Diss. ETH No. 12294

Komponentenorientierte Modellierung und Simulation kombinierter Systeme

Abhandlung zur Erlangung des Titels
Doktor der Technischen Wissenschaften der
EIDGENÖSSISCHEN TECHNISCHEN HOCHSCHULE
Zürich

vorgelegt von
Markus P. Kottmann
Dipl. El.-Ing. ETH
geboren am 5. 4. 1964
Bürger von Luzern,

angenommen auf Antrag von
Prof. Dr. W. Schaufelberger, Referent
Prof. Dr. J. Gutknecht, Korreferent

1997

ISBN 3-906487-02-4
IfA Publication No. 9
http://www.edu-net.ethz.ch/COSIMO/COSIMO.html

Vorwort

Die vorliegende Arbeit entstand während meiner Arbeit am Institut für Automatik (IfA) der Eidgenössischen Technischen Hochschule (ETH) Zürich. Ich bedanke mich bei Prof. W. Schaufelberger für seine Unterstützung und für die Freiheit, die er mir in der Forschung gewährte. Weiterer Dank gebührt Prof. J. Gutknecht für die Übernahme des Korreferats.

In meine Arbeit eingeflossen sind Beiträge von verschiedener Seite:

- das Beispiel der Anästhesiesimulation geht auf die Initiative von Marco Derighetti zurück,
- die Eisenbahnsteuerung ist mithilfe von Hannes Wichser realisiert worden,
- Franta Kraus und Urban Brunner sind exzellente Kenner der Untiefen der Regelungstechnik,
- Martin Wirth, Daniel Keller, Xiaobing Qiu, David Farruggio und Martin Rickli waren immer für Diskussionen über alle un-/möglichen Aspekte von Software zu haben,
- June Hörrmann-Clark hätte gerne auch das Vorwort korekturgelesen.

Bei vielen Leuten, die mich während meiner Dissertationszeit begleiteten, möchte ich mich für ihre Freundschaft bedanken; neben anderen sind dies:

- Guillermo Peretti, der die Produktion von Tortillas nicht nur simuliert,
- Esther Hagenow, deren Theke jederzeit Quell köstlichster Palindrome war,
- Christopher Ganz, der auf alle Fragen die passende Antwort hatte[1],
- Peter Kolb, Jacques Chapuis und Christoph Eck, mit denen ich mir Nächte laufend um die Ohren schlug.

Ein spezieller Dank gebührt aber meinen Eltern, meinen Geschwistern und meiner Frau Daniela; ohne ihre Unterstützung wäre diese Arbeit nicht möglich gewesen.

---

[1] 42

Inhalt

Abstract

The presented work deals with modelling and simulation of complex systems, especially systems of a hybrid nature combining continuous-time, discrete-time and discrete-event elements. Examples of such systems are traffic systems or production plants like smelting furnaces or paper mills. The main difficulty in developing a simulator for combined systems is the successful synthesis of two programming paradigms into one package. On one hand there is the world of differential equations where solutions are normally calculated using numerical approximations. Numerical integration routines are usually programmed in a procedural way due to their algorithmic character. On the other hand there are (discrete-) event-oriented systems where the need for simulation has essentially contributed to the development of object-oriented programming and simulation languages. The main accent in the connection of procedural integration routines with object-oriented concepts lies in an efficient and precise detection of state-events and in a highly flexible propagation of these events.

Todays programs for the simulation of dynamic systems are monolithic packages whose extensibility lies within narrow borders. Usually there exists a block library which makes a limited number of different block types available. These blocks may be combined using an editor; the single blocks are configured by choosing parameters or by using a script-language. Significant changes or extensions of the functionality can normally only be expected by waiting for a new release of the whole package - the user cannot perform changes behind the scenery.

The modelling and simulation environment COSIMO is based on a configurable system (Oberon System 3) which allows the flexible composition of single elements (Gadgets). This enables extensions in various directions such as changes of the simulation behaviour of modell components or new ways of visualization and animation. Beyond it, the document-oriented user interface allows the embedding of simulation scenarios into different media such as hypertext or online-documents under conservation of its full functionality.

A concept which goes beyond the pure simulation is running different scenarios in the same environment. Possible scenarios are repeated simulation runs with varied parameters, optimization of models, testing of model components in an experimental frame, analysis of models, control of hardware in the loop.

The user of COSIMO is able - depending on his needs and his programming skills - to make use of a broad palette of supplements and extensions without having to compile the existing modules. On one hand there are the possibilities of graphical programming, the composition of new models and graphical interfaces. On the other hand textual programming allows the development and the integration of new model components, integration routines, editors and scenarios.

Kurzfassung

Die vorliegende Arbeit befasst sich mit der Modellierung und Simulation komplexer Systeme, im speziellen solchen mit einem kombiniert zeitkontinuierlichen, zeitdiskreten und ereignisorientierten Charakter. Beispiele dafür sind Verkehrssysteme oder Produktionsanlagen. Die Schwierigkeit, einen Simulator für kombinierte Systeme zu entwerfen, liegt darin, aus zwei verschiedenen Programmierstilen eine Einheit zu synthetisieren. Auf der einen Seite liegt die Welt der Differentialgleichungen, in der Lösungen im allgemeinen numerisch durch Integrationsroutinen approximiert werden. Auf der anderen Seite stehen ereignisorientierte Systeme, bei denen das Bedürfnis nach Simulationen wesentlich dazu beigetragen hat, dass objektorientierte Programmier- und Simulationssprachen überhaupt entwickelt worden sind. Das Hauptgewicht bei der Verknüpfung von Integrationsroutinen mit objektorientierten Konzepten liegt in einer effizienten und genauen Detektion von Zustandsereignissen, - Ereignissen, die durch das Erreichen bestimmter kontinuierlicher Zustände ausgelöst werden -, sowie grosser Flexibilität bei der Propagierung von Ereignissen.

In den heute gängigen Simulationsprogrammen ist es üblich, monolithische Pakete zu schnüren, deren Erweiterbarkeit sich in engen Grenzen hält. So können beispielsweise in einem Editor Blöcke verwendet werden, wobei die einzelnen Blöcke aus einer Bibliothek mit einer limitierten Anzahl verschiedener Blocktypen stammen. Das Konfigurieren einzelner Blöcke erfolgt durch Anpassen von Parametern und die Eingabe von Skripts. Signifikant veränderte oder erweiterte Funktionalität kann der Benutzer im Normalfall nur von einer neuen Version des Programms erwarten.

Die Modellierungs- und Simulationsumgebung COSIMO baut auf einem konfigurierbaren System (Oberon System 3) auf, das es erlaubt, einzelne Komponenten (Gadgets) flexibel zusammenzustellen. So sind zur Laufzeit des Programms verschiedene Erweiterungen möglich, z.B. bezüglich des Simulationsverhaltens von Teilmodellen, der Visualisierung oder der Animation. Darüber hinaus ergibt sich durch die dokumentenbasierte Benutzeroberfläche eine natürliche Einbettung in verschiedenartige Umgebungen wie Internetseiten oder Hypertextdokumente.

Ein Konzept, das über die reine Simulation hinausreicht, besteht darin, in derselben Umgebung verschiedene Szenarien ablaufen zu lassen. Mögliche Szenarien sind z.B. wiederholte Simulationsläufe mit variierenden Parametern, das Optimieren von Modellen, das Austesten von Teilmodellen in einem Experimentalrahmen, die Analyse von Modellen, Regelung und Steuerung.

Der Benutzer von COSIMO kann je nach Bedürfnissen und Programmierkenntnissen eine breite Palette von Ergänzungs- und Erweiterungsmöglichkeiten nutzen, ohne dass eine Neukompilation des bereits existierenden nötig ist. Einerseits bestehen Möglichkeiten der graphischen Komposition neuer Modelle und Oberflächen, andererseits können durch textuelle Programmierung Teilmodelle, Integrationsalgorithmen, Editoren und Szenarien an verschiedene Bedürfnisse angepasst werden.

Einleitung

1

1.1. Synthese und Analyse in der Automatik

Der Entwurf von Regelungen und Steuerungen ist ein iterativer Prozess. Ausgehend von einem realen System wird versucht, über verschiedene Zwischenschritte ein

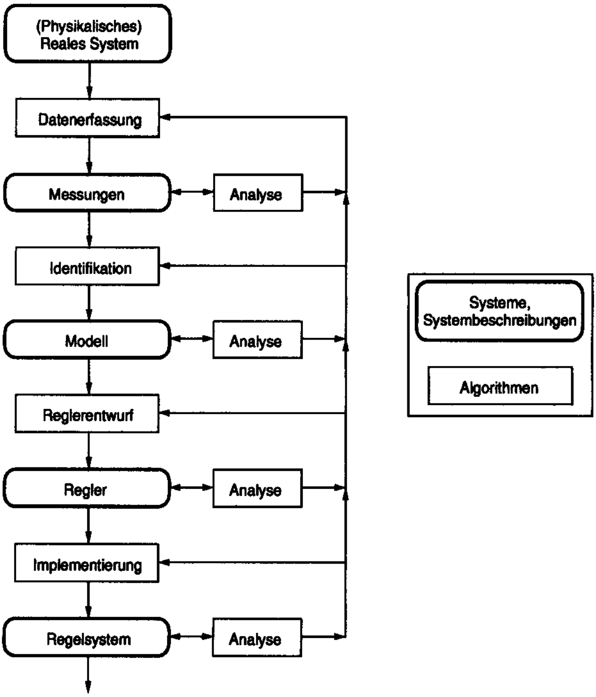


Fig. 1.1: Entwurfszyklus für Regelsysteme

Regelsystem zu entwerfen, das bestimmte Spezifikationen erfüllt (Fig. 1.1). Durch Analyse und Bewertung kann entschieden werden, ob eine Zwischenstufe akzeptiert wird oder ob Anlass besteht, einen oder mehrere Schritte zu wiederholen. Die Algorithmen und Methoden, die bei den Entwurfsschritten (*Synthese*) und der *Analyse* eingesetzt werden, können grob eingeteilt werden in klassische Methoden wie sie ab Anfang des 20. Jahrhunderts entstanden sind und moderne Methoden wie sie seit den sechziger Jahren entwickelt wurden. Vertreter des ersten Typs sind beispielsweise Bodediagramme oder Wurzelortskurven. Sie beruhen darauf, dass die zu behandelnden Modelle einfach sind - üblicherweise linear und zeitinvariant - und können zumindest approximativ mit Papier und Bleistift angewendet werden. Die Entwicklung neuerer Methoden wie z.B. des Kalmanfilters, der dynamischen Programmierung oder von Model Predictive Control (MPC) ist durch verschiedene Faktoren beschleunigt worden. Zum einen ist dies die Notwendigkeit, auch komplexere Strekken regeln zu können, zum anderen ist eine ganze Reihe der modernen Techniken gar nicht denkbar ohne die Rechenleistung heutiger und künftiger Generationen von Computern, die in der Lage sind, die entsprechenden Algorithmen innerhalb einer angemessenen Zeit abzuarbeiten.

Komplexe Systeme zeichnen sich durch eine Kombination von Eigenschaften und Effekten wie

- viele Zustände, Ein- und Ausgänge
- Nichtlinearitäten
- zeitvariante Koeffizienten / Struktur
- Unsicherheiten in Koeffizienten / Struktur
- Komponenten mit uneinheitlicher Modellierung

aus. Mit der komplexeren Darstellung der Systeme geht auch die Formulierung neuer Spezifikationskriterien für Regelsysteme einher, wie Optimalität oder robuste Stabilität, die häufig nur mit grossem numerischen Aufwand in entsprechende Regler umgesetzt werden können.

Etwas pauschalierend können die modernen Methoden in zwei Gruppen aufgeteilt werden: analytische und numerische. *Analytisch* in diesem Zusammenhang wird gebraucht im mathematischen Sinne geschlossener Lösungen, wogegen *numerisch* für einen Ansatz steht, bei dem durch 'brute force' ein Problem auf dem Computer gelöst wird. Die analytischen Methoden vermitteln eine tiefere mathematische Einsicht in spezielle Klassen von Systemen, die numerischen sind auf allgemeine Systeme besser anwendbar, liefern aber Aussagen mit lokal beschränkter Gültigkeit.

Ein Beispiel auf der eleganten mathematischen Seite ist der Stabilitätstest von Kharitonov, bei dem durch den (positiven) Test von vier Extrempolynomen eine Aussage über ein Intervallpolynom, also eine ganze Familie von Polynomen gemacht werden kann. Auf der rechenintensiven Seite kann die dynamische Programmierung angeführt werden, die auf dem mathematisch eher simplen Optimalitätsprinzip beruht. Es existieren Zwischenstufen, z.B. Verfahren, wie Branch & Bound, durch das analytisch fundierte Abkürzungen innerhalb der dynamischen Programmierung möglich werden, oder Kollektionen von Werkzeugen (Toolboxes), die numerische Unterstützung für verschiedene Zweige der Theorie bieten.

Fig. 1.2 zeigt eine qualitative Plazierung verschiedener Methoden. Der in Graustufen angetönte Komplexitätsgrad der behandelbaren Systemklassen soll verdeutlichen, dass mit rechenintensiven numerischen Methoden relativ erfolgreich auch Probleme behandelt werden können, für die es extrem schwierig wird, ein theoretisches Framework zu erstellen.

Eine mächtige numerische Methode ist diejenige der Simulation. Die Plazierung der Simulation in Fig. 1.2 bezieht sich auf ihre Anwendung; sie soll nicht darüber hinwegtäuschen, dass die Entwicklung von Simulationssprachen und Simulatoren eng mit der analytischen Systemtheorie verflochten ist. Numerisch robuste und effiziente Simulationsprogramme wurzeln tief in der Systemtheorie. Simulation und Systemtheorie haben sich in vielen Gebieten gegenseitig beeinflusst:

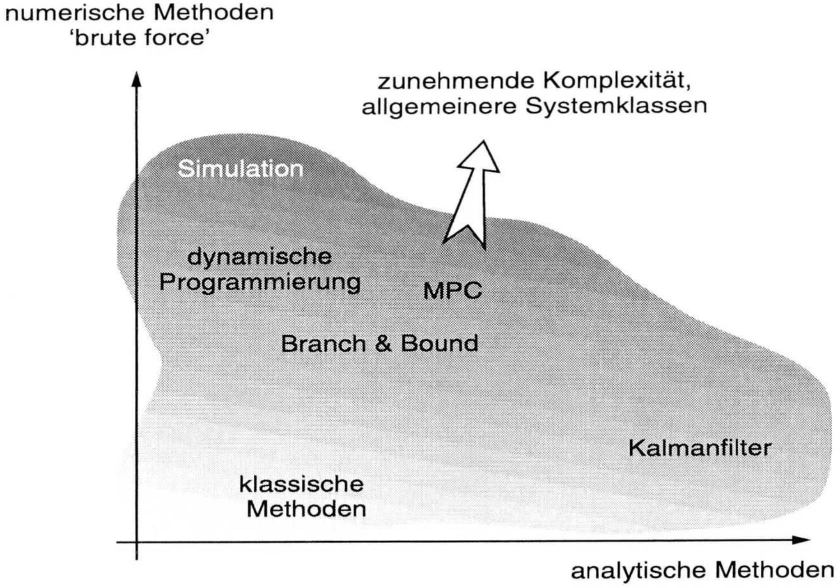


Fig. 1.2: Methoden der Analyse und Synthese

- Die Simulation steifer Systeme ist eng verbunden mit der systemtheoretischen simultanen Betrachtung verschiedener Zeitskalen und auch mit den *Averaging*-Methoden im Bereich der adaptiven Regelung [66].

- Die Chaostheorie erhielt in den siebziger Jahren neue Impulse durch unerwartete Simulationsresultate bei Wetterprognosen [42], nachdem sie theoretisch bereits im 19. Jahrhundert durch Poincaré [57], [58] vorgespurt worden war.

1.2. Wozu Simulation?

Simulation ist eine wichtige Analyse- wie auch Synthesemethode in der Automatik. Historisch gesehen stammt sie aus den Anfangszeiten der Computer. Auf Analogrechnern wurden bereits Differentialgleichungen implementiert, auf Hybridrechnern erste kombinierte Modelle. Die Simulation ereignisorientierter Systeme schliesslich hat die Entwicklung objektorientierter Programmiersprachen wie Simula 67 oder Smalltalk entscheidend vorangetrieben.

Die Relevanz der Simulation ist weiter zunehmend, da die Komplexität der zu regelnden und steuernden Prozesse weiter steigt. Theorie als Alternative zur Simulation gestaltet sich zunehmend schwierig; Resultate sind häufig praktisch nicht umsetzbar, weil sie nur unter sehr einschränkenden Bedingungen gültig oder in ihrer Aussage zu konservativ sind. Auch in Fällen, in denen primär durch Anwendung der Theorie ein Regler entworfen werden kann, stellen sich bei der Umsetzung in einen praktisch ver-

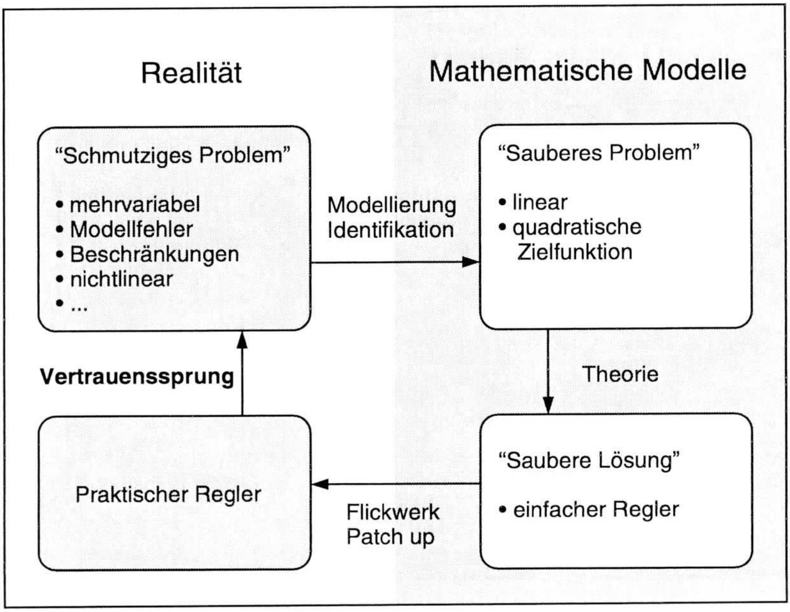


Fig. 1.3: *Vertrauenssprung bei Inbetriebnahme eines Reglers*

wendbaren Regler bis zur Inbetriebnahme viele Probleme. Simulationen können zu deren Lösung wesentlich beitragen und so den in Fig. 1.3 skizzierten Vertrauenssprung vermindern helfen.

Das Experiment als weitere Alternative zur Simulation ist unter Umständen zu teuer, zu gefährlich oder zu wenig flexibel, wenn es darum geht, verschiedene Konfigurationen zu testen. Darüberhinaus gibt es bei der Simulation die Möglichkeit, eine geeignete Zeitskala zu wählen, während ein Experiment üblicherweise in einer typischen Geschwindigkeit abläuft.

Eine Weiterführung der Ideen, die im Hybridrechner bereits vorgezeichnet sind und die eher wieder in Richtung des Experiments geht, bilden Simulationskonfigurationen mit *hardware in the loop*. Dabei steht die Möglichkeit im Zentrum, nach Bedarf Teile des realen Prozesses, die schwierig oder aufwendig zu modellieren sind, an eine Echtzeitsimulation anzukoppeln. Fig. 1.4 zeigt Varianten, Elemente der realen Welt in die Simulation zu integrieren. Neben Komponenten des Prozesses können auch externe,

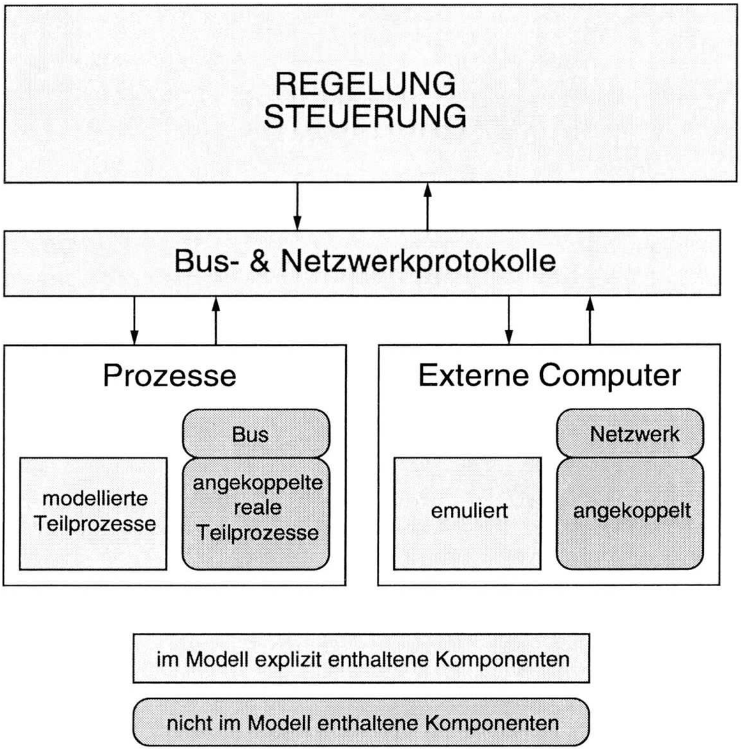


Fig. 1.4: Simulation mit 'hardware-in-the-loop'

z.B. übergeordnete Computer, entweder mitmodelliert werden oder als hardware in the loop betrachtet werden. Die eingezeichnete Zwischenschicht der Busse und Netzwerke gewinnt heute zusehends an Bedeutung. Gerade durch Verwendung standardisierter Protokolle kann eine hohe Flexibilität und Effizienz beim Konfigurieren solcher Simulationssysteme erreicht werden.

1.3. Motivation für COSIMO

Der Ursprung des Projekts COSIMO (COmbined SImulation & MOdelling) gründet in verschiedenen Anforderungen, die an eine moderne Simulationssoftware gestellt werden. Einerseits existiert bei vielen Systemen der Bedarf, beim Modellierungsprozess und bei der Simulation verschiedene zeitliche Strukturen sowie Kombinationen dieser Strukturen einzusetzen. Daneben besteht der Wunsch, neben textueller Programmierung auch eine flexible und erweiterbare graphische Benutzeroberfläche zur Verfügung zu haben, die auf spezielle Bedürfnisse wie interaktive Simulation, Visualisierung und Animation von Simulationsresultaten usw. zugeschnitten werden kann. Im weiteren sollen im Hinblick auf den Unterricht mit Studenten verschiedene Kriterien erfüllt werden wie: einfaches Erstellen von interaktiven Lernprogrammen, freie Erhältlichkeit, Verfügbarkeit der Quellcodes, möglichst problemloses Portieren zwischen gängigen Plattformen wie IBM-PC oder Apple Macintosh.

Im kommerziellen Bereich existiert eine Vielzahl von Simulationspaketen, die kombinierte Simulation in der einen oder anderen Form ermöglichen. Unser Bedürfnis, eine Synthese aus numerischer Integration, Detektion von Zustandsereignissen[1] und objektorientierter Ereignispropagierung, wurde von keinem der Pakete hinreichend abgedeckt. Im akademischen Bereich erfüllte OMOLA/OMSIM diesen Punkt, ist aber wenig flexibel, wenn strukturelle Änderungen, z.B. der Systemordnung, notwendig sind.

Eine detaillierte Diskussion verschiedener bestehender Programme findet sich in Kapitel 2.5.; dabei werden neben den Aspekten der eigentlichen Simulation vor allem auch die Benutzeroberfläche besprochen sowie die verwendete Programmiersprache, bzw. die Möglichkeiten, die in ihr implizit enthalten sind.

Die Programmiersprache Oberon als hybride Sprache - prozedural mit zusätzlichen objektorientierten Elementen - ermöglicht die Verschmelzung der verschiedenen Simulationsparadigmen zu einem kombinierten Simulator. Die Verwendung von Oberon System 3 erleichtert die graphische Definition von Modellen, da ein Editor für Blockdiagramme und Netzwerke benutzt werden kann, der auf der komponentenorientierten Klassenbibliothek der Gadgets basiert. Die dokumentenzentrierte Grund-

---

[1] Als Zustandsereignisse werden Ereignisse bezeichnet, die durch das Erreichen bestimmter Punkte im Zustandsraum ausgelöst werden. Sobald Zustandsereignisse diejenigen Teile des Zustandes betreffen, die als kontinuierliche Grössen formuliert sind, wird die korrekte Lokalisierung von Zeitpunkten, in denen Zustandsereignisse auftreten, nichttrivial.

philosophie der Oberfläche unterstützt auf natürliche Art auch Erweiterungen, die im ursprünglichen Projekt nicht explizit geplant sind. Beispiele dafür sind die Integration des Szenarios der Simulation in alternative Umgebungen wie Hypertext, aber auch die Möglichkeit, in derselben Umgebung alternative Szenarien wie z.B. eine Analyse oder eine Steuerung sich abspielen zu lassen.

1.4. Beitrag und Organisation

Die vorliegende Arbeit beschreibt die Konzepte hinter dem numerischen Kern des Simulators, den Einbezug der graphischen Oberfläche und die Möglichkeiten die bestehen, um das Paket zu erweitern. Neue Simulationstechniken sind insbesondere bei der Verwendung einschrittiger Integrationsroutinen mit Interpolationsformeln zur Detektion von Zustandsereignissen zu finden, bei der Kombination von Integration mit objektorientierter Ereignispropagierung sowie bei der Verwaltung der Information, die während der Simulation anfällt. Gegenüber diesen Aspekten, die leicht in einer Vielzahl von Sprachen implementiert werden können, ist der Aufbau des Editors, des Simulationsszenarios und alternativer Szenarien eine konsequente Weiterführung der Paradigmen wie sie durch die Benutzeroberfläche von Oberon [64], [74] und durch das Gadgets System [44] vorgegeben sind. Darunter ist vor allem die baukastenartige Komponentenarchitektur zu verstehen, aber auch die Möglichkeiten, die sich dem Benutzer durch Programmierung auf verschiedenen Ebenen, z.B. auf der textuellen oder der graphischen, bieten.

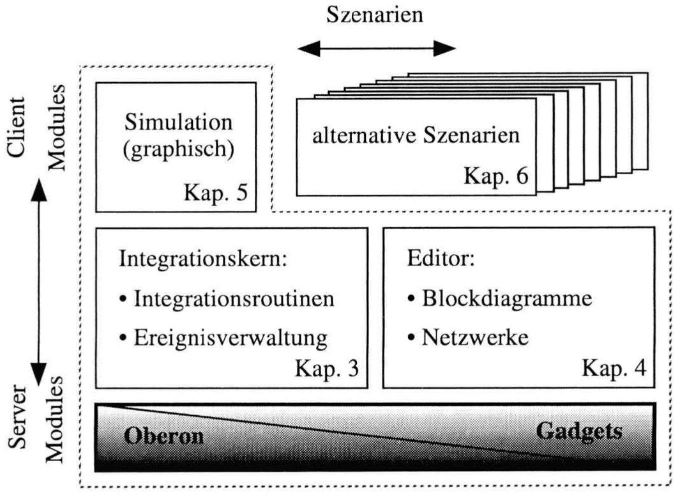


Fig. 1.5: Modulhierarchie von COSIMO

Die Modulhierarchie des Simulators ist recht umfangreich. In Fig. 1.5 wird die Struktur grob durch die wesentlichen Modulgruppen charakterisiert; die verschiedenen Modulgruppen werden in den entsprechenden Kapiteln genauer spezifiziert. Dadurch, dass kompilierte Module dynamisch geladen und gelinkt werden können, ist die Modulhierarchie nicht statisch abgeschlossen, sondern kann zur Laufzeit gegen oben beliebig erweitert werden.

In Kapitel 2 wird die Modellierung und Simulation aus systemtheoretischer Sicht beleuchtet und die verschiedenen Simulationstypen, die auf die zeitliche Entwicklung von Zuständen abzielen, gezeigt. Beispiele von kombinierten Modellen sowie Hybride Automaten zu ihrer Darstellung werden präsentiert. Verschiedene bestehende Simulationsprogramme werden neben anderen Kriterien danach charakterisiert, wie sie die kombinierte Simulation unterstützen.

In Kapitel 3 werden die verschiedenen Phasen der kombinierten Simulation dargestellt. Es wird gezeigt, wie Elemente der prozeduralen und objektorientierten Programmierung zu einem Integrationskern zusammengefügt werden, der die kombinierte Simulation unterstützt. Insbesondere wird Gewicht gelegt auf die Verwendung von Integrationsroutinen mit Interpolationsmethoden sowie das Konzept der Ereignispropagierung. Anhand eines Beispiels werden einige Pakete verglichen.

Kapitel 4 stellt einige Grundkonzepte von Oberon System 3 vor, einer dokumentenzentrierten Umgebung, die eine flexible Integration verschiedenster Komponenten unterstützt. Die Erweiterung dieses Systems um Editoren für Blockdiagramme und Netzwerke wird diskutiert. Es wird gezeigt, wie beim Modellieren Hierarchien unterstützt werden und welche Mechanismen z.B. für die Simulation nutzbar sind, um alle Modellkomponenten anzusprechen.

Die Simulation graphisch komponierter Modelle - als ein mögliches Szenario - ist der zentrale Punkt in Kapitel 5. Die Programmierung neuer Modellkomponenten wird dargestellt bezüglich ihrer graphischen Darstellung, ihrer Integration in den Editor (als Knoten bzw. als Block) und ihres Simulationsverhaltens. Die verschiedenen Modi der Simulation, vor allem die unterschiedlichen Grade der Interaktionsfähigkeit, werden beleuchtet. Anhand von Beispielen werden die Möglichkeiten von COSIMO dargestellt.

Kapitel 6 führt in das Konzept des Szenarios ein: eine zentrale Instanz prägt dem komponierten Modell ein gewisses Verhaltensmuster auf, innerhalb dessen die einzelnen Komponenten oder Akteure ihre spezifische Rolle spielen. Durch Auswechseln dieser Instanz lässt sich erreichen, dass sich in derselben Umgebung andere Szenarien abspielen. Diese reichen von orchestrierten Simulationen über Optimierungen bis zu Analyse, Regelung und Steuerung. Je nachdem, ob sich ein Szenario vollständig auf bereits implementierte Protokolle und Komponenten abstützt, können diese übernommen oder durch Erweiterungen ergänzt werden.

Kapitel 7 fasst die Konzepte zusammen, die in COSIMO realisiert wurden. Die verschiedenen Richtungen, die dem Benutzer zu Erweiterungen offenstehen, werden dargestellt. Es wird ein Ausblick gegeben, welche wünschenswerten ergänzenden Fähigkeiten und Konzepte in COSIMO integriert werden könnten.

Anhang A stellt einschrittige Runge-Kutta-Integrationsalgorithmen mit Interpolationsformeln vor. Dieser Typ ist speziell geeignet für die Aufgabe der Simulation kombinierter Modelle; in das Simulationspaket können auch andere Algorithmen integriert werden.

In Anhang B sind die detaillierten Resultate zum numerischen Beispiel aus Kapitel 3 zusammengestellt. Der Vergleich der Resultate bezieht sich auf Genauigkeit und Effizienz, wobei als Mass der Effizienz die Anzahl durchgeführter - sowohl akzeptierte wie auch abgelehnte - Integrationsschritte verwendet wird.

Anhang C enthält eine Liste der wichtigsten Prinzipien objektorientierter Programmierung. Den Begriffen der klassisch objektorientierten Terminologie werden die korrespondierenden Ausdrücke in der hybriden Sprache Oberon gegenübergestellt.

Eine Liste der Module von COSIMO, ihre Grösse und andere Kenndaten, ist in Anhang D enthalten. Ebenfalls werden die Beispiele zusammengefasst, die sich in den Kapiteln 5 und 6 finden.

Die Codefragmente, die sich in den Kapiteln 3-6 finden, sind - abgesehen von Kürzungen - exakte Wiedergaben aus dem Quellcode. Insbesondere sind darin die Sterne (*), die den Export von Konstanten, Typen, Variablen und Prozeduren über die Modulgrenze hinaus bewirken[1], enthalten. Anhang E enthält eine Liste der Schnittstellen aller wesentlichen Module.

Im Internet findet sich ein Einstieg zu COSIMO - zumindest bis zum Ende des laufenden Jahrtausends - unter http://www.edu-net.ethz.ch/COSIMO/COSIMO.html.

---

[1] Während beispielsweise in Modula-2 ein separates Schreiben von Quellcode (Implementation Module) und Schnittstelle (Definition Module) nötig ist, wird in Oberon die Beschreibung der Schnittstelle aus dem Quellcode automatisch generiert.

Modelle und Simulatoren

2

2.1. Klassifikation von Modellen

Nach Niemeyer [50] sind Modelle "... materielle oder immaterielle Strukturen, die andere Systeme so darstellen, dass eine experimentelle Manipulation der abgebildeten Strukturen und Zustände möglich ist." Die bei der Modellbildung stattfindende Abstraktion vereinfacht, bildet relevante Eigenschaften des Systems nach und lässt unwesentliches weg. Die Trennlinie zwischen dem Relevanten und dem Unwesentlichen ist allerdings im allgemeinen nicht a priori gegeben. Sie kann sich je nach Anwendung verschieben.

Page [52] klassifiziert Modelle nach dem Medium der Abbildung und der verwendeten Untersuchungsmethode:

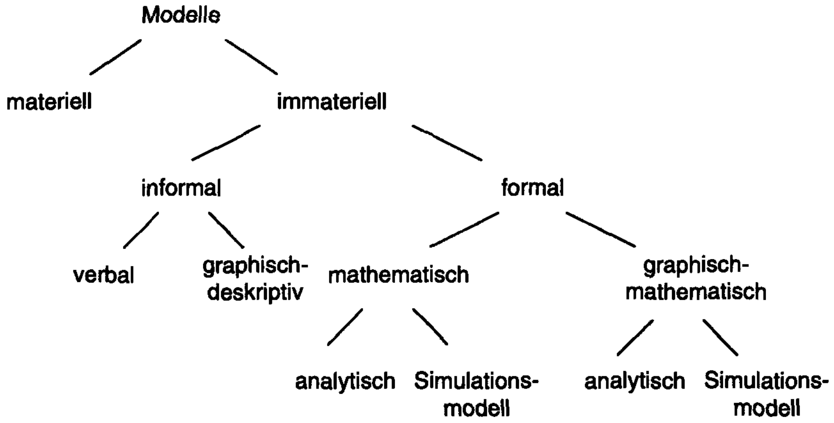


Fig. 2.1: *Klassifikation nach Abbildungsmedium und Untersuchungsmethode*

Beispiele sind:

- materiell: eine Modelleisenbahn
- verbal: eine umgangssprachliche Beschreibung
- graphisch-deskriptiv: ein grobes Blockdiagramm
- mathematisch: ein Satz von Differentialgleichungen
- graphisch-mathematisch: ein detailliertes Blockdiagramm

Während des Modellierungsprozesses werden verschiedene Stadien durchlaufen: beispielsweise vom verbalen Modell über ein graphisch-deskriptives zum graphisch-mathematischen Simulationsmodell.

Die vorliegenden Arbeit ist fokussiert auf quantitative Simulation von Modellen; im folgenden beschränkt sich der Begriff des Modells deshalb auf mathematische und graphisch-mathematische Modelle.

Eine weiterer Aspekt, unter dem die Klassifikation von Modellen möglich ist, ist der des Verwendungszwecks. Pritsker [62] listet folgende Zwecke auf:

- als Erklärungsmodell
- als Prognosemodell
- als Gestaltungsmodell
- als Optimierungsmodell

Beim Erklärungsmodell steht das Bedürfnis im Vordergrund, Erkenntnisse über ein Originalsystem zu gewinnen. Da durch die gängigen mathematischen Modelle die Eigenschaften bestimmter Systeme nicht immer befriedigend repräsentieren werden, hat die Suche nach Erklärungsmodellen grossen Einfluss auf die Systemtheorie. Physikalische Gebiete wie die klassische Mechanik, Thermodynamik oder Quantenmechanik haben die Entwicklung und die Formulierung mathematischer Modelltypen wie z.B. gewöhnlicher oder partieller Differentialgleichungen oder stochastischer Prozesse wesentlich motiviert.

Prognosemodelle dienen dazu, ausgehend von einem aktuellen Zustand, den weiteren Verlauf bestimmter interessierender Grössen abzuschätzen. Die genaue innere Struktur von System und Modell ist dabei nicht von primärem Interesse; unter Umständen sind Extrapolationen basierend auf einem reinen Datenmodell ausreichend.

Gestaltungs- und Optimierungsmodelle schliesslich werden in der Synthese verwendet, wobei sie dazu dienen, Struktur (Gestaltungsmodell) und Parameter (Optimierungsmodell) festzulegen.

2.2. Ebenen der Systembeschreibung

Übereinstimmend mit systemtheoretischen Konzepten, wie sie von Zadeh und Desoer [77], Ören [51] und Wymore [76] dargestellt werden, definiert Zeigler eine Hierarchie von Systembeschreibungen [78], [79]. Geschichtet sind die einzelnen Ebenen dabei von der Verhaltensbeschreibung her hin zur Beschreibung der Struktur, vom Abstrakten zum Konkreten.

Ebene 0 (Observation Frame)

$O = <T, X, Y>$

Durch einen Beobachtungsrahmen werden die Systemgrenzen definiert. Die Mengen X und Y entsprechen den Mengen der Werte, die der Eingang bzw. der Ausgang[1] annehmen kann. Die Zeit T, üblicherweise repräsentiert entweder durch die reellen Zahlen ($T=\mathfrak{R}$) oder ganzen Zahlen ($T=\mathfrak{I}$), dient dazu, die beobachteten Daten zu ordnen.

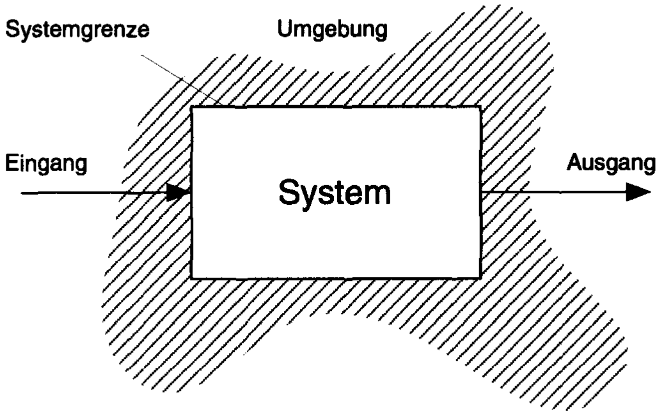


Fig. 2.2: Abgrenzung eines Systems

---

[1] Die Schnittstellen des abgegrenzten System werden in der skizzierten Klassifizierung von Anfang an eingeteilt in Ein- & Ausgänge. Dies setzt voraus, dass Kenntnis darüber vorhanden ist, wie die Kausalität innerhalb des Systems aussieht. In regelungstechnischen Prozessen hat man meist eine klare Vorstellung davon, wie beispielsweise ein Signalfluss verläuft; in einer elektrischen Schaltung ist diese Kenntnis nicht a apriori vorhanden. Willems [73] gibt eine allgemeinere Sicht dynamischer Systeme und betont implizite Aspekte der Darstellung.

Ebene 1 (Relation Observation)

$IORO = \langle T, X, \Omega, Y, R \rangle$

Ein Segment ω beispielsweise des Eingangs ist definiert über einem Intervall der Zeit: $\omega: \langle t_1, t_2 \rangle \rightarrow X$. Ω ist eine Untermenge aller möglichen Eingangssegmente, z.B. die Menge derjenigen Segmente, mit denen experimentiert wird. In dieser Abstraktionsstufe wird kein interner Zustand des Systems verwendet. Deshalb stellt R die Relation dar zwischen den einzelnen Elementen von Ω und *allen* dazugehörigen Ausgangssegmenten.

Ebene 2 (Function Observation)

$IOFO = \langle T, X, \Omega, Y, F \rangle$

Diese Ebene unterscheidet sich von Ebene 1 dadurch, dass F die Relation R ersetzt. F ist eine Menge von Funktionen f, wobei jede Funktion zu einem Eingangssegment genau ein Ausgangssegment erzeugt. Die Kenntnis von F ist äquivalent zur Kenntnis des Zustandes, den das System zum Startpunkt der Segmente hat.

Ebene 3 (Input / Output System)

$IOS = \langle T, X, \Omega, Q, Y, \delta, \lambda \rangle$

Auf dieser Ebene wird der Zustand des Systems, Q, eingeführt. Die Dynamik des Systems wird durch die Eingangsabbildung $\delta: Q \times \Omega \rightarrow Q$ repräsentiert. Die Ausgangsabbildung $\lambda: Q \rightarrow Y$ definiert die Abhängigkeit des Ausgangs vom Zustand. Es ist üblich, das Input/Output System als 'Dynamisches System' zu bezeichnen.

Ebene 4 (Structured Input / Output System)

$IORO = \langle T, X, \Omega, Q, Y, \delta, \lambda \rangle$

Auf dieser Ebene werden multivariable Mengen als Eingang, Zustand und Ausgang zugelassen.

Ebene 5 (Coupled Systems)

$CS = \langle T, X, Y, Komponenten, Verbindungen \rangle$

In gekoppelten Systemen werden verschiedene Teilsysteme unterschieden. Die Verbindungen definieren, wie Eingänge des Gesamtsystems und Ausgänge von Teilsystemen mit Ausgängen des Gesamtsystems und Eingängen von Teilsystemen zusammenhängen; der Zustand des Gesamtsystems ergibt sich durch Aggregation der Einzelzustände der Teilsysteme. Es können verschachtelte Hierarchien gebildet werden, indem gekoppelte Systeme wieder als Teilsysteme verwendet werden.

Die skizzierten Ebenen der Systembeschreibung spiegeln sich in den zugehörigen Modellen wider. Als Simulationsmodelle werden im folgenden Modelle betrachtet, die der 3. oder einer höheren Ebene zugehören. Das heisst, dass die Simulation sich auf Modelle beschränkt, die die folgenden Bedingungen erfüllen:

- Zu jedem Zeitpunkt ist der Zustand des Modells bzw. aller Teilmodelle bekannt.
- Die Verbindungen zwischen den Teilmodellen sind zu jedem Zeitpunkt eindeutig als Ein- bzw. Ausgänge der Teilmodelle markiert.
- Eine Verbindung kann nicht gleichzeitig Ausgang mehrerer Teilmodelle sein.
- Für jedes Teilmodell sind sowohl Eingangs- wie auch Ausgangsabbildung bekannt.

2.3. Simulation im Zeitbereich

Bereits auf der abstraktesten Ebene nach Zeigler ist die Zeitbasis explizit in der Systembeschreibung enthalten, wobei zwischen kontinuierlicher und diskreter Zeitbasis unterschieden wird.

Prähofer [59] unterscheidet aufgrund der Begriffe zeitkontinuierlich und zeitdiskret und unter Miteinbezug zweier Arten von Zustandsübergängen vier Typen von Modellen bzw. Simulationen im Zeitbereich.

Kategorien von im Zeitbereich simulierten Modellen		Zustandsübergänge	
		kontinuierlich	diskret
Zeit	kontinuierlich	Differentialgleichungen	ereignisorientierte Systeme
	diskret	Differenzengleichungen	Zustandsautomaten

Fig. 2.3: Typen im Zeitbereich simulierter Modelle (nach Prähofer)

Die Zeitbasis der ereignisorientierten Systeme wird dabei als kontinuierlich eingestuft. Diese Betrachtungsweise ist motiviert durch den Umstand, dass die Menge der Ereignisse bzw. der zugehörigen Zeitpunkte zwar abzählbar ist, die Zeitpunkte aber normalerweise auf die Menge der reellen Zahlen, $\Re$, abgebildet werden.

Eine unterschiedliche Betrachtungsweise insbesondere der ereignisorientierten Systeme wird von Frauenstein, Pape und Wagner [23] angegeben. Ihre Interpretation des Begriffs *kontinuierlich* geht dahin, dass der kontinuierlichen Simulation ein stetiger

Zeitablauf zugrundeliegt, während bei diskreter Simulation einzelne Zeitpunkte herausgegriffen werden, nämlich diejenigen, zu denen sich der Zustand des Modells ändert.

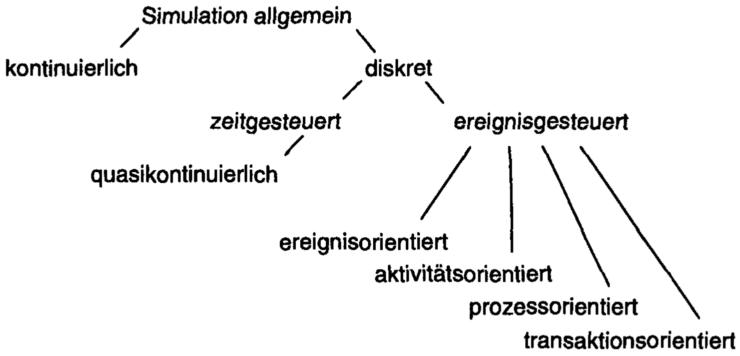


Fig. 2.4: Simulation im Zeitbereich (nach Frauenstein, Pape und Wagner)

Die diskrete Simulation wird weiter unterteilt in zeit- und ereignisgesteuerte Simulation. In der zeitgesteuerten Simulation sind die Zeitpunkte der Zustandsübergänge bereits zu Beginn der Simulation bekannt[1]. Bei der ereignisgesteuerten Simulation treten die Zustandsübergänge zu beliebigen diskreten Zeitpunkten auf. Die Kenntnis der einzelnen Punkte ergibt sich in der Regel erst zur Laufzeit der Simulation durch das Propagieren von Ereignissen. Künftige Ereignisse werden in einen zeitlich geordneten Ereigniskalender eingetragen, wobei zum Fortgang der Simulation jederzeit mindestens ein zukünftiges Ereignis bekannt sein muss. Die weitere Aufsplitterung der ereignisgesteuerten Simulationen ist dadurch motiviert, dass verschiedene Begriffe wie Ereignis, Aktivität, Prozess oder Transaktion bei der Modellierung von zentraler Bedeutung sind. Das Ereignis ist eine diskrete Zustandsänderung, die Aktivität eine Operation zwischen einem Anfangs- und Endereignis, der Prozess eine Reihe von Aktivitäten, die von einem Objekt durchlaufen werden. Bei der Transaktion bewegt sich ein mobiles Objekt zwischen statischen Objekten; Zustandsänderungen können am Objekt selbst wie auch in seiner Umgebung erfolgen.

Der Standpunkt, den der Modellierende bei der Formulierung ereignisgesteuerter Modelle einnimmt, wird wesentlich mitbeeinflusst von der benutzten Simulationssprache. Bei Verwendung einer sequentiellen, prozeduralen Sprache wird die Modellierung eher ereignisorientiert ausfallen. Enthält die Sprache jedoch objektorientierte Elemente oder Konstrukte zur parallelen Programmierung, so wird der Modellierende eher zu prozess- oder transaktionsorientierten Formulierungen tendieren.

---

[1] Sind die Zeitpunkte äquidistant und entspricht der Algorithmus zur Zustandsänderung der Integration eines kontinuierlichen Modells, so ist der Ausdruck *quasikontinuierlich* plausibel.

2.4. Kombinierte Modelle

Setzen sich Modelle aus Teilen unterschiedlichen Typs - kontinuierlich bzw. diskret - zusammen, so wird von kombinierten Modellen gesprochen. Die Notwendigkeit, Modelle in einer kombinierten Weise formulieren und simulieren zu können, ist bereits 1970 von Fahrland [22] ausführlich aufgezeigt worden.

2.4.1. Beispiele kombinierter Modelle

Neben einer breiten Palette von Systemen industrieller Art wie Papierfabriken, Schmelzöfen oder chemischen Prozessen listet Fahrland auch elektrische Systeme, Verkehrssysteme und biologische Prozesse wie z.B. neuro-muskuläre Systeme auf. Gerade durch die Verbreitung des Digitalrechners zur Regelung und Steuerung von Prozessen aller Art ist die Zahl der Systeme mit kombiniertem Charakter in den letzten Jahren enorm angestiegen.

Am Beispiel von Verkehrssystemen zeigt Cellier [11] auf, wie sehr die Perspektive beim Vorgang der Modellierung variiert werden kann: In makroskopischen Modellen ist der Verkehrs*fluss* von Interesse; er wird durch partielle Differentialgleichungen dargestellt, ohne einzelne Fahrzeuge zu modellieren. In mikroskopischen Modellen werden die Fahrzeuge als Einheiten betrachtet, die sich in diskreten Schritten bewegen. Bei submikroskopischen Modellen schliesslich wird jedes Fahrzeug durch sein kontinuierliches dynamisches Verhalten beschrieben, welches gewissen Diskontinuitäten unterliegt.

Analog zum Beispiel des Verkehrs ist es z.B. bei Abfertigungsprozessen möglich, entweder auf einer tieferen Ebene ein physikalisches kontinuierliches Modell auszuformulieren oder auf eine höhere Ebene zu abstrahieren und Abfertigungszeiten durch diskrete Zufallsvariablen zu modellieren. Wenn als Resultat einer Simulation statistische Aussagen genügen, dann reicht der diskrete Modellierungsansatz aus, ist dagegen der konkrete Zeitverlauf bestimmter Grössen von Interesse, so wird das kontinuierliche Teilmodell miteinbezogen. Die Möglichkeit, ein umfangreicheres kombiniertes Modell zu simulieren, kann so auch genutzt werden, um zu prüfen, ob auch ein einfacheres reduziertes diskretes Modell ausreicht.

Neben den obigen Beispielen, bei denen ganz offensichtlich diskrete mit kontinuierlichen Aspekten verknüpft sind, gibt es auch eine strukturell einfache Klasse von Modellen, die zwar formal durch Differentialgleichungen dargestellt werden können,

$$\dot{x} = f(x) = \begin{cases} f_1(x) & \textit{falls} \quad g(x) > 0 \\ f_2(x) & \textit{falls} \quad g(x) < 0 \end{cases}$$

bei denen aber - je nach Beschaffenheit der Vektorfelder f_1 & f_2 - die Differenzierbarkeit der Funktion $f(x)$ die Anforderungen der Integrationsroutine nicht erfüllt.

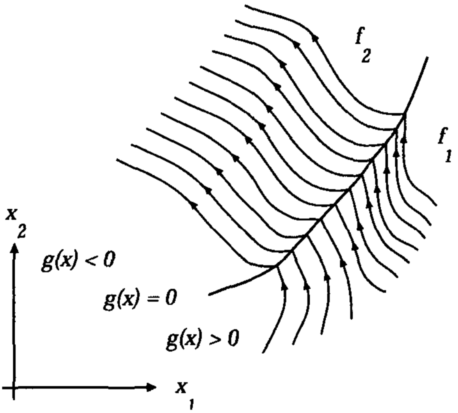


Fig. 2.5: Aufteilung des Zustandsraumes durch eine Schaltlinie

Fig. 2.5 zeigt eine Trajektorienschar, bei der entlang einer Schaltlinie singuläre Punkte in den einzelnen Trajektorien auftreten. Die korrekte Handhabung solcher Punkte als Zustandsereignisse ist eine der zentralen numerischen Schwierigkeiten bei der kombinierten Simulation.

2.4.2. Hybride Automaten

Der Term *hybrid* bezeichnet im Bereich der Automatik und Simulationstechnik traditionellerweise das Zusammenwirken analoger und digitaler Elemente. In den siebziger Jahren fand der Hybridrechner, eine Verbindung von Analogrechnern mit digitalen Bauelementen, weite Verbreitung bei der Simulation kombinierter Modelle. Durch die rasche Entwicklung des Digitalrechners sind Hybridrechner weitgehend verschwunden.

Für die theoretische Forschung im Gebiet der hybriden Systeme, die ab etwa 1990 intensiviert worden ist, wird bei der Modellbildung meist eine klare Trennung vollzogen zwischen einem kontinuierlichen und einem diskreten Teilmodell. Die zu kontrollierende Strecke und allfällige Filter werden als kontinuierlich betrachtet, die in einem Digitalrechner oder in einer digital arbeitenden Schaltung realisierte Steuerung als diskret. So definieren Stiver und Antsaklis [69] den hybriden Automaten in direkter Analogie zur gebräuchlichen Beschreibung abgetasteter Systeme (vgl. Fig. 2.6)

Das Interface innerhalb des hybriden Automaten ist dabei äquivalent zu den Elementen Abtaster/Halteglied (ZOH; zero order hold) des abgetasteten Systems. Das Interface arbeitet allerdings im Gegensatz zu Abtaster und Halteglied nicht zeitgesteuert nach einer fixen Taktrate, sondern ereignisgesteuert.

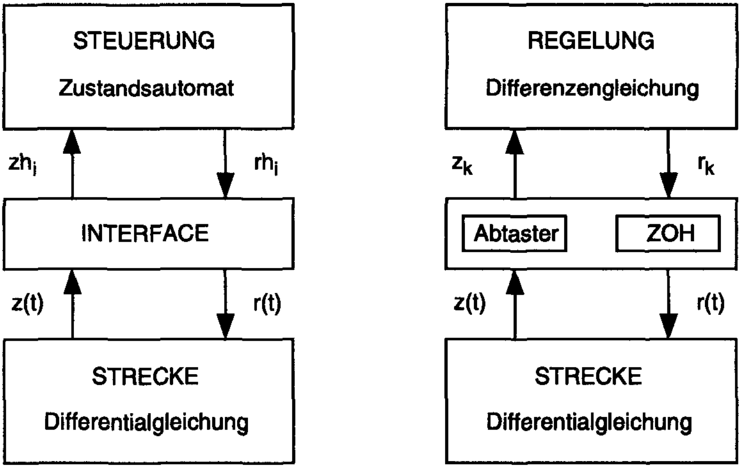


Fig. 2.6: Hybrider Automat nach Stiver und Antsaklis / abgetastetes System

Für die Simulation kontinuierlicher Systeme ist die explizite Trennung in Strecke und Steuerung häufig nicht möglich oder sinnvoll. Ausserdem ist eine eher artifizielle Konstruktion nötig, um in die Struktur von Stiver und Antsaklis einen Ereigniskalender zu integrieren, in dem Ereignisse vorausgeplant bzw. angekündigt werden können.

Alur, Courcoubetis, Henzinger und Ho [1] geben eine alternative Definition hybrider Automaten als Verallgemeinerung endlicher Zustandsautomaten. Die einzelnen Elemente des Tupels

$$A = \langle V_K, Q, \mu_1, \mu_2, \mu_3 \rangle$$

haben folgende Bedeutungen:

- Menge kontinuierlicher Variablen: V_K

  V_K ist eine endliche Menge von n reellwertigen Variablen, die den kontinuierlichen Zustand σ verkörpern. $\Sigma_K \subseteq \Re^n$ ist die Menge aller erlaubten kontinuierlichen Zustände.

- Diskreter Zustandsraum: Q

  Q ist eine endliche Menge diskreter Zustände oder Lokationen. Eine Kontrollvariable, deren Wertebereich sich über die Menge Q aller Lokationen erstreckt, komplettiert den Zustand. Dieser ist somit gegeben durch ein Paar (l, σ), wobei $l \in Q$ und $\sigma \in \Sigma_K$ ist. Die Menge aller Zustände ist $\Sigma = Q \times \Sigma_K$.

- Kontinuierliche Dynamik: μ_1

  μ_1 ordnet jeder Lokation eine Menge möglicher Aktivitäten zu. Jede Aktivität ist eine C^∞-Funktion von $\Re^+$ nach Σ_K. Aktivitäten definieren die zeitliche Entwicklung des kontinuierlichen Zustandes innerhalb einer Lokation.

  NB: Sind die Aktivitäten keine C^∞-Funktionen, müssen die Lokationen weiter verfeinert werden.

- Auslöser von Zustandsereignissen: μ_2

  μ_2 ordnet jeder Lokation $l \in Q$ eine invariante Menge $\mu_2(l) \subseteq \Sigma_K$ zu. Befindet sich das System in der Lokation l, der kontinuierliche Zustand aber nicht (mehr) in $\mu_2(l)$, wird ein Zustandsereignis generiert.

- Menge von Übergangsrelationen: μ_3

  μ_3 definiert zu jedem geordneten Paar von Lokationen eine Übergangsrelation, eine Abbildung des kontinuierlichen Zustandes in l_1 zu einem kontinuierlichen Zustand in l_2: $\mu_3(l_1, l_2) \subseteq (\Sigma_K \times \Sigma_K)$.

Beispiel:
In die geschaltete Differentialgleichung aus Fig. 2.5 kann ein solcher hybrider Automat hineininterpretiert werden:

- die Menge Q hat dabei zwei Elemente,
- die Menge V_K hat die zwei Elemente x_1 und x_2,
- die kontinuierliche Dynamik μ_1 ist gegeben durch die (fortgesetzten) Trajektorienscharen,
- der Auslöser von Zustandsereignissen, μ_2, ist für die eine Lokation gegeben durch die Schaltlinie; innerhalb der anderen Lokation werden keine Zustandsereignisse ausgelöst,
- μ_3, die Menge der Übergangsrelationen, ist degeneriert zu einer Abbildung des kontinuierlichen Zustands auf sich selbst.

Um einen konkreten zeitlichen Ablauf zu gewährleisten, reicht die obige Definition nicht aus. So fehlt beim Beispiel der geschalteten Differentialgleichung das Bindeglied zwischen dem Erreichen der Schaltlinie und dem Weiterfahren auf der anderen Trajektorienschar. Nötige Ergänzungen sind

- die Auswertung der durch μ_2 generierten Ereignisse;
- zur Aktivierung von μ_3, bzw. zur Auslösung von Übergängen, wird die Angabe einer neuen Lokation benötigt;
- daneben soll auch die Möglichkeit eingeführt werden, Ereignisse in einen Kalender einzutragen.

Der hybride Automat wird zu diesem Zweck vervollständigt:

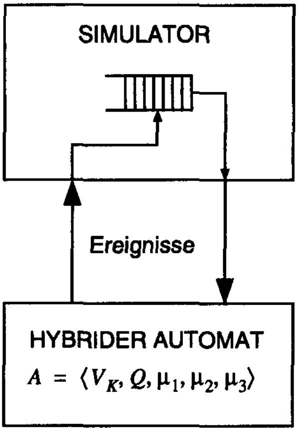


Fig. 2.7: Vervollständigung des hybriden Automaten

Der in Fig. 2.7 eingezeichnete Simulator kann interpretiert werden als eine überwachende Kontrollinstanz (Supervisor), wie sie im Bereich der rein diskreten Zustandsautomaten gängig sind [7], [75]. Die Aufgabe des Simulators ist ebenfalls eine Art der Überwachung, nämlich diejenige des geordneten Abarbeitens der anfallenden Ereignisse. In einfachen Fällen - wie im Beispiel der geschalteten Trajektorien - wird ein Ereignis, das vom hybriden Automaten kommt, verzögerungsfrei zurückgeschickt, wobei der neu einzunehmende diskrete Zustand (Lokation) als Information mitgegeben wird. In komplexeren Situationen kann ein Ereignis in einem Kalender zwischengespeichert und in einem späteren Zeitpunkt reaktiviert werden.

Der Simulator, wie er in COSIMO implementiert ist, besitzt Möglichkeiten, die über das bisher beschriebene hinausgehen: liegt während der Simulation die Kontrolle bei der überwachenden Instanz, so kann sie beliebig auf die Struktur des Automaten einwirken. Im einzelnen heisst das:

- die Anzahl Zustände in Q verändern,
- n, die Anzahl Zustände in V_K, verändern,
- die kontinuierliche Dynamik μ_1 neu definieren,
- die invarianten Mengen μ_2 neu bestimmen,
- μ_3, die Übergangsrelation, neu bestimmen.

Diese Möglichkeit, zu Zeitpunkten von Ereignissen die Struktur des Simulationsmodells zu verändern, sozusagen während des Spiels die Regeln zu ändern, wird damit bezahlt, dass der kontinuierliche Teil des Automaten in einer expliziten Form deklariert werden muss. Implizite Differentialgleichungen werden durch die strukturelle Variabilität deutlich schwieriger zu handhaben[1].

Eine weitere in COSIMO implementierte Erweiterung besteht darin, Differentialgleichung zu simulieren, die Verzögerungen beinhalten:

$$\dot{\sigma}(t) = f(\sigma(t), \sigma(t-\tau_1), \sigma(t-\tau_2), \ldots)$$

bzw.

$$\dot{\sigma}(t) = f(\sigma(t), \sigma(t-\tau_1(t)), \sigma(t-\tau_2(t)), \ldots)$$

Ermöglicht wird dies durch die Verwendung von Interpolationsformeln, die es erlauben, während der Simulation vergangene Werte des Zustands σ zu berechnen.

2.5. Kombinierte Simulatoren

Die Simulation kombinierter Modelle - die kombinierte Simulation - wird auf dem Digitalrechner letztlich auf die diskrete Simulation abgebildet, da die numerische Integration in diskreten Schritten durchgeführt wird[2]. Eine mögliche Abfolge der relevanten Zeitpunkte einer kombinierten Simulation ist in Fig. 2.8 dargestellt. Bei den Ereignissen sind zwei Typen zu unterscheiden: Zeitereignisse, deren Auftreten im vornherein bekannt ist, und Zustandsereignisse, die durch den Integrationsvorgang ausgelöst werden.

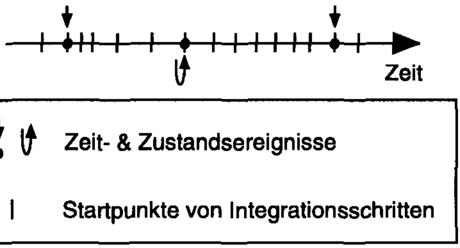


Fig. 2.8: Ablauf einer kombinierten Simulation

---

[1] Die Vorzüge der impliziten Darstellung liegen häufig darin, eine allgemeinere Form zu repräsentieren, die ein Sortieren von Gleichungen erspart; vielfach ist durch Umformulierung aber eine explizite Darstellung möglich. Wird die implizite Form auf Probleme angewendet, die explizit nicht darstellbar sind, treten vielfach numerische Probleme auf, wie beispielsweise in Initialisierungsphasen, wenn ein konsistenter kontinuierlicher Zustand gefunden werden muss.

[2] Dies gilt gleichermassen für die rein kontinuierliche Simulation. Im Gegensatz zur echt diskreten Simulation ist aber bei kontinuierlicher wie kombinierter Simulation im allgemeinen der Endpunkt eines Integrationsschrittes nicht zum vornherein bekannt.

Im Zusammenhang mit der Handhabung von Zustandsereignissen ergeben sich einige Aspekte, nach denen sich kombinierte Simulatoren charakterisieren lassen, nämlich:

- Wird eine explizite Deklaration/Detektion von Zustandsereignissen unterstützt?
- Ist es möglich, auf Zustandsereignisse aktiv zu reagieren?
- Sind während der Simulation strukturelle Änderungen des Modells möglich?

Daneben ist der Hintergrund eines Pakets von Bedeutung:

- Die ursprüngliche Ausrichtung; z.B. auf kontinuierliche Simulation.
- Die zugrundeliegende Programmiersprache: sie entscheidet wesentlich mit, ob z.B. eigene Typendeklarationen oder objektorientierte Elemente benutzt werden dürfen; darüberhinaus ist es von Bedeutung, ob eigene Erweiterungen interpretiert werden oder durch Kompilieren und Linken zugefügt werden.

Einige weitere Kriterien beziehen sich darauf, wie komfortabel die Software für den Ersteller von Modellen oder den Konsumenten von Simulationen ist:

- Ist graphisches Konfigurieren / Modellieren möglich?
- Kann ein Modell hierarchisch aufgebaut sein?
- Welche Möglichkeiten bestehen zur Visualisierung, Animation und interaktiven Simulation?

Die folgende Liste kommerzieller und akademischer Programme ist unvollständig, soll aber summarisch eine Idee vermitteln, entlang welcher Linien kombinierte Simulatoren sich entwickelt haben.

2.5.1. GASP, SLAM, COSY

Nachdem Fahrland die Bedeutung der kombinierten Simulation erkannt hatte, benutzte er für eine Machbarkeitsstudie eine Synthese aus GASP II und PACTOLUS, die die Bereiche ereignisorientierte bzw. kontinuierliche Simulation abdeckten. Die Synchronisation zwischen den beiden Programmen war relativ einfach zu implementieren, da beide unmittelbar auf FORTRAN[1] aufbauen.

In einer weiteren Version von GASP IV [61] wurden 1974 kombinierte Aspekte eingebracht. Darauf aufbauende Pakete wie SLAM [62] oder COSY [11] haben die Limitierungen geerbt, die auf der unmittelbaren Verbindung zu FORTRAN beruhen. Modelle werden in einer FORTRAN-ähnlichen Sprache geschrieben, kompiliert und gelinkt. Es ist nicht möglich, eigene Datentypen zu definieren; programmiert wird prozedural, FORTRAN-Subroutinen sind einbindbar.

Die explizite Deklaration von Zustandsereignissen ist möglich, detektiert werden die Ereignisse durch wiederholtes Integrieren mit angepasster Schrittweite. Das Auftreten eines Ereignisses kann mit dem Aufruf einer Subroutine verknüpft werden.

---

[1] Mit FORTRAN ist jeweils FORTRAN IV bzw. FORTRAN 77 gemeint, die lingua franca der Naturwissenschaftler, eine prozedurale Sprache, die grosse Verbreitung im Bereich numerischer Berechnungen gefunden hat.

Modifikationen der Modellstruktur wie das Ändern der Anzahl kontinuierlicher Zustände sind während der Simulation nicht möglich.

Die Modelle werden textuell editiert und in einer kommandozeilenorientierten Umgebung simuliert. Animation oder interaktive Simulation ist nicht möglich.

2.5.2. SIMAN

SIMAN [55], eine weitere Entwicklung in der obig skizzierten Linie, ist eher prozess- und aktionsorientiert und im Bereich von Operations Research sehr populär. Die Modellbeschreibungen sind etwas weniger stark an FORTRAN angelehnt; sowohl die Einbindung von FORTRAN- wie auch C-Code ist möglich.

Für die Behandlung der Zustandsereignisse und für die Inflexibilität der Dimension des kontinuierlichen Zustandsraums gilt dasselbe wie bei der GASP-Familie.

Im Bereich der graphischen Oberflächen sind Ergänzungen entwickelt worden, wie SIMANIM, das eine Echtzeitanimation im Rahmen des IBM-Graphikzeichensatzes ermöglicht, oder CINEMA, des neben echt graphischer Animation auch Interaktion erlaubt.

2.5.3. ACSL, EASY5

Für ACSL und EASY5 gilt im wesentlichen dasselbe wie für die GASP-Familie. Beide sind stark FORTRAN-orientiert, stammen aber von der Seite der kontinuierlichen Simulation her und basieren auf dem CSSL-Standard (Continuous System Simulation Language), der bereits 1967 etabliert wurde [5].

Im Verlaufe der Jahre sind diese Programme bedienerfreundlicher geworden, wie z.B. durch die Produkte PROTOBLOCK oder ACSL Graphic Modeller. Allerdings sind diese graphischen Oberflächen nur für Darstellungen geeignet, bei denen die Verbindungen zwischen den Blöcken Signalflüsse darstellen, aber keine Kontrollflüsse; das Propagieren von Ereignissen beispielsweise ist so nur sehr schwerfällig formulierbar.

Interaktive Simulation wird rudimentär unterstützt; Simulationen können unterbrochen und Parameter verändert werden.

2.5.4. MATRIXx, MATLAB

Bei MATRIXx und MATLAB liegt der Ursprung in den Bibliotheken LINPACK und EISPACK, die zur Lösung numerischer Probleme im Bereich der linearen Algebra implementiert worden sind. Die heutigen Versionen basieren auf der Sprache C; kompilierter C-Code ist verwendbar.

Die ursprüngliche Verwendung von MATRIXx und MATLAB liegt im CACSD-Bereich, wobei über eine Kommandozeile der Interpreter aktiviert wird. Die Daten - mit Namen versehene Matrizen - liegen in einem Workspace. Die Möglichkeiten zur Simulation waren ursprünglich auf nichtlineare kontinuierliche Modelle beschränkt.

Heute existieren graphische Oberflächen, die das Konfigurieren hierarchischer Modelle erlauben.

In SystemBuild, einer Erweiterung von MATRIXx, ist die Modellierung abgetasteter Systeme möglich. Zusätzlich wird die graphische Modellierung von Zustandsautomaten unterstützt. Allerdings ist die Simulation auf Signalflüsse fokussiert. Auf Strukturen wie einen Ereigniskalender kann nicht zugegriffen werden.

SIMULINK, das Pendant zu SystemBuild bei MATLAB, ermöglicht die Modellierung abgetasteter Systeme; Zustandsereignisse werden nicht explizit detektiert und können nicht propagiert werden. SIKOSS, ein ergänzendes Paket [60], erlaubt es, SIMULINK-Diagramme durch Petri-Netze zu ergänzen und so hybride Automaten im Sinne von Stiver und Antsaklis [69] zu implementieren; SIKOSS verbessert allerdings nur die Möglichkeiten zur Propagierung von Ereignissen; eine hinreichende Genauigkeit bei deren Detektion muss ebenfalls über die Schrittweite der Integration erzwungen werden.

Taylor [70] schlägt Erweiterungen der Integrationsalgorithmen vor, die die Detektion von Zustandsereignissen auf der Basis wiederholter - verkürzter - Integrationsschritte unterstützen.

2.5.5. Dymola

Dymola ist primär auf zeitkontinuierliche Modelle zugeschnitten; die Modelle können *in einer akausalen, impliziten Form beschrieben werden [19], [20]*. Durch symbolische Manipulation wird diese in eine äquivalente Zustandsraumbeschreibung gebracht, die dann zu Simulationen beispielsweise in ACSL verwendet werden kann. Bei der Übersetzung werden darüber hinaus diskontinuierliche Funktionen - solche die z.B. eine IF-Anweisung enthalten - in entsprechende Deklarationen von Zustandsereignissen umformuliert.

Beispielsweise durch Zustandsereignisse hervorgerufene Strukturänderungen laufen weitgehend automatisiert ab; allerdings können nur *Um*strukturierungen behandelt werden, wie sie z.B. in elektromechanischen Systemen auftreten, in denen die Anzahl der Zustände konstant ist.

Die Möglichkeiten des graphischen Editierens, der Visualisierung und der Animation sind weit entwickelt; Interaktion ist nicht möglich.

2.5.6. OMOLA / OMSIM

Eine Modellierungssprache, die aus Dymola hervorgegangen ist, ist OMOLA (Object-oriented MOdeling LAnguage) [46], [2], [3]. Die Definition der einzelnen Modellkomponenten erfolgt objektorientiert; durch die Konstrukte *ISA* und *WITH* werden sowohl Klassenhierarchien als auch Hierarchien durch Verschachtelung der Modellkomponenten ermöglicht.

OMOLA ist gedacht als Möglichkeit, Modelle in einer Bibliothek abzulegen und weiterzuentwickeln. Unterschiedliche Programme wie Simulatoren oder solche zur numerischen bzw. symbolischen Analyse können auf die Bibliothek zugreifen und mit den Modellen arbeiten. Die Benutzung der Modelle erfolgt über einen Parser; die einzelnen Komponenten sind bezüglich ihrer Funktionalität völlig transparent und können deshalb z.B. auch symbolisch manipuliert werden.

OMSIM ist eine Simulationsumgebung, die OMOLA-Modelle simulieren kann. Der Simulator ist an die Definition hybrider Automaten angelehnt, wie sie von Alur, Courcoubetis, Henzinger und Ho [1] gegeben ist.

Detektion und Propagierung von Ereignissen wird flexibel unterstützt. Wie bei Dymola ist Strukturänderung nur im Sinne von *Um*strukturierung möglich, dynamisches Generieren neuer Modellkomponenten wird nicht unterstützt. Differentialgleichungen mit Verzögerungen können nicht modelliert werden.

Graphisches Editieren wird in bescheidenem Rahmen unterstützt, hierarchische Modelle ebenfalls. Die Visualisierung von Graphen ist möglich, weitergehende Animation und Interaktion nicht.

2.5.7. SAM-Set, ModelWorks

Sowohl SAM-Set (Simulation And Modelling Set) [71] wie auch ModelWorks [26] basieren auf Modula-2 und wurden vorwiegend im Hinblick auf den Einsatz in Lehre und Forschung konzipiert und auf dem Apple Macintosh implementiert. SAM-Set deckt kontinuierliche wie diskrete Simulation ab; ein rudimentärer Kopplungsmechanismus existiert. In ModelWorks wird die kombinierte Simulation besser unterstützt; allerdings - wie bei der reinen Integration - ohne numerisch besonders exakte Resultate zu liefern. Dies rührt daher, dass ModelWorks primär im Bereich der Ökologie verwendet wird, wo auch die Modelle nicht sehr präzis gegeben sind.

Modelle in SAM-Set wie auch in ModelWorks werden textuell definiert und können hierarchisch verschachtelt werden. Bei der Implementierung neuer Modelle kann neben simulatorspezifischen Elementen auch der ganze Sprach- und Funktionsumfang von Modula-2 verwendet werden.

Es können eigene Typen verwendet werden. Einige der in Anhang C aufgelisteten Prinzipien objektorientierter Programmierung wie z.B. Datenabstraktion werden in Modula-2 unterstützt, die Vererbung allerdings nicht. Deshalb ergeben sich natürliche Limiten bei der Verwendung dieser Simulatoren, sobald man mit inhomogenen Listen arbeiten will oder wenn bestehende Modelle erweitert werden sollen.

Für Visualisierung und Animation wird die übliche graphische Oberfläche des Macintosh verwendet, wobei die sogenannte Dialog Machine [25] als Zwischenschicht benutzt werden kann. Interaktion wird in ModelWorks unterstützt, z.B. durch die Installation eines Handlers, der Mausklicks verarbeitet.

Konzipierung eines kombinierten Simulators

3

3.1. Ablauf einer kombinierten Simulation

Der Ablauf einer kombinierten Simulation besteht im wesentlichen aus dem wiederholten Ablauf der folgenden vier Phasen: (Fig. 3.1):

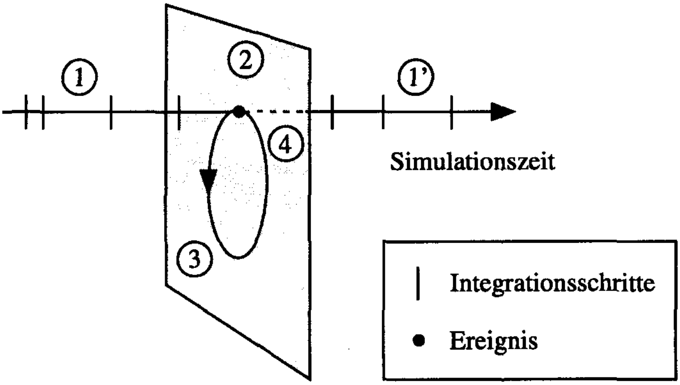


Fig. 3.1: *Phasen der kombinierten Simulation*

1. Integrieren einer Differentialgleichung
2. Detektieren eines Ereignisses
3. Propagieren des Ereignisses
4. Finden konsistenter Anfangsbedingungen

Die Integration einer Differentialgleichung entspricht dem Ablauf einer Simulation bei einem zeitkontinuierlichem Modell. Die Integrationsschritte werden in der Regel dynamisch an die momentane Situation adaptiert. In der Terminologie des in Kap. 2.4.2 vorgestellten hybriden Automaten nach Alur entspricht die Integration der Bewegung innerhalb einer Lokation.

Ein auftretendes Ereignis kann von unterschiedlicher Natur sein: Zeitereignis oder Zustandsereignis. Beim Zeitereignis ist der Zeitpunkt des Eintretens zu Simulationsbeginn bekannt (exogenes Zeitereignis) oder er wird im Verlauf der Simulation durch ein bereits früher aufgetretenes Ereignis bestimmt (endogenes Zeitereignis). Das Zustandsereignis hingegen wird vom zeitkontinuierlichen Teilsystem her ausgelöst; z.B. durch Schwellwertüberschreitung von Zuständen oder algebraischen Grössen. Beim hybriden Automaten werden Zeitereignisse von aussen durch den Ereigniskalender herangetragen; Zustandsereignisse dagegen vom Automaten selbst generiert.

Die Propagierung eines Ereignisses umfasst alle Reaktionen, die das Ereignis nach sich ziehen kann, von Zustands- über Parameter- und Strukturänderungen bis zum Generieren endogener Zeitereignisse. Der Automat nach Alur bildet den kontinuierlichen Zustand ab; die anderen Änderungen obliegen dem Simulator.

Vor der Wiederaufnahme der Integration muss sichergestellt sein, dass alle anliegenden Ereignisse abgearbeitet sind. Im weiteren muss dafür gesorgt werden, dass konsistente Anfangsbedingungen für die Integration vorliegen. Dies entspricht dem Lösen eines algebraischen Gleichungssystems.

Kombinierte Simulation kann somit aufgefasst werden als Synthese von diskreter und kontinuierlicher Simulation, wobei die Kontrolle über den Weitergang der Simulation zwischen den beiden beteiligten Komponenten hin- und hergegeben wird:

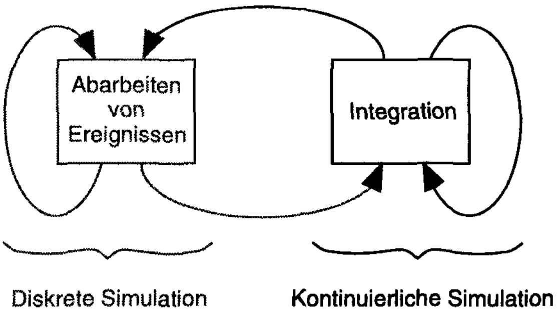


Fig. 3.2: Zusammenspiel diskreter und kontinuierlicher Simulation

Die Hauptaspekte, die bei der Konzipierung des Simulationskerns beachtet werden müssen, sind:

- Die verwendete Integrationsroutine soll die Detektion von Zustandsereignissen unterstützen. Nach Strukturänderungen muss sie in der Lage sein, flexibel und effizient weiterzufahren.
- Die Propagierung eines Ereignisses hin zu verschiedenen Teilen des Gesamtmodells sollte im Sinne eines aktiven Verschickens von Meldungen unterstützt werden.

3.2. Integration von Differentialgleichungen

Zur Simulation zeitkontinuierlicher Systeme, bzw. zur Integration von Differentialgleichungen, existiert eine Vielzahl ausgereifter Algorithmen [34], [35], die je nach Problemstellung und Anforderungen zur Anwendung kommen. Kriterien, nach denen sie charakterisiert werden, sind:

- die Klasse der behandelbaren Differentialgleichungen
- die Mechanismen, nach denen ein Algorithmus im wesentlichen abläuft
- die Genauigkeit und Effizienz eines Algorithmus
- Möglichkeiten zur Detektion von Zustandsereignissen
- weitere Zusatzfunktionen, welche über das eigentliche Integrieren hinausgehen.

3.2.1. Klassen von Differentialgleichungssystemen

- explizite DE (Differential Equation):

$$\dot{x}(t) = f(x(t), t))$$

- semiexplizite DAE (Differential-Algebraic Equation):

$$\begin{aligned} \dot{x}(t) &= f(x(t), y(t), t)) \\ 0 &= g(x(t), y(t), t)) \end{aligned}$$

- implizite DAE:

$$0 = g(\dot{x}(t), x(t), y(t), t)$$

NB: Durch Einführen von Verzögerungselementen ergeben sich weitere Klassen.

3.2.2. Einschrittverfahren und Mehrschrittverfahren

In Einschrittverfahren wird Information über den Zustand in einem Zeitpunkt verwendet, um den Zustand in einem späteren Zeitpunkt zu berechnen. In Mehrschrittverfahren wird dazu auf Information aus mehreren früheren Zeitpunkten zugegriffen. Dies bringt mit sich, dass bei Mehrschrittverfahren ein spezieller Startvorgang nötig ist.

3.2.3. Explizite und Implizite Verfahren

Während die Algorithmen expliziter Verfahren pro Integrationsschritt eine fixe Anzahl Operationen benötigen und somit 'straightforward' durchgerechnet werden können, müssen bei impliziten Verfahren Iterationen in Kauf genommen werden, um algebraische Schleifen aufzulösen.

3.2.4. Effizienz und Genauigkeit

Die Effizienz wird durch die Rechenzeit oder die Anzahl Operationen bei bestimmten Integrationsproblemen (Benchmarks) definiert.

Die Simulation kontinuierlicher Prozesse auf einem Digitalrechner ist inhärent nicht genau; ihr approximativer Charakter[1] hat verschiedene Ursachen.

- Die Darstellung beliebiger reeller Zahlen auf einem Digitalrechner ist per definitionem fehlerbehaftet.
- Zeitdiskrete Integrationsalgorithmen können die zeitkontinuierliche Dynamik im allgemeinen nur näherungsweise beschreiben.
- Bei steifen Systemen oder solchen, die Unstetigkeiten (geschaltete Differentialgleichungen) aufweisen, verschärft sich dieser Problem weiter. Bei chaotischen Systemen schliesslich wird jede beliebig exakte Simulation bei einer hinreichend langen Laufzeit Resultate erbringen, die höchstens noch statistische Aussagekraft haben.[2]

Der globale Fehler, der sich über eine ganze Simulation akkumuliert, ist meist schwieriger abzuschätzen als der lokale Fehler, der sich innerhalb eines Integrationsschrittes ergibt. Zur Klassifizierung der Genauigkeit wird deshalb der lokale Fehler verwendet.

3.2.5. Behandlung von Zustandsereignissen

Die korrekte Behandlung von Zustandsereignissen ist im wesentlichen durch zwei Gründe motiviert. Einerseits soll Simulation eine Differentialgleichung auch an denjenigen Stellen genau nachbilden, an denen sie die Anforderungen bezüglich Differenzierbarkeit nicht erfüllt, andererseits kann das Auftreten des Ereignisses an sich von Interesse sein für den Fortgang der Simulation.

In Anlehnung an Fig. 2.5 - respektive an die zugehörige Differentialgleichung - zeigt Fig. 3.3 ein qualitatives Beispiel für den ersten, Fig. 3.4 eines für den zweiten Aspekt.

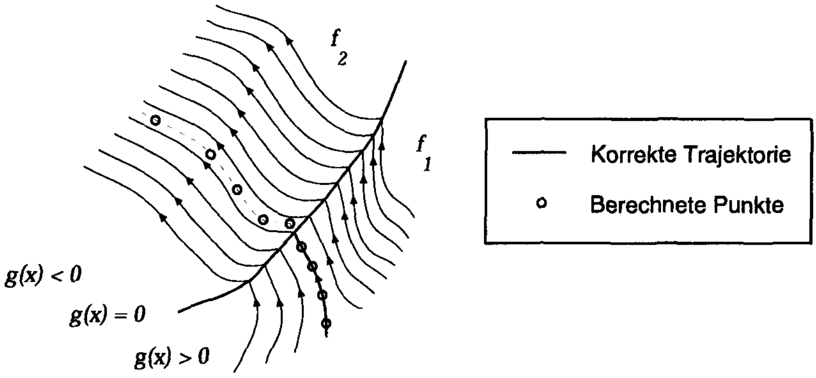


Fig. 3.3: Schalten mit Verzögerung

---

1. Das symbolische exakte Lösen, das bei einfacheren Klassen von Differentialgleichungen möglich ist, wird im allgemeinen nicht als Simulation verstanden.
2. *Chaos macht jeden Computer nieder.* Kapitelüberschrift aus *Bausteine des Chaos: Fraktale* von Peitgen, Jürgens und Saupe [56].

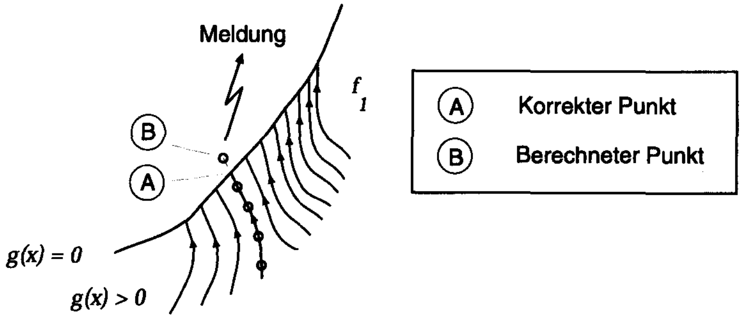


Fig. 3.4: Verzögertes Verschicken einer Meldung

In beiden Fällen hängt es vom konkreten Problem ab, ob die Verzögerung tolerierbar ist oder nicht.

Die gängigen Methoden zur Behandlung von Zustandsereignissen lassen sich in die folgenden Klassen gliedern:

1. Ignorieren der Singularität $g(x) = 0$
2. Detektion der Singularität durch wiederholtes Integrieren
3. Detektion der Singularität durch Interpolation

Die erste Methode arbeitet völlig implizit. Weder Kenntnis der Schaltlinie noch der Struktur der Funktion f ist vorhanden; der Fehler wird einzig in Grenzen gehalten über die Abschätzung des lokalen Fehlers, der die Schrittweite beeinflusst. Diese Methode ist allenfalls geeignet, bei unkritischen Fällen über Unstetigkeiten hinwegzuintegrieren; es kann aber keine Meldung generiert werden.

Bei den anderen zwei Methoden wird nach jedem Integrationsschritt explizit getestet, ob $g(x)$ das Vorzeichen gewechselt hat. Falls ja, wird die Einhaltung einer Toleranz geprüft. Ist diese verletzt, wird versucht, Zeitpunkt und Zustand hinreichend genau zu berechnen ('g-stop facility'). Bei Methode zwei erfolgt dies durch Verkleinerung der Integrationsschrittweite und wiederholtes Integrieren, bei Methode drei über Interpolationsformeln.

Die zweite und dritte Methode können weiter unterteilt werden in Methoden, die bei der Integration

A. den geschlossenen Ausdruck $f(x)$ benützen [21] oder

B. explizit unterscheiden zwischen $f_1(x)$ und $f_2(x)$ und die beiden Ausdrücke konsequent nur vor bzw. nach der Detektion verwenden.

Für Methoden der ersten Klasse, die zwar Ereignisse nicht explizit detektieren können, aber über eine Schrittweitenanpassung verfügen, gibt es ein eher etwas saloppes Verfahren, genauere Detektion zu erzwingen. Es besteht darin, die Differentialgleichung in der Umgebung der Singularität *g(x) = 0* künstlich zu versteifen, indem sie um einen Zustand in der folgenden Art erweitert wird[1]:

$$\dot{x}_{n+1}(t) = \frac{1}{tol} sign(g(x_{1\ldots n}))$$

Dieser Ansatz wird als ungünstig beurteilt, da

- die Systemordnung unnötig aufgebläht wird,
- die Singularität - zumindest bei Algorithmen mit fixer Anzahl kontinuierlicher Zustände - bei Bedarf nicht deaktiviert werden kann,
- die an und für sich explizit verfügbare Information über die Lage der Diskontinuität in ein numerisch ungünstigeres implizites Problem eingekleidet wird,
- die Lösung über eine Reduzierung der Schrittweite erfolgt: vor einer Diskontinuität müssen üblicherweise mehrere Integrationsschritte zurückgewiesen werden, nach der Diskontinuität muss die Schrittweite über mehrere Schritte - abhängig von f_2 - nach oben angepasst werden.

3.2.6. Zusätzliche Funktionalität

Eignung für steife Systeme / Steifigkeitsdetektion

Die korrekte mathematische Definition steifer Systeme ist umstritten[2]; im allgemeinen wird darunter verstanden, dass die Jacobi-Matrix - die Systemmatrix des linearisierten Systems - Eigenwerte aufweist, die um Grössenordnungen auseinanderliegen.

Automatisches Finden einer Startschrittweite / Schrittweitenadaption

Die Steuerung der Schrittweite erfolgt meist durch Vorgabe des maximal erlaubten relativen Fehlers, der in einem Schritt auftreten darf. Die Abschätzung des tatsächlich gemachten Fehlers hilft zu entscheiden, ob ein Schritt verkürzt wiederholt werden muss, aber auch bei der Festlegung der Grösse weiterer Schritte. Häufig werden für die Schrittweite untere und obere Schranken vorgegeben, um den Simulationsfortgang in einem gewünschten zeitlichen Rahmen zu halten.

Interpolationsformeln

Nicht nur der Zustand an den einzelnen Stützstellen der Integration ist von Interesse, sondern auch die Zwischengeschichte; z.B. zu graphischen Zwecken. Interpolationsinformation soll idealerweise nebenbei, ohne zusätzliche Rechenoperationen, anfallen, um die Effizienz nicht zu mindern.

---

1. *tol* ist dabei die Integrationstoleranz. Der neue Zustand x_{n+1} wird bei dieser Methode üblicherweise durch einen Reset-Mechanismus vor einem Überlauf geschützt.
2. Eine der frühesten wie auch pragmatischsten Definitionen stammt von Curtiss & Hirschfelder [13]: "Stiff equations are equations where certain implicit methods, (...), perform better, usually tremendously better, than explicit ones."

Unterstützung zeitlicher Verzögerungselemente im Modell

Enthält ein Modell Verzögerungselemente, so muss während der Integration auf vergangene Zustände oder Teile des Zustands zugegriffen werden können. Methoden, die mit konstanter Integrationsschrittweite arbeiten, unterstützen nur konstante Verzögerungen, die einem ganzzahligen Vielfachen des Integrationsschritts entsprechen. Mit Interpolationsformeln dagegen wird diesbezüglich volle Flexibilität erreicht.

3.3. Wahl einer geeigneten Klasse von Algorithmen

Die obigen Klassifizierungskriterien sind in hohem Masse voneinander abhängig, implizieren einander oder schliessen einander aus, so dass es nicht möglich ist, die verschiedenen Kriterien isoliert zu betrachten, um für eine Klasse von Problemen geeignete Algorithmen zu bestimmen.

Unter Berücksichtigung der speziellen Anforderungen der kombinierten Simulation fiel die Wahl der prioritär zu verwendenden Integrationsalgorithmen auf die Klasse der einschrittigen Runge-Kutta-Methoden mit Integrationsformeln. Im folgenden wird diese Wahl durch einige Betrachtungen untermauert:

- Obwohl die Möglichkeit zur Behandlung impliziter Differentialgleichungen wünschenswert ist, wurde darauf verzichtet, weil in diesem Bereich eine Reihe ungelöster Probleme existiert. Zum einen bestehen Schwierigkeiten, bei Zustandsereignissen einen konsistenten Folgezustand zu bestimmen [53], [54]. Zum andern - weniger schwer wiegend - sind die heutigen Algorithmen nicht in der Lage, Probleme beliebig hoher Indizierung zu lösen [10].
- Die Klassen der behandelbaren Systembeschreibungen ist damit eingeschränkt auf semiexplizite Differentialgleichungen. Der Verzicht auf die Klasse von impliziten Differentialgleichungen fällt nicht so stark ins Gewicht, da bei den meisten Anwendungen durch Sortieren - bzw. Umsortieren während der Simulation - die gewünschte Struktur erreicht werden kann.
- Mit zunehmender Häufigkeit des Auftretens von Ereignissen werden Mehrschrittverfahren ineffizient, da nach jedem Ereignis eine Initialisierung notwendig ist. Andererseits sind Mehrschrittverfahren traditionell eher geeignet, Interpolationsinformation zu liefern, wie sie zur Detektion von Ereignissen verwendet werden kann. Neuere Entwicklungen im Bereich der Einschrittverfahren, z.B. Runge-Kutta-Methoden nach Dormand & Prince [17], können nun allerdings diese Lücke schliessen.
- Es existieren heute mehrere Runge-Kutta-Methoden, die automatische Schrittweitenbestimmung, Steifigkeitsdetektion sowie Interpolationsformeln unterstützen. In Anhang A ist skizziert, nach welchem Schema diese Verfahren ablaufen und wie durch *eingebettete Formeln* versucht wird, den Zusatzaufwand gegenüber der reinen Integration möglichst klein zu halten.

3.4. Signale und Segmente

3.4.1. Signalverläufe

Signale, die in einer kombinierten Simulation auftreten, haben typischerweise Verläufe, wie sie in Fig. 3.5 dargestellt sind.

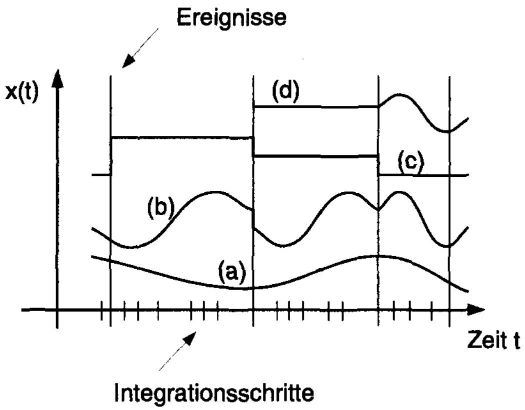


Fig. 3.5: Signalverläufe bei der kombinierten Simulation

Signal a) hat einen kontinuierlichen Verlauf, Signal b) ebenfalls, allerdings mit einem diskreten Übergang sowie einer Unstetigkeit in der ersten Ableitung. Das Signal c) scheint diskreter Natur zu sein, Signal d) ist nicht über die ganze Zeitdauer definiert und weist einen Übergang von diskret nach kontinuierlich auf.

Ist die Beschreibung der kontinuierlichen Dynamik in einem kombinierten Simulationsmodell hinreichend glatt, bzw. sind alle Stellen, an denen dies nicht zutrifft, durch explizite Formulierung von Zustandsereignissen eliminiert, so haben alle Signale innerhalb eines Integrationsintervalls einen glatten Verlauf. Auch über mehrere Intervalle hinweg trifft dies zu, solange keine Zustands- oder Zeitereignisse auftreten. Sobald dies passiert, sind verschiedene Effekte möglich, die von Unstetigkeiten des Signals oder seiner Ableitungen über Änderungen des Signaltyps bis zur Erzeugung neuer Signale bzw. zur Eliminierung bestehender Signale reichen.

In Anlehnung an Kap. 2.2 könnten Trajektorien so segmentiert werden, dass jeweils die Zeitpunkte aufeinanderfolgender Ereignisse als Start- bzw. Endpunkt eines Trajektorienstücks oder Segmentes benutzt werden. Im Hinblick auf die Implementierung scheint es aber vorteilhafter, die Information, die sich bei den einzelnen Integrationsschritten ergibt, in Segmente abzubilden.

3.4.2. Segmente

In COSIMO wird für jedes deklarierte Signal eine verkettete Liste von Segmenten angelegt, die in ihrer Gesamtheit die Entwicklung des Signals über die Zeit dokumentieren. In der Regel wird für jeden Integrationsschritt pro Signal ein neues Segment generiert. Das Signal x(t) hat zu Beginn des Intervalls $[t_0, t_1]$ den Wert x_0, am Ende den Wert x_1. Durch Interpolationsformeln (vgl. Anhang A), können die Werte von x(t)

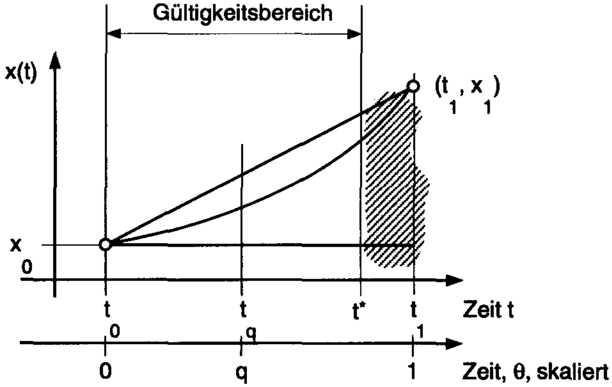


Fig. 3.6: Interpolation innerhalb eines Segments

$$x(t) \approx u\left(\frac{t-t_0}{t_1-t_0}\right) = u(\theta)$$

innerhalb des Intervalls approximiert werden. Fig. 3.6 zeigt die Approximationen der Ordnungen 0, 1 und 2.

Wird zur Zeit t* innerhalb eines Intervalls $[t_0, t_1]$ ein Zustandsereignis detektiert, so wird die Gültigkeit der Interpolationsformel auf den Bereich $[t_0, t^*]$ eingeschränkt.

Der Lebenszyklus eines Segmentes besteht aus den in Fig. 3.7 gezeigten drei Phasen:

1. Durch einen Integrationsalgorithmus werden bei jedem Schritt neue Segmente angelegt[1]; sie stehen zur Verfügung, um Zustandsereignisse zu detektieren.
2. Bereits bestehende Segmente können verwendet werden zur Rekonstruktion vergangener Signalwerte oder zur graphischen Darstellung.
3. Um während einer Simulation nicht unbeschränkt Information zu akkumulieren, zieht jedes Signal ein Fenster hinter sich her, das einen bestimmten Horizont abdeckt. Segmente, die nicht mehr im Fenster sichtbar sind, werden gelöscht.

---

[1] Bei verzugslos aufeinander folgenden Ereignissen besteht die Möglichkeit, die Zustandsübergänge 'innerhalb' eines Zeitpunkts durch Segmente zu protokollieren. Das Generieren der Segmente erfolgt dann nicht durch den Integrationsalgorithmus, sondern durch das Ereignis.

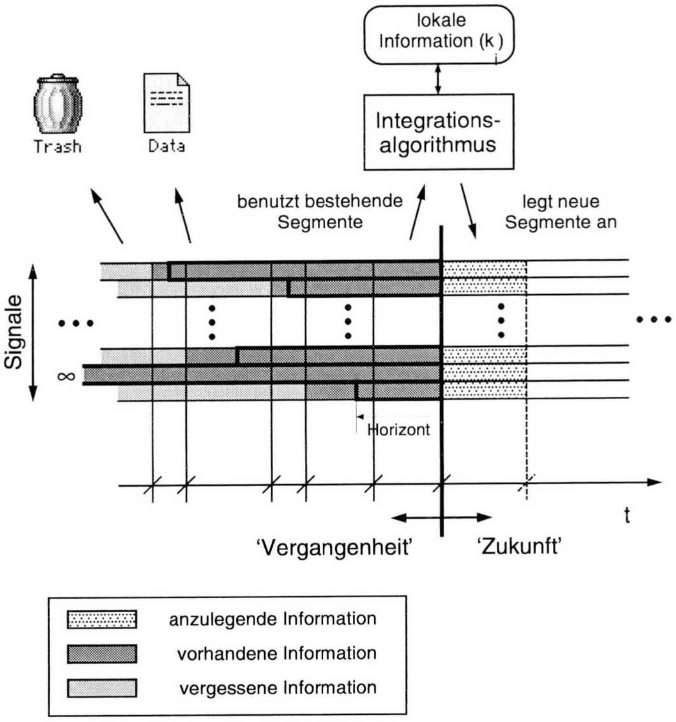


Fig. 3.7: Lebenszyklus von Segmenten

Bei einer Speicherung können verschiedene Strategien verfolgt werden:

- Die Segmente werden als solche gespeichert; die Signalverläufe werden damit vollständig dokumentiert.
- In zeitlich äquidistanten Stützpunkten berechnete Werte der Signale werden gespeichert; dies entspricht dem herkömmlichen Kommunikationsintervall.
- Es sind verschiedene Mischformen der obigen Extremvarianten denkbar, die je nach dem verfügbaren Speicherplatz und den Anforderungen an die gespeicherte Information sinnvoll sind:

  Aus Segmenten kann diejenige Information weggefiltert werden, die nur für Interpolationen hoher Ordnung benötigt wird.

  Das Zusammenlegen aufeinander folgender Segmente kann unter bestimmten Bedingungen ohne Informationsverlust erfolgen.

  Bei der Variante des Kommunikationsintervalls können Unstetigkeiten durch Einfügen zusätzlicher Stützstellen dokumentiert werden.

3.4.3. Detektion von Zustandsereignissen

Wird in einem Integrationsalgorithmus die Detektion von Zustandsereignissen explizit unterstützt, so entspricht die Implementierung meist der in Fig. 3.8 skizzierten Abfolge: Ein Integrationsschritt, der wegen Überschreitung der Fehlertoleranz zurückgewiesen wird, wird mit reduzierter Schrittweite so lange wiederholt, bis er akzeptiert werden kann.

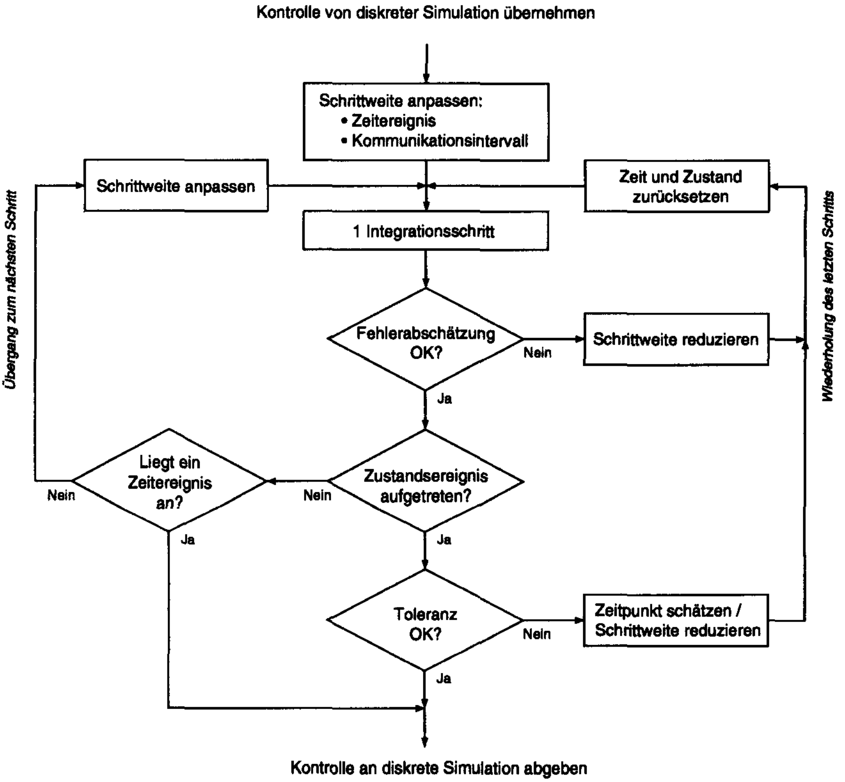


Fig. 3.8: *Detektion von Zustandsereignissen durch repetierte Integration*

Wenn innerhalb des akzeptierten Schrittes kein Zustandsereignis aufgetreten ist und am Ende des Schrittes auch kein Zeitereignis anliegt, so kann zum nächsten Schritt übergegangen werden. In den beiden Fällen, in denen kein Zustandsereignis, aber ein Zeitereignis anliegt bzw. ein Zustandsereignis innerhalb der erlaubten Toleranz aufgetreten ist (vgl. Fig. 3.9a), wird die Kontrolle an die diskrete Simulation zurückgeben. Der relativ häufige Fall, dass ein auftretendes Zustandsereignis nicht innerhalb der Toleranz liegt, führt zur Wiederholung der Integration mit reduzierter Schrittweite.

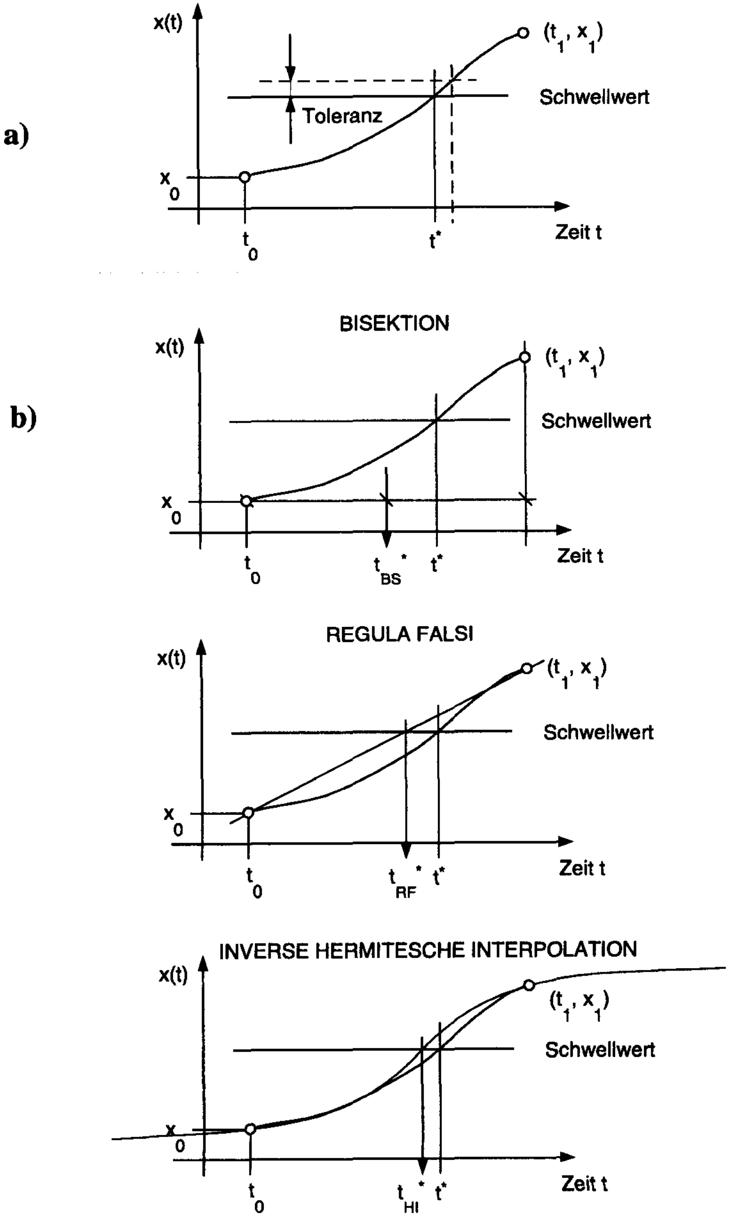


Fig. 3.9: *a) Toleranz bei Zustandsereignissen b) Schätzung von t^**

In konventionellen Verfahren wird versucht, unter Ausnutzung vorhandener Information den Zeitpunkt t* zu schätzen, in dem das Zustandsereignis aufgetreten ist und anschliessend bis zum geschätzten Punkt zu integrieren. Verfahren wie Bisektion oder Regula Falsi sind zwar einfach zu implementieren, führen aber häufig zu vielen Iterationen. Cellier [11] schlägt das Verfahren der inversen hermiteschen Interpolation vor, bei dem eine kubische Parabel benutzt wird, die beim Anfangs- und Endpunkt des Intervalls jeweils mit Funktionswert und erster zeitlicher Ableitung des Segments übereinstimmt, und so eine analytische Berechnung von t_{HI}* erlaubt. Dieses Schätzverfahren für t* ist allerdings nicht immer anwendbar.

Neben dieser Bestrebung, t* möglichst gut zu schätzen, können Integrationsroutinen mit eingebetteten Interpolationsformeln als Ergänzung gesehen werden, da sie es erlauben, den Zustand zum geschätzten Zeitpunkt effizient zu berechnen: x(t*) wird nicht durch aufwendige Nachintegration, sondern über die schnellere Interpolation gewonnen. Beim modifizierten Flussdiagramm in Fig. 3.10 wird ersichtlich, dass die

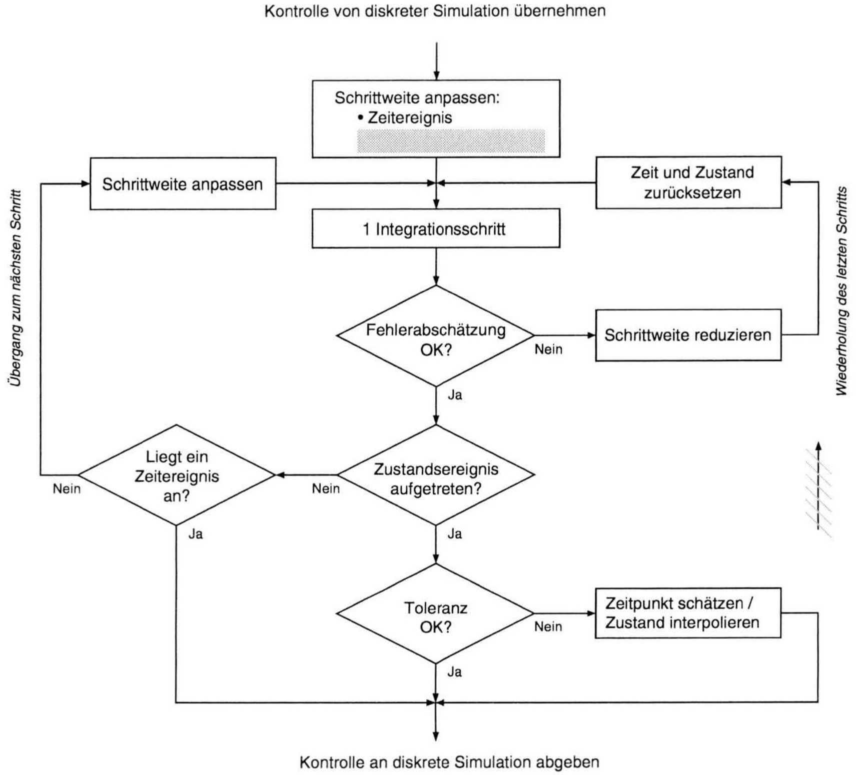


Fig. 3.10: *Detektion von Zustandsereignissen durch Interpolation*

Rückführschleife für die Wiederholung der Integration wegfällt. Das Zustandsereignis wird mithilfe von Interpolation lokalisiert; die iterative Natur des Lokalisierens bleibt dabei natürlich bestehen.

Die Schrittweitensteuerung der Integration ist somit abhängig von:

- den Eigenschaften der Differentialgleichung in der Region des momentanen Zustandes
- den im vornherein bekannten Zeitereignissen[1], auf die hinintegriert werden kann

Dagegen ist sie entkoppelt von:

- Zustandsereignissen
- Kommunikationsintervallen[2].

3.5. Implementierung des Simulators

3.5.1. Simulation auf dem Digitalrechner

Beim Versuch, Simulationsprogramme für zeitkontinuierliche Systeme auf Digitalrechnern zu implementieren, bietet sich zur Umsetzung von Problemen der Form

$$\dot{x}(t) = f(x(t), t)$$

die Eulersche Methode,

$$x(t + \Delta t) = x(t) + \Delta t \cdot f(x(t), t)$$

also die rekursive Berechnung obiger Formel, geradezu an. Der Schwerpunkt liegt hier auf der Funktion und weniger auf der Datenstruktur. In Programmiersprachen, die stark dem Algorithmischen verhaftet sind, wie z.B. ALGOL oder FORTRAN, sind Simulationen dieser Art schon früh durchgeführt worden. Den Fokus auf numerische Aspekte legend, sind viele Klassen von Integrationsalgorithmen entwickelt worden, während die Daten im wesentlichen in Vektoren und Matrizen fixer Grösse organisiert geblieben sind (vgl. Anhang A).

Die Techniken objektorientierter Programmierung sind seit ihrem ersten Aufkommen zu Zwecken der Simulation eingesetzt worden; sie sind auch wesentlich durch die diskrete Simulation motiviert und geprägt worden. Für Petri-Netze und Zustandsautomaten oder auch für Modelle, bei denen Transaktionen - das Verschieben dynamischer Objekte zwischen statischen Objekten, Protokolle oder schnell variierende Modellstruktur im Vordergrund stehen, bietet sich zur Modellierung und Simulation eine objektorientierte Umgebung an.

Zur Simulation kombinierter Modelle schliesslich sind hybride Sprachen geeignet.

---

[1] Zu den Zeitereignissen kann auch das Simulationsende gezählt werden.

[2] Die Kommunikationsintervalle können natürlich nach wie vor als Zeitereignisse formuliert werden, haben aber durch die Interpolationsinformation nicht mehr die gleiche Bedeutung.

3.5.2. Modulhierarchie des Integrationskerns

In der konkreten Implementierung stellt das Modul `Signals` Signalobjekte bereit mit darin gekapselten Listen von Segmenten. `Integrator` dient als Bindeglied zwischen diesen Objekten und Integrationsalgorithmen, wie sie z.B. in `IntegDP54` enthalten sind.

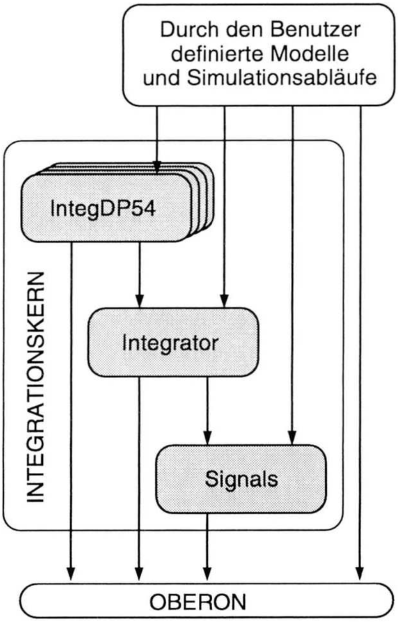


Fig. 3.11: Modulhierarchie des Integrationskerns

Definitionen von Simulationsmodellen, die auf den Integrationskern zugreifen, können

- textuell gegeben sein, wobei die Beschreibungen üblicherweise relativ kompakt in wenigen zusätzlichen Modulen untergebracht sind (Bsp. in Kap. 3.5.3 / Anhang B).
- graphisch erfolgen. Die einzelnen Komponenten des Simulationsmodells entsprechen in diesem Fall Objekten, deren Beschreibung und spezifisches Verhalten üblicherweise über mehrere Module verteilt definiert wird (Bsp. in Kap. 5).

Durch den Benutzer definierte Simulationsabläufe können sich beispielsweise darin voneinander unterscheiden, ob sie Interaktionen zulassen oder nicht.

Die Integration und die Detektion von Zustandsereignissen werden entkoppelt gelöst; die entsprechenden Routinen können unabhängig voneinander überschrieben werden.

```
MODULE Integrator

  CONST
    NoEventOccured*=0; (* Rückgabeargumente von Integ *)
    EventOccured*=-1;
    KeyInterrupt*=-2;

  TYPE
    ...
    IntegRoutine*=PROCEDURE(VAR step: LONGREAL;
                minStep, maxStep, t0, tEnd, relTol: LONGREAL): INTEGER;
    ThreshRoutine*=PROCEDURE(VAR threshOK, eventOccured: BOOLEAN;
                endOfStep: LONGREAL; VAR factor: LONGREAL);
    ...

  VAR
    ...
    Integ*: IntegRoutine;
    Thresh*: ThreshRoutine;
    time*: LONGREAL;
    ...
```

`Integrator.Integ` integriert - üblicherweise über mehrere Schritte - bis `tEnd` erreicht ist, bis ein Zustandsereignis aufgetreten ist oder bis der Benutzer die Simulation unterbrochen hat; durch das Rückgabeargument können die Fälle unterschieden werden.

COSIMO kann nach Bedarf mit zusätzlichen Integrationsroutinen versehen werden, solange sie bezüglich Parameterliste mit den bestehenden konform sind:

```
MODULE IntegDP54;
                      (* Integrationsroutine: *)
  PROCEDURE DormandPrince54*(VAR step: LONGREAL;
              minStep, maxStep, t0, tEnd, relTol: LONGREAL): INTEGER;
  PROCEDURE DP54*;    (* Wahl der aktuellen Routine (global) *)
  PROCEDURE Set*;     (* Verbinden mit einem Simulationspanel
                                                   (lokal im Kontext) *)
```

Erweiternde Routinen sollten mit zusätzlichen Befehlen versehen sein, mit denen sie entweder als globale Integrationsroutine gesetzt oder an ein Simulationspanel geheftet werden können.

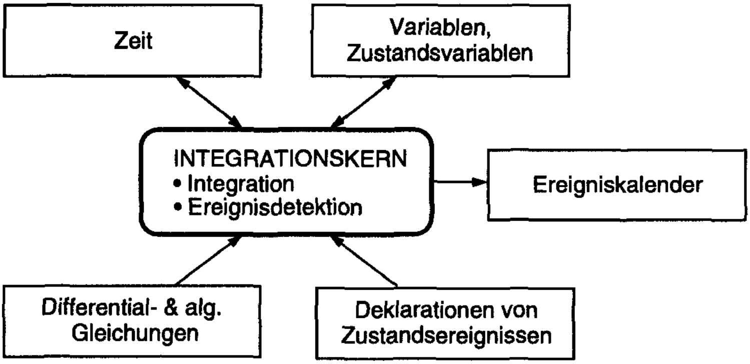


Fig. 3.12: Elemente eines Integrationkerns für kombinierte Modelle

Der Integrationskern arbeitet mit der Zeit `Integrator.time` und mit Listen von Signalen: `Signals.algSignalList` & `Signals.stateSignalList`.

```
MODULE Signals;

   CONST
      getVal*=0;
      insertSegment*=1;
      ...
   TYPE                                                 (* vgl. Anhang A: *)
      CoeffArray*=POINTER TO ARRAY OF LONGREAL;         (* alpha-Werte *)
      TimeArray*=POINTER TO ARRAY 3 OF LONGREAL;        (* t0, t1, t* *)
      KArray*=POINTER TO ARRAY OF LONGREAL;             (* k-Werte *)

      SignalMsg*=RECORD
         id*: INTEGER;
         times*: TimeArray;
         coeffs*: CoeffArray;
      END;
      Signal*=POINTER TO SignalDesc;
      SignalHandler*=PROCEDURE(signal: Signal; VAR M: SignalMsg);
      SignalDesc*=RECORD
         (* Segmente sind in den Signalen gekapselt *)
         handle*: SignalHandler;
         next*: Signal;
         x*: LONGREAL;
         xAdr*: LONGINT;
      END;
      StateSignal*=POINTER TO StateSignalDesc;
      StateSignalDesc*=RECORD
         (SignalDesc)
         derAdr*: LONGINT;
         k*: KArray;
         nextState*: StateSignal;
      END;

   VAR
      stateSignalList*: StateSignal;
      algSignalList*: Signal;

   PROCEDURE InitLists*;
   PROCEDURE DeclAlgSignal* (VAR x: LONGREAL; horizon: LONGREAL);
   PROCEDURE DeclStateSignal* (VAR x, xDot: LONGREAL;
                                                    x0, horizon: LONGREAL);
   PROCEDURE InitStateSignal* (x: LONGREAL);
   ...
```

Die kontinuierliche Dynamik, Differentialgleichungen und algebraische Gleichungen, ist in `Integrator.Dynamic` enthalten, potentielle Auslöser von Zustandsereignissen in `Integrator.threshList`. Wird innerhalb des Intervalls [`t0, tEnd`] ein - oder mehrere gleichzeitige - Zustandsereignisse detektiert, so werden sie in den Ereigniskalender `Integrator.calendar` eingetragen. Neben der Ausführungszeit kann der Verursacher eines Ereignisses dieses mit Typ und Adressat (`model`) kennzeichnen.

```
MODULE Integrator;

   TYPE
      Threshold*=POINTER TO ThresholdDesc;   (* nur POINTER wird exportiert
      Event*=POINTER TO EventDesc;
      EventDesc*=RECORD
         time*: LONGREAL;
         type*: INTEGER;
         model*: Objects.Object;
      END;
      ...
```

```
VAR
  Dynamic*: PROCEDURE;
  calendar*: Event;
  threshList*: Threshold;
  ...
```

Bei Simulationsabläufen, die auf dieser Stufe aufsetzen, besteht grosse Flexibilität. Allerdings muss dafür gesorgt sein, dass Ereignisse richtig abgearbeitet werden und die Simulation einen korrekten Fortgang nimmt. Konkret bedeutet dies, dass auftretende Ereignisse im Kalender abgeholt und weitergeleitet werden und danach die Integrationsroutine wieder aufgerufen wird.

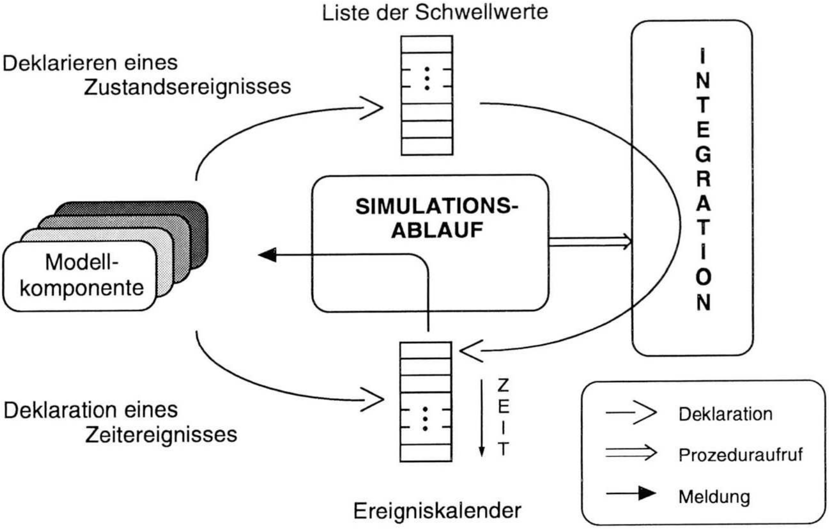


Fig. 3.13: Behandlung von Zeit- und Zustandsereignissen

```
MODULE Integrator;
   ...
   PROCEDURE InitLists*;
 (* Routinen zur Detektion von Zustandsereignissen: *)
   PROCEDURE Bisection*(VAR threshOK, eventOccured: BOOLEAN;
                        endOfStep: LONGREAL; VAR factor: LONGREAL);
   PROCEDURE SetBisection*;
   PROCEDURE RegulaFalsi*(VAR threshOK, eventOccured: BOOLEAN;
                        endOfStep: LONGREAL; VAR factor: LONGREAL);
   PROCEDURE SetRegulaFalsi*;
 (* Deklaration von Zustandsereignissen: *)
   PROCEDURE DeclTH*(VAR var, tar: LONGREAL; relTol, absTol: LONGREAL;
                      status, type: INTEGER; model: Objects.Object);
   PROCEDURE SetStatusTH*(status, type: INTEGER; model: Objects.Object);
 (* Handhabung von Zustandsereignissen / Ereigniskalendern: *)
   PROCEDURE Create*(VAR cal: Event);
   PROCEDURE Empty*(VAR cal: Event): BOOLEAN;
   PROCEDURE GetEvent*(VAR cal, x: Event);
   PROCEDURE GetNextTime*(cal: Event): LONGREAL;
   PROCEDURE PutEvent*(VAR cal: Event; x: Event);
```

3.5.3. Beispiel

Anhand des nichtlinearen Systems zweiter Ordnung

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = -sign(x_1)$$

werden Simulationen sowohl mit Standardpaketen als auch in COSIMO durchgeführt und verglichen.

Die Lösungen für das gegebene System können auf analytischem Weg ermittelt werden, wobei sich durch Eliminierung der Zeit folgende Darstellung ergibt:

$$x_2^2 + 2|x_1| = C \geq 0$$

Alle Trajektorien sind geschlossen und verlaufen entlang Parabelästen.

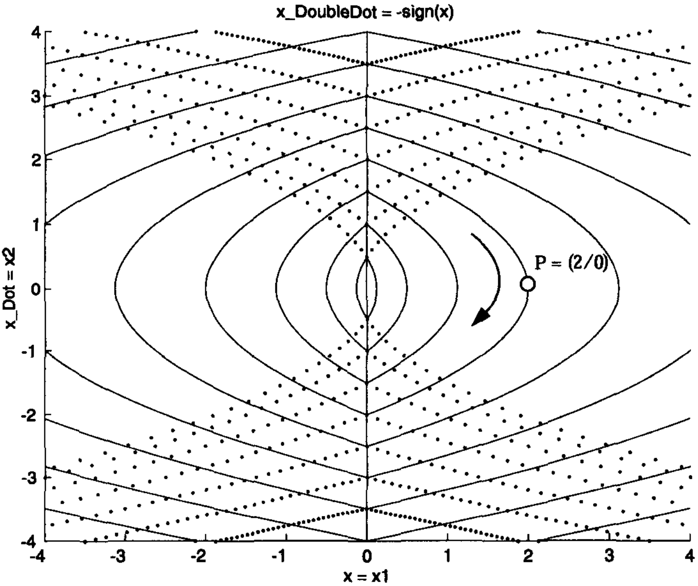


Fig. 3.14: Kurvenschar: indifferente Schwingung

Das Problem der korrekten Simulation stellt sich im wesentlichen beim Vorzeichenwechsel von x_1, d.h. zu den Zeitpunkten, an denen die Trajektorie von der einen Parabelschar auf die andere umschaltet.

Für die Tests wurde ausgehend vom Startpunkt P = (2/0) über 8 Zeiteinheiten integriert; im Idealfall wird dabei wieder der Punkt P erreicht.

Bei den kommerziellen Produkten MATLAB/SIMULINK und ACSL[1] wurde mit einer Runge-Kutta-Fehlberg-Methode gearbeitet (Ordnung 5; Fehlerabschätzung und Schrittweitenanpassung der Ordnung 4; Anzahl Funktionsaufrufe pro Schritt: 6).

Die in COSIMO benützte Runge-Kutta-Methode von Dormand und Prince hat dieselbe Ordnung, ermöglicht allerdings mit einem zusätzlichen 7. Funktionsaufruf Interpolationen der Ordnung 4.

Die Tests der verschiedenen Methoden sind soweit wie möglich vereinheitlicht worden; so z.B. wurde in ACSL das sonst gebräuchliche Integrationsintervall nicht verwendet, um die Funktionsweise der autonom arbeitenden Integrationsroutine beobachten zu können.

Details über die verwendeten Modellbeschreibungen und Parametersätze sind in Anhang B gegeben.

- In allen Paketen wurde ein Versuch durchgeführt, bei dem das Zustandsereignis nicht explizit formuliert wurde (M1, S1, A1, C1).
- In SIMULINK ist zusätzlich ein Versuch gemacht worden durch Versteifung des Systems in der Umgebung von $x_1 = 0$ (S1*).
- ACSL unterstützt die Detektion von Zustandsereignissen mit Nachintegration. Es wurde zusätzlich je ein Versuch mit/ohne explizites Umschalten der Systemgleichung durchgeführt (A2A, A2B).
- In COSIMO wurden zusätzliche Versuche gemacht mit explizit umgeschalteter Systemgleichung und Nachinterpolation (C3B). Anders als z.B. in ACSL kann die Toleranz für die Zustandsereignisdetektion unabhängig von der Integrationsgenauigkeit vorgegeben werden. Anhang B enthält zusätzliche Informationen zum Zusammenhang zwischen dem erreichten Fehler und der Anzahl Iterationen bei der Interpolation.

Paket	*Detektionsmethode*[a]	*Kurzbezeichnung*	*Integrationsschritte (akzeptierte / abgelehnte / total)*			*Fehler: $\|x(0) - x(8)\|$*
MATLAB	1	M1	25	24	49	$3.4\ 10^{-2}$
SIMULINK	1	S1	26	49	75	$1.9\ 10^{-2}$
SIMULINK	1*	S1*	28	64	92	$6.6\ 10^{-5}$
ACSL	1	A1	21	20	41	$2.2\ 10^{-2}$
ACSL	2A	A2A	19	31	50	$1.0\ 10^{-3}$
ACSL	2B	A2B	6	11	17	$5.0\ 10^{-4}$
COSIMO	1	C1	13	53	66	$4.2\ 10^{-2}$
COSIMO	3B	C3B	6	0	6	$9.2\ 10^{-6}$

[a] Die Nummern der Detektionsmethoden sind konform mit Kap. 3.2.5.

---

[1] MATLAB/SIMULINK wurde in der Version 4.2c bzw. 1.3c benutzt, ACSL in der Version 11.4.1

Modellieren mit Gadgets

4

4.1. Gadgets

Um mit COSIMO graphisch modellieren zu können, ist das Gadgets System verwendet worden, eine Komponentenarchitektur *(Componentware)*, die auf Oberon aufgesetzt ist. Im Gegensatz zu den Ideen objektorientierter Programmierung - wie sie in Anhang C kurz umrissen sind - ist die komponentenorientierte Softwareentwicklung im Bereich der Simulationstechnik noch wenig verbreitet. Deshalb werden im folgenden einige der Grundkonzepte hinter dem Gadgets System[1] erläutert.

Nach Gutknecht stellt es ein Management für persistente Objekte sowie einen Grundstock von Gadgets bereit, um graphische Oberflächen zu komponieren und zu benutzen [30]. Neben dem Gebrauch bestehender Gadgets können durch Modifizierung und Erweiterung aus bestehenden Typen eigene Versionen von Komponenten abgeleitet werden, die speziellen Bedürfnissen gerecht werden.

Komponentenorientierte Software zeigt Alternativen auf zu konventionellen monolithischen Programmpaketen, bei denen es üblich ist, dass sie mit jeder neuen Version noch etwas an Grösse und Speicherbedarf zulegen. Zwar werden Hersteller von Programmen vom Hersteller des entsprechenden Betriebssystems angehalten, sich an einen bestimmten einheitlichen Standard anzulehnen. Dies betrifft vor allem die graphischen Oberflächen, die z.B. durch Richtlinien von Apple [65] seit längerem einen Trend zur Vereinheitlichung aufweisen. Davon abgesehen werden Ressourcen, die von mehreren Programmen gemeinsam genutzt werden könnten, in der Regel aber nicht geteilt, sondern mehrere Male unabhängig voneinander entwickelt. Beispiele dafür sind Wörterbücher und Orthographiekorrektur oder Zeichnungs- und Formeleditoren, die in vielen Programmen als Supplement enthalten sind und so deren Umfang aufblähen[2].

---

[1] Mit dem Ausdruck *Gadgets System* ist die gesamte Komponentenarchitektur gemeint; *Gadgets* dagegen werden die einzelnen Komponenten genannt.

[2] Architekturen mit Einschüben (Plug-ins) lösen das Problem nur ansatzweise.

Bereits die ursprüngliche Version von Oberon wirkt dem entgegen, indem Module dynamisch geladen werden, sobald sie das erste Mal benötigt werden. So wird der Arbeitsspeicher nicht mit überflüssigem Ballast überfüllt.

Bei der Entwicklung des Gadget Systems ist darüber hinaus Wert darauf gelegt worden, eine grösstmögliche Entkopplung zwischen Programm und Oberfläche zu erreichen. Dies dient verschiedenen Zwecken:

1. Die Benutzung von Komponenten identischen Typs - z.B. von Textfeldern - auf Oberflächen, die zu verschiedenen *Programmen*[1] gehören, ist effizient, da die Komponenten nur einmal entwickelt werden müssen. Der Bedarf an Ressourcen wird auch kleiner, sobald mehrere Programme aktiviert sind, in deren Oberflächen Textfelder enthalten sind.
2. Es besteht die Möglichkeit, Oberflächen - wie üblich - durch das Programm generieren zu lassen, aber auch die wesentlich interessantere Variante, eine so generierte Oberfläche interaktiv zu ändern oder eine Oberfläche von Grund auf zu konzipieren und komponieren.
3. Die Oberfläche der Gadgets geht auch über Anforderungen hinaus, wie sie beispielsweise von Johannsen [37] für die ergonomischer Gestaltung von Arbeitsplätzen im Bereich der Mensch-Maschine-Kommunikation gefordert werden:

   > "...dass Bildsymbole und Bilder an den Arbeitskontext des Bedieners oder Benutzers angepasst werden und mindestens teilweise auch interaktiv manipulierbar sind. "

   Dadurch, dass mit Gadgets nicht nur parametrische Interaktivität erlaubt ist, sondern der ganze Aufbau einer Oberfläche verändert werden kann, ist es möglich, zu einem Program mehrere Oberflächen zu erstellen. Die verschiedenen Varianten können genutzt werden, um die Möglichkeiten der Einflussnahme auf ein Programm abzustufen und dem Benutzer anzupassen.

Der Editor, der auf der Basis des Gadgets Systems entwickelt worden ist, kann zu Zwecken der Simulation benutzt werden. Die Implementierung des Editors erfolgte allerdings unabhängig vom Verwendungszweck und steht orthogonal zum Integrationskern von Kap. 3. Deshalb erschöpft sich seine Nutzung nicht in der Unterstützung von Simulationen. Blockdiagramm- und Netzwerkstrukturen können zu beliebigen alternativen Szenarien verwendet werden, zu orchestrierten Aktionen mehrerer beteiligter Komponenten, wie sie in Kap. 6 dargestellt sind.

---

[1] Der Ausdruck *Programm* bedarf in Oberon der Erklärung: Üblicherweise werden in Oberon für ein Programm eines oder mehrere Module geschrieben, die mindestens eine parameterlose Prozedur exportieren. Das Ausführen von M.P, eines Befehls, der sich aus Modul- & Prozedurname zusammensetzt, startet ein Programm. Programme im Sinne zusammengelinkter Einheiten (*binaries*) existieren in Oberon nicht.

4.1.1. Typen von Gadgets

Die verschiedenen Typen von Gadgets können in folgende Klassen eingeteilt werden:

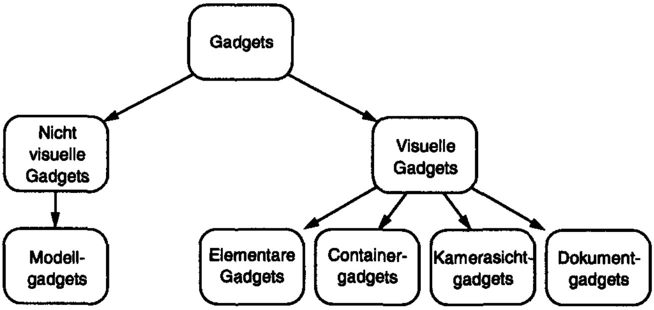


Fig. 4.1: Typen von Gadgets

Die Unterscheidung zwischen visuellen und nicht visuellen Gadgets ist motiviert durch das Konzept der *Model-View-Controller*-Separation (MVC), wie es z.B. in Smalltalk [28] enthalten ist. In Gadgets wird *View* und *Controller* zusammengefasst (visuelles Gadget, Sicht) und vom *Model* (nicht visuelles Gadget, Modell[1]) separiert.

Die Separation von Modell und Sicht erlaubt die getrennte Behandlung von Daten, die in einem Modell gekapselt sind, und der Sicht darauf. Durch ein Protokoll wird sichergestellt, dass die Sicht das Modell auf dem Bildschirm korrekt repräsentiert, dass aber auch durch Benutzereingaben über die Sicht auf das Modell einwirken können.

Die Vorteile der Entkopplung liegen darin, dass

- verschiedene Typen von Sichten in Verbindung mit demselben Typ von Modell verwendet werden können,
- simultan verschiedene Sichten am selben Modell hängen können (vgl. Fig. 4.2),
- Programme, die auf der Benutzeroberfläche im wesentlichen nur Modelle (keine Sichten) ansprechen, von Änderungen der Sichten relativ unabhängig sind.

Die visuellen Gadgets werden weiter aufgegliedert in

- elementare Gadgets: atomar, z.B. Textfelder, Schieber,
- Container-Gadgets: erlauben die hierarchische Verschachtelung visueller Gadgets,
- Kamerasicht-Gadgets: erlauben verschiedene Sichten (Ausschnitte, Perspektiven),
- Dokument-Gadgets: sind mit Namen versehene speicherbare Einheiten.

---

[1] Der Ausdruck Modell, wie er hier verwendet wird, soll grundsätzlich nicht mit dem in Kap. 2.1 eingeführten Modellbegriff verwechselt werden. obwohl sich die beiden Begriffe in der vorliegenden Implementierung weitgehend überlappen und im folgenden nicht mehr explizit auseinandergehalten werden.

4.1.2. Handler und Meldungen

Alle beschriebenen Klassen von Gadgets stammen von einem gemeinsamen Grundtyp `Objects.Object` ab, der einen Handler besitzt. Der Handler entspricht einer Art Universalmethode, die Meldungen (*Messages*) verarbeiten kann:

```
TYPE
   Object*=POINTER TO ObjDesc;

   ObjMsg*=RECORD
      ...
   END;

   Handler*=PROCEDURE(obj: Object; VAR M: ObjMsg);

   ObjDesc*=RECORD
      ...
      handle*: Handler;
   END;
```

Durch Aufruf von

```
thisObj.handle(thisObj, thatMsg);
```

wird dem Objekt `thisObj` eine Meldung `thatMsg` geschickt, beziehungsweise dem Handler des Objekts `thisObj` das Parameterpaar `(thisObj, thatMsg)` übergeben.

Die Klassenhierarchien der Objekte sowie der Meldungen werden weitgehend parallel entwickelt. In den Handlern spezialisierter Objekttypen kann zwischen verschiedenen Meldungstypen unterschieden werden:

```
PROCEDURE MyHandler*(obj: Object; VAR M: ObjMsg);
BEGIN
   IF M IS CopyMsg THEN ...
   ELSIF M IS StoreMsg THEN ...
   ELSIF M IS LoadMsg THEN ...
   ELSE ...
   END
END MyHandler;
```

Diese Handhabung von Meldungen unterschiedlichen Typs ist direkt umsetzbar in Methoden, wie sie z.B. in Oberon-2 verwendet werden, indem für jeden Meldungstyp eine Methode geschrieben wird.

Ein Vorteil bei der Verwendung von Meldungen liegt darin, dass jederzeit neue Meldungsvarianten durch Typerweiterung hinzugefügt werden können. Diese Flexibilität, die sich bei der Verwendung von Meldungen anstelle von Methoden ergibt, wird bezahlt durch gewisse Nachteile bzw. Unsicherheiten:

- Die Hierarchie der Meldungstypen muss im Handler durch explizite Programmierung widergespiegelt werden. Wird darauf nicht geachtet, so kann die Meldung nicht korrekt entgegengenommen werden.
- Das zu einer Meldung gehörende Protokoll muss separat dokumentiert werden, da der entsprechende Quellcode stark dezentralisiert ist: typischerweise durchläuft eine Meldung eine Vielzahl von Handlern, von denen jeder die Meldung, bzw. ihre Felder, bearbeiten kann.
- Zur Laufzeit ergibt sich ein zusätzlicher Zeitaufwand durch Fallunterscheidungen.

4.1.3. Hierarchie in der Darstellung

Oberon System 3 basiert in den obersten Stufen der Displayhierarchie auf den rechteckigen Tracks und Viewers, die im Grundsystem implementiert sind. Allerdings erlaubt System 3 neben transparenten Objekten auch eine beliebige lokale Überlappung.

Gadgets können entweder unter Programmkontrolle oder interaktiv komponiert werden. Die Struktur des Displayraums entspricht einem gerichteten azyklischen Graphen (*DAG*). Fig. 4.2 zeigt ein Panel, auf das verschiedene Komponenten montiert sind, sowie den entsprechenden Teil des Graphen. Auf dem inneren Panel sind drei Sichten auf ein gemeinsames Modell enthalten.

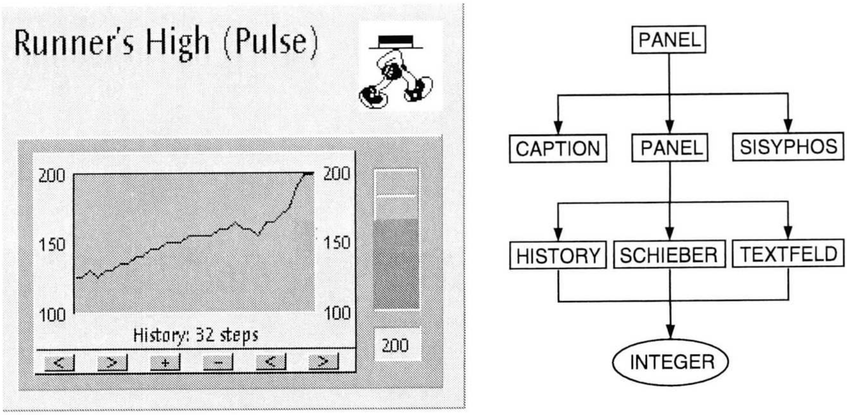


Fig. 4.2: *Darstellung eines Panels als gerichteter azyklischer Graph*

Wird durch Bewegen des Schiebers oder durch Editieren des Textfelds eine Sicht verändert, so schickt sie eine entsprechende Meldung direkt an ihr Modell. Das Modell, das wiederum dafür sorgen muss, dass *alle* Sichten synchronisiert werden, erreicht dies über eine *Broadcast*-Meldung. Dabei wird die Meldung nicht explizit adressiert, sondern an die Wurzel des DAG geschickt, von wo sie auf rekursivem Weg zu allen beteiligten Sichten gelangt.

Bei der Implementierung eines Broadcast-Mechanismus müssen verschiedene Effekte berücksichtigt werden:

- Es können mehrere Pfade zu einem Objekt führen, wie z.B. zum INTEGER-Modell in Fig. 4.2.
- Der Pfad zu einem Objekt kann dynamisch ändern; z.B. wenn der SISYPHOS auf das innere Panel verschoben und von diesem konsumiert wird.

In solchen Fällen muss sichergestellt sein, dass die Bearbeitung von Meldungen im richtigen Zeitpunkt abgebrochen wird.

4.1.4. Weitere Konzepte des Gadgets Systems

Elterliche Kontrolle

Dies bezieht sich auf den DAG; Eltern, hierarchisch höher angesiedelte Gadgets, haben Kontrolle über ihre Kinder. Der Meldungsfluss zu den Kindern wird von den Eltern kontrolliert und gefiltert - mindestens soweit, wie Meldungen nicht direkt an die Kinder adressiert sind. Andererseits sind die Eltern angehalten, ein Minimalprotokoll zu unterstützen.

Kapselung

Von hierarchisch geschachtelten Komponenten wird im allgemeinen nur die äussere Hülle als Objekt wahrgenommen; die darin gekapselten Komponenten werden nicht direkt angesprochen, da sie der elterlichen Kontrolle unterliegen. Bei Änderungen in der Implementierung der gekapselten Objekte wird damit gewährleistet, dass das bestehende Protokoll beibehalten werden kann.

Abstrakte Attribute und Links

Die Deklaration von Typen umfasst neben Feldern für Methoden auch solche für Daten und für Verbindungen zu anderen Objekten. Diese Felder sind häufig nicht direkt einsehbar, sondern gekapselt; für Inspektion und Modifikation ihrer Inhalte sind zwei Meldungstypen implementiert: *AttrMsg* für die eigentlichen Daten (Attribute), *LinkMsg* für die Verbindungen (Zeiger). Durch ein spezielles Werkzeug *(Columbus)* wird der Umgang mit Daten und Zeigern auf der graphischen Oberfläche ermöglicht.

Adaptivität

Idealerweise sollen Komponenten sich unabhängig von der konkreten Verwendung, für die sie gebaut worden sind, in verschiedenen Umgebungen benutzen lassen. Dieses Prinzip ist um so schwieriger umzusetzen, je mehr Randbedingungen durch gegenseitige Abhängigkeiten zwischen verschiedenen Komponenten auftreten. So ist der in Kap. 4.2 beschriebene Editor auf einem Standardpanel nicht voll funktionsfähig. Die einzelnen Komponenten, wie z.B. die Verbindungen, können aber durchaus zu Dekorationszwecken benutzt werden.

Persistenz

Persistenz verhilft einem Objekt dazu, portierbar zu sein. Die Möglichkeiten, dasselbe Objekt in verschiedenen Applikationen zu benutzen, sind auf gängigen Betriebssystemen relativ langsam weiterentwickelt worden. Die Integration von Operationen wie Kopieren und Einsetzen (Copy & Paste), Ziehen und Loslassen (Drag & Drop) oder von Techniken wie OpenDoc und OLE zielt in Richtung Persistenz.

Gadgets - persistente Objekte - haben genügend Kenntnisse über sich selbst, um z.B. vollständige Kopien von sich anzulegen oder sich nach dem Speichern korrekt wiederherzustellen. Portierbare Dokumente *(self contained documents)* sind eine konsequente Weiterentwicklung dieser Ideen. Sie basieren darauf, dass ein plattformunabhängiger Objektcode an Dokumente angehängt werden kann, der beim Öffnen der Dokumente bei Bedarf vollständig kompiliert werden kann.

4.1.5. Ebenen der Programmierung

Wie bereits eingangs des Kapitels erwähnt, werden in Oberon Befehle ausgeführt, indem Zeichenketten vom Format

```
M.P
```

angeklickt werden. Dabei ist `M` ein Modulname und `P` der Name einer formal parameterlosen Prozedur, die vom Modul `M` exportiert wird. Bei Befehlen wie

```
System.Copy
```

wird der Kontext, also die Umgebung, aus der heraus der Befehl aktiviert worden ist, als Parameter verwendet. Weitere - z.B. numerische - Parameter können in der Form

```
M.P par1 par2 ... parN
```

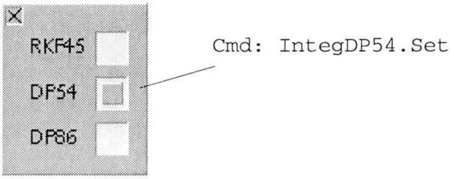
übergeben werden, wobei sie entsprechend interpretiert werden müssen. Korrekte Parametersätze können z.B. mittels EBNF definiert werden [64]:

```
System.Free {moduleName [*]} (~|^)
```

Diese Konventionen gelten auch im Gadgets System. Darüber hinaus ist es mit Gadgets möglich, die graphische Oberfläche zu verschiedenen Graden zu nutzen: vom Einkleiden der textuellen in eine graphische Oberfläche bis hin zur Programmierung eigener Gadgets.

Ebene 0

Die Programmierung auf der untersten Ebene spielt sich ab im Bereich graphischer Komposition. Zu Modulen werden interaktiv Oberflächen erstellt, Befehle und Parameter in die Oberfläche eingebettet. Durch Aktivieren von Komponenten können die dahinter verborgenen Befehle ausgeführt werden. Eine Makrosprache erlaubt die einfache Verknüpfung mehrerer Komponenten; dies wird genutzt, um Befehle und Parameter auf verschiedenen Elementen unterzubringen.



```
Cmd: IntegDP54.Set
```

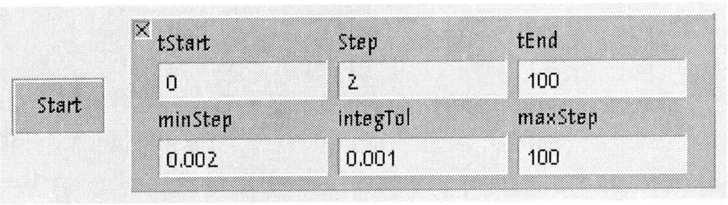
Fig. 4.3: Programmieren auf Ebene 0: Setzen einer Integrationsroutine

Fig. 4.3 zeigt das Setzen der Integrationsmethode aus einer Auswahl von drei Möglichkeiten. Die Komponenten (radio buttons) sind dabei so konfiguriert, dass nur die zuletzt getroffene Wahl angezeigt wird.

In Modulen, die auf diese Weise mit einer graphischen Oberfläche versehen sind, muss das Wissen um das Vorhandensein einer Oberfläche bzw. deren konkrete Ausgestaltung nicht vorhanden sein; das Programmieren auf Ebene 0 findet statt auf der Basis fertig definierter Modulschnittstellen.

Ebene 1

Auf Ebene 1 werden durch textuelle Programmierung Verbindungen zwischen Komponenten hergestellt. Komponenten können via ihren Namen im Kontext gesucht, ihre Attribute eingelesen und auch modifiziert werden. Der Programmierer braucht den Meldungsfluss nicht zu kennen; er kann mit Prozeduren arbeiten.



tStart	Step	tEnd
0	2	100
minStep	**integTol**	**maxStep**
0.002	0.001	100

Fig. 4.4: *Programmieren auf Ebene 1: Simulationsparameter*

In Fig. 4.4 ist hinter der Schaltfläche der Befehl 'SimuCore.Start' verborgen. Das Aktivieren dieses Startbefehls hat verschiedene Aktionen zur Folge:

- zusammensuchen der nötigen Simulationsparameter im Kontext,
- ergänzen fehlender Parameter durch voreingestellte Werte,
- ersetzen offensichtlich falsch gewählter Parameter durch plausiblere,
- ersetzen der Aufschrift 'Start' durch 'Pause',
- ersetzen des Befehls 'SimuCore.Start' durch 'SimuCore.Pause',
- starten der Simulation.

Ebene 2

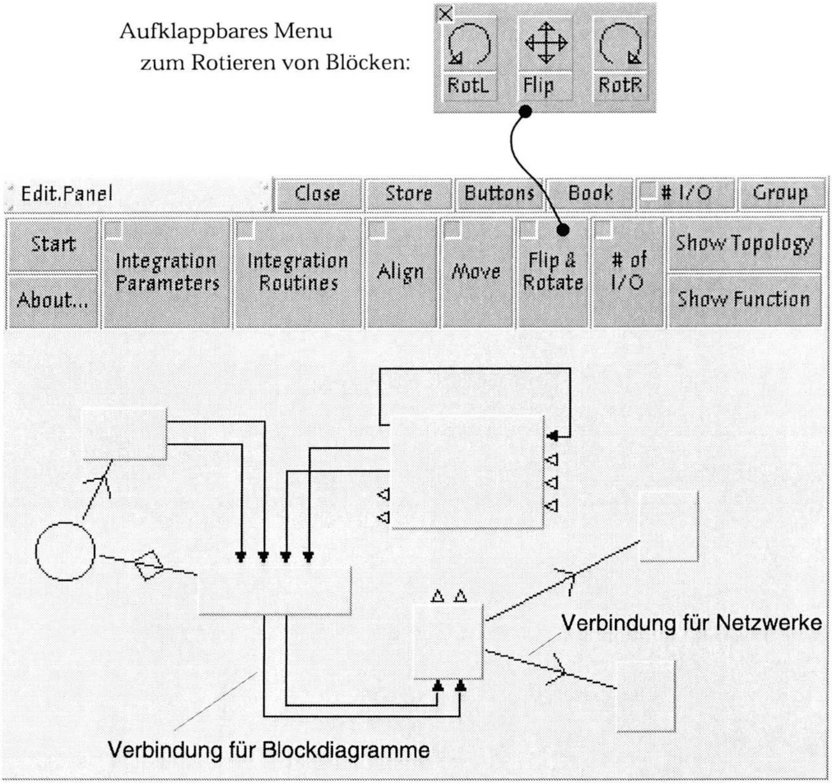
Die Programmierung auf Ebene 2 besteht im Erweitern bereits existierender Typen von Gadgets. Bestehende Meldungsprotokolle können unverändert übernommen, modifiziert oder ergänzt werden.

Der im folgenden beschriebene Editor ist - wie auch die darauf bauenden Szenarien - zu grossen Teilen auf dieser Ebene programmiert worden. Der Editor seinerseits erschliesst wiederum neue Möglichkeiten auf unteren Ebenen: so können durch rein graphische Komposition neue Interaktionsmuster zwischen verschiedenen Komponenten erzeugt werden (Fig. 4.5).

4.2. Implementierung eines Editors

4.2.1. Typen von erweiterten Gadgets

Die Module `EditCore`, `EditFrames` und `EditLinks` enthalten die nötigen Elemente eines Editors für Netzwerke und Blockdiagramme. Dieser Editor ist völlig unabhängig vom Integrationskern, der in Kap. 3 beschrieben ist.


Aufklappbares Menu
zum Rotieren von Blöcken:

Verbindung für Netzwerke

Verbindung für Blockdiagramme

Fig. 4.5: Panel zum Editieren von Blöcken und Knoten

Die in Fig. 4.5 arrangierten Komponenten haben keinerlei Simulationsfähigkeiten. Sie unterstützen lediglich das Editieren, sowie einige Grundprotokolle, die zur Simulation, aber auch für andere Szenarien, eingesetzt werden können.

Traditionellerweise ist es in der Regelungstechnik üblich, verschiedene Komponenten eines Modells in Blockdiagrammen aufzuzeichnen, wobei die Signale auf den Verbindungen permanent vorhanden sind. Entsprechend hat es sich in der Steuerungstechnik eingebürgert, Zustandsautomaten oder Petrinetze als Netzwerke von Knoten zu zeichnen, wobei die Signale nur zu diskreten Zeitpunkten über die Verbindung laufen. In COSIMO ist der Benutzer frei, wie er die verschiedenen Darstellungsformen benutzt; Konvention ist allerdings, dass für die Datenflüsse (pseudo-)kontinuierlicher Modelle Blockdiagramme verwendet werden, für die Kontrollflüsse diskreter Modelle Netzwerke.

Sichten

Im Modul `EditFrames` sind Modifikationen verschiedener Sichten enthalten. Die Änderungen betreffen im nur die Handler, die Meldungen bestimmter Typen an die Modelle weiterleiten. Der Hauptanteil zusätzlich implementierter Funktionalität liegt in den Modellen, an die die Sichten angehängt werden.

`EditCore` enthält Dokument- & Containertypen. Diese bieten massgeschneiderte Menuversionen an und unterstützen editorspezifische Arbeitsschritte.

Modelle

In `EditCore` ist ein Modelltyp enthalten, der mit den beiden Feldern `frame` sowie `panel` das Prinzip des azyklischen Graphen durchbricht. Diese Felder werden nur temporär benutzt und erlauben durch Vermeiden von Broadcast-Meldungen während der Simulation eine zeitliche Optimierung.

```
Model* = POINTER TO ModelDesc;
ModelDesc* = RECORD (Gadgets.ObjDesc)
  frame*: Gadgets.Frame;        (* link zur Sicht des Modells *)
  panel*: Panels.Panel;         (* link zum Container der Sicht *)
  next*: Model;
END;
```

Darauf aufbauend stellt `EditLinks` Modelle zur Verfügung, die miteinander verbunden werden können:

```
Model* = POINTER TO ModelDesc;
ModelDesc* = RECORD (EditCore.ModelDesc)
  inListOfNodes*, outListOfNodes*: ConnList;
  NI*, NO*: INTEGER;          (* Anzahl benutzter Blockverbindungen*)
  in*, out*: ARRAY 10 OF BlockConn;
END;
```

NB: In der Regel steht hinter allen Sichten (Blöcken/Knoten) ein *separates* Modell; ein Modell hat also nur eine Sicht. Somit können die zu einem Modell gehörenden Verbindungen eindeutig bei der entsprechenden Sicht eingezeichnet werden; ebenso sind die Felder `frame` und `panel` eindeutig.

Verbindungen

`EditLinks` stellt auch die Verbindungstypen zur Verfügung. Bei der Initialisierung eines Modells können Restriktionen angegeben werden

```
PROCEDURE SetIO*(obj: Model; dir, NI, NO: INTEGER;
  fixedInBlocks, fixedOutBlocks, fixedInNodes, fixedOutNodes: BOOLEAN);
```

die festlegen, ob die Anzahl Verbindungen eines bestimmten Typs veränderbar ist. Darüber hinaus können auch Regeln vorgegeben werden, nach denen nur bestimmte Typen von Knoten miteinander verbunden werden dürfen.

Hierarchien

In Simulationsmodellen besteht häufig der Bedarf, zueinander gehörende Modellkomponenten zusammenzufassen und durch Verstecken der Details platzsparend und übersichtlich darzustellen.

Hierarchische Organisation im Gadgets System geht mit einer flächenmässig zunehmend kleineren Erscheinung der Objekte einher, je weiter unten sie in der Hierarchie angesiedelt sind. Deshalb ist darauf geachtet worden, dass Gruppen von Modellkomponenten auf verschiedene Weise expandiert werden können:

- im richtigen Umfeld durch Umklappen
- platzsparend durch das Öffnen eines eigenes Dokuments:

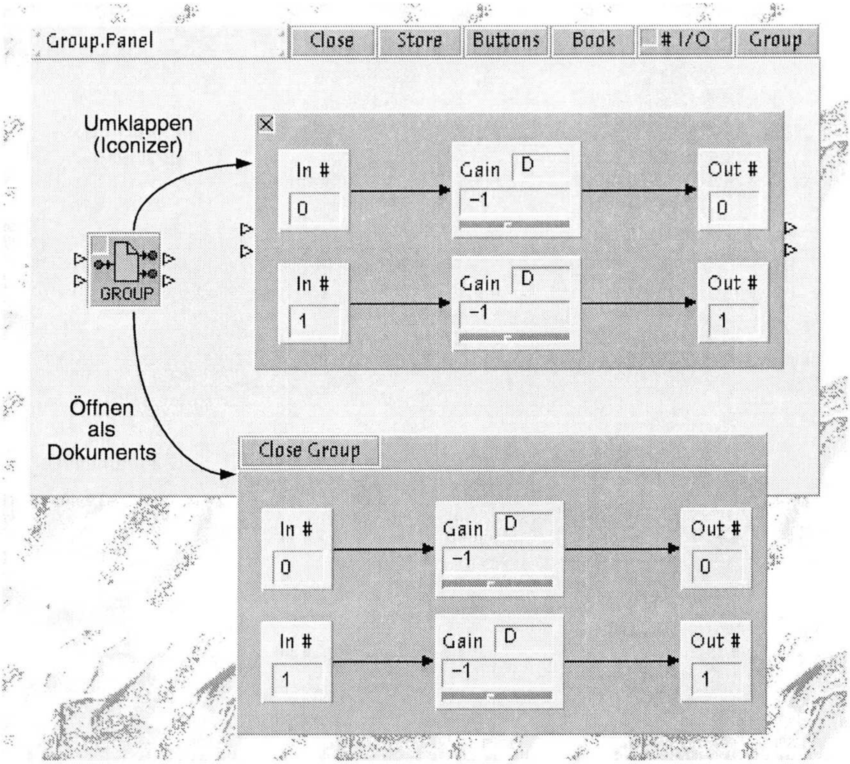


Fig. 4.6: *Hierarchie in einem Blockdiagramm*

Für Netzwerkverbindungen ist keine hierarchische Verschachtelung implementiert.

In der gleichen Art wie die gezeigte Blockgruppe können auch Sichten von Modellen, deren graphische Konfigurierung platzraubend ist, aufgefaltet werden. Ein Beispiel dafür ist das Fuzzy-Modell in Fig. 5.7.

4.2.2. Protokolle des Editors

Blöcke und Knoten werden durch graphische Verbindungen miteinander verknüpft. Dies bringt mit sich, dass zwischen Blöcken, Knoten, aber auch Verbindungen, die sich im selben Container befinden, viele Interaktionen stattfinden. So sind in Fig. 4.7 ausgehend von zwei verbundenen Blöcken A & B Nachfolgekonstellationen gezeigt für a) die Rotation von Block B, b) das Verschieben von Block B, c) das Löschen der Verbindung und d) das Löschen von Block B. Da die Protokolle in Gadgets relativ stark auf die vertikale Traversierung von Hierarchien ausgerichtet sind, wurde ein Container-Gadget geschaffen, das die Kommunikation zwischen den verschiedenen editorspezifischen Sichten und Modellen, die in ihm enthalten sind, unterstützt. Im wesentlichen sind es zwei Meldungstypen, die die zusätzliche Funktionalität erbringen:

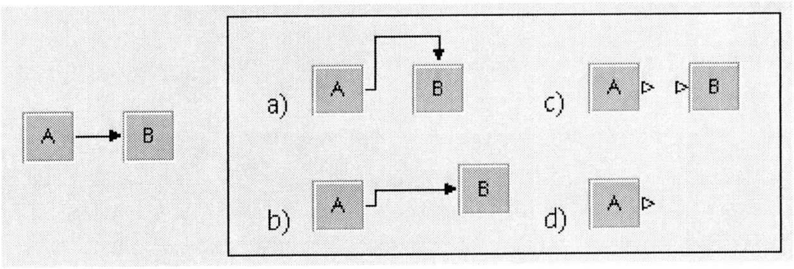


Fig. 4.7: Editieraktionen und Konsistenz

- `EditCore.CoreMsg` ist zuständig für die editorspezifischen Aufgaben wie das Erstellen, konsistent Halten und Löschen von Verbindungen.
- `EditCore.BroadMsg` ermöglicht einen ersten Schritt hin zur Implementierung von Szenarien wie dem der Simulation.

4.2.3. EditCore.CoreMsg

id = getModel

Gegenseitiges Zuordnen von Sicht und Model;

NB: für den Editor ist pro Modell nur eine Sicht relevant.

id = refreshFrame

Erneuern aller Verbindungen, z.B. nach dem Verschieben oder Rotieren von Blöcken.

id = finishFrame

Fertigstellen der Verbindungen, wenn eine Sicht mit einem Modell verbunden wird.

id = finishDelete

Aufräumen beim Löschen von Elementen.

id = selLeftButton

Weiterleiten des Zustandes {selektiert, linke Maustaste gedrückt} von der Sicht zum Modell; löst üblicherweise eine Rotation aus.

id = leftButton

Weiterleiten des Zustandes {linke Maustaste gedrückt} von der Sicht zum Modell; kann zum Erstellen einer Netzwerkverbindung führen.

id = establishLink

Wird benutzt zur Anfrage, ob eine Netzwerkverbindung mit einem speziellen Typ von Knoten etabliert werden kann.

id = changeIn / changeOut

Ändern der Anzahl der Blockdiagrammverbindungen.

id = initGUI

Wird benutzt, um Sicht und Modell zu synchronisieren.

id = showFunc / showTopo

Anzeige von Funktion und Topologie der einzelnen Komponenten.

4.2.4. EditCore.BroadMsg

Um alle Modellkomponenten und Verbindungen, die sich - auch hierarchisch verschachtelt - innerhalb eines Containers befinden, zu erreichen, kann eine Art begrenzter Broadcast-Meldung innerhalb des Containers verschickt werden. Durch die Begrenzung des Empfängerkreises wird nicht nur Zeit gewonnen, sondern auch vermieden, dass Seiteneffekte auftreten wie z.B. die Störung einer bereits laufenden Simulation.

id = resetConnections

Wird durch `EditCore.ResetConnections(panel)` ausgelöst; die Signale aller Verbindungen werden als nicht verfügbar gekennzeichnet. In einer darauf folgenden Initialisierungsphase können Modellkomponenten, die auf gemeinsamen Verbindungen bzw. Signalen operieren, bezüglich der Reihenfolge der Abarbeitung koordiniert werden.

id = collectModels

Kann durch `EditCore.CollectModels(panel)` ausgelöst werden; die Modelle tragen sich in die globale Liste `EditCore.modelList` ein. Die Hierarchie wird dabei flachgedrückt.

4.2.5. Modulhierarchie des Editors

Die drei Grundmodule des Editors stellen sicher, dass zwischen Modellen, Sichten und ihrem Container die Kommunikation funktioniert, so dass die Konsistenz von graphischer Erscheinung und Modellstruktur gewährleistet ist. Durch die von Modul `EditCore` zur Verfügung gestellten Funktionen, insbesondere das Initialisieren von Verbindungen sowie das Einsammeln von Modellkomponenten, können verschiedene Szenarien wie z.B. dasjenige der Simulation entwickelt werden.

Bei Bedarf werden in einem Szenariomodul Erweiterungen des Containers sowie spezielle Meldungen deklariert. Sichten und Modelle werden im allgemeinen getrennt voneinander weiterentwickelt. Es ist kein (gegenseitiger) Import der Module von Sicht und Modell notwendig; statt dessen wird Gebrauch von der gemeinsamen Basis gemacht.

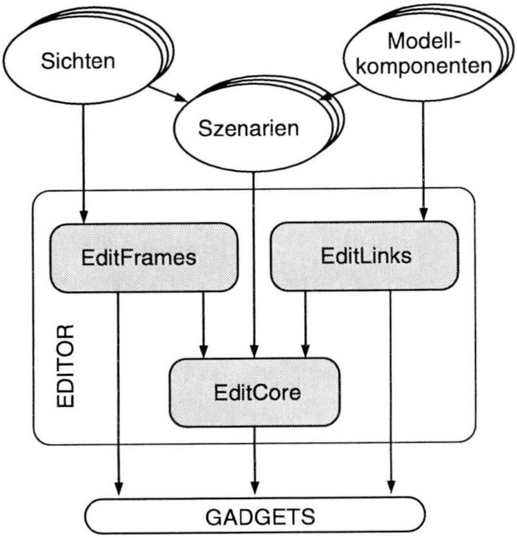


Fig. 4.8: Modulhierarchie Editor

Simulation komponierter Modelle

5

Zur Simulation von Modellen mit komplexer Struktur empfiehlt es sich, durch Bildung von Objekten einzelne Komponenten voneinander abzugrenzen. Diese Kapselung von Teilen des Modells ermöglicht

- eine flexible Komposition von Modellen,
- das Anlegen von Bibliotheken mit Modellkomponenten,
- gezieltes Austesten von Komponenten,
- Modifikationen des Verhaltens durch Erweiterung von Komponenten.

Die Bildung von Komponenten ist bereits in einer textuell orientierten Umgebung wie der Grundversion von Oberon vorteilhaft. Die einzelnen Komponenten bauen dabei auf den Funktionen eines Integrationskerns auf, wie er in Kap. 3 gezeigt ist.

In einer weiteren Stufe ist die Einbindung von Modellkomponenten in eine graphische Oberfläche sinnvoll. Dies bedeutet

- eine bedienerfreundlichere Komposition von Simulationsmodellen,
- eine bessere Visualisierung von Zuständen und ihrer Entwicklung,
- erweiterte Möglichkeiten der Interaktion.

Graphisch verwendbare Komponenten verhalten sich bezüglich der Kommunikation mit dem Integrationskern gleich wie rein textuell benutzte Objekte; darüberhinaus unterstützen sie Protokolle, die die gegenseitigen Beziehungen und ihre Repräsentation auf der graphischen Oberfläche betreffen.

Im folgenden werden Phasen skizziert, die in der Simulation, aber auch in anderen Szenarien, eine Rolle spielen. Darauf aufbauend werden Beispiele gegeben von Modellkomponenten. Interaktionen zwischen Komponenten und der graphischen Oberfläche, dem Integrationskern und mit anderen Komponenten werden beschrieben.

5.1. Unterstützung von Szenarien

Die Implementation des Editors bietet eine Basis, um innerhalb eines Containers eine Simulation oder ein anderes Szenario ablaufen zu lassen: es besteht die Möglichkeit, auf alle Modellkomponenten zuzugreifen und sie in einen geordneten Ablauf miteinzubeziehen.

5.1.1. Phasen von Szenarien

Die Modellkomponenten, die in einem Szenario mitwirken, durchlaufen während dessen Ablauf typischerweise die folgenden Phasen:

- Initialisieren
- Evaluieren / Propagieren
- Visualisieren
- Aufbereiten von Information

Initialisieren

Während der Initialisierung werden von den einzelnen Komponenten Aktionen wie Deklarationen oder das Setzen von Anfangswerten ausgeführt. Daneben müssen auch häufig die Beziehungen zwischen verschiedenen interagierenden Komponenten geordnet werden. Insbesondere ist es von Interesse, bestimmte Reihenfolgen festzulegen, die in anderen Phasen, z.B. beim Evaluieren, immer wieder unverändert Verwendung finden.

Beispiele dafür sind Blockdiagramme (Fig. 5.1) oder Schemata elektrischer oder elektronischer Schaltungen. Um für einen Integrationsalgorithmus verwendbar zu sein, müssen vielfach die Gleichungen, die die Dynamik beschreiben, bzw. die graphischen Elemente, die die Gleichungen repräsentieren, kausal geordnet werden [18], [36]. So

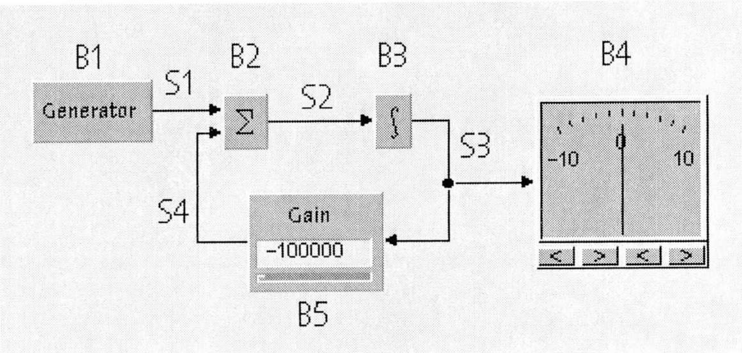
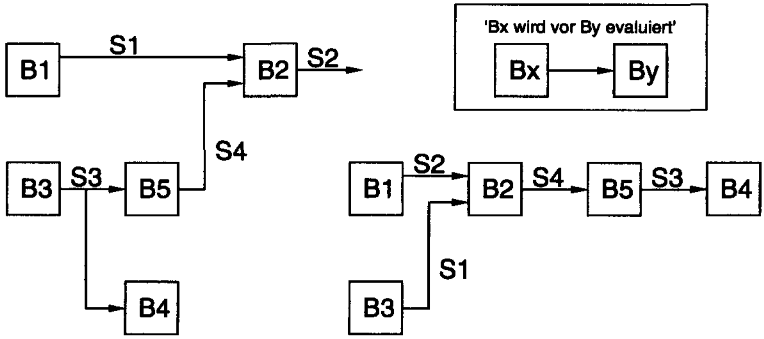


Fig. 5.1: Zu sortierende Komponenten B1 - B5

kann z.B. die Reihenfolge $\{B_1, B_3, B_5, B_2, B_4\}$ in einem Integrationsalgorithmus benutzt



Sortiert zur Integration

Sortiert zur Bestimmung von Gleichgewichtslagen

Fig. 5.2: Sortierte Modellkomponenten

werden, zur Bestimmung von Gleichgewichtslagen dagegen nicht. Es ist sinnvoll, während der Simulation ein Umsortieren zuzulassen, ein erneutes Sortieren vor jeder Evaluationsphase ist aber wenig effizient.

Auch in Simulationspaketen, die eine implizite Formulierung der Gleichungen zulassen, ist eine solche Vorverarbeitungsstufe sinnvoll, da dabei häufig das Gleichungssystem reduziert werden kann.

Evaluieren / Propagieren

Während eines Szenarios verändert sich der interne Zustand einer Modellkomponente. Dies kann im wesentlichen auf drei Arten erfolgen:

1. Die Komponenten werden von einer zentralen Instanz in einer bestimmten Reihenfolge aktiviert, damit sie bestimmte anliegende Eingangsgrössen auswerten und als Ausgangsgrössen ausgeben. Üblicherweise werden Signalflüsse in Blockdiagrammen so gehandhabt; die Reihenfolge der beteiligten Komponenten bleibt weitgehend erhalten.
2. Die Komponenten geben untereinander direkt Meldungen weiter, die als Ein- und Ausgang interpretiert werden können. Sie organisieren sich selbst, der Meldungsfluss - sowohl ihr Inhalt als auch die Adressaten - hängt von Typ und Zustand der Meldungen wie auch der Komponenten ab.
3. Ist Interaktion zugelassen, kann der Benutzer über die graphische Oberfläche Einfluss nehmen auf die Daten von Modellkomponenten. In bestimmten Fällen ist es sinnvoll, solche Benutzereingaben auch durch eine Modellkomponente emulieren zu lassen (vgl. Kap. 6.1).

Visualisieren

Der Zustand, den eine Modellkomponente über die Simulationsdauer annimmt, muss nicht in jedem Zeitpunkt ein gültiger Zustand sein. Während der Ausführung einer Integrationsroutine werden z.B. fiktive Zustände als Zwischenresultate gespeichert.

Deshalb wird nicht jede Zustandsänderung a priori zum Anlass genommen, die Sicht auf ein Modell zu erneuern. Visualisiert wird nur in bestimmten Zeitpunkten.

Aufbereiten von Information

Diese Phase dient dazu, Informationen, die sich über längere Zeit angesammelt haben, zu filtern und verdichten. Nicht mehr benötigte Information kann eliminiert werden, das Abspeichern von Signalen bzw. Segmenten z.B. gemäss Kap. 3.4.2 erfolgen.

5.1.2. Phasen von Simulationen

Ein Simulationslauf wird typischerweise durch das Initialisieren (A) eröffnet und durch die Informationsaufbereitung (D) abgeschlossen. Dazwischen wird eine Anzahl Schritte ausgeführt, die sich aus mehrmaligem Evaluieren (B) und abschliessender Visualisierung (C) zusammensetzen:

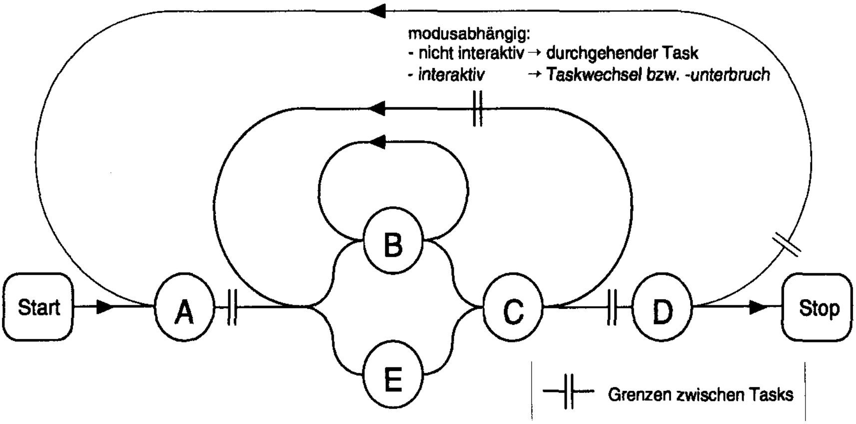


Fig. 5.3: Phasen der Simulation

Das Abarbeiten eines Ereignisses (E) ist eine Alternative zur Evaluationsphase. Den Ausgangspunkt bildet dabei das Verschicken einer Meldung von der zentralen Instanz an eine Modellkomponente. Sobald die direkte Propagierung, die unmittelbare Weitergabe von Meldungen zwischen verschiedenen Komponenten, abgeschlossen ist, geht die Kontrolle zur Visualisierung über. Ist eine Visualisierung zwischen einzelnen Propagierungsschritten nötig, muss anstelle der direkten Weitergabe ein Umweg über den Ereigniskalender gemacht werden, indem darin ein Ereignis mit der aktuellen Zeit und dem entsprechenden Adressaten eingetragen wird.

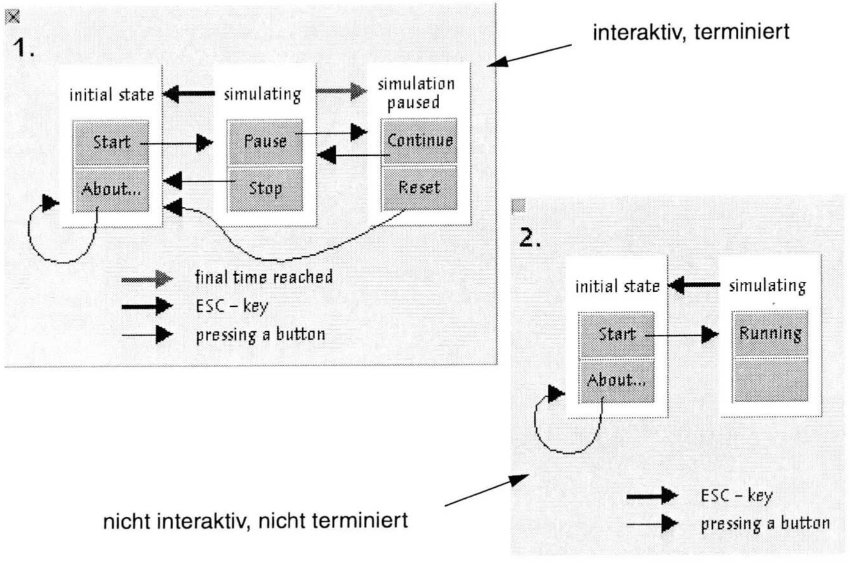
5.1.3. Szenarien der Simulation

Bei einer einmalig ablaufenden Simulation sind zwei wesentliche Kategorien von Szenarien zu unterscheiden:

1. Entweder läuft die Simulation durchgehend ab oder es sind Möglichkeiten vorgesehen, in die Simulation interaktiv einzugreifen. Im Task-System von Oberon bedeutet der interaktive Modus, dass der mittlere Task eines Simulationslaufs nur einen kleinen Zeitabschnitt abarbeitet, die Kontrolle dann abgibt und später die Simulation wieder aufnimmt (vgl. Fig. 5.3).

   Neben Eingaben durch den Benutzer ermöglicht der interaktive Modus auch ein pseudoparalleles Simulieren mehrerer Modelle. Dies kann genutzt werden, um verschiedene Varianten eines Modells simultan zu simulieren oder verschiedene Simulationen durch Zugriff auf gemeinsame Daten synchronisiert laufen zu lassen.
2. Die Simulation wird durch einen vorbestimmten Endzeitpunkt terminiert oder sie läuft endlos, z.B. zu Demonstrationszwecken.

Simulationsumgebungen können entsprechend konfiguriert werden; Simulationen unterliegen je nach Modus, in dem sie ablaufen, einem entsprechenden Zustandsautomaten:



interaktiv, terminiert

nicht interaktiv, nicht terminiert

Fig. 5.4: Modi von Simulationen

Zustandsübergänge werden ausgelöst durch das Ende einer Simulation, das Anklicken von Schaltflächen und das Drücken der ESC-Taste.

5.2. Beispiele komponierter Modelle

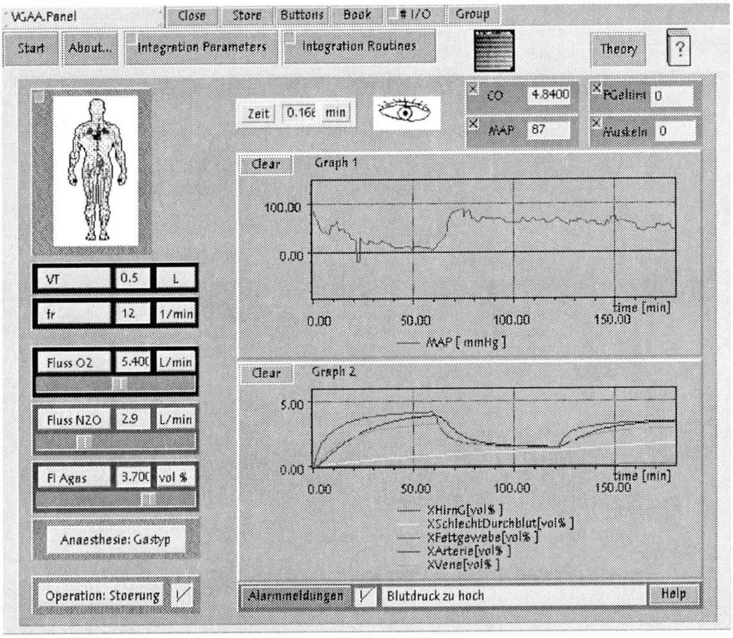
Die Beispiele sollen verschiedene Aspekte der Implementierung von Modellkomponenten sowie das Erstellen von Simulationsmodellen durch Komposition beleuchten:

- die Möglichkeiten der Visualisierung und parametrischen Interaktion,
- das Generieren neuer Instanzen von Komponenten,
- die Verbindungen zwischen verschiedenen Komponenten im Stile von Blockdiagrammen und Netzwerken,
- Phasen des Evaluierens von Gleichungen und Propagierens von Ereignissen,
- kombinierte Simulation anhand einer transaktionsorientierten Simulation,
- Einbetten von Simulationen in andere Umgebungen.

5.2.1. Verbindungen zur graphischen Benutzeroberfläche

Anästhesie

Die Simulation eines Anästhesiekreislaufs ist als Lernprogramm für angehende Anästhesistinnen und Anästhesisten konzipiert und soll diesen die Fähigkeit vermitteln, den Einfluss bestimmter Parameter abzuschätzen:



Zeit 0.16(min

☒ CO 4.8400

☒ MAP 87

☒ PGehirn 0

☒ Muskeln 0

VT 0.5 L

fr 12 1/min

Fluss O2 5.40(L/min

Fluss N2O 2.9 L/min

FI Agas 3.70(vol %

Operation: Stoerung ✓

Alarmmeldungen ✓ Blutdruck zu hoch

Fig. 5.5: Lernprogramm für die Regelung der Anästhesietiefe

Neben patientenspezifischen Parametern sind Charakteristika der künstlichen Beatmung wie Fluss und Typ des Anästhetikums interaktiv einstellbar. Optional kann eine Störung aufgeschaltet werden.

Die gesamte Beschreibung des Anästhesiekreislaufs ist konzentriert in einer Modellkomponente enthalten. Ihre Deklaration hat die Form

```
Model* = POINTER TO ModelDesc;
ModelDesc* = RECORD (EditCore.ModelDesc)
  FlowN2O: BasicGadgets.Real;  (* Zeiger zum Einlesen von Parametern *)
  ...
  pEndt: LONGREAL;             (* interne Signale *)
  ...

  MAP: BasicGadgets.Real;      (* Zeiger zum Darstellen von Signalen *)
  MAPgraph: GraphXY.XYFrame;
  MAPcurve: GraphXY.Curve;     (* mehrere Kurven pro Graph möglich *)
  eye: Rembrandt.Frame;
  ...
END;
```

und umfasst alle modellspezifischen Signale, sowie die Elemente der Oberfläche:

- `FlowN2O` bildet die Verbindung zu einem Parameter (`FlowN2O.val`), der interaktiv geändert werden kann (Fluss des Lachgases N_2O);
- pEndt ist ein internes Signal (endtidaler Druck);
- `MAP`, `MAPgraph` & `eye` bilden Verbindungen zu Elementen, die das Signal `MAP.val` visualisieren. Bei `MAP` wird von der in Gadgets üblichen Modell-Sicht-Struktur Gebrauch gemacht: `MAP` ist ein Modell, das mit Sichten wie Textfeldern oder Schiebern verknüpft ist, die nach einem Standardprotokoll aktualisiert werden. Andere graphische Elemente wie der Graph oder die Ikone mit dem sich schliessenden und öffnenden Auge benutzen dafür spezielle Mechanismen.

Der Handler reagiert auf die `ScenarioCore.CoreMsg` durch Einleiten entsprechender Aktionen:

```
PROCEDURE Handler*(model: Objects.Object; VAR M: Objects.ObjMsg);
BEGIN
  WITH model: Model DO
    IF M IS ScenarioCore.CoreMsg THEN
      WITH M: ScenarioCore.CoreMsg DO
        IF M.id = ScenarioCore.initialize THEN
          M.result:= ScenarioCore.accepted;
          Init(model)
        ELSIF M.id = ScenarioCore.evaluate THEN
          Dynamic(model);
        ELSIF M.id = ScenarioCore.visualize THEN
          Results(model);
        ELSIF M.id = ScenarioCore.collectData THEN
          Backup(model);
        END
      END
    ELSIF
      (* andere Meldungen, die einer speziellen Behandlung bedürfen *)
      ...
    ELSE
      (* Weiterleiten von Meldungen zum Standard-Handler: *)
      EditCore.ModelHandle(model, M)
    END
  END
END Handler;
```

- `ScenarioCore.initialize:`

  ```
  M.result:= ScenarioCore.accepted;
  ```

  Gibt an, ob die Modellkomponente akzeptiert ist, was z.B. dann nicht der Fall ist, wenn nicht alle nötigen Eingangssignale vorhanden sind. Eine Modellkomponente, die mit `ScenarioCore.refused` antwortet, wird zu einem späteren Zeitpunkt nochmals angefragt. Dies kann sich so lange wiederholen, bis die anfragende Instanz nur noch Absagen erhält.

  Elemente, die auf der graphischen Oberfläche enthalten sind, werden im Gadgets System üblicherweise durch ihren Namen gesucht[1]. Wird eine Oberfläche komponiert, muss bekannt sein, welche Typen von Komponenten auf der Oberfläche erwartet werden und welche Namen sie haben. Da die Suche bei komplex strukturierten Oberflächen lange dauern kann, ist es sinnvoll, in der Initialisierungsphase die entsprechenden Zeiger zu setzen und speichern und so in späteren Phasen Zeit zu sparen:

  ```
  PROCEDURE Init(model: Model);
  BEGIN
    GadgetsUtils.FindBGReal("FlowN2O", model.frame, model.FlowN2O);
    GadgetsUtils.FindBGReal("MAP", model.frame, model.MAP);
    ...
  END Init;
  ```

  Wird ein Element nicht gefunden, so wird es intern alloziert, denn Signale der Oberfläche (wie z.B. `model.MAP.val`) werden häufig im Modell verwendet.

- `ScenarioCore.evaluate:`

  ```
  PROCEDURE Dynamic(model: Model);
  BEGIN
    ......... := Function(model.FlowN20.val);
    model.pEndt := ...;
    model.MAP.val := ... ;
    ...
  END Dynamic;
  ```

  Der algebraische Teil der Systemgleichung - soweit durch die Komponente beschrieben - wird evaluiert[2]. Beeinflusst der Benutzer während der Simulation das Element `FlowN20`, so wird automatisch der aktuelle Wert benutzt.

- `ScenarioCore.visualize`

  ```
  PROCEDURE Results(model: Model);
  BEGIN
    BasicGadgets.SetValue(model.MAP);
    GraphXY.Plot(model.MAPgraph, model.MAPcurve,
                                   Integrator.time, model.MAP.val);
    AnestEyes.Set(model.MAPeye, model.MAP.val);
    ...
  END Results;
  ```

  NB: `BasicGadgets.SetValue` beeinflusst alle Sichten von MAP. Die zwei anderen Anweisungen beziehen sich auf ganz konkrete Elemente, die nicht flexibel durch Instanzen eines anderen Typs ausgetauscht werden können.

---

[1] Dies gilt für interaktiv komponierte Oberflächen. Bei Oberflächen, die durch Programmkontrolle aufgebaut sind, sind diese Verbindungen zum vornherein bekannt.

[2] Details über das konkret verwendete Modell sind in [24] und [49] enthalten.

- `ScenarioCore.collectData`

  ```
  PROCEDURE Backup(model: Model);
  BEGIN
    GraphXY.Backup(model.MAPgraph)
    ...
  END Backup;
  ```

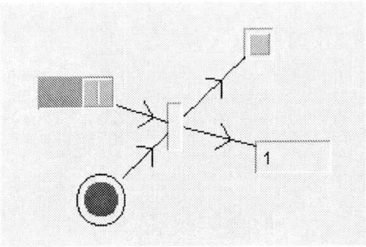
  Um bei weiteren Versuchen eine Vergleichsmöglichkeit für die simulierten Signale zu haben, werden die bereits berechneten weiterhin angezeigt. Durch den Clear-Befehl auf dem Graphen können alle früheren Signale gelöscht werden.

Modell - Sicht - Komposition

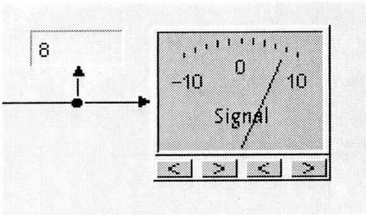
Im Anästhesiebeispiel enthält die Sicht - ein Panel - verschiedene Subkomponenten, die flexibel ausgetauscht werden können. Eine andere Sicht als das Panel ist allerdings wenig sinnvoll.

Bei weniger komplexen Modellen wie den Stellen eines Petri-Netzes oder Messanzeigen, die auf Signale aufgesetzt sind, ist es dagegen häufig von Interesse, die Modelle mit verschiedenen Typen von Sichten zu kombinieren:

Visualisierung von Stellen:



Visualisierung von Signalen:



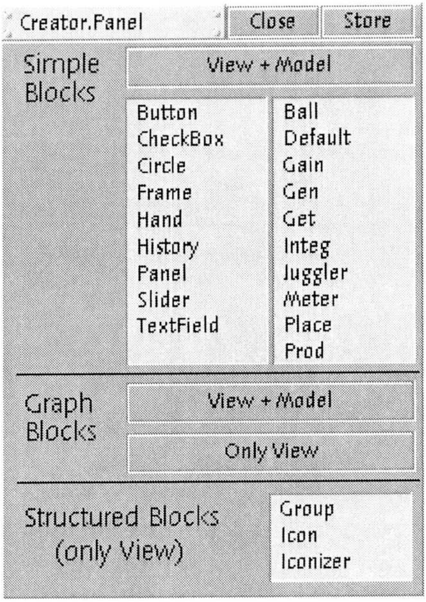
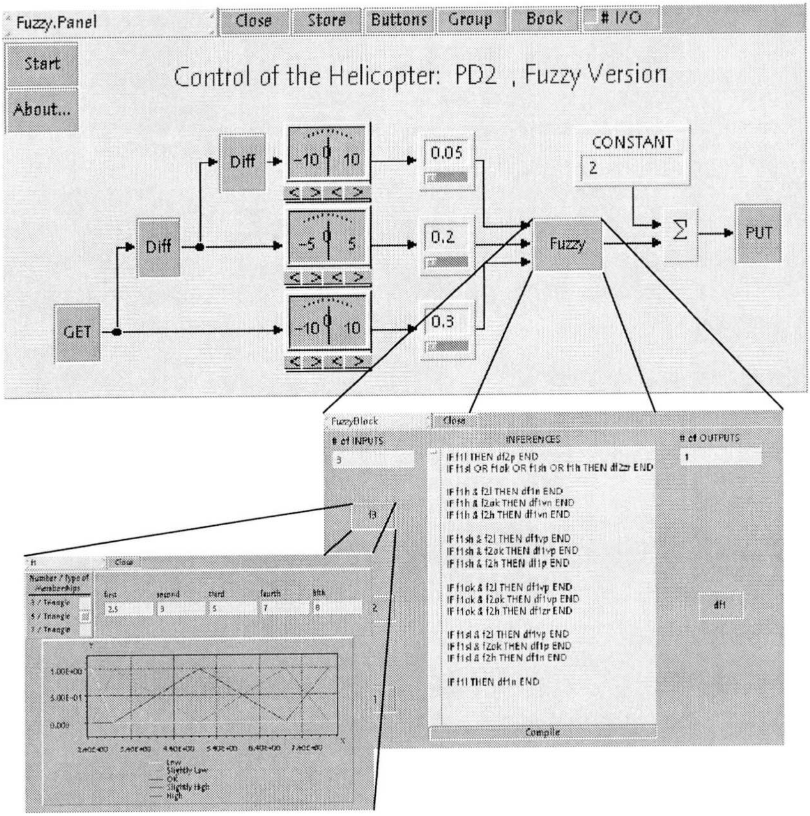


Fig. 5.6: ***Verschiedene Kombinationen von Modell und Sicht***

Modell-Sicht-Kombinationen können unter Benutzung des Creator.Panels erstellt werden; die Handhabung ist dieselbe wie beim Gadgets.Panel des Gadgets Systems. Bereits kreierte Komponenten können durch Kopieren wiederverwendet werden.

Oberflächen für komplexere Modellkomponenten

Bei Modellen, deren Konfigurierung aufwendig ist, kann ähnlich vorgegangen werden wie im Falle der hierarchischen Verschachtelung von Blockdiagrammen (Fig. 4.6):



\# of INPUTS: 3

\# of OUTPUTS: 1

first: 2.5 | second: 3 | third: 5 | fourth: 7 | fifth: 8

```
IF f1l THEN df2p END
IF f1sl OR f1ok OR f1sh OR f1h THEN df2zr END

IF f1h & f2l THEN df1n END
IF f1h & f2ok THEN df1vn END
IF f1h & f2h THEN df1vn END

IF f1sh & f2l THEN df1vp END
IF f1sh & f2ok THEN df1vp END
IF f1sh & f2h THEN df1p END

IF f1ok & f2l THEN df1vp END
IF f1ok & f2ok THEN df1vp END
IF f1ok & f2h THEN df1zr END

IF f1sl & f2l THEN df1vp END
IF f1sl & f2ok THEN df1p END
IF f1sl & f2h THEN df1n END

IF f1l THEN df1n END
```

Fig. 5.7: Mehrstufiges Editieren eines Fuzzy-Modells

Durch das Öffnen eigenständiger Dokumente wird das Editieren der Modellparameter ermöglicht. Beim in Fig. 5.7 gezeigten Beispiel wird auf einer ersten Ebene die Anzahl Ein- und Ausgänge sowie die Regelbasis eines Fuzzy-Modells festgelegt, auf der zweiten Ebene können die Zugehörigkeitsfunktionen jedes Ein- und Ausgangs bestimmt werden[1].

Das Öffnen der Dokumente wird durch Anklicken der Komponenten ausgelöst. Das bedingt, dass das Cmd-Feld der Komponenten mit dem entsprechenden Befehl versehen ist.

---

[1] Details der Implementierung und möglicher Erweiterungen finden sich in [16].

5.2.2. Blockdiagramme

Da Modelle sowohl Blockdiagramm- als auch Netzwerkverbindungen aufweisen können, ist die separierte Darstellung der beiden Aspekte nur aus Gründen der besseren Übersichtlichkeit gewählt.

Blockdiagramme werden meist benutzt, um Differentialgleichungen und algebraische Gleichungen graphisch darzustellen. Die Signale auf den einzelnen Verbindungen sind permanent vorhanden und verändern sich kontinuierlich:

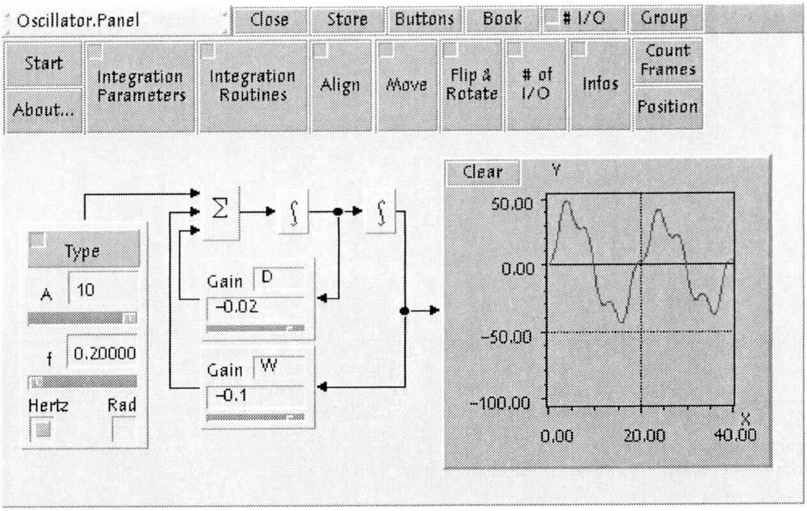


Fig. 5.8: *Blockdiagramm eines Oszillators*

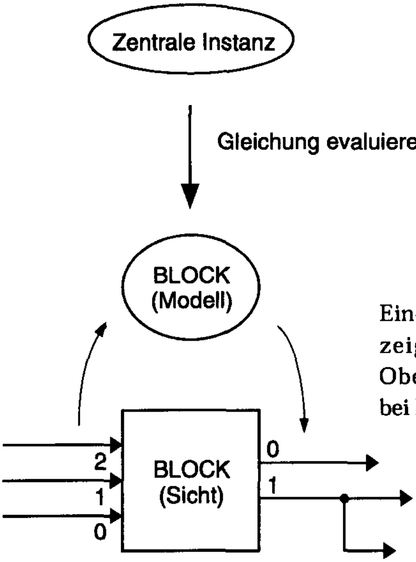
Bei der Integration der Gleichungen sind die Möglichkeiten der Dezentralisierung limitiert. In der Regel werden die Gleichungen immer in derselben Reihenfolge abgearbeitet, nachdem sie einmal sortiert worden sind. Anstelle eines dezentralen Propagierens von Signaländerungen durch das Diagramm hindurch werden die Blockmodelle von einer zentralen Instanz aufgefordert, ihre algebraischen Gleichungen zu evaluieren.

In der Initialisierungsphase wird beim Sortiervorgang ein Integratormodell sofort akzeptiert und der Ausgang freigegeben:

```
PROCEDURE IntegBlockInit*(model: Gadgets.Object;
                                    VAR M: ScenarioCore.CoreMsg);
BEGIN
   IF (model # NIL) & (model IS EditLinks.Model) THEN
      WITH model: EditLinks.Model DO
         M.result := ScenarioCore.accepted;
         model.out[0].signal.atDisposal := TRUE;
      END
   END
END IntegBlockInit;
```

Bei einem statischen Block dagegen müssen alle Eingänge verfügbar sein, bevor die Meldung positiv beantwortet und die Ausgänge des Blocks freigegeben werden[1]:

```
PROCEDURE StaticBlockInit*(model: Gadgets.Object; VAR M:
ScenarioCore.CoreMsg);
  VAR n: INTEGER;
BEGIN
  IF (model # NIL) & (model IS EditLinks.Model) THEN
    WITH model: EditLinks.Model DO
      M.result := ScenarioCore.accepted;
      (* Test der Eingänge *)
      FOR n := 0 TO model.NI-1 DO
        IF EditLinks.ConnHasInput(model.in[n]) &
                             (model.in[n].signal.atDisposal = FALSE) THEN
          M.result := ScenarioCore.refused;
          RETURN
        END
      END;
      (* Ausgänge freigeben*)
      FOR n := 0 TO model.NO-1 DO
        model.out[n].signal.atDisposal := TRUE;
      END
    END
  END
END StaticBlockInit;
```



Ein- und Ausgänge sind im Uhrzeigersinn numeriert; gemäss Oberon-Konvention beginnend bei Null.

Fig. 5.9: Evaluieren von Gleichungen in Blockmodellen

Im weiteren werden in der Initialisierungsphase Zustände von Integratoren, aber auch Zeit- und Zustandsereignisse deklariert.

---

[1] Feinere Auflösung, d.h. ein genaues Prüfen, welche Ausgänge die Verfügbarkeit welcher Eingänge bedingen, ist nicht implementiert. Der Block wird diesbezüglich als atomare Einheit behandelt.

5.2.3. Netzwerke

Die Verbindungen in einem Netzwerk sind gegenüber denjenigen in einem Blockdiagramm ungeordnet. Anstelle numerierter Verbindungen treten hier zwei Mengen von benachbarten Knoten. Bidirektionale Verbindungen sind erlaubt.

Petri-Netze

Eine Anwendung für solche Strukturen sind Petri-Netze:

> *Ein Petri-Netz ist ein Paar (G, μ), wobei G = (E, V) ein bipartiter Graph mit einer endlichen Anzahl Knoten (die Menge V) ist. V ist unterteilt in die disjunkten Mengen P und Q; E besteht aus Paaren der Form (p_i, q_j) und (q_j, p_i), wobei $p_i \in P$ und $q_j \in Q$ gilt. Die Anfangsmarkierung μ ist ein $|P|$-Vektor nichtnegativer ganzer Zahlen* [6].

Beim Feuern einer Transition wird von den Stellen in ihrem Vorbereich eine Marke abgezogen[1], bei den Stellen im Nachbereich eine zugefügt.

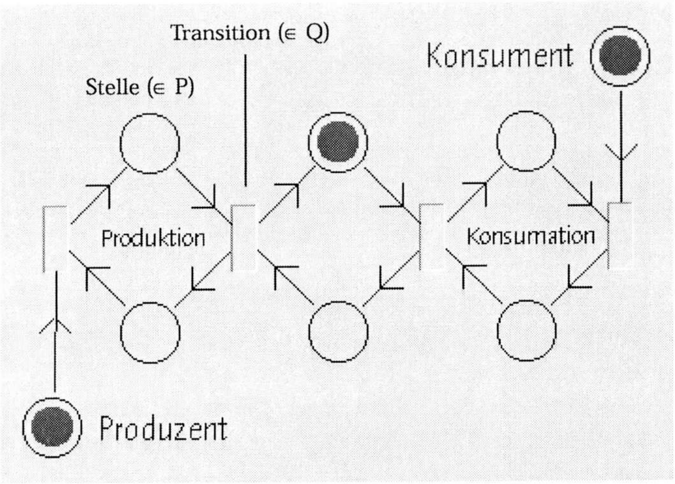


Fig. 5.10: Synchronisation von Produzent und Konsument

Die Synchronisation von Produzent und Konsument ist ein typisches Modellierungsproblem, für das Petri-Netze geeignet sind (Fig. 5.10). Die mittleren zwei Stellen, durch die Produktions- und Konsumationsprozess gekoppelt sind, zeigen den Status 'voll' bzw. 'leer' eines Buffers mit Kapazität eins an. Sie garantieren, dass die beiden Prozesse synchronisiert sind.

In COSIMO ist neben der Simulation von Petri-Netzen auch ein schrittweises Feuern der Transitionen durch Interaktion möglich und zwar sowohl beim Stillstand der Simulation als auch während der interaktiven Simulation.

---

[1] Die Markierung darf nicht negativ werden; dies entspricht der Feuerbedingung.

Der Handler der Transitionen bearbeitet im wesentlichen zwei editor- bzw. simulationsspezifische Meldungstypen:

```
PROCEDURE TransHandler*(trans: Objects.Object; VAR M: Objects.ObjMsg);
BEGIN
  WITH trans: Trans DO
    IF M IS EditCore.CoreMsg THEN
      WITH M: EditCore.CoreMsg DO
        IF M.id = EditCore.establishLink THEN
          IF M.model IS Place THEN M.number := ScenarioCore.accepted;
          ELSE M.number := ScenarioCore.refused
          END
        ELSIF
          ...
        END
      END
    ELSIF M IS ScenarioCore.CoreMsg THEN
      WITH M: ScenarioCore.CoreMsg DO
        IF M.id = ScenarioCore.initialize THEN
          M.result := ScenarioCore.accepted;
          IF TransEnabled(trans) THEN
            InsertEvent(trans)
          END;
        ELSIF M.id = ScenarioCore.schedEvent THEN
          IF TransEnabled(trans) THEN
            InsertEvent(trans)
          END;
        ELSIF M.id = ScenarioCore.handleEvent THEN
          IF TransEnabled(trans) THEN
            FireTrans(trans);
          END;
        END
      END
    ELSIF
      ...
    END
  END
END TransHandler;
```

- EditCore.CoreMsg

  id = EditCore.establishLink:

  stellt sicher, dass Transitionen nur mit Stellen (Typ: Place) oder mit Erweiterungen davon verbunden werden.

- ScenarioCore.CoreMsg

  id = ScenarioCore.initialize:

  Meldung positiv beantworten; eventuell ein Ereignis in den Kalender eintragen.

  id = ScenarioCore.schedEvent:

  Diese Meldung wird von Stellen im Vor- und Nachbereich der Transition generiert, deren Markierung geändert hat. Wird dadurch das Feuern der Transition möglich, wird ein Eintrag in den Kalender vorgenommen.

  id = ScenarioCore.handleEvent:

  Feuern der Transition; der Test `TransEnabled(trans)` fängt die Fälle ab, in denen eine Transition mehrfach im Ereigniskalender eingetragen ist, obwohl sie nur einmal feuern kann.

Prozeduren wie `TransEnabled` operieren auf den Listen der Modelle, die sich in Vor- und Nachbereich befinden:

```
PROCEDURE TransEnabled*(trans: Trans): BOOLEAN;
  VAR connList: EditLinks.ConnList; M: Objects.AttrMsg;
BEGIN
  connList := trans.inListOfNodes;
  WHILE connList # NIL DO
    M.id := Objects.get; M.name := "Value";
    connList.model.handle(connList.model, M);
    IF (M.class # Objects.Int) OR (M.i <= 0) THEN
      RETURN FALSE;
    END;
    connList := connList.next;
  END;
  RETURN TRUE;
END TransEnabled;
```

Die Typen der benachbarten Knotenmodelle sind nicht von unmittelbarem Interesse; die Meldungen können anonym verschickt werden.

Beispiel einer transaktionsorientierten Simulation

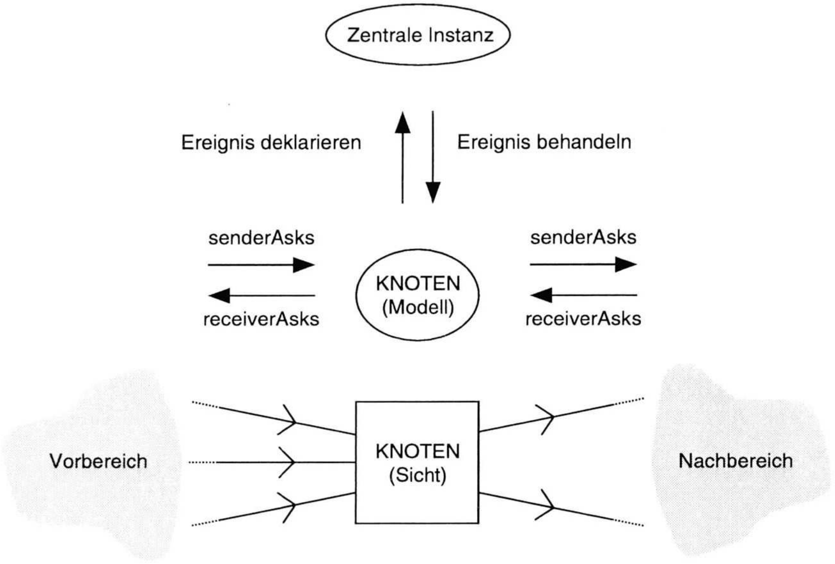


Fig. 5.11: Kommunikation von Knotenmodellen

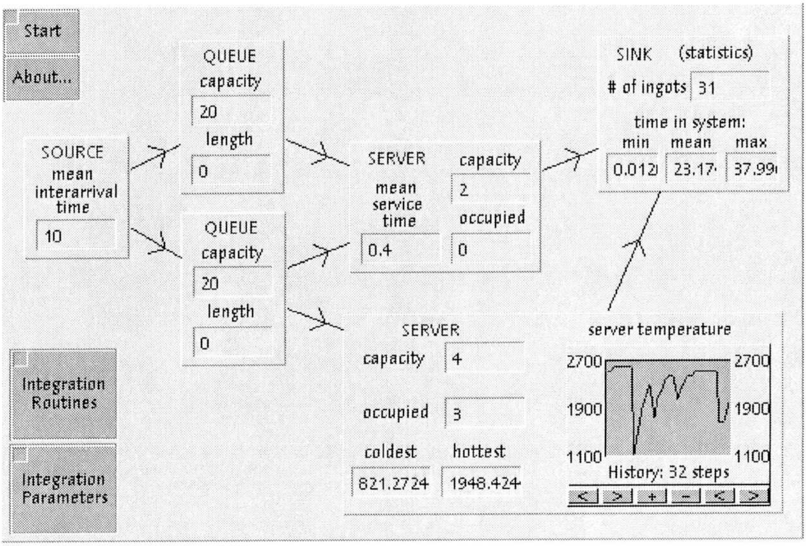
Die beschriebene Art, zwischen benachbarten Knoten ein Protokoll aufzuziehen, kann verwendet werden, um transaktionsorientierte Simulationen durchzuführen. Zwischen verschiedenen Objekten werden mobile Objekte verschoben, die selber den Charakter von Modellkomponenten haben.

Nötig ist dazu lediglich die Erweiterung der Meldungstypen um die Zeiger, die auf die zu transferierenden Objekte weisen.

Ein bekanntes Beispiel dieser Art der Modellierung und Simulation besteht darin, in einem Netz von Stationen Barren kursieren zu lassen. Die Stationen können Quellen (Sources), Warteschlangen (Queues), Bedienelemente bzw. Schmelzöfen (Servers) und Senken (Sinks) sein (Fig. 5.12). Die Barren weisen eine Temperatur auf, die sich so verhält, wie es die momentane Station verlangt: entweder einer Differentialgleichung oder diskreten Veränderungen gehorchend. Daneben haben die Barren den Zeitpunkt ihres Eintretens ins System gespeichert, damit in den Senken statistische Kenngrössen ermittelt werden können.

Die Schmelzöfen sind in zwei Ausprägungen implementiert:

- Stochastischer Typ: die Bedienzeiten werden über eine Verteilungsdichte gewonnen; Modelle, die nur diesen Typ aufweisen, können rein diskret simuliert werden.
- Deterministischer Typ: die Bedienzeiten werden durch exakte Simulation der Temperaturverläufe berechnet; die Simulation hat kombinierten Charakter.


SOURCE
mean interarrival time: 10

QUEUE
capacity: 20
length: 0

QUEUE
capacity: 20
length: 0

SERVER
mean service time: 0.4
capacity: 2
occupied: 0

SERVER
capacity: 4
occupied: 3
coldest: 821.2724
hottest: 1948.424

SINK (statistics)
# of ingots: 31
time in system:
min: 0.012
mean: 23.17
max: 37.99

Fig. 5.12: *Beispiel einer transaktionsorientierten Simulation*

Die Kommunikation zwischen den Modellkomponenten und zentralen Instanzen umfasst während der eigentlichen Simulation im wesentlichen die Handhabung

- von Zeitereignissen in Quellen und in stochastischen Servern
- von Differentialgleichungen und Zustandsereignissen in deterministischen Servern.

Die Interaktionen zwischen den verschiedenen Modellkomponenten sind komplex und werden im folgenden anhand der Typendeklaration skizziert.

```
MODULE Ingots;

IMPORT EditLinks, ...;

CONST (* zur IngotMsg *)

  senderAsks = 0; recipientAsks = 1; accepted = 2; refused = 3;
  ...
TYPE

  Ingot* = POINTER TO IngotDesc;
  IngotDesc* = RECORD (EditLinks.ModelDesc)
    time, temp, dTemp: LONGREAL; (* dTemp: Ableitung von temp *)
    served: BOOLEAN;
  END;
```

Neben den physikalischen Attributen weisen die Barren auch eine Kennzeichnung als abgefertigt (served) auf, wenn sie in einem Ofen wegen Rückstaus festsitzen.

```
  IngotMsg* = RECORD (Objects.ObjMsg)
    ingot: Ingot;
    id: SHORTINT;
  END;

  Source* = POINTER TO SourceDesc;
  SourceDesc* = RECORD (EditLinks.ModelDesc)
    meanTime: BasicGadgets.Real;(* exponentiell verteilte Zeiten*)
  END;
```

Beim Ausführen eines Zeitereignisses wird ein Barren erzeugt, weitergeschickt und ein neues Zeitereignis eingetragen.

```
  Queue* = POINTER TO QueueDesc;
  QueueDesc* = RECORD (EditLinks.ModelDesc)
    capacity, length: BasicGadgets.Integer;
    ingots: POINTER TO ARRAY OF Ingot;
  END;
```

Warteschlangen nehmen Barren auf, sofern ihre Kapazität noch nicht voll ausgeschöpft ist. Ist die Schlange vor der Aufnahme leer gewesen, wird direktes Weiterschicken des Barrens versucht.

Warteschlangen geben auf Anfrage Barren weiter, falls sie einen beinhalten. Ist die Schlange vor der Weitergabe voll gewesen, wird im Vorbereich aktiv um einen neuen Barren nachgesucht, um einen allfälligen Stau abzubauen.

```
  Server* = POINTER TO ServerDesc;
  ServerDesc* = RECORD (EditLinks.ModelDesc)
    capacity, occupied: BasicGadgets.Integer;
    ingots: POINTER TO ARRAY OF Ingot;
  END;

  DeterministicServer* = POINTER TO DeterministicServerDesc;
  DeterministicServerDesc* = RECORD (ServerDesc)
    coldest, hottest, serverTemp: BasicGadgets.Real;
    dTemp: LONGREAL;
  END;

  StochasticServer* = POINTER TO StochasticServerDesc;
  StochasticServerDesc* = RECORD (ServerDesc)
    meanTime: BasicGadgets.Real;(* exponentiell verteilt *)
  END;
```

Für die Server gilt bezüglich aktiven Weiterschickens und Entgegennehmens dasselbe wie für die Warteschlangen. Zusätzlich müssen sie bei der Annahme eines Barren ein entsprechendes Ereignis deklarieren und beim Eintreten desselben versuchen, den Barren weiterzuschicken.

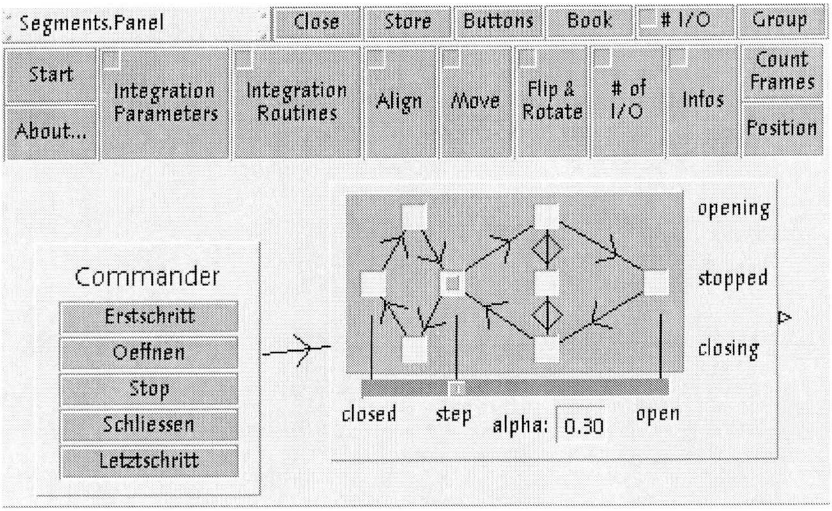
```
Sink* = POINTER TO SinkDesc;
SinkDesc* = RECORD (EditLinks.ModelDesc)
  minTime, meanTime, maxTime: BasicGadgets.Real;
  nrOfIngots: BasicGadgets.Integer;
END;
```

Senken nehmen jeden Barren an und werten seine Aufenthaltszeit im System aus.

Auch Modelle dieser komplexen Art können rein textuell und ohne jegliches objektorientierte Konstrukt beschrieben und simuliert werden [62]. Die Vorzüge der objektorientierten Programmierung und der komponentenorientierten Oberfläche zeigen sich aber, sobald Erweiterungen oder Modifikationen der Funktionalität, Änderungen der Struktur oder der graphischen Darstellung verlangt sind.

Modell mit Zustandsautomat

Fig. 5.13 zeigt eine Komponente, die ein Wehr bei einem Flusskraftwerk repräsentiert. Neben der unterlagerten, kontinuierlichen Dynamik weist sie einen Zustandsautomaten auf. Zustandsübergänge können durch Zustandsereignisse wie das Erreichen bestimmter Positionen ausgelöst werden oder durch Befehle, die an das Wehr erteilt werden. Je nach Zustand werden Befehle ignoriert; so wird z.B. während des Erstschritts überhaupt kein Befehl akzeptiert, da zuerst der Erstschritt vollständig abgeschlossen werden muss.



alpha: 0.30

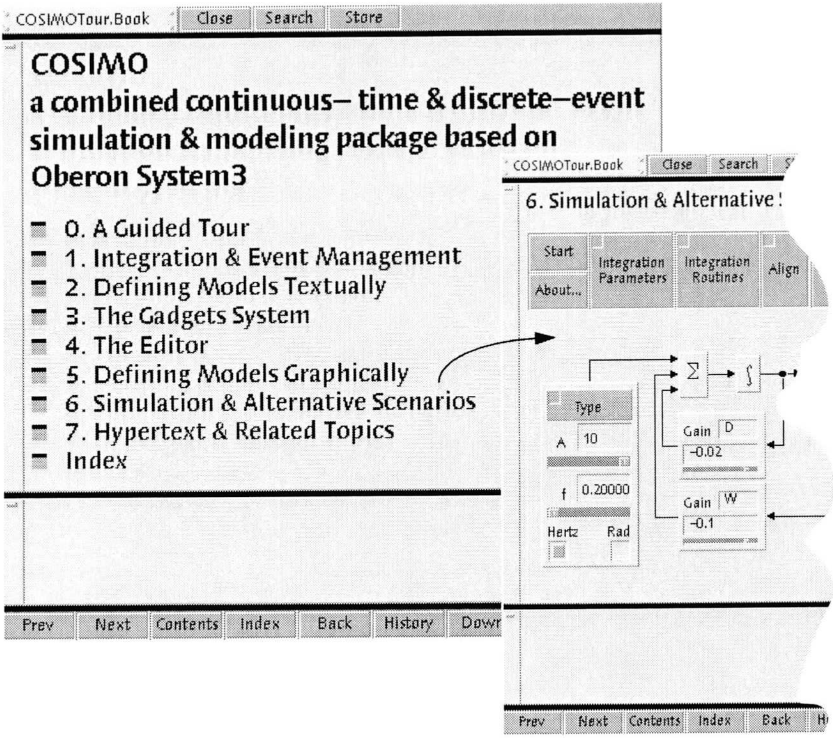
Fig. 5.13: Komponente mit einem Zustandsautomat

Der Commander ist ein Beispiel für Elemente, mit denen andere Komponenten gezielt gesteuert bzw. ausgetestet werden können.

5.2.4. Einbetten von Simulationen in andere Umgebungen

Die folgenden Möglichkeiten ergeben sich automatisch durch die Verwendung des dokumentenzentrierten Gadgets System; zusätzlicher Programmieraufwand tritt nicht auf.

Die Einbettung von Simulationsmodellen in Hypertexte, die mit Fussnoten und einem Index versehen werden können, wird unterstützt. Die Simulationen bleiben dadurch voll funktionsfähig.



COSIMO
a combined continuous– time & discrete–event simulation & modeling package based on Oberon System3

- 0. A Guided Tour
- 1. Integration & Event Management
- 2. Defining Models Textually
- 3. The Gadgets System
- 4. The Editor
- 5. Defining Models Graphically
- 6. Simulation & Alternative Scenarios
- 7. Hypertext & Related Topics
- Index

6. Simulation & Alternative

A 10
f 0.20000
Gain D -0.02
Gain W -0.1

Fig. 5.14: Simulation eingebettet in einem Hypertext

In derselben Art können Simulation auch in Internet-Dokumente wie html-Seiten integriert werden. Allerdings bedingt dies, dass der Betrachter einen Oberon-Browser benutzt[1]. Dagegen ist es nicht notwendig, dass der Client den Code fertig kompiliert verfügbar hat, da es verschiedene Wege gibt, ihn Dokumenten anzuheften.

---

[1] Denkbar ist auch, Plug-Ins für herkömmliche Browser zu erstellen, wie dies beim Juice-Paket existiert.

Alternative Szenarien

6

Laurel plädiert in *Computers as Theatre* [41] dafür, bei der Entwicklung von Software weniger den Entwurf konsistenter Objekte und Umgebungen als Ausgangspunkt zu nehmen, sondern mehr Gewicht auf die Aktion zu legen, aus der sich das weitere dann ergibt. Insbesondere wird für die abzulaufende Aktion die Metapher des Theaters bzw. des Dramas benutzt. Ein Drama hat verschiedene Akteure, es setzt sich aus bestimmten Phasen zusammen und es hat ein Ziel, auf das es sich hinbewegt.

Wird diese Metapher auf das vorliegende Simulationspaket angewendet, so entsprechen die Komponenten des Modells den mitspielenden Akteuren; die Phasen, die durchlaufen werden, sind Initialisierung, Simulation und das Verarbeiten von Ergebnissen. Das Ziel, das durch den Ablauf der verschiedenen Phasen erreicht werden soll, ist der Gewinn verschiedener Einsichten wie z.B. der erreichte Endzustand der Simulation oder Effekte, die auf dem Weg zum Endzustand demonstriert werden können.

In Anlehnung an die Metapher des Dramas kann der typische Ablauf einer Simulation aufgefasst werden als ein spezielles Szenario, als Typus eines Bühnenstücks, das von einem Ensemble gespielt wird. Andere Szenarien können vom selben Ensemble oder von Ensembles, die bestimmte Typen von Akteuren bedingen, zur Aufführung gebracht werden.

Die Idee, das Szenario der Simulation zu modifizieren, entspringt ursprünglich dem Gedanken, ein Werkzeug zur Verfügung zu haben, um elektronische Lehrbücher (Tutorial) wirkungsvoll zu unterstützen, in denen gezielt der Einfluss bestimmter Parameter auf das Simulationsverhalten demonstriert wird. Das Konzept hat sich aber über die eigentliche Intention hinaus als universeller einsetzbar erwiesen. Szenarien sind für eine breite Palette von Anwendungen entwickelt worden, wie Regelungen und Steuerungen oder solche, bei denen die Analyse eines Simulationspanels im Vordergrund steht.

Bei der Implementierung eines Szenarios kann auf dieselben Funktionen und Protokolle zurückgegriffen werden, die auch bei der Simulation benutzt werden, wie z.B. das Suchen und Sortieren der beteiligten Teilmodelle oder das Verschicken von Meldungen an alle Modellkomponenten. Um ein Szenario in der gewünschten Weise zu unterstützen, bedarf es gegebenfalls Modifikationen und Ergänzungen der bestehenden Teilmodelle.

6.1. Simulationsablauf gemäss einem Drehbuch

Simulationsabläufe nach einem Drehbuch ablaufen zu lassen, heisst im wesentlichen, die Parametersätze während und zwischen den verschiedenen Läufen in einer automatisierten Art ändern zu können. Dies ist von Interesse bei der Durchführung von Optimierungen oder bei der Untersuchung und Demonstration des Einflusses bestimmter Parameter. Konkrete Beispiele sind die Beziehung zwischen Parametern und der Dämpfung eines Systems oder strukturelle Effekte, bei denen z.B. durch Parametervariation ein System in deterministisches Chaos abkippt.

Die Möglichkeiten, ein Drehbuch zu implementieren, sind im wesentlichen:

1. Das Modell enthält eine spezielle Komponente, durch die das Drehbuch definiert wird. Diese kann - muss aber nicht - aktiv in die eigentliche Simulation involviert sein. Auf die Meldung, die das Simulationsende ankündigt, reagiert sie durch eine Parameteränderung, entweder eines lokalen Parameters oder von Parametern, die sich im Kontext befinden; danach löst sie einen Neustart der Simulation aus.
2. Durch den Startknopf wird nicht der bei der Simulation übliche Befehl ausgelöst, sondern z.B. `MyScenario.Do`, wobei hinter diesem Befehl die kompilierte Version des abzufahrenden Drehbuchs steckt.
3. Durch den Startknopf wird nicht der bei der Simulation übliche Befehl ausgelöst, sondern z.B. `Scenario.Do MyScenario.Text`, wobei `MyScenario.Text` einer Batch-Datei entspricht.

Die ersten beiden Varianten sind sehr flexibel; sie können unter Ausschöpfung der ganzen Mächtigkeit der zugrundeliegenden Programmiersprache formuliert werden. Zwischen- oder Endresultate der Simulation können in natürlicher Weise in weitere Parametermodifikationen einfliessen. Dies prädestiniert sie für Anwendungsfälle, wie Optimierungen in mehreren Dimensionen wo der interaktive Aspekt eher zweitrangig ist. Für Probleme, bei denen es darum geht, aufbauend auf einer graphischen Oberfläche bestimmte Effekte zu zeigen, die durch Parameteränderung verursacht sind, bietet sich die dritte Variante an. Beim Abarbeiten der Batch-Datei wird die erste Zeichenkette jeder Zeile als formal parameterloser Oberonbefehl interpretiert[1].

---

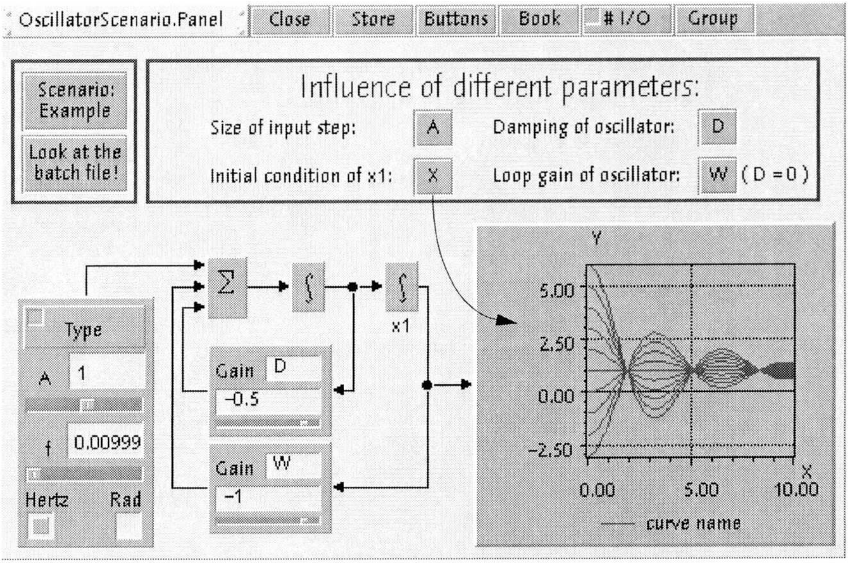
[1] Der Befehl `Scenario.Do` entspricht dem `Configuration.Do` in der Standarddistribution 'The Spirit of Oberon'.

Beispiel: *InfluenceOf X10.Text*

```
Scenario.Init                             (* initialisiert Kontext *)

Scenario.FindObject Graph ~               (* Name in Kontext *)
Scenario.NameCurrentObject GraphObj       (* 'interner' Name *)
Scenario.GraphClear GraphObj
Scenario.GraphAxis GraphObj 0 10 -3 6

Scenario.FindObject x1 ~
Scenario.NameCurrentObject x1Obj
Scenario.RunEnsemble x1Obj InitialValue 0 1 -1 2 -2 3 -3 4 5 6
                                          (* Variation eines Attributs *)
```



A 1

f 0.00999

Gain D -0.5

Gain W -1

Fig. 6.1: *System zweiter Ordnung mit variierter Anfangsbedingung*

Das Vorgehen besteht darin, Batch-Dateien zu entwerfen, die sich auf fertig komponierte Simulationsumgebungen abstützen. Bestimmte Objekte können im Kontext gesucht und verändert werden. Um dasselbe Objekt nicht mehrmals suchen zu müssen, wird im Modul `Scenario.Mod` eine interne Liste von Objekten unterhalten.

Es ist möglich, durch eigene Module die Mächtigkeit der Drehbücher zu steigern und an eigene Bedürfnisse anzupassen. Denkbar sind z.B. die Einflussnahme auf die Struktur des Modells (scriptable editor) oder Kontrollstrukturen wie bedingte Verzweigungen.

6.2. Regelungen

6.2.1. Aspekte der Regelung

Bei der Umwandlung eines Simulationsszenarios in ein Regelungsszenario sind verschiedene Aspekte von Bedeutung, die die Synchronisation zwischen Strecke und Regler betreffen:

1. Es müssen Schnittstellen vorhanden sein, die es ermöglichen, mit dem Prozess zu kommunizieren. In Fig. 6.2. sind dies die GET- bzw. PUT-Blöcke.

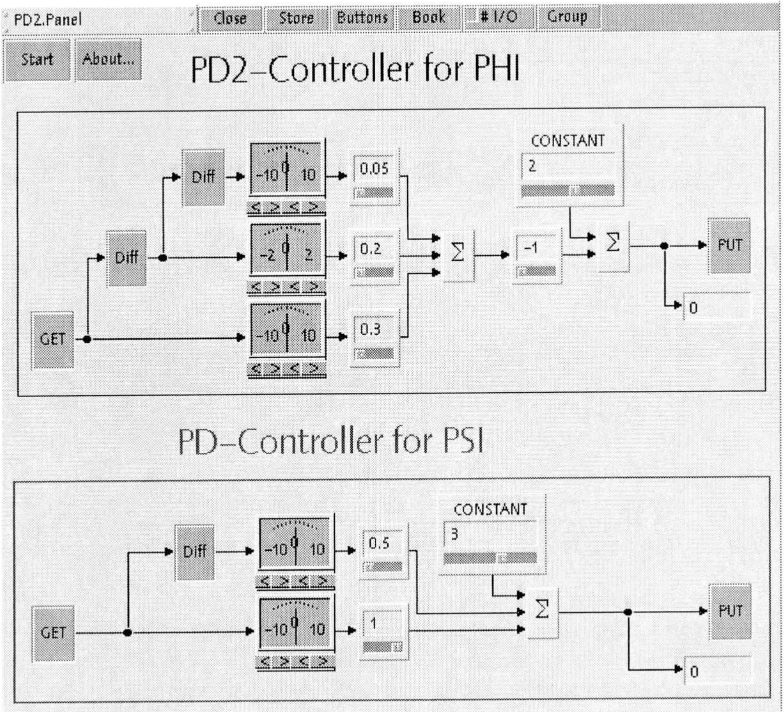


Fig. 6.2: Panel zur Regelung eines Helikoptermodells

2. Die Implementierung der Simulation schliesst die Möglichkeit des Simulierens in Echtzeit nicht ein. Soll ein Regler dynamische Elemente wie die in Fig. 6.2 gezeigten Differenzierblöcke enthalten, so müssen diese sich zeitlich an der Rechnerzeit und nicht an der Simulationszeit orientieren.
3. Im Entwurf eines Abtastreglers ist immer - explizit oder implizit - die Dynamik der Strecke sowie die Abtastrate, mit der Messwerte eingelesen und Stellgrössen ausgegeben werden, involviert. Wird diese Abtastrate nicht eingehalten, kann die Rege-

lung an Qualität verlieren bis hin zum Verlust der Stabilität. Auch kurzfristige Schwankungen der Abtastrate (*Jitter*) sind zu vermeiden.

6.2.2. Helikoptermodell

Eine der Strecken, die in den Praktika des Instituts für Automatik verwendet werden, um verschiedene Reglerentwürfe durchzuführen, ist ein elektromechanisches System, das - obwohl nicht freifliegend - Ähnlichkeit mit einem Helikopter hat (Fig. 6.3). Das Modell hat zwei mechanische Freiheitsgrade; die Lage des Helikopters wird durch zwei Messgrössen, den Nickwinkel φ und den Gierwinkel ψ, erfasst. Die zwei Propeller werden durch Motoren angetrieben, deren Eingangsspannungen durch die Stellgrössen u_φ und u_ψ gegeben sind. Was den Reglerentwurf nichttrivial macht, sind neben diesem Multi-Input/Multi-Output-Charakter (MIMO) verschiedene Nichtlinearitäten. Einerseits werden diese durch die Geometrie der Konstruktion verursacht, andererseits ist die Kennlinie des Propellers nichtlinear und überdies die Wirkung des Propellers ortsabhängig.

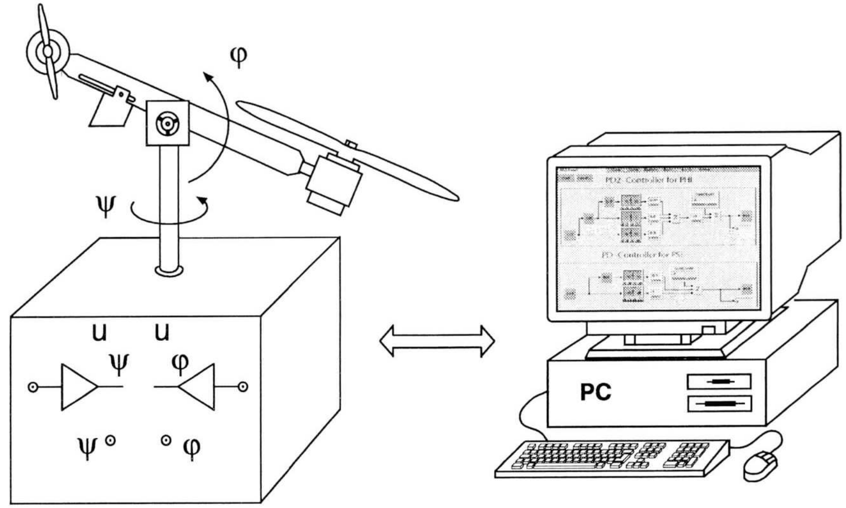


Fig. 6.3: Konfiguration des Versuchs 'Helikoptermodell'

6.2.3. Regelung des Helikoptermodells

Bei der konkreten Realisierung des Reglers in COSIMO ist Wert darauf gelegt worden, in kurzer Zeit einen funktionierenden Regler zu implementieren, dessen Einstellwerte interaktiv durch direkten Eingriff in die graphische Oberfläche verändert werden können (fast prototyping). Ermöglicht wurde dies unter anderem durch das Entfallen jeglicher Zeitverluste durch Kompilationen, wie sie z.B. in den MATLAB- oder

MATRIXx-Umgebungen nötig sind. Die Struktur wurde von einem bereits existierenden Regler übernommen. Die Regler für die beiden Winkel sind entkoppelt; es handelt sich dabei um einen PD-Regler für den Gierwinkel und einen PD^2-Regler für den Nickwinkel (Fig. 6.2).

1. Die PUT- & GET-Funktionen, mit denen die Hardware angesprochen wird, sind plattformspezifisch implementiert worden. Im vorliegenden Fall läuft Oberon unter Windows 95, die anzusprechende IO-Karte ist ein Burr-Brown PCI-20000 System. Die Schnittstelle wird gebildet durch eine 32-bit-DLL (dynamic link library).
2. Die numerische Differenzierung von *u(t)* wird nach der einfachst möglichen Formel

$$\left.\frac{du}{dt}\right|_{t = k\Delta t} = \frac{u(k\Delta t) - u((k-1)\Delta t)}{\Delta t}$$

gebildet. Da die Zeitauflösung in der Standarddistribution von Oberon bei 300 oder 1000 Ticks pro Sekunde liegt, ist es relativ ungenau, ein einzelnes Abtastintervall Δt zu bestimmen.

```
CONST
  ticksPerSecond = 300;             (* platform dependent *)

TYPE
  Diff* = POINTER TO DiffDesc;
  DiffDesc* = RECORD (EditLinks.ModelDesc)
    oldInValue: LONGREAL;
    oldTime: LONGINT;
  END;

PROCEDURE DiffInit(obj: Diff);      (* id = ScenarioCore.initialize *)
BEGIN
  obj.oldInValue := 0;
  obj.oldTime := MIN(LONGINT)
END DiffInit;

PROCEDURE DiffOnce(obj: Diff);      (* id = ScenarioCore.evaluate *)
  VAR currentTime: LONGINT; deltaT: LONGREAL;
BEGIN
  currentTime := Oberon.Time();
  deltaT := (currentTime - obj.oldTime) / ticksPerSecond;
  obj.out[0].signal.value
               := (obj.in[0].signal.value - obj.oldInValue) / deltaT;
  obj.oldInValue := obj.in[0].signal.value;
  obj.oldTime := currentTime
END DiffOnce;
```

3. Da das Regelungsszenario sowieso nicht harten Echtzeitbedingungen genügen kann, ist dieser Gesichtspunkt etwas vernachlässigt worden. Dadurch ist aber auf der anderen Seite die Möglichkeit geschaffen worden, die Koeffizienten des Reglers während der Laufzeit interaktiv einzustellen. Das Reglerszenario funktioniert gleich wie das Szenario der interaktiven Simulation: nach dem Abarbeiten aller Blöcke wird die Kontrolle abgegeben und eine Eingabe durch den Benutzer möglich. Durch solche Interaktionen kann der Jitter natürlich beträchtliche Ausmasse annehmen. Dazu kommen noch Effekte, wie sie beispielsweise durch Broadcasting von Meldungen erzeugt werden; je nach Anzahl Objekte auf der Oberfläche benö-

tigt die Abarbeitung einer solchen Meldung unterschiedlich viel Zeit. Da die Regelsstrecke sich gegenüber langfristigen Abweichungen der Abtastrate von ca. ± 20% robust verhält und auch einige Ausreisser verträgt, ist der Gesamtaufbau für einige Experimente durchaus nützlich. Für eine korrekte Implementierung, die Interaktionen zulässt, muss aber notwendigerweise ein Basissystem benutzt werden, mit dem es möglich ist, das Einhalten harter Echtzeitanforderungen zu garantieren.

Möglichkeiten, das Regelungsszenario zu erweitern, sind unter anderem:

- Vorgabe einer Solltrajektorie für den Helikopters, z.B. durch Signalgeneratorblöcke.
- Mechanismen, durch die in Anlehnung an die Optimierungen in Simulationsszenarien in einem adaptiven Sinn die Regelung optimiert werden kann.

6.3. Steuerungen

6.3.1. Aspekte der Steuerung

1. Wie bei der Regelung werden Schnittstellen zum Prozess benötigt.
2. Im Gegensatz zur Regelung sind bei Steuerungen die Restriktionen bezüglich harter Echtzeit etwas gelockert in dem Sinne, dass es meist nicht nötig ist, Mess- und Stellglieder in ganz bestimmten Zeitpunkten zu bedienen, solange dies schnell genug erfolgt.
3. Bei einer Steuerung wird üblicherweise nicht wie bei der Regelung das ganze Diagramm zyklisch abgearbeitet, sondern nur der momentan aktuelle Teil.
4. Schliesslich ist bei einer graphisch dargestellten Steuerung im Normalfall die Komplexität der Funktion einzelner Modellkomponenten einiges höher als bei einer Regelung, wo die einzelnen Blöcke meist ein ziemlich transparentes IO-Verhalten aufweisen.

6.3.2. Eisenbahnmodell

Am Institut für Automatik wird den Studierenden im Praktikum ein Versuch angeboten, bei dem es um die Steuerung einer Eisenbahnanlage geht. Die Anlage ist trotz des spartanischen Aufbaus (Fig. 6.4) hinreichend komplex, um in die Problematik von Verklemmungen (Deadlocks) einzuführen. Als Ausgangspunkt des Versuchs dient eine Steuerung, die ein kollisionsfreies Zirkulieren zweier Züge im Uhrzeiger- bzw. Gegenuhrzeigersinn ermöglicht. Kollisionen sind deshalb ausgeschlossen, weil die Freigabe der Fahrt über ein Reservierungssystem der Blöcke (B1 - B7) gesteuert wird. Die mögliche Verklemmung besteht darin, dass die Züge in den Blöcken 2 und 3 aufeinander zufahren können. Sobald die beiden Sensoren angesprochen haben, blockieren sich die zwei Züge gegenseitig. Dieses Problem gilt es beim Praktikumsversuch zu erkennen zu beheben.

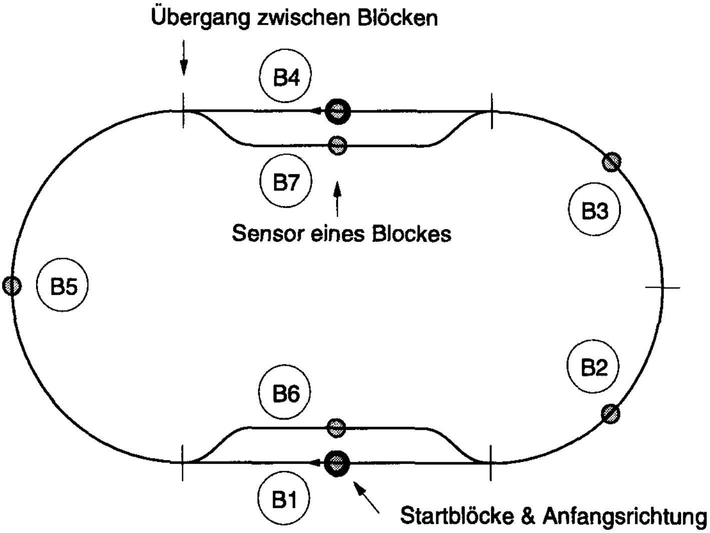


Fig. 6.4: Modelleisenbahn, Blöcke und Sensoren

Die beiden klassischen Ansatzpunkte zur Entwicklung geeigneter Strategien bestehen darin, einen Ausweg aus Verklemmungszuständen anzubieten (recovery) oder sie zu verhindern (look ahead).

- Recovery: Die Verklemmung wird zugelassen. Der Verklemmungszustand wird erkannt und durch eine Ausnahmebehandlung in einen regulären Zustand überführt. Konkret kann dies durch Zurückfahren des einen Zugs und Nachfolgen des anderen Zugs implementiert werden.
- Look ahead: im kritischen Bereich wird bei der Blockreservierung weiter vorausgeschaut, so dass nicht nur Zusammenstösse, sondern auch Verklemmungen verhindert werden. Eine mögliche Realisierung besteht darin, die Blöcke zwei und drei nur gemeinsam reservieren zu können.

Wird beim Entwurf der Steuerung mit dem Modell eines Zustandsautomaten gearbeitet, so sind die Auswirkungen, die die obigen Modifikationen haben, unmittelbar ersichtlich. Die Frage, ob die Gesamtsteuerung verklemmungsfrei arbeitet, kann direkt beantwortet werden.

Basiert dagegen die Modellierung auf Petri-Netzen, so ist diese Frage nicht trivial und die Verklemmungsfreiheit der synthetisierten Steuerung muss explizit nachgewiesen werden.

6.3.3. Modellierung

Da die Sensoren jeweils in Blockmitte angebracht sind - und nicht etwa bei den Übergängen zwischen zwei Blöcken-, wird folgende Modellierung verwendet:

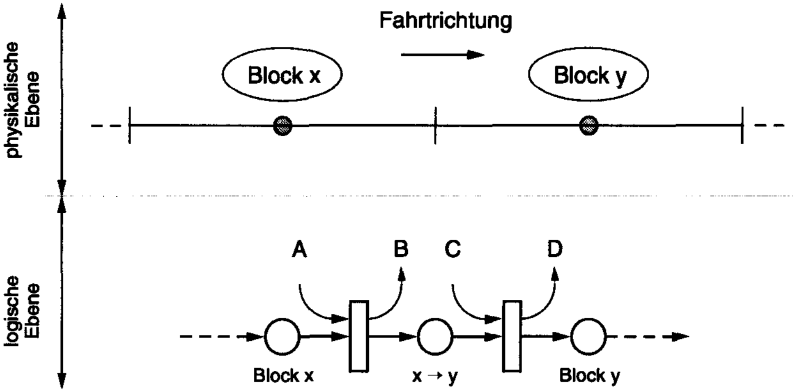


Fig. 6.5: Modellierung der Eisenbahnanlage

Die Fahrt eines Zuges von Block x nach Block y wird mit drei logischen Zuständen dargestellt (Block x / x → y / Block y). Um von Block x nach Block y losfahren zu dürfen, muss Block y verfügbar sein (A). Sobald Sensor y anspricht (C), befindet sich der Zug im Zustand 'Block y'. Die nötigen Aktionen, die beim Eintritt in einen neuen Zustand ausgelöst werden müssen, sind (B) das Reservieren von Block y, das Anlegen einer Fahrspannung in der korrekten Richtung, evt. das Stellen einer Weiche bzw. (D) das Freigeben von Block x und das Wegnehmen der Fahrspannung.

Die Verfügbarkeit der Blöcke könnte im Prinzip auf graphische Weise dargestellt werden, indem pro Block eine Stelle für den Zustand 'frei' (d.h. weder reserviert noch besetzt) eingefügt würde. In der gleichen Weise könnten auch Fahrspannungen, Weichenstellungen etc. visualisiert werden. Es ist allerdings aus folgenden Gründen auf diese explizite Visualisierung verzichtet worden.

1. Das zusätzliche Einzeichnen bedeutet redundante Information. Der Benutzer müsste bereits bei kleinen Modifikationen eine Vielzahl von Elementen explizit nachführen. Dies bildet höchstens eine Quelle von Fehlern, ohne zu grösserer Flexibilität bei der Implementierung der Steuerung zu verhelfen.
2. Die Darstellung würde sehr schnell unübersichtlich, da globale Zustände wie z.B. die Verfügbarkeit der Blöcke einen hohen Vernetzungsgrad haben. Dies wird bereits in minimalen Beispielen deutlich: In Fig. 6.6 ist eine reduzierte Anlage mit drei Blöcken und zwei Zügen mit identischem Umlaufsinn gezeichnet. Die dabei entstehende Netzstruktur ist gerade noch lesbar, weil das zusätzliche Stellen-Tripel (Sensor / Fahrspannung / Verfügbarkeit) nur für einen, nämlich den Block x, eingezeichnet ist.

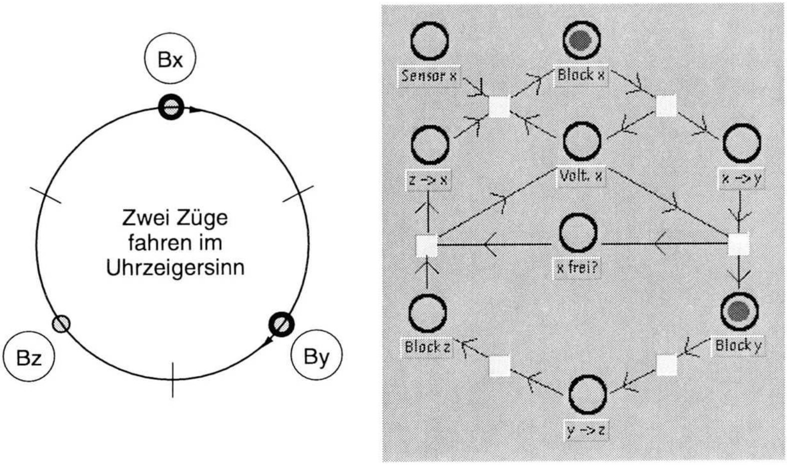


Fig. 6.6: Explizite Formulierung

Die Modellierung der Anlage kann demnach als interpretiertes Petri-Netz erfolgen; die Interpretation lehnt sich an Grafcet an [14], [15]. In Grafcet ist das Feuern von Transitionen an Bedingungen und Ereignisse gebunden. Durch das Erreichen von Schritten (Stellen) werden Aktionen ausgelöst, die entweder Impuls- oder Schrittcharakter haben [1]:

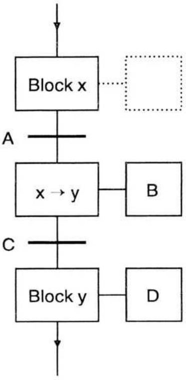


Fig. 6.7: Grafcet-Formulierung des Ausschnittes aus Fig. 6.5

---

[1] Der Unterschied zu einer Grafcet-Implementierung besteht darin, dass die Stellen nicht Werte vom Typ BOOLEAN, sondern solche vom Typ INTEGER repräsentieren. Damit ist es möglich, Stellen z.B. als Rundenzähler zu verwenden.

6.3.4. Steuerung des Eisenbahnmodells

Beim Entwurf der Steuerung ist davon ausgegangen worden, dass die Initialisierungsphase korrekt abgeschlossen wird, dass also der rechnerinterne Zustand der Steuerung mit der physikalischen Position der Züge übereinstimmt. Im weiteren wird die Annahme gemacht, dass die Sensoren und Stellorgane korrekt funktionieren. Ein auf Petri-Netzen basierter Steuerungsentwurf, der solche Sicherheitsaspekte miteinbezieht, findet sich bei Montigel [47].

Ein Panel, das als Grundlage für einen Praktikumsversuch dienen kann, ist in Fig. 6.8 gezeigt. Die darauf komponierte Steuerung enthält in impliziter Form die Ansteue-

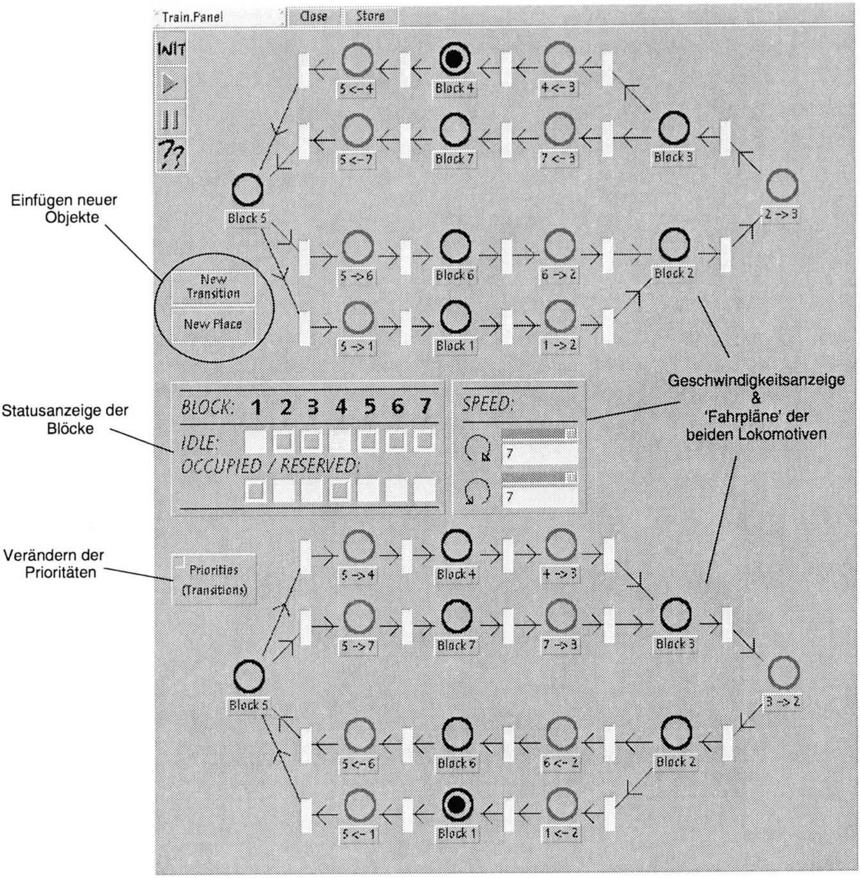


Fig. 6.8: Grundpanel der Eisenbahnsteuerung

rung der Hardware und das Blockreservationssystem. Was im Grundpanel noch nicht gelöst ist, ist das Verhindern der in Kap. 6.3.2 erläuterten Verklemmung.

Gegenüber Simulationen von Petri-Netzen mit COSIMO sind bei der Eisenbahnsteuerung zwei wesentliche Modifikationen eingeführt worden:

1. Die Modellkomponente `TrainNodeModels.Trans` ist eine Erweiterung von `NodeModels.Trans`. Die Transitionen suchen sich während der Initialisierungsphase über die Beschriftungen der benachbarten Stellen die notwendige Information, testen sie auf Konsistenz und physikalische Umsetzbarkeit. Transitionen können mit Prioritäten versehen werden, wobei diese durch den Umfang der Sicht der Transition repräsentiert wird.
2. Das Szenario ist zeitlich optimiert; der Polling-Mechanismus betrachtet nur 'heisse' Transitionen, d.h. solche, die sich im Vorfeld der Züge befinden. Bei gleichzeitig zum Feuern bereiten Transitionen wird die höher priorisierte gefeuert.

Die restlichen Komponenten, insbesondere die Modelle und Sichten der Stellen und die Sichten der Transitionen, sind dieselben wie bei der Simulation.

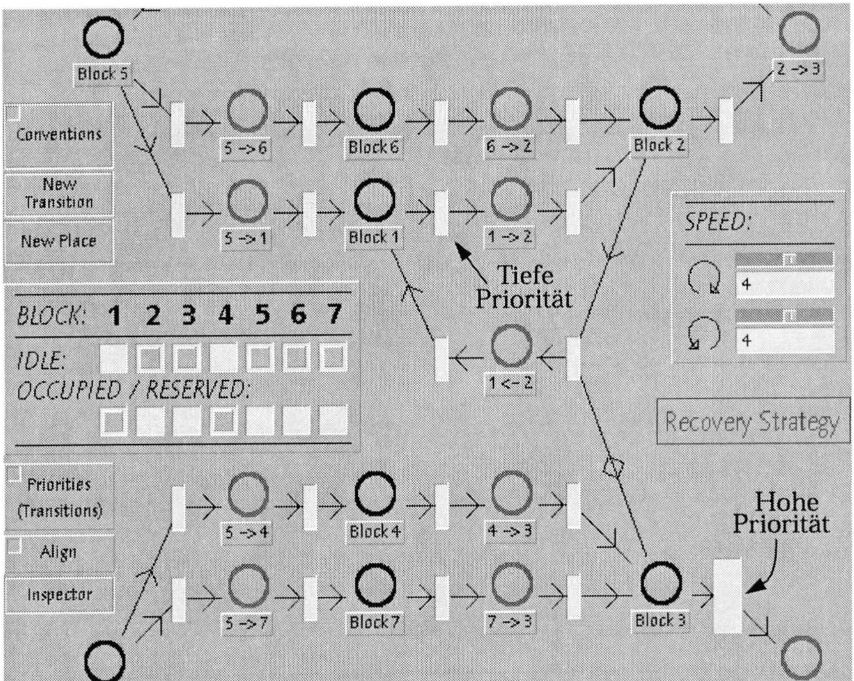


Fig. 6.9: Lösen einer Verklemmung: Ergänzungen im Grundpanel (Fig. 6.8)

Strategien, die z.B. den Fall einer Verklemmung auflösen können, lassen sich durch einfache graphische Ergänzungen des Grundpanels implementieren. Eine mögliche Lösung zeigt Fig. 6.9: derjenige Zug, der sich in Block 2 befindet, fährt zurück nach Block 1. Die Transition, die den Vorgang auslöst, hat eine entsprechende Testkante, mit der sichergestellt wird, dass tatsächlich der Verklemmungszustand vorliegt. Nach dem Zurückfahren liegt eine nichtdeterministische Situation vor; beide Züge könnten nach Block 2 geschickt werden. Um sicherzustellen, dass der richtige Zug losfährt, bekommt die entsprechende Transition höhere Priorität[1].

6.4. Analyse

Die Analyse eines Modells erfolgt auf verschiedenen Ebenen:

- Einordnen des Modells in bestimmte Klassen wie linear/nichtlinear, deterministisch/stochastisch.
- Beschreibung struktureller Eigenschaften wie Beobachtbarkeit, Steuerbarkeit, Existenz und Anzahl von Gleichgewichtslagen und Grenzzyklen.
- Bestimmen von Eigenschaften, die sich auf einen konkreten (Anfangs-)Zustand beziehen, wie z.B. der von der Anfangsmarkierung abhängige Erreichbarkeitsbaum bei einem Petri-Netz.

Allerdings existiert keine Möglichkeit, für beliebig komplexe und inhomogen modellierte Strukturen eine Analysemethode anzugeben. Aussagekräftige Analyseresultate können nur bei eingeschränkten Modelltypen erzielt werden. Auf einer höheren Ebene der Analyse kann Expertenwissen eingesetzt werden, um abzuschätzen, welche Methoden überhaupt zur Anwendung gelangen können.

Im Bereich der kombinierten Modelle sind z.B. die folgenden Werkzeuge entwickelt worden:

- Für Subklassen zeitbehafteter Petri-Netze kann die Max-Plus Algebra benutzt werden [6].
- Bei geschalteten zeitkontinuierlichen Systemen mit übereinstimmenden Vektorfeldern in allen Umschaltpunkten kann eine Erweiterung des Theorems von Bendixon zur Untersuchung der Existenz von Grenzzyklen verwendet werden [8].
- Bei geschalteten zeitkontinuierlichen Systemen kann durch Verwendung von einer Lyapunovfunktion pro Lokation die verallgemeinerte Stabilitätsanalyse nach Lyapunov zur Anwendung gelangen [9].

---

[1] Fährt der falsche Zug los, befindet sich das System in einer Verklemmung höheren Grades, indem nämlich eine Reihe von Zuständen zyklisch durchlaufen wird, ohne dass eine reguläre Fortsetzung möglich ist.

Die Grundlagen zur konkreten Analyse eines graphisch definierten Modells können auf zwei Arten gelegt werden:

1. Die Sprache, durch die die Modelle repräsentiert werden, ist in einem fixen Rahmen festgelegt. Werkzeuge zur Analyse kennen den Sprachumfang, sowie die semantische Bedeutung aller einzelnen Sprachelemente. Das Gesamtverhalten kann eruiert werden, da vollständige Transparenz aller Modellteile gewährleistet ist.
2. Die Granularität der einzelnen Elemente, die dem Analysewerkzeug bekannt sind, ist gröber. Anstelle der zugrundeliegenden Sprache ist beispielsweise die Bedeutung von (Teil-)Modellen bekannt. Das Gesamtverhalten ergibt sich aus der Kenntnis der Funktionalität der einzelnen Teile sowie der gegenseitigen Kopplungen.

Modellierungssprachen wie z.B. SIMNON [4] oder OMOLA [2], die einen fest umrissenen Sprachumfang aufweisen, unterstützen Analysen der ersten Art. Der Preis, der dafür zu zahlen ist, ist die grosse Einengung der Flexibilität bei der Formulierung von Modellen.

In COSIMO ist die zweite Variante implementiert. Die Vorteile, die sich bei einigen anderen Szenarien gezeigt haben, nämlich Möglichkeiten, mit der Aussenwelt zu kommunizieren, sind bei der Analyse eher hinderlich. So ist es beispielsweise schwierig, eine deterministische Lookup-Verbindung zu einem Server von einer stochastischen Verbindung zu einer I/O-Karte zu unterscheiden:

```
norm := ModuleX.LookUpNorm(address, argument);    (* Server *)
gVal := ModuleY.PutGetNow(address, pVal);         (* IO-Karte *)
```

Ein Werkzeug, das zur Analyse von COSIMO-Modellen entworfen wird, bekommt eine gewisse Schwerfälligkeit, weil es nur für bestimmte Objekttypen, von denen es das genaue Verhalten kennt, entworfen werden kann. Insbesondere reicht es nicht aus, einen herkömmlichen Typencheck in der Art

```
IF currentPartOfModel IS aKnownType THEN ...
ELSIF currentPartOfModel IS anotherKnownType THEN ...
...
END;
```

durchzuführen, da ein erweiterter Typ

```
CompletelyDifferentType* = POINTER TO CompletelyDifferentTypeDesc;
CompletelyDifferentTypeDesc* = RECORD (aKnownTypeDesc)
  ...
END
```

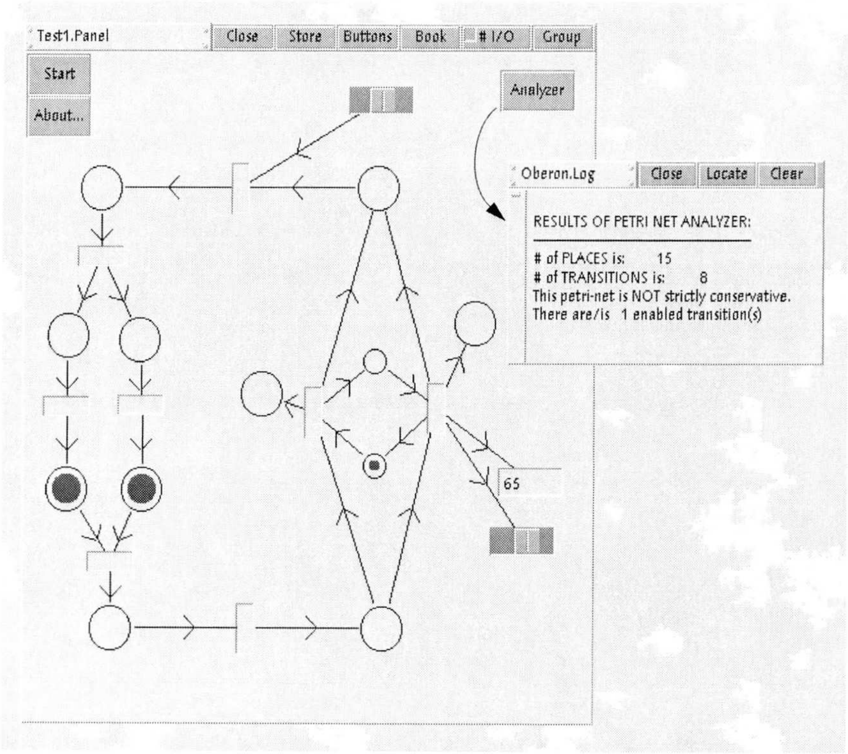
ein völlig anderes (Simulations-) Verhalten aufweisen kann. Der Typ wird deshalb mit Vorteil über den Generatorstring einer Instanz getestet:

```
IF ScenarioCore.IsCreator(currentPartOfModel, "Nodes.NewNode") THEN ...
```

Solange allerdings kein Analysewerkzeug existiert, das aus einer feineren Granularität der Modellbeschreibung Nutzen zöge, spielt es für praktische Belange keine Rolle, wenn ein Analysewerkzeug sich auf bestimmte Typen von Modellen abstützen muss.

Analyse eines Petri-Netzes:

- Die Modellkomponenten werden eingesammelt (Kap. 4.2.4).
- Es wird - über den Generatorstring - geprüft, ob alle Komponenten von gültigem Typ, also Stellen oder Transitionen sind.
- Prüfen der Netzstruktur liefert erste Aussagen.
- Der Zustand (Markierung) liefert weitere Aussagen.
- Dynamische Analysemöglichkeiten wie das Erstellen von Erreichbarkeitsbäumen liessen sich darauf aufbauen [29].



```
RESULTS OF PETRI NET ANALYZER:

# of PLACES is:        15
# of TRANSITIONS is:        8
This petri-net is NOT strictly conservative.
There are/is  1 enabled transition(s)
```

Fig. 6.10: *Analysewerkzeug für ein Petri-Netz*

Um eine Grafcet-Steuerung wie in Fig. 6.8 zu analysieren, müsste das beschriebene Analysewerkzeug angepasst werden an den speziellen Transitionstyp, der dort verwendet wird.

Zusammenfassung und Ausblick

7

7.1. Zusammenfassung

Die vorliegende Arbeit zeigt Konzepte und Implementierung eines Simulators für kombinierte Modelle. Da die verwendete Sprache Oberon hybrid ist, d.h. neben herkömmlichen prozeduralen auch objektorientierte Elemente beinhaltet, ist die Einbettung und Realisierung verschiedener Ideen, die dem einen oder anderen Bereich zuzuordnen sind, in einer natürlichen Weise möglich.

- Die numerisch verlässliche und effiziente Behandlung der Nahtstelle zwischen kontinuierlichen und diskreten Modellteilen, typischerweise zwischen Differentialgleichungen und ereignisgetriebenen Komponenten, wird erreicht durch den Einsatz von einschrittigen Integrationsroutinen mit Interpolationsinformation.
- Durch das aktive Propagieren von Meldungen kann zu bestimmten Zeitpunkten das Aufdatieren des Modellzustands - oder eines Teils davon - forciert werden. Dies ist eine Möglichkeit, die bei vielen kombinierten Simulationspaketen nicht vorhanden ist. Vor allem sind dies die Pakete, die von der Simulation von Differentialgleichungen herstammen; in diesen werden die einzelnen Komponenten häufig auf eine passive Rolle als Funktionen bzw. Lookup-Tabellen reduziert.
- Der auf dem Gadgets System basierende Editor erlaubt das flexible Zusammenstellen von Modellen aus Komponenten. Sowohl Blockdiagramme wie auch Netzwerke werden unterstützt, wobei erstere eher für zentral koordinierte Signalflüsse genutzt werden, letztere für Kontrollflüsse. Das Zuschneiden der visuellen Darstellung einzelner Modellkomponenten auf aktuelle Bedürfnisse wird unterstützt durch die Möglichkeit hierarchischer Verschachtelung sowie durch das Konzept der Modell-Sicht-Separierung.
- Dadurch, dass die einzelnen Modellkomponenten eigenständige Objekte sind, ist es einfach, die zentrale Instanz, die eine Simulation vorantreibt, zu ersetzen, wenn anstelle der Simulation ein anderes Szenario ausgeführt werden soll.

7.2. Mögliche Erweiterungen

Gutknecht [31] betont, dass bei der Beurteilung der Qualität von Software neben dem momentanen Leistungsumfang auch das Potential künftiger Entwicklung betrachtet werden muss. Obwohl Oberon keine klassisch objektorientierte Sprache ist - es gibt darin keine expliziten Klassen oder Methoden -, bietet es durch die Typenerweiterung Möglichkeiten, die darüber hinausgehen, was beispielsweise in Modula-2[1] realisierbar ist.

So können in COSIMO z.B. inhomogene Listen verwaltet werden, was von Umgebungen wie ModelWorks oder SAM-Set, die auf Modula-2 basieren, nicht unterstützt wird. Als Elemente der Liste sind alle Objekte erlaubt, die einen Typ aufweisen, der von einem bestimmten gemeinsamen Basistyp abgeleitet ist. Es ist so denkbar, dass sogar während der Laufzeit der (interaktiven) Simulation neue Modellkomponenten entwickelt oder z.B. vom Internet heruntergeladen werden und nach dem Kompilieren und Laden in die Simulation miteinbezogen werden.

Die konkreten Richtungen, in die COSIMO von einem Benutzer erweitert werden kann, hängen davon ab, wie weit er vertraut ist mit der graphischen und textuellen Programmierung des Systems. Die verschiedenen Ebenen der Programmierung des Gadgets Systems, auf die Bezug genommen wird, sind in Kap. 4.1.5 genauer erläutert:

- Ebene 0: graphische Komposition von Oberflächen zu bestehenden Programmen unter Verwendung bestehender Komponenten.
- Ebene 1: textuelle Programmierung zur Verbindung bestehender Komponenten.
- Ebene 2: textuelle Programmierung neuer Komponenten.

Die Programmierung in einer bestimmten Ebene eröffnet jeweils Möglichkeiten, die in darunter liegenden Ebenen genutzt werden können.

Ebene 0:

- Existierende Komponenten können zu Simulationsmodellen arrangiert werden. Neben den Standardmöglichkeiten der Komposition, wie sie im Gadgets System angeboten werden, können dabei die speziellen Editiermöglichkeiten von COSIMO genutzt werden.
- Modell-Sicht-Paare können rekombiniert werden. Modellparameter können via Sicht durch verschiedenen Komponenten dargestellt und beeinflusst werden. Nach Bedarf können neue Sichten erstellt werden (Ebene 2).

---

[1] Modula-2 unterstützt das Geheimnisprinzip, die Datenabstraktion sowie dynamische Bindung, aber keine Vererbung.

Ebene 1:

- Integrationsalgorithmen können entwickelt werden für spezielle Bedürfnisse wie die Behandlung steifer Systeme oder eine modifizierte Handhabung der Detektion von Zustandsereignissen.
- Weitere denkbare Szenarien sind[1]:

  -Die Integration eines Simulators, der sich an eine schmale Modellierungssprache wie OMOLA anlehnt und Simulationen über einen Parser durchführt[2].

  -Das Szenario des Simulationsablaufs nach einem Drehbuch lässt sich durch deskriptive Modellbeschreibungen ergänzen.

  -Das gezielte Austesten einzelner Modellkomponenten in einem Testrahmen. Kontinuierliche Eingangssignale, aber auch Meldungen, können entsprechend einem Skript oder durch interaktives Auslösen an die verschiedenen Modellkomponenten verschickt werden.

  -Dezentrale Simulation: die Simulation wird dabei auf mehreren Rechnern verteilt durchgeführt. Das globale Zeitdiktat wird von einer bestimmten Maschine vorgegeben.

  -Eine weitere Möglichkeit der Dezentralisierung ist eine Client-Host-Architektur, bei der der Host rechenintensive Aufgaben übernimmt oder Datenbanken zur Verfügung stellt.

  -Die Verwaltung von Simulationen in einem objektorientierten Kern. Ähnlich wie CACSD-Datensätze in Leporello [27], [39] innerhalb einer graphischen Oberfläche (action tree) verwaltet werden, können Stadien und Resultate einzelner Simulationen in Objekte gekapselt werden.

  -Durch Echtzeitsimulation, also durch Synchronisation mit der realen Zeit, könnte grössere Flexibilität erzielt werden, sobald bei der Simulationen ***hardware in the loop*** verwendet wird oder wenn mehrere Rechner beteiligt sind. *Active Oberon* [33] scheint ein valabler Kandidat, um solche Echtzeitsimulationen durchzuführen, da aktive Objekte benutzt werden können.

  -Qualitatives Schliessen und qualitative Simulation beispielsweise entlang der Linien, wie sie von Rasmussen [63] und Koch [38] skizziert werden. Die Integration der Simulation logischer und symbolischer Zustände ist einfach möglich; Erweiterungen von Aufzählungstypen sind allerdings in Oberon nicht vorgesehen und erschweren die Programmierung etwas.

  -Der Einbezug weiterer Pakete im Bereich numerischen oder symbolischen Rechnens wie MATLAB, MATRIXx, Maple oder Mathematica. Erweiterungen in dieser Richtung sind allerdings in der Regel mit einem Verlust auf der Seite der Portabilität verbunden.

---

[1] Gewisse Szenarien bedürfen erweiterter Typen von Komponenten und müssten konsequenterweise in der Ebene 2 eingereiht werden.

[2] Das in Kap. 5.2.1 gezeigte Fuzzy-Modell ist ein erster Ansatz in diese Richtung; während der Initialisierungsphase werden die nötigen Strukturen zum Abarbeiten der Regelbasis dynamisch aufgebaut.

Ebene 2:

- Modelle: Sobald das gewünschte Simulationsverhalten eines Blocks oder Knotens nicht mehr durch bestehende atomare Modellkomponenten - oder durch Kompositionen aus solchen - beschreibbar ist, kann das Modell durch Programmierung auf Ebene 2 erstellt werden.
- Sichten: Wenn die bestehenden Sichten nicht ausreichen, um bestimmte Aspekte von Modellen angemessen darzustellen, können neue Sichten programmiert werden. In der Regel werden dabei die bestehenden Protokolle, die die Verbindung zum Modell regeln, soweit möglich beibehalten.
- Die Editiermöglichkeiten können angereichert werden, so dass beispielsweise eine direkte Eingabe von Systemen aus dem elektrischen oder mechanischen Bereich unterstützt wird. Eine andere Variante ist die Entwicklung eines Editors für Bond-Graphen, der eine Reihe solcher Gebiete abdeckt [72].

Anhang A: Integrationsmethoden mit Interpolationsformeln

Es soll hier das wesentliche der Integrationsmethoden mit eingebetteten Interpolationsformeln skizziert werden, wobei dies aus Gründen der einfacheren Darstellung am Beispiel der expliziten Runge-Kutta Verfahren erfolgt; Interpolationsformeln existieren auch für andere Klassen von Integrationsroutinen.

Betrachten wir die Differentialgleichung

$$\dot{x}(t) = f(x(t), t))$$

die von einem Zeitpunkt t_0 mit $x(t_0) = x_0$ über ein Intervall h bis zur Zeit $t_1 = t_0 + h$ integriert werden soll. Es wird angenommen, dass die Taylorreihe in diesem Intervall existiert und dass sie zu $x(t)$ konvergiert.

In einem expliziten n-schrittigen Runge-Kutta Verfahren wird $x_1 = x(t_1)$ durch

$$\begin{aligned} k_1 &= f(x_0, t_0) \\ k_2 &= f(x_0 + ha_{21}k_1, t_0 + c_2h) \\ k_3 &= f(x_0 + h(a_{31}k_1 + a_{32}k_2), t_0 + c_3h) \\ &\dots \\ k_n &= f(x_0 + h(a_{n1}k_1 + \dots + a_{n,n-1}k_{n-1}), t_0 + c_nh) \\ x_1 &= x_0 + h(b_1k_1 + \dots + b_nk_n) = x_0 + h\sum_{i=1}^{n} b_ik_i \end{aligned}$$

approximiert, wobei n der Anzahl Funktionsaufrufe von $f(.,.)$ entspricht.

Die ursprünglichen RK-Verfahren wurden entwickelt mit dem Ziel, mit möglichst wenig Schritten n möglichst viele Glieder der Taylorreihe nachzubilden, den lokalen Fehler ε also klein zu halten, bzw. eine Methode hoher Ordnung p zu schaffen:

$$\varepsilon = \|x(t_0 + h) - x_1\| \le Kh^{p+1}$$

Eine erste Erweiterung besteht darin, Methoden verschiedener Ordnung zu kombinieren und daraus Information über den lokalen Fehler zu erhalten. Durch Vorgabe einer Toleranz für diesen kann die Schrittweite h optimiert werden. Bei der Erarbeitung dieser Methoden ist darauf zu achten, 'eingebettete Formeln' zu erzeugen, bei denen beispielsweise die Approximation höherer Ordnung durch möglichst wenige zusätzliche Evaluationen von $f(.,.)$ - im Idealfall eine - berechnet werden kann.

Interpolationsformeln können als eine konsequente Weiterführung dieser Idee aufgefasst werden. Sie werden dazu benutzt, zusätzlich Information über die Werte *x(t)* innerhalb des Integrationsintervalls $[t_0, t_1]$ erhalten:

$$x(t) \approx u\left(\frac{t-t_0}{t_1-t_0}\right) = u(\theta) = x_0 + h\sum_{i=1}^{\tilde{n}} b_i(\theta)k_i$$

Dabei sind $\tilde{n}$-n zusätzliche Funktionsaufrufe nötig. $\theta \in [0, 1]$ entspricht dem Intervall $[t_0, t_1]$, $b_i(\theta)$ sind Polynome in θ.

Für unsere Zwecke ist die Interpolation in obiger Form nicht zweckmässig, da bei der Evaluation immer die ganze Formel durchgerechnet werden muss, unabhängig davon, ob vielleicht eine qualitativ schlechtere Interpolation für eine Visualisierung ausreichte. Als Alternative kann

$$\begin{aligned} u(\theta) &= \alpha_0 + \theta(\alpha_1 + (1-\theta)(\alpha_2 + \theta(\alpha_3 + (1-\theta)(\alpha_4 + \theta(\alpha_5 + \ldots))))) \\ &= \alpha_0 + \theta\alpha_1 + \theta(1-\theta)\alpha_2 + \theta^2(1-\theta)\alpha_3 + \theta^2(1-\theta)^2\alpha_4 + \ldots \end{aligned}$$

verwendet werden. Durch den Hermiteschen Aufbau der Formel sieht man, dass die α-Koeffizienten mit aufsteigendem Index jeweils alternierend zwischen Anfang und Ende des Integrationsintervalles Funktionswert, erste, zweite und höhere Ableitungen festlegen. Je nach Anforderungen an die interpolierten Werte kann die Auswertung der Formel vorzeitig abgebrochen werden.

Anhang B: Beispiel zur Detektion von Zustandsereignissen

B.1. Modellbeschreibungen und Aufrufe

Für die MATLAB/SIMULINK- sowie die ACSL-Simulationen des Beispiels aus Kap. 3.5.3 wird jeweils die Modellbeschreibung in textueller oder graphischer Form gezeigt. Davon separiert wird in einer weiteren Datei der Aufruf der Simulation angegeben[1].

In COSIMO enthalten die gezeigten Dateien neben dem Modell auch die Verbindung zum Integrationskern und eine Nachbearbeitung der Resultate.

Die Integrationstoleranz - der relative Fehler - ist bei allen Versuchen auf 10^{-4} gesetzt. Beschränkungen des Integrationsschrittes sind in MATLAB nicht möglich; bei den anderen Programmen wurde das Minimum 10^{-4} und das Maximum 2.5 verwendet.

MATLAB (M1):

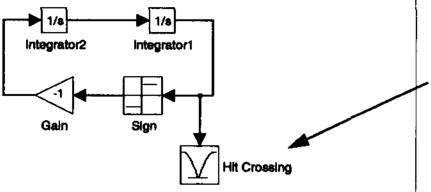
```
function xDot = two_point_dyn(t,x)
xDot(1) =  x(2);
xDot(2) = -sign(x(1));
```

Dateiname:
two_point_dyn.m

```
[t, x] = ode45('two_point_dyn', 0, 8, [2 0], 0.0001);
       % ODE45('func', T0, Tfinal, X0, TOL)
```

SIMULINK (S1, S1):*

Dateiname:
graph_two_point.m



Hit Crossing wird nur beim Versuch mit Zustandsdetektion (S1*) verwendet

```
[t, x] = rk45('graph_two_point', 8, [2 0], [0.0001, 0.0001, 2.5]);
       % RK45('SYSTEM', Tfinal, X0, [TOL MinStep MaxStep])
```

Für die Simulation S1* muss beim Aufruf zusätzlich der Anfangszustand des im 'Hit Crossing'-Block gekapselten Integrators angegeben werden.

---

[1] Die nötigen Ergänzungen zum Eruieren der Anzahl abgelehnter Integrationsschritte werden in den Programmen nicht gezeigt.

ACSL (A1, A2A, A2B):

```
Program TwoPointA1

Constant  x10 = 2.0, x20 = 0.0
Constant  fintim = 8.0
Cinterval cint = 8.0
Minterval mint = 0.0001
Maxterval maxt = 2.5
Algorithm ialg = 9

Dynamic
   Derivative
      x1 = integ(x2, x10)
      x2 = integ(-sign(1.0, x1), x20)
      Call LogD(.TRUE.)
   End

   termt(t.gt.fintim)
End

End
```

Dateiname:
TwoPointA1.csl

spezifischer Teil

```
prepare t, x1, x2
start
matlab
stop
```

Dateiname:
TwoPointA1.cmd
(wird identisch für alle
ACSL-Versuche verwendet)

```
...
Dynamic
   Derivative
      Schedule .XZ. x1
      x1 = integ(x2, x10)
      x2 = integ(-sign(1.0, x1), x20)
      Call LogD(.TRUE.)
   End

   termt(t.gt.fintim)
End
...
```

Dateiname:
TwoPointA2A.csl
(nur spezifischer Teil)

```
...
Initial
   x2Dot = - 1
End

Dynamic
   Derivative
      Schedule SwitchX2Dot .XZ. x1
      x1 = integ(x2, x10)
      x2 = integ(x2Dot, x20)
      Call LogD(.TRUE.)
   End

   termt(t.gt.fintim)
End

Discrete SwitchX2Dot
   x2Dot = -x2Dot
End
...
```

Dateiname:
TwoPointA2B.csl
(nur spezifischer Teil)

COSIMO (C1):

```
MODULE TwoPointC1; (* Simulation without event detection *)

IMPORT Signals, Integrator, IntegDP54, Functions, In, Out;

VAR
   x1, x1Dot, x2, x2Dot,
   Step, minStep, maxStep, tStart, tEnd, integTol: LONGREAL;

PROCEDURE Dynamic;
BEGIN
   x1Dot := x2;
   x2Dot := -Functions.Sign(x1)
END Dynamic;

PROCEDURE WithoutEvents*; (* Aufruf der Simulation *)
   VAR i: INTEGER;
BEGIN
   Integrator.Dynamic := Dynamic;
   Integrator.Integ := IntegDP54.DormandPrince54;

   Integrator.InitLists;
   Signals.DeclStateSignals(x1, x1Dot, 2, Signals.infiniteHorizon);
   Signals.DeclStateSignals(x2, x2Dot, 0, Signals.infiniteHorizon);

   Step := 0.1; minStep := 0.0001; maxStep := 2.5;
   tStart := 0; tEnd := 8; integTol := 0.0001;

   i := Integrator.Integ(Step, minStep, maxStep, tStart, tEnd, integTol);

END WithoutEvents;

PROCEDURE ShowResults*; (* Anzeige der Resultate; eingelesener
                           Parameter: 'Kommunikationsintervall' *)
   VAR time, dTime: LONGREAL;
BEGIN
   In.Open;
   In.LongReal(dTime);
   time := tStart;
   WHILE time <= tEnd DO
      Signals.InterpolateStateSignals(time);
      Out.String("Signals: x1 = "); Out.LongReal(x1, 9);
      Out.String(" x2 = "); Out.LongReal(x2, 9); Out.Ln;
      time := time + dTime
   END
END ShowResults;

END TwoPointC1.
```

```
TwoPointC1.WithoutEvents  TwoPointC1.ShowResults 0.1
```

- die Integrationsroutine operiert auf der Systembeschreibung in `Dynamic`
- `Integrator.InitLists` initialisiert die Listen für alle Signale (Zustände / algebraische Grössen), Schwellwerte, Ereignisse (Kalender)
- `i := Integrator.Integ` resultiert in

  `NoEventOccured* = 0; EventOccured* = -1; KeyInterrupt* = -2;`

  was in C1 nicht von Bedeutung ist
- `Signals.InterpolateStates(time);` setzt die Signale auf die entsprechenden Werte (sofern ihr Horizont hinreichend gross ist)

COSIMO (C3B):

```
MODULE TwoPointC3B;

IMPORT Signals, Integrator, IntegDP54;

VAR
  x1, x1Dot, x2, x2Dot,
  Step, minStep, maxStep, tStart, tEnd, integTol,
  zero, signOFx1: LONGREAL;
  dummy: INTEGER;

PROCEDURE Dynamic;
BEGIN
  x1Dot := x2;
  x2Dot := - signOFx1;
END Dynamic;

PROCEDURE WithEvents*; (* Aufruf der Simulation *)
  VAR i : INTEGER;
BEGIN
  Integrator.Dynamic := Dynamic;
  Integrator.Integ := IntegDP54.DormandPrince54;

  Integrator.InitLists;
  Signals.DeclStateSignal(x1, x1Dot, 2, Signals.infiniteHorizon);
  Signals.DeclStateSignal(x2, x2Dot, 0, Signals.infiniteHorizon);

  signOFx1 := 1;
  Integrator.DeclTH(x1, zero, 0.00001, 0.00001,
                                Integrator.AnySlope, dummy, NIL);

  Step := 0.1; minStep := 0.0001; maxStep := 2.5;
  tStart := 0; tEnd := 8; integTol := 0.0001;

  i := Integrator.Integ(Step, minStep, maxStep, tStart, tEnd, integTol);

  WHILE i = Integrator.EventOccured DO
    signOFx1 := - signOFx1;
    i := Integrator.Integ(Step, minStep, maxStep, Integrator.time, tEnd,
integTol);
  END;
END WithEvents;

BEGIN
  zero := 0;
END TwoPointC3B.
```

```
TwoPointC3B.WithEvents
```

- `Integrator.DeclTH` definiert einen Schwellwert für beide Flanken
- die Signumfunktion ist nicht explizit in `Dynamic` enthalten, sondern wird durch ein Ereignis gesteuert
- da in C3B die Bedeutung eines Ereignisses eindeutig ist, muss der Ereigniskalender nicht inspiziert werden; üblicherweise werden Ereignisse abgeholt und einem Adressaten übermittelt.

B.2. Vergleich der Simulationsresultate

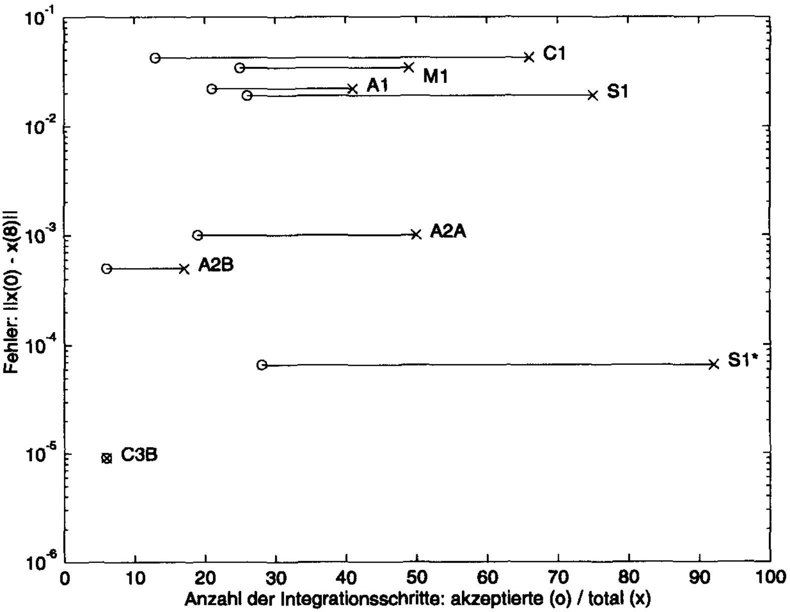


Fig. B.1: Anzahl Integrationsschritte und erzielter Fehler

- Die vier Versuche, bei denen Zustandsereignisse nicht explizit berücksichtigt werden (M1, S1, A1, C1), liegen dicht beieinander, was den Fehler und die Anzahl akzeptierter Schritte anbetrifft. Abgelehnte Schritte treten bei C1 relativ häufig auf, weil die Anpassung der Schrittweite eher grosszügig erfolgt - es wird davon ausgegangen, dass Unstetigkeiten als Ereignisse formuliert sind.
- Bei S1* (SIMULINK) wird das System in der Nähe der Schaltpunkte künstlich versteift. Dies ergibt gegenüber S1 ein beträchtlich genaueres Resultat, allerdings auf Kosten der Effizienz.
- In den Versuchen A2A und A2B[1] (ACSL) kann der Fehler ohne wesentlichen Effizienzverlust verkleinert werden.
- Beim Versuch C3B wird die Integrationstoleranz immer eingehalten; es existieren also keine zurückgewiesenen Integrationsschritte.

---

[1] A2B entspricht allerdings nicht dem Standard der Modellbeschreibung, der in den ACSL-Handbüchern zu finden ist.

Die Figuren B.3 - B.10 zeigen die Punkte, die durch die verschiedenen Algorithmen berechnet wurden. Deutlich sichtbar sind die Unterschiede bei der Schrittweitensteuerung vor und nach den kritischen Stellen:

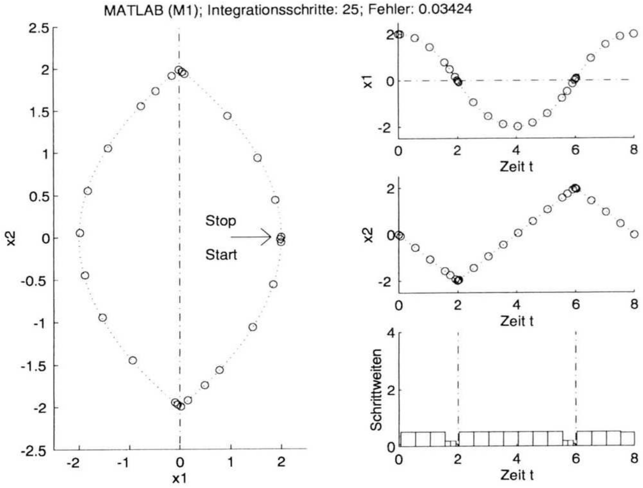


Fig. B.2: *MATLAB (M1): ohne explizite Detektion*

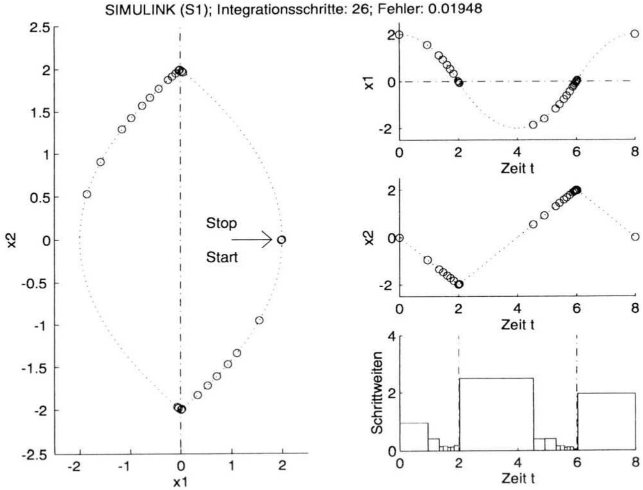


Fig. B.3: *SIMULINK (S1): ohne explizite Detektion*

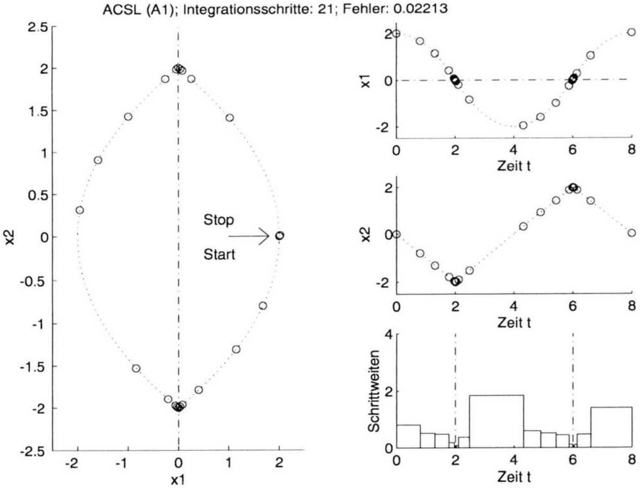


Fig. B.4: ACSL (A1): ohne explizite Detektion

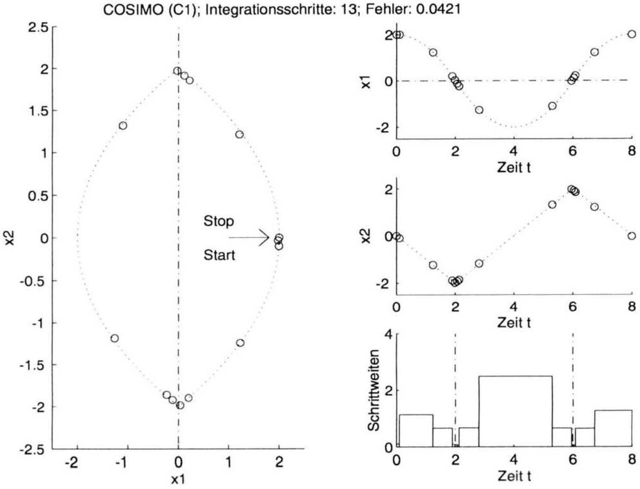


Fig. B.5: COSIMO (C1): ohne explizite Detektion

S1* und A2A unterscheiden sich optisch nicht wesentlich von S1 bzw. A1; die Differenzen liegen in den nicht visualisierten abgelehnten Integrationsschritten an den Schaltpunkten.

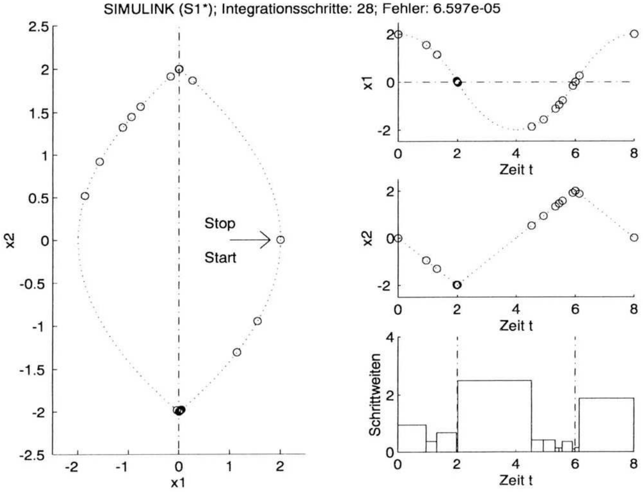


Fig. B.6: *SIMULINK (S1*): mit zusätzlicher Versteifung des Systems*

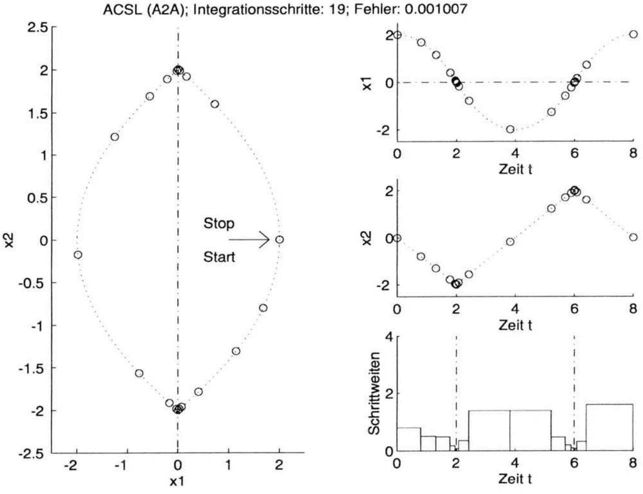


Fig. B.7: *ACSL (A2A): Detektion; kein Umschalten der Systemgleichung*

In A2B und C3B wird mit grossen Schrittweiten über die kritischen Stellen integriert; bei C3B ist zusätzlich die Interpolationsinformation skizziert.

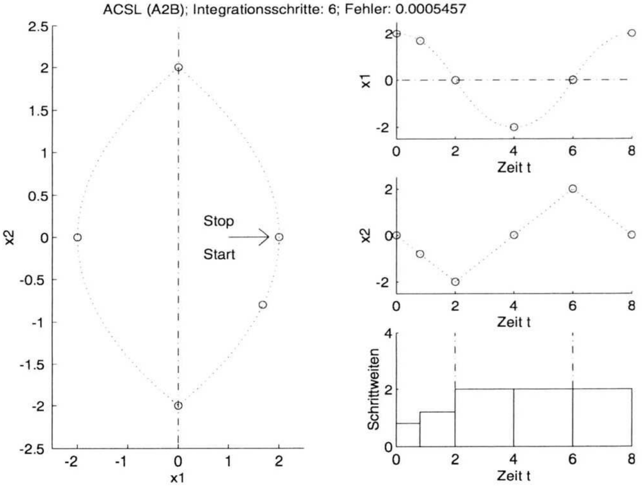


Fig. B.8: *ACSL (A2B): Detektion; Umschalten der Systemgleichung*

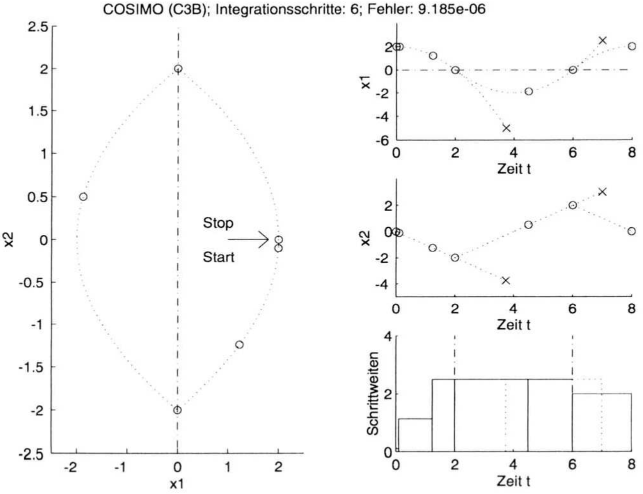


Fig. B.9: *COSIMO (C3B): Detektion; Umschalten der Systemgleichung*

B.3. Toleranz bei der Detektion von Zustandsereignissen

In COSIMO werden die Toleranzen zur Detektion von Zustandsereignissen unabhängig von der Integrationstoleranz vorgegeben. Durch Verkleinern der Detektionstoleranz kann bei konstanter[1] Anzahl Integrationsschritte der Fehler gesenkt werden:

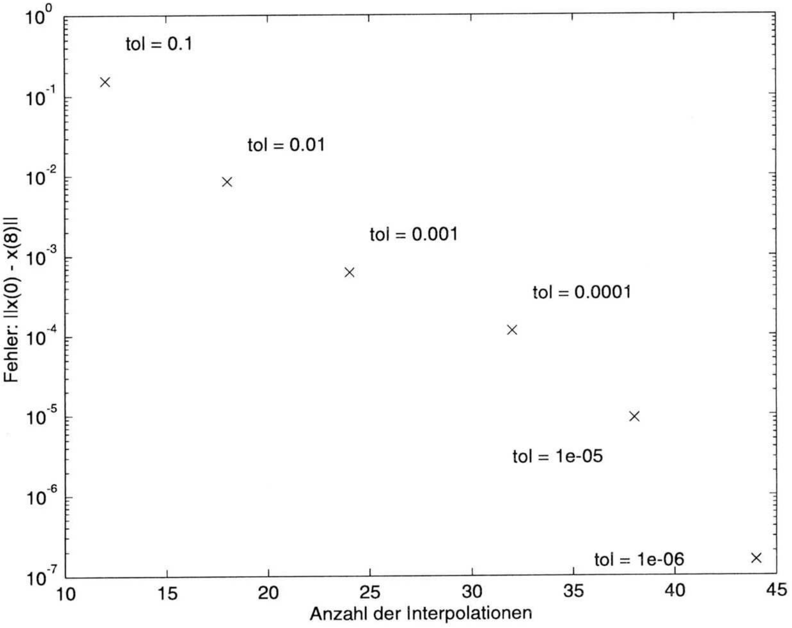


Fig. B.10: Zusammenhang zwischen Detektionstolerenz und Fehler bei C3B

Es ist darauf verzichtet worden, die beiden Detektionsmethoden, iteratives Integrieren bzw. Interpolieren, durch Abschätzen der jeweils nötigen Anzahl Operationen einander gegenüberzustellen, da dies stark vom konkret zu lösenden Problem abhängt. Immerhin kann festgehalten werden, dass die Interpolation zwei klare Vorzüge aufweist: die Berechnungen sind effizient, da sie lokal bei den interessierenden Grössen durchgeführt werden können und weil sie im wesentlichen im Evaluieren einfacher Polynome bestehen.

---

[1] Die Anzahl Integrationsschritte ist nicht per se unabhängig von der Detektionstoleranz; allerdings ist ihr Einfluss äusserst gering.

Anhang C: Objektorientierte Programmierung

Allgemeines

Die Ansichten, was unter objektorientierter Programmierung zu verstehen ist, gehen auseinander. Trotzdem kann ein harter Kern von Eigenschaften herauskristallisiert werden, die notwendigerweise zu einer objektorientierter Sprache gehören. Nach Mössenböck [48] sind dies: Unterstützung des Geheimnisprinzips, Datenabstraktion, Vererbung und dynamische Bindung.

Geheimnisprinzip	Daten werden in einem Baustein gekapselt; die Implementierung bleibt verborgen; auf die Daten kann nicht direkt, sondern nur über Prozeduren, die zum Baustein gehören, zugegriffen werden.
Datenabstraktion	Eine Weiterführung des Geheimnisprinzips; ein abstrakter Datentyp ist eine Einheit aus Daten und darauf anwendbarer Operationen. Verwendung mehrerer Variablen eines Datentyps ist möglich.
Vererbung	Möglichkeit, einen vorhandenen Datentyp zu erweitern, also zusätzliche Daten und Operationen hinzuzufügen, während die bestehenden geerbt werden. Geerbte Operationen können abgeändert werden.
dynamische Bindung	Erst zur Laufzeit wird entschieden, welche Prozedur ausgeführt wird, wenn beispielsweise eine Meldung an ein Objekt geschickt wird.

Tabelle C.1: Notwendige Eigenschaften objektorientierter Programmiersprachen

Von den vier aufgelisteten Eigenschaften ist das Konzept der Vererbung das einzige, welches sich in keiner prozeduralen Sprache findet. Geheimnisprinzip & Datenabstraktion kann beispielsweise in Modula-2 durch die Verwendung von Modulen erreicht werden, dynamische Bindung durch den Einsatz von Prozedurvariablen.

Die Möglichkeiten, die sich durch die Vererbung erschliessen, bauen insbesondere darauf, dass ein erweiterter Datentyp mit dem ursprünglichen Typ kompatibel bleibt. So können z.B., basierend auf einem Grundbaustein, verschiedene Ausprägungen desselben erstellt werden. Für den Grundbaustein implementierte Operationen sind auch auf dessen Verfeinerungen anwendbar. Im weiteren erlaubt die Vererbung auch das Arbeiten mit inhomogenen Listen: Listen, die für einen Grundbaustein implementiert sind, die aber auch Objekte von kompatiblem Typ enthalten können.

Hybride objektorientierte Sprachen

In rein objektorientierten Sprachen wie Smalltalk [28] sind neben Klassen und Meldungen keine anderen Datentypen bzw. Operationen zugelassen. Hybride objektorientierte Sprachen dagegen sind auf prozedurale Sprachen aufgesetzt und beinhalten Elemente beider Welten.

In Oberon, einer hybriden Sprache, können die objektorientierten Begriffe durch herkömmliche Begriffe aus prozeduralen Sprachen ausgedrückt werden:

OO Begriffe	*Herkömmliche (prozedurale) Begriffe*
Klasse	Recordtyp (mit Prozedurfeldern)
Subklasse	erweiterter Typ
Vererbung	Recorderweiterung
Objekt	Variable, Instanz einer Klasse
Methode	Prozedurvariable bzw. typengebundene Prozedur
Handler	Universalmethode, eine als Parameter übergebene Meldung spezifiziert, welche Methode aufgerufen werden soll. Akzeptiert beliebige Erweiterungen des Basismeldungstyps.
Meldung schicken	Aufruf eines Handlers

Tabelle C.2: Terminologie

In objektorientierten Sprachen werden Methoden einem Objekt üblicherweise implizit während seiner Generierung zugewiesen. In (Standard-) Oberon muss der Programmierer explizit dafür sorgen, dass diese Zuweisungen bei der Erzeugung erfolgen.

Oberon-2 [48] und Oberon mit Active Objects [33] bieten Möglichkeiten, typengebundene Prozeduren zu implementieren.

Anhang D: Statistik der Module

Um einen Eindruck über den Umfang von COSIMO und die Grösse der verschiedenen Teile zu geben, werden im folgenden einige Kenndaten aufgelistet. Sämtliche nicht in der Standardverteilung des Gadgets Systems (Version 2.1) enthaltenen Module sind dabei aufgeführt. Ihre Auflistung folgt der Beschreibung in den Kapiteln 3 - 6 und entspricht der Kompilationsreihenfolge.

COSIMO - Module	*Anzahl Zeilen*	*Zeilen netto*	*leere Zeilen*	*Kommentarzeilen*	*Anzahl Worte*	*Anzahl Bytes*
Total	21577	19929	1020	628	106695	728725

Prozentualer Anteil an Kommentarzeilen:	2.91%
Prozentualer Anteil leerer Zeilen:	4.73%
Durchschnittliche Anzahl Zeichen pro Zeile (netto):	35.45

Dienstmodule:

Dateiname	*Anzahl Zeilen*	*Zeilen netto*	*leere Zeilen*	*Kommentarzeilen*	*Anzahl Worte*	*Anzahl Bytes*
GadgetsUtils.Mod	719	620	72	27	3773	24780
MinMax.Mod	25	15	10	0	112	586
SimPanelMacros.Mod	314	273	31	10	2044	13377
NumbersUtils.Mod	62	58	2	2	305	1909
ISOStrings.Mod	480	335	48	97	1505	12597
GraphXY.Mod	1848	1702	87	59	12244	74124
Hands.Mod	310	287	18	5	2509	13451
Histories.Mod	322	297	20	5	2661	14206
Total	4080	3587	288	205	25153	155030

Die Dienstmodule stellen einige grundlegende Funktionalität zur Verfügung, sowie 'normale' Gadgets, die losgelöst von COSIMO in einem beliebigen Kontext verwendet werden können.

Module des Integratorkerns (Kapitel 3):

Dateiname	*Anzahl Zeilen*	*Zeilen netto*	*leere Zeilen*	*Kommentarzeilen*	*Anzahl Worte*	*Anzahl Bytes*
Signals.Mod	519	439	52	28	2007	15455
Integrator.Mod	384	326	41	17	1737	12814
IntegRKF45.Mod	289	216	25	48	1327	9082
IntegDP54.Mod	320	233	30	57	1628	10476
IntegDP86.Mod	482	425	33	24	2788	20268
NoIntegration.Mod	43	27	9	7	116	976
Total	2037	1666	190	181	9603	69071

Module zum Austesten des Integratorkerns (Kapitel 3 & Anhang B):

Dateiname	*Anzahl Zeilen*	*Zeilen netto*	*leere Zeilen*	*Kommentarzeilen*	*Anzahl Worte*	*Anzahl Bytes*
TwoPointC1.Mod	53	41	6	6	162	1285
TwoPointC3B.Mod	47	40	7	0	157	1231
Total	100	81	13	6	319	2516

Module des Editors (Kapitel 4):

Dateiname	*Anzahl Zeilen*	*Zeilen netto*	*leere Zeilen*	*Kommentarzeilen*	*Anzahl Worte*	*Anzahl Bytes*
EditCore.Mod	607	586	12	9	2552	19204
EditFrames.Mod	1443	1397	29	17	6023	45537
EditLinks.Mod	3320	3171	64	85	16139	111157
Total	5467	5237	116	114	25053	178519

Von Interesse ist das Verhältnis der Grössen von Integrationskern und Editor, das zeigt, wie aufwendig das Programmieren graphischer Oberflächen ist. Die Unterschiede rühren in diesem Fall weitgehend davon her, dass einerseits beim Editor viele mögliche verschiedene (Maus-) Aktionen des Benutzers behandelt werden und dass andererseits der Integrationskern in einer ziemlich starren Weise arbeitet.

Module zur Simulation komponierter Modelle (Kapitel 5):

Dateiname	*Anzahl Zeilen*	*Zeilen netto*	*leere Zeilen*	*Kommentarzeilen*	*Anzahl Worte*	*Anzahl Bytes*
ScenarioCore.Mod[a]	97	83	11	3	339	2621
SimuCore.Mod	676	660	13	3	3017	23375
AnestEyes.Mod[b]	88	79	9	0	296	2003
AnestGases.Mod	280	262	17	1	1208	8402
AnestPatients.Mod	56	52	4	0	269	1907
AnestDisturbance.Mod	1782	1776	6	0	6816	36285
Anesthesia.Mod	382	353	27	2	1985	13672
DefuzzyMacros.Mod[c]	146	133	13	0	707	4963
FuzzyMacros.Mod	112	98	14	0	530	3786
Defuzzy.Mod	502	460	25	17	2926	19436
Fuzzy.Mod	378	339	28	11	2021	13126
Fuzz.Mod	1238	1175	54	9	5103	36206
GraphModels.Mod[d]	145	134	10	1	608	4302
BlockModels.Mod	810	743	46	21	3876	27218
NodeModels.Mod[e]	406	363	28	15	1979	14250
Ingots.Mod[f]	914	883	23	8	4163	32139
Segments.Mod[g]	330	308	14	8	1341	10733
Total	8342	7901	342	99	37184	254424

a. Module für Standardszenarien und für Simulationen
b. Fig. 5.5: Anästhesietiefenregelung; kontinuierliches Problem
c. Fig. 5.7: Fuzzy-Blöcke für kontinuierliche Probleme
d. Fig. 5.8: Graphen, Signalgeneratoren, Integratoren, Summatoren usw. für kontinuierliche Probleme
e. Fig. 5.10: Stellen und Transitionen für Petri-Netze; diskrete Probleme
f. Fig. 5.12: Bsp. des Schmelzofens; kombiniertes Problem
g. Fig. 5.13: Bsp. des Wehrs: kombiniertes Problem

Module für alternative Szenarien (Kapitel 6):

Dateiname	*Anzahl Zeilen*	*Zeilen netto*	*leere Zeilen*	*Kommentarzeilen*	*Anzahl Worte*	*Anzahl Bytes*
Scenario.Mod[a]	299	271	19	9	1301	8373
PutGetModels.Mod[b]	344	313	26	5	1402	11478
TimedDiffModels.Mod	88	78	8	2	347	2545
TrainIO.Mod[c]	9	9	0	0	2665	19863
TrainUtilities.Mod	234	220	10	4	937	6825
TrainCore.Mod	229	222	4	3	901	6951
TrainNodeModels.Mod	344	337	7	0	1789	12774
PetriNetAnalyzer.Mod[d]	101	90	8	3	380	2977
Total	1648	1540	82	26	9722	71786

[a] Kap. 6.1: Abarbeiten von Drehbüchern

[b] Kap. 6.2: Regelung des Helikoptermodells (zusätzlich ist eine DLL - dynamic link library - nötig)

[c] Kap. 6.3: Steuerung des Eisenbahnmodells (zusätzlich ist eine DLL nötig)

[d] Kap. 6.4: Analyse von Petrinetzen

Anhang E: Schnittstellen der Module

```
DEFINITION Signals;

   CONST
      getVal*=0;
      infiniteHorizon*=-1;
      insertSegment*=1;
      showInfo*=3;
      showTimes*=4;

   TYPE
      CoeffArray*=POINTER TO ARRAY OF LONGREAL;
      KArray*=POINTER TO ARRAY OF LONGREAL;
      Signal*=POINTER TO SignalDesc;
      SignalDesc*=RECORD
         handle*: SignalHandler;
         next*: Signal;
         x*: LONGREAL;
         xAdr*: LONGINT;
      END;
      SignalHandler*=PROCEDURE(signal: Signal; VAR M: SignalMsg);
      SignalMsg*=RECORD
         id*: INTEGER;
         times*: TimeArray;
         coeffs*: CoeffArray;
      END;
      StateSignal*=POINTER TO StateSignalDesc;
      StateSignalDesc*=RECORD
         (SignalDesc)
         derAdr*: LONGINT;
         k*: KArray;
         nextState*: StateSignal;
      END;
      TimeArray*=POINTER TO ARRAY 3 OF LONGREAL;

   VAR
      algSignalList*: Signal;
      stateSignalList*: StateSignal;

   PROCEDURE CheckLengthOfKArray*(length: INTEGER);
   PROCEDURE CleanUpMonitors*(currentTime: LONGREAL);
   PROCEDURE CorrectMonitors*(deltaT: LONGREAL);
   PROCEDURE DeclAlgSignal*(VAR x: LONGREAL; horizon: LONGREAL);
   PROCEDURE DeclStateSignal*(VAR x, xDot: LONGREAL; x0, horizon: LONGREAL);
   PROCEDURE InitLists*;
   PROCEDURE InitStateSignal*(x: LONGREAL);
   PROCEDURE InitStateSignalBy*(x, val: LONGREAL);
   PROCEDURE InterpolateStateSignals*(time: LONGREAL);
   PROCEDURE ShowGraphAllOutputs*;
   PROCEDURE ShowResultsAll*;
   PROCEDURE ShowResultsTimes*;
   PROCEDURE StateAdr*;
   PROCEDURE StateSize*;

END Signals.
```

```
DEFINITION Integrator;

   IMPORT
      Objects, Files;

   CONST
      AnySlope*=3;
      DownSlope*=2;
      EndOfSim*=-3;
      EventOccured*=-1;
      KeyInterrupt*=-2;
      NoEventOccured*=0;
      NoSlope*=0;
      UpSlope*=1;
      procSet*=2;
      setInteg*=0;
      setThresh*=1;

   TYPE
      Event*=POINTER TO EventDesc;
      EventDesc*=RECORD
         time*: LONGREAL;
         type*: INTEGER;
         model*: Objects.Object;
      END;
      IntegRoutine*=PROCEDURE(VAR step: LONGREAL; minStep, maxStep, t0, tEnd, relTol:
LONGREAL): INTEGER;
      ProcMsg*=RECORD
         (Objects.ObjMsg)
         id*: INTEGER;
         s*: ARRAY 40 OF CHAR;
      END;
      ThreshRoutine*=PROCEDURE(VAR threshOK, eventOccured: BOOLEAN; endOfStep:
LONGREAL; VAR factor: LONGREAL);
      Threshold*=POINTER TO ThresholdDesc;

   VAR
      Dynamic*: PROCEDURE;
      Integ*: IntegRoutine;
      Thresh*: ThreshRoutine;
      calendar*: Event;
      threshList*: Threshold;
      time*: LONGREAL;

   PROCEDURE BS*;
   PROCEDURE Bisection*(VAR threshOK, eventOccured: BOOLEAN; endOfStep: LONGREAL; VAR
factor: LONGREAL);
   PROCEDURE Create*(VAR cal: Event);
   PROCEDURE DeclTH*(VAR var, tar: LONGREAL; relTol, absTol: LONGREAL; status, type:
INTEGER; model: Objects.Object);
   PROCEDURE Empty*(VAR cal: Event): BOOLEAN;
   PROCEDURE EquationsDummy*;
   PROCEDURE GetEvent*(VAR cal, x: Event);
   PROCEDURE GetIntegParameters*(VAR tStart, tEnd, minStep, Step, maxStep, integTol:
LONGREAL);
   PROCEDURE GetNextTime*(cal: Event): LONGREAL;
   PROCEDURE InitLists*;
   PROCEDURE PutEvent*(VAR cal: Event; x: Event);
   PROCEDURE RF*;
   PROCEDURE RegulaFalsi*(VAR threshOK, eventOccured: BOOLEAN; endOfStep: LONGREAL;
VAR factor: LONGREAL);
   PROCEDURE SetBisection*;
   PROCEDURE SetInteg*(s: ARRAY OF CHAR);
   PROCEDURE SetRegulaFalsi*;
   PROCEDURE SetStatusTH*(status, type: INTEGER; model: Objects.Object);
   PROCEDURE SetThresh*(s: ARRAY OF CHAR);
   PROCEDURE SetThresholds*;
   PROCEDURE ShowCalendar*(cal: Event);
   PROCEDURE TestIntegParameters*(VAR step, minStep, maxStep, t0, tEnd, relTol:
LONGREAL);

END Integrator.
```

```
DEFINITION IntegRKF45;

   PROCEDURE RKF45*;
   PROCEDURE RungeKuttaFehlberg45*(VAR step: LONGREAL; minStep, maxStep, t0, tEnd,
relTol: LONGREAL): INTEGER;
   PROCEDURE Set*;

END IntegRKF45.
```

```
DEFINITION EditCore;

   IMPORT
      Gadgets, Objects, Files, Display, Attributes, Texts, Links, Display3, Fonts,
Pictures, Macintosh, Kernel, TB, Oberon, Viewers, Panels, Documents;

   CONST
      changeIn*=8;
      changeOut*=9;
      collectModels*=14;
      establishLink*=7;
      finishDelete*=4;
      finishFrame*=3;
      getModel*=1;
      initGUI*=10;
      leftButton*=6;
      refreshFrame*=2;
      resetConnections*=13;
      selLeftButton*=5;
      showFunc*=11;
      showTopo*=12;

   TYPE
      BroadMsg*=RECORD
         (Objects.ObjMsg)
         id*: INTEGER;
      END;
      CoreMsg*=RECORD
         (Objects.ObjMsg)
         id*, x*, y*, number*: INTEGER;
         model*: Model;
         list*: Objects.Object;
      END;
      Document*=POINTER TO DocumentDesc;
      DocumentDesc*=RECORD
         (Documents.DocumentDesc)
         icon*, menuLib*: ARRAY 40 OF CHAR;
         menu*: ARRAY 100 OF CHAR;
      END;
      GetPanelMsg*=RECORD
         (Display.FrameMsg)
      END;
      Model*=POINTER TO ModelDesc;
      ModelDesc*=RECORD
         (Gadgets.ObjDesc)
         panel*: Panels.Panel;
         frame*: Gadgets.Frame;
         next*: Model;
      END;
      RemoveMsg*=RECORD
         (Display.ControlMsg)
      END;

   VAR
      methods*: Panels.Methods;
      modelList*: Model;
      trackSelection*: BOOLEAN;
```

```
   PROCEDURE CollectModels*(panel: Panels.Panel);
   PROCEDURE CopyModel*(VAR M: Objects.CopyMsg; model, model0: Model);
   PROCEDURE DeleteChildren*(panel: Panels.Panel; VAR M: Display.ControlMsg);
   PROCEDURE DocHandler*(D: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE GetModel*(frame: Gadgets.Frame): Model;
   PROCEDURE InitAliases*;
   PROCEDURE InitFrameGUI*;
   PROCEDURE InsertModelToList*(model: Model);
   PROCEDURE InsertModelToThisList*(model: Model; VAR modelList: Model);
   PROCEDURE InsertPanel*;
   PROCEDURE ModelHandler*(model: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE NewDoc*;
   PROCEDURE NewPanel*;
   PROCEDURE OpenDoc*;
   PROCEDURE PanelHandler*(P: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE ResetConnections*(panel: Panels.Panel);
   PROCEDURE Show*;
   PROCEDURE TransferModel*(transfer: Model; VAR fromList, toList: Model);

END EditCore.


DEFINITION EditFrames;

   IMPORT
      Objects, Files, TextFields, Gadgets, Display, Attributes, Texts, Links,
Display3, Fonts, Pictures, Macintosh, Kernel, TB, Oberon, Viewers, BasicGadgets,
Panels, Icons, Histories, Hands, GraphXY, BasicFigures;

   TYPE
      Button*=POINTER TO BasicGadgets.ButtonDesc;
      CheckBox*=POINTER TO BasicGadgets.CheckBoxDesc;
      Circle*=POINTER TO BasicFigures.FigureDesc;
      Frame*=POINTER TO Gadgets.FrameDesc;
      Graph2D*=POINTER TO GraphXY.XYFrameDesc;
      Hand*=POINTER TO Hands.FrameDesc;
      History*=POINTER TO Histories.FrameDesc;
      Icon*=POINTER TO Icons.IconDesc;
      Iconizer*=POINTER TO Icons.IconizerDesc;
      Panel*=POINTER TO Panels.PanelDesc;
      Slider*=POINTER TO BasicGadgets.SliderDesc;
      TextField*=POINTER TO TextFields.TextFieldDesc;

   PROCEDURE ArrangeNewGroup*;
   PROCEDURE ButtonHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE CheckBoxHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE CircleHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE DeleteGraphHistory*;
   PROCEDURE FrameHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE Graph2DHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE HandHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE HistoryHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE IconHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE IconizerHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE NewButton*;
   PROCEDURE NewCheckBox*;
   PROCEDURE NewCircle*;
   PROCEDURE NewFrame*;
   PROCEDURE NewGraph2D*;
   PROCEDURE NewHand*;
   PROCEDURE NewHistory*;
   PROCEDURE NewIcon*;
   PROCEDURE NewIconizer*;
   PROCEDURE NewPanel*;
   PROCEDURE NewSlider*;
   PROCEDURE NewTextField*;
   PROCEDURE OpenGroup*;
   PROCEDURE PanelHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE SliderHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE TextFieldHandler*(frame: Objects.Object; VAR M: Objects.ObjMsg);

END EditFrames.
```

```
DEFINITION EditLinks;

   IMPORT
      EditCore, Gadgets, Objects, Files, Display, Attributes, Texts, Links, Display3,
Fonts, Pictures, Macintosh, Kernel, TB, Oberon, Viewers, Panels, Documents;

   CONST
      NIO*=10;

   TYPE
      ConnList*=POINTER TO ConnListDesc;
      ConnListDesc*=RECORD
         model*: Model;
         next*: ConnList;
      END;
      Connection*=POINTER TO ConnectionDesc;
      ConnectionDesc*=RECORD
         (Gadgets.FrameDesc)
         signal*: Signal;
      END;
      GroupModel*=POINTER TO GroupModelDesc;
      GroupModelDesc*=RECORD
         (ModelDesc)
      END;
      Model*=POINTER TO ModelDesc;
      ModelDesc*=RECORD
         (EditCore.ModelDesc)
         NI*, NO*: INTEGER;
         inListOfNodes*, outListOfNodes*: ConnList;
         in*, out*: ARRAY 10 OF BlockConn;
      END;
      Signal*=POINTER TO SignalDesc;
      SignalDesc*=RECORD
         (Gadgets.ObjDesc)
         value*: LONGREAL;
         atDisposal*: BOOLEAN;
      END;

   PROCEDURE ConnHasInput*(conn: BlockConn): BOOLEAN;
   PROCEDURE CopyModel*(VAR M: Objects.CopyMsg; model, model0: Model);
   PROCEDURE GroupModelHandler*(groupModel: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE IncNrOfInputs*;
   PROCEDURE IncNrOfOutputs*;
   PROCEDURE ModelHandler*(model: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE NewArrow*;
   PROCEDURE NewBlockConn*;
   PROCEDURE NewCircle*;
   PROCEDURE NewGroupModel*;
   PROCEDURE NewModel*;
   PROCEDURE NewNodeConn*;
   PROCEDURE NewRectLine*;
   PROCEDURE NewSignal*;
   PROCEDURE RotateFrame*;
   PROCEDURE SetIO*(obj: Model; dir, NI, NO: INTEGER; fixedInBlocks, fixedOutBlocks,
fixedInNodes, fixedOutNodes: BOOLEAN);
   PROCEDURE TestLink*;

END EditLinks.
```

```
DEFINITION ScenarioCore;

   IMPORT
      EditCore, Gadgets, Objects, Files, Display, Attributes, Texts, Links, Display3,
Fonts, Pictures, Macintosh, Kernel, TB, Oberon, Viewers, Panels, Documents;

   CONST
      accepted*=1;
      collectData*=3;
      evaluate*=1;
      handleEvent*=4;
      initialize*=0;
      refused*=2;
      schedEvent*=5;
      undefined*=0;
      visualize*=2;

   TYPE
      CoreMsg*=RECORD
         (Objects.ObjMsg)
         id*, result*: INTEGER;
      END;

   PROCEDURE ModelsInitialized*(VAR fromList, toList: EditCore.Model): BOOLEAN;
   PROCEDURE SendMsgToModels*(msgID: INTEGER; model: EditCore.Model);

END ScenarioCore.
```

```
DEFINITION SimuCore;

   IMPORT
      Oberon, Viewers, Display, Objects, Files, Texts, Fonts, EditCore, Gadgets,
Attributes, Links, Display3, Pictures, Macintosh, Kernel, TB, Panels, Documents,
BasicGadgets, Signals, Integrator;

   CONST
      paused*=-2;
      running*=-1;
      startPos*=0;

   TYPE
      Panel*=POINTER TO PanelDesc;
      PanelDesc*=RECORD
         (Panels.PanelDesc)
         scheduler*: Scheduler;
         status*: INTEGER;
         simTime*: LONGREAL;
         calendar*: Integrator.Event;
      END;

   PROCEDURE Buttons*;
   PROCEDURE Continue*;
   PROCEDURE CopyPanel*(VAR M: Objects.CopyMsg; panel, panel0: Panel);
   PROCEDURE GetSimuPanel*(model: EditCore.Model; VAR panel: Panel): BOOLEAN;
   PROCEDURE InsertPanel*;
   PROCEDURE NewDoc*;
   PROCEDURE NewPanel*;
   PROCEDURE OpenDoc*;
   PROCEDURE Pause*;
   PROCEDURE Reset*;
   PROCEDURE Schedule*(me: Oberon.Task);
   PROCEDURE Start*;
   PROCEDURE Stop*;

END SimuCore.
```

```
DEFINITION NodeModels;

   IMPORT
      Objects, Files, EditLinks, EditCore, Gadgets, Display, Attributes, Texts,
Links, Display3, Fonts, Pictures, Macintosh, Kernel, TB, Oberon, Viewers, Panels,
Documents, BasicGadgets, Integrator;

   TYPE
      Place*=POINTER TO PlaceDesc;
      PlaceDesc*=RECORD
         (EditLinks.ModelDesc)
      END;
      Trans*=POINTER TO TransDesc;
      TransDesc*=RECORD
         (EditLinks.ModelDesc)
      END;

   PROCEDURE CopyPlace*(VAR M: Objects.CopyMsg; place, place0: Place);
   PROCEDURE CopyTrans*(VAR M: Objects.CopyMsg; trans, trans0: Trans);
   PROCEDURE FireTransition*;
   PROCEDURE InitPlace*(place: Place);
   PROCEDURE InitTrans*(trans: Trans);
   PROCEDURE NewPlace*;
   PROCEDURE NewTrans*;
   PROCEDURE PlaceHandler*(place: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE SwitchPlaceModel*;
   PROCEDURE TransEnabled*(trans: Trans): BOOLEAN;
   PROCEDURE TransHandler*(trans: Objects.Object; VAR M: Objects.ObjMsg);

END NodeModels.
```

```
DEFINITION GraphModels;

   IMPORT
      Objects, Files, EditLinks, EditCore, Gadgets, Display, Attributes, Texts,
Links, Display3, Fonts, Pictures, Macintosh, Kernel, TB, Oberon, Viewers, Panels,
Documents, GraphXY;

   TYPE
      Graph2D*=POINTER TO Graph2DDesc;
      Graph2DDesc*=RECORD
         (EditLinks.ModelDesc)
      END;

   PROCEDURE Graph2DHandler*(obj: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE NewGraph2D*;

END GraphModels.
```

```
DEFINITION BlockModels;

   IMPORT
      Objects, Files, Gadgets, Display, Attributes, Texts, Links, Display3, Fonts,
Pictures, Macintosh, Kernel, TB, Oberon, Viewers, ScenarioCore, EditCore, Panels,
Documents, EditLinks, BasicGadgets;

   CONST
      constant*=1;
      hertz*=1;
      rad*=0;
      sawtooth*=5;
      sine*=3;
      square*=4;
      time*=2;

   TYPE
      Gain*=POINTER TO GainDesc;
      GainDesc*=RECORD
         (EditLinks.ModelDesc)
      END;
      Gen*=POINTER TO GenDesc;
      GenDesc*=RECORD
         (EditLinks.ModelDesc)
      END;
      Input*=POINTER TO InputDesc;
      InputDesc*=RECORD
         (EditLinks.ModelDesc)
      END;
      Integ*=POINTER TO IntegDesc;
      IntegDesc*=RECORD
         (EditLinks.ModelDesc)
      END;
      Meter*=POINTER TO MeterDesc;
      MeterDesc*=RECORD
         (EditLinks.ModelDesc)
      END;
      Output*=POINTER TO OutputDesc;
      OutputDesc*=RECORD
         (EditLinks.ModelDesc)
      END;
      Prod*=POINTER TO ProdDesc;
      ProdDesc*=RECORD
         (EditLinks.ModelDesc)
      END;
      Sum*=POINTER TO SumDesc;
      SumDesc*=RECORD
         (EditLinks.ModelDesc)
      END;

   PROCEDURE GainHandler*(obj: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE GenHandler*(obj: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE InputHandler*(obj: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE IntegBlockInit*(model: Gadgets.Object; VAR M: ScenarioCore.CoreMsg);
   PROCEDURE IntegHandler*(obj: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE MeterHandler*(obj: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE NewGain*;
   PROCEDURE NewGen*;
   PROCEDURE NewInput*;
   PROCEDURE NewInteg*;
   PROCEDURE NewMeter*;
   PROCEDURE NewOutput*;
   PROCEDURE NewProd*;
   PROCEDURE NewSum*;
   PROCEDURE OutputHandler*(obj: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE ProdHandler*(obj: Objects.Object; VAR M: Objects.ObjMsg);
   PROCEDURE StaticBlockInit*(model: Gadgets.Object; VAR M: ScenarioCore.CoreMsg);
   PROCEDURE SumHandler*(obj: Objects.Object; VAR M: Objects.ObjMsg);

END BlockModels.
```

Anhang F: Literaturverzeichnis

[1] Alur, R., Courcoubetis, C., Henzinger, T. und Ho, P. (1993): *An Algorithmic Approach to the Specification and Verification of Hybrid Systems.* In *Hybrid Systems*, Lecture Notes in Computer Science 736, pp. 209-229, Springer.

[2] Andersson, M. (1990): Omola: *An Object-Oriented Language for Model Representation.* Lund Institute of Technology, Sweden, Report No. LUTFD2/(TFRT-3208)/1-102.

[3] Andersson, M. (1994): *Object-Oriented Modeling and Simulation of Hybrid Systems.* Dissertation, Lund Institute of Technology.

[4] Åström, K., J. (1985): *A Simnon Tutorial.* Lund Institute of Technology, Sweden, Report No. LUTFD2/(TFRT-3176)/1-87

[5] Augustin, D. C. et. al. (1967): *The SCi Continuous System Simulation Language (CSSL).* SIMULATION, pp. 281 -303.

[6] Baccelli, F., Cohen, G., Olsder, G. J. und Quadrat, J.-P. (1991): *Synchronization and Linearity: an Algebra for Discrete Event Systems.*. John Wiley & Sons, Chichester.

[7] Balemi, S. (1992): *Control of Discrete Event Systems: Theory and Application.* Dissertation, ETH Zürich.

[8] Branicky, M. S. (1994): *Analyzing continuous switching systems: Theory and exaamples.* In Proc. American Control Conf., pp. 3110-3114, Baltimore.

[9] Branicky, M. S. (1994): *Stability of Switched and Hybrid Systems.* In Proc. IEEE Conf. Decision and Control, pp. 3498-3503, Lake Buena Vista.

[10] Brenan, K. E., Campbell, S. L. und Petzold, L. R. (1989): *Numerical solution of initial-value problems in differential-algebraic equations.* Elsevier, New York.

[11] Cellier, F.E. (1979): *Combined Continuous/Discrete System by Use of Digital Computers: Techniques and Tools,* Dissertation, ETH Zürich.

[12] Cellier, F. E., Elmquist, H., Otter, M. und Taylor, J. H. (1993): *Guidelines for Modeling and Simulation of Hybrid Systems.* Proc. IFAC World Congress, vol. 8, pp. 391-397, Sidney, Australien.

[13] Curtiss, C. F. und Hirschfelder, J. O. (1952): *Integration of stiff equations.* Proceedings of the National Academy of Sciences of U. S., Vol. 38, pp. 235-243.

[14] David, R. (1995): *Grafcet: A Powerful Tool for Specification of Logic Controllers.* IEEE Trans. Contr. Syst. Tech., vol. 3, pp. 253-268.

[15] David, R. und Alla, H. (1992): *Petri Nets & Grafcet: Tools for modelling discrete event systems.* Prentice Hall.

[16] Davidts, P., (1996): *Graphical Fuzzy Tool.* Diplomarbeit Nr. 8795, Institut für Automatik, ETH Zürich.

[17] Dormand, J. R. und Prince, P. J. (1989): *Practical Runge-Kutta processes.* SIAM Journal, Sci. Stat. Comput., vol. 10, pp. 977-989.

[18] Ellis, R. D., Ravikanth, R. und Kumar, P. R. (1994): *Automating the Simulation of Complex Discrete-Time Control Systems: A Mathematical Framework, Algorithms, ans a Software Package.* IEEE Trans. Automat. Contr., vol. AC-39, no. 9, pp. 1795 - 1801.

[19] Elmqvist, H., Cellier, F., Otter, M., (1993): *Objekt-Oriented Modeling of Hybrid Systems.* European simulation Symposium, Delft.

[20] Elmqvist, H., Cellier, F., Otter, M., (1994): *Objekt-Oriented Modeling of Power-Electronic Circuits Using Dymola.* First Joint Conference of International Simulation Societies Proceedings, Zürich, pp. 156-161.

[21] Enright, W. H., Jackson, K. R., Nørsett, S. P. und Thomsen, P. G. (1988): *Effective Solution of Discontinuous IVPs Using a Runge-Kutta Formula Pair with Interpolants.* Appl. Math. and Comput., Vol. 27, pp. 313-335.

[22] Fahrland, D. (1970): *Combined discrete event continuous systems simulation.* SIMULATION, Vol. 14, Nr. 2, pp. 61-72.

[23] Frauenstein, T., Pape, U. und Wagner, O. (1990): *Objektorientierte Sprachkonzepte und Diskrete Simulation.* Springer-Verlag, Berlin.

[24] Frei, C. W., Derighetti, M. und Zbinden, A. M. (1997): *Modeling for Control of Mean Arterial Blood Pressure (MAP) During Anesthesia.* 2nd MATHMOD, Wien.

[25] Fischlin, A. (1987): *The Dialog Machine for the Macintosh.* ETH Zürich, Projekt-Zentrum IDA, Interner Bericht.

[26] Fischlin, A., Gyalistras, D., Roth, O., Ulrich, M., Thöny, J., Nemecek, T., Bugmann, H., Thommen, F. (1994): *Modelworks 2.2: An Interactive Simulation Environment for Personal Computers and Workstations.* Report Nr. 14, Institut für terrestrische Ökologie, ETH Zürich.

[27] Ganz, C. (1995): *Visual Aspects of Computer Aided Control Systems Design.* Dissertation, ETH Zürich.

[28] Goldberg, A. und Robson, D. (1983): *Smalltalk-80, the language and its implementation.* Addison-Wesley.

[29] Greim, T. (1994): *Simulation und Analyse von Steuerungssystemen auf der Basis von Petri-Netzen.* Dissertation, Ruhr-Universität, Bochum.

[30] Gutknecht, J. (1994): *Oberon System 3 Vision of a Future Software Technology.* Software - Concepts and Tools 15, pp. 26-33.

[31] Gutknecht, J. (1994): *Oberon - Perspectives of Evolution.* In Schulthess P. (ed.): *Advances in Modular Languages.* Proceedings of the Joint Modular Languages Conference, Ulm.

[32] Gutknecht, J. (1996): *Oberon, Gadgets and Some Archetypal Aspects of Persistent Objects.*

[33] Gutknecht, J. (1997): *Do the fish really need remote control?* Proceedings of the JMLC, Linz, Österreich.

[34] Hairer E., Nørsett, S.P. und Wanner, G. (1987): *Solving Ordinary Differential Equations I.* Volume 8 of Springer Series in Computational Mathematics.

[35] Hairer E. und Wanner, G. (1991): *Solving Ordinary Differential Equations II.* Volume 14 of Springer Series in Computational Mathematics.

[36] Iwasaki, Y. (1991): *Causal Ordering Analysis.* In Fishwick, P. A., und Luker, P. A. (Eds.): *Qualitative Simulation Modeling Analysis.* Springer, New York.

[37] Johannsen, G. (1993): *Mensch-Maschine-Systeme.* Springer-Verlag, Berlin.

[38] Koch, G. (1993): *Modular Reasoning: A new Approach Towards Intelligent Control.* Dissertation, ETH Zürich.

[39] Kolb, P. (1995): *Nutzung der objektorientierten Methodologie für den computerunterstützten Entwurf von Regelsystemen.* Dissertation, ETH Zürich.

[40] Kottmann M. (1996): *Combined Modelling and Simulation of Dynamic Systems using Oberon.* Proceedings of the IEEE International Symposium on CACSD, Dearborn, pp. 358-362.

[41] Laurel, B. (1991): *Computers as Theatre.* Addison-Wesley, Reading.

[42] Lorenz, E. N. (1979): *On the Prevalence of Aperiodicity in Simple Systems.* In Grmela, M. und Marsden, J. E. (Eds.): *Lecture Notes in Mathematics,* Vol. 755, pp. 53-75, Global Analysis, Calgary.

[43] Marais, J. L. (1991): *The GADGETS User Interface Management System.* Structured Programming, vol. 12, pp. 75-89.

[44] Marais, J. L. (1994): *Towards End-User Objects: The Gadgets User Interface System.* In Schulthess P. (ed.): *Advances in Modular Languages.* Proceedings of the Joint Modular Languages Conference, Ulm.

[45] Marais, J. L. (1996): *Design and Implementation of a Compound Architecture for Oberon.* Dissertation, ETH Zürich.

[46] Mattsson, S. E. und Andersson, M. (1990): *A Kernel for System Representation.* Proc. of the 11th IFAC World Congress, pp 91-96, Tallinn.

[47] Montigel, M. (1992): *Elemente eines computergestützten Werkzeugs zur Entwicklung von Eisenbahnsicherungsanlagen mit Petri-Netzen.* Schriftenreihe des IVT, ETH, Zürich.

[48] Mössenböck H. (1993): *Objektorientierte Programmierung in Oberon-2.* Springer-Verlag, Berlin.

[49] Nicolet, A. (1995): *Programme de simulation de la pharmacocinétique et de la pharmacodynamique des anesthésiques par inhalation.* Dissertation, Université de Berne.

[50] Niemeyer, G. (1977): *Kybernetische System- und Modelltheorie.* System Dynamics, Vahlen, München.

[51] Ören, T. I. (1989): *Simulation Models: Taxonomy.* In Singh, M. (Ed.): *Encyclopedia of Systems and Control.* Pergamon Press, NewYork.

[52] Page, B. (1991): *Diskrete Simulation. Eine Einführung mit Modula-2.* Springer-Verlag, Berlin.

[53] Park, T. und Barton, P.I. (1994): *Towards Dynamic Simulation of a Process and its Automatic Protective System.* International Symposium and Workshop on Safe Chemical Process Automation, Houston.

[54] Park, T. und Barton, P.I. (1996): *State Event Location in Differential-Algebraic Models.* ACM Transactions on Modeling and Computer Simulation, vol. 6, pp. 137-165.

[55] Pegden, C. D., Shannon, R. E., Sadowski, R. P. (1990): *Introduction to simulation Using SIMAN.* McGraw-Hill, New York.

[56] Peitgen, H.-O., Jürgens, H. und Saupe, D. (1992): *Bausteine des Chaos: Fraktale.* Springer-Verlag, Berlin

[57] Poincaré, H. (1892): *Les méthodes nouvelles de la mécanique céleste.*, Tome I: *Solutions périodiques, non-existence des integrals uniformes, solutions asymptotique*, Gauthier-Villars, Paris.

[58] Poincaré, H. (1993): *New Methods of Celestial Mechanics*, Part I: *Periodic and Asymptotic Solutions*, Goroff, D. L. (Editor & Vorwort), American Institute of Physics.

[59] Prähofer, H. (1991): *System theoretic foundations for combined discrete-continuous system simulation.* Dissertation, J. Kepler-Universität Linz.

[60] Pretschner, A. und Büttner, R. (1996): *A Contribution to the Design and Simulation of Discrete-Event Driven Systems.* Proceedings of the IEEE International Symposium on CACSD, Dearborn, pp 363-367.

[61] Pritsker, A. A. B. (1974): *The Gasp IV Simulation Language.* John Wiley & Sons, New York.

[62] Pritsker, A. A. B. (1986): *Introduction to Simulation and SLAM II.* John Wiley & Sons, 3. Auflage, New York.

[63] Rasmussen, J. (1990): *Mental Models and the Control of Action in Complex Environments.* In Ackermann, D. und Tauber, M. J. (eds.): *Mental Models and Human-Computer Interaction 1.* Elsevier Science, North Holland.

[64] Reiser, M. (1991): *The Oberon System. User Guide and Programmer's Manual.* Addison-Wesley.

[65] Rose, C. et al. (1985): *Inside Macintosh.* Vol. I - III, Addison-Wesley, Reading.

[66] Sastry, S., Bodson, M. (1989): *Adaptive Control: Stability, Convergence, and Robustness.* Prentice-Hall, New Jersey.

[67] Schaufelberger, W. und Xiaobing, Q. (1994): *Control Engineering Courseware Development with Oberon.* Proc. International Conference on Computer Aided Learning and Instruction in Science and Engineering, Paris, pp. 477-480.

[68] Standridge, C. R. und Pritsker, A. A. B. (1982): *Using Data Base Capabilities in Simulation.* In Cellier, F. E. (Ed.): *Progress in Modelling and Simulation.* Academic Press.

[69] Stiver, J. A. und Antsaklis, P. J. (1992): *Modeling and Analysis of Hybrid Control Systems.* Proc. 31st. IEEE CDC, Tucson, Arizona.

[70] Taylor, J. H. (1995): *Rigorous Handling of State Events in MATLAB.* Proc. IEEE Conference on Control Applications, pp 156-161, Albany, NY.

[71] Vancso-Polacsek, K. (1990): *Theory and Practice of Computer Assisted Simulation and Modeling on Professional Workstations.* Dissertation, ETH Zürich.

[72] Van Dixhoorn, J. J. (1982): *Bond Graphs and the Challenge of a Unified Modelling Theory of Physical Systems.* In Cellier, F. E. (ed.): *Progress in Modelling & Simulation.* Academic Press. London.

[73] Willems, J.C. (1991): *Paradigms and Puzzles in the Theory of Dynamical Systems*, IEEE Trans. Automat. Contr., vol. AC-36, no. 3, pp. 259-294.

[74] Wirth, N. und Gutknecht, J. (1992): *Project Oberon.* Addison Wesley.

[75] Wonham, W. M., Ramadge, P. J. (1988): *Modular Supervisory Control of Discrete Event Systems.* Mathematics of Control Signals and Systems, 1:13-30.

[76] Wymore, A. W. (1993): *Model-Based Systems Engineering.* CRC Press, Boca Aton.

[77] Zadeh, L.A. und Desoer, C.A. (1963): *Linear Systems Theory*, MacGraw Hill, New York.

[78] Zeigler, B. P. (1976): *Theory of Modelling and Simulation.* John Wiley, New York.

[79] Zeigler, B. P. (1984): *System Theoretic Foundations of Modelling and Simulation.* In Ören, T.I., Zeigler, B. P., Elzas, M.S. (Eds.): *Simulation and Model Based Methodologies: An Integrative View*, Springer-Verlag.

Anhang G: Sachwortregister

Curriculum Vitae

---

5. April 1964	geboren in St. Gallen, Schweiz.
1971 - 1979	Primar- und Sekundarschule in St. Gallen.
1979 - 1983	Kantonsschule am Burggraben, Matura Typus C (mathematisch-naturwissenschaftlich).
1984 - 1990	Studium der Elektrotechnik an der ETH Zürich, Diplom als Dipl. El.-Ing. ETH.
1991	Anstellung als Assistent am Institut für Automatik der ETHZ
1993	Nachdiplom in Informationstechnik der Abteilung für Elektrotechnik.
1994 - 1997	Konzipierung und Implementierung des in dieser Dissertation beschriebenen Modellierungs- und Simulationswerkzeugs. Teile der Arbeit wurden im Rahmen des Schulratsmillion-Projekts 'Discrete Event Modeling and Control of Urban Traffic Systems', sowie über einen ETHZ-internen Forschungskredit finanziert.