Komponenten. Dies ist notwendig, da die Initialisierung der höheren Komponenten die der untergeordneten überschreiben soll. Ist im MDL-Text kein Initialwert für abhängige Größen oder Sensorgrößen angegeben, wird mit Null initialisiert.

### e) Belegung der Zeiger für Sensorgrößen

Die Sensorgrößen der Basiskomponenten stehen als Platzhalter für Modellgrößen fremder Komponenten. Ihre Identität beschreibt ein Verbindungspfad, der in höheren Komponenten angelegt wird. Während der Aktivierungsphase werden die Zeiger der Sensorgrößen mit den Adressen der ihnen zugeordneten Modellgrößen belegt.

Zu diesem Zweck wird der Verbindungspfad von der Modellgröße zur Sensorgröße verfolgt. Ist kein Pfad angegeben, verweist der Zeiger auf den Initialwert der Sensorgröße.

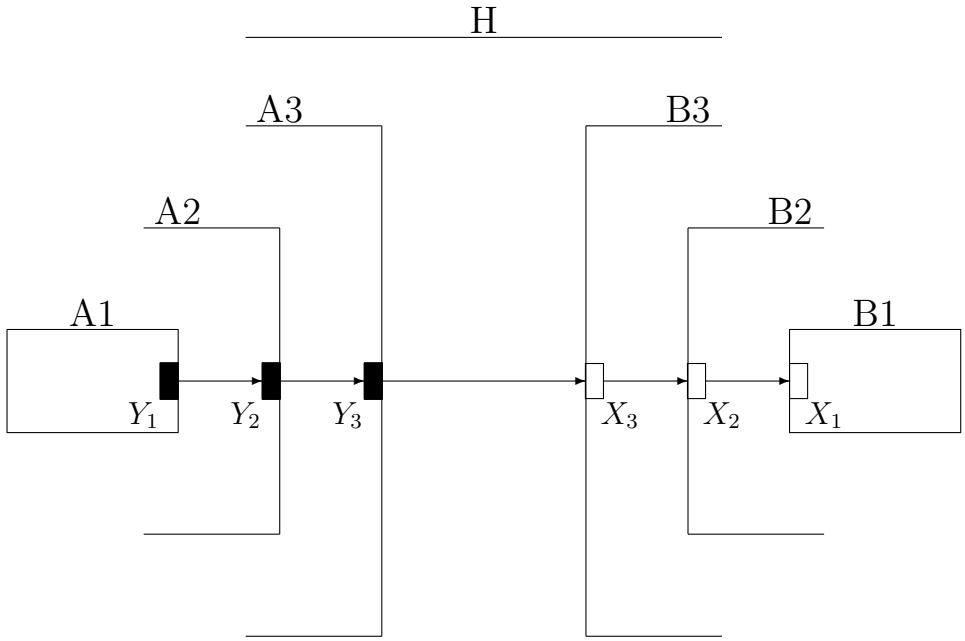



Abb 6.4.3-2: Verbindungspfad zwischen zwei Basiskomponenten

A1 und B1 seien Basiskomponenten. Ziel des Verbindungsaufbaus ist es, den Zeiger der Sensorgröße X1 auf die Größe Y1 zu legen. Von den Datenstrukturen der beteiligten Größen, interessieren uns diejenigen Elemente, die in der folgender Abbildung enthalten sind:

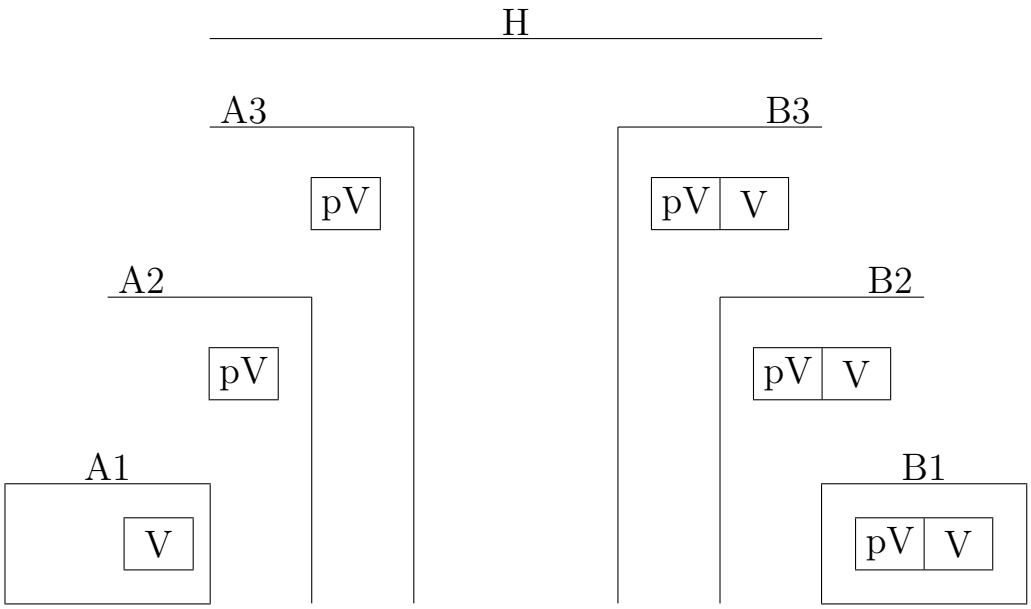



Abb. 6.4.3-3: Datenstrukturelemente zum Aufbau eines Verbindungspfads

Mit V wird der Wert, mit pV der Zeiger auf einen Wert symbolisiert. Während der Aktivierungsphase wird der Modellbaum zweimal traversiert und dabei die Komponenten in folgender Reihenfolge durchlaufen:

1. H → A3 → A2 → A1 → A2 → A3 →
   → B3 → B2 → B1 → B2 → B3 →
2. H → A3 → A2 → A1 → A2 → A3 →
   → B3 → B2 → B1 → B2 → B3 → H

Während des Aufstiegs beim ersten Durchlauf werden die Output Equivalences geschlossen und die Zeiger der Sensorgrößen vorbelegt.

| | |
|---|---|
| Output Equivalence: | `A2.pV = Adr (A1.V)` |
| | `A3.pV = A2.pV` |
| Vorbelegung der Sensor-Zeiger: | `B1.pV = Adr (B1.V)` |
| | `B2.pV = Adr (B2.V)` |
| | `B3.pV = Adr (B3.V)` |

Diese Vorbelegung bewirkt, daß der Zeiger einer nicht angeschlossenen Sensorgröße auf den eigenen Initialwert verweist.

Während des Abstiegs beim zweiten Durchlauf werden die Component Connections und Input Connections geschlossen.

| | |
|---|---|
| Component Connections: | `B3.pV = A3.pV`<br>weitere, nicht eingezeichnete Möglichkeit:<br>`B3.pV = Adr (A1.V)` |
| Input Connections: | `B2.pV = B3.pV`<br>`B1.pV = B2.pV` |

Am Ende der beiden Durchläufe sind alle Zeiger `pV` auf den Wert `V` gerichtet.

### f) Belegung und Verzeigerung der Effect-Einträge

Zur Verbesserung des Laufzeitverhaltens (siehe Abschnitt *Die Simulationsphase*) ist es erforderlich festzuhalten, welche Auswirkungen die Veränderung einer Modellgröße nach sich zieht. Zu diesem Zweck wird jeder Zustandsvariablen, abhängigen Variablen und Sensorvariablen die Datenstruktur

```
struct  EFFECT
{
   unsigned short    CompNo;    Komponentennummer
   unsigned short   EffFlag;    betroffene Aktionstypen
   struct EFFECT * eff_next;    Zeiger auf weitere Auswirkungen
}
```

beigeordnet. Die Einträge in dieser Struktur zeigen an, in welcher Komponente die Variable verwendet wird und auf welche Aktionstypen (algebraische Gleichungen, Ereignisse, Differentialgleichungen) eine Wertänderung dieser Variablen Einfluß hätte. Die betreffende Information liefert der MDL-Compiler. Sie wird während der Aktivierungsphase eingetragen.

Ist die Modellgröße über einen Verbindungspfad an eine oder mehrere Sensorvariablen angeschlossen, führt dies zu Auswirkungen in den betreffenden fremden Komponenten. Zu diesem Zweck wird während der Aktivierungsphase eine Liste aufgebaut, welche die EFFECT-Strukturen miteinander verbindet. Durch Verfolgung dieser Liste lassen sich alle Auswirkungen einer Modellgröße feststellen.

**Beispiel:**
Die abhängige Variable $Y$ der Komponente $A$ sei mit den Sensorgrößen $X$ der Komponenten $B1$ und $B2$ verbunden.

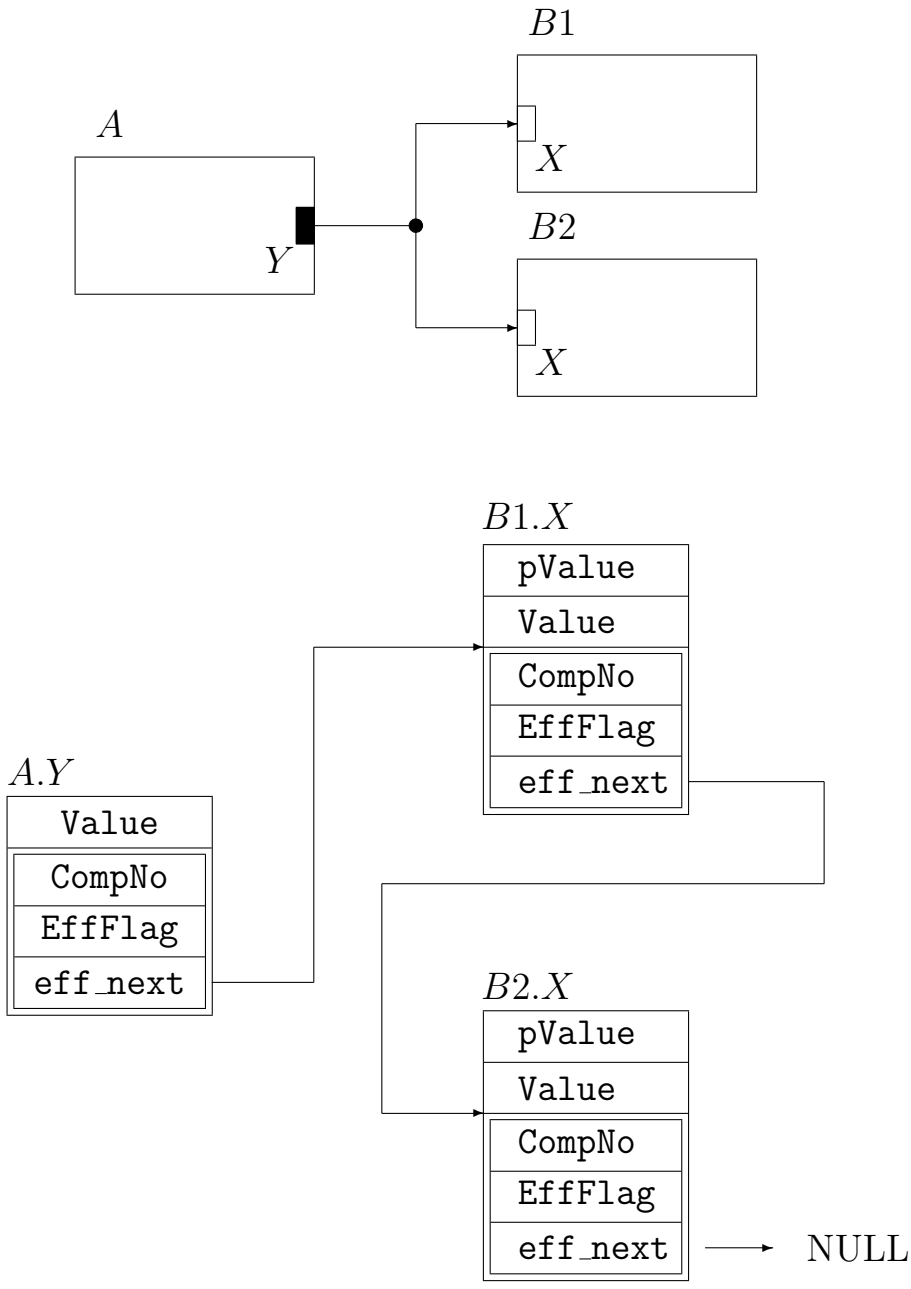



Abb. 6.4.3-4: Verzeigerung der EFFECT-Einträge

Das Anhängen der EFFECT-Struktur einer Sensorvariablen an die (evtl. noch nicht bestehende) Liste besorgt die Funktion

```
void   AddChain (pLstEffect, pNewEffect)

register struct EFFECT * pLstEffect;
register struct EFFECT * pNewEffect;
```

welcher der Zeiger auf das Listenende und der Zeiger auf die anzuhängende EFFECT-Struktur mitgegeben wird.

Die Aufrufe dieser Funktion werden in Verbindung mit den *Component Connections* bzw. *Input Connections* generiert.

Die Fortschaltung der Zeit als standardmäßige Modellgröße muß ebenfalls erfaßt werden. Hierzu enthält auch jeder Zeitvergleich eine EFFECT-Struktur.

Da die Zeit auch als kontinuierliche Größe verwendet werden kann, besitzt jede Basiskomponente eine EFFECT-Struktur `TimeEff`, die Auswirkungen einer Zeitfortschaltung anzeigt. Auch in diesem Fall liefert der MDL-Compiler die nötige Information, welche während der Aktivierungsphase eingetragen wird.

### g) Festlegen der Basisindizes für Crossings, Regions und Zeitvergleiche

Für Crossings, Regions und Zeitvergleiche (siehe später) wird je ein globales Feld von Strukturen angelegt, deren Dimensionierung in der Datei `dimensions.h` vermerkt ist. Jeder Modellkomponente wird aus diesem Feld ein Abschnitt zugeteilt, der die notwendige Anzahl an Feldelementen umfaßt.

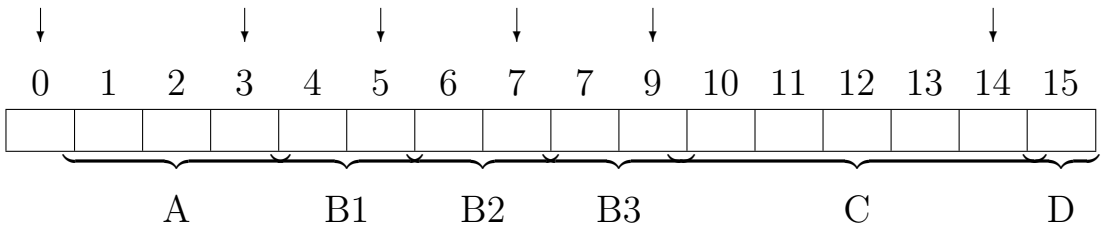



Abb. 6.4.3-5: Vergabe der Feldelemente

Im Modelldatenbereich jeder Komponente werden in den Variablen

```
CroBase, RegBase, TCmpBase
```

die um 1 verminderten Anfangsindizes des jeweiligen Abschnitts abgelegt. Hierfür sind die globalen Zählvariablen

```
CroCount, RegCount, TCmpCount
```

erforderlich.

Der Zugriff auf das i-te Feldelement innerhalb einer Komponente (Numerierung beginnt bei 1) erfolgt über die Makros:

```
CroNo(i)    pData --> CroBase + i
RegNo(i)    pData --> RegBase + i
TCmpNo(i)   pData --> TCmpBase + i.
```

Das Feldelement 0 wird nicht verwendet.

## 6.5 Die Simulationsphase

Während der Simulationsphase werden einzelne Simulationsläufe ausgeführt. Ein Anfangszustand wird eingelesen und über eine Reihe von Zwischenzuständen in einen Endezustand überführt.

Der zugrundeliegende Algorithmus zur Auswertung der Modellbeziehungen wurde bereits in Kapitel 3 entwickelt. In diesem Abschnitt wird die Implementierung dieses Algorithmus besprochen. Insbesondere ist das Zusammenspiel zwischen der Ablaufsteuerung und dem aus der Modellspezifikation generierten C-Code sowie Maßnahmen zur Verbesserung der Effizienz von Interesse.

Nach einem Überblick über den Aufbau des Laufzeitsystems und die vorbereitenden Tätigkeiten des Hauptprogramms, besprechen wir zunächst die Ablaufsteuerung für rein diskrete Modelle und anschließend für kontinuierliche sowie kombinierte Modelle. Es wird auf die verschiedenen Aufzeichnungsverfahren und auf die Behandlung von Laufzeitfehlern eingegangen.

### 6.5.1 Der Aufbau des Laufzeitsystems

Das Laufzeitsystem enthält die zur Ausführung eines Simulationslaufs notwendigen Funktionen. Es setzt sich zusammen aus den Funktionen zur

- *Ablaufsteuerung:*

| | |
|---|---|
| `FlowCtl`: | Ausführung diskreter Zustandsübergänge |
| `IntCtl`: | Ausführung kontinuierlicher Zustandsübergänge |
| `AlgSolve`: | Iterative Ausführung statischer Beziehungen |
| `CroCheck`: | Erkennen von Crossings |
| `RegCheck`: | Erkennen von Regions |
| `TCompare`: | Zeitvergleich ausführen |
| `EnterEffect`: | Auswirkungen einer Zustandsänderung vormerken |
| `IntHdl`: | Unterbrechung über Tastatur bearbeiten |

- *Integration von Differentialgleichungen:*

  `IntStep`: Ausführen eines Integrationsschritts

  `Euler`: Integrationsverfahren nach Euler

  `RK4`: Runge-Kutta-Verfahren 4. Ordnung

  `RKF6`: Runge-Kutta-Fehlberg-Verfahren 5./6. Ordnung

  `RKIMP6`: implizites Runge-Kutta-Verfahren 6. Ordnung

  `Extpol6`: Extrapolationsverfahren nach Bulirsch-Stoer 6. Ordnung

  `RoWa`: Rosenbrock-Wanner-Verfahren 4. Ordnung

- *Vorbereiten der Zustandsübergänge:*

  `StaAsgnI`: Wertzuweisung an ganzzahlige Zustandsvariable

  `StaAsgnD`: Wertzuweisung an reell-diskrete Zustandsvariable

  `StaAsgnC`: Wertzuweisung an reell-kontinuierliche Zustandsvariable

  `StaAsgnL`: Wertzuweisung an logische Zustandsvariable

  `StaAsgnE`: Wertzuweisung an Zustandsvariable mit Aufzählungsmenge

- *Ausführen statischer Beziehungen:*

  `DepAsgnI`: Wertzuweisung an ganzzahlige abh. Variable

  `DepAsgnR`: Wertzuweisung an reelle abh. Variable

  `DepAsgnL`: Wertzuweisung an logische abh. Variable

  `DepAsgnE`: Wertzuweisung an abh. Variable mit Aufzählungsmenge

- *Auswertung von Tabellenfunktionen:*

  `TabCoeff`: Berechnung der Koeffizienten zur Interpolation

  `TabFunc`: Interpolation einer Tabellenfunktion

- *Auswertung reellwertiger Standardfunktionen:*
  Datei `RealFunc.c`: Implementierung der Standardfunktionen

- *Beobachtung von Modellgrößen:*
  `MonRead`: Einlesen der Monitorspezifikation
  `Monitor`: Aufzeichnen des Modellverhaltens
  `MonWrite`: abschließende Einträge in Inhaltsverzeichnisdatei
  `WriteSim.c`: Funktionen zum Beschreiben der Monitordatei

- *Sonstige Funktionen:*
  `Display`: Ausführen der Display-Anweisung
  `Read`: Lesen von Pipe
  `Write`: Schreiben auf Pipe
  `Error`: Behandlung von Laufzeitfehlern
  `Prtkol`: Ausgabe von Protokollmeldungen
  `CtlRead`: Einlesen der Steuerparameter
  `Statist`: Ausgabe der Ausführungsstatistik

Alle Funktionen sind als Objektcode in der Bibliothek `librts.a` abgelegt.

Einige weitere Funktionen, die ebenfalls in der Laufzeitbibliothek enthalten sind, werden während der Aktivierungsphase benötigt:

`SigHdl`: Reaktion auf Hardware-Interrupts festlegen

`SendAdr`: Offset-Adressen an Experimentierprozeß senden

`AddChain`: Aufbau der Auswirkungslisten

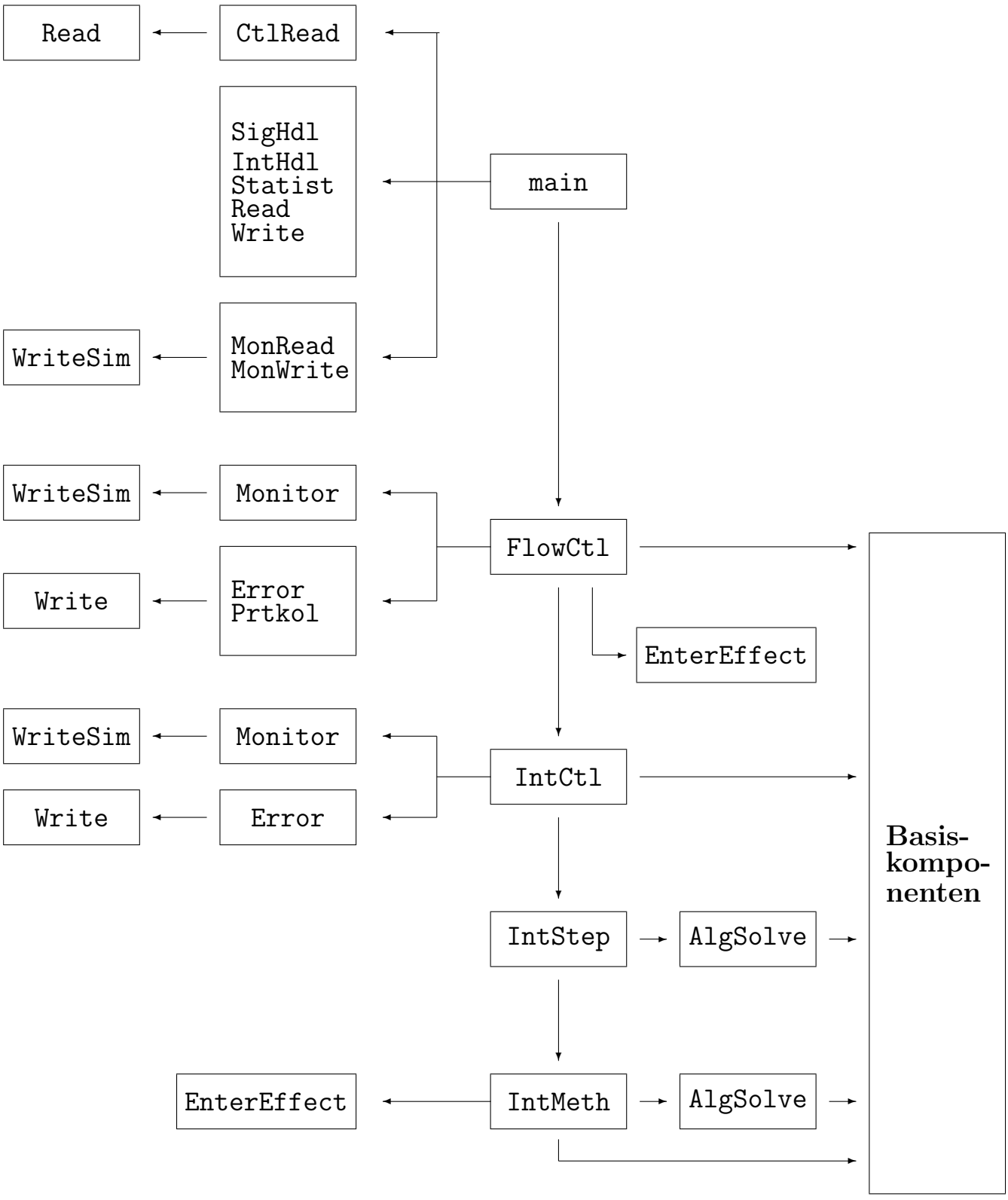
Die beiden Header-Dateien

`rts_struct.h`: Definition der Datentypen und Datenstrukturen

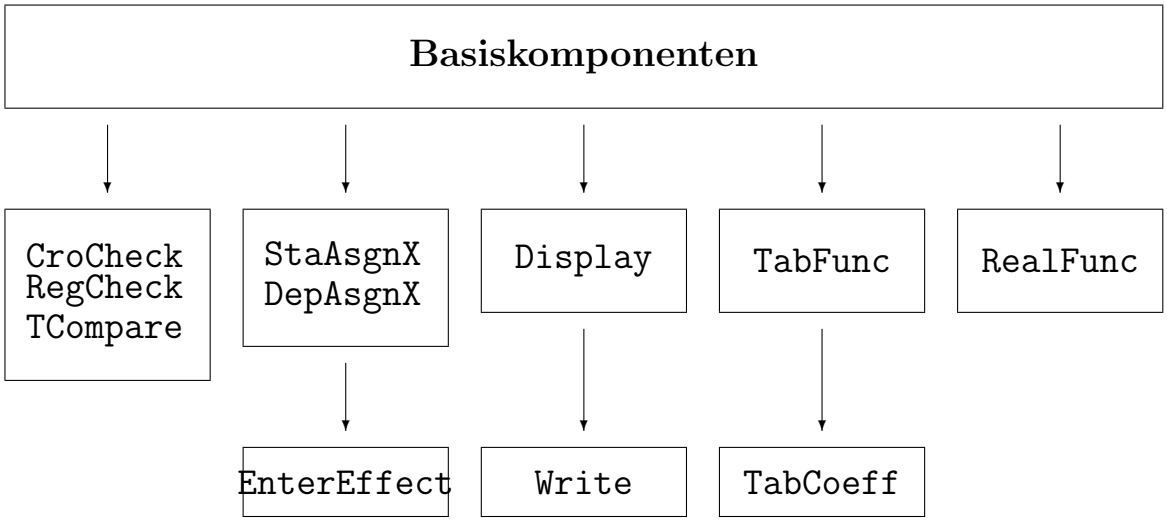
`rts_extern.h`: Deklaration der globalen Variablen

sind in sämtliche Funktionen eingebunden.

Die folgende Übersicht zeigt, welche Funktionen einander aufrufen:



Die Basiskomponenten rufen wiederum – meist sehr kurze – Funktionen auf:




### 6.5.2 Der Rahmen für den Modellablauf im Hauptprogramm

Nach Ausführung der Aktivierungsphase ist der Simulationsprozeß bereit zur Durchführung von Simulationsläufen. Er betritt eine Endlos-Schleife und wartet auf eine Meldung vom Experimentierprozeß. Diese Meldung veranlaßt den Start oder die Fortsetzung eines Simulationslaufs oder das Beenden des Simulationsprozesses.

2. Simulationsphase

——— Schleifenanfang ———

2.1 Meldung vom Experimentierprozeß entgegennehmen

`Start`: Starten eines neuen Simulationslaufs

`Continue`: Fortsetzen eines Simulationslaufs

`Termin`: Beenden des Simulationsprozesses

2.2 Verlassen der Schleife bei Ende-Meldung

2.3 Öffnen der Protokolldatei `Protocol.out`

2.4 Initialisierung der Modelldaten

2.5 Modellablauf

2.6 Ausgabe der Ergebnisse

2.7 Schließen der Protokolldatei `Protocol.out`

2.8 Schleife bei unkontrollierbarem Fehler verlassen

Schleifenende

Anhand der Protokolldatei `Protocol.out` kann der Benutzer die Geschehnisse während der Simulationsphase nachvollziehen.

Im Falle eines nicht kontrollierbaren Laufzeitfehlers (siehe *Fehlerbehandlung*) wird der Simulationsprozeß beendet und anschließend versucht, von neuem zu aktivieren. Dadurch wird der zuletzt erreichte Modellzustand, der zum Fehler führte, wieder hergestellt.

Mit dem Beenden der Simulationsphase, wird auch das Hauptprogramm des Simulationsprozesses verlassen. Der Simulationsprozeß kann auch in der Funktion `Error` beendet werden, falls ein Mehrfachfehler vorliegt (siehe *Die Behandlung von Laufzeitfehlern*).

Die Initialisierung der Modelldaten umfaßt:

2.4.1 Initialisieren der Ausführungsstatistik

2.4.2 Initialisieren der Zeitvergleiche

2.4.3 Einlesen der Anfangszeit und des Anfangszustands

2.4.4 Einlesen der Steuerparameter

2.4.5 Einlesen der Monitorbeschreibung

2.4.6 Speicher zum Aufnehmen des Ausführungsprofils löschen

Über die Anzahl der ausgeführten Zustandsübergänge sowie die Genauigkeit der Integrationsschritte wird Buch geführt (Ausführungsstatistik) und am Ende des Simulationslaufs in der Datei `Statistics.out` ausgegeben.

Von den globalen Modelldatenbereichen sind die Einträge der Zeitvergleiche so zu initialisieren, daß der ganzzahlige und der reelle Wert für den eingetragenen Zeitpunkt übereinstimmen. Der Einfachheit halber werden beide Werte mit Null vorbesetzt.

Der Anfangszustand und die dazugehörige Anfangszeit werden vom Experimentierprozeß empfangen. Das gleiche gilt für die Steuerparameter.

Die Informationen über die Beobachter werden der Datei `Monitor.cont` entnommen.

Über die Ausführungszeit der einzelnen C-Funktionen werden intern Aufzeichnungen geführt (Ausführungsprofil), die beim Übersetzen der Programmteile des Simulationsprozesses mit der Compiler-Option `-p` aktiviert werden. Der Datenbereich für diese Aufzeichnungen wird in der Initialisierungsphase rückgesetzt und bei der Ergebnisaufbereitung auf Datei `mon.out` ausgegeben (Funktion: `monitor`).

Der Modellablauf umfaßt:

2.5.1 Erlaubnis zum Senden von Meldungen erteilen

2.5.2 Erlaubnis zur Unterbrechung des Simulationslaufs erteilen

2.5.3 Prozeßzustand sichern (Wiederaufsetzpunkt im Fehlerfall)

2.5.4 Ablaufsteuerung aufrufen (bei Fehlerfreiheit)

2.5.5 Erlaubnis zur Unterbrechung des Simulationslaufs entziehen

2.5.6 Aktuelle Simulationszeit anzeigen

2.5.7 Erlaubnis zum Senden von Meldungen entziehen

2.5.8 Erfolgs- bzw. Fehlermeldungen ans Experimentiersystem schicken

`Success`: erfolgreich beendet

`Error`: kontrollierbarer Laufzeitfehler aufgetreten

`Fatal`: unkontrollierbarer Laufzeitfehler aufgetreten

Während des Modellablaufs ist der Experimentierprozeß empfangsbereit für Meldungen, die am Bildschirm ausgegeben werden sollen. Dementsprechend wird eine Sendeerlaubnis erteilt.

Der Simulationslauf darf jederzeit über Tastatur mit (CTRL \) unterbrochen werden. Hierzu wird das Signal 'SIGQUIT' freigegeben.

Im Falle eines Laufzeitfehlers kann der Modellablauf nicht ordnungsgemäß fortgesetzt werden. Um nach der Ausgabe der Fehlermeldung einen gesicherten Wiederaufsetzpunkt zu haben, wird unmittelbar vor dem Aufruf der Ablaufsteuerung der Prozeßzustand gesichert.

Die Durchführung der Simulation durch Aufrufen der Ablaufsteuerung erfolgt daher unter der Bedingung, daß kein Laufzeitfehler vorausging.

Nach der Rückkehr aus der Ablaufsteuerung wird der Endezeitpunkt am Bildschirm ausgegeben und die erteilten Genehmigungen zurückgenommen.

Eine Nachricht ans Experimentiersystem informiert dieses über das erreichte Ende der Simulation und ob diese erfolgreich oder mit Fehler beendet wurde.

Abschließend werden die Ergebnisse an den Experimentierprozeß bzw. auf Datei ausgegeben.

#### 2.6.1 Monitor abschließen

#### 2.6.2 Endezeit und Endezustand an Experimentierprozeß übergeben

#### 2.6.3 Ausgeben der Ausführungsstatistik auf Datei `Statistics.out`

#### 2.6.4 Ausgeben des Ausführungsprofiles auf Datei `mon.out`

Die Darstellung und Weiterverarbeitung der Ergebnisse übernimmt der Experimentierprozeß.

### 6.5.3 Die Ablaufsteuerung für diskrete Zustandsübergänge (FlowCtl)

Zur Durchführung der Simulation ruft das Hauptprogramm die Ablaufkontrolle *FlowCtl* auf. Sie besitzt den folgenden Aufbau:

1. Vorbelegungen
2. Zustandsübergänge vom Anfangszustand bis Endezustand ausführen

   —————— Schleife mit Zeitfortschaltung (Zeitschleife) ——————

   2.1 ...
   2.2 Diskrete Zustandsübergänge zum aktuellen Zeitpunkt ausführen

      1. ——— Schleife mit Taktfortschaltung (Ereignisschleife) ———
         1.1 Statische Beziehungen auswerten
            1. ...
            2. ———————— Iterationsschleife ————————
               2.1 Verlassen der Iterationsschleife bei Endebedingung
               2.2 Zykluszähler erhöhen
               2.3 ...
               2.4 Abhängige Variablen bestimmen
               —————— Ende der Iterationsschleife ——————
         1.2 Differentialquotienten bestimmen
         1.3 (Erweiterten) Modellzustand aufzeichnen
         1.4 Ereignisbedingungen prüfen und diskreten Zustandsübergang vorbereiten
         1.5 Verlassen der Ereignisschleife bei Endebedingung
         1.6 Taktfortschaltung
         1.7 Diskreten Zustandsübergang ausführen
         ———————— Ende der Ereignisschleife ————————
      2. ...

   2.3 Verlassen der Zeitschleife bei Endebedingung
   2.4 Kontinuierliche Zustandsübergänge ausführen bis Unstetigkeit eintritt (Zeitfortschaltung bzw. Integration)

   ———————— Ende der Zeitschleife ————————

Die Ablaufkontrolle ist die Implementation des in Abschnitt 3.5.2 beschriebenen Algorithmus. Für rein diskrete Modelle enthält *FlowCtl* die vollständige Implementation, zur Ausführung kontinuierlicher Zustandsübergänge bedient sie sich des Unterprogramms *IntCtl*.

Die Ablaufsteuerung *FlowCtl* besteht aus drei ineinander geschachtelten Schleifen. Die innerste Schleife dient der Auswertung statischer Beziehungen durch Iteration der algebraischen Gleichungen. Die mittlere Schleife stellt den Endezustand eines Zeitpunkts her, indem solange diskrete Zustandsübergänge (Ereignisse) ausgeführt werden, bis keine Auslösebedingung mehr erfüllt ist. Die äußerste Schleife führt einen Zustandsübergang über die Zeit aus.

Im Falle eines diskreten Modells reicht es, nur die Zeit fortzuschalten. Sind auch kontinuierliche Variablen im Modell, wird deren Zeitverlauf vom Unterprogramm *IntCtl* durch numerische Integration bestimmt. Die Zeitfortschaltung wird bis zu dem Zeitpunkt ausgeführt, zu dem ein algebraischer Ausdruck im Modell eine Unstetigkeitsstelle passiert. Hierzu zählen insbesondere auch Zeitbedingungen wie `T >= 50`. Dann besteht wieder die Möglichkeit einer diskreten Zustandsänderung.

#### 6.5.3.1 Auswertung der statischen Beziehungen

Nach einer Veränderung von Zustandsgrößen oder der Zeit $T$ sind die abhängigen Variablen neu zu bestimmen.

Da auf die Ausführungsreihenfolge der algebraischen Gleichungen kein Einfluß genommen werden kann und abhängige Größen wiederum auf anderen abhängigen Größen beruhen können, werden die statischen Beziehungen im Modell iterativ ausgewertet, bis sich die Werte der abhängigen Variablen nicht weiter verändern.

Der Algorithmus zur Durchführung der Iteration besitzt die folgende Gestalt:

1. Initialisierung
   `Mode = AlgEqu;  Cycle = 0;`

2. ——————————— Iterationsschleife ———————————
   ```
   while (ExecAlg)
   {
   ```

   2.1 Schleife ausführen, solange Auswirkungen
   auf abh. Variablen vorliegen

   2.2 Zyklus-Zähler erhöhen
   Fehler anzeigen bei Überschreitung der max. Zyklenzahl
   ```
   Cycle = Cycle + 1;
   if (Cycle > MaxCyc)  Error (1);
   ```

   2.3 Flag, das Auswirkungen auf abh. Variable
   in irgendeiner Komponente anzeigt, rücksetzen
   ```
   ExecAlg = No
   ```

   2.4 Abhängige Variablen bestimmen

   ——————— Schleife über alle Basiskomponenten ———————
   ```
   for (i = 1;  i <= NBComp;  i ++)
   {
   ```
   Falls Auswirkungen auf Basiskomponente `i`
   ```
   if (AlgEffect [i])
   {
   ```
   Auswirkungs-Flag rücksetzen
   ```
   AlgEffect [i] = No;
   ```
   Aufruf der Basiskomponente `i` und
   Ausführung der alg. Gleichungen
   ```
   (* FBComp [i]) (DBComp [i]);
   }
   }
   ```
   ——————— Ende der Komponentenschleife ———————

   ```
   }
   ```
   ——————————— Ende der Iterationsschleife ———————————

3. Eintrag in Ausführungsstatistik vornehmen

Algorithmus 6.5.3-1: Auswertung der statischen Beziehungen

Durch das Setzen der Steuervariablen

```
Mode = AlgEqu;
```

werden die Basiskomponenten dazu veranlaßt, nur die algebraischen Gleichungen auszuwerten.

Der Aufruf der Basiskomponente i erfolgt durch die Anweisung

```
(* FBComp[i]) (DBComp[i]);
```

Der i-te Funktionszeiger `FBComp` enthält die Funktionsadresse der i-ten Basiskomponente und der i-te Datenbereichszeiger `DBComp` verweist auf den Datenbereich dieser Basiskomponente.

Innerhalb der Basiskomponenten wird für eine statische Beziehung

```
DepVar := expression;
```

folgender Code angelegt:

```
if (Mode == AlgEqu)
{
   ActLine = n;
   DepAsgnX (Adr (<DepVar>), <expression>);
}
```

Der für den `expression` generierte C-Code wird durch `<expression>` gekennzeichnet.

Je nach Wertemenge der Variablen `DepVar` steht `DepAsgnX` für die Funktionen `DepAsgnI, DepAsgnR, DepAsgnL` bzw. `DepAsgnE`.

Die Funktion `DepAsgnX` prüft, ob der Wert des Ausdrucks `expression` vom bisherigen Wert der abhängigen Variablen `DepVar` verschieden ist. In diesem Fall wird `DepVar` der neue Wert zugewiesen und durch Aufruf der Funktion `EnterEffect` mögliche Auswirkungen dieser Wertänderung festgestellt.

Der Aufbau der Funktion `DepAsgnX` sei beispielhaft für ganzzahlige Variable angegeben:

```
void     DepAsgnI  (pDepI, ExprI)
register struct IDep * pDepI;
register integ         ExprI;
{
  if (ExprI != pDepI -> Value )
  {
     pDepI -> Value = ExprI;
     EnterEffect (& (pDepI -> Effect) );
  }
}
```

Aus Gründen der Effizienz ist es von besonderer Bedeutung, die Ausführungszeit der innersten Schleifen einer Schachtelung zu minimieren. Es ist die Aufgabe der Funktion `EnterEffect`, festzustellen

1. ob die aktuelle Wertänderung der abhängigen Variablen Auswirkungen auf andere abhängige Variablen hat (Flag: `ExecAlg`)
2. in welchen Modellkomponenten diese abhängigen Variablen zu finden sind (Array von Flags: `AlgEffect[i]`)

Das Flag `AlgEffect [i]` wird beim Aufruf der Basiskomponente `i` ausgewertet, während das Flag `ExecAlg` dazu herangezogen wird, die Iteration zu beenden.

Im Falle eines Modellierungsfehlers würde dieses Flag möglicherweise immer wieder neu gesetzt und die Iterationsschleife nie wieder verlassen. Um in diesem Fall den Simulationslauf mit einer ordentlichen Fehlermeldung zu verlassen, wird geprüft, ob die Zahl der Iterationen eine obere Schranke

`MaxCyc,`

die als Steuerparameter vorgegeben wird, überschreitet.

Nach Abschluß der Iteration wird in der Ausführungsstatistik gezählt, wie häufig eine bestimmte Anzahl von Iterationen notwendig war. Mit dieser Information soll sich der Benutzer ein Bild davon machen können, wofür die Rechenzeit aufgewendet wurde.

#### 6.5.3.2 Ausführung diskreter Zustandsübergänge

Nach der Fortschaltung der Simulationszeit T wird zunächst der erweiterte Modellzustand $\mathbf{W}(T)$ wieder hergestellt. Hierzu werden die statischen Beziehungen ausgewertet und die Differentialquotionen neu bestimmt. An dieser Stelle wird der Monitor aufgerufen, um den erreichten Zustand aufzuzeichnen.

Gehört der Zustand, wie er nun vorliegt, der Auslösemenge von Ereignissen an, werden diese Ereignisse ausgeführt und der diskrete Zustandsübergang vorbereitet. Löst der aktuelle Zustand kein einziges Ereignis aus, wird die Ereignisschleife beendet.

Ansonsten erfolgt die Erzeugung eines neuen Zustands. Der Taktzähler wird um eins erhöht und kennzeichnet damit diesen neuen Zustand. Der Zustandsübergang wird ausgeführt, indem den Zustandsvariablen ihre neuen Werte zugewiesen werden. Der Programmfluß kehrt nun an den Anfang der Schleife zurück, um von neuem einen erweiterten Modellzustand herzustellen.

Die Durchführung eines diskreten Zustandsübergangs erfolgt in zwei Schritten: In einem ersten Schritt (1.4) werden die neuen Werte der Zustandsvariablen zwischengespeichert und in einem zweiten Schritt (1.7) diese Werte zum aktuellen Zustand gemacht.

Das Durchlaufen der Basiskomponenten erfolgt ebenso wie bei den statischen Beziehungen. Die Steuervariable `Mode` ist dabei auf

```
Mode = Events
```

gesetzt.

Der Algorithmus, der die diskreten Zustandsübergänge zu einem Zeitpunkt ausführt, hat folgende Gestalt:

1. ———— Schleife mit Taktfortschaltung (Ereignisschleife) ————

```
for (;;)
{
```

1.1 Statische Beziehungen auswerten

1.2 Differentialquotienten bestimmen

1.3 Modellzustand aufzeichnen (Monitor aufrufen)

1.4 Ereignisbedingungen prüfen und Zustandsübergang vorbereiten

1.4.1 Globales Auswirkungsflag rücksetzen, Modus setzen

```
ExecEvt = No;  Mode = Events;
```

1.4.2 ———— Schleife über alle Basiskomponenten ————

```
for (i = 1;  i <= NBComp;  i ++)
{
    Falls Auswirkungen auf Basiskomponente i
    if (EvtEffect [i])
    {
        Lokales Auswirkungsflag und Bedingungsflag
        rücksetzen
        EvtEffect [i] = No;  EvtCond = No;
        Aufruf der Basiskomponente i und Ausführung der
        Ereignisse
        (* FBComp [i])  (DBComp [i]);
        Falls Ereignis ausgelöst, weiteres Bearbeiten des Er-
        eignisses veranlassen
        if (EvtCond)  ExecEvt = EvtEffect [i] = Yes;
    }
}
```

———— Ende der Komponentenschleife ————

1.5 Verlassen der Ereignisschleife, falls keine Auslösebedingung erfüllt war

```
if (! ExecEvt)  break;
```

1.6 Taktfortschaltung
Fehler anzeigen bei Überschreitung der max. Taktzahl

```
Takt = Takt + 1;
if (Takt > MaxTkt)  Error (2);
```

1.7 Zustandsübergang ausführen, Auswirkungen feststellen

```
struct IState * pStaVar = HeadISta;
while (pStaVar)
{
    pStaVar->Value = pStaVar->NxtValue;
    EnterEffect (&(pStaVar -> Effect ));
    pNext = pStaVar->NxtVar;  pStaVar->NxtVar = NULL;
    pStaVar = pNext;
}
HeadISta = TailISta = NULL;
```

——————— Ende der Ereignisschleife ———————

2. Eintrag in Ereignis-Statistik

Algorithmus 6.5.3-2: Zustandsübergänge zu einem festen Zeitpunkt

Es werden nur diejenigen Basiskomponenten durchlaufen, deren Auslösebedingungen bereits im letzten Zustand erfüllt waren oder Variablen enthalten, die ihren Wert verändert haben. Dies wird durch das Array von Flags `EvtEffect [i]` angezeigt.

Für ein Ereignis

```
WHENEVER  expression  DO transition_statement_sequence  END
```

wird innerhalb der Basiskomponente folgender Code erzeugt:

```
if (Mode == Events)
{
   ActLine = n;
   if ( <expression> )
   {
      <transition_statement_sequence>
   }
}
```

Für eine Wertzuweisung an eine Zustandsvariable

```
StaVar := expression;
```

wird der Funktionsaufruf

```
StaAsgnX (Adr( <StaVar> ), <expression> );
```

angelegt.

Je nach Wertemenge und Zeitverlauf der Variablen `StaVar` steht `StaAsgnX` für `StaAsgnI, StaAsgnD, StaAsgnC, StaAsgnL` bzw. `StaAsgnE`.

Die Funktion `StaAsgnX` bereitet einen Zustandsübergang vor, indem sie den neuen Zustand zwischenspeichert und die betreffende Zustandsvariable in eine Liste einkettet.

Der Aufbau der Funktion `StaAsgnX` sei beispielhaft für ganzzahlige Zustandsvariable angegeben:

```
void      StaAsgnI (pStaI, ExprI)
register  struct IState *  pStaI;
register  integ            ExprI;

{
   Neuen Zustand zwischenspeichern
   ===============================
   pStaI -> NxtVal = ExprI;

   Pruefen, ob Zustandsvariable bereits eingekettet ist
   ====================================================
   if (pStaI -> NxtVar  || TailISta == pStaI)  Error(12);

   Zustandsvariable einketten
   ==========================
   if (HeadISta)  TailISta -> NxtVar = pStaI;
           else   HeadISTa = pStaI;

   TailISta = pStaI;
}
```

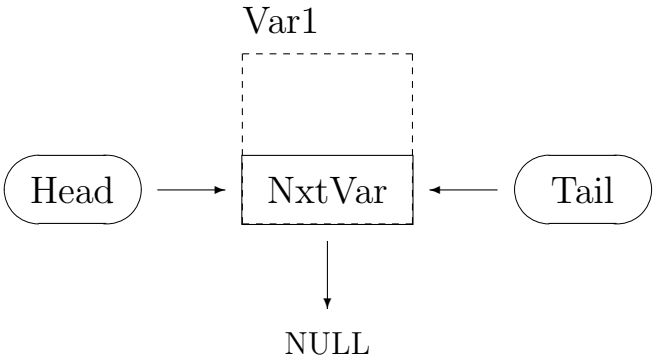
Die globalen Variablen

```
struct IState *  HeadISta, TailISta,
```

zeigen auf den Listenanfang und das Listenende.

**Leere Liste:**


**Liste mit einem Element:**




**Liste mit mehreren Elementen:**

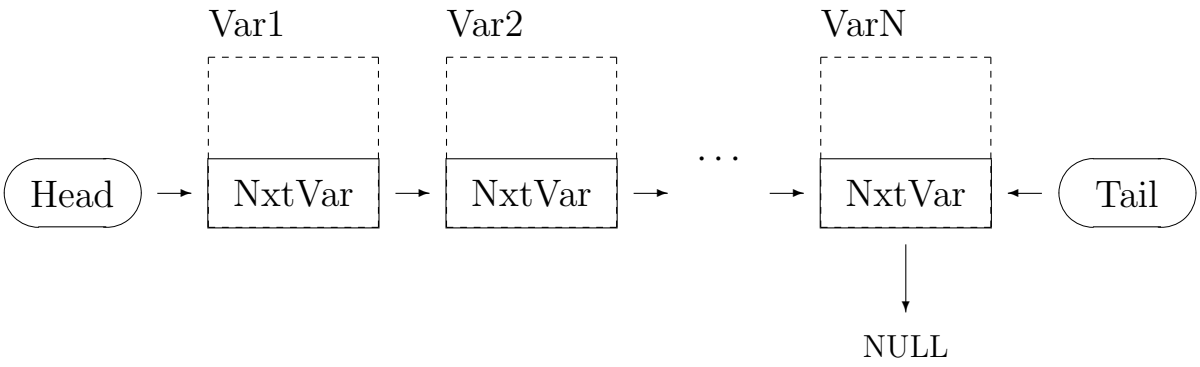



Abb. 6.5.3.2-1: Aufbau einer Liste von Zustandsvariablen

Durch die Verkettung der Zustandsvariablen läßt sich auch feststellen, ob einer Zustandsvariablen bereits ein Wert für den Folgezustand zugewiesen wurde. Ist dies der Fall und die Variable eingekettet, liegt ein Modellierungsfehler vor. Das Verhalten der Zustandsvariablen war nicht eindeutig definiert und der Simulationslauf wird daher mit einer entsprechenden Fehlermeldung abgebrochen.

In einem zweiten Schritt (1.7) wird der Zustandsvariablen der Wert zugewiesen, den sie im folgenden Takt erhalten soll. Zu diesem Zweck wird die aufgebaute Liste abgelaufen und für jede Zustandsvariable die Zuweisungen

```
pStaVar -> Value   =   pStaVar -> NxtValue;
pStaVar -> NxtVar  =   NULL;
```

ausgeführt.

Die Zeiger `HeadISta` und `TailISta` werden auf `NULL` zurückgesetzt. Damit ist die Liste abgebaut und der Zustandsübergang ausgeführt.

Die Veränderung einzelner Zustandsvariablen zieht Auswirkungen auf andere Modellgrößen und Ereignisbedingungen nach sich. Diese Auswirkungen werden durch Aufruf der Funktion `EnterEffect` festgestellt.

Es ist nicht ausreichend, eine Ereignisbedingung nur dann zu prüfen, wenn sich Variablen, die in der Bedingung enthalten sind, verändert haben. Ist eine Auslösebedingung einmal erfüllt, bleibt dies auch in den folgenden Takten so, falls die beteiligten Variablen ihre Werte behalten. Falls in einer Basiskomponente ein Ereignis ausgelöst wurde, wird das Flag `EvtCond` gesetzt, das ein erneutes Setzen der Auswirkungsflags bewirkt (1.4.2).

Im Falle eines Modellierungsfehlers kann es zu endlosen Ereignisfortpflanzungen kommen. Das Flag `ExecEvt` wird dann immer wieder aufs Neue gesetzt. Um in diesem Fall den Simulationslauf mit einer sinnvollen Fehlermeldung zu verlassen, wird geprüft, ob die Zahl der Ereignisfortpflanzungen eine obere Schranke

```
MaxTkt,
```

die als Steuerparameter vorgegeben wird, überschreitet.

#### 6.5.3.3 Die diskrete Zeitfortschaltung

Immer wenn zu einem bestimmten Zeitpunkt ein Modellzustand erreicht wurde, der sich nicht weiter verändert, sorgt die diskrete Zeitfortschaltung dafür, daß die Zeit solange weitergesetzt wird, bis wieder eine Aktivität im Modell stattfindet. Eine schematische Darstellung des hier beschriebenen Algorithmus findet sich am Ende dieses Abschnitts.

Beim Aufruf der Ablaufkontrolle sind alle Zustandsgrößen des Modells mit dem vorgegebenen Anfangszustand belegt und die Simulationszeit $T$ auf den Anfangszeitpunkt gesetzt.

Vor dem Eintritt in die Zeitschleife werden die Listenzeiger mit Null initialisiert und sämtliche Flags, die Auswirkungen anzeigen, gesetzt.

Innerhalb der Schleife werden mit jedem neuen Zeitpunkt die diskreten Zustandsübergänge ausgeführt. Es finden solange Ereignisse statt, bis keine der Auslösebedingungen mehr erfüllt ist.

An dieser Stelle kann der Simulationslauf beendet werden, wenn ein Endekriterium erfüllt ist. Auf diesem erreichten Endezustand kann der Simulationslauf jederzeit fortgesetzt werden.

Es sind die folgenden Endekriterien realisiert:

1. Der vorgegebene Endezeitpunkt `TEnd` ist erreicht.
2. Es wurde ein Interupt-Signal (`CTRL\`) über Tastatur abgegeben

Die Simulationszeit $T$ ist nun weiterzuschalten. In der Spezifikation (Kapitel 3) wurde festgelegt, daß $T$ um das Auflösungsvermögen *TScal* weiterzuschalten ist. Da die nächste Aktivität meist erst viel später stattfindet, wird versucht, diesen Zeitpunkt vorherzusagen, um Rechenzeit einzusparen.

Bei der Fortschaltung der Simulationszeit wird unterschieden, ob kontinuierliche Zustandsübergänge auszuführen sind oder nicht.

Enthält das Modell Differentialgleichungen oder algebraische Ausdrücke, die von der Zeit abhängen (ausgenommen Zeitvergleiche, siehe später), wird die Integrationssteuerung (*IntCtl*) aufgerufen, die solange (quasi-) kontinuierliche Zustandsübergänge ausführt, bis man einen vorgegebenen Zeitpunkt erreicht, zu dem eine diskrete Aktivität erwartet wird oder man an eine Unstetigkeitsstelle gelangt ist.

Bei einem rein diskreten Modell kann andernfalls die Simulationszeit unmittelbar auf den Zeitpunkt weitergesetzt werden, zu dem eine diskrete Aktivität möglich ist.

Ist ein Monitor aktiv, der in regelmäßigen Zeitabständen Aufzeichnungen vornimmt, wird die Zeit zunächst auf die entsprechenden Zwischenzeitpunkte gesetzt und der Monitor aufgerufen.

Die Fortschaltung der Simulationszeit hat möglicherweise Auswirkungen auf

- abhängige Variablen
- Differentialquotienten
- Ereignisbedingungen

Ob solche Auswirkungen vorhanden sind und welche Basiskomponenten dies betrifft wird durch Aufruf der Funktion `EnterEffect` festgestellt, bevor die Zeitschleife erneut durchlaufen wird.

Wie zu Beginn des Simulationslaufs liegt nun ein definierter Modellzustand vor, zu dem (wegen der veränderten Zeit $T$) der erweiterte Modellzustand und die Folgeereignisse zu ermitteln sind.

Der Algorithmus zur diskreten Fortschaltung der Simulationszeit besitzt folgende Gestalt:

1. Vorbelegungen
```
HeadISta = TailISta = NULL;
ExecAlg = ExecEvt = ExecDtl = Yes;
for (i=1; i <= NBComp; i++)
{ AlgEffect[i] = EvtEffect[i] = DtlEffect[i] = Yes; }
```

2. Zustandsübergänge vom Anfangszustand bis Endezustand ausführen

——— Schleife mit Zeitfortschaltung (Zeitschleife) ———

2.1 Protokollieren des Zeitpunkts

2.2 Diskrete Zustandsübergänge zum aktuellen Zeitpunkt ausführen

2.3 Verlassen der Zeitschleife bei Endebedingung

```
if (T = TEnd) return;
if (Intrpt)   return;
```

2.4 Zeitfortschaltung

2.4.1 Zeitpunkt für nächste diskrete Aktivität bestimmen (`TNext`)

2.4.2-A Falls kontinuierliche Zustandsübergänge erforderlich

```
if (AnyDtl || TimeDep)
{
   Aufruf der Integrationssteuerung:
   IntCtl (TNext);
}
```

2.4.2-B sonst:

```
else
{
```

B1 Zwischenschritte für Monitor ausführen

B2 Simulationszeit auf nächste diskrete Aktivität setzen

```
   T = TNext;  Takt = 0;
}
```

2.4.3 Auswirkungen der Zeitfortschaltung durch Zeitvergleiche feststellen

```
for (i=1;  i <= NTimeCmp;  i++)
{
    if (TimeCmp[i].Time == T)
    { EnterEffect (&(TimeCmp[i].Effect)); }
}
```

——— Ende der Zeitschleife ———

Algorithmus 6.5.3-3: diskrete Zeitfortschaltung

#### 6.5.3.4 Ermittlung des nächsten diskreten Zeitpunkts

Damit die Zeit nicht in der kleinsten Zeiteinheit *TScal* weitergeschaltet werden muß, ist es erforderlich, auf anderem Wege festzustellen, zu welchem Zeitpunkt eine diskrete Größe möglicherweise ihren Wert ändert.

In einem Modell, das ausschließlich diskrete Größen enthält, können diese sich zu einem zukünftigen Zeitpunkt nur ändern, wenn sie von der Simulationszeit $T$ abhängen.

Dies bedeutet, daß die Zeit $T$ entweder

- im algebraischen Ausdruck einer abhängigen Variablen oder
- in einer Ereignisbedingung
- in der Bedingung eines IF-Konstrukts

auftreten muß.

Da die Zeit eine quasi-kontinuierliche Größe darstellt, kommen nur solche algebraische Ausdrücke in Betracht, welche eine Konvertierung auf einen diskreten Zeitverlauf durchführen. Hierzu zählen

- Vergleichsoperationen
- stufenförmige Standardfunktionen wie TRUNC, RND, etc.

Es ist nun das Ziel vorherzusagen, zu welchem Zeitpunkt ein solcher diskreter algebraischer Ausdruck seinen Wert ändert. Zu diesem Zeitpunkt ändert sich dann möglicherweise auch eine diskrete, abhängige Variable oder eine Ereignisbedingung.

Diese Vorhersage ist nur in einfach gearteten Fällen wie z.B.

```
T >= TReady
```

möglich. Komplizierte Ausdrücke wie bereits

```
T - TReady >= 0
```

erfordern ein symbolisches, explizites Auflösen nach der Zeit $T$.

Für Ausdrücke wie z.B.

```
TRUNC (T + TService)
```

kann eine Vorhersage nur unter Einbeziehung der beteiligten Standardfunktion getroffen werden.

Obwohl für dieses Problem keine allgemein-gültige Lösung herbeigeführt werden kann, ist es aus Gründen der Effizienz unerläßlich, die Simulationszeit in möglichst großen Schritten fortzuschalten. Wir streben daher einen Kompromiß an, der

- in fast allen Fällen eine Vorhersage erlaubt
- die Möglichkeiten der Modellierung nicht einschränkt
- dem Benutzer verständlich ist

Hierzu sehen wir die folgende Regelung vor:
In Vergleichsausdrücken darf die Zeit nur alleine auf der rechten oder linken Seite auftreten. Auf der jeweils anderen Seite ist die Zeit nicht zulässig.

Unter der genannten Bedingung ist es möglich, vorherzusagen, wann ein Vergleichsausdruck seinen Wert wechselt. Es sind hiermit nur noch die folgenden Vergleichsausdrücke erlaubt.

```
T >= expression       expression <= T
T >  expression       expression <  T

T <= expression       expression >= T
T <  expression       expression >  T
```

Benötigt man einen komplexeren Vergleichsausdruck, ist der Term, welcher die Zeit $T$ enthält durch eine abhängige Variable zu ersetzen.

**Beispiel:**

`SIN(T) > 0.5` läßt sich ersetzen durch
`S > 0.5` mit `S = SIN(T);`

In aller Regel läßt sich ein Vergleichsausdruck, der die Zeit $T$ enthält, auf die geforderte Form bringen. Da eine andere Formulierung nur auf dem Umweg über eine zusätzliche Variable zu erreichen ist, wird der Benutzer diese nach Möglichkeit vermeiden.

Ist es aufgrund der vorliegenden Modellbeschreibung nicht möglich, den nächsten Zeitpunkt vorherzusagen, erfolgt die Zeitfortschaltung kontinuierlich (Aufruf von *IntCtl*) und der Zeitpunkt mit der nächsten Aktivität wird durch eine Region- bzw. Crossingsuche bestimmt. Dasselbe geschieht, falls der algebraische Ausdruck im Zeitvergleich kontinuierliche Größen enthält.

Für jeden im Modell vorkommenden Zeitvergleich

```
T comp_op   expression
```

wird ein Funktionsaufruf

```
TCompare ( <expression>, <comp_op>, TCompNo (i) )
```

und ein Feldelement im globalen Feld

```
struct TIMECMP  TimeCmp [];
```

mit der Datenstruktur

```
struct TIMECMP
{
   time             Time;
   struct EFFECT    Effect;
}
```

angelegt.

Der Funktion

```
logic     TCompare ( ExprVal, comp_op, index )
double    ExprVal;
short     comp_op;
long      index;
```

wird der Wert des Ausdrucks, die Art des Vergleichsoperators sowie ein Index für den Zugriff auf das Feld `TimeCmp` mitgegeben.

Für jeden Zeitvergleich im Modell wird ein eigener Index vergeben. Innerhalb einer Basiskomponente werden die Zeitvergleiche, beginnend bei 1, durchnumeriert. Zu dieser Nummer wird ein für diese Komponente spezifischer Basisindex

```
long     TCompBase
```

hinzuaddiert, der in der Aktivierungsphase belegt wird. Dies besorgt das Makro

```
TCompNo (i)       pData -> TCmpBase + i,
```

welches in der Datei `c_macros.h` definiert ist.

Die Funktion `TCompare` liefert den logischen Wert des Zeitvergleichs zurück. Darüber hinaus trägt sie den Zeitpunkt, zu dem die Bedingung ihren Wert wechselt, in der Datenstruktur `TIMECMP` ein. Hierzu wird der reelle Wert des Ausdrucks in einen ganzzahligen Zeitpunkt konvertiert.

Für die Operatoren `'>='` und `'<'` gilt:

```
Time = (time)  ceil (ExprVal / TScal);
```

und für die Operatoren `'>'` und `'<='` gilt:

```
Time = (time)  floor (ExprVal / TScal) + 1
```

Um nun denjenigen Zeitpunkt zu finden, zu dem möglicherweise die nächste diskrete Aktivität stattfindet, durchsucht die Ablaufsteuerung das Feld `TimeCmp` nach dem nächstgelegenen Zeitpunkt in der Zukunft (`TNext`).

```
TNext = TEnd;
for (i = 1;  i <= NTimeCmp;  i++)
{
   register  time  TActiv;
   TActiv =  TimeCmp[i]. Time;
   if  (TActiv > T  &&  TActiv < TNext) TNext = TActiv;
}
```

Die Datenstruktur `TIMECMP` enthält im Eintrag `Effect` die Auswirkungen, die ein Wechsel der Zeitbedingung hervorruft. Diese Information wird vom MDL-Compiler gewonnen und während der Aktivierungsphase eingetragen. Aus der Struktur `Effect` geht hervor, in welcher Basiskomponente der Zeitvergleich auftritt und auf welche Typen von Modellgrößen er Einfluß hat.

```
for (i = 1;  i  <=  NTimeCmp;  i++)
{
   if (TimeCmp[i]. Time == T)
   {
      EnterEffect  (& (TimeCmp[i].Effect));
   }
}
```

Die Funktion `EnterEffect` setzt die Flags, welche die Auswirkungen der Zeitfortschaltung anzeigen. Eine genaue Beschreibung, wie auf diese Art das Laufzeitverhalten verbessert wird, findet sich in einem späteren Abschnitt.

### 6.5.4 Die Ausführung kontinuierlicher Zustandsübergänge

Enthält ein Modell Differentialgleichungen oder algebraische Ausdrücke, die von der Zeit abhängen (außer Zeitvergleiche), sind parallel zur Zeitfortschaltung quasi- kontinuierliche Zustandsübergänge auszuführen.

Die Funktion

```
void  IntCtl  (TDiscrete)
time  TDiscrete;
```

führt solange quasi- kontinuierliche Zustandsübergänge aus, bis der vorgegebene Zeitpunkt einer diskreten Aktivität *TDiscrete* erreicht ist oder bis die Funktion *IntCtl* auf eine Unstetigkeitsstelle stößt, die ebenfalls ein Indiz für mögliche diskrete Aktivitäten liefert.

Hierzu löst die Funktion *IntCtl* den gesamten, zu überbrückenden Zeitraum in einzelne Integrationsschritte auf. Die Ausführung eines einzelnen Integrationsschritts, d.h. den quasi-kontinuierlichen Zustandsübergang eines Zustands $\mathbf{Z}(T_k)$ nach $\mathbf{Z}(T_k + l \cdot R)$ mit Schrittweite $l$ überträgt sie der Funktion *IntStep*.

```
void IntStep (TStart, TStep, ErrEval, Repeated, IntMeth,
              RErrMax, Offmax)

time     TStart;
time     TStep;
short    IntMeth;
logic    Repeated;
short    ErrEval;
double * RErrMax;
long   * Offmax;
```

*IntStep* veranlaßt die Durchführung eines Integrationsschritts von *TStart* nach *TStop* mit dem Integrationsverfahren *IntMeth*. Ist das Flag *Repeated* gesetzt, wird zunächst der Zustand zum Zeitpunkt *TStart* wiederhergestellt, indem die Werte aller Zustandsgrößen (Eintrag: *Value*) aus dem Eintrag *LstValue* übernommen werden.

Ist *ErrEval* `> 0`, wird eine Fehlerabschätzung durchgeführt. Der größte relative Fehler wird in der Variablen *RErrMax* zurückgeliefert, die Offset-Adresse derjenigen Zustandsvariablen, die diesen Fehler verursacht hat, in Offmax abgelegt.

Die Funktion *IntStep* bedient sich zur Durchführung der Integration der angebotenen Integrationsverfahren. Die zur Zeit implementierten Integrationsverfahren sind

| | | |
|---|---|---|
| 1) | `void Euler (TcStart, TcStep):` | Eulersches Polygonzugverfahren |
| 2) | `void RK4  (TcStart, TcStep):` | Runge-Kutta 4. Ordnung |
| 3) | `void RKF6 (TcStart, TcStep):` | Runge-Kutta-Fehlberg 5./6. Ordnung |
| 4) | `void RKIMP6 (TcStart, TcStep):` | implizites Runge-Kutta-Verfahren 6. Ordnung vom Gauß-Typ |
| 5) | `void Extpol6 (TcStart, TcStep):` | Extrapolationsverfahren nach Bulirsch-Stoer |
| 6) | `void RoWa (TcStart, TcStep):` | Rosenbrock-Wanner-Verfahren |

Im gegenwärtigen Entwicklungsstand sind nur Einschrittverfahren implementiert. An der Einbringung von Mehrschrittverfahren wird gearbeitet. Die Verfahren sind der Literatur über Numerische Mathematik [ConBo 72, StoBu 83, Grig 72, WerArn 86] entnommen und an die speziellen Datenbereiche des Simulators angepaßt worden [Gött 82, Weig 84].

Jedes Verfahren erhält als Übergabeparameter den Startzeitpunkt und die Schrittweite der Integration.

Die Ausführung kontinuierlicher Zustandsübergänge läuft demnach über die Abrufhierarchie

*IntCtl* → *IntStep* → *Integrationsverfahren*

ab.

#### 6.5.4.1 Die Ablaufsteuerung für kontinuierliche Zustandsübergänge (IntCtl)

Die Funktion

```
void  IntCtl (TDiscrete)
```

steuert die Ausführung kontinuierlicher Zustandsübergänge.

Der realisierte Algorithmus besitzt die folgende Gestalt:

1. Reguläre Integrationsschrittweite mit Anfangsschrittweite vorbesetzen
2. Ausführen von Integrationsschritten mit Zeitfortschaltung

   ——— Schleifenanfang (Integrationsschleife) ———

   2.1 Protokollieren des Zeitpunkts
   2.2 Crossingvermerke und Regionvermerke vorbesetzen
   2.3 Nächsten Monitor-Abtastzeitpunkt bestimmen
   2.4 Schrittweite festlegen
   2.5 Durchführung der Integration (Aufruf von IntStep)
   2.6 Reduzierung der Schrittweite auf stetigen Bereich, falls nötig (Regionsuche)
   2.7 Schrittweitenanpassung, falls gewünscht und relativer Fehler außerhalb der vorgegebenen Grenzen
   2.8 Reduzierung der Schrittweite, bis Unstetigkeit in Ereignisbedingung gerade erreicht ist (Crossingsuche)
   2.9 Einträge in Integrationsstatistik vornehmen
   2.10 Zeitfortschaltung ausführen
   2.11 Extrapolation über Unstetigkeit hinweg, falls keine Schrittweitenreduzierung und kein Crossing
   2.12 Monitoraufruf, falls Abtastzeitpunkt
   2.13 Abbruch der Schleife, falls Ereignis auszuführen ist

   ——— Ende der Integrationsschleife ———

3. Monitoraufruf: Sichern des erreichten Zustands

Algorithmus 6.5.4-1: Steuerung der Integration

Beim Betreten der Funktion *IntCtl* kann vorausgesetzt werden, daß der Modellzustand in seiner erweiterten Form (einschließlich abhängige Variable und Differentialquotienten) vorliegt. Nach Ausführung der Schleife (2) ist diese Voraussetzung ebenfalls erfüllt.

Beim Betreten der Funktion, d.h. nach der Ausführung eines diskreten Zustandsübergangs wird wieder die vom Benutzer im Steuerparameter

`StepSize`

festgelegte Anfangsschrittweite hergestellt.

Innerhalb der Schleife (2) werden solange Integrationsschritte ausgeführt, bis eine diskrete Aktivität möglich geworden ist. Dies ist der Fall, wenn

1. der vorgegebene Zeitpunkt *TDiscrete* erreicht ist
2. eine Unstetigkeit in einer Ereignisbedingung (Crossing) auftritt
3. ein Bereichswechsel stattgefunden hat
4. eine Unterbrechung des Simulationslaufs über Tastatur erfolgt ist

Vor der Rückkehr zur diskreten Ablaufsteuerung *FlowCtl* sichert sich der Monitor den erreichten Zustand (3), um eventuelle diskrete nderungen feststellen und protokollieren zu können.

Jeder Schleifendurchlauf der Integrationsschleife bringt die Integration um einen Schritt und damit um einen Zeitabschnitt voran.

Es wird regulär mit der Schrittweite aus dem letzten Integrationsschritt bzw. mit der vorgegebenen Anfangsschrittweite integriert. Der Integrationsschritt wird jedoch verkürzt, wenn der nächste diskrete Zeitpunkt *TDiscrete* oder der Zeitpunkt des nächsten Monitoraufruf *TMon* vor das reguläre Integrationsende fällt.

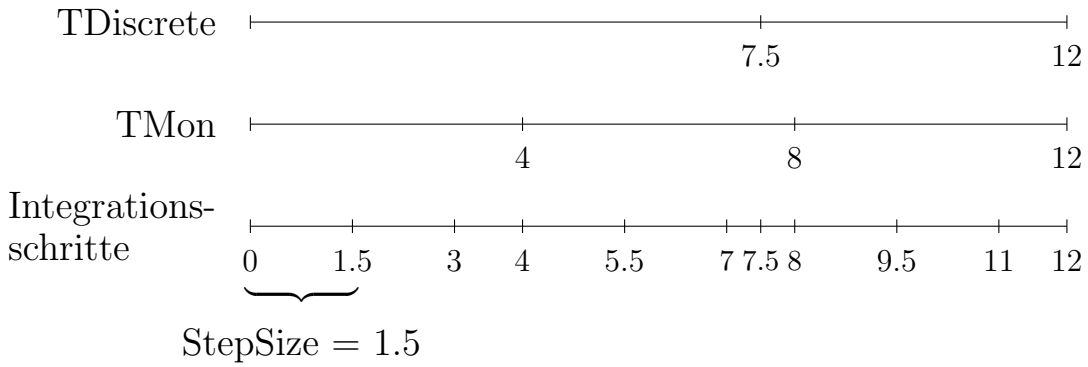



Abb. 6.5.4.1-1: Verkürzung der Integrationsschrittweite

Die Abbildung zeigt in der untersten Zeile die Lage der Integrationsschritte, wenn die reguläre Integration mit Schrittweite $StepSize = 1.5$ durch Monitoraufrufe und diskrete Aktivitäten unterbrochen wird.

Mit dieser (eventuell verkürzten) Schrittweite wird nun die Integration ausgeführt (2.5). Während des Integrationsschritts kann es zu einem Bereichswechsel kommen, der eine Verminderung der Schrittweite auf den stetigen Bereich erforderlich macht (2.6).

Für eine Integration im stetigen Bereich ist eine korrekte Fehlerbestimmung möglich, auf deren Grundlage eine Schrittweitenanpassung durchgeführt werden kann, falls dies erwünscht ist. Die Schrittweitenanpassung vermindert die Schrittweite bei zu großem relativen Fehler und merkt sich bei zu kleinem relativen Fehler eine vergrößerte Schrittweite für den nächsten Integrationsschritt vor.

Eine weitere Reduzierung der Schrittweite wird nötig, wenn in einer Ereignisbedingung, die kontinuierliche Variablen enthält, eine Unstetigkeit (Crossing) auftritt. Hierzu zählen insbesondere Vergleichsausdrücke, dereren Wert wechselt. In diesem Fall wird möglicherweise eine Ereignisbedingung wahr. Die Schrittweite ist bis zu dem Zeitpunkt zu vermindern, an dem das Crossing erstmalig erkannt wird.

Es kann demnach notwendig werden, die ursprünglich gewählte Schrittweite (2.4) wegen eines Bereichswechsel, einer Schrittweitenanpassung oder einer Crossingsuche zu vermindern. Der Integrationsschritt mit der zuletzt erreichten Schrittweite ist der für diesen Schleifendurchlauf gültige.

Der relative Fehler für diesen Schritt wird in die Integrationsstatistik aufgenommen. Ebenso wird gezählt, wie häufig und aufgrund welcher Vorkommnisse die Schrittweite vermindert werden mußte und wieviele Suchschritte zur Bereichssuche oder Crossingsuche benötigt wurden (2.9).

Die Unterprogramme zur Ausführung eines Integrationsschritts setzen nur die kontinuierliche Zeit $Tc$ weiter. Die diskrete Zeit 'T' bleibt stehen, bis die endgültige Schrittweite geklärt ist. Nun kann auch die diskrete Zeit nachgeführt werden (2.10).

Falls bis zu einem Bereichswechsel integriert wurde, erfolgt ein Extrapolationsschritt, um in den anderen Bereich überzuwechseln. Dies ist verbunden mit Aufrufen des Monitors, um eventuelle Sprung- oder Knickstellen aufzuzeichnen.

Vor dem Ende der Schleife liegt ein Monitoraufruf, der ausgeführt wird, wenn bis zu einem Abtastzeitpunkt integriert wurde.

Der Monitoraufruf unmittelbar vor dem Verlassen der Funktion sichert den erreichten Zustand, ohne ihn aufzuzeichnen. Diese Werte werden benötigt, falls sich nach der Rückkehr zur Ablaufkontrolle eine Variable sprunghaft verändert. In diesem Fall ist der Wert vor und nach der Sprungstelle aufzuzeichnen.

#### 6.5.4.2 Die Ausführung eines einzelnen Integrationsschritts (IntStep)

Die Funktion

```
void  IntStep (TStart, TStep, ErrEval, Repeated, IntMeth,
               RErrMax, Offmax)
```

führt einen einzelnen Integrationsschritt aus:

1. Protokoll abschalten

2-A Falls Wiederholung des Integrationsschritts:
→ Rücksetzen des Zustands

2-B Falls keine Wiederholung:
→ Aufbewahren des Zustands in *LstValue*

3. Rücksprung bei Schrittweite 0

4. Falls Fehlerabschätzung eingeschaltet und Integrationsverfahren ohne eingebaute Fehlerabschätzung:
→ Zweifache Integration mit halber Schrittweite

5. Integration mit vorgegebener Schrittweite

6. Protokoll einschalten und Integrationsschritt protokollieren

7-A Falls Fehlerabschätzung eingeschaltet:

7-A.1 Maximalen relativen Fehler bestimmen

7-A.2 Werte der kontinuierlichen Zustandsgrößen und geschätzte Fehler protokollieren

7-B Keine Fehlerabschätzung:
→ Werte der kontinuierlichen Zustandsgrößen protokollieren

8. Algebraische Gleichungen auswerten und protokollieren

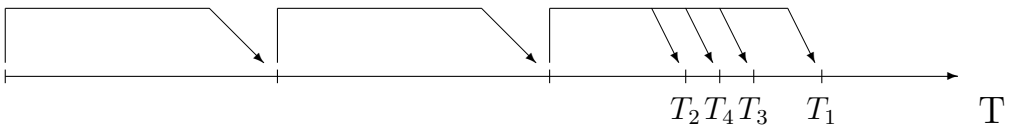
9. Differentialgleichungen bestimmen

10. Bereichswechsel prüfen

Algorithmus 6.5.4-2: Ausführung eines Integrationsschritts

Die Funktion *IntStep* setzt auf dem erweiterten Zustand zum Zeitpunkt *TStart* auf und liefert den erweiterten Zustand zum Zeitpunkt *TStart + TStep* zurück.

Die Funktion wird sowohl aufgerufen, um vom erreichten Zustand weiterzuintegrieren, als auch, um von einem Ankerpunkt aus die Integration zu wiederholen.




Bei einer Fortsetzung der Integration wird der aktuelle Zustand der Zustandsvariablen (Eintrag: *Value*) in den Eintrag *LstValue* kopiert. Im Falle einer Wiederholung wird der Zustand durch Kopieren in umgekehrte Richtung zurückgesetzt sowie die abhängigen Variablen und Differentialgleichungen ermittelt.

Integrationsverfahren mit eingebauter Fehlerbestimmung liefern den genaueren Wert der Integration im Eintrag *NxtValue* ab. Für die übrigen Verfahren wird der genauere Wert durch zwei aufeinanderfolgende Integrationsschritte mit halber Schrittweite bestimmt und im Eintrag *NxtValue* abgelegt.

Nach Ausführung der Integration mit vorgegebener Schrittweite wird der maximale relative Fehler und die Offset-Adresse der Variablen, die diesen Fehler verursacht, bestimmt.

Auf Wunsch werden die Werte der Zustandsvariablen und der ermittelte Integrationsfehler auf Protokolldatei ausgegeben.

#### 6.5.4.3 Die Integrationsverfahren

Beim Betreten des Verfahrens ist der erweiterte Zustand zum Zeitpunkt $Tc = TcStart$ vollständig bestimmt. Nach dem Verlassen liegen die Schätzungen der Zustandsgrößen vor und die Zeit $Tc$ steht am Ende des Integrationsschritts. Zum Zweck der Protokollierung erfolgen die übrigen Schritte zur Bestimmung des erweiterten Zustands und die Prüfung auf Bereichswechsel in der übergeordneten Funktion *IntStep*.

Die Verfahren zur Integration, welche die Numerische Mathematik zur Verfügung stellt, erfordern den Zugriff auf sämtliche kontinuierliche Zustandsvariablen des Modells.

Zu diesem Zweck wird ein Feld von Zeigern auf diese Größen

```
struct CState * pCState [],
```

angelegt, das während der Aktivierungsphase belegt wird.

Die Integrationsverfahren schätzen die Werte der Zustandsgrößen an bestimmten Stützstellen im Integrationsintervall. Im Anschluß an eine solche Schätzung ist das Integrationsverfahren zu ergänzen durch

- das Setzen der aktuellen kontinuierlichen Simulationszeit $Tc$
- das Feststellen der Auswirkungen dieser Veränderungen auf andere Variablen
- das Bestimmen der abhängigen Variablen
- das Ermitteln der Differentialquotienten
- die Prüfung auf einen Bereichswechsel

Der folgende Ablaufplan zeigt anhand des Runge-Kutta-Verfahrens 4. Ordnung, welche prinzipiellen Erweiterungen an einem Integrationsverfahren durchzuführen sind.

*Voraussetzungen:*

1. Zustand zum Zeitpunkt $Tc = TcStart$
2. algebraische Gleichungen bereits ausgewertet
3. Differentialquotienten ermittelt

1.1 Koeffizienten K1 bestimmen

```
for (i = 1;  i <= NCState;  i++)
{ K1[i] = pCState[i] -> Deriv * TcStep; }
```

1.2 Zwischenzustand $\mathbf{Z}_1(Tc1)$ ermitteln und Auswirkungen bestimmen

```
for (i = 1;  i <= NCState;  i++)
{
   pCState[i] -> Value = pCState[i] -> LstValue + K1[i]/2.;
   EnterEffect (&(pCState[i] -> Effect));
}
```

1.3 Zeitpunkt auf $Tc1$ setzen und Auswirkungen bestimmen

```
Tc = TcStart + TcStep/2.;
for (i = 1; i <= NBComp; i++)  EnterEffect (&TimeEffect[i]);
```

1.4 Algebraische Gleichungen auswerten

```
AlgSolve();
```

1.5 Differentialquotienten bestimmen und Flags zurücksetzen

```
Mode = DtlEqu;  ExecDtl = No;
for (i = 1;  i <= NBComp;  i++)
{
   if (DtlEffect[i]) { DtlEffect[i] = No;
                         (* FBComp[i]) (DBComp[i]);
                       }
}
```

1.6 Bereichswechsel prüfen

```
for (i = 1;  i <= NRegion;  i++)
{
  if (Region[i].ActRegion != Region[i].LstRegion) RegFlag = 1;
}
```

2.1 Koeffizienten K2 bestimmen
⋮

2.6 Bereichswechsel prüfen

3.1 Koeffizienten K3 bestimmen
⋮

4.1 Koeffizienten K4 bestimmen

4.2 Endezustand $\mathbf{Z}(Tc4 = TcStart + TcStep)$ ermitteln
und Auswirkungen bestimmen

```
for (i = 1;  i <= NCState;  i++)
{
   pCState[i] -> Value =
        pCState[i] -> LstValue
      + ( K1[i] + 2. * K2[i] + 2. * K3[i] + K4[i] ) / 6. ;

   EnterEffect (&(pCState[i] -> Effect));
}
```

4.3 Zeitpunkt auf $Tc4$ setzen und Auswirkungen bestimmen

Algorithmus 6.5.4-3: Integrationsschritt nach Runge-Kutta 4. Ordnung

Zum Bestimmen der Differentialquotienten werden alle betroffenen Modellkomponenten durchlaufen.

Für Differentialgleichungen im MDL-Quelltext

```
DIFFERENTIAL EQUATIONS
    Deriv' :=  expression;
END
```

wird folgender Code erzeugt.

```
if  (Mode == DtlEqu)
{
     StD (VAR) =  <expression> ;
}
```

Das Makro

```
# define  StD(VAR)     pData -> VAR. Deriv
```

stellt den Zugriff auf die Ableitung der Zustandsvariablen `VAR` her.

Das Auswerten der statischen Beziehungen ist in der Funktion

```
void  AlgSolve ()
```

zusammengefaßt und entspricht dem Verfahren, das bei diskreten Zustandsübergängen beschrieben wurde.

Die Maßnahmen zur Effizienzverbesserung wurden hier in gleicher Weise durchgeführt wie bei diskreten Zustandsübergängen.

Auswirkungen, die eine Veränderung der Zustandsgrößen mit sich bringen, werden im Integrationsverfahren durch den Aufruf der Funktion

```
EnterEffect (&(pCState[i] -> Effect));
```

vorgemerkt.

Ein kontinuierlicher Zustandsübergang geht mit der Fortschaltung der Zeit einher.

Im Datenbereich

```
struct EFFECT  TimeEffect [NBComp];
```

werden daher vom MDL-Compiler die Auswirkungen einer veränderten Zeit für jede Basiskomponente eingetragen. Solche Auswirkungen bestehen, wenn die Zeit explizit

1. in einer algebraischen Gleichung
2. in einer Differentialgleichung
3. in einer Ereignisbedingung
4. in einem IF-Konstrukt

enthalten ist.

Beim Bestimmen der abhängigen Variablen und der Differentialquotienten wird von den gesetzten Auswirkungsflags Gebrauch gemacht, indem nur die betroffenen Basiskomponenten aufgerufen werden.

#### 6.5.4.4 Suche des stetigen Bereichs

Da die Integrationsverfahren nur korrekt arbeiten, solange die Differentialquotienten stetige und differenzierbare Funktionen der Zeit sind, gilt es, mögliche Unstetigkeitsstellen zu erkennen.

Mit der vorliegenden Modellbeschreibungssprache gibt es drei Möglichkeiten, nicht stetige Differentialquotienten zu formulieren.

1. durch ein IF-Konstrukt

```
IF  (Z > 0)  OR  (T >= TX)  OR  L
    DO   DIFFERENTIAL EQUATION
         X' := expression;
         END
    END

ELSE DO  DIFFERENTIAL EQUATION
         X' := expression;
         END
    END
```

2. durch eine diskontinuierliche oder diskrete Funktion

```
DIFFERENTIAL EQUATIONS
   X' :=  MAX (Y,Z);           # Y und Z seien
   Y' :=  SIGN (Y) * Z;        # kontinuierliche Variablen
END
```

3. durch einen Operanden mit nicht stetig differenzierbarem Zeitverlauf

```
DIFFERENTIAL EQUATIONS
   X' := Y;                    # Y sei nicht stetig
END
```

Im ersten und zweiten Fall läßt sich die Unstetigkeit eines Differentialquotienten am Umspringen der Bedingung im IF-Konstrukt bzw. am Überqueren einer Unstetigkeit in der betreffenden Standardfunktion erkennen.

Für den dritten Fall ist es erforderlich, die Unstetigkeit von anderen numerischen Modellgrößen, d.h. Zustandsvariablen, abhängigen Variablen und Sensorvariablen zu überwachen.

Unstetigkeiten von Sensorvariablen treten gemeinsam mit der Unstetigkeit der zugeordneten Ausgangsvariablen, d.h. Zustandsvariablen oder abhängigen Variablen auf. Ist der unstetige Operand eine Zustandsvariable, wurde bereits bei dieser die Unstetigkeit erkannt.

Es ist daher nur die Überwachung von abhängigen Variablen mit numerischer Wertemenge erforderlich. Für diese gilt ebenso wie für Differentialquotienten, daß sie ihre Unstetigkeit durch

- ein IF-Konstrukt
- eine diskontinuierliche oder diskrete Funktion
- einen Operanden mit nicht stetig differenzierbarem Zeitverlauf

erhalten.

Wir können daraus folgen, daß alle möglichen Unstetigkeitsstellen gefunden werden, wenn wir

1. das Umspringen logischer Ausdrücke in IF-Konstrukten, die in ihren Zweigen Differentialgleichungen oder Zuordnungen an kontinuierliche, abhängige Variablen enthalten
2. das Umspringen diskreter und diskontinuierlicher Funktionen in Differentialquotienten und algebraischen Gleichungen

während eines Integrationsschritts überwachen.

Zur Erkennung eines Bereichswechsels wird für jeden logischen Ausdruck in einem IF-Konstrukt und für jede diskrete oder diskontinuierliche Funktion in Differentialquotienten und algebraischen Gleichungen eine Datenstruktur

```
struct REGION
{
   long  ActRegion;
   long  LstRegion;
}
```

angelegt. Alle diese Datenstrukturen des Modells liegen in einem Feld

```
struct  REGION   Region[];
```

vor, das in der Aktivierungsphase mit der Dimension aus `dimensions.h` angelegt wird. Die Elemente dieses Feldes werden innerhalb einer Basiskomponente fortlaufend durchnumeriert und über das Makro

```
# define   RegNo(i)   pData -> RegBase + i
```

adressiert (vgl. Zeitvergleiche).

Während des Modellablaufs wird im Eintrag *ActRegion* festgehalten, in welchem Bereich gerade gearbeitet wird. Vor der Ausführung eines Integrationsschritts wird der Eintrag *ActRegion* nach *LstRegion* kopiert.

Während und nach einem Integrationsschritt wird geprüft, ob sich die beiden Einträge unterscheiden. Ist dies der Fall, liegt ein möglicher Bereichswechsel vor.

Für die Bedingung in einem IF-Konstrukt

```
IF  expression
```

wird folgender Code erzeugt.

```
if ( Actline = n  &&  SetRegion(i) = <expression> )
```

Das Makro `SetRegion(i)` löst dabei den Zugriffspfad auf den Eintrag `ActRegion` auf:

```
# define   SetRegion(i)   Region [ RegNo(i) ]. ActRegion
```

In diskontinuierlichen und diskreten Standardfunktionen sind zwei oder mehrere Bereiche zu unterscheiden. Als Beispiel für eine Unterteilung in

zwei Bereiche durch einen logischen Vergleich sei die Funktion MAX angeführt.

```
double MAX  (x,y, RegIndex)

register double  x,y;
register long    RegIndex;
{
    register int Comp = (x >= y);
    Region [RegIndex].ActRegion = Comp;
    return ( Comp \Fragezeichen  x : y );
}
```

Der Index des Feldelements, welches der Standardfunktion zugeordnet ist, wird im Parameter *RegIndex* übergeben.

Als Beispiel für eine Unterteilung in mehrere Bereiche sei die diskrete Funktion `TRUNC(x)` angeführt, welche die Nachkommastellen abschneidet.

```
double  TRUNC (x, RegIndex)

register  double x;
register  long   RegIndex;
{
   register double Value;
   Value = ((x >= 0.0) \Fragezeichen  floor (x) : ceil (x));
   Region [RegIndex].ActRegion = (long) Value;
   return (Value);
}
```

Der Aufruf der beiden Funktionen erfolgt durch

```
MAX   ( <expr1>, <expr2>, RegNo (i) )
TRUNC ( <expr1>, RegNo (j) ).
```

Der MDL-Compiler ergänzt die vom Benutzer angegebene Parameterliste um den Index des Region-Elements.

Tritt eine solche Funktion an einer Stelle auf, an der sie keine unstetige Ableitung verursachen kann, wird der Index `0L` eingesetzt, beispielsweise

```
TRUNC ( <expr1>, 0L ) .
```

Das 0-te Feldelement wird nicht benutzt und es ist daher bedeutungslos, wenn es laufend überschrieben wird.

Durch Vergleich der beiden Struktur-Einträge *ActRegion* und *LstRegion* kann entschieden werden, ob ein Bereichswechsel stattfand. Das Ergebnis zeigt die globale Variable

```
logic RegFlag;
```

an.

Dieses Flag steuert die Suche nach dem größtmöglichen stetigen Bereich, die durch sukzessive Intervallhalbierung ausgeführt wird. Es wird derjenige Zeitpunkt gesucht, zu dem *RegFlag* gerade noch *false* ist.

Zum Überwinden der Unstetigkeitsstelle wird daraufhin ein Integrationsschritt mit einer Schrittweite ausgeführt, die dem Auflösungsvermögen entspricht. Anschließend wird zur Ablaufkontrolle *FlowCtl* zurückgekehrt, um zu prüfen, ob aufgrund der Unstetigkeit Ereignisse eintreten.

#### 6.5.4.5 Überprüfung kontinuierlicher Ereignisbedingungen

Wird während eines Integrationsschritts eine Bedingung wahr, die ein Ereignis auslöst, dann ist der Integrationsschritt bis zu dem Zeitpunkt zu vermindern, zu dem die Bedingung erstmalig erfüllt ist.

Wir wollen auch hier die Möglichkeiten diskutieren, die zur Auslösung eines Ereignisses während eines Integrationsschritts führen:

1. Ereignisbedingung enthält kontinuierliche Variablen einschließlich der Zeit $T$

```
WHENEVER   (X > 0)  OR  (Y > 0)     DO  ...  END
WHENEVER   (T >= 10)  AND  (A = 0)  DO  ...  END
WHENEVER   MAX (X, Y) >= 0          DO  ...  END
WHENEVER   ITRUNC (X) >= 0          DO  ...  END
```

2. Ereignisbedingung enthält abhängige Größen, die auf kontinuierlichen Variablen beruhen

```
WHENEVER  A > 0          mit   A := X - Y;
DO ...                         A := MAX (X, Y);
                               A := ITRUNC (X);

WHENEVER  L              mit   L := (X > 0);
DO ...                         L := (MAX (X,Y) > 0);
                               L := (ITRUNC (X) > 0;
```

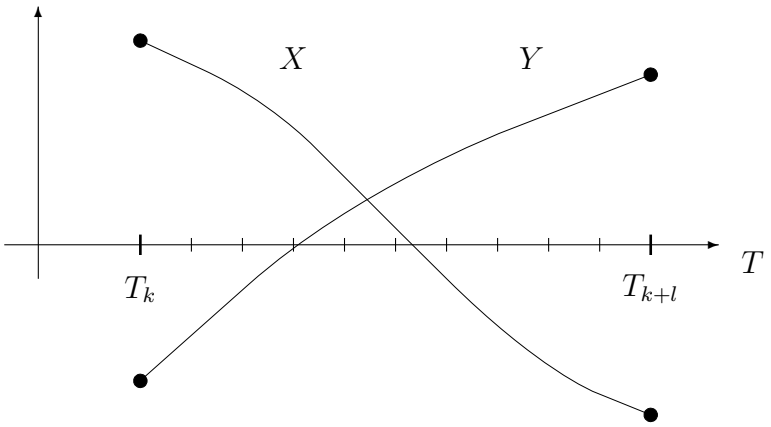
3. eine übergeordnete IF-Bedingung beruht auf einer stetigen Größe

```
IF  X > 0
DO
    WHENEVER   L
    DO  ...
```

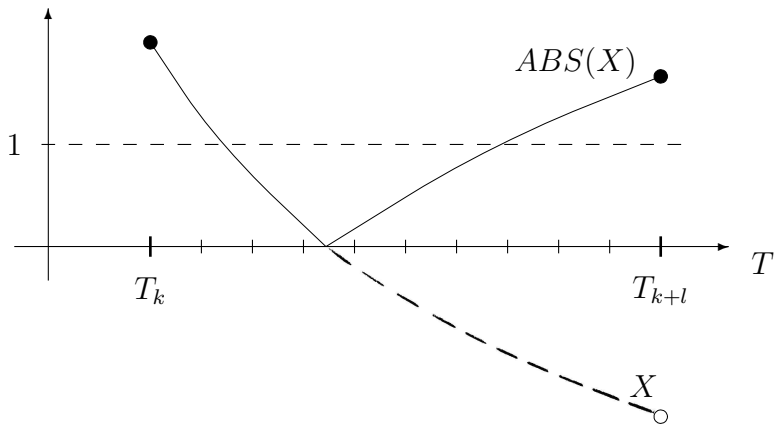
Es genügt, nach Ausführung eines Integrationsschritts, alle Ereignisbedingungen daraufhin zu prüfen, ob eine davon wahr geworden ist und daraufhin mit dem Halbierungsverfahren den genauen Zeitpunkt des Eintretens zu bestimmen.

Mit dieser Vorgehensweise läuft man jedoch Gefahr, das zwischenzeitliche Eintreten einer Bedingung nicht zu erkennen, wie die beiden folgenden Beispiele zeigen.

1. `WHENEVER  (X > 0)  AND  (Y > 0)`



```
2.  WHENEVER    ABS (X) < 1
```




Die Integration wird von $T_k$ nach $T_{k+l}$ ausgeführt. Weder am Anfang noch am Ende des Integrationsintervalls ist die Ereignisbedingung erfüllt, wohl aber zu dazwischenliegenden Zeitpunkten. Da die Spezifikation vorsieht, daß alle auf dem Zeitraster liegenden Zeitpunkte (festgelegt durch das Auflösungsvermögen *TScal*) geprüft werden, ist es mit einer Auswertung der Ereignisbedingungen am Ende eines Integrationsschritts nicht getan.

Der Wechsel einer Ereignisbedingung kann nur von kontinuierlichen Variablen ausgehen. Wir betrachten daher diejenigen Ausdrücke, in denen das Ergebnis einer kontinuierlichen Operation eine diskrete Operation ist. Der Zeitpunkt zu dem ein numerisches Ergebnis resultiert, wird bereits durch die Bereichssuche gefunden. Es sind daher nur Operationen mit logischem Ergebnis, d.h. Vergleichsoperationen von Interesse.

Wir können daher folgern: Wenn ein Vergleichsausdruck (aus einer Ereignisbedingung oder dem Ausdruck einer abhängigen, logischen Variablen), der wenigstens einen kontinuierlichen Operanden enthält, seinen Wert ändert, besteht die Möglichkeit, daß eine Ereignisbedingung wahr wird.

Diese Vergleichsausdrücke werden in der Literatur, z.B. [Schm 84], auch als *Crossings* bezeichnet.

Das Auffinden des Zeitpunkts, zu dem der Wert eines Vergleichsausdrucks wechselt, ist gleichbedeutend mit dem Problem der numerischen Nullstellensuche.

Jeder Vergleichsausdruck

```
A(T) > B(T)
```

läßt sich nach

```
F(T) = A(T) - B(T) > 0
```

umformen.

Für eine effiziente Lösung stehen die Verfahren zur Verfügung, welche die Numerische Mathematik anbietet [ConBo 72].

Als geeignetste Methode hat sich die *modifizierte regula falsi* erwiesen, wie sie bereits im Kapitel 3.6 beschrieben wurde. Dieses Verfahren grenzt die Nullstelle von zwei Seiten her ein und eignet sich daher auch für nicht stetige Funktionen. Andererseits ist die Zahl der Suchschritte sehr gering.

Für jeden kontinuierlichen Vergleichsausdruck wird eine Datenstruktur

```
struct  CROSS
{
   logic  LstCond;
   double LstDiff;
}
```

angelegt. Alle Datenstrukturen des Modells liegen in einem Feld

```
struct  CROSS   Cross [].
```

vor, das in der Aktivierungsphase mit der Dimension aus `dimensions.h` angelegt wird. Die Elemente dieses Feldes werden innerhalb einer Basiskomponente fortlaufend durchnumeriert und über das Makro

```
CroNo (i)   pData -> CroBase + i
```

adressiert (vgl. Zeitvergleiche).

Da während der Integration diejenigen Vergleichsausdrücke, die direkt in einer Ereignisbedingung stehen, nicht erreichbar sind, werden sie in einem eigenen Programmabschnitt des C-Codes der Komponente untergebracht.

```
if (Mode == Detect)
{
    .
    .
    .
}
```

In diesem Programmabschnitt steht für jeden kontinuierlichen Vergleich (aus einer Ereignisbedingung oder dem Ausdruck einer abhängigen, logischen Variablen)

```
expr1  comp_op  expr2     mit  comp_op = { >, >=, <, <= }
```

der Funktionsaufruf

```
CroCheck ( <expr1>, <k_comp_op>, <expr2>, CroNo(i) )
```

Die Funktion

```
logic  CroCheck  (LExpr, CompOp, RExpr, CroIndex)
real   LExpr, RExpr;
int    CompOp;
long   CroIndex
```

kennt zwei Betriebsmodi.

Bei Aufruf vor Ausführung eines Integrationsschritts belegt sie die Einträge der Datenstruktur `CROSS`. In *LstCond* wird das Ergebnis des Vergleichs und in *LstDiff* die Differenz zwischen rechter und linker Seite eingetragen.

Bei Aufruf nach Ausführung eines Integrationsschritts prüft sie, ob die Bedingung gewechselt hat. Ist dies der Fall wird in die globale Variable

```
long   CroFlag
```

der Index des betreffenden Vergleichsausdrucks eingetragen.

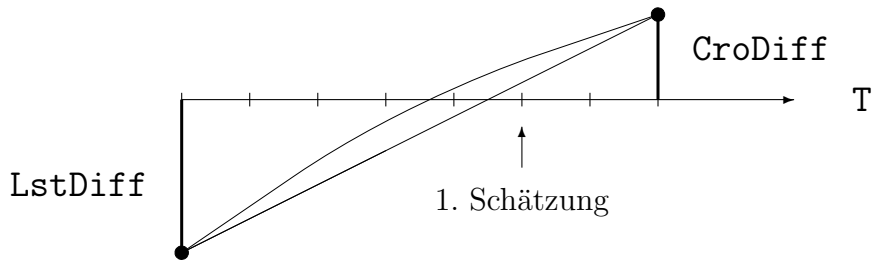
In die globalen Variablen

```
logic  CroCond;
real   CroDiff;
```

wird der aktuelle Wert und die aktuelle Differenz des Vergleichsausdrucks eingetragen. Dies geschieht jedoch nur, falls es sich um das gesuchte Crossing handelt. Dies erkennt man daran, daß die Bedingung `CroFlag = CroIndex` erfüllt ist.

Treten während eines Integrationsschritts mehrere Bedingungswechsel auf, dann zeigt die globale Variable *CroFlag* den zuletzt gefundenen Wechsel an. *CroFlag* liefert so den Index des Feldelements zu einen bestimmten Vergleichsausdruck. Über diesen Index kann man sich nun die Differenz *LstDiff* vor Ausführung der Integration beschaffen.

Die beiden Differenzen zur Nullinie *LstDiff* und *CroDiff* bilden die Grundlage für eine Schätzung der Nullstelle.




Bis zum neuen Schätzwert der Nullstelle wird der nächste Integrationsschritt ausgeführt, ohne den Wert des *CroFlag* zurückzusetzen. Dadurch erhält man für diesen Zeitpunkt den Abstand von der Nullinie *CroDiff*.

Es werden solange neue Schätzwerte bestimmt, bis sichergestellt ist, daß links vom gesuchten Zeitpunkt die Bedingung noch nicht erfüllt ist und rechts davon die Bedingung bereits wahr ist.

Danach wird in die Ablaufkontrolle zur Ausführung diskreter Zustandsübergänge zurückgekehrt und das betreffende Ereignis ausgeführt.

### 6.5.5 Verbesserung des Laufzeitverhaltens

Der besprochene Algorithmus zur Auswertung der Modellbeschreibung führt in den Basiskomponenten des Modells Aktionen unterschiedlichen Typs

- zur Bestimmung der abhängigen Größen
- zur Bestimmung der Differentialquotienten
- zur Auslösung von Ereignissen und zum Vorbereiten der diskreten Zustandsübergänge

aus.

Es ist das Ziel eines effizienten Laufzeitverhaltens, die Anzahl der erforderlichen Aktionen minimal zu halten.

Hierzu ist der Frage nachzugehen, unter welchen Bedingungen eine bestimmte Aktion auszuführen ist. Es gilt:

Eine abhängige Größe und ein Differentialquotient sind dann neu zu bestimmen, wenn sich ein oder mehrere Operanden, von denen diese Größen abhängen, verändert haben.

Ein Ereignis ist dann auszuführen, wenn

1. im letzten Takt die Ereignisbedingung nicht erfüllt war und sich ein oder mehrere Operanden der Ereignisbedingung verändert haben.
2. im letzten Takt die Ereignisbedingung erfüllt war (denn: bei unveränderten Operanden bleibt die Bedingung wahr)

Kernstück der Effizienzverbesserung ist daher die Verfolgung der Auswirkungen, welche die Veränderung einer Modellgröße mit sich bringt.

Hat eine Modellgröße oder die Simulationszeit ihren Wert verändert, so hat dies Einfluß auf

- eine Menge von algebraischen Gleichungen
- eine Menge von Differentialgleichungen
- eine Menge von Auslösebedingungen,

d.h. auf eine Menge von Aktionen.

Die aufgrund einer Wertänderung notwendig gewordenen Aktionen lassen sich nun

1. auf eine Menge von betroffenen Komponenten
2. auf bestimmte Aktionstypen (z.B. algebraische Gleichungen) in diesen Komponenten
3. auf bestimmte Aktionen innerhalb der Komponente

einschränken.

Für jede Zustandsvariable, für jede abhängige Variable und für jeden Zeitvergleich ist daher festzustellen,

1. auf welche Komponenten liegt ein Einfluß vor
2. welche Aktionstypen in diesen Komponenten sind betroffen
3. welche speziellen Aktionen in diesen Komponenten sind auszuführen

Da der dritte Punkt einen erheblichen Aufwand an Speicherplatz und Code hervorrufen würde, wurden nur die ersten beiden Punkte realisiert.

Die Datenstruktur

```
struct  EFFECT
{
   unsigned short    CompNo;     Komponentennummer
   unsigned short   EffFlag;     betroffene Aktionstypen
   struct EFFECT * eff_next;     Zeiger auf weitere Auswirkungen
}
```

wird jeder Zustandsvariablen, abhängigen Variablen und Sensorvariablen sowie jedem Zeitvergleich beigeordnet und zeigt an, in welcher Modellkomponente die Variable bzw. der Zeitvergleich vertreten ist und auf welche Aktionstypen eine Wertänderung dieser Variablen Einfluß hätte.

Hierzu werden die letzten drei Bit von *EffFlag* herangezogen.

*EffFlag*

Die gesetzten Bits A, E und D zeigen Auswirkungen auf algebraische Gleichungen, Ereignisbedingungen bzw. Differentialgleichungen an.

Ist die Modellgröße über eine Component Connection an eine oder mehrere Sensorvariablen angeschlossen, wird während der Aktivierungsphase eine Liste aufgebaut, welche die EFFECT-Strukturen miteinander verbindet (siehe *Die Aktivierungsphase*).

ndert sich während des Modellablaufs der Wert einer Modellgröße `varname`, dann werden durch Aufruf der Funktion

```
EnterEffect (Adr (varname.Effect))
```

die Auswirkungen dieser Wertänderung festgehalten.

Die Funktion

```
void EnterEffect (pVarEffect)
register struct EFFECT * pVarEffect;
```

verfolgt die Auswirkungsliste und trägt die Auswirkungen in globale Variable ein.

Die global definierten Felder

```
logic   AlgEffect  [NBComp];
logic   EvtEffect  [NBComp];
logic   DtlEffect  [NBComp];
```

mit der Anzahl der Basiskomponenten als Dimension erhalten im i-ten Element den Eintrag '1', falls Auswirkungen auf den entsprechenden Aktionstyp in der i-ten Komponente vorliegen.

Die global definierten Flags

```
logic   ExecAlg;
logic   ExecEvt;
logic   ExecDtl;
```

zeigen an, ob überhaupt Auswirkungen auf einen entsprechenden Aktionstyp vorliegen.

Diese Flags liefern die Entscheidung, ob die entsprechenden Programmabschnitte überhaupt zu durchlaufen sind und wann die Schleifen beendet werden dürfen. Auf die Verwendung dieser Informationen wurde in den Abschnitten davor bereits eingegangen.

Die beschriebenen Maßnahmen für ein effizientes Laufzeitverhalten kommen nur zum Tragen, wenn das Modell auf mehrere, nicht zu umfangreiche Modellkomponenten verteilt wurde.

Da dies auch Voraussetzung für ein modular aufgebautes, leicht verständliches und änderungsfreundliches Modell ist, kann man davon ausgehen, daß diese Voraussetzung in aller Regel erfüllt ist.

### Abschätzung der Komplexität des Algrorithmus

Die folgende Abschätzung soll zeigen, in welcher Weise die aufgeführten Verbesserungen die Komplexität des Algorithmus beeinflussen. Um die Betrachtungen einfach zu halten, gehen wir von einem Modell aus, das keine anderen Aktionen als Ereignisse enthält.

Aktivitäten finden immer zu solchen Zeitpunkten statt, an denen ein zeitbedingtes Ereignis ausgelöst wird. Dieses kann weitere zustandsbedingte Ereignisse anstoßen.

Für unsere Abschätzung gehen wir weiterhin von folgenden Annahmen aus:

1. Die Anzahl zeitbedingter Ereignisse im Modell sei $N_{TEvent}$
2. Die Gesamtzahl an Ereignissen betrage $\alpha \cdot N_{TEvent}, \quad \alpha > 1$
3. Der mittlere zeitliche Abstand des gleichen zeitabhängigen Ereignisses sei $\tau$
4. Zeitbedingte Ereignisse erfolgen nicht oder nur selten zum gleichen Zeitpunkt

5. Die mittlere Anzahl an Takten, die auf ein zeitbedingtes Ereignis folgen, sei $N_J$
6. Die mittlere Ausführungszeit für ein Ereignis sei $T_{Event}$
7. Die Simulation laufe über einen Zeitraum $\Delta T$ ab.

Hieraus ergibt sich der mittlere zeitliche Abstand zweier beliebiger Ereignisse als $\frac{\tau}{N_{TEvent}}$.
Die Anzahl der Ereigniszeitpunkte über den Simulationszeitraum $\Delta T$ liegt damit bei $N_{TEvent} \cdot \frac{\Delta T}{\tau}$ und die Anzahl der Takte $N_{Takt}$ über den Simulationszeitraum bei

$$N_{Takt} = N_J \cdot N_{TEvent} \cdot \frac{\Delta T}{\tau}$$

Wir überlegen nun, welche Ausführungszeit benötigt wird, wenn wir das Modell einerseits in einer einzigen Komponente beschreiben, andererseits auf $N_B = N_{TEvent}$ Komponenten aufteilen, d.h. jedes zeitbedingte Ereignis in einer eigenen Komponente unterbringen.

a) Modell in einer Komponente

Mit der Ausführungszeit für einen Aufruf der einzigen Komponente des Modells

$T_{Model} = \alpha N_{TEvent} \cdot T_{Event}$

ergibt sich als Ausführungszeit für den Simulationszeitraum $\Delta t$:

$$T_{Exec}^{(a)} = N_{Takt} \cdot T_{Model} = \underbrace{\alpha N_{TEvent}^2}_{1} \cdot \underbrace{N_J}_{2} \cdot \underbrace{\frac{\Delta T}{\tau}}_{3} \cdot \underbrace{T_{Event}}_{4}$$

Die Ausführungszeit setzt sich aus 4 Faktoren zusammen:

1. *Modellgröße:*
   Es fällt auf, daß die Zahl zeitbedingter Ereignisse quadratisch eingeht
2. *Komplexität:*
   Eine hohe Anzahl von Folgeereignissen bedeutet eine starke Vernetzung des Modells
3. *zeitliche Ereignisdichte*
4. *Ausführungszeit für ein Ereignis*

b) Zerlegung des Modells in $N_{TEvent}$ Komponenten

Von einem zeitbedingten Ereignis seien im Mittel $k$ Modellkomponenten betroffen ($k \geq 1$, normalerweise $k \leq 2$).

Mit der Ausführungszeit für einen Komponentenaufruf

$$T_{Comp} = \frac{1}{N_{TEvent}} \cdot T_{Model} = \alpha T_{Event}$$

ergibt sich als Ausführungszeit für den gesamten Simulationszeitraum

$$\begin{aligned} T_{Exec}^{(b)} &= N_{Takt} \cdot k \cdot T_{Comp} = \\ &= k \cdot \underbrace{\alpha N_{TEvent}}_{1} \cdot \underbrace{N_J}_{2} \cdot \underbrace{\frac{\Delta T}{\tau}}_{3} \cdot \underbrace{T_{Event}}_{4} \end{aligned}$$

Die Anzahl zeitbedingter Ereignisse und damit die Modellgröße geht in diesem Fall nunmehr linear ein. Dies ist auf die selektive Ausführung der Modellkomponenten zurückzuführen, welches die wesentliche Maßnahme zur Effizienzsteigerung darstellt.

### 6.5.6 Die Beobachter

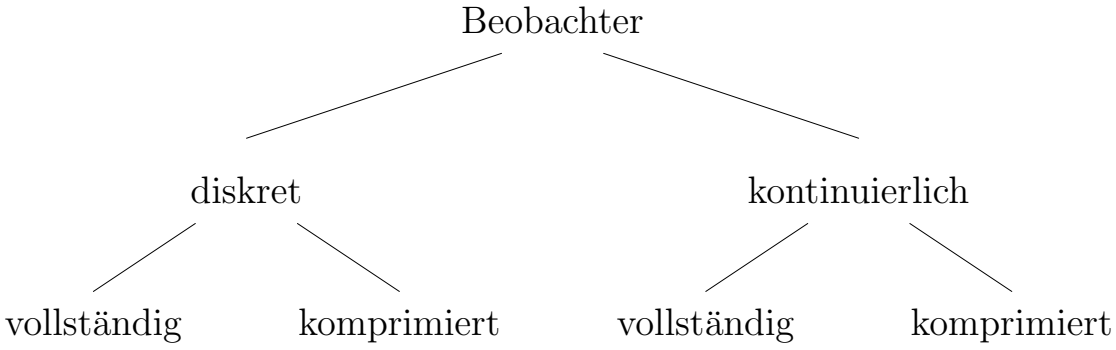
#### Anforderungen an die Beobachter

Ein Beobachter zeichnet Informationen über die Zwischenzustände eines Simulationslaufs auf. Dadurch erhält der Benutzer Kenntnis über den zeitlichen Verlauf der Modellgrößen. Diese Informationen werden auf Datei ausgegeben und können im Anschluß an den Simulationslauf ausgewertet und dargestellt werden.

Die Beobachtung von Modellgrößen ist Bestandteil eines Experiments. Es ist daher wichtig, daß ein Beobachter während des Experimentierens angelegt werden kann und nicht in die Modellbeschreibung aufgenommen werden muß. Nur mit diesem Konzept ist die Ausführung von Experimenten möglich, ohne ständig das Modell modifizieren und neu übersetzen zu müssen.

Jeder spezifizierte Beobachter zeichnet den Zeitverlauf für eine gegebene Menge von Variablen auf. Bezüglich des Typs der Variablen gibt es keine Einschränkungen. Es ist möglich Zustandsvariable ebenso wie abhängige Größen und Sensorgrößen, aber auch Ein- und Ausgangsgrößen höherer Komponenten aufzuzeichnen.

Wir unterscheiden Beobachter für diskrete und kontinuierliche Größen sowie mit vollständiger und komprimierter Aufzeichnung.




Während diskrete Größen bei jeder Wertänderung aufgezeichnet werden, erfolgt die Aufzeichnung kontinuierlicher Größen in einem festen zeitlichen Abstand zuzüglich eventueller Unstetigkeiten.

Eine komprimierte Zeitreihe beschränkt sich in der Aufzeichnung auf Minimum, Maximum und zeitlichen Mittelwert von Intervallen vorgegebener Breite.

Mit dieser Art der Aufzeichnung läßt sich der Wertebereich, in dem sich eine Modellgröße bewegt, auch über große Zeiträume hinweg verfolgen, ohne übergroße Dateien zu produzieren.

Für jede Variable, die ein Beobachter aufzeichnet, wird eine Wertetabelle angelegt. Diese Wertetabelle enthält die folgenden Einträge:

| **Zeit** | **Takt** | **Wert** |
|---|---|---|
| *double* | *short* | *union TYPES* |

Da der Wert je nach Variable verschiedenen Wertemengen angehören kann, wird er zuvor durch Kopie in eine Variable vom Typ *union TYPES* auf ein einheitliches internes Darstellungsformat gebracht.

```
union TYPES  { integ I;  real R;  logic L;  enumera E; }
```

### Spezifikation der Beobachter

Wir beschreiben nun, wie die verschiedenen Typen von Beobachtern im Experimentiersystem spezifiziert werden und welche Information aufgezeichnet wird.

### 1.) Zeitreihen für diskrete Variablen

Erforderliche Angaben:

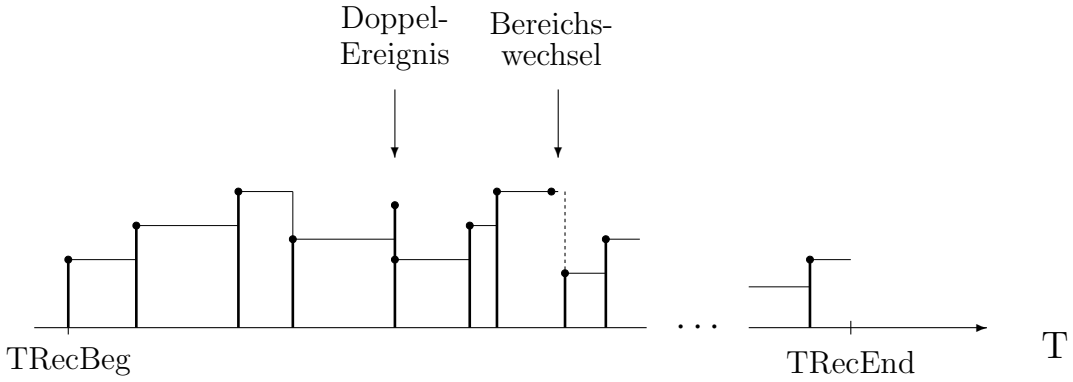
- Beginn der Aufzeichnung *TRecBeg*
- Ende der Aufzeichnung *TRecEnd*

zusätzlich für komprimierte Zeitreihe:

- Intervallbreite *Width*

Aufzeichnung der vollständigen Zeitreihe:

zum Zeitpunkt *TRecBeg* (Takt 0): Werte aller Variablen
bei Wertänderung einer Variablen: Wert dieser Variablen




Wertetabelle:

| *Zeit* | *Takt* | *Wert* |
|---|---|---|
| 0 | 0 | 10 |
| 10 | 1 | 15 |
| 25 | 1 | 20 |
| 33 | 1 | 13 |
| 48 | 1 | 18 |
| 48 | 2 | 10 |
| 59 | 1 | 15 |
| 63 | 1 | 20 |
| 73 | -1 | 8 |
| 79 | 1 | 13 |

Die '-1' im Takt (anstelle der '0') zeigt an, daß die Wertänderung aufgrund eines Bereichswechsels zustande kam.

Aufzeichnung der komprimierten Zeitreihe

Die Aufzeichnung erfolgt zu den äquidistanten Zeitpunkten

$$TRecBeg \quad + \quad n \cdot Width \quad , \qquad n = 1, 2, 3, \ldots$$

sowie zum Zeitpunkt

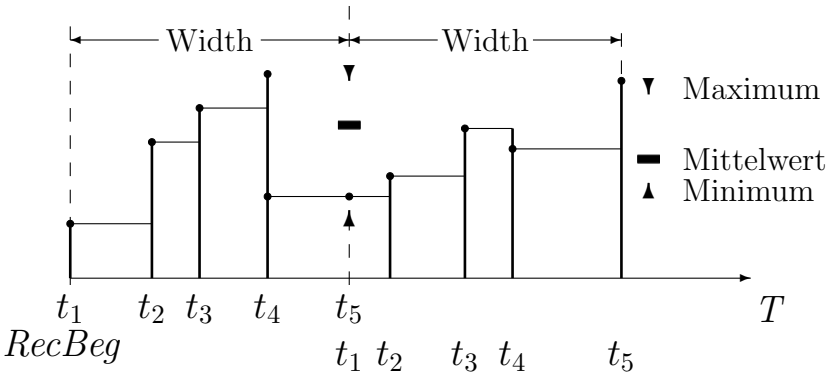
*TRecEnd*.

Zur Bestimmung des Minimums bzw. Maximums eines Intervalls werden alle Werte, die in diesem Intervall liegen, herangezogen, einschließlich denen, die auf der Intervallgrenze liegen. Zur Ermittlung des zeitlichen Mittelwerts wird das Integral unter der Funktion gebildet und durch die Intervallbreite dividiert.

Ein Intervall enthalte $M$ Werte $x(t_i), i\epsilon\{1..M\}$ und beginne zum Zeitpunkt $t_1$ und ende zum Zeitpunkt $t_M$. Liegen zu einem Zeitpunkt mehrere Werte vor, wird nur der zuletzt gültige herangezogen.

Der zeitliche Mittelwert des betrachteten Intervalls ergibt sich damit zu

$$Mean = \frac{1}{Width} \cdot \sum_{i=1}^{M-1} x(t_i)(t_{i+1} - t_i)$$




Es werden drei Wertetabellen angelegt:

**für Mittelwert**

| *Zeit* | *Wert* |
|---|---|
| 40 | 21 |
| 80 | 18 |
| ⋮ | |

**für Minimum**

| *Zeit* | *Wert* |
|---|---|
| 40 | 8 |
| 80 | 12 |
| ⋮ | |

**für Maximum**

| *Zeit* | *Wert* |
|---|---|
| 40 | 30 |
| 80 | 29 |
| ⋮ | |

## 2.) Zeitreihen für kontinuierliche Variablen

Erforderliche Angaben:

- Beginn der Aufzeichnung *TRecBeg*
- Ende der Aufzeichnung *TRecEnd*
- Abtastabstand *DTSample*

zusätzlich für komprimierte Zeitreihe:

- Intervallbreite in Vielfachen des Abtastabstands *NSample*
  $\rightarrow Width = NSample \cdot DTSample$

Aufzeichnungszeitpunkte der vollständigen Zeitreihe:

- vom Beginn der Aufzeichnung im Abtastabstand

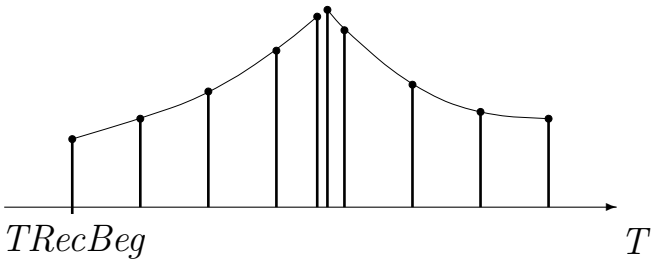
  $$TRecBeg + n \cdot DTSample, \quad n = 0, 1, 2, \ldots$$

- zum Ende der Aufzeichnung *TRecEnd*
- vor und nach einem Bereichswechsel
- vor und nach einem Ereignis

Es werden stets alle Variablen des Beobachters aufgezeichnet. An Unstetigkeitsstellen wird auch abweichend vom Zeitraster aufgezeichnet, um auch eventuelle Sprung- oder Knickstellen im Zeitverlauf zu erfassen. Damit bei der Weiterverarbeitung der aufgezeichneten Information Bereichswechsel noch als solche erkannt werden, wird als *Takt* statt der '0' eine '-1' zu dem Zeitpunkt eingetragen, der auf den Bereichswechsel folgt.

Da Unstetigkeiten in Modellen mit kontinuierlichen Zeitverläufen eine besondere Rolle spielen, wollen wir die Aufzeichnung des Zeitverlaufs an diesen Stellen gesondert diskutieren. Wir können dabei vier verschiedene Fälle unterscheiden. Für die Beispiele gehen für einen Abtastintervall *DTSample = 10* und einem Auflösungsvermögen *TScal = 1* aus.

a) Bereichswechsel mit Knickstelle




z.B. durch Zustandsgröße:

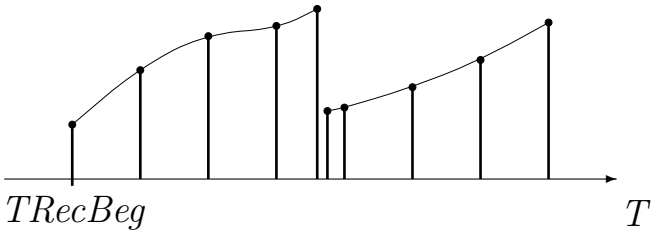
$$z' := \begin{cases} z & \text{für } T < 36.5 \\ -z & \text{für } T \geq 36.5 \end{cases}$$

z.B. durch abh. Größe:
$y := \text{MIN}(u, v);$

Wertetabelle:

| *Zeit* | *Takt* | *Wert* |
|---|---|---|
| ⋮ | | |
| 20 | 0 | 17 |
| 30 | 0 | 23 |
| 36 | 0 | 28 |
| 37 | -1 | 29 |
| 40 | 0 | 26 |

b) Bereichswechsel mit Sprungstelle



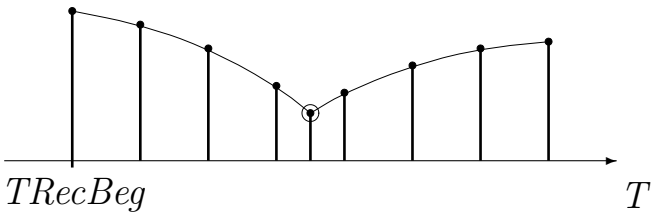

z.B. durch abh. Größe:

$$y := \begin{cases} u & \text{für } T < 36.5 \\ v & \text{für } T \geq 36.5 \end{cases}$$

Wertetabelle:

| *Zeit* | *Takt* | *Wert* |
|---|---|---|
| ⋮ | | |
| 20 | 0 | 21 |
| 30 | 0 | 24 |
| 36 | 0 | 25 |
| 37 | -1 | 10 |
| 40 | 0 | 11 |
| ⋮ | | |

c) Ereignis mit Knickstelle




z.B. durch Zustandsgrößen:

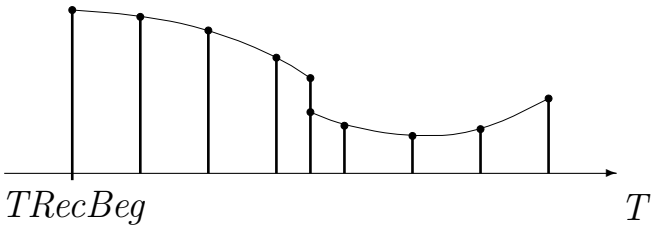
$$z_1' := u;$$
$$\hat{z_1} := w \; sobald \; T \geq 34.5$$

$$z_2' := z_{1;}$$

Wertetabelle:

| *Zeit* | *Takt* | *Wert* |
|---|---|---|
| ⋮ | | |
| 20 | 0 | 16 |
| 30 | 0 | 11 |
| 35 | 0 | 7 |
| 35 | 1 | 7 |
| 40 | 0 | 10 |
| ⋮ | | |

d) Ereignis mit Sprungstelle




z.B. durch Zustandsgröße:

$$z' := u;$$
$$\hat{z} := w \; sobald \; T \geq 34.5$$

z.B. durch abh. Größe:

$$y := 2z + 1;$$

Wertetabelle:

| *Zeit* | *Takt* | *Wert* |
|---|---|---|
| ⋮ | | |
| 20 | 0 | 21 |
| 30 | 0 | 18 |
| 35 | 0 | 14 |
| 35 | 1 | 9 |
| 40 | 0 | 7 |
| ⋮ | | |

Es ist möglich, daß sich eine Unstetigkeit auf die aufgezeichnete Variable überhaupt nicht auswirkt. Da sich dies im gegenwärtigen Entwicklungs-

stand nicht feststellen läßt, wird vorsichtshalber bei jeder kontinuierlichen Variablen eine Unstetigkeit eingetragen.

Aufzeichnungszeitpunkte der komprimierten Zeitreihe:

Die Aufzeichnung erfolgt zu den Zeitpunkten

$$TRecBeg + n \cdot NSample \cdot DTSample \quad , \quad n = 1, 2, 3, \ldots$$

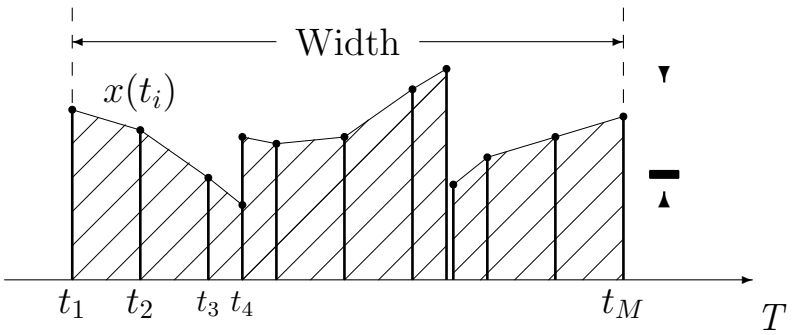
sowie zum Zeitpunkt

*TRecEnd*.

Aufgezeichnete Werte der komprimierten Zeitreihe:

Zur Bestimmung des Minimums bzw. Maximums eines Intervalls werden alle bekannten Werte, die in diesem Intervall liegen, herangezogen. Zur Ermittlung des zeitlichen Mittelwerts wird das Integral unter der Funktion nach der Trapezregel gebildet und durch die Intervallbreite dividiert.

Abschnitte im Intervall, zu denen Bereichswechsel vorliegen, werden nicht zur Bestimmung des zeitlichen Mittelwerts herangezogen. Innerhalb dieser Abschnitte ist der zeitliche Verlauf der Variablen unbekannt.




Wird mit $\bar{}x(t)$ der erste Wert sowie mit $x^{\bar{}}(t)$ der letzte Wert eines Zeitpunkts bezeichnet und liegen im betrachteten Intervall $k$ Bereichswechsel vor, ergibt sich der zeitliche Mittelwert zu

$$\text{Mean} = \frac{1}{\text{Width} - k * \text{TScal}} \cdot \sum_{\substack{i=1 \\ Takt(i+1) \neq -1}}^{M-1} \frac{1}{2} \left[ \bar{}x(t_{i+1}) + x^{\bar{}}(t_i) \right] (t_{i+1} - t_i)$$

### Die Implementierung der Beobachter

Der Experimentierprozeß hinterlegt die Spezifikation der angelegten Beobachter auf der Datei

```
Monitor.cont .
```

Die gewünschten Wertetabellen werden auf der Datei

```
Monitor.out .
```

aufgezeichnet. Die Information wird dabei nach einem index-sequentiellen Verfahren abgelegt, um auch bei großen Datenmengen einen schnellen Zugriff auf Ausschnitte der Wertetabelle zu ermöglichen.

Zum Bearbeiten dieser beiden Dateien wurde eine Schnittstelle geschaffen, die aus den Funktionen

```
IniWrite   WrData   EndWirte
```

besteht, welche in der Datei

```
WriteSim.c
```

abgelegt sind.

Die Dateiorganisation sowie die Funktionen der Schnittstelle werden in der Dissertation von Klaus Dörnhöfer [Dörn 90] beschrieben.

Zum Betreiben der Beobachter enthält das Laufzeitsystem die Funktionen

```
MonRead   Monitor   MonWrite
```

Die Funktion `MonRead` liest vor der Ausführung eines Simulationslaufs die Spezifikation der vereinbarten Beobachter sowie Offset-Adressen und Typen der aufzuzeichnenden Variablen ein und legt diese Informationen im Arbeitsspeicher ab.

Die Funktion `Monitor` zeichnet die gewünschten Wertetabellen auf. Sie wird von der Ablaufsteuerung immer dann aufgerufen, wenn ein vollständiger Modellzustand hergestellt ist.

Die Funktion `MonWrite` wird am Ende eines Simulationslaufs ausgeführt und nimmt abschließende Einträge in die Wertetabelle vor.

### 6.5.7 Die Behandlung von Laufzeitfehlern

Ein Laufzeitfehler kann durch die Prüfung einer Bedingung oder durch ein Signal vom Betriebssystem ausgelöst werden.

In beiden Fällen verzweigt der Programmfluß in die Funktion

```
void    Error (ErrNo)
int     ErrNo;
```

der die Nummer des erkannten Fehlers mitgegeben wird.

Diese Routine gibt eine entsprechende Fehlermeldung sowohl im Arbeitsfenster des Bildschirms als auch auf die Datei *Protocol.out* bzw. *Structure.out* aus.

Die ausgegebene Fehlermeldung enhält:

- den Zeitpunkt
- den vollständigen Pfadnamen der Modellkomponente, in welcher der Fehler auftrat
- die Zeilennummer im Quelltext
- den Fehlerkommentar

Der Pfadname der Modellkomponente läßt sich durch Verfolgung des globalen Zeigers

```
struct  NAME_ELEM  *  ActComp;
```

rekonstruieren, der auf die Namensstruktur der aktuell bearbeiteten Komponente verweist.

Die Zeilennummer im Quelltext kann der globalen Variablen

```
int  ActLine
```

entnommen werden.

In einigen Fällen ist es notwendig, den Simulationsprozeß unmittelbar abzubrechen. Normalerweise jedoch wird am Wiederaufsetzpunkt im Hauptprogramm (unmittelbar vor Aufruf der Ablaufkontrolle) fortgefahren. Der weitere Ablauf ist der gleiche als wäre der Simulationslauf ohne Fehler beendet worden.

Laufzeitfehler werden danach unterschieden, wie nach ihrem Auftreten fortgefahren werden kann. Dies zeigt der Fehlertyp an:

Fehlertyp 1: Weitere Simulationsläufe sind möglich, Simulationsprozeß wird nicht beendet.

Fehlertyp 2: Simulationsprozeß wird nach Übergabe der Ergebnisse an Experimentierprozeß beendet.

Fehlertyp 3: Simulationsprozeß wird ohne Rückkehr ins Hauptprogramm nach Ausgabe einer Fehlermeldung beendet.

Fehlertyp 4: Simulationsprozeß wird sofort beendet.

Fehler vom Typ 1 treten auf bei einer fehlerhaften Belegung der Steuerparameter oder bei einer fehlerhaften Beschreibung des Modells. Sie sind stets auf den Benutzer zurückzuführen. Hierzu zählen:

1. Obergrenze Iterationen überschritten
2. Obergrenze Ereignisfortpflanzungen überschritten
3. Obergrenze Crossingsuchschritte überschritten
4. Obergrenze Regionsuchschritte überschritten
5. Argument einer Standardfunktion außerhalb des zulässigen Wertebereichs
6. Argument einer Tabellenfunktion außerhalb des zulässigen Wertebereichs
7. Mehrfache Zuweisung an Zustandsvariable im selben Ereignistakt
8. Integrationsschrittweite kleiner `TScal` erforderlich
9. Gleitkomma-Überlauf

Fehler vom Typ2 treten bei einer fehlerhaften Programmierung des Laufzeitsystems oder bei einer fehlerhaften Codegenerierung des MDL-Compilers auf. Hierzu zählen:

1. Fehler in der Zeitführung des Simulators
2. Signal: *Bus Error*
3. Signal: Segmentation Violation

Um in einem solchen Fall eine Fehlerverfolgung zu ermöglichen, wird versucht, den Endezustand an den Experimentierprozeß zu übertragen, bevor der Simulationsprozeß beendet wird.

Fehler vom Typ 3 treten bei Kommunikationsschwierigkeiten zwischen den Prozessen und mit dem Terminal auf. Hierzu zählen:

1. Signal: Leitungsunterbrechung zum Terminal
2. Signal: Broken Pipe

In diesem Fall ist eine Fortsetzung des Simulationsprozesses nicht sinnvoll.

Der Fehlertyp 4 liegt vor, wenn die Funktion `Error` betreten wird, obwohl bereits ein Fehler angezeigt wurde. Es muß daher ein Folgefehler vorliegen. Der Simulationsprozeß wird sofort beendet.

# 7 Anwendungsbeispiele

Einige Anwendungsbeispiele sollen zeigen, wie Modelle in der Beschreibungssprache SIMPLEX-MDL formuliert werden. Wie wir hoffen, wird damit demonstriert, daß ein Modell auch ohne weitere Vorkenntnisse über das betrachtete System verstanden werden kann.

Die Modelle wurden so ausgewählt, daß alle Konstrukte der Sprache Verwendung finden. Die aus mehreren Komponenten zusammengesetzten Modelle wurden den Gebieten der Regelungstechnik und des digitalen Schaltungsentwurfs entnommen, die für eine systemtheoretische Betrachtungsweise typisch sind.

Es ist uns bewußt, daß die Modelle sehr unzulänglich sind, wollte man sie auf reale Gegebenheiten anwenden. Da jedoch die Demonstration der Sprache im Vordergrund stand, wurde auf viele notwendige Details verzichtet und eine Reihe von Vereinfachungen getroffen.

## 7.1 Räuber-Beute-Modell

Das nachstehende Modell beschreibt die Beziehungen zwischen einer Räuber- und einer Beutepopulation, die im gleichen Revier leben.

Eine Weidefläche ernährt eine bestimmte Anzahl von Hasen. Bei Erreichen dieser Anzahl geht die Zuwachsrate der Hasen auf Null zurück, d.h. der Bestand bleibt konstant. Ist die maximale Weidekapazität noch nicht ausgeschöpft, erhöht sich die Zuwachsrate entsprechend.

Der Bestand an Hasen wird durch Füchse dezimiert. Die Anzahl der gefangenen Hasen ergibt sich aus der Begegnungswahrscheinlichkeit auf der Weidefläche. Um den Bestand an Füchsen zu halten, liegt der Nahrungsbedarf $e$ eines Fuchses bei 1 Hasen pro Woche. Erhält ein Fuchs keine Nahrung, nimmt seine Biomasse mit der Rate $b = 0.2/Woche$ ab. Die Bestände der Hasen und Füchse pendeln sich mit der Zeit auf feste Werte ein.

Zum Zeitpunkt *THalb* und alle weiteren 1000 Zeiteinheiten (Wochen) wird die Weidekapazität halbiert. Diese Veränderungen führen jeweils zu neuen Gleichgewichtszuständen.

```
BASIC COMPONENT   HaFuchs

DECLARATION OF ELEMENTS

   CONSTANTS
       a (REAL) :=  0.08  ,  # Zuwachsrate der Hasen [1/Woche]
       b (REAL) :=  0.2   ,  # Verlust an Biomasse eines Fuchses [1/Woche]
       c (REAL) :=  0.002 ,  # gefangene Hasen pro Woche, Fuchs und Hasendichte
                             # [Hasen / (Woche * Fuchs * (Hasen/ha))]
       e (REAL) :=  1.0   ,  # Nahrungsbedarf eines Fuchses [Hasen/(Fuchs*Woche)]
       F (REAL) :=  1.0   ,  # Weideflaeche [ha]
   THalb (REAL) :=   500  ,  # Zeitpunkt einer Halbierung der Weidekap. [Woche]

   STATE VARIABLES
   DISCRETE    relKap (REAL) :=  1000    # relative Weidekapazitaet [Hasen/ha]

   CONTINUOUS   Hasen (REAL) :=   500 ,  # Biomasse (Anzahl) der Hasen
              Fuechse (REAL) :=    10    # Biomasse (Anzahl) der Fuechse

   DEPENDENT VARIABLES
   DISCRETE       Kap (REAL)         # Weidekapazitaet auf betrachteter Flaeche
   CONTINUOUS                        # [Hasen]
               FHasen (REAL) ,       # Anzahl gefangener Hasen pro Woche [1/Woche]
                 EWBV (REAL)         # Einfluss der Weidebeschraenkung auf
                                     # Vermehrung

DYNAMIC BEHAVIOUR

    Kap    :=  relKap * F;
    FHasen :=  c * (Hasen/F) * Fuechse;
    EWBV   :=  (Kap - Hasen) / Kap;

    DIFFERENTIAL EQUATIONS
         Hasen' :=  EWBV * a * Hasen   -       FHasen;
      Fuechse' :=       - b * Fuechse + b/e * FHasen;
    END

    WHENEVER   T >= THalb
    DO
       relKap^ :=  relKap / 2;
        THalb^ :=  T + 1000;
    END


END OF   HaFuchs
```

## 7.2 Modell „Springender Ball“

Das Modell *Springender Ball* beschreibt einen Ball, der aus einer Höhe von 1 Meter und einer Horizontalgeschwindigkeit von 0.3 m/sec losgelassen wird und eine Treppe hinunterspringt.

Die rücktreibende Kraft während der Reflexion des Balls auf den Treppenstufen setzt sich aus einem linearen Anteil und einem hyperbolischen Anteil zusammen. Die Dämpfung erfolgt geschwindigkeitsproportional. Die Position des Balls wird durch seinen Mittelpunkt beschrieben.

Das Modell zeigt verschiedene Varianten zur Erzeugung diskontinuierlicher Zeitfunktionen. Insbesondere weist die Ableitung der Geschwindigkeit Knickstellen auf.

```
BASIC COMPONENT   Ball                # Modell eines springenden Balls

DECLARATION OF ELEMENTS

   CONSTANTS
               M (REAL) :=  0.05   ,  # [kg]         Masse des Balls
          Radius (REAL) :=  0.05   ,  # [m]          Radius des Balls
              K1 (REAL) :=  125.0  ,  # [N/m]        Federkonstante (linear)
              K2 (REAL) :=  15.0   ,  # [Nm]         Federkonstante (nichtlinear)
               D (REAL) :=  2.5    ,  # [N/(m/sec)] Daempfung
               g (REAL) :=  9.81   ,  # [m/sec^2]    Erdbeschleunigung
              Vx (REAL) :=  0.3    ,  # [m/sec]      horizontale Geschwindigkeit
        TrBreite (REAL) :=  1.5    ,  # [m]          Gesamtbreite der Treppe
        StBreite (REAL) :=  0.2    ,  # [m]          Breite einer Treppenstufe
         StHoehe (REAL) :=  0.2    ,  # [m]          Hoehe einer Treppenstufe

   STATE VARIABLES
   CONTINUOUS
               Y (REAL) :=  1.0 ,     # [m]          vertikale Position des Balls
              Vy (REAL) :=  0.0       # [m/sec]      vertikale Geschwindigkeit
                                      #              des Balls
   DEPENDENT VARIABLES
   DISCRETE
            Stufe (INT) ,         #             Anzahl ueberquerter Stufen
        Y_Treppe (REAL)           # [m]         vertikale Position der Treppenstufe

   CONTINUOUS
               X (REAL) ,         # [m]         horizontale Position des Balls
              Ay (REAL) ,         # [m/sec^2]   Beschleunigung des Balls bei Reflexion
        Abstand (REAL) := 1.0     # [m]         Abstand des Balls zur Treppenstufe
```

```
DYNAMIC BEHAVIOUR

#  -------------------------- Verhalten des Balls ----------------------------

   X :=  Vx * T;                        # gleichfoermige horizontale Bewegung

   DIFFERENTIAL EQUATIONS
      Y' :=  Vy;
     Vy' :=  Ay - g;
   END

   IF   Abstand < Radius
      DO                                                  # Reflexion:
          Ay :=   (K1/M) * (Radius - Abstand)             # linearer Anteil
                + (K2/M) * (1/Abstand - 1/Radius)         # nichtlinearer Anteil
                - (D/M)  * Vy;                            # Daempfung
      END
   ELSE
      DO
          Ay :=  0;
      END

#  ---------------- Abstand des Balls von der Treppenstufe --------------------

   IF   X < TrBreite   DO   Stufe :=  ITRUNC (X/StBreite);         END
               ELSE    DO   Stufe :=  ITRUNC (TrBreite/StBreite);  END

   Y_Treppe :=  - Stufe * StHoehe;

   Abstand  :=  Y - Y_Treppe;

END OF   Ball
```

## 7.3 Modell „Heizungsregelung“

Das Modell *Heizungsregelung* zeigt einen typischen einläufigen Regelkreis. Die Regelstrecke ist ein Zimmer, das auf konstanter Temperatur gehalten werden soll. Eine Vergleichskomponente liefert die Abweichung der Ist-Temperatur von einer vorgegebenen Soll-Temperatur. Der Regler liefert als Stellgrößen die Vorlauftemperatur und den Volumenstrom des Heizwassers.

Die Temperaturen von Zimmer und Heizkörper werden indirekt über die aktuellen Wärmemengen bestimmt. Die Wärmemenge der Zimmerluft nimmt durch den Wärmestrom vom Heizkörper zu und durch den Wärmestrom, der durch die Außenwand geht, ab. Die Wärmemenge des Heizkörpers vergrößert sich durch das ausgetauschte Wasser und vermindert sich durch die abgegebene Heizleistung. Die Temperatur im Ablauf des Heizkörpers liegt zwischen der mittleren Temperatur des Heizkörpers und der Zimmertemperatur. Um das Modell einfach zu halten, wird der arithmetische Mittelwert eingesetzt.

Das Verhalten des Reglers wird durch zwei Kennlinien beschrieben, die in Tabellenfunktionen abgelegt sind.

Die Zusammenschaltung der einzelnen Basiskomponenten ergibt eine höhere Komponente, die wiederum Anschlüsse zur Außenwelt erhält. Für einen Testbetrieb mit konstanten Werten, können diese Anschlüsse geeignet vorbesetzt werden.

```
HIGH LEVEL COMPONENT   Heizung

SUBCOMPONENTS
   Zimmer, Regler, Vergl

INPUT CONNECTIONS
   TAussen  -->  (Regler.TAussen, Zimmer.TAussen);
     TSoll  -->  Vergl.TSoll;

OUTPUT EQUIVALENCES
   TIst :=  Zimmer.TRaum;

COMPONENT CONNECTIONS
   TIst             -->  Vergl.TIst;
   Vergl.TAbw       -->  Regler.TAbw;
   Regler.IVol      -->  Zimmer.IVol;
   Regler.TVorlauf  -->  Zimmer.TVorlauf;

INITIALIZE
   TAussen :=  - 10;
   TSoll   :=  + 20;

END OF   Heizung
```

*Heizung*

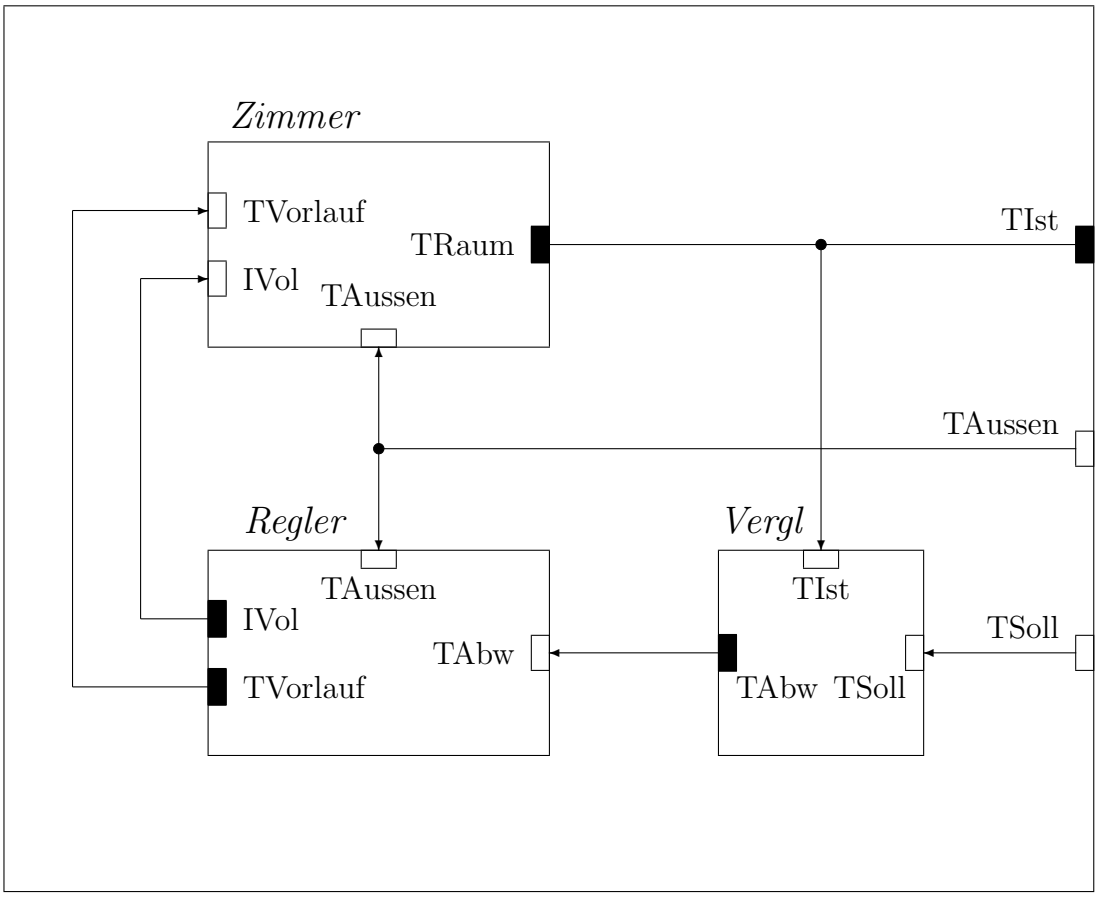



Abb. 7.3-1: Modell einer Heizungsregelung

```
BASIC COMPONENT   Zimmer

DECLARATION OF ELEMENTS

   CONSTANTS

      KapLuft (REAL) :=  1.0 E+3 ,   # spez. Waermekapazitaet der Luft [J/(kg*K)]
    KapWasser (REAL) :=  4.2 E+3 ,   # spez. Waermekap. des Wassers [J/(kg*K)]
      RhoLuft (REAL) :=  1.3     ,   # Dichte der Luft [kg/m^3]
    RhoWasser (REAL) :=  1000    ,   # Dichte des Wassers [kg/m^3]
      VolRaum (REAL) :=  40.0    ,   # Raumvolumen [m^3]
     VolHeizk (REAL) :=  10.0 E-3,   # Inhalt des Heizkoerpers [m^3]
        KWand (REAL) :=   2.0    ,   # Waermedurchgangskoeff. Wand [W/(K*m^2)]
       KHeizk (REAL) :=  60.0    ,   # Waermeuebergangskoeff. Heizk. [W/(K*m^2)]
        AWand (REAL) :=  20.0    ,   # Flaeche der Wand [m^2]
       AHeizk (REAL) :=   6.0    ,   # Flaeche des Heizkoerpers [m^2]

   STATE VARIABLES
   CONTINUOUS
        QRaum (REAL) :=  0 ,         # Waermemenge im Raum [J]
```

```
      QHeizk (REAL) :=  0           # Waermemenge im Heizkoerper [J]
                                    # (Waermemengen bezogen auf 0 Grad Celsius)
   DEPENDENT VARIABLES
   CONTINUOUS
       TRaum (REAL) ,     # Lufttemperatur im Raum [Grad Celsius]
      THeizk (REAL) ,     # mittlere Wassertemperatur im Heizk. [Grad Celsius]
     TAblauf (REAL)       # Wassertemperatur am Ablauf des Heizk. [Grad Celsius]

   SENSOR VARIABLES
   CONTINUOUS
     TAussen (REAL) :=   0 ,        # Aussentemperatur [Grad Celsius]
    TVorlauf (REAL) :=  60 ,        # Vorlauftemperatur des Heizk. [Grad Celsius]
        IVol (REAL) :=   0.001      # Volumenstrom durch Heizkoerper [m^3/sec]


DYNAMIC BEHAVIOUR

     TRaum :=  QRaum  / (KapLuft   * VolRaum  * RhoLuft);
    THeizk :=  QHeizk / (KapWasser * VolHeizk * RhoWasser);

   TAblauf :=  0.5 * (THeizk + TRaum);

   DIFFERENTIAL EQUATIONS
       QRaum' :=    KWand  * AWand  * (THeizk - TRaum)
                  - KHeizk * AHeizk * (TRaum  - TAussen);

      QHeizk' :=   KapWasser * IVol * (TVorlauf - TAblauf)
                 - KHeizk * AHeizk * (THeizk - TRaum);
   END

END OF   Zimmer

------------------------------------------------------------------------------

BASIC COMPONENT   Regler

LOCAL DEFINITIONS

   TABULAR FUNCTION   StellVor (REAL --> REAL)  # Vorlauftemperatur [Grad Celsius]
   CONTINUOUS                                   # in Abhaengigkeit von der
   BY LINEAR INTERPOLATION                      # Aussentemperatur [Grad Celsius]
   ON   ( -100, -20,  0, 20, 100 )
   -->  (   80,  80, 60, 50,  50 )

   TABULAR FUNCTION   StellVol (REAL --> REAL)   # Reglereingriff in Abhaengigkeit
   CONTINUOUS                                    # von der Temperaturabweichung
   BY LINEAR INTERPOLATION                       # [Grad Celsius]
```

```
   ON   ( -100, -5, 0.0, 5, 100 )
   -->  (    1,  1, 0.5, 0,   0 )

DECLARATION OF ELEMENS

   CONSTANTS       A (REAL) := 4.0 E-4    # Leitungsquerschnitt [m^2]
                   v (REAL) := 2.5        # Fliessgeschwindigkeit [m/sec]

   DEPENDENT VARIABLES
   CONTINUOUS      IVol (REAL) ,          # Volumenstrom des Heizwassers [m^3/sec]
               TVorlauf (REAL)            # Vorlauftemperatur des Heizwassers
                                          # [Grad Celsius]

   SENSOR VARIABLES
   CONTINUOUS
        TAbw (REAL) ,          # Abweichung von der Solltemperatur [Grad Celsius]
     TAussen (REAL)            # Aussentemperatur [Grad Celsius]


DYNAMIC BEHAVIOUR

       IVol :=  StellVol (TAbw) * A * v;
   TVorlauf :=  StellVor (TAussen);

END OF   Regler

-------------------------------------------------------------------------------

BASIC COMPONENT   Vergl

DECLARATION OF ELEMENTS

   DEPENDENT VARIABLES
   CONTINUOUS   TAbw (REAL)            # Abweichung der Temperatur [Grad Celsius]

   SENSOR VARIABLES
   CONTINUOUS  TSoll (REAL) ,          # Soll-Temperatur [Grad Celsius]
               TIst (REAL)             # Ist-Temperatur  [Grad Celsius]

DYNAMIC BEHAVIOUR

   TAbw :=  TIst - TSoll;

END OF   Vergl
```

## 7.4 Modell „4-Bit-Addierer“

Das Modell *4-Bit-Addierer* ist ein schaltungstechnisches Modell, das aus den elementaren Logik-Bausteinen AND, OR und EXOR aufgebaut ist. Diese werden zu Halbaddierern, Volladdierern und schließlich zu einem 4-Bit-Addierer verschaltet. Die Strukturen können beispielsweise /TieSche 78/ entnommen werden.

Dieses Beispiel zeigt die Gestaltung von Modellen mit höheren Hierarchiestufen sowie die Anwendung des Klassenkonzepts. Man erkennt, daß es für eine höhere Komponente keinen Unterschied macht, ob deren Subkomponenten Basiskomponenten oder ebenfalls höhere Komponenten sind.

Betreibt man die Basiskomponenten (Logik-Gatter) des Modells in der Version *Logic*, kann lediglich die Funktion des Schaltnetzes nachgewiesen werden. Mit der Version *Delay* können die Signallaufzeiten verfolgt werden.

Eine Verfeinerung des Modells kann andere Wertemengen der Eingangs- und Ausgangsgrößen mit sich bringen. Eine genauere Modellierung eines Gatters bedingt beispielsweise einen Übergang von logischen Werten auf Aufzählungswerte oder gar auf reelle Zahlen. Da Größen in übergeordneten Komponenten implizit angelegt werden, indem die Datentypen übernommen werden, bleibt die Modellbeschreibung auf höherer Ebene im wesentlichen unverändert. Nur zur Initialisierung der Modellgrößen sind unter Umständen andere Werte einzusetzen.

Die Eingangs- und Sensorgrößen sind im angegebenen Beispiel so vorbelegt, daß ein offener Eingang die Funktion des Schaltkreises nicht beeinträchtigt.

```
BASIC COMPONENT   And                # Version "Logic"

DECLARATION OF ELEMENTS

   DEPENDENT VARIABLES   Out (LOGICAL)

   SENSOR VARIABLES      In1 (LOGICAL) :=  TRUE ,
                         In2 (LOGICAL) :=  TRUE ,
                         In3 (LOGICAL) :=  TRUE ,
                         In4 (LOGICAL) :=  TRUE
```

```
DYNAMIC BEHAVIOUR

   Out :=  In1  AND  In2  AND  In3  AND  In4;

END OF   And

-----------------------------------------------------------------------

BASIC COMPONENT   Or                # Version "Logic"

DECLARATION OF ELEMENTS

   DEPENDENT VARIABLES   Out (LOGICAL)

   SENSOR VARIABLES      In1 (LOGICAL) :=  FALSE ,
                         In2 (LOGICAL) :=  FALSE ,
                         In3 (LOGICAL) :=  FALSE ,
                         In4 (LOGICAL) :=  FALSE

DYNAMIC BEHAVIOUR

   Out :=  In1  OR   In2   OR   In3   OR   In4;

END OF   Or

-----------------------------------------------------------------------

BASIC COMPONENT   XOr               # Version "Logic"

DECLARATION OF ELEMENTS

   DEPENDENT VARIABLES   Out (LOGICAL)

   SENSOR VARIABLES      In1 (LOGICAL) :=  FALSE,
                         In2 (LOGICAL) :=  FALSE

DYNAMIC BEHAVIOUR

   Out :=  In1  <>  In2;

END OF   XOr

-----------------------------------------------------------------------

BASIC COMPONENT   And                         # Version "Delay"

LOCAL DEFINITION

   VALUE SET   State: (high, low, undef)    # Werte der Zeitfunktionen
```

```
DECLARATION OF ELEMENTS

   CONSTANT      Delay (REAL)  :=  10       # Gatterlaufzeit [nsec]

   STATE VARIABLES
                   Out (State) :=  undef ,  # Ausgangssignal
                NxtOut (State) :=  undef ,  # Zukuenftiges Ausgangssignal
               TSwitch (REAL)  :=  0     ,  # Schaltzeitpunkt [nsec]

   DEPENDENT VARIABLES
                    In (State)              # Wert der Schaltfunktion

   SENSOR VARIABLES
                   In1 (State) :=  high ,   # Eingangssignale
                   In2 (State) :=  high ,
                   In3 (State) :=  high ,
                   In4 (State) :=  high

DYNAMIC BEHAVIOUR

   IF  (In1 = high)  AND  (In2 = high)  AND  (In3 = high)  AND  (In4 = high)
     DO
       In :=  high;
     END
   ELSIF  (In1 = low)  OR  (In2 = low)  OR  (In3 = low)  OR  (In4 = low)
     DO
       In :=  low;
     END
   ELSE
     DO
       In :=  undef;
     END

   WHENEVER   In  <>  NxtOut
   DO
           Out^ :=  undef;
        NxtOut^ :=  In;
       TSwitch^ :=  T + Delay;
   END

   WHENEVER   (Out <> NxtOut)  AND  (T >= TSwitch)
   DO
      Out^ :=  NxtOut;
   END

END OF   And
```

```
HIGH LEVEL COMPONENT   HalfAddi

SUBCOMPONENTS   And, XOr

INPUT CONNECTIONS
   a -->  (XOr.In1,  And.In1);          # 1. Summand
   b -->  (XOr.In2,  And.In2);          # 2. Summand

OUTPUT EQUIVALENCES
   s :=  XOr.Out;                       # Summen-Bit
   c :=  And.Out;                       # Uebertrags-Bit

END OF   HalfAddi
```

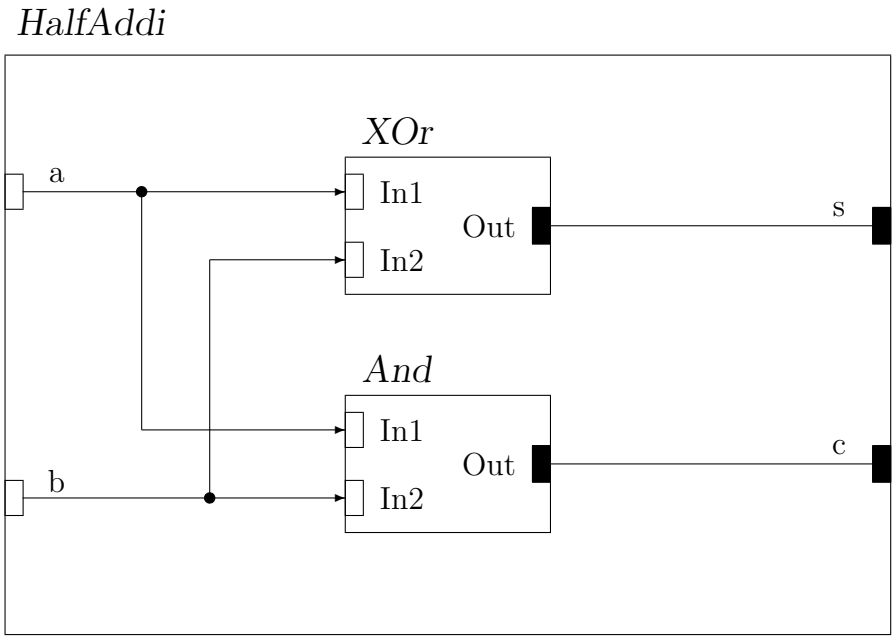



Abb. 7.4-1: Modellkomponente Halbaddierer

```
HIGH LEVEL COMPONENT   Addi

SUBCOMPONENTS   Half1  OF CLASS  HalfAddi,
                Half2  OF CLASS  HalfAddi,
                Or
```

```
INPUT CONNECTIONS
   cl  -->  Half2.a;                  # uebernommenes Uebertrags-Bit
    a  -->  Half1.a;                  # 1. Summand
    b  -->  Half1.b;                  # 2. Summand

OUTPUT EQUIVALENCES
   s :=  Half2.s;                     # Summen-Bit
   c :=  Or.Out;                      # Uebertrags-Bit

COMPONENT CONNECTIONS
   Half1.s  -->  Half2.b;
   Half1.c  -->  Or.In1;
   Half2.c  -->  Or.In2;

INITIALIZE
   cl :=  FALSE;

END OF   Addi
```

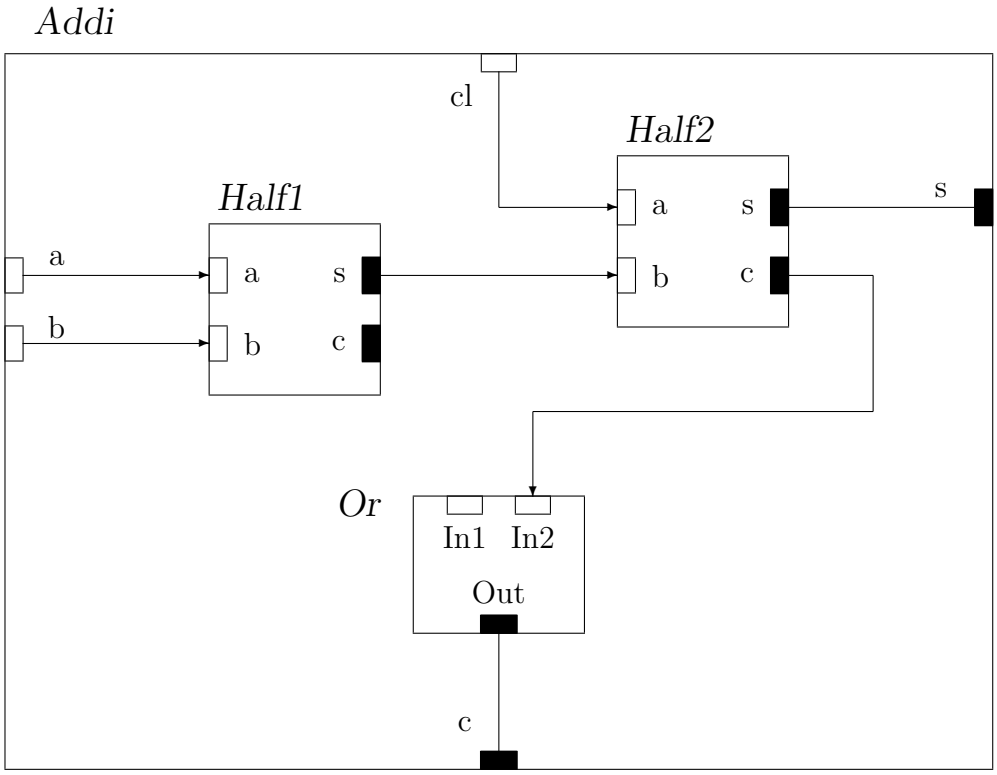



Abb. 7.4-2: Modellkomponente Volladdierer

```
HIGH LEVEL COMPONENT   Addi4Bit

SUBCOMPONENTS   Addi0  OF CLASS  Addi,
                Addi1  OF CLASS  Addi,
                Addi2  OF CLASS  Addi,
                Addi3  OF CLASS  Addi


INPUT CONNECTIONS

   cl  -->  Addi0.cl;               # uebernommenes Uebertrags-Bit

   a0  -->  Addi0.a;                # Summanden der 0-ten Stelle
   b0  -->  Addi0.b;

   a1  -->  Addi1.a;                # Summanden der 1. Stelle
   b1  -->  Addi1.b;

   a2  -->  Addi2.a;                # Summanden der 2. Stelle
   b2  -->  Addi2.b;

   a3  -->  Addi3.a;                # Summanden der 3. Stelle
   b3  -->  Addi3.b;


OUTPUT EQUIVALENCES

   s0 :=  Addi0.s;                  # Summe 0-tes Bit
   s1 :=  Addi1.s;                  # Summe 1. Bit
   s2 :=  Addi2.s;                  # Summe 2. Bit
   s3 :=  Addi3.s;                  # Summe 3. Bit

    c :=  Addi3.c;                  # Summe Ueberlaufbit


COMPONENT CONNECTIONS

   Addi0.c  -->  Addi1.cl;
   Addi1.c  -->  Addi2.cl;
   Addi2.c  -->  Addi3.cl;


INITIALIZE

   cl :=  FALSE;

END OF   Addi4Bit
```

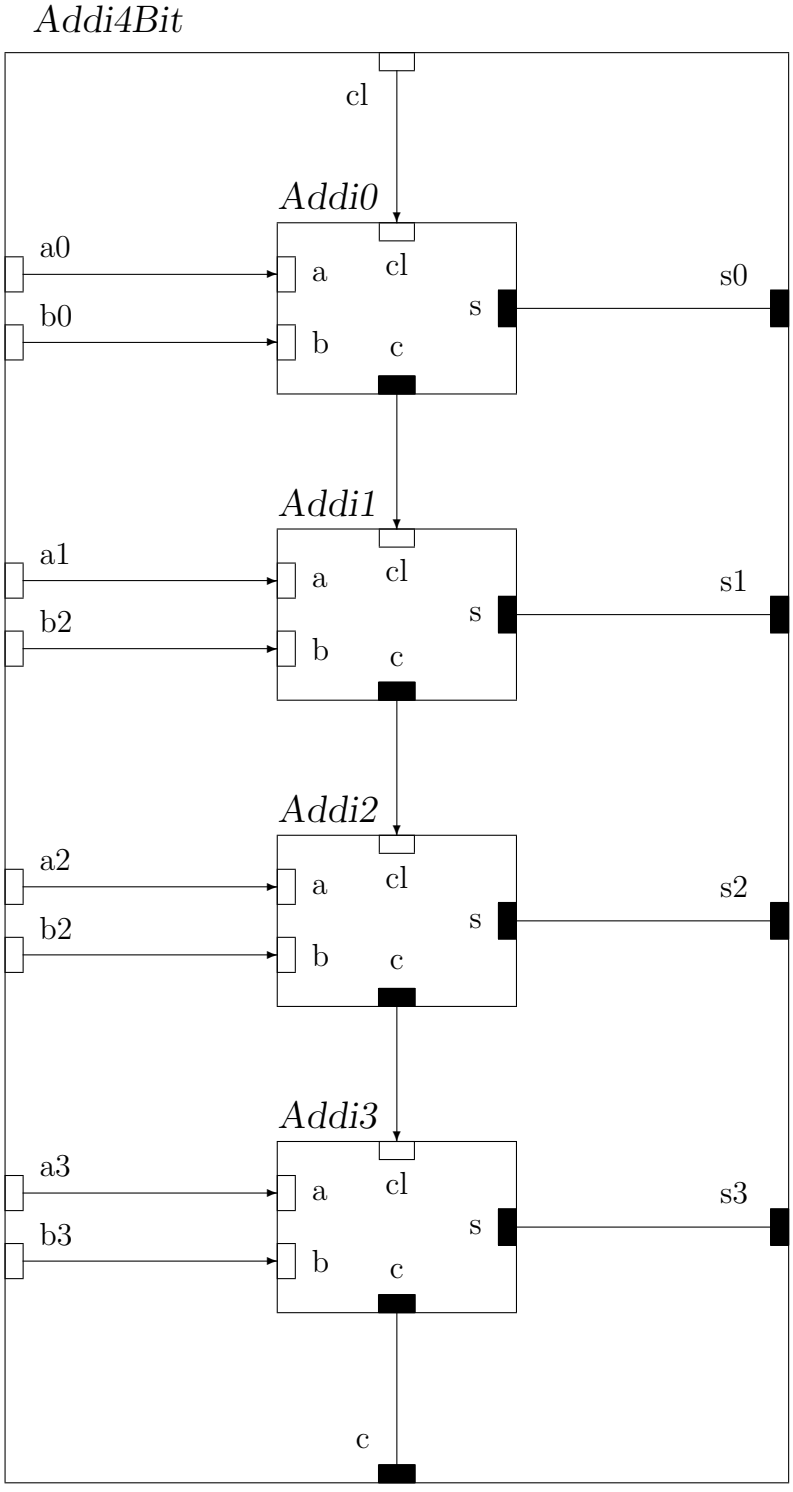



Abb. 7.4-3: Modellkomponente 4-Bit-Addierer

# 8 Zusammenfassung und Ausblick

Die vorliegende Arbeit beschreibt die Spezifikation und Implementierung der Modellbeschreibungssprache SIMPLEX-MDL mit Ausnahme einiger Erweiterungen, die in jüngster Zeit vorgenommen wurden.

Als Grundlage dieser Sprache wurde die systemtheoretische Betrachtungsweise dargestellt sowie die methodologischen Überlegungen, die zu dieser Sichtweise führen.

Der klassische systemtheoretische Ansatz liefert die Basis für eine komponentenweise Zerlegung eines Modells. Als Zustandsüberführungsfunktion wurde eine Funktion eingeführt, welche kontinuierliche mit diskreten Zustandsübergängen verbindet und damit die Modellierung kombinierter Modelle erlaubt.

Gegenüber anderen Simulationssprachen erscheint es uns wesentlich, daß diskrete Zustandsübergänge im Sinne der Automatentheorie interpretiert werden. Eine gegenseitige Beeinflussung diskreter Zustandsübergänge ist damit ausgeschlossen. Wir sind der Ansicht, daß sich nur auf diese Art die Wirkung gleichzeitig ausgeführter Ereignisse korrekt nachvollziehen läßt.

Mit der Festlegung der Zustandsüberführungsfunktion ergibt sich, mit welchen Mitteln Modelldynamik zu beschreiben ist:

- durch Differentialgleichungen
  (kontinuierlicher Anteil der Zustandsüberführungsfunktion)
- durch Ereignisse
  (diskreter Anteil der Zustandsüberführungsfunktion)
- durch algebraische Gleichungen zum Ausdrücken direkter Abhängigkeiten

Die weitere Aufgabe bestand nun darin, einen Simulator zu bauen, der Modellabläufe, so nachvollzieht, wie es die mathematische Modellspezifikation vorsieht.

Beim Entwurf eines Algorithmus für Digitalrechner besteht eine wesentliche Schwierigkeit darin, daß reelle Zahlen und kontinuierliche Zustandsübergänge nur approximativ nachgebildet werden können. Man kann daher dem

Benutzer eines Simulators gewisse Grundkenntnisse über die interne Arbeitsweise nicht ersparen.

Um kombinierte Zustandsübergänge oder diskrete Zustandsübergänge in Verbindung mit einer Zeitfortschaltung korrekt auszuführen, wurde eine zweidimensionale Zeitdarstellung eingeführt.

Eine Modellspezifikation, insbesondere bei einer Zerlegung des Modells in Komponenten, nimmt keine Rücksicht auf die Erfordernisse einer Abarbeitung durch einen sequentiellen Algorithmus. Es wurde dargestellt, welche Verfahren in Betracht kommen, um eine solche deklarative Beschreibung algorithmisch zu behandeln.

Die mathematische Notation zur Formulierung von Modellen wurde in eine für digitale Rechenanlagen lesbare Form gebracht. Das Resultat ist die Modellbeschreibungssprache SIMPLEX-MDL. Es wurde versucht, Syntax, Semantik und Pragmatik der Sprache so präzise wie möglich darzustellen.

Bei der Gestaltung der Sprache wurde größter Wert darauf gelegt, daß Modelle in einer gut lesbaren Form beschrieben werden und auch ohne genauere Kenntnis der Sprache zu verstehen sind. Wir hoffen, damit einen Beitrag geleistet zu haben, daß erstellte Modelle eine weitere Verarbeitung als bisher erfahren und in einem größeren Kreis von Wissenschaftlern diskutiert und weiterentwickelt werden.

Wir haben uns bemüht, daß implementierungstechnische Notwendigkeiten keinen Niederschlag im Sprachentwurf finden. Wie wir meinen, ist dies mit einer einzigen Ausnahme gelungen: Für eine effiziente Zeitfortschaltung bei rein diskreten Modellen mußte der Gebrauch der Simulationszeit in Zeitvergleichen etwas eingeschränkt werden.

Ein wesentliches Merkmal von SIMPLEX-MDL ist die Möglichkeit, Modelle modular aufzubauen, d.h. hierarchisch in Komponenten zu zerlegen. Im Gegensatz zu anderen am Markt verfügbaren Simulationssprachen können nicht nur Strukturen aus Grundbausteinen zusammengefügt werden, sondern die erforderlichen Grundbausteine auch vom Benutzer selbst definiert werden. Damit ist er in der Lage, ein für seine Anwendungen geeignetes Sortiment von Modellkomponenten zu einer sogenannten Modellbank zusammenzustellen.

Zur Implementation der Sprache wurde ein MDL-Compiler und ein Laufzeitsystem in der Programmiersprache C erstellt. Modelle, die in SIMPLEX-MDL formuliert wurden, werden zunächst vom MDL-Compiler nach C und darauf vom C-Compiler in Objektcode übersetzt. Der Objektcode der Modellkomponenten und das Laufzeitsystem werden zum fertigen Simulationsprogramm gebunden.

Modellerstellung und Modellablauf erfolgen unter der Kontrolle der Simulationsumgebung. Dadurch wird eine weitreichende Unterstützung beim Aufbau hierarchischer Modelle und bei der Ausführung von Simulationsexperimenten gewährleistet.

Aufbau und Implementierung des Simulationsprozesses wurden detailliert dargestellt. Nur so erschien es uns möglich, dem interessierten Leser ein tiefgreifendes Verständnis des Simulators zu ermöglichen. Insbesondere das Zusammenspiel der Ablaufsteuerung mit dem aus den Modellkomponenten generierten Code sollte damit deutlich werden.

Besonderes Augenmerk wurde auch auf die Effizienz der Abarbeitung gelegt. Durch die komponentenweise Zerlegung eines Modells fällt ein Mehraufwand an, der möglichst klein gehalten werden muß, um mit anderen Simulationssprachen konkurrieren zu können. Es wurde gezeigt, wo das theoretische Minimum an auszuführenden Operationen liegt und wie es implementierungstechnisch erreicht werden kann.

Es wurden bereits zahlreiche Anwendungen durchgeführt, um Erfahrungen mit dem implementierten System zu sammeln. Neben einer Reihe kleinerer Modelle, mit denen ein Proseminar über Modellbildung durchgeführt wird, entstanden im Rahmen von Hauptseminaren sowie Studien- und Diplomarbeiten umfangreiche Modelle aus unterschiedlichen Anwendungsbereichen wie etwa: Dynamik eines sensorgeführten Roboters, Sauerstoffversorgung in der Leber, neuronale Netze, Abwasserreinigung in Kläranlagen. Weiterhin wurden auch einige Modellbanken aufgebaut, wie etwa eine Modellbank mit Analogrechnerkomponenten und eine mit Digitalrechnerkomponenten.

Der in dieser Arbeit beschriebene Stand der Modellbeschreibungssprache deckt bereits einen relativ großen Anwendungsbereich ab. Dennoch sind wir weiterhin darum bemüht, die Sprache zu ergänzen und vervollständigen.

Im wesentlichen sind hier folgende Punkte zu nennen:

- prozedurale Darstellung von Funktionen
- Totzeitvariablen
- Reihungen (Felder) von Modellgrößen und Modellkomponenten
- Zufallsprozesse
- physikalische Maßeinheiten
- Inhaltsgrößen und bewegliche Komponenten
  (zur Modellierung von Warteschlangenmodellen)
- Indexmengen und Operationen über Indexmengen zur Beschreibung von Strategien

Ein Teil dieser Punkte wurde bereits begonnen und es liegen auch schon Konzepte und Implementierungen vor. Da diese Konzepte noch nicht ganz in Einklang mit der bisher verfolgten Philosophie stehen und daher noch der Weiterentwicklung bedürfen, wurde auf ihre Darstellung im Rahmen dieser Arbeit verzichtet.

Auch weiterhin sind wir darum bemüht, simulationstechnisches Know-how in Software umzusetzen, die bei ernsthaften Anwendern Verwendung findet. Wir sind der Überzeugung, daß sich neue Ideen und Konzepte nur auf dem Weg der praktischen Erprobung weiterentwickeln und weiterverbreiten lassen. Daher hoffen wir auf eine breite Resonanz vom Leser und Anwender und sind für positive wie negative ußerungen dankbar.

# 9 Literatur


/ACSL 87/ Mitchell and Gauthier Associates (Hrsg.):
Advanced Continuous Simulation Language (ACSL)
Reference Manual,
Mitchell and Gauthier Associates, Concord, Mass. 01742

/BORIS 85/ Siemens AG (Hrsg.):
Benutzerhandbuch, BORIS-Version 1.5

/Boss 85/ Bossel, H.:
Umweltdynamik,
te-wi Verlag, 1985
ISBN-3-921 803-36-5

/Brau 85/ H. Braun:
SIDAS II Benutzeranleitung Version 1.0,
Hrsg.: Institut für Meß- und Regelungstechnik,
Universität Karlsruhe, Januar 1985

/BroSen 84/ Bronstein, I.N.; Semendjajew, K.A.:
Taschenbuch der Mathematik,
Verlag Harri Deutsch, Thun und Frankfurt/Main 1984

/Cell 83/ Cellier, Francois:
Simulation Software: Today and Tomorrow
in: Simulation in Engineering Sciences,
IMACS Symposium International, Nantes 1983

/Cin 84/ Cin, M.D; Lutz, J.; Risse T.:
Programmierung in Modula-2
Teubner, Stuttgart 1984

/ConBo 72/ Conte, S.D.; DeBoor, Carl:
Elementary Numerical Analysis,
Mc Graw-Hill, Düsseldorf 1972

/Crae 85/ Craemer, Diether:
Mathematisches Modellieren dynamischer Vorgänge;
Eine Einführung in die Programmiersprache DYNAMO,
Teubner, Stuttgart, 1985

/CSSL 84/ Simulation Services (Hrsg.):
CSSL-IV Reference Manual,
Simulation Services, 20926 Germain Street,
Chatsworth, California 91311, USA

/EsSch 86/ Eschenbacher, P.; Schmidt, B.:
Empirische Modellbildung und formale Modellbeschreibung
in: Biethahn, J., Schmidt, B. (Hrsg.):
Simulation als betriebliche Entscheidungshilfe,
Fachberichte Simulation Bd. 6,
Springer, Berlin 1987

/Forr 73/ Forrester, Jay W.:
World Dynamics,
Wright-Allen Press, Cambridge, MA, 1973

/Göld 87/ Göldner, Klaus:
Mathematische Grundlagen der Systemanalyse
Verlag Harri Deutsch; Thun, Frankfurt/Main 1987

/Gött 82/ Göttlicher, Rainer:
Simulation zeitkontinuierlicher Modelle – Implementierung neuer numerischer Integrationsmethoden,
Diplomarbeit, angefertigt am IMMD IV der Universität Erlangen-Nürnberg 1982

/Grig 72/ Grigorieff, R. D.:
Numerik gewöhnlicher Differentialgleichungen,
Teubner Verlag, Stuttgart 1972

/JenWir 74/ Jensen, K.; Wirth, N.:
PASCAL User Manual and Report
Springer, New York 1974

/Joh 78/ Johnson, S.C.:
YACC: Yet Another Compiler-Compiler
in: B.W. Kernighan, M.D. Ilroys,
UNIX Programmer's Manual, Bell Laboratories, 1978,
7th edition

/Kaehl 86/ Kaehler, Ted; Patterson, David:
A Taste of Smalltalk
Norton, New York 1986

/KerRit 83/ Kernighan, B.; Ritchie, D.:
Programmieren in C,
Carl Hanser Verlag, München 1983

/Kett 83/ Kettenis, D.L.:
The COSMOS Modelling and Simulation Language
in: W. Ameling (Hrsg.),
Proceedings of the First European Simulation Congress 1983,
Springer, Berlin 1983, pp 251-260

/Kett 88/ Kettenis, D.L.:
The COSMOS Simulation Environment,
in: Simulation Environments,
Proceedings of the ESM, Nice 1988,
Hrsg.: R.C. Huntsinger, SCS International

/KöMa 72/ Köcher, D.; Matt, G.; Oertel, C.; Schneeweiß, H.:
Einführung in die Simulationstechnik,
Hrsg.: Deutsche Gesellschaft für Operations Research (DGOR), Beuth Vertrieb, Berlin 1972

/Lang 90/ Langer, Klaus-Jürgen:
Entwurf und Implementierung eines Simulationssystems mit integrierter Modellbank- und Experimentdatenverwaltung, Dissertation im IMMD IV der Universität Erlangen-Nürnberg

/LaSch 87/ Langer, K.-J.; Schmidt, B.:
The Simulation System SIMPLEX-II – A Model Construction And Experimentation Environment
in: Proceedings of the ESM, Wien 1987

/LeSch 78/ Lesk, M.E.; Schmidt, E.:
LEX: A Lexical Analyzer Generator
in: B.W. Kernighan, M.D. Ilroys,
UNIX Programmer's Manual, Bell Laboratories, 1978, 7th edition

/MarLed 87/ Marcotty, M.; Ledgard, H.:
The World of Programming Languages
Springer, Berlin 1987

/May 82/ Mayer, Otto:
Syntaxanalyse
Bibliographisches Institut Mannheim, 1982

/Ören 84/ Ören, Tuncer I.:
GEST – A Modelling and Simulation Language
Based on System Theoretic Concepts,
in: Simulation and Model-Based Methodologies:
An Integrative View, Hrsg.: T.I. Ören,
Nato ASI Series, vol. F10,
Springer, Berlin 1984, pp 281-334

/ÖrZi 79/ Ören, Tuncer; Ziegler, Bernard;
Concepts for advanced simulation methodologies
in: SIMULATION, Volume 32, No. 3, March 1979
pp. 69 - 82

/Peg 86/ Pedgen, C. Dennis:
Introduction to SIMAN,
Systems Modeling Corporation, Calder Square, PN, 1986

/Petsch 86/ Petsch, Sabine:
Teilimplementierung eines Übersetzers (Syntaxanalyse) für
die Modellbeschreibungssprache SIMPLEX-MDL,
Studienarbeit, angefertigt am IMMD IV der Universität Erlangen-
Nürnberg 1986

/Pich 75/ Pichler, Franz:
Mathematische Systemtheorie, Dynamische
Konstruktionen
Walter de Gruyter, Berlin 1975

/Prit 74/ Pritsker, A. Alan. B.:
The GASP IV Simulation Language,
John Wiley, New York, 1974

/Prit 79/ Pritsker, A. Alan B.; Pedgen C. D.:
Introduction to Simulation and SLAM,
John Wiley, New York 1979

/Prit 82/ Pritsker & Associates (Hrsg.):
A Comparison of three general purpose Simulation Languages:
SLAM, SIMSCRIPT and GPSS; Private Veröffentlichung:
P.O. Box 2413, West Lafayette, Indiana 47906;
Oktober 1982

/Prat 75/ Pratt, Terrence, W.:
Programming Languages – Design and Implementation
Prentice-Hall, London 1975

/Rohl 73/ Rohlfing, Helmut:
SIMULA,
Mannheim Bibliographisches Institut 1973

/Russ 83/ Russel, Edward C.:
Building Simulation Models with SIMSCRIPT II.5,
CACI, Los Angeles 1983

/SchFr 85/ Schreiner, A.T.; Friedman, H.G.:
Compiler bauen mit UNIX
Hanser Verlag, München 1985

/Schm 84a/ Schmidt, Bernd:
Der Simulator GPSS-FORTRAN Version 3,
Fachberichte Simulation Bd. 2,
Springer, Berlin 1984

/Schm 84b/ Schmidt, Bernd:
Modellbildung mit GPSS-FORTRAN Version 3,
Fachberichte Simulation Bd. 3,
Springer, Berlin 1984

/Schm 85/ Schmidt, Bernd:
Systemanalyse und Modellaufbau – Grundlagen der Simulationstechnik,
Fachberichte Simulation Bd. 1, Springer, Berlin 1985

/Schnei 75/ Schneider, Hans-Jürgen:
Compiler – Aufbau und Arbeitsweise
Walter de Gruyter, Berlin 1975

/Schri 74/ Schriber, Thomas J.
Simulation Using GPSS,
John Wiley, New York 1974

/Schür 77/ Schürmann, Hans Werner:
Theoriebildung und Modellbildung
Akademische Verlagsgesellschaft, Wiesbaden 1977

/Schü 84/ Schüßler, H.W.:
Netzwerke, Signale und Systeme; Band II
Springer Verlag, Berlin 1984

/Schwa 84/ Schwald, Andreas:
ADA – Eine Einführung
Bibliographisches Institut, Mannheim 1984

/Sol 85/ Solar, D. (Hrsg.):
HYBSYS – User Manual, Release 1.1, Dez. 85
Hrsg.: Technische Universität Wien;
EDV-Zentrum, Abt. Hybridrechenanlage,
Gußhausstr. 27 - 29, A-1040 Wien

/StoBu 83/ Stoer, J.; Bulirsch, R.:
Einführung in die numerische Mathematik,
Springer, Berlin 1983

/Strou 87/ Stroustrup, Bjarne:
The C++ Programming Language
Addison Wesley, Reading, MA, 1987

/SYSMOD 86/ System Designers (Hrsg.):
SYSMOD User Manual, Release 1.0, April 1986
SYSMOD Language Definition, Release 1.0, Juni 1986
Systems Designers Scientific, Ferneberga House,
Alexandra Road,
Farnborough, Hampshire, United Kingdom

/TieSche 78/ Tietze, U.; Schenk, Ch.:
Halbleiter – Schaltungstechnik, 4. Auflage
Springer, Berlin 1978

/TreSor 85/ Tremblay, J.-P.; Sorenson, P.G.:
The Theory and Practice of Compiler Writing,
Mc Graw-Hill, Hamburg 1985

/Unbe 80/ Unbehauen, Rolf:
Systemtheorie – Eine Darstellung für Ingenieure
Oldenbourg Verlag, München Wien 1980

/WaiGo 85/ Waite, W.M.; Goos, G.:
Compiler Construction,
Springer, Berlin 1985

/Weig 84/ Weigel, Roland:
Numerische Integrationsverfahren für die Simulation zeitkontinuierlicher Systeme (Implementierung und Vergleich),
Diplomarbeit, angefertigt am IMMD IV der Universität Erlangen-Nürnberg 1984

/WerArn 86/ Werner, H., Arndt, H.:
Gewöhnliche Differentialgleichungen
Springer Verlag, Berlin 1986

/Wun 86/ Wunsch, Gerhard:
Handbuch der Systemtheorie
Oldenbourg Verlag, München Wien 1986

/ZeiElz 79/ Ziegler, B.; Elzas, M.; Klir, Ören, T. (Hrsg.):
Methodology in Systems Modelling and Simulation,
North Holland, Amsterdam 1979