CARL VON OSSIETZKY universität OLDENBURG

Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Mechatronic Processing Objects - Eine verarbeitungsorientierte Modellrepräsentation als Basis einer offenen Entwurfsumgebung für mechatronische Systeme

Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften

vorgelegt von
Dipl.-Inform. Carsten Homburg

Tag der Disputation: 14. Mai 2012

Erstreferent: Prof. Dr. rer. nat. Achim Rettberg
Koreferent: Prof. Dr.-Ing. Axel Hahn

Zusammenfassung

Beim modellbasierten Entwurf eines mechatronischen Systems wird der Ingenieur mit der Herausforderung konfrontiert, eine Vielfalt von unterschiedlichen physikalischen Wirkprinzipien in einem Gesamtsystem abbilden zu können. Zum anderen muss er in der Lage sein, Mischszenarien von im Rechner modellierten und teilweise real existierenden Komponenten unter Echtzeitbedingungen nachzustellen. Basierend auf der verarbeitungsorientierten Modellrepräsentation wird eine offene Entwurfsumgebung vorgestellt, die es ermöglicht, ein Modell durchgängig in allen Entwicklungsschritten zu verwenden. Abbildbar sind modular-hierarchische, gemischt diskret-kontinuierlichen Modelle und dynamisch rekonfigurierbare Systemstrukturen bis hin zu komplexen, autonomen Systemstrukturen. Unterstützt wird eine homogene Modellintegration mit transparentem Modell-Debugging, interpretative Modellauswertung, Codegenerierung für Simulation unter Echtzeitbedingungen als auch der Einsatz linearer Verfahren.

Abstract

One challenge in the model-based design of mechatronic systems is to integrate a variety of different physical principles into one single, combined model of an entire system. In addition, for rapid control prototyping and hardware-in-the-loop scenarios, models need to be executed under real-time conditions, connected to real hardware. The open design environment presented here is based on processing-oriented model representation and supports the use of one model throughout all development stages. Processing-oriented model representation is able to describe modular, hierarchical models that can be continuous, discrete, or mixed, and dynamically reconfigurable system structures — even complex, autonomous systems. It supports homogenous model integration with transparent model debugging, interpretative model execution, code generation for real-time simulation, and linear control theory methods.

Danksagung

Ich möchte mich bei allen, die mich während meiner Zeit als Doktorand begleitet und unterstützt haben, ganz herzlich bedanken.

Zunächst möchte ich da Prof. Dr.-Ing. Joachim Lückel (†28.11.2008) nennen, der während meiner Zeit als Mitarbeiter am Mechatronik Laboratorium Paderborn die initialen Impulse zu dieser Arbeit gegeben hat.

Mein besonderer Dank gilt Prof. Dr. rer. nat. Achim Rettberg, der die Betreuung meiner Promotion übernommen hat. Durch seine Unterstützung und sein stetiges großes Vertrauen in meine Arbeit konnte meine Promotion erfolgreich abgeschlossen werden.

Darüber hinaus danke ich auch meinem Zweitgutachter Prof. Dr.-Ing. Axel Hahn für seine hilfreichen Anmerkungen und die kritische Durchsicht dieser Dissertation.

Bei meinen ehemaligen Kollegen vom Mechatronik Laboratorium Paderborn möchte ich mich für die konstruktiven Diskussionen bedanken, die sehr wertvolle Anregungen für die Erstellung meiner Dissertation geliefert haben. Mein Dank geht ebenfalls an die Arbeitsgruppe von Prof. Rettberg für das konstruktive Feedback beim internen Kolloquium an der Carl von Ossietzky Universität Oldenburg.

Auch meinen Eltern Gisela und Wilfried Homburg möchte ich Danke sagen. Sie haben mir das Streben nach Bildung als wichtigen Wert vermittelt und mich zum Abitur und der anschließenden Aufnahme eines Hochschulstudiums motiviert.

Abschließend möchte ich mich besonders herzlich bei meiner Frau Sonja und meinen Kindern Julian und Johanna bedanken. Ohne ihre liebevolle familiäre Unterstützung wäre diese Arbeit nicht möglich gewesen. Sie haben mir den zeitlichen Freiraum geschaffen, der für die Erstellung dieser Arbeit nötig war. Dabei mussten sie an vielen Abenden und Wochenenden auf mich verzichten.

Carsten Homburg

Inhaltsverzeichnis

Kapitel 1

Einleitung

Betrachtet man den Aufbau und die Funktionsprinzipien von Maschinen und anderen technischen Systemen, so lässt sich beobachten, dass der Anteil der elektronischen und informationsverarbeitenden Komponenten von Generation zu Generation zunimmt. Anstatt aufwändiger, materialintensiver konstruktiver Änderungen zur Beseitigung von Unzulänglichkeiten der Mechanik wird moderne Regelungstechnik in Form von elektronischen und informationsverarbeitenden Komponenten zur Kompensation der mechanischen Unzulänglichkeiten eingesetzt. Durch Ausstattung mit intelligenten, informationsverarbeitenden Komponenten lassen sich neue, innovative Produkte entwickeln, deren Leistungsfähigkeit erst durch die Kombination von Maschinenbau, Elektrotechnik und Informatik möglich ist. Systeme, die auf einer Kombination von Wirkprinzipien aus diesen Einzeldisziplinen beruhen, werden *mechatronische Systeme* genannt.

Repräsentative Beispiele für mechatronische Systeme sind Kraftfahrzeuge oder auch Bahnsysteme. Für einen nachhaltigen Umgang mit unserer Umwelt müssen energiesparende und emissionsarme Antriebskonzepte entwickelt werden. Beispiele aus diesem Bereich sind eine Start-Stop-Automatik, die den Motor im Stand automatisch abschaltet sowie der Hybridantrieb, der einen Betrieb des Verbrennungsmotors im optimalen Betriebspunkt erlaubt. Ebenso besteht bei dem stetig wachsenden Verkehrsaufkommen ein Bedarf von intelligenten Verkehrsmanagementsystemen, die eine möglichst störungsfreie Mobilität für jeden einzelnen ermöglichen. Für den schienengebundenen Verkehr besteht Bedarf an neuen, flexiblen Antriebskonzepten wie dem RailCab, die auch effizient für den Individualverkehr einsetzbar sind.

1.1 Motivation

An den aufgeführten Beispielen wird deutlich, dass sich mechatronische Systeme aufgrund ihrer Komplexität nicht mehr nach dem klassischen Trial-and-Error-Verfahren entwickeln lassen. Zum einen ist aufgrund der Breite

der beteiligten Fachdisziplinen ein vielschichtiges Wissen erforderlich, zum anderen ist es aus Zeit- und Kostengründen nicht möglich, zunächst Prototypen für alle beteiligten Systemkomponenten zu bauen und diese dann sukzessiv zusammenzufügen und iterativ zu verbessern. Eine durchgängige Rechnerunterstützung bei der Entwicklung eines mechatronischen Systems ist daher unabdingbar. Hat der Ingenieur früher jede neue Idee und jeden Verbesserungsschritt am Prüfstand oder durch Prototypen überprüft, so werden diese Laborversuche mehr und mehr in den Rechner verlagert. Anstatt des Prototypen wird ein Modell des Systems im Rechner erstellt, an dem zunächst die Systemanalyse und der Reglerentwurf durchgeführt werden. Der Bau eines Prototypen erfolgt jetzt erst deutlich später, wenn aufgrund der am Rechner bereits durchgeführten Untersuchungen bereits abgesicherte Vorstellungen über das zu entwickelnde System existieren. Das spart Entwicklungszeiten und führt aufgrund der Betrachtung des gesamten Systems im Rechner zu einer optimaleren Auslegung des Gesamtsystemverhaltens.

Für eine effektive Unterstützung des Ingenieurs bei der Entwicklung eines mechatronischen Systems ist es wünschenswert, dass eine Entwicklungsumgebung zur Verfügung steht, die eine durchgängige Verwendung derselben Modelle für alle Entwicklungsschritte ermöglicht. Dabei stellt sich die besondere Herausforderung, dass auf der einen Seite auf der Modellierungsebene eine Vielfalt von unterschiedlichen physikalischen Wirkprinzipien abbildbar sein muss. Auf der anderen Seite muss eine Echtzeitumgebung zur Verfügung stehen, mit der Mischszenarien von teilweise rein im Rechner modellierten und teilweise real existierenden Komponenten untersucht werden können (sogenannte Hardware-in-the-Loop- bzw. Rapid-Control-Prototyping-Szenarien).

1.2 Ziele und Aufbau der Arbeit

Ziel der vorliegenden Arbeit ist es, die Basis für eine offene Entwurfsumgebung für mechatronische Systeme zu schaffen, mit der der Ingenieur durchgängig in allen Entwicklungsschritten arbeiten kann. Auf der einen Seite muss sie die Modellierung von Teilkomponenten aus unterschiedlichsten Fachdisziplinen unterstützen, auf der anderen Seite soll sie modular aufgebaut und einfach um neue Entwurfs- und Analysewerkzeuge erweiterbar sein. Konkret sollen mit der in dieser Arbeit vorgestellten offenen Entwicklungsumgebung hinsichtlich der Modellierung die folgenden Ziele umgesetzt werden:

- Unterstützung der Modellierung von modular-hierarchischen Strukturen;
- homogene Integration von Komponenten unterschiedlicher Fachdiszi-

plinen;

- transparente Anknüpfung an ursprüngliche Modellspezifikationen in allen weiterverarbeitenden Werkzeugen;
- schnelle Umsetzbarkeit von Modelländerungen;
- Unterstützung bei der Suche nach Modellierungsfehlern.

Bezüglich der Modellauswertung sind die folgenden Ziele aufgestellt:

- Abbildung linearer Systeme als Matrizen und Verkopplung symbolischer, linearer Teilsysteme;
- Bereitstellung einer schlanken und portablen Software-Architektur für die Modellauswertung, die auch auf Mikrocontroller anwendbar ist;
- Echtzeitfähigkeit;
- Unterstützung der Parallelverarbeitung;
- automatische Codegenerierung von C-Programmcode für laufzeitkritische Anwendungen;
- Anbindung von Sensorik und Aktorik.

Aus softwaretechnischer Sicht ergeben sich weitere Ziele. Die Entwicklungsumgebung soll modular erweiterbar sein, d. h.:

- neue Verfahren zur Simulation, linearen Analyse, Reglersynthese oder sonstigen Verarbeitung sollen hinzugefügt werden können, ohne dass neue Modellierungs-Frontends geschaffen werden müssen. Die vorhandenen Frontends sollen von den neuen Methoden benutzt werden können.
- Ebenfalls soll zur Unterstützung einer weiteren, neuen fachspezifischen Modellbeschreibungssprache nur die Erstellung eines zusätzlichen Modellierungs-Frontends erforderlich sein, alle weiterverarbeitenden und auswertenden Werkzeuge soll unverändert benutzt werden können.
- Zur Reduktion der Komplexität der einzelnen Modellierungs-Frontends und weiterverarbeitenden Werkzeuge sollen für die Weiterverarbeitung im Rechner relevante Aspekte der einzelnen Teilmodelle in Form einer verarbeitungsorientierten Modellrepräsentation vorliegen. Diese verarbeitungstechnisch relevanten Informationen sollen von den einzelnen Werkzeugen direkt genutzt werden können. Eine separate Implementierung der Ermittlung dieser Informationen in jedem weiterverarbeitenden Werkzeug soll nicht erforderlich sein.

Zur Erreichung dieser Ziele sind die Mechatronic Processing Objects und die ihnen zugrunde liegende verarbeitungsorientierte Modellbeschreibungssprache DSC (**D**ynamic **S**ystem **C**ode) entwickelt worden, die in der vorliegenden Arbeit vorgestellt werden.

In **Kapitel 2** erfolgt zunächst eine Einordnung dieser Arbeit in den Gesamtkontext des rechnergestützten Entwurfs mechatronischer Systeme und der dort eingesetzten Modellbeschreibungsformen. Nach Vorstellung des Entwurfsprozesses für mechatronische Systeme wird eine Klassifikation der Modellrepräsentationen gemäß ihres Abstraktionsgrads in drei Modellbeschreibungsebenen vorgenommen. Das Kernthema dieser Arbeit ist die unterste dieser Beschreibungsebenen, die *verarbeitungsorientierte Modellrepräsentation*. Dem wichtigen Thema der Integration unterschiedlichst modellierter Teilmodelle ist anschließend ein eigenes Unterkapitel gewidmet. Es folgt ein Überblick über die aktuellen mechatronischen Entwurfsumgebungen. Ihre Leistungsfähigkeit wird untereinander und insbesondere mit dem in dieser Arbeit verfolgten Ansatz verglichen. Zur Einordnung der Arbeit in den Gesamtkontext der Softwareaktivitäten am MLaP (**M**echatronik **La**boratorium **P**aderborn) erfolgt abschließend eine kurze Vorstellung der vorhandenen bzw. projektierten Softwarewerkzeuge und der umschließenden Entwurfsumgebung CAMeL (**C**omputer-**A**ided **M**echatronics **L**aboratory).

In **Kapitel 3** wird die Gesamtkonzeption der im Rahmen dieser Arbeit entwickelten offenen, verarbeitungsorientierten Modellrepräsentation vorgestellt. Ausgehend vom Entwurfsprozess für mechatronische Systeme wird das Anforderungsprofil an die verarbeitungsorientierte Modellrepräsentation und die damit verbundene Entwurfsumgebung abgeleitet und anschließend das Gesamtkonzept der offenen Entwurfsumgebung auf der Basis von Mechatronic Processing Objects und der verarbeitungsorientierten Modellbeschreibungsfrom DSC im Überblick dargestellt.

Kapitel 4 enthält die detaillierte Vorstellung der Mechatronic Processing Objects (MPO) und ihrer Repräsentation. Zunächst wird der modulare, anwendungsabhängige Aufbau eines MPO dargestellt. Es folgt die Vorstellung des zugrundeliegenden mathematischen Modells, das die Beschreibung modular-hierarchischer, gemischt diskret-kontinuierlicher Systeme erlaubt. Anschließend erfolgt eine Vorstellung der Interaktionen, die Mechatronic Processing Objects durchführen können. Abschließend werden zentrale Aspekte der Implementierung vorgestellt. Zur Implementierungsunterstützung wurde im Rahmen dieser Arbeit IRL2 (**I**ntermediate **R**epresentation **L**anguage **2**) als Spezifikationssprache für komplexe Datenstrukturen entwickelt, mit deren Hilfe ein Großteil der Implementierung eines MPO aus einer einfachen, kompakten Spezifikation automatisch generiert werden kann.

In den **Kapiteln 5** bis **8** werden Aspekte des Einsatzes der Mechatronic Processing Objects bei den verschiedenen Entwurfsschritten eines mechatronischen Systems beschrieben. **Kapitel 5** behandelt die Simulation mechatro-

nischer Systeme. Dabei wird sowohl auf die interpretative Modellauswertung als auch auf die Erzeugung von C-Code zur Auswertung der Systemgleichungen eingegangen. Zum Abschluss des Kapitels wird das Simulationswerkzeug SIMBA vorgestellt, in dem die vorgestellten Konzepte umgesetzt worden sind. **Kapitel 6** ist der verteilten Informationsverarbeitung und der Umsetzung von autonomen Systemen gewidmet. Es wird das Konzept der *Cluster* zur Partitionierung von verteilten Systemen vorgestellt. Zur Abbildung von autonomen Systemen wird der Aufbau von dynamischen Systemstrukturen mit Mechatronic Processing Objects beschrieben. **Kapitel 7** behandelt den Einsatz der Mechatronic Processing Objects bei den Entwurfsverfahren der linearen Analyse und Synthese. **Kapitel 8** beschäftigt sich mit der Fehlersuche in den Modellen mechatronischer Systeme. Es wird ein Anforderungsprofil für einen Modell-Debugger aufgestellt und ein entsprechendes Lösungskonzept vorgestellt. Abschließend wird der im Rahmen dieser Arbeit entstandene Prototyp eines Debuggers auf Basis der verarbeitungsorientierten Beschreibungsform DSC vorgestellt.

Kapitel 9 vermittelt dem Leser anhand diverser Anwendungsbeispiele einen Gesamteindruck über die Einsatzmöglichkeiten der Mechatronic Processing Objects. Die vorgestellten Projekte eines seriellen Hybridantriebs, eines dezentralen Kreuzungsmanagements sowie eines Nutzfahrzeugs sind allesamt am MLaP unter Einsatz der Mechatronic Processing Objects und der verarbeitungsorientierten Modellrepräsentation DSC durchgeführt worden. An den Anwendungsbeispielen werden insbesondere die Aspekte der Modellintegration, der Umsatzung von dynamischen Systemstrukturen und der verteilten Informationsverarbeitung beleuchtet.

Abschließend fasst **Kapitel 10** die wesentlichen Aspekte noch einmal zusammen und gibt einen Ausblick auf zukünftige Entwicklungen und Einsatzgebiete.

Kapitel 2

Modellbeschreibungen und Werkzeuge für den Entwurf mechatronischer Systeme

Mechatronische Systeme bestehen aus einer Vielzahl von mechanischen, hydraulischen, pneumatischen und elektrischen Komponenten, verknüpft durch Informationsverarbeitung. Aufgrund der hohen Komplexität ist es nicht mehr möglich, diese Systeme allein von Hand, also im Trial-and-Error-Verfahren, zu entwickeln. Zur Modellierung und Entwicklung mechatronischer Systeme steht bereits eine Vielzahl von Modellbeschreibungssprachen und Entwurfswerkzeugen zur Verfügung. Im vorliegenden Kapitel soll ein Überblick über die derzeit eingesetzten Modellrepräsentationen und Entwicklungsumgebungen auf dem Gebiet der Mechatronik gegeben werden.

Zunächst wird anhand des Entwicklungskreislaufs die typische Vorgehensweise beim rechnergestützten Entwurf mechatronischer Produkte vorgestellt. In den einzelnen Entwurfsphasen kommen dabei unterschiedliche Modellbeschreibungen zum Einsatz. Diese Modellrepräsentationen eines mechatronischen Produkts werden in Kapitel 2.2 näher betrachtet. Es werden drei Beschreibungsebenen mechatronischer Systeme vorgestellt, denen sich die einzelnen Modellrepräsentationen abhängig von ihrem Abstraktionsgrad und ihrem Informationsgehalt zuordnen lassen. Das Kernthema dieser Arbeit ist die unterste Beschreibungsebene, die *verarbeitungsorientierte Modellrepräsentation*.

Mechatronische Systeme bestehen typischerweise aus einer Vielzahl von Komponenten mit unterschiedlichen physikalischen Wirkprinzipien. Die einzelnen Komponenten werden daher häufig in fachspezifischen Modellierungssprachen beschrieben, die dann in das Gesamtsystem integriert werden müssen. Mit diesem Thema der Modellintegration beschäftigt sich Kapitel 2.3.

In Kapitel 2.4 wird ein Überblick über aktuelle mechatronische Entwurfsumgebungen gegeben. Ihre Leistungsfähigkeit wird untereinander und

insbesondere mit dem in dieser Arbeit vorgestellten Ansatz der Mechatronic Processing Objects verglichen.

Abschließend wird die am Mechatronik Laboratorium Paderborn (MLaP) entstandene mechatronische Entwurfsumgebung CAMeL vorgestellt. Sie stellt den Rahmen dar, in dem die in dieser Arbeit vorgestellten Mechatronic Processing Objects als verarbeitungsorientierte Modellrepräsentation eingesetzt werden.

2.1 Der rechnergestützte Entwurf mechatronischer Systeme

Für den systematischen Entwurf mechatronischer Systeme wurde am Mechatronik Laboratorium Paderborn (MLaP) ein Entwicklungskreislauf [Jae91, Wä01] formuliert, der die einzelnen Entwurfsschritte und die eventuell dabei notwendigen Iterationen beschreibt (Abbildung 2.1). Am Anfang und

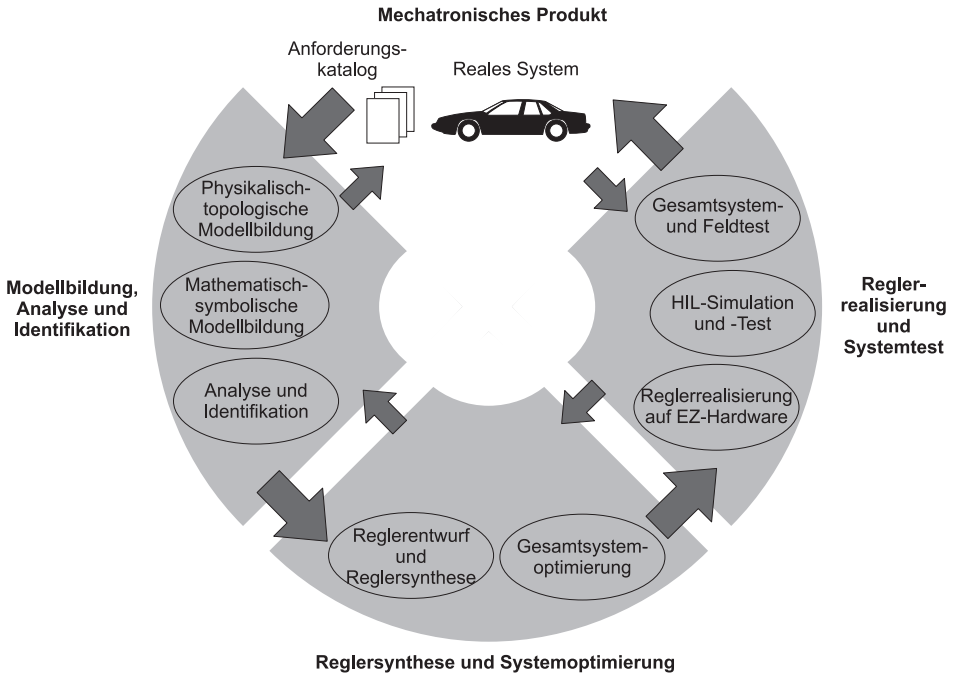


Abbildung 2.1: Entwicklungskreislauf mechatronischer Systeme

Ende des Entwicklungskreislaufs steht jeweils ein mechatronisches Produkt. Den Ausgangspunkt bildet im Allgemeinen ein existierendes Produkt, das hinsichtlich einer Funktion oder Eigenschaft verbessert werden soll, oder eine Produktidee, die umgesetzt werden soll. Der hier vorgestellte Entwicklungskreislauf konzentriert sich dabei auf das eigentliche Produkt von der Konzipierung über den Entwurf bis hin zur Ausarbeitung in Form eines

Prototypen. Aspekte der Produktfindung, der Fertigungsplanung und der Fertigung werden hier nicht weiter betrachtet, eine Einbettung des Entwicklungskreislaufs in eine allgemeine Entwurfssystematik für Produkte der Mechatronik ist in [GL00] dargestellt.

2.1.1 Modellbildung, Analyse und Identifikation

Grundvoraussetzung für den rechnergestützten Entwurf ist die Abbildung der relevanten Aspekte des Systemverhaltens im Rechner in Form eines Modells. Dieser Prozess wird als *Modellbildung* bezeichnet. Aufgrund der Komplexität und der interdisziplinären Natur mechatronischer Systeme ist es sinnvoll, die Modellbildung in zwei Schritten zu betreiben. Zunächst wird ein *topologisch-physikalisches Modell* erstellt. Dies beschreibt die relevanten physikalischen Phänomene der beteiligten Komponenten sowie die Wechselwirkungen zwischen den Komponenten und ihre Verknüpfungsstrukturen.

Ausgehend vom topologisch-physikalischen Modell wird im nächsten Schritt das *mathematisch-symbolische Modell* aufgestellt. Dabei werden die physikalischen Phänomene und Wechselwirkungen durch mathematische Gleichungen beschrieben. Die mathematisch-symbolische Modellbildung kann sehr effizient durch Computeralgebra-Programme unterstützt werden.

Ist nun ein mathematisches Modell erstellt, so kann im nächsten Schritt die *rechnergestützte Analyse des Systemverhaltens* erfolgen. Zunächst kann mit Hilfe der nichtlinearen Simulation das zeitliche Verhalten des Systems überprüft werden. Der Ingenieur erhält damit qualitative Aussagen über das modellierte Systemverhalten. Decken sich diese Aussagen nicht mit dem Produktverhalten, so ist das Modell noch unzureichend und muss korrigiert oder erweitert werden. Die nichtlineare Simulation ist ein sehr wichtiges Mittel zur ersten Überprüfung des Modells. Der Ingenieur muss sich aber dabei stets bewusst machen, dass es sich dabei nur um das Nachspielen einiger weniger Szenarien handelt, bestimmt durch den Satz der verwendeten Stimuli. Die aus der nichtlinearen Simulation gewonnenen Aussagen haben nur exemplarischen Charakter, es sind keine allgemeinen Aussagen über das Systemverhalten.

Allgemeine strukturelle Aussagen zum System können mittels der linearen Analyse des Systemverhaltens gewonnen werden. Dabei liegt das System für einen gewissen Betriebspunkt als lineares Modell vor. Auf Basis dieses linearen Modells können dann die Analysemethoden der klassischen Regelungstechnik angewendet werden. Hier sind insbesondere die Eigenwert- und Frequenzanalyse zu nennen, die strukturelle Aussagen zur Dynamik des Systems liefern wie z. B. zur Stabilität. Aber auch hier ist zu beachten, dass diese Aussagen nur um den gewählten Betriebspunkt gelten. Daher wird ein System in der Regel um verschiedene Betriebspunkte linearisiert und analysiert.

Ein weiterer wichtiger Prozess in dieser Entwicklungsphase ist die *Iden-*

tifikation des Systems. Ein Modell eines mechatronischen Systems enthält in der Regel eine Menge von Modellparametern. Diese müssen anhand von Messungen am realen System bzw. an den schon existierenden Komponenten abgeglichen werden, damit das Systemverhalten nicht nur qualitativ, sondern auch quantitativ hinreichend genau abgebildet wird.

2.1.2 Reglersynthese und Systemoptimierung

Steht ein hinreichend genaues Modell der realen Strecke, d. h. des zu regelnden Teilsystems, zur Verfügung, so kann auf dieser Basis die *Reglersynthese* erfolgen. Unter Berücksichtigung der vorhandenen und beobachtbaren Messgrößen wählt der Ingenieur Regelgrößen und eine Reglerstruktur aus, um das Systemverhalten der Strecke gemäß den gestellten Anforderungen zu beeinflussen. In diesem Schritt fließt das regelungstechnische Erfahrungswissen des Ingenieurs ein.

Ist eine Reglerstruktur gefunden, so kann dafür die Optimierung der Regler- und Streckenparameter durchgeführt werden. Dies erfolgt anhand des nun um die Reglerstrukturen erweiterten Modells. Wird dabei das Systemverhalten durch lineare Beziehungen hinreichend genau beschrieben, so können wiederum die Methoden der klassischen Regelungstechnik verwendet werden, um einen optimalen Parametersatz analytisch zu bestimmen. Bei nichtlinearen Synthesemethoden erfolgt die Optimierung von System- und Reglerparametern auf Basis der nichtlinearen Simulation. Durch Parametervariation erzeugt ein Optimierungsalgorithmus unterschiedliche Sätze von Simulations-Zeitdaten, die jeweils mit den Referenzdaten verglichen werden. Optimierungsziel ist dabei die Minimierung der Differenz.

2.1.3 Reglerrealisierung und Systemtest

Genügt das Verhalten des im Rechner abgebildeten Systems den gestellten Anforderungen, so kann der Übergang zum realen System erfolgen. Dieser Übergang sollte sinnvollerweise aber erst dann erfolgen, wenn das Gesamtsystemverhalten im Rechner ausreichend getestet wurde, da ab diesem Zeitpunkt das Durchführen von Veränderungen am System deutlich kostenaufwendiger wird. Liegt das System vollständig im Rechner vor, so lassen sich Veränderungen relativ schnell im Rechner durchführen. Veränderungen am realen Prototypen mit seiner aufwendigen Messtechnik erfordern hingegen einen deutlich höheren Aufwand. Daher erfolgt der Übergang vom rein im Rechner simulierten Modell zum realen Prototypen schrittweise: Bei der *Hardware-in-the-Loop-Simulation (HIL)* wird nur ein Teil des Systems auf einem Prüfstand in realer Hardware aufgebaut, während der Rest des Systems in Echtzeit auf einer Rapid-Prototyping-Hardware mit leistungsfähigen Prozessoren und entsprechender I/O-Hardware im geschlossenen Kreis simuliert wird. Auch die Reglerfunktionalität wird häufig zunächst auf dem Rapid-

Prototyping-System realisiert, das erheblich mehr Rechenleistung und Bedienungskomfort bietet als die endgültige Zielhardware.

Zur Realisierung der Regler werden im Allgemeinen Codegeneratoren verwendet, die anhand des Modells automatisch den entsprechenden Programmcode erzeugen. Der typischerweise in der Programmiersprache C vorliegende Programmcode wird dann in Maschinencode übersetzt, mit weiteren Modulen zusammengebunden und auf die Echtzeithardware heruntergeladen.

Haben sich die entwickelten Reglerfunktionalitäten und Systemkomponenten im HIL-Betrieb bewährt, so kann der Feldtest ausgeführt werden, in dem die neu entwickelten Prototypen im realen Betrieb getestet werden. Hat sich der Prototyp im Feldtest bewährt, so ist ein neues, verbessertes mechatronisches Produkt entstanden. Der Übergang zur Serienentwicklung schließt den hier vorgestellten Entwicklungszyklus ab.

2.2 Beschreibungsebenen mechatronischer Systeme

Zur Gewinnung von aussagekräftigen und interpretierbaren Ergebnissen müssen mechatronische Systeme aufgrund ihres hohen Komplexitätsgrads unter unterschiedlichen Sichtweisen untersucht und betrachtet werden können. Ordnet man diese Sichtweisen nach ihrem Abstraktionsgrad, so bilden sich drei unterschiedliche Ebenen der Modellbeschreibung heraus. Diese drei Beschreibungsebenen des topologischen, des Verhaltens- und des Verarbeitungsmodells sind in dem am MLaP entwickelten objektorientierten Mechatronikmodell (**O**bject-oriented **M**echatronic **M**odel = **OMM**) gebündelt [Hah99]. Einen Überblick über das OMM und die drei Beschreibungsebenen mechatronischer Systeme gibt die Abbildung 2.2. Eine Vorstellung der einzelnen Modellbeschreibungsebenen erfolgt in den nachfolgenden Unterkapiteln. Für die am MLaP entwickelten Sprachen Objective-DSS (Objective-**D**ynamic **S**ystem **S**tructure), Objective-DSL (Objective-**D**ynamic **S**ystem **L**anguage) und DSC (**D**ynamic **S**ystem **C**ode) gibt es Werkzeuge, die eine automatische Modelltransformation zur jeweils tieferen Ebene durchführen können.

2.2.1 Physikalische, topologische Strukturbeschreibung

Bei der Modellbeschreibungsebene mit dem höchsten Abstraktionsgrad handelt es sich um die anwendernahe, strukturorientierte Ebene. Der Ingenieur beschreibt im Wesentlichen die Topologie seines Systems, die Verknüpfung der einzelnen Komponenten. Dabei werden häufig einfach Komponenten aus einer Bibliothek genommen, dem konkreten System entsprechend parametrisiert und entsprechend der Gesamtsystemstruktur miteinander verknüpft.

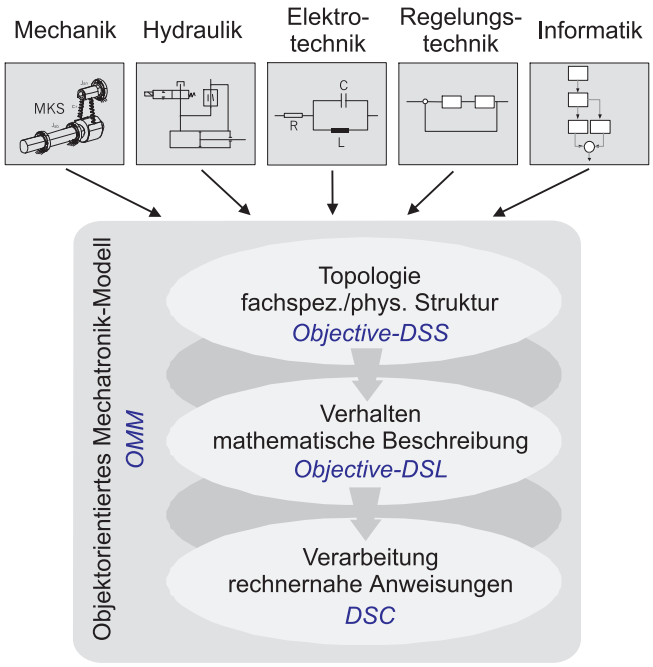


Abbildung 2.2: Objektorientiertes Mechatronikmodell (OMM)

Die Spezifikation des physikalischen Sachverhalts der einzelnen Komponenten erfolgt mit fachspezifischen Beschreibungsformen der Mechanik, Hydraulik, Elektrotechnik oder Informationsverarbeitung. Die einzelnen Disziplinen selbst verwenden wiederum unterschiedliche Bauteile und Methoden zur Hierarchie- und Strukturbildung. Im Bereich der Mechanik werden z. B. Mehrkörpersystemmodelle als Topologiegraphen von starr verbundenen, elementaren mechanischen Bauteilen (z. B. starre Körper, Gelenke, Aktoren u. a.) beschrieben. Beim Schaltkreisentwurf in der Elektrotechnik wird ein Schaltkreis meist über Netzlisten modelliert, die ebenfalls die Topologie der verwendeten Bauteile abbilden.

Ein Vorteil topologischer Modelle ist, dass noch keine Aussagen über die mathematische Beschreibung eines Systems notwendig sind. Der Ingenieur kann sich bei der ersten Konzipierung seines Systems auf die physikalischen Wirkprinzipien beschränken und nach dem Prinzip eines Modulbaukastens das Gesamtsystem erstellen. Das mathematische Modell des Systemverhaltens kann dann in einem weiteren Schritt aus dem topologischen Modell abgeleitet werden.

2.2.2 Mathematische Modelle kontinuierlicher, diskreter und hybrider Systeme

Soll das Systemverhalten eines mechatronischen Systems untersucht werden, so muss das Systemverhalten in Form eines mathematischen Modells vorliegen. Mit den Methoden der Mathematik und Rechentechnik können aus dieser mathematischen Modellbeschreibung dann das Zeitverhalten und andere Kenngrößen des Systems ermittelt werden. Ein Großteil der verfügbaren Modellbeschreibungssprachen ist dieser mittleren Ebene der mathematischen Beschreibung zuzuordnen.

Weit verbreitet bei der mathematischen Modellierung mechatronischer Systeme sind die Zustandsraumdarstellung 1. Ordnung für kontinuierliche Systeme, Bedingung/Ereignis-Systeme zur Modellierung von diskreten und gemischt diskret-kontinuierlichen Systemen und hybride Automaten zur Modellierung von hybriden Systemen.

Zustandsraumdarstellung 1. Ordnung

Das Systemverhalten kontinuierlicher Systeme wird üblicherweise durch ein System von Differentialgleichungen beschrieben. Dabei hat sich die explizite Form der Zustandsraumdarstellung 1. Ordnung bewährt:

$$\underline{\dot{x}} = f(\underline{x}, \underline{u}, \underline{p}, t) \tag{2.1}$$

$$\underline{y} = g(\underline{\dot{x}}, \underline{x}, \underline{u}, \underline{p}, t) \tag{2.2}$$

Das Systemverhalten wird durch eine Systemgleichung (2.1) und eine Ausgangsgleichung (2.2) beschrieben. Beschreibende Größen des Systemverhaltens sind die Eingangsgrößen $\underline{u}$, die Ausgangsgrößen $\underline{y}$, die Zustandsgrößen $\underline{x}$ sowie die für eine konkrete Systeminstanz festen Parametergrößen $\underline{p}$.

Aufgrund der expliziten Form des Differentialgleichungssystems ist die Zustandsraumdarstellung 1. Ordnung sehr gut für die Modellierung komplexer, hierarchischer Systeme geeignet. Bei der Aggregation von mehreren Teilmodellen müssen keine algebraischen Umformungen an den Gleichungen der einzelnen Teilsysteme durchgeführt werden, die mathematischen Modelle der einzelnen Teilsysteme können für das mathematische Modell des Gesamtsystems als unveränderte Module übernommen werden. Der Aufbau von hierarchischen Modellstrukturen erfolgt üblicherweise mit der Blockschaltbild-Darstellung. Nach dem Ursache-Wirkungs-Prinzip können die einzelnen Teilsysteme über ihre Ein- und Ausgangsgrößen miteinander verknüpft werden.

Bedingung/Ereignis-Systeme

Bedingung/Ereignis-Systeme (B/E-Systeme) [SK91, Kro93] sind eine Modellform zur modularen Beschreibung von Systemen mit diskretem oder ge-

mischt diskret-kontinuierlichem Systemverhalten. Sie ermöglichen die Anwendung der bei kontinuierlichen Systemen üblichen Blockschaltbild-Darstellung auch für diskrete Systeme.

Kennzeichnend für die Beschreibung des Verhaltens diskreter Systemaspekte mit B/E-Systemen ist die Unterteilung der diskreten Signale in zwei Arten:

Bedingungssignale (B-Signale) Diese Signale dienen zur Übertragung von Informationen über aktuelle Zustände. Diese Werte liegen kontinuierlich vor, Wertänderungen erfolgen aber nur zu diskreten Zeitpunkten. Formal gesehen ist ein B-Signal eine abschnittsweise konstante Funktion, die die Zeit in eine Menge von Bedingungen abbildet. An den Sprungstellen gilt der rechtsseitige Grenzwert als Funktionswert.

Ereignissignale (E-Signale) Diese Signale liegen nur punktuell vor und erzwingen diskrete Zustandsübergänge bzw. spiegeln solche wieder. An diesen Signalen können die Ereignisse im System abgelesen werden. Formal gesehen sind E-Signale Zeitfunktionen, die nur zu diskreten Zeitpunkten Werte aus einer Menge von Ereignissen annehmen, die übrige Zeit ist ihr Funktionswert Null.

Die Beschreibung des inneren Systemverhaltens eines B/E-Systems ist in keiner Weise festgelegt. Kommuniziert das System mit seiner Umgebung nur über B- und E-Signale, so kann jede Art der Modellbeschreibung für sein inneres Systemverhalten verwendet werden. Damit ist eine beliebige Kombination von B/E-Systemen mit unterschiedlichsten Beschreibungsformen des jeweils inneren Systemverhaltens möglich. Die graphische Notation eines B/E-Systems ist in Abbildung 2.3 dargestellt.

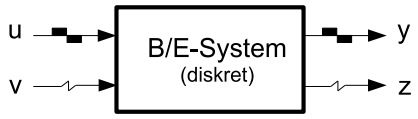


Abbildung 2.3: Graphische Notation eines B/E-Blocks

Diskrete B/E-Systeme

Eine übliche Methode zur Modellierung von diskreten B/E-Systemen ist die Beschreibung der inneren Dynamik auf Grundlage einer endlichen Menge X von diskreten Zuständen [SK91]. Die innere Systemdynamik und das Ausgabeverhalten sind durch die folgenden drei Funktionen definiert:

$$x(t) \in f(x(t^-), u(t^-), v(t)), \qquad x(0) = x_0 \in X, \tag{2.3}$$

$$y(t) = g(x(t), u(t)), \tag{2.4}$$

$$z(t) = h(x(t^-), x(t), v(t)) \tag{2.5}$$

Die Ausdrücke $x(t^-)$ und $u(t^-)$ sind dabei Abkürzungen für $lim_{\Delta t \to 0} x(t - \Delta t)$ bzw. $lim_{\Delta t \to 0} u(t - \Delta t)$, geben also jeweils den zuletzt gültigen Wert vor dem Zeitpunkt t an.

Die *Zustandsübergangsfunktion* f gibt nach Gleichung 2.3 für jeden Zustand und jeden Wert der beiden Eingangssignale die *Menge* der möglichen Folgezustände an. Damit ist das Verhalten eines diskreten B/E-Systems im allgemeinen Fall nichtdeterministisch. Im Weiteren soll jedoch nur der für mechatronische Systeme relevante, deterministische Fall betrachtet werden. Für Bedingungs- und Ereignissignale existiert jeweils eine eigene Ausgabefunktion. Die Bedingungsausgabefunktion g (2.4) entspricht der für Zustandssysteme üblichen Form. Bei der Ereignisausgabefunktion h (2.5) ist der Ausgabewert sowohl vom alten als auch vom neuen Zustand abhängig. Damit können Ereignisse als Reaktion auf einen internen Zustandswechsel ausgesendet werden. Zur sauberen Trennung der Wirkung von Bedingungs- und Ereignissignalen und damit zur Gewährleistung einer einfachen, übersichtlichen Modellierung müssen die Funktionen f und h die Nebenbedingungen $x \in f(x, u, 0)$ und $0 = h(x, x, 0)$ für alle $x(t)$ und $u(t)$ einhalten. Zustandsübergänge und Ereignisausgaben können also nur durch eingehende Ereignisse bzw. andere Zustandsübergänge ausgelöst werden.

Hybride B/E-Systeme

Zur Modellierung von gemischt diskret-kontinuierlichen Systemen wurde der Sprachumfang der B/E-Systeme um zwei neue Blocktypen erweitert [Kro93]. Die beiden neuen Blocktypen sind in Abbildung 2.4 dargestellt. Zur Ver-

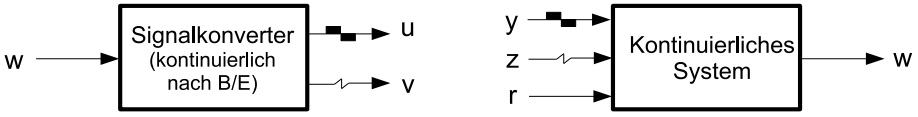


Abbildung 2.4: B/E-Blöcke für gemischt diskret-kontinuierliche Systeme

bindung einer kontinuierlichen Ausgangsgröße (w) mit einem B/E-System steht ein Signalkonverter-Block zur Verfügung, der das Ausgangssignal des kontinuierlichen Systems in Bedingungs- (u) und Ereignissignale (v) als Eingangssignale des B/E-Systems umwandelt. Das Zeitverhalten $u(t)$ und $v(t)$ kann dabei jeweils als Funktion von $w(t)$ spezifiziert werden.

Die Beschreibung des kontinuierlichen Blocks basiert auf der Zustandsraumdarstellung 1. Ordnung. Mit dem kontinuierlichen Eingang r, dem kontinuierlichen Ausgang w und dem kontinuierlichen Zustand ξ entspricht die Ausgangsgleichung (2.6) vollständig der Zustandsraumdarstellung 1. Ordnung:

$$w(t) \quad = \quad \rho(\xi(t), r(t), t) \tag{2.6}$$

Zusätzlich erhält der kontinuierliche Block einen Bedingungs- (y) und einen Ereignis-Eingang (z). Der Bedingungseingang dient direkt zur Auswahl der rechten Seite der kontinuierlichen Zustandsgleichung, kenntlich am Subskript $y(t^-)$ der Zustandsfunktion ϕ:

$$\dot{\xi}(t) \quad = \quad \phi_{y(t^-)}(\xi(t^-), r(t^-), t) \tag{2.7}$$

Die Zustandsgleichung 2.7 ist immer dann gültig, wenn der Ereigniseingang z den Wert 0 hat, also kein Ereignis anliegt. Liegt ein Ereignis an, so ist das gleichbedeutend mit einer Unstetigkeit im kontinuierlichen Systemverhalten und die kontinuierlichen Zustandsgrößen müssen mit neuen Werten besetzt werden. modelliert durch die Ereigniszustandsübergangsfunktion E:

$$\xi(t) \quad = \quad E(\xi(t^-), z(t)), \quad z(t) \neq 0 \tag{2.8}$$

Hybride Automaten

Zur Beschreibung einer inneren Systemdynamik mit sowohl diskreten als auch kontinuierlichen Anteilen bieten sich die hybriden Automaten an [Hen96]. Ein hybrider Automat H ist ein Tupel

$$H =< X, V, E, \Sigma, init, inv, flow, jump, event >$$

mit

- den kontinuierlichen Zustandsvariablen $X = x_1, \ldots, x_n$. Die Zahl n wird als die *Dimension* von H bezeichnet.
- dem gerichteten, endlichen Graphen (V, E), auch als *Kontrollgraph* bezeichnet. Ein Knoten aus V heißt *Kontrollmodus*, die Kanten aus E werden *Kontrollübergänge* genannt.
- der endlichen Menge von Events Σ.
- den Anfangsbedingungen $init : V \rightarrow P(X)$. Jede Anfangsbedingung $init(v)$ ist ein Prädikat, dessen freie Variablen aus X sind. Sie gibt an, welche Bedingung beim Eintritt in den Kontrollmodus v gültig ist.
- den Invarianzbedingungen $invariant : V \rightarrow P(X)$. Jede Invarianzbedingung $invariant(v)$ ist ein Prädikat, dessen freie Variablen aus X sind. Sie gibt an, welche Bedingung ununterbrochen gültig ist, solange der Kontrollmodus v aktiv ist.
- den Flussbedingungen $flow : V \rightarrow P(X \cup \dot{X})$. Jede Flussbedingung $flow(x)$ ist ein Prädikat, dessen freie Variablen aus X und $\dot{X}$ sind. Sie beschreibt das kontinuierliche Systemverhalten für den Kontrollmodus v.

- den Sprungbedingungen $jump : E \to X \cup X'$. Jede Sprungbedingung $jump(e)$ ist ein Prädikat, dessen freie Variablen aus X und X' sind. Sie beschreibt für den Kontrollübergang e, unter welcher Bedingung er ausgelöst wird und welchen Wert X' die kontinuierlichen Zustandsvariablen nach Abschluss des Kontrollwechsels haben.

- der Event-Zuordnungsfunktion $event : E \to \Sigma$. Sie ordnet jedem Kontrollübergang e einen Event σ zu, der ausgelöst wird, wenn der Kontrollübergang schaltet.

Ein Beispiel für einen hybriden Automaten ist in Abbildung 2.5 dargestellt. Es modelliert einen Temperaturregler. Die Variable x beschreibt die Temperatur. Im Kontrollmodus *Aus* ist die Heizung aus und die Temperatur sinkt gemäß der Flussbedingung $\dot{x} = -0.1x$. Im Kontrollmodus *Ein* ist die Heizung in Betrieb und die Temperatur steigt gemäß der Flussbedingung $\dot{x} = 5 - 0.1x$. Zu Beginn ist die Heizung aus und die Temperatur hat den Wert 20. Gemäß der Sprungbedingung $x < 19$ schaltet die Heizung ein, sobald die Temperatur den Wert 19 unterschreitet. Zusätzlich sorgt die Invarianzbedingung $x \geq 18$ dafür, dass die Heizung spätestens einschaltet, wenn die Temperatur auf den Wert 18 sinkt.

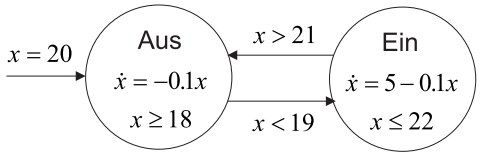


Abbildung 2.5: Temperaturregler als Hybrider Automat [Hen96]

2.2.3 Verarbeitungsorientierte Modellrepräsentation

Hat man das Gesamtsystem in Form eines mathematischen Modells vorliegen, so kann dieses noch nicht direkt im Rechner abgearbeitet werden. Aus den hierarchisch verkoppelten Gleichungen muss ein Berechnungsgraph erzeugt werden, der explizit die Berechnung des gesamten Systemverhaltens angibt. Dazu muss zunächst das mathematische Modell auf syntaktische und semantische Korrektheit überprüft werden. Namen von Systemgrößen müssen identifiziert und den Knoten des Berechnungsgraphen zugeordnet werden. Bei imperativen Formulierungen zur Berechnung von Modellgrößen sind weitere Transformationen durchzuführen wie die Eliminierung von Mehrfachzuweisungen an eine Variable und die Umwandlung von bedingten Anweisungen in funktionale Konstrukte. Zusätzlich sollten aus der hierarchischen Modellierung stammende Verkopplungen aus Laufzeitgründen bis auf Gleichungsebene komprimiert werden. Damit erhält man einen kompakten, flachen Graphen, der das Systemverhalten des Modells in

Form von rein funktionalen Zusammenhängen beschreibt, Hierarchien sind eliminiert. Durch topologisches Sortieren des gesamten Graphen erhält man eine korrekte sequentielle Auswertereihenfolge.

Dieser Berechnungsgraph ist der wesentliche Informationsgehalt der Modellbeschreibung auf der Ebene der Verarbeitung durch den Rechner. Hier liegt das Modell des Systems in einer verarbeitungsorientierten Form als eine Menge von rechnernahen Anweisungen vor. Bei den auf dieser Ebene aufsetzenden Simulations-, Analyse- und Synthesewerkzeugen für mechatronische Systeme reduziert sich die Aufbereitung des Modells für die Berechnung auf ein Minimum. Damit vereinfacht sich auch die Entwicklung und Wartung dieser Werkzeuge. Zusammenfassend ergibt sich die folgende Definition der verarbeitungsorientierten Modellrepräsentation:

Definition:
Die *verarbeitungsorientierte Modellrepräsentation* ist eine symbolische Modellrepräsentation eines mechatronischen Systems, die durchgängig durch den Entwicklungszyklus als zentrale Repräsentation des zu entwickelnden Systems verwendet wird. Sie ermöglicht es, das gesamte betrachtete System als ein homogenes Gesamtmodell zu behandeln.
Bezüglich des Abstraktionsgrades ist die verarbeitungsorientierte Modellrepräsentation unterhalb der mathematischen Modellierung eingeordnet. Sie enthält die Beschreibung des gesamten Systemverhaltens in Form eines expliziten Berechnungsgraphen mit rechnernahen Anweisungen, so dass Simulations-, Analyse- und Synthesewerkzeuge direkt den Berechnungsgraphen für ihre Auswertungen nutzen können.
Die verarbeitungsorientierte Modellrepräsentation einer Komponente eines mechatronischen Systems ist ein bezüglich des Systemverhaltens semantikerhaltendes, nicht diskretisiertes Abbild des vom Anwender erstellten physikalisch-topologischen bzw. mathematischen Modells der Komponente. Zur Interaktion der auf der verarbeitungsorientierten Modellrepräsentation basierenden Simulations-, Analyse- und Synthesewerkzeuge mit dem Anwender stellt die verarbeitungsorientierte Modellrepräsentation einen Mechanismus bereit, der einen transparenten Zugriff auf die ursprüngliche Modellbeschreibung ermöglicht.
Die verarbeitungsorientierte Modellrepräsentation hat die Fähigkeit, in jedem Prozessschritt um auswertungsrelevante Modellinformationen angereichert zu werden, die von den weiteren Prozessschritten genutzt werden können.

Die verarbeitungsorientierte Modellrepräsentation ist das Kernthema dieser Arbeit. Nach dem derzeitigen Stand der Technik ist sie bei den verfügbaren Modellierungs- und Simulationswerkzeugen höchstens in Form von nicht zugänglichen, werkzeuginternen und auf ein Fachgebiet spezialisierten Datenstrukturen existent. Ausführliche Erläuterungen zu der im Rahmen die-

ser Arbeit entwickelten verarbeitungsorientierten Modellrepräsentation der Mechatronic Processing Objects und der zugrundeliegenden Modellierungssprache DSC finden sich in den Kapiteln 3 und 4.

2.3 Modellintegration

Mechatronische Systeme bestehen typischerweise aus einer Vielzahl von Komponenten mit unterschiedlichen Wirkprinzipien. Die einzelnen Komponenten sind häufig in fachspezifischen Modellierungssprachen beschrieben, die dann in das Gesamtsystem integriert werden müssen. Die aktuell verfügbaren Ansätze zur Integration von Teilmodellen lassen sich in drei Klassen einteilen: Einbindung von Teilmodellen als Programmcode, Simulatorkopplung und transparente Gesamtmodelle.

2.3.1 Einbindung von Teilmodellen als Programmcode

Zur Lösung des Problems der Integration von Teilmodellen in einer anderen Modellierungssprache bieten viele Simulationsumgebungen eine Programmierschnittstelle in C oder FORTRAN an. Über diese Programmierschnittstelle können externe Teilmodelle aus einer anderen Simulationsumgebung als Programmcode eingebunden werden (s. Abbildung 2.6). Beispiele für eine

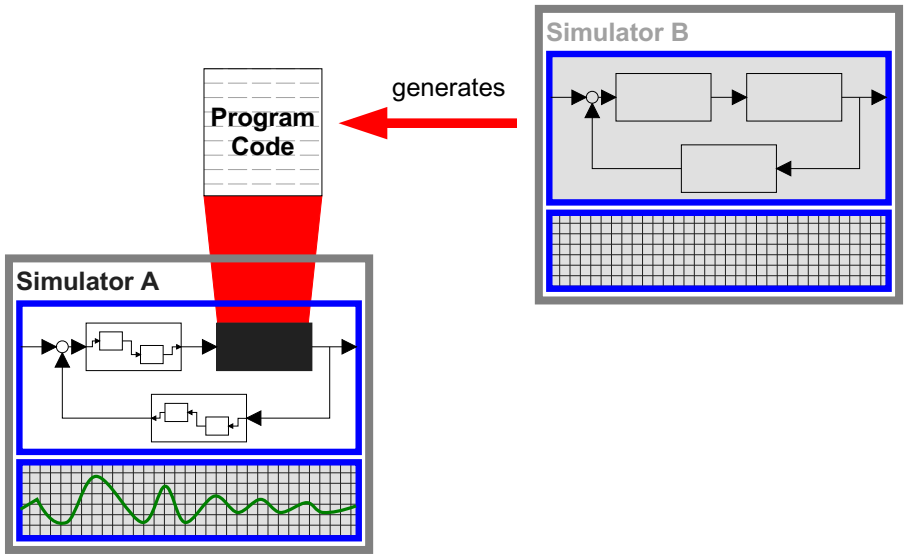


Abbildung 2.6: Einbindung eines Teilmodells als Programmcode

solche Programmierschnittstelle sind das SimStruct von Simulink [MAT92] und der UCB (User Code Block) von SystemBuild [Int94]. Einen allgemeineren Ansatz für den Austausch von Modellkomponenten als Programmcode

bietet die Modellschnittstelle DSblock [OE95]. Bei der Einbindung einer externen Modellkomponente über eine dieser Programmierschnittstellen muss der Anwender jedoch die folgenden, prinzipbedingten Nachteile in Kauf nehmen:

- Ein als Programmcode vorliegendes Teilmodell hat den Charakter einer *Black Box*. Es erscheint in der Simulationsumgebung des Gesamtsystems als ein verschlossener Block mit einem Satz von Schnittstellengrößen wie z. B. Parametern, Ein- und Ausgangsgrößen. Die innere Struktur des Teilmodells ist in der Simulationsumgebung des Gesamtsystems nicht zugänglich.

- Die Generierung des Programmcodes für das Teilmodell erfolgt ohne Berücksichtigung des Gesamtsystemkontexts. Dies kann zu unnötigen *algebraischen Schleifen* bei der Auswertung der Systemgleichungen führen. Im Allgemeinen kann eine gültige Auswertereihenfolge der Systemgleichungen erst ermittelt werden, wenn alle Modellgleichungen des gesamten Systems bekannt sind.

- Wird eine Modellkomponente als Programmcode eingebunden, so kann die ursprüngliche, fachspezifische Werkzeugumgebung mit ihren für das Fachgebiet spezifischen Konfigurationsmöglichkeiten und ihren spezialisierten Gleichungslösern bei der Simulation des Gesamtsystems nicht eingesetzt werden. Das Resultat kann sowohl ein höherer Verbrauch an Rechenleistung als auch ein Genauigkeitsverlust bei den Simulationsergebnissen sein.

2.3.2 Simulatorkopplung

Ein besserer Ansatz als die Einbindung von Teilmodellen als Programmcode ist die Kopplung von Simulatoren für die Simulation von heterogen modellierten mechatronischen Systemen. Bei der Simulatorkopplung werden die einzelnen Teilmodelle jeweils in ihren eigenen, fachspezifischen Simulationsumgebungen gerechnet (s. Abbildung 2.7). Der Austausch der numerischen Daten zwischen den einzelnen Simulatoren erfolgt über gemeinsame Schnittstellen. Dieser Ansatz erlaubt dem Ingenieur einen vollständigen Zugriff auf alle Komponenten des gesamten Systems bis in das kleinste Detail. Aber auch die Kopplung von Simulatoren ist mit einigen Nachteilen behaftet:

- Viele Simulationsumgebungen lassen sich nur als Simulations-Master einsetzen. Sie sind in der Lage, einen anderen Simulator schrittweise anzusteuern (z. B. über eine C-Programmierschnittstelle), stellen aber selbst keine offene Schnittstelle zur Verfügung, mit der sie selbst angesteuert werden können.

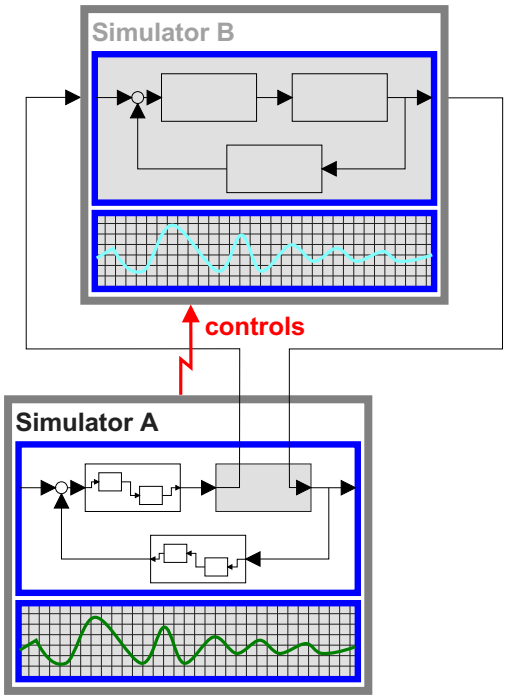


Abbildung 2.7: Einbindung eines Teilmodells über Simulatorkopplung

- Bei der Durchführung der Simulation des Gesamtsystems hat der Ingenieur keinen einheitlichen Zugang zum Modell des Systems. Er muss mit einer Menge von Softwarewerkzeugen, Modellbeschreibungen, Konfigurationen und partiellen Simulationsergebnissen umgehen. Dabei muss viel Zeit investiert werden, um das gesamte System konsistent zu halten. Dies trifft im Besonderen zu, wenn der Ingenieur Änderungen an der Modellstruktur vornehmen muss, die über die Grenzen eines Simulationswerkzeugs hinausreichen.

- Der unvermeidbare, intensive Austausch von Simulationsdaten zwischen den dezentralisierten, Modellkomponenten-spezifischen Simulationsprozessen führt zu einer deutlich langsameren Simulationsgeschwindigkeit. Da bei der Simulatorkopplung mehrere schwergewichtige Prozesse gleichzeitig aktiv sind, werden beim Simulationsrechner deutlich höhere Anforderungen an das Betriebssystem und die Größe des Hauptspeichers gestellt als im Fall der Einbindung von Teilmodellen als Programmcode.

2.3.3 Transparente Gesamtmodelle

Mit der Einführung einer Zwischensprache in Form einer verarbeitungsorientierten Modellrepräsentation lässt sich das Problem der Integration von

Modellen lösen, ohne dass dabei die mit den beiden zuvor beschriebenen Lösungsansätzen verbundenen Nachteile in Kauf genommen werden müssen.

Unter Einsatz einer verarbeitungsorientierten Modellrepräsentation liegt das gesamte System als homogenes Modell vor. Dadurch ist ein einheitlicher, zentraler Zugang auf alle Komponenten des Systems möglich, auch wenn sie ursprünglich aus einer sehr heterogenen Vielfalt von fachspezifischen Modellierungsumgebungen stammen. Die ursprüngliche Modellstruktur bleibt dabei erhalten und ist jederzeit transparent. Durch die Bereitstellung entsprechender, kleiner Frontends für die einzelnen Modellierungswerkzeuge kann auf jede Modellkomponente bis in das kleinste Detail zugegriffen werden. Das reduziert die Komplexität bei der Handhabung des gesamten Systems erheblich, ausserdem muss der Ingenieur nur noch mit einer einzigen Simulationskonfiguration umgehen.

Die Verwendung einer einheitlichen, verarbeitungsorientierten Modellrepräsentation für das gesamte System ermöglicht den Einsatz eines zentralen Simulationsalgorithmus mit optimierter Auswertung des gesamten Satzes aller Modellgleichungen des komplexen, heterogen modellierten mechatronischen Systems. Ein Austausch von numerischen Simulationsdaten zwischen den unterschiedlichen Modellkomponenten über die Interprozess-Kommunikationsdienste des Betriebssystems ist nicht mehr erforderlich.

Über die Vorteile des zentralen, einheitlichen Modellzugangs und der schnellen Simulationsgeschwindigkeit hinaus bringt der Einsatz einer verarbeitungsorientierten Modellrepräsentation eine breitere Verfügbarkeit von speziellen Gleichungslösern und Integrationsverfahren mit sich. Die Gleichungslöser und Integrationsverfahren sind nicht mehr an ein spezielles, fachspezifisches Modellierungs- und Simulationswerkzeug und die damit modellierten Teilmodelle gebunden, sondern lassen sich, soweit numerisch sinnvoll, auch für andere Teilmodelle einsetzen.

In der vorliegenden Arbeit wird der hier beschriebene Ansatz der Modellintegration über transparente Gesamtmodelle unter Einsatz einer verarbeitungsorientierten Modellrepräsentation verfolgt. Eine genauere Darstellung dieses Lösungsansatzes ist in Kapitel 3.2 zu finden.

Prinzipiell könnte ein transparentes Gesamtmodell anstatt auf der Ebene der verarbeitungsorientierten Modellrepräsentation auch auf der Ebene der mathematischen Modellbeschreibung positioniert sein. Alle Teilmodelle müssten dann in eine gemeinsame, mathematische Modellierungssprache wie z. B. VHDL-AMS transformiert werden. In der Vergangenheit hat es einige Versuche gegeben, semantikerhaltende Modelltransformationen u. a. nach VHDL-AMS und Simulink zu entwickeln. Aufgrund der hohen Komplexität und der starken Unterschiede der einzelnen Semantiken funktionieren diese Transformationen aber nur für einen Bruchteil der Modelle mit stark eingeschränktem Modellierungsumfang zufriedenstellend. Für einen Großteil der Modelle lässt sich eine automatische Transformation unter vollständiger Bewahrung der ursprünglichen Semantik nicht umsetzen.

2.4 Vergleich mechatronischer Entwurfsumgebungen

Für die Entwicklung mechatronischer Systeme existiert bereits eine Reihe von Entwurfswerkzeugen, die in verschiedenartigster Ausprägung Teile des Entwurfsprozesses unterstützen. Dieses Unterkapitel gibt einen Überblick über die für den Entwurf mechatronischer Systeme relevanten Entwurfswerkzeuge und bewertet sie hinsichtlich ihrer Leistungsfähigkeit. Diese Bewertung wird auch für den in dieser Arbeit beschriebenen Ansatz der Mechatronic Processing Objects vorgenommen, der die im Vergleich höchste Leistungsfähigkeit hat.

2.4.1 Vergleichskriterien

Zwecks Bewertung der Leistungsfähigkeit und Vergleich der aktuellen Entwurfswerkzeuge für mechatronische Systeme werden hier die wichtigsten Kriterien aufgestellt, die für die Unterstützung des mechatronischen Entwurfsprozesses erforderlich sind. Ziel der Aufstellung dieser Vergleichskriterien ist ein übersichtsartiger, tabellarischer Vergleich der relevanten Entwurfswerkzeugklassen für mechatronische Systeme. Für die detaillierte Ableitung dieser Vergleichskriterien aus den einzelnen Prozessschritten des mechatronischen Entwurfs sei an dieser Stelle auf das Kapitel 3.1 verwiesen.

Echtzeitfähigkeit

Sowohl das Modell des Reglers als auch das Modell der Strecke können unter Echtzeitbedingungen ausgeführt werden. Die Durchführung von Simulationen mit der Einbindung von realen Komponenten ist möglich (Hardware-in-the-Loop).

Modell-Debugging

Zur Unterstützung der Fehlersuche in der Modellbildungsphase stellt die Entwurfsumgebung Möglichkeiten zum schrittweisen Verfolgen der Modellauswertung und zum Protokollieren von ausgewählten Modellgrößen bereit. Zur Untersuchung von Grenzfällen ist eine Neubesetzung einzelner Größen möglich. Die schrittweise Verfolgung der Modellauswertung erfolgt anhand der ursprünglichen Modellspezifikation. Treten numerische Fehler wie z. B. eine Division durch Null oder eine Überschreitung eines Wertebereichs auf, so wird die Auftrittsstelle des Fehlers direkt in der ursprünglichen Modellspezifikation angezeigt.

Modellierung kontinuierlicher Systeme

Die Entwurfsumgebung unterstützt die Modellierung von kontinuierlichen Systemen, beschrieben durch ein System von Differentialgleichungen. Bewährt hat sich hier die explizite Form der Zustandsdarstellung 1. Ordnung. Eine weitere Form sind die differential-algebraischen Gleichungssysteme (DAE). Separiert von den modellierten Systemgleichungen stellt die Entwurfsumgebung entsprechende Gleichungslöser/Integrationsverfahren bereit. Ermöglicht eine Entwurfsumgebung ausschließlich die Abbildung von kontinuierlichen Systemen in einer diskretisierten Form, so gilt dieses Bewertungskriterium als nicht erfüllt.

Modellierung diskreter Systeme

Die Entwurfsumgebung unterstützt die Modellierung von diskreten, ereignisgesteuerten Systemen.

Modellierung dynamischer Systemstrukturen

Die Entwurfsumgebung unterstützt die Modellierung von dynamischen Systemstrukturen. Es können Systeme abgebildet werden, die zur Laufzeit situationsabhängig in neue Organisationsstrukturen eintreten oder ihre bisherige Teilstruktur verlassen können. Zur Laufzeit können neue Instanzen eines Teilsystems erzeugt und in das Gesamtsystem eingebettet als auch wieder entfernt werden.

Domänenübergreifende Simulation/Analyse durch Codeintegration

Zur domänenübergreifenden Simulation und Analyse des Gesamtsystems bietet die Entwurfsumgebung eine Programmierschnittstelle an, um Teilmodelle unterschiedlicher Domänen als Programmcode einzubinden. Die Modellierung geschieht bei diesem Ansatz zunächst domänenspezifisch pro Teilmodell in unterschiedlichen Werkzeugen. Bei Betrachtung auf Gesamtsystemebene stehen die Teilmodelle der vom Hauptwerkzeug nicht direkt unterstützten Domänen dann als ausführbarer Programmcode zur Verfügung.

Domänenübergreifende Simulation/Analyse durch Simulatorkopplung

Zur domänenübergreifenden Simulation des Gesamtsystems lässt sich die Entwurfsumgebung mit Simulationsumgebungen anderer Domänen koppeln. Die Entwurfsumgebung ist in der Lage, einen anderen Simulator schrittweise anzusteuern. Soll die Hauptsteuerung von dem anderen Simulator aus geschehen, so muss die Entwurfsumgebung eine offene Schnittstelle zur eigenen Ansteuerung bereitstellen.

Domänenübergreifendes, homogenes Gesamtmodell

Das gesamte System liegt über alle Domänen hinweg für die Simulation und Analyse als homogenes Modell vor. Die Entwurfsumgebung bietet einen einheitlichen, zentralen Zugang auf alle Komponenten des Systems. Ist die homogene Repräsentation des Gesamtsystems durch Modelltransformationen entstanden, so muss die Semantik der ursprünglichen Modellierung vollständig abgebildet sein. Die ursprüngliche Modellstruktur bleibt dabei erhalten und ist jederzeit transparent.

Verteilte Simulation

Die Durchführung von verteilten, nebenläufigen Simulationen ist möglich. Die Entwurfsumgebung bietet eine Möglichkeit, das Gesamtsystem zu partitionieren und die einzelnen Anteile auf unterschiedlichen Rechenkernen einer Multiprozessor-Architektur bzw. eines Prozessornetzwerks auszuführen. Für die Erfüllung dieses Bewertungskriteriums ist es ausreichend, wenn eine Partitionierung auf Basis der modellierten Subsystemhierarchie vorgenommen werden kann.

Deployment unabhängig von Modellstruktur

Die Entwurfsumgebung bietet die Möglichkeit, die Partitionierung des Gesamtsystems für eine verteilte, nebenläufige Verarbeitung unabhängig von der modellierten Subsystemhierarchie vornehmen zu können. Es ist nicht erforderlich, dass die sich bei der Umsetzung auf ein Prozessornetzwerk ergebenden informationstechnischen Strukturen schon zu Beginn der Modellbildung bei der Systemstrukturierung berücksichtigt werden müssen. Eine Umsetzung beliebiger Partitionierungen des Gesamtsystems ist möglich, ungeachtet der modellierten Subsystemgrenzen.

Unabhängige Erweiterbarkeit

Die Entwurfsumgebung ist offen für eine einfache Erweiterung um zusätzliche Frontends für die Einbindung weiterer, domänenspezifischer Modellbeschreibungsformen sowie die Anbindung weiterer Simulations-, Analyse- oder sonstiger Verfahren für die Modellverarbeitung. Dazu bietet die Entwurfsumgebung die folgenden Möglichkeiten:

- Es existiert ein offengelegtes API mit vollständigem Zugriff auf das Modell des gesamten Systems.
- Es existiert eine einheitliche, formal definierte Semantik für das gesamte System.

- Es existiert eine offengelegte Schnittstelle für die Erweiterung um Gleichungslöser und Integrationsverfahren. Eine zentrale Anwendung des Integrationsverfahrens für das gesamte System ist möglich.
- Eine offene C-Schnittstelle zur Hardware-Anbindung steht zur Verfügung.
- Es existiert ein offengelegter Zugriff auf den Berechnungsgraphen, der explizit die Berechnung des gesamten Systemverhaltens in Form von rein funktionalen Zusammenhängen angibt. Hierarchische Verkopplungen sind dabei bis auf Gleichungsebene komprimiert.
- Es existiert eine offengelegte Schnittstelle für eine externe Ansteuerung der Simulation.

Lineare Systemanalyse/Synthese

Die Entwurfsumgebung bietet die Möglichkeit, Verfahren der linearen Systemanalyse und Reglersynthese auf das modellierte System anwenden zu können. Das Systemverhalten des linearisierten Systems kann in Form von Matrizen beschrieben werden.

Codegenerierung

Die Entwurfsumgebung stellt eine automatische Codegenerierung bereit, die für das modellierte System kompilierbaren Programmcode, z. B. in der Programmiersprache C, erzeugt, der das modellierte Systemverhalten umsetzt. Die Codegenerierung ist sowohl für modellierte Strecken- als auch für Regleranteile möglich. Üblicherweise ist die automatische Codegenerierung für die Bedürfnisse der Durchführung von Simulationen während des Entwurfsprozesses ausgelegt. Einige Entwurfsumgebungen können darüber hinaus optimierten, serienreifen C-Programmcode liefern, der direkt als Implementierung des Regel- bzw. Steuerungsalgorithmus in das Produkt übernommen werden kann.

Interpretative Auswertung

Die Entwurfsumgebung bietet einen interpretativen Ansatz zur Auswertung des Systemverhaltens. Das modellierte System liegt als interpretativ auswertbare Datenstruktur vor. Die Auswirkungen einer Modellmodifikation auf das Zeitverhalten können vom Ingenieur sofort untersucht werden.

Verifikation/Validierung

Die Entwurfsumgebung bietet die Möglichkeit, geforderte Eigenschaften des modellierten Systems nachzuweisen. Dies kann z. B. über die Möglichkeit der

Anbindung von Verifikations- oder Model-Checking-Werkzeugen umgesetzt sein.

2.4.2 Mechatronische Entwurfsumgebungen

Mit den zuvor aufgestellten Vergleichskriterien kann der aktuelle Stand der Technik mechatronischer Entwurfsumgebungen hinsichtlich ihrer Leistungsfähigkeit bewertet werden. Neben einer Grobcharakterisierung der zu Klassen zusammengefassten Entwurfswerkzeuge wird in Tabelle 2.1 eine konkrete Bewertung anhand der einzelnen Vergleichskriterien vorgenommen. Als Abschluss der Vergleichstabelle ist der in dieser Arbeit vorgestellte Ansatz der Mechatronic Processing Objects aufgeführt.

Modelica

Modelica[1] [Mod10] ist eine frei verfügbare, objektorientierte Modellbeschreibungssprache, basierend auf differential-algebraischen Gleichungssystemen (DAE). Für Modelica stehen eine Reihe von kommerziellen als auch frei verfügbaren Simulationsumgebungen zur Verfügung, darunter CATIA Systems, Dymola, LMS AMESim, JModelica.org, MapleSim, MathModelica, OpenModelica, SCICOS, SimulationX und Vertex. Aufgrund der DAE-basierten Modellierung und umfangreichen Modellbibliotheken ist es mit Modelica möglich, in einem Gesamtmodell alle für die Mechatronik relevanten Domänen homogen abzubilden.

Hinsichtlich Echtzeitfähigkeit ist Modelica nur sehr bedingt geeignet. Die iterativen DAE-Gleichungslöser sind nur in den seltensten Fällen für eine Simulation unter Echtzeitbedingungen anwendbar und stellen dann noch sehr hohe Anforderungen an die Rechnerhardware. Für die Echtzeitsimulation muss das Modell in ein System aus gewöhnlichen Differentialgleichungen (ODE) transformiert werden. Eine automatische symbolische Transformation der Gleichungen ist nur in einfachen Fällen möglich [BDL09]. In vielen Fällen muss der Ingenieur eingreifen und manuell eine Modellreduktion durchführen. Werkzeuge wie SimulationX versuchen hier, dem Ingenieur unterstützende Hilfsmittel zur Verfügung zu stellen [BB09].

Die bezüglich der Echtzeitfähigkeit genannten Einschränkungen gelten auch für die verteilte Simulation. Liegt das Gesamtsystem in seiner DAE-Form vor, so kann dieses in sich geschlossene Gleichungssystem nicht direkt auf mehrere Rechenkerne verteilt werden. Hier ist wieder eine Transformation in die ODE-Form erforderlich. Seit Modelica 3.1 gibt es aber die Möglichkeit, auf Subsystemebene die spätere Abbildung auf parallele Ausführungseinheiten im Modell zu annotieren.

Die Modellierungssprache Modelica selbst ist frei verfügbar und bietet mit dem Functional Mockup Interface (FMI) Schnittstellen für Codeinte-

[1] http://www.modelica.org

gration und Simulatorkopplung/Co-Simulation. Aber eine unabhängige Erweiterbarkeit ist dennoch nicht gegeben. In jedem Werkzeug muss die Aufbereitung der Gleichungssysteme neu implementiert werden. Gleichungslöser sind werkzeug- und sogar versionsabhängig, die für die erfolgreiche Lösung der DAE erforderlichen Sätze von Initialwerten lassen sich nicht übertragen. Auch gibt es keine frei zugängliche Aufbereitung der Modelle für eine modulare Codegenerierung, auf der die einzelnen Codegeneratoren aufsetzen können.

Für Dymola [Das10] gibt es eine experimentelle Erweiterung [OME+09], die alle booleschen Gleichungen aus einem Modell extrahieren kann, um dann darauf den externen Model Checker NuSMV[2] [CCJ+10] anwenden zu können.

EasyLab/EasyKit

EasyLab/EasyKit [BGBK08, BGH+10] ist eine Entwicklungsumgebung und Methodik zur Hardware- und Firmwareentwicklung von Mikrocontroller-basierten, mechatronischen Steuerungskomponenten. Basierend auf Blockdiagrammen kann nach dem Baukastenprinzip aus getesteten Bausteinen eine Steuerung zusammengesetzt, am Rechner simuliert und dann auf der Zielhardware ausgeführt werden. Der Fokus von EasyLab/EasyKit ist ausschließlich auf die Entwicklung der eigentlichen Steuerungskomponenten gerichtet, es lassen sich nur diskrete Systeme modellieren. Eine Modellierung kontinuierlicher Systemanteile sowie der Regelstrecke wird nicht angeboten, diese Anteile können mittels entsprechender Schnittstellen an die Simulation angebunden werden. Eine Stärke von EasyLab ist der vorlagenbasierte Codegenerator, über benutzerdefinierte Codevorlagen (Templates) ist eine Anpassung der Codegenerierung an neue Mikrocontroller-Plattformen möglich.

MATLAB/Simulink

MATLAB/Simulink [Mat10d] ist eine weit verbreitete Entwicklungsumgebung mit einem großen Angebot von Toolboxen für die unterschiedlichsten Anwendungsbereiche. Simulink erlaubt die Modellierung kontinuierlicher Systeme in Form von Blockdiagrammen. Für die lineare Systemanalyse und Synthese stehen umfangreiche Toolboxen zur Verfügung. Die Abbildung ereignisdiskreter Systemanteile in Form von Statecharts ist über die Erweiterung Stateflow [Mat10e] möglich. MATLAB/Simulink bietet aber keine Unterstützung für die Abbildung dynamischer Systemstrukturen.

Mit dem Real-Time Workshop [Mat10b] wird eine Codegenerierung für Rapid Control Prototyping und Hardware-in-the-Loop-Szenarien unterstützt. Mit den Werkzeugen TargetLink [dSP10b] und Real-Time Workshop Embedded Coder [Mat10a] besteht sogar die Möglichkeit, direkt aus dem Mo-

[2] http://nusmv.fbk.eu

dell hocheffizienten, serienreifen C-Code für die Reglerimplementierung zu generieren.

Für die Durchführung verteilter Simulationen ist die Blockbibliothek RTI-MP [dSP10a] verfügbar, die eine Zuordnung von Subsystemen auf unterschiedliche Prozessorknoten ermöglicht. Eine von der Modellstruktur unabhängige Verteilung wird aber nicht unterstützt, jeder Prozessorknoten kann genau einem Subsystem aus der Modellhierarchie zugeordnet werden.

Ein großer Schwachpunkt von MATLAB/Simulink ist das Fehlen einer formalen Definition der Semantik. Das Systemverhalten der einzelnen Blocktypen sowie das Verhalten von Subsystem-Kompositionen ist nur informal dokumentiert. Das konkrete Verhalten im Detail ergibt sich aus der Implementierung der einzelnen Blöcke und des Simulators. Eine formale, Blocktypen-übergreifende, gemeinsame Repräsentation des gesamten modellierten Systems steht in MATLAB/Simulink nicht zur Verfügung. Soll ein neues Verfahren zur Systemanalyse, Reglersynthese oder sonstigen Verarbeitung hinzugefügt werden, so muss dieses jeweils für jeden einzelnen Blocktyp eine spezifische Unterstützung umsetzen. Damit ist eine unabhängige Erweiterbarkeit von MATLAB/Simulink nicht gegeben.

Aufgrund des Fehlens einer formalen Definition der Semantik ist die Anwendung formaler Verifikationstechniken nur mit Einschränkungen möglich. Das Werkzeug Embedded Validator [BTC10] bietet die Möglichkeit, Model Checking auf dem für ein Simulink/Stateflow-Modell generierten C-Programmcode durchzuführen.

SCADE

SCADE [Est11] ist eine auf der synchronen Sprache Lustre [AM10] basierende Entwicklungsumgebung. Der Fokus von SCADE ist ausschließlich auf die Entwicklung der eigentlichen Steuerungskomponenten gerichtet, es lassen sich nur diskrete Systeme modellieren. Eine Modellierung kontinuierlicher Systemanteile sowie der Regelstrecke direkt in SCADE wird nicht angeboten, diese Anteile können mittels entsprechender Schnittstellen an die Simulation angebunden werden. Für Echtzeitanwendungen existiert eine Anbindung an LabVIEW [Nat11], eine Unterstützung für verteilte Simulation existiert nicht.

SCADE bietet ein umfangreiches API für Eclipse [Ecl11] an, das neben dem Lese- und Schreibzugriff auf Projekt-und Modelldateien auch den Austausch sowohl semantischer als auch graphischer Information erlaubt. Allerdings ist der Zugriff auf die Modellsemantik ausschließlich auf dem Niveau des mathematischen Modells möglich. Das gesamte Systemverhalten in Form eines aufbereiteten, flachen, rein funktional zusammenhängenden Berechnungsgraphen ist in SCADE nicht verfügbar. Soll ein neues Verfahren zur Codegenerierung oder sonstigen Verarbeitung hinzugefügt werden, so muss die Erstellung des Berechnungsgraphen jeweils neu implementiert

werden.

Zur formalen Verifikation der erstellten Steuerungskomponenten steht der SCADE Suite Design Verifier zur Verfügung, der Model Checking auf Basis des Prover Plug-Ins [Pro11] bietet.

ASCET

Die ASCET Produktfamilie [LE09, ETA11] wurde speziell für die modellbasierte Entwicklung von Automotive Software entwickelt. Schwerpunkt ist die diskrete Modellierung von Software-Modellkomponenten. Diese können anhand von Blockdiagrammen, sowohl graphisch als auch textuell mit der Sprache ESDL (Embedded Software Description Language), und Zustandsautomaten spezifiziert werden. Zur Modellierung kontinuierlicher Systeme stehen zusätzlich sogenannte CT-Blöcke (Continuous time) zur Verfügung.

ASCET bietet einen RTOS-Konfigurator zur Spezifikation des zeitlichen Ausführungsverhaltens. Mit LABCAR-RTPC [CS09] ist eine verteilte Simulation unter Echtzeitbedingungen auf Multicore-Hardware möglich. Eine von der Modellstruktur unabhängige Verteilung ist nur eingeschränkt möglich: Der Code für das Simulationsmodell muss auf oberster Ebene vom Anwender in einzelne LABCAR-Module partitioniert werden, dies muss bei der hierarchischen Modellierung des Systems bereits berücksichtigt werden. Die einzelnen LABCAR-Module können dann jeweils frei den einzelnen Prozessorkernen zugeordnet werden, wobei ein Prozessorkern auch für die Ausführung von mehreren Modulen zuständig sein kann. Kleinste Einheit der Verteilung ist das bereits als C-Code vorliegende LABCAR-Modul.

Zur Einbindung von anderen domänenspezifischen Modellierungen bietet ASCET einen Import von Simulink-Modellen an, hier ist aber der abgedeckte Simulink-Sprachumfang stark eingeschränkt und in vielen Fällen auch keine vollständige Erhaltung der Simulink-Semantik möglich. Ausserdem ist nach dem Import kein transparenter Zugriff auf das Simulink-Quellmodell mehr gegeben.

ASCET stellt ein API bereit, das die Steuerung von Modellsimulationen und Codegenerierung ermöglicht. Das modellierte System ist in Form von dokumentierten XML-Dateien für externe Anwendungen verfügbar, der Informationsgehalt ist auf demselben Niveau wie das vom Anwender erstellte Blockdiagramm. Eine unabhängige Erweiterbarkeit von ASCET ist damit aber nicht gegeben. So gibt es keinen Zugriff auf den Berechnungsgraphen für weitere Anwendungen. Externe Anwendungen, die über die Steuerung von Simulations- oder Codegenerierungsprozessen hinausgehen, können ausschließlich auf dem mathematischen Modell oder dem fertig konfektionierten C-Code aufsetzen. Die Anwendung von Model Checking für ASCET-Modelle geht ebenfalls über den generierten C-Code [DSS+03].

SysML

Die Systems Modeling Language (OMG SysML) [Obj10] ist eine auf UML 2 basierende, standardisierte, graphische Sprache für die Modellierung von komplexen Systemen. Sie basiert auf einer Untermenge von UML 2 sowie speziellen Erweiterungen gegenüber UML für die Modellierung von Systemanforderungen, Systemverhalten, Systemstruktur und Parametrierung.

Ziel von SysML ist die Darstellung des gesamten Systems in einer einheitlichen Darstellung, um Systemanforderungen modellieren und für die einzelnen Komponenten zur Verfügung stellen zu können, Schnittstellen zu evaluieren und Systeminformationen zwischen den einzelnen Beteiligten unmissverständlich kommunizieren zu können. Um Freiraum für die Integration anderer Modellierungswerkzeuge zu schaffen, unterstützt SysML wie auch UML standardmäßig nur eine qualitative Verhaltensspezifikation, das rein in SysML modellierte Systemverhalten ist also weder ausführbar noch kann dafür direkt Code für z. B. eine Simulation oder Reglerimplementierung generiert werden. Standardmäßig liegt der Fokus von SysML auf der Modellierung diskreter Systeme, es ist aber die Entwicklung eines SysML4Modelica-Profils in Arbeit, mit dem in Modelica modellierte, kontinuierliche Modelle in eine entsprechende SysML-Repräsentation transformiert werden können [PBB+10].

Mechatronic UML

Mechatronic UML [BGH+07] ist eine im Softwarewerkzeug Fujaba Real-Time Tool Suite umgesetzte Modellierung des diskreten Echtzeitverhaltens mit Real-Time Statecharts. Fokus ist die Modellierung der nachrichtenbasierten, diskreten Echtzeitkoordination zwischen mechatronischen Systemen. Eine Modellierung kontinuierlicher Systemanteile sowie der Regelstrecke wird von der Fujaba Real-Time Tool Suite nicht angeboten. Mit dem Werkzeug CAMeL-View gibt es aber die Möglichkeit, Mechatronic UML-Modelle als C-Code einzubinden [THB+10].

UML SPT

Das *UML Profile for Schedulability, Performance and Time (UML SPT)* [Obj05] ist eine Erweiterung von UML für die Modellierung von Echtzeitsystemen. Es ermöglicht die Modellierung von Zeitverhalten, Ressourcennutzung, Nebenläufigkeit, Scheduling und Systemleistung in Form von Durchsatz und Antwortzeiten. Hauptziel von UML SPT ist es, Modelle bereitzustellen, mit denen quantitative Vorhersagen bezüglich dieser Echtzeiteigenschaften aufgestellt, auf standardisiertem Weg zwischen den beteiligten Entwicklern während des Designs kommuniziert und auf Werkzeugebene ausgetauscht werden können. Standardmäßig lassen sich mit UML SPT nur

diskrete Systeme modellieren. Die in UML SPT direkt mögliche Modellierung des Systemverhaltens ist nicht ausreichend, um daraus ein ausführbares Modell zu erhalten. Zur konkreten Simulation des Modells müssen Teile des Systemverhaltens in den einzelnen Methodenaufrufen implementiert werden, z. B. in Form von C, Java, SystemC, oder VHDL.

MARTE

Das *UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)* [Obj09] ist eine Weiterentwicklung von UML SPT. Bezüglich dem Einsatz als mechatronische Entwurfsumgebung gelten die gleichen Aussagen wie für UML SPT.

UPPAAL

UPPAAL [Dep11] ist eine integrierte Entwicklungsumgebung mit dem Schwerpunkt auf der Verifikation von Echtzeitsystemen. Basierend auf Networked Timed Automata ermöglicht sie ausschließlich die Modellierung diskreter Systeme. Zur domänenübergreifenden Simulation/Analyse bietet UPPAAL nur die Integration von Java-Code an.

SHIFT

Die Modellbeschreibungssprache SHIFT [AG97] wurde für eine auf hybriden Systemen basierende Simulation von Verkehrsautomatisierung entwickelt. Neben der Abbildung von diskreten und kontinuierlichen Systemanteilen ermöglicht SHIFT auch die Abbildung dynamischer Systemstrukturen. Zur Auswertung des Modellverhaltens bietet SHIFT ausschließlich die Simulation auf Basis von generiertem C-Programmcode an. Eine Simulation unter Echtzeitbedingungen sowie eine verteilte Simulation sind nicht verfügbar.

CAMeL-View

In dem kommerziell verfügbaren Werkzeug CAMeL-View [THB+10] ist ein Teil der in dieser Arbeit vorgestellten Ansätze bereits umgesetzt. CAMeL-View setzt bereits die verarbeitungsorientierte Modellbeschreibungsform DSC in einer frühen Form ein. Von CAMeL-View noch nicht unterstützt werden interpretative Auswertung, Modell-Debugging, die Modellierung dynamischer Systemstrukturen sowie eine von der Subsystemhierarchie unabhängige Partitionierung des Gesamtsystems für eine verteilte, nebenläufige Verarbeitung.

MPO

Als Abschluss der Vergleichstabelle ist der in dieser Arbeit vorgestellte Ansatz der Mechatronic Processing Objects (MPO) aufgeführt. Mit Ausnahme

der Verifikation/Validierung werden alle aufgeführten Kriterien vollständig erfüllt. Eine Anbindung eines Verifikations- oder Model-Checking-Werkzeugs hat bisher noch nicht stattgefunden, kann aber auf Basis des frei zugänglichen Berechnungsgraphen, den jedes MPO bereitstellt, problemlos durchgeführt werden. Darüber hinaus ist die Anwendung eines Model Checkers auf dem für ein MPO generierten C-Code bereits möglich.

	Echtzeitfähigkeit	Modell-Debugging	Modellierung kontinuierlicher Systeme	Modellierung diskreter Systeme	Modellierung dynamischer Systemstrukturen	Domänenübergreifende Simulation/Analyse durch Codeintegration	Domänenübergreifende Simulation/Analyse durch Simulatorkopplung	Domänenübergreifendes, homogenes Gesamtmodell	Verteilte Simulation	Deployment unabhängig von Modellstruktur	Unabhängige Erweiterbarkeit	Lineare Systemanalyse/Synthese	Codegenerierung	Interpretative Auswertung	Verifikation/Validierung
Modelica	○	+	+	+	−	+	+	+	○	−	−	+	+	+	○
EasyLab/EasyKit	+	−	−	+	−	+	+	−	−	−	−	−	+	−	−
MATLAB/Simulink	+	+	+	+	−	+	+	−	+	−	−	+	+	+	○
SCADE	+	+	−	+	−	+	+	−	−	−	○	−	+	+	+
ASCET	+	+	+	+	−	+	+	−	+	○	−	−	+	+	○
SysML	−	−	−	+	−	−	−	+	−	−	+	−	−	−	−
Mechatronic UML	○	−	−	+	−	+	+	−	−	−	−	−	+	−	+
UML SPT	+	−	−	+	−	○	−	−	−	+	+	−	○	−	+
MARTE	+	−	−	+	−	○	−	−	−	+	+	−	○	−	+
UPPAAL	+	+	−	+	−	○	+	−	−	−	−	−	−	+	+
SHIFT	−	−	+	+	+	+	+	−	−	−	−	−	+	−	−
CAMeL-View	+	−	+	+	−	+	+	+	+	−	+	+	+	−	○
MPO	+	+	+	+	+	+	+	+	+	+	+	+	+	+	○

Legende: + erfüllt ○ erfüllt mit Einschränkungen − nicht erfüllt

Der in dieser Arbeit beschriebene Ansatz der Mechatronic Processing Objects (MPO) ist in der letzten Zeile aufgeführt und hat im Vergleich die höchste Leistungsfähigkeit.

Tabelle 2.1: Vergleich der Mechatronic Processing Objects (MPO) mit relevanten Entwurfswerkzeugklassen

2.5 Die mechatronische Entwurfsumgebung CAMeL

Den Ausgangspunkt dieser Arbeit bildet die am MLaP entwickelte mechatronische Entwurfsumgebung CAMeL (**C**omputer-**A**ided **M**echatronics **L**aboratory [Ric96, Ric98]), die den kompletten Entwicklungskreislauf eines mechatronischen Systems abdeckt [Jae91, JKL+91].

Anfang der 90er Jahre entstand als gemeinsame Schnittstelle aller in CAMeL realisierten Programme die blockorientierte, gleichungsbasierte Modellbeschreibungssprache DSL (**D**ynamic **S**ystem **L**anguage [Sch94]). DSL-Systeme können vollständig parametrisiert beschrieben werden; die Eingabe erfolgt standardmäßig textuell. Basierend auf DSL stehen eine Reihe von Werkzeugen für den mechatronischen Entwurf zur Verfügung (s. Abbildung 2.8): Mit Hilfe des graphikunterstützten Modellierungswerkzeugs

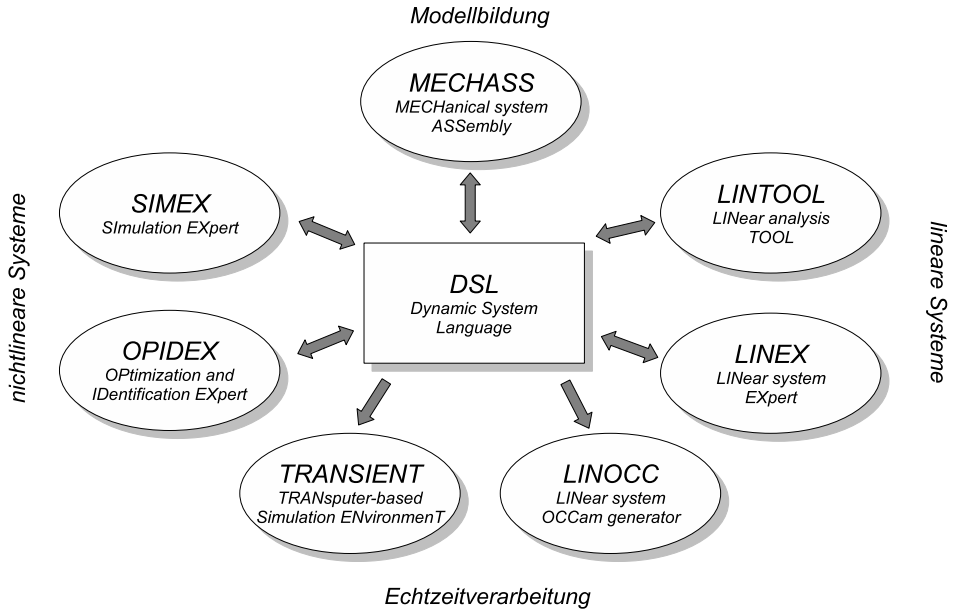


Abbildung 2.8: Stand der CAMeL-Entwicklung zu Beginn dieser Arbeit

für Mehrkörpersysteme MechAss (**Mech**anical System **Ass**embly [Hah91]) kann aus einem interaktiv zusammengebauten Mehrkörpersystem ein DSL-System auf der Basis von dynamischen Bindungen generiert werden. Zur linearen Analyse steht das Paket LINTOOL (**LIN**ear Analysis **TOOL** [Uni92]) zur Verfügung, das die Berechnung der Eigenwerte und des Frequenzgangs ermöglicht. Für die Analyse von nichtlinearen Systemen im Zeitbereich sowie die Durchführung von Parameterstudien kann das Programmpaket SIMEX (**SIM**ulation **EX**pert [Lef96]) eingesetzt werden. Die Aufgabe der Reglerauslegung und der Systemoptimierung wird von dem Pro-

grammpaket LINEX (**LIN**ear System **EX**pert) auf der Basis des Parameteroptimierungsverfahrens MOPO (**M**ulti-**O**bjective **P**arameter **O**ptimization [Kas92]) abgedeckt. Für nichtlineare Systeme kann dies unter Einsatz des Programmpakets OPIDEX (**O**ptimization and **P**arameter **I**dentification **E**xpert [Uni96]) geschehen. Für die verteilte Echtzeitsimulation steht die Transputer-basierte Simulationsumgebung TRANSIENT (**TRAN**sputer-based **SI**mulation **EN**vironmen**T** [Eng95]) zur Verfügung, darüber hinaus als Speziallösung der OCCAM-Codegenerierung für die Simulation linearer Systeme das Programmpaket LINOCC (**LIN**ear System **OCC**AM Generator).

Die einzelnen Werkzeuge der 1. Generation von CAMeL stellten jeweils ein eigenes ausführbares Programm dar, das in der Programmiersprache ADA implementiert wurde. In jedem der Werkzeuge musste die Aufbereitung der DSL-Modellbeschreibung für die Weiterverarbeitung im Rechner neu implementiert werden, außerdem konnten ausschließlich kontinuierliche Systeme mit statischer Topologie mit den in DSL zur Verfügung stehenden Mitteln beschrieben werden. Eine Anbindung weiterer Modellierungssprachen war nicht möglich.

Das führte dazu, am MLaP die Konzeption und Entwicklung einer Nachfolgegeneration der Entwicklungsumgebung CAMeL anzugehen. Grundprinzip der neuen Generation von CAMeL ist dabei die Aufteilung der Modellbeschreibung in die drei Ebenen des topologischen, des Verhaltens- und des Verarbeitungsmodells [HHR94], repräsentiert jeweils durch die Sprachen Objective-DSS, Objective-DSL und DSC. Die in dieser Arbeit beschriebenen Mechatronic Processing Objects mit der zugehörigen Beschreibungssprache DSC bilden auf Verarbeitungsebene die Grundlage der neuen Generation von CAMeL. Große Anteile der neuen CAMeL-Generation sind mittlerweile mit dem Werkzeug CAMeL-View (**C**omputer-**A**ided **M**echatronics **L**aboratory - **Vi**rtual **E**ngineering **W**orkbench) kommerziell verfügbar.

Kapitel 3

Gesamtkonzeption einer offenen, verarbeitungsorientierten Modellrepräsentation

Im vorliegenden Kapitel wird das Gesamtkonzept der Mechatronic Processing Objects dargestellt. Als verarbeitungsorientierte Modellrepräsentation bilden sie die Basis einer universellen, offenen Entwurfsumgebung für mechatronische Systeme. Dazu werden zunächst die Anforderungen an die Entwurfsumgebung vorgestellt, die sich in den einzelnen Phasen des in Kapitel 2.1 vorgestellten Entwicklungskreislaufs ergeben, ergänzt um softwaretechnische Anforderungen. Im zweiten Teil dieses Kapitels wird die zugrundeliegende Softwarearchitektur vorgestellt.

3.1 Anforderungsprofil

Bei der Entwicklung eines komplexen, mechatronischen Systems wird in vielen Phasen des Entwurfskreislaufs eine ausführbare Version des modellierten Systems benötigt. Im Folgenden werden die einzelnen Entwurfsphasen betrachtet und für jede Phase die daraus resultierenden Anforderungen an eine verarbeitungsorientierte Modellrepräsentation aufgestellt.

3.1.1 Modellbildung, Analyse und Identifikation

Je nach den zugrundeliegenden physikalischen Wirkprinzipien erfolgt die Modellierung der einzelnen Komponenten mechatronischer Systeme in jeweils unterschiedlichen, fachspezifischen Modellierungssprachen. Dabei verfeinert der Ingenieur schrittweise die Modellierung der einzelnen Komponenten. Jede überarbeitete Version des Modells muss mit Hilfe der nichtlinearen

Simulation und linearer Analyseverfahren überprüft werden. Hat der Ingenieur eine qualitativ korrekte Abbildung des Systemverhaltens gefunden, so muss er die Systemparameter identifizieren, um auch eine quantitativ korrekte Abbildung sicherstellen zu können. Bei näherer Betrachtung dieses Szenarios lassen sich daraus die nachfolgend aufgeführten Anforderungen an eine verarbeitungsorientierte Modellrepräsentation ableiten. Auf Anforderungen, die sich aus der Anwendung der linearen Systemtheorie ergeben, wird hier zunächst nicht eingegangen, sie sind in Kapitel 3.1.2 aufgeführt.

Anforderung MOD.1: Unterstützung modular-hierarchischer Strukturen

Mechatronische Systeme bestehen typischerweise aus einer Vielzahl von Komponenten, die hierarchisch zu höherwertigen Strukturen zusammengebaut werden. Diese Art der modular-hierarchischen Strukturierung muss auch von der verarbeitungsorientierten Modellrepräsentation unterstützt werden. Während der Modellbildung nimmt der Ingenieur häufig Veränderungen vor, die sich jeweils nur auf eine Teilkomponente beschränken. Im Sinne kurzer Entwicklungszeiten sollte dann nur eine Aktualisierung des entsprechenden Teilmodells in der verarbeitungsorientierten Modellrepräsentation erforderlich sein und nicht die Repräsentation für das gesamte System neu erstellt werden müssen.

Anforderung MOD.2: Homogene Integration unterschiedlicher Fachdisziplinen

Um Komponenten des mechatronischen Systems im Gesamtsystemkontext analysieren zu können, ist es erforderlich, dass die einzelnen, fachspezifisch formulierten Teilmodelle in ein Gesamtmodell integriert werden. Um Genauigkeitsverluste zu vermeiden, sollte die Modellrepräsentation auf dieser integrativen Ebene so beschaffen sein, dass sie das in den fachspezifischen Modellierungsformen formulierte mathematische Verhalten äquivalent abbilden kann. Damit erreicht man auf der integrativen Modellebene eine homogene Abbildung des gesamten Systemverhaltens. Auf diese einheitliche Modellierung des Gesamtsystems können dann die mathematischen Analysemethoden angewendet werden, ohne dass auf zusätzliche numerische Näherungen von Schnittstellengrößen an Subsystemgrenzen zurückgegriffen werden muss.

Anforderung MOD.3: Transparente Anknüpfung an ursprüngliche Modellspezifikation

Beim Umsetzungsprozess von der ursprünglichen, fachspezifischen Modellspezifikation zur äquivalenten verarbeitungsorientierten Modellrepräsentation finden in der Regel mehrere Modelltransformationen statt. Dies kann dazu führen, dass für den Ingenieur der Bezug zu der von ihm erstellten,

ursprünglichen Modellspezifikation nur noch schwer erkennbar ist. Es ist daher an eine verarbeitungsorientierte Modellrepräsentation die Anforderung zu stellen, dass sie Mechanismen anbietet, die für alle Modellobjekte eine transparente Zuordnung zur ursprünglichen Spezifikation ermöglichen. Aus der Sicht des Ingenieurs müssen sich die Entwurfswerkzeuge so verhalten, als würden sie direkt auf der ursprünglichen Modellspezifikation arbeiten.

Anforderung MOD.4: Schnelle Umsetzbarkeit von Modelländerungen

Hat der Ingenieur eine Veränderung am Modell vorgenommen, so will er das veränderte Systemverhalten unter möglichst geringem Zeiteinsatz mit Hilfe der nichtlinearen Simulation überprüfen. Für die nichtlineare Simulation muss das Modell dazu entweder als interpretativ auswertbare Datenstruktur oder als direkt ausführbarer Maschinencode vorliegen. Der interpretative Ansatz bietet hier den Vorteil, dass bei einer Veränderung am Modell, die in dieser Entwurfsphase viele Male vorgenommen wird, nicht jedes Mal die aufwendige Erzeugung von Maschinencode durchgeführt werden muss. Die Auswirkungen der Modellmodifikationen auf das Zeitverhalten können sofort untersucht werden. Bei der Wahl des zweiten Ansatzes, der Erzeugung von direkt ausführbarem Maschinencode, erhält man sehr kurze Rechenzeiten pro Simulationszeitschritt. Diese Variante ist insbesondere dann vorzuziehen, wenn das Modellverhalten über ein langes Simulationszeitintervall untersucht werden soll. Ist nur eine Untersuchung der ersten Zeitschritte gewünscht, so empfiehlt sich der interpretative Ansatz.

Zusammenfassend ist an eine verarbeitungsorientierte Beschreibungsform die Anforderung zu stellen, dass sie sowohl die interpretative Modellauswertung als auch die Erzeugung von Maschinencode zur Abbildung des Zeitverhaltens unterstützen sollte. Welche Variante im Einzelfall zum Einsatz kommt, kann dann von der Modellgröße und dem zu simulierenden Zeitintervall abhängig gemacht werden.

Anforderung MOD.5: Unterstützung bei der Suche nach Modellierungsfehlern

Von besonderer Wichtigkeit während der Modellbildungsphase ist die Unterstützung des Ingenieurs beim sogenannten *Modell-Debugging*, der Suche nach Modellierungsfehlern. Zeigt ein Modell bei seiner Auswertung qualitativ falsches Zeitverhalten oder treten numerische Fehler wie z. B. eine Division durch Null oder eine Überschreitung eines Wertebereichs auf, so muss der entsprechende, fehlerhafte Teil der Modellspezifikation gefunden werden. Dazu müssen ähnlich einem Programmiersprachen-Debugger Möglichkeiten zum schrittweisen Verfolgen der Modellauswertung und zum Protokollieren von ausgewählten Modellgrößen bereitgestellt werden. Zur Untersuchung

von Grenzfällen muss eine Neubesetzung einzelner Größen möglich sein.

Die weiter oben bereits aufgestellte Forderung nach einer transparenten Anknüpfung an die ursprüngliche Modellspezifikation (Anforderung M.2) ist für das Modell-Debugging eine Grundvoraussetzung. Im Kontext der Suche nach Modellierungsfehlern verschärft sich diese Forderung sogar noch, da auch die schrittweise Verfolgung der Modellauswertung möglichst eng an die ursprüngliche Modellspezifikation geknüpft sein sollte.

3.1.2 Reglersynthese und Systemoptimierung

Bei der Reglersynthese und anschließenden Systemoptimierung kommen schwerpunktmäßig Verfahren der linearen Systemtheorie zum Einsatz, daher soll im Folgenden auf die besonderen Anforderungen an eine verarbeitungsorientierte Modellrepräsentation im Hinblick auf die Verarbeitung linearer Systeme eingegangen werden. Nichtlineare Synthese- und Optimierungsverfahren beruhen auf der nichtlinearen Simulation, die daraus resultierenden Anforderungen sind bereits im vorangegangenen Kapitel 3.1.1 aufgeführt worden.

Als Ergebnis der Modellbildung hat der Ingenieur üblicherweise nichtlineare Modelle der einzelnen Komponenten vorliegen. Um darauf die Verfahren der linearen Systemtheorie zur Reglersynthese anwenden zu können, müssen die Modelle in eine lineare Darstellung überführbar sein. Dieser als *Linearisierung* bezeichnete Vorgang muss von der verarbeitungsorientierten Modellrepräsentation unterstützt werden. Im Besonderen muss die verarbeitungsorientierte Modellrepräsentation zusätzlich zum linearen Modell alle für die linearen Verfahren notwendigen Informationen bereitstellen. Konkret ergeben sich daraus im Wesentlichen die nachfolgend aufgeführten Anforderungen.

Anforderung SYN.1: Abbildung linearer Systeme als Matrizen

Die Verfahren der linearen Systemtheorie setzen voraus, dass das Systemverhalten der einzelnen Komponenten in Form von Matrizen beschrieben ist. Die Darstellung der Matrizen kann sowohl rein numerisch als auch symbolisch in Abhängigkeit vom Betriebspunkt und den Systemparametern erfolgen. Aus Genauigkeitsgründen ist in der Regel die symbolische Variante vorzuziehen. Ausserdem sind aus der symbolischen Matrixdarstellung noch einige Systemzusammenhänge ablesbar, die anhand der rein numerische Werte nicht mehr erkennbar sind. Die verarbeitungsorientierte Modellrepräsentation muss also eine Mechanismus bieten, der es ermöglicht, das Systemverhalten in Form von ausführbaren symbolischen Matrizen komponentenweise bereitzustellen.

Anforderung SYN.2: Verkopplung symbolischer, linearer Teilsysteme

Um effizientes, komponentenorientiertes Arbeiten zu ermöglichen, muss die verarbeitungsorientierte Modellrepräsentation einen Mechanismus zur Verkopplung der linearen Teilsysteme bereitstellen. So ist es möglich, dass jedes Komponentenmodell nur einmal linearisiert werden muss. Bei zusammengesetzten Systemen sind dann jeweils nur die Verkopplungsmatrizen neu zu ermitteln. Damit lassen sich auch die linearen Modelle modular verwenden, eine Veränderung am Gesamtsystem zieht nicht jedes Mal eine komplette Neuermittlung der Gesamtsystemmatrizen nach sich.

3.1.3 Reglerrealisierung und Systemtest

In der Entwurfsphase der Reglerrealisierung und des Systemtests geschieht die schrittweise Ersetzung des Modells durch reale Komponenten. Bei der Reglerrealisierung werden die Regler auf einem Digitalrechner implementiert. Der Rest des Systems liegt dabei üblicherweise zunächst noch als Modell im Rechner vor. Ein zweiter, sehr wichtiger Bestandteil in dieser Entwurfsphase ist die Hardware-in-the-Loop-Simulation. Hier werden Teilmodelle in Echtzeit mit Anbindung an reale Systemkomponenten simuliert. Aus diesen Hauptszenarien der Reglerimplementierung auf einem Digitalrechner und der Echtzeit-Simulation mit eingebundener realer Hardware ergeben sich an eine verarbeitungsorientierte Modellrepräsentation die nachfolgend aufgeführten Anforderungen.

Anforderung REA.1: Schlanke und portable Software-Architektur

Soll eine verarbeitungsorientierte Modellrepräsentation sowohl bei der Offline-Simulation als auch bei der Implementierung der Regler und der HIL-Simulation eingesetzt werden, so stellt dies besondere Anforderungen an die Software-Architektur. Um einen hohen Wiederverwendungsgrad zu erreichen, ist ein hohes Maß an Portabilität erforderlich, da Offline-Simulation, Reglerimplementierung und HIL-Simulation üblicherweise auf jeweils unterschiedlichen Hardware-Plattformen durchgeführt werden. Des Weiteren stehen bei der Reglerimplementierung und der HIL-Simulation im Vergleich zu einer Workstation oder einem PC nur beschränkte Ressourcen zur Verfügung. Für diese Zwecke muss also ein sehr schlanker Simulationskern zur Verfügung stehen.

Anforderung REA.2: Echtzeitfähigkeit

Sowohl für die Implementierung eines Reglers als auch für die HIL-Simulation ist es von höchster Wichtigkeit, dass der Reglercode bzw. das zu simulierende

Restmodell echtzeitfähig sind. Für eine verarbeitungsorientierte Modellrepräsentation ist daher unbedingt zu fordern, dass sie echtzeitfähige Modelle liefert. Dies bedeutet zum einen ein deterministisches, beschränktes Laufzeitverhalten. Modellierungsansätze, die z. B. auf iterativen Lösungsverfahren beruhen, sind ungeeignet, ebenso Implementierungen, die unkontrolliert dynamische Speicherverwaltung und Garbage Collection einsetzen. Des Weiteren müssen Mechanismen integrierbar sein, die eine Laufzeitüberwachung erlauben. Echtzeit-Simulationen

Anforderung REA.3: Codeeffizienz

Bei der Reglerimplementierung und auch bei Echtzeit-Simulationen liegt das ausführbare Modell üblicherweise in Form von kompiliertem Code vor. Die Effizienz des aus der verarbeitungsorientierten Modellrepräsentation generierten Codes muss sowohl hinsichtlich der Laufzeit als auch hinsichtlich des Speicherplatzes sehr hoch sein. Bezüglich der Laufzeit muss der gesamte Code für einen Zeitschritt innerhalb eines Abtastschritts abgearbeitet sein, anderenfalls wären die Echtzeitbedingungen nicht eingehalten. Des Weiteren steht bei der Echtzeithardware nur beschränkter Speicherplatz zur Verfügung, so dass eine speichersparende Codierung der Modelle erforderlich ist.

Anforderung REA.4: Geringe Hardwareanforderungen

Die bei der Reglerrealisierung zu implementierenden Regler sind in vielen Fällen Bestandteil eines eingebetteten Systems, das als lokale Intelligenz einer Systemkomponente eine komponenteninterne Funktion zu erfüllen hat. Zu diesem Zweck werden aus Kostengründen typischerweise Mikrocontroller eingesetzt. Für die Implementierung der verarbeitungsorientierten Modellrepräsentation muss sichergestellt sein, dass auch diese einfachen Prozessoren unterstützt werden können.

Anforderung REA.5: Unterstützung der Parallelverarbeitung

Parallelität ist eine inhärente Eigenschaft mechatronischer Systeme. Die vielen verschiedenen Komponenten arbeiten gleichzeitig, teilweise abhängig, teilweise unabhängig voneinander. Eine verarbeitungsorientierte Modellrepräsentation muss daher in der Lage sein, verteilte, nebenläufige Systeme zu unterstützen. Dabei müssen zum einen Modellierungsmittel zur Beschreibung paralleler Systemstrukturen bereitgestellt werden, zum anderen muss eine verteilte, nebenläufige Implementierung der Laufzeitplattform zur Verfügung stehen, um verteilte Lösungen unter realitätsnahen Bedingungen evaluieren zu können. Des Weiteren ist bei großen Modellen die Leistungsfähigkeit eines einzelnen Prozessors nicht ausreichend, alle erforderlichen Berechnungen innerhalb eines Abtastschritts auszuführen. Es ist ein

paralleles Prozessornetzwerk erforderlich, um die HIL-Simulation in der geforderten Geschwindigkeit durchführen zu können.

Eine sehr aufwendige Arbeit beim Entwurf verteilter Systeme ist die Ermittlung der optimalen parallelen Struktur. Neben geeigneten Sprachmitteln zur Beschreibung der Systempartitionierung ist es sehr hilfreich, wenn auf der Ebene der verarbeitungsorientierten Modellrepräsentation Verfahren zur halb- und vollautomatischen Systempartitionierung zur Verfügung stehen.

Anforderung REA.6: Automatische Codegenerierung

Die unter Echtzeitbedingungen erforderlichen sehr kurzen Rechenzeiten lassen sich in vielen Fällen nur mit Hilfe von kompiliertem Code erreichen. Von einer verarbeitungsorientierten Modellrepräsentation ist also zu fordern, dass sie umfangreiche Unterstützung für eine automatische Codegenerierung bietet. Optimalerweise ist sie so angelegt, dass sie direkt ohne umfangreiche Transformationen in Programmcode z. B. der Programmiersprache C umgesetzt werden kann. Dazu sollten verarbeitungsrelevante Informationen wie z. B. die Auswertereihenfolge und Verkopplungsstrukturen bereits in der verarbeitungsorientierten Modellrepräsentation entsprechend aufbereitet sein.

Um den Zeitbedarf für eine automatische Codegenerierung auch bei großen Systemen in vertretbaren Grenzen zu halten, muss eine komponentenweise, kontextunabhängige Codegenerierung möglich sein. Um Totzeiten innerhalb des Systems zu minimieren, sollte dabei auch bei den bereits kompilierten Modulen die innere Auswertereihenfolge noch beeinflussbar sein.

Anforderung REA.7: Anbindung von Sensorik/Aktorik

Bei der HIL-Simulation bildet die Sensorik und Aktorik die Schnittstelle zwischen der realen Systemkomponente und dem simulierten Modell des übrigen Systems. Die verarbeitungsorientierte Modellrepräsentation muss also eine Schnittstelle zur Anbindung von Sensoren und Aktoren bieten. Dazu müssen zunächst entsprechende Sprachkonstrukte zur Verfügung stehen. Des Weiteren muss es möglich sein, beliebige externe Programmbibliotheken anbinden zu können, nur so ist die Offenheit gegenüber jeder möglichen Hardwareanbindung gewährleistet. Aufgrund der hohen Verbreitung der Programmiersprache C in diesem Umfeld ist es sinnvoll, diese Schnittstelle in C zu realisieren.

3.1.4 Softwaretechnische Anforderungen

Bisher hat sich der bereits aufgestellte Anforderungskatalog für eine verarbeitungsorientierte Modellrepräsentation auf die Bedürfnisse des Ingenieurs mit dem dazugehörigen Entwurfsprozess konzentriert. Berücksichtigt man die Tatsache, dass auch die vom Ingenieur eingesetzte Entwicklungsumgebung mit den dazugehörigen Softwarewerkzeugen entwickelt, gewartet und

erweitert werden muss, so lassen sich aus softwaretechnischer Sicht weitere Anforderungen an eine verarbeitungsorientierte Modellrepräsentation formulieren.

Will man dem Anspruch einer universellen, offenen Entwicklungsumgebung für mechatronische Systeme genügen, so sind konkret die nachfolgend aufgeführten Anforderungen zu erfüllen.

Anforderung SWT.1: Domänenübergreifende Systemkomposition

Mechatronische Systeme unterschiedlichster Art und Herkunft, in unterschiedlichen Modellbeschreibungssprachen und in unterschiedlichen Modellierungsarten spezifiziert, können miteinander kombiniert werden.

Anforderung SWT.2: Modulare Erweiterbarkeit

Die Entwicklungsumgebung ist modular erweiterbar:

- Neue Verfahren zur Simulation, linearen Analyse, Reglersynthese oder sonstigen Verarbeitung können hinzugefügt werden, dabei brauchen keine neuen Frontends zur Verarbeitung der Modellbeschreibungssprachen geschaffen werden. Die Vorhandenen können von den neuen Methoden benutzt werden.

- Zur Einführung einer weiteren Modellbeschreibungssprache ist nur ein zusätzliches Frontend erforderlich. Dieses kann mit Übersetzerentwicklungsumgebungen recht einfach erzeugt werden. Alle weiterverarbeitenden und auswertenden Module können unverändert benutzt werden.

Anforderung SWT.3: Schnittstelle zwischen Modellbeschreibung und Weiterverarbeitung

Die verarbeitungsorientierte Modellrepräsentation bildet die Schnittstelle zwischen der Beschreibung der Systeme und ihrer Weiterverarbeitung.

Architektur einer offenen Entwicklungsumgebung

Aus den genannten, softwaretechnischen Anforderungen ergibt sich direkt die in Abbildung 3.1 dargestellte Architektur für die Entwicklungsumgebung. Zugunsten einer hohen Modularität und einfacher Wartbarkeit ist es dabei sinnvoll, dass die einzelnen Frontends und Backends als kleine, unabhängige Programme realisiert werden. In einzelnen Fällen kann es auch sinnvoll sein, sie in Form von anderen Betriebssystemobjekten wie z. B. Shared Libraries zur Verfügung zu stellen. Dabei sollte aber stets die Unabhängigkeit von den verwendeten Implementierungswerkzeugen gegeben sein.

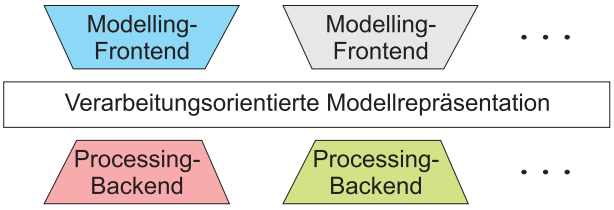


Abbildung 3.1: Architektur einer offenen Entwicklungsumgebung

3.2 Mechatronic Processing Objects als Basis einer offenen Entwurfsumgebung

Nachdem im vorangegangenen Unterkapitel die wesentlichen Anforderungen an eine verarbeitungsorientierte Modellrepräsentation für mechatronische Systeme aufgelistet wurden, soll nun das Konzept einer universellen, offenen Entwurfsumgebung vorgestellt werden, das diesen Anforderungen genügt. Die Basis dieser Entwurfsumgebung bilden die im Mittelpunkt dieser Arbeit stehenden Mechatronic Processing Objects (MPO), die die Rolle der verarbeitungsorientierten Modellrepräsentation einnehmen.

Einen Gesamtüberblick über das Konzept gibt Abbildung 3.2. Bevor näher auf die Arbeitsweise der Entwurfsumgebung eingegangen wird, werden zunächst die beteiligten Objekte vorgestellt:

- **Modeller**
  Ein Modeller-Objekt repräsentiert ein physikalisch-topologisches oder mathematisches Modellierungswerkzeug, mit dem der Ingenieur ein Teilmodell erstellt. Zur besseren Übersichtlichkeit wird an dieser Stelle nicht explizit zwischen der physikalisch-topologischen und der mathematischen Modellierungsebene unterschieden. Die Modellierungswerkzeuge werden jeweils als eine Einheit angesehen, bestehend aus einer Komponente zur physikalisch-topologischen Modellbildung und einer Komponente zur Überführung in die entsprechende mathematische Darstellung.

- **Dynamic System Code (DSC)**
  Dynamic System Code (DSC) ist eine maschinenunabhängige, verarbeitungsorientierte Modellbeschreibungsform in ASCII-Notation, die einfach weiterverarbeitet und problemlos zwischen unterschiedlichsten Rechnerplattformen ausgetauscht werden kann. DSC dient zur textuellen Beschreibung der einzelnen Mechatronic Processing Objects.

- **DSC frontend**
  Ein DSC-Frontend ist ein Übersetzer, der eine mathematische Modellbeschreibung in die verarbeitungsorientierte Modellbeschreibungsform

DSC umsetzt. Für jeden eingesetzten Modeller ist ein eigenes DSC-Frontend zu erstellen. Setzt man Übersetzerentwicklungsumgebungen ein, so lässt sich ein neues DSC-Frontend ohne großen Aufwand implementieren.

- **model.dsc**
  Dieses Dateiobjekt ist die DSC-Darstellung eines Teilmodells. Als ASCII-Notation können die DSC-Darstellungen der einzelnen Teilmodelle einfach weiterverarbeitet und problemlos zwischen unterschiedlichsten Rechnerplattformen ausgetauscht werden.

- **MPO**
  Ein Mechatronic Processing Object (MPO) ist die ausführbare, verarbeitungsorientierte Modellrepräsentation einer Komponente eines mechatronischen Systems. Das MPO bildet den Mittelpunkt der hier vorgestellten, offenen Entwurfsumgebung. Ein MPO ist unter anderem in der Lage, die verarbeitungsorientierte Systembeschreibungsform DSC direkt auszuführen. Des Weiteren können sich mehrere MPOs zu einem höherwertigen System zusammenschließen.

- **MPO-NameServer**
  Damit ein MPO mit einem anderen kommunizieren kann, muss ihm sein Kommunikationspartner bekannt sein. Dazu dient der MPO-NameServer, der eine Art Namensdienst verkörpert. Über die Angabe eines eindeutigen Namens bekommt ein MPO vom MPO-NameServer seinen Kommunikationspartner geliefert.

- **Backend**
  Ein Backend implementiert ein Verfahren zur Simulation, linearen Analyse, Reglersynthese, Optimierung oder sonstigen Verarbeitung eines MPO. Besteht ein System aus mehreren MPOs, so geschieht die Anbindung an das Backend üblicherweise über das MPO auf der höchsten Hierarchieebene.

Insgesamt lässt sich das Grundprinzip der Entwurfsumgebung folgendermaßen darstellen: Der Ingenieur spezifiziert Modelle in unter Umständen unterschiedlichen Modellbeschreibungssprachen oder Modellierungsarten. Diese Modelle werden dann durch entsprechende Frontends in die jeweils entsprechende Darstellung der verarbeitungsorientierten Modellbeschreibungsform DSC übersetzt. Dabei erzeugt jeder unterschiedliche Modellbeschreibungszweig mindestens ein DSC-Modul. Jedes DSC-Modul wird von einem Mechatronic Processing Object eingelesen und steht damit als ausführbares Modell zur Verfügung. Mit Hilfe eines Namensdienstes können sich die vielen einzelnen MPOs zu einem Gesamtsystem formieren. Das resultierende Gesamtsystem kann dann in unterschiedlichster Art weiterverarbeitet werden:

- Generierung von Hochsprachen-Programmcode
- Generierung von Maschinencode (z. B. für digitale Regler)
- Interpretative Simulation oder Modell-Debugging
- Aussortieren der Zustandsmatrizen bei linearen Systemen und Weiterverarbeitung in üblichen Verfahren der Linearen Theorie
- Parameteroptimierung

Die vorgestellte Entwurfsumgebung entspricht den in Kapitel 3.1.4 aufgestellten softwarearchitektonischen Anforderungen. Ein Modellierungswerkzeug mit dem entsprechenden DSC-Frontend bildet jeweils ein Modelling-Frontend. Die Interface-Schicht der verarbeitungsorientierten Modellrepräsentation wird von der Modellbeschreibungsform DSC und den Mechatronic Processing Objects einschließlich des Namensdienstes gebildet. Die Processing-Backends sind jeweils eigenständige Werkzeuge, die auf dem MPO des Gesamtsystems arbeiten.

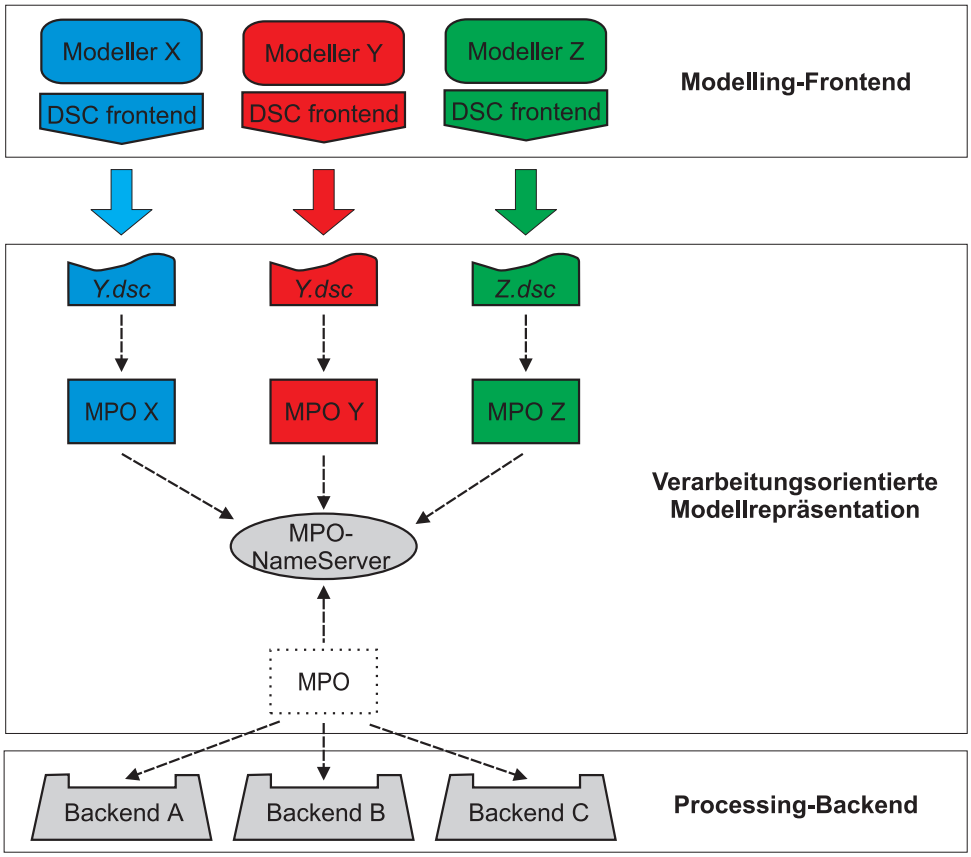


Abbildung 3.2: Gesamtkonzept der offenen Entwurfsumgebung

Kapitel 4

Mechatronic Processing Objects und ihre Repräsentation

Nachdem im vorangegangenen Kapitel die Gesamtkonzeption einer offenen Entwurfsumgebung für mechatronische Systeme auf Basis einer verarbeitungsorientierten Modellrepräsentation vorgestellt wurde, sollen nun die eigentlichen Modellobjekte auf verarbeitungsorientierter Ebene, die Mechatronic Processing Objects, näher beschrieben werden. Im ersten Abschnitt wird zunächst eine modulare Objektstruktur präsentiert, die ein Teilmodell in Form einer interpretativ auswertbaren Datenstruktur, in Form von effizient ausführbarem Zielcode oder auch als lineare Systemrepräsentation enthalten kann. Bei Bedarf kann die Repräsentationsform dynamisch ausgetauscht werden. Im zweiten Abschnitt wird das zugrundeliegende mathematische Modell vorgestellt. Es erlaubt die Beschreibung modular-hierarchischer, gemischt diskret-kontinuierlicher Systeme. Dabei werden im Besonderen die Randbedingungen auf dem Gebiet der Mechatronik berücksichtigt. Der dritte Abschnitt dieses Kapitels beschäftigt sich mit den Aktionen, die ein Mechatronic Processing Object ausführen kann. Interaktionen mehrerer MPOs untereinander als auch die Schnittstelle zu anderen Programmobjekten werden näher erläutert. Im letzten Abschnitt werden die zentralen Aspekte der Implementierung vorgestellt. Aufbauend auf einer speziell für diese Zwecke entwickelten Spezifikationssprache zur Beschreibung komplexer Datenstrukturen kann ein Großteil der Funktionalität eines MPO aus einer einfachen, kompakten Spezifikation automatisch generiert werden. Des Weiteren werden unterschiedliche Varianten zur Realisierung nebenläufiger MPOs vorgestellt. Die problemlose Portierbarkeit auf unterschiedlichste Rechnerarchitekturen ist damit sichergestellt.

4.1 Objektstruktur

Ein Mechatronic Processing Object (MPO) ist die verarbeitungsorientierte Repräsentation eines autonomen, dynamischen Systems unter besonderer Berücksichtigung der Randbedingungen auf dem Gebiet der Mechatronik. Als eigenständig ablauffähiges, nebenläufiges Programmobjekt repräsentiert es ein Teilmodell eines mechatronischen Systems in Form einer ausführbaren Systemspezifikation mit gerichtetem Ein-/Ausgangsverhalten. Dabei lassen sich sowohl kontinuierliche als auch diskrete Aspekte abbilden.

Zur Modellierung komplexer Systeme stellen MPOs drei Grundelemente bereit: generische Systemtypen, Systeminstanzen und hierarchische Koppelstrukturen.

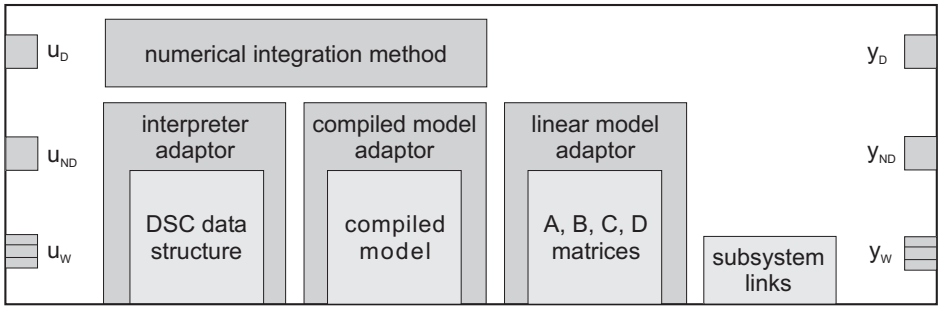


Abbildung 4.1: Mechatronic Processing Object

Ein MPO kann als Block mit gerichtetem Ein-/Ausgangsverhalten betrachtet werden (siehe Abbildung 4.1). Die ausführbare Spezifikation des Systemverhaltens kann dabei intern in den folgenden drei Repräsentationen vorliegen:

1. *als interpretative Datenstruktur*
   Diese Repräsentation enthält alle verarbeitungsrelevanten Informationen des entsprechenden Teilsystems inkl. der Rückannotation auf die ursprüngliche Beschreibung auf mathematischer bzw. topologischer Modellierungsebene. Textuell kann der Inhalt dieser Datenstruktur in Form der verarbeitungsorientierten Beschreibungsform DSC (Dynamic System Code) abgelegt werden. Die Datenstruktur kann vom MPO-Kern interpretativ abgearbeitet oder auch modifiziert werden. Diese Repräsentation eignet sich insbesondere zum Modell-Debugging und zur Modellpartitionierung. Des Weiteren bildet sie den Ausgangspunkt für die beiden anderen Repräsentationsformen.

2. *in linearer bzw. linearisierter Form als Matrizen $\underline{A}$, $\underline{B}$, $\underline{C}$, $\underline{D}$*
   Zur linearen Systemanalyse und zur Reglersynthese wird das Systemverhalten in linearer bzw. linearisierter Form benötigt. Die Ermittlung

dieser Matrizen kann durch symbolische Linearisierung auf Gleichungsebene und anschließende numerische Verkopplung oder durch numerische Linearisierung erfolgen. Auf diese einmal gewonnenen Darstellungen können dann sehr effizient Methoden der linearen Systemanalyse und -synthese angewendet werden.

3. *als kompilierter Code (DLL)*
   Für Systemsimulationen mit langen Simulationsintervallen, Parameterstudien und insbesondere Echtzeitanwendungen ist die Laufzeitperformanz des Modells von höchster Bedeutung. Dazu bietet jedes MPO die Einbindung des Systemverhaltens als kompilierter Code an. Dieser kann zur Laufzeit geladen, entfernt oder ausgetauscht werden. Dies erfordert besonders im Hinblick auf die Vernetzung mehrerer MPOs eine hohe Flexibilität des Codes. Auch die kompilierte Repräsentation eines MPO kann sich an den Gesamtsystemkontext adaptieren, so kann z. B. zur Laufzeit die Auswertereihenfolge der Modellgleichungen verändert bzw. neu ermittelt werden.

Jedes MPO kann wahlweise eine oder auch mehrere dieser drei Repräsentationen annehmen, zur Laufzeit wechseln, hinzufügen oder auch entfernen. Zum Zugriff auf diese Repräsentationen steht jeweils ein Adapter zur Verfügung, der ebenfalls entfernt werden kann, falls eine Repräsentationsform nicht verwendet wird.

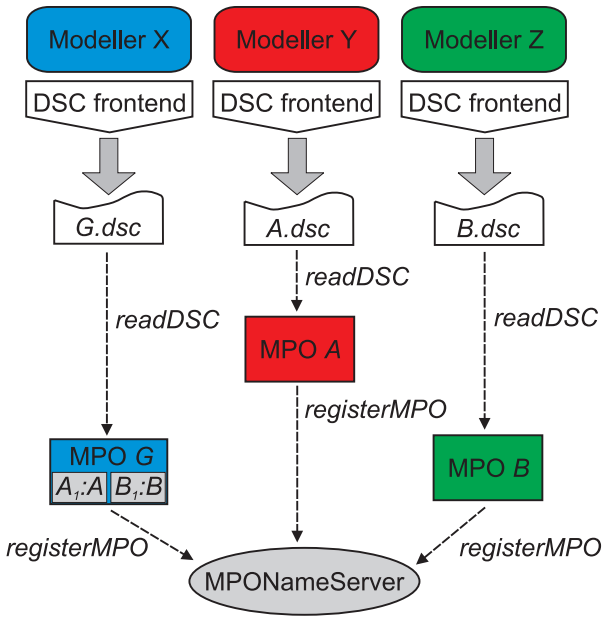


Abbildung 4.2: Anbindung von Mechatronic Processing Objects an die Modellbildung

Die Anbindung der MPOs als Modellbasis auf der Ebene der rechentechnischen Verarbeitung an die Modellbildung auf topologischer und mathematischer Ebene ist in Abbildung 4.2 schematisch dargestellt. Einzelne Komponenten des Systems werden mit evtl. unterschiedlichen, fachspezifischen Modellierungswerkzeugen beschrieben. Entsprechende Übersetzer-Frontends erzeugen daraus jeweils eine verarbeitungsorientierte Beschreibung in Form der textuellen Notation DSC, pro Komponente mindestens ein Textobjekt. Für jedes dieser DSC-Textobjekte wird ein MPO erzeugt, welches das jeweilige Textobjekt mit Hilfe der Methode *readDSC* einliest. Damit stehen die einzelnen Modellkomponenten auf verarbeitungsorientierter Ebene jeweils in Form eines MPO zur Verfügung.

Über Koppelpunkte kann es sich mit anderen Mechatronic Processing Objects verbinden und somit dynamisch zur Laufzeit ein beliebig komplexes, strukturvariantes System aufbauen.

4.2 Mathematisches Modell

Das einem Mechatronic Processing Object zugrundeliegende mathematische Modell erlaubt eine einheitliche Darstellung von kontinuierlichen, diskreten und hybriden Systemen. Das mathematische Modell ist unter Anlehnung an die Hybriden Automaten [Hen96] entstanden. Dabei werden insbesondere die Anforderungen der Mechatronik sowohl bezüglich strukturierter, modular-hierarchischer Modellbildung als auch bezüglich effizienter, rechentechnischer Verarbeitung berücksichtigt. Im Gegensatz zu vielen auf Spezialformen hybrider Automaten [ACHH93] basierenden Werkzeugen wie z. B. HYTECH werden die kontinuierlichen Anteile nicht in ein diskretes Abbild umgewandelt, sondern bleiben als ODE erhalten.

Schnittstellendefinition / Ein- und Ausgangsgrößen

Zur Beschreibung der Schnittstelle zwischen dem modellierten Subsystem und seiner Umgebung stehen die in Tabelle 4.1 aufgeführten Größen zur Verfügung. Im allgemeinen Fall kann das Verhalten eines mechatronischen

$t \in \mathbb{R}_{\geq 0}$	Systemzeit
$\underline{p} \in \mathbb{R}^{n_p}$	Systemparameter
$\underline{u} \in \mathbb{R}^{n_u}$	kontinuierliche Eingangsgrößen
$\underline{y} \in \mathbb{R}^{n_y}$	kontinuierliche Ausgangsgrößen
$\underline{v} \in \Sigma_{in} \times \mathbb{R}^*$	diskrete Eingänge (Messages)
$\underline{z} \in \Sigma_{out} \times \mathbb{R}^*$	diskrete Ausgänge (externe Events)

Tabelle 4.1: Schnittstellengrößen

Systems zeitabhängig sein. Dazu muss im Modell eine globale Zeit zur Verfügung stehen, hier repräsentiert durch die Systemzeit t. Eine weitere, für

technische Systeme sehr wichtige Schnittstellengröße sind die Systemparameter $\underline{p}$. Die Systemparameter bezeichnen Kenngrößen eines Systems, die für eine konkrete Ausprägung konstant sind, wie z. B. die Masse oder Länge eines Bauteils. Zur Abbildung der Ein- und Ausgänge eines Subsystems stehen entsprechende Ein- und Ausgangsgrößen zur Verfügung. Die Abbildung kontinuierlichen Verhaltens erfolgt mit den kontinuierlichen Eingangsgrößen $\underline{u}$ und den kontinuierlichen Ausgangsgrößen $\underline{y}$. Zur Modellierung ereignisorientierter Systemanteile stehen die diskreten Eingänge $\underline{v}$ (*Messages*) und die diskreten Ausgänge $\underline{z}$ (*externe Events*) zur Verfügung. Die Menge aller möglichen Messages ist wie bei einem endlichen Automaten definiert durch das Eingangsalphabet Σ_{in}. Darüber hinaus kann jede Message zusätzlich ein n-Tupel reeller Werte enthalten, dies ist insbesondere bei der Modellierung von Kommunikationsverbindungen zwischen Subsysteme sinnvoll. Die Menge der externen Events wird durch das Ausgabealphabet Σ_{out} beschrieben. Auch hier ist als Erweiterung die Verknüpfung mit reellen Werten möglich.

Systemgrößen

In der Systembeschreibung verwendete Größen, die ausserhalb der Systemgrenze nicht sichtbar sind, werden als Systemgrößen bezeichnet. Eine Aufstellung der im vorliegenden Modell verwendeten Systemgrößen ist in Tabelle 4.2 zu finden. Die Systemgrößen werden häufig auch als Zustandsgrößen

$\underline{x} \in \mathbb{R}^{n_x}$	kontinuierliche Zustandsgrößen mit Startzustand $\underline{x}_0$
$\underline{s} \in \mathbb{R}^{n_s}$	diskrete Speichergrößen mit Startzustand $\underline{s}_0$
$q \in Q$	diskreter Zustand mit Startzustand q_0
$\underline{e} \in \Sigma_{int} \times \mathbb{R}^*$	interne Events

Tabelle 4.2: Systemgrößen

bezeichnet. Ihre Werte charakterisieren den aktuellen Systemzustand. Im vorliegenden Modell gibt es drei Arten von Zustandsgrößen. Die kontinuierlichen Zustandsgrößen $\underline{x}$ sind durch sowohl zeit- als auch wertkontinuierliche Veränderung gekennzeichnet. Die diskreten Speichergrößen $\underline{s}$ haben zwar einen kontinuierlichen Wertebereich, aber ihre Werte ändern sich nur zu diskreten Zeitpunkten, nur dann, wenn das System seinen Kontrollmodus wechselt. Der diskrete Zustand q beschreibt den Kontrollmodus, in dem sich das System aktuell befindet. Die Menge aller möglichen Kontrollmodi ist in der Zustandsmenge Q zusammengefasst. Zur Erreichung einer definierten Anfangssituation muss für jede der Zustandsgrößen ein Wert vorgegeben werden, der dem Startzustand entspricht.

Eine Besonderheit bei den Systemgrößen sind die internen Events $\underline{e}$. Sie stellen systeminterne Ereignisse dar, die abhängig von den übrigen Systemgrößen ausgelöst werden können (siehe Triggerfunktion 4.3). Diese systeminternen Ereignisse bilden die Schnittstelle zwischen dem diskreten und dem

kontinuierlichen Modellanteil.

Systemgleichungen

Die Beschreibung der Zustandsänderungen eines Systems erfolgt in den Systemgleichungen. Die Beschreibung der kontinuierlichen Zustandsänderungen erfolgt in der Zustandsgleichung f:

$$\underline{\dot{x}} \quad = \quad f(q, \underline{s}, \underline{x}, \underline{u}, \underline{p}, t) \tag{4.1}$$

Vergleicht man diese Darstellung mit der Zustandsraumdarstellung 1. Ordnung, so besteht eine zusätzliche Abhängigkeit zum Kontrollmodus und den diskreten Speichergrößen. Diese Abhängigkeit kann man auch dahingehend interpretieren, dass es für jeden Kontrollmodus einen eigenen Satz von kontinuierlichen Zustandsgleichungen gibt.

Die Beschreibung der Kontrollübergänge geschieht durch die Transitionsfunktion δ:

$$(q', \underline{s}', \underline{x}_{start}) \quad = \quad \delta(q, \underline{s}, \underline{v}, \underline{e}, \underline{x}, \underline{u}, \underline{p}, t) \tag{4.2}$$

Liegt eine Message v oder ein interner Event e vor, so kann das System seinen Kontrollmodus ändern. In Abhängigkeit vom aktuellen Systemzustand und den Eingangsgrößen wechselt das System in den Kontrollmodus q'. Dabei können ebenfalls neue Startwerte $\underline{s}'$ und $\underline{x}_{start}$ für die diskreten Speichergrößen bzw. die kontinuierlichen Zustandsgrößen gesetzt werden.

Die Schnittstelle zwischen dem diskreten und dem kontinuierlichen Systemverhalten ist durch die internen Events $\underline{e}$ definiert. Die Erzeugung von internen Events aus den kontinuierlichen Systemgrößen erfolgt mit Hilfe der Triggerfunktion μ:

$$\underline{e} \quad = \quad \mu(q, \underline{s}, \underline{x}, \underline{u}, \underline{p}, t) \tag{4.3}$$

Ausgangsgleichungen

Die Berechnung der Ausgangsgrößen ist in den Ausgangsgleichungen festgelegt. Auch hier gibt es wieder jeweils separate Gleichungen für den kontinuierlichen und den diskreten Systemanteil. Die Ermittlung der kontinuierlichen Systemausgänge erfolgt mit Hilfe der kontinuierlichen Ausgangsgleichung g:

$$\underline{y}(t + \underline{t}_{delay,y}) \quad = \quad g(q, \underline{s}, \underline{x}, \underline{u}, \underline{p}, t) \tag{4.4}$$

Vergleicht man diese Darstellung mit der Zustandsraumdarstellung 1. Ordnung, so besteht wie bei der Zustandsgleichung auch hier eine zusätzliche Abhängigkeit zum Kontrollmodus und den diskreten Speichergrößen. Ausserdem kann noch eine Signalverzögerung über die Angabe einer Verzögerungskonstante $\underline{t}_{delay,y}$ angegeben werden. Hier ist allerdings zu beachten,

dass die mathematischen Analysemöglichkeiten bei Verwendung der Verzögerung sehr eingeschränkt sind. Diese Modellierungsoption eignet sich im Wesentlichen nur für die Simulation.

Das Auslösen von externen Events wird durch die diskrete Ausgangsgleichung λ definiert:

$$\underline{z} \quad = \quad \lambda(q, q', \underline{v}, \underline{e}) \tag{4.5}$$

Bei jedem Kontrollübergang einschließlich dem Übergang eines Kontrollmodus auf sich selbst kann ein externes Ereignis ausgelöst werden. Das auszulösende Ereignis wird bestimmt durch den Start- und Ziel-Kontrollmodus sowie eine Message oder einen internen Event.

4.3 Objekt-Interaktionen

Nachdem die Struktur und das mathematische Modell eines Mechatronic Processing Objects bereits vorgestellt worden sind, soll nun auf die Interaktionen eines MPO eingegangen werden. Dazu verfügt jedes MPO über eine Reihe von Methoden, die im Folgenden näher erläutert werden sollen.

4.3.1 Repräsentation als ASCII-Text

Zur Bereitstellung einer offenen, einfach handhabbaren, plattformübergreifenden Schnittstelle bietet jedes MPO eine Lese- und eine Schreibmethode für die textuelle Systembeschreibungsform DSC an.

readDSC

Die Methode `readDSC` liest eine DSC-Modellrepräsentation in Form eines ASCII-Textes ein und erzeugt daraus die entsprechenden internen Datenstrukturen des MPO. Nach Abschluss dieser Methode entspricht das im MPO abgebildete Modell vollständig der eingelesenen DSC-Spezifikation.

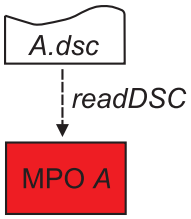


Abbildung 4.3: Einlesen der DSC-Spezifikation

writeDSC

Die Methode `writeDSC` dient dazu, das aktuell repräsentierte Modell in textueller Form abzulegen. Das MPO gibt dabei den Inhalt seiner internen Datenstrukturen in Form einer DSC-Spezifikation aus. Mit der Methode `readDSC` kann ein jedes MPO zu einem beliebigen Zeitpunkt diese Textausgabe wieder einlesen und damit denselben Modellzustand wieder einnehmen.

Abbildung 4.4: Ausgabe des repräsentierten Modells als DSC-Spezifikation

4.3.2 Instanzbildung und Einbindung externer Subsysteme

Beim Aufbau von komplexen, hierarchischen Systemen ist eine effiziente Wiederverwendbarkeit von einzelnen Komponenten erforderlich. Mechatronic Processing Objects bieten dazu eine Anzahl von Methoden, die die Bildung von Instanzen und die Einbindung externer Subsysteme unterstützen.

Eine wichtige Rolle bei den Vorgängen der Instanzbildung und der Verknüpfung mehrerer MPOs spielt der *MPONameServer*. Der *MPONameServer* repräsentiert einen Namensdienst, bei dem sich jedes MPO anmelden muss, wenn es für andere MPOs sichtbar sein soll.

Im Folgenden sollen nun die wichtigsten Methoden zur Kommunikation mit dem Namensdienst, zur Bildung von Instanzen und zur Kopplung von MPOs vorgestellt werden.

registerMPO

Die Registrierung eines Mechatronic Processing Objects beim Namensdienst geschieht durch das Ausführen der Methode `registerMPO`. Dabei werden der zu registrierende Name und der Objektverweis auf das MPO an den Namensdienst übergeben. Der Namensdienst hält eine Tabelle, in der für jeden registrierten Namen eines MPO ein Objektverweis auf das MPO abgelegt ist. Je nach Ausprägung der Laufzeitumgebung kann es sich bei einem Objektverweis z. B. um eine einfache Speicheradresse oder auch um den Verweis auf eine Task in einem Netzwerk handeln.

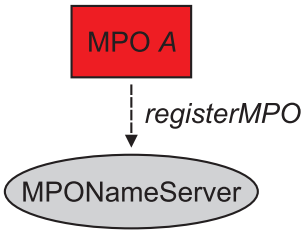


Abbildung 4.5: Registrierung eines MPO beim MPONameServer

getRegisteredMPO

Um zu einem registrierten Namen das entsprechende MPO zu erhalten, steht beim *MPONameServer* die Methode `getRegisteredMPO` zur Verfügung. Sie liefert zu dem angegebenen Namen den Objektverweis auf das dazugehörige MPO. Um den Kontakt mit einem anderen MPO aufzunehmen, ist nur ein einmaliger Aufruf dieser Methode erforderlich. Die weitere Kommunikation mit dem MPO kann dann über den zurückgelieferten Objektverweis direkt erfolgen. Eine Kommunikation mit dem Namensdienst ist also nur in den Konfigurationsphasen erforderlich. In laufzeitkritischen Phasen erfolgt die Kommunikation direkt über die maschinennahen Objektverweise.

getInstance

Ein mechatronisches System enthält oft mehrere Instanzen desselben Systemtyps. In solchen Fällen ist es sinnvoll, dass nur die instanzspezifischen Informationen mehrfach angelegt werden, die übrigen Informationen sollten aus Gründen der Speichereffizienz nur einfach vorhanden sein. Zu diesem Zweck stellt jedes MPO die Methode `getInstance` zur Verfügung. Sie liefert eine neue Instanz des MPO, wobei nur die instanzspezifischen Datenbereiche neu angelegt werden. Bei den übrigen Daten wird auf das ursprüngliche MPO verwiesen.

bind

Der eigentliche Bindevorgang, der aus mehreren MPOs ein gekoppeltes Gesamtsystem erzeugt, wird in der Methode `bind` durchgeführt. Eine schematische Darstellung des Bindevorgangs findet sich in Abbildung 4.6.

Gegeben seien drei Systemtypen G, A, und B. Das System G sei dabei ein hierarchisches System, das ein Subsystem A_1 vom Typ A und ein Subsystem B_1 vom Typ B enthalte. Jeder dieser Systemtypen wird durch ein entsprechendes MPO repräsentiert. Die Repräsentation der Subsysteme A_1 und B_1 enthält dabei zunächst nur die Angabe, dass es sich um Instanzen der Systemtypen A bzw. B handelt.kompilierter Struktur und Verhalten dieser Subsysteme sind dem MPO G noch nicht bekannt.

Die Auflösung der externen Referenzen erfolgt im Bindevorgang: Über eine Anfrage an den *MPONameServer* stellt das MPO des Gesamtsystems G eine Verbindung zu den MPOs der Systemtypen A und B her und erhält als Kopien die Instanzen A_1 und B_1, die es in seine innere Struktur einbaut. Enthält ein System mehrere Instanzen desselben Systemtyps, so werden nur die instanzspezifischen Datenbereiche mehrfach angelegt.

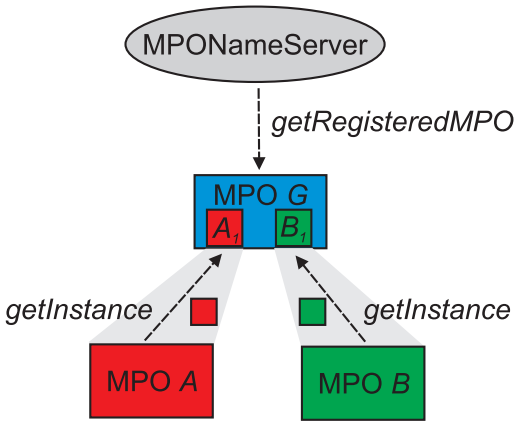


Abbildung 4.6: Bindevorgang

Die Instanzen können sowohl als erweiterte DSC-Datenstruktur, als kompilierter Code als auch als lineare Systemmatrizen eingebaut werden. Die im originalen Systemtyp abgelegten Repräsentationen können dazu unverändert übernommen worden. Eine erneute Erzeugung des kompilierten Codes für das gesamte System ist nicht mehr erforderlich, kompilierte Codestücke können modular ausgetauscht werden. Dies verkürzt besonders bei größeren Systemen die Turn-around-Zeiten bei Veränderungen an Teilsystemen erheblich.

4.3.3 Codeerzeugung

Ein weiteres wichtiges Anwendungsszenario einer verarbeitungsorientierten Systemrepräsentation ist die Erzeugung von kompiliertem Maschinencode zur Abbildung des Systemverhaltens. Diese Aufgabe wird von dem Dienst *Codegenerator* übernommen, der zentral oder bei Bedarf auch verteilt allen MPOs zur Verfügung steht.

getCompiledCode

Zur Erzeugung von kompiliertem Maschinencode steht die Methode `getCompiledCode` zur Verfügung. Mit der Anfrage `getCompiledCode` nimmt ein MPO Verbindung zu dem Dienst *Codegenerator* auf und stellt ihm seine DSC-Datenstruktur bzw. seine lineare Repräsentation zur Verfügung. Der

Codegenerator erzeugt daraus C-Programmcode, der mit Hilfe handelsüblicher Compiler übersetzt und als dynamisch ladbares Objekt (z. B. DLL, Shared Library) dem MPO zur Verfügung gestellt wird. Der erzeugte Code ist modular und kann sich an den Gesamtsystemkontext adaptieren. Dies ist insbesondere für Simulationsanwendungen von großer Bedeutung.

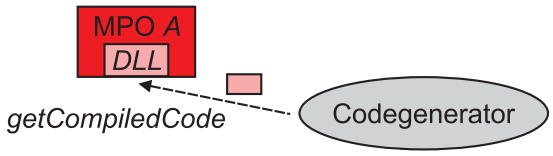


Abbildung 4.7: Erzeugung kompilierten Codes

4.3.4 Aufteilung in Cluster

Soll verteilte Informationsverarbeitung unterstützt werden, so müssen Partitionierungsmechanismen bereitgestellt werden. Mechatronic Processing Objects bieten dazu das Konzept der *Cluster* an. Ein Cluster ist ein Teil eines Systems, der z. B. einem Prozess, einem Prozessor oder einem Netzwerk zugeordnet ist. Für eine detaillierte Beschreibung der Cluster und ihrer Anwendung zur Abbildung verteilter Informationsverarbeitung sei hier auf Kapitel 6 verwiesen.

distribute

Zur Durchführung der Aufteilung eines Mechatronic Processing Objects in Cluster dient die Methode `distribute`. Ein entsprechendes Szenario ist in Abbildung 4.8 dargestellt. Gegeben sei ein MPO mit dem Gesamtsystem G.

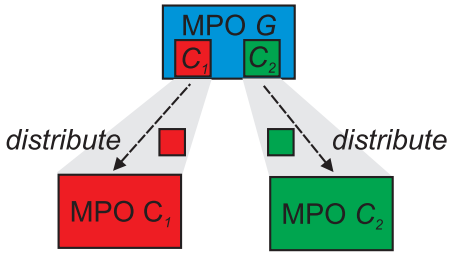


Abbildung 4.8: Aufteilung in Cluster

Einige Teile aus G sollen dem Cluster C_1 zugeordnet werden, andere dem Cluster C_2. Diese Bestimmung der Zuordnung kann z. B. durch eine automatische Modellpartitionierung (Lastverteilung) oder auch durch manuelle Benutzervorgaben erfolgt sein und ist bereits im MPO G abgelegt. Sofern noch nicht vorhanden, veranlasst MPO G die Erzeugung und Registrierung der

MPOs C_1 und C_2, die den jeweiligen Cluster repräsentieren. Anschließend werden die entsprechenden Teile von G in den jeweiligen Cluster ausgelagert und aus dem MPO G entfernt.

4.3.5 Verkopplung

Mechatronische Systeme bestehen typischerweise aus einer Hierarchie von gekoppelten Teilsystemen. Hierarchische Koppelstrukturen stellen daher auch den Normalfall bei der Repräsentation von gekoppelten Systemen durch Mechatronic Processing Objects dar. Zusätzlich bieten MPOs noch weitere Arten des Aufbaus von Systemkopplungen, die im Folgenden kurz vorgestellt werden sollen.

coupleOnBasicBlockLevel

Bei hierarchischen Ein-/Ausgangskopplungen werden nur die Zuordnungen der Ein- und Ausgänge zu den jeweils um eine Ebene tiefer liegenden Subsystemen sowie die Verkopplung dieser Subsysteme untereinander betrachtet, bei den hierarchischen Parameterkopplungen nur die Neubesetzung der Parameter der um eine Ebene tiefer liegenden Teilsysteme. Diese Informationen werden im Wesentlichen für die Interaktion mit dem Benutzer in weiterverarbeitenden Werkzeugen gebraucht.

Für eine effiziente Auswertung des Systemverhaltens ist es jedoch sinnvoller, die einzelnen Basissysteme direkt untereinander zu verkoppeln. Damit lässt sich die Anzahl der Schnittstellenvariablen deutlich reduzieren, was sich sowohl auf die Laufzeit als auch die Codegröße positiv auswirkt.

Zur Ermittlung der Verkopplung auf Basissystemebene steht die Methode `coupleOnBasicBlockLevel` zur Verfügung. Sie leitet aus den gegebenen hierarchischen Ein-/Ausgangs- und Parameterkopplungen die entsprechenden Verkopplungen auf Basissystemebene ab.

connectTo

Neben der Abbildung statischer Verkopplungsstrukturen unterstützen Mechatronic Processing Objects auch den Aufbau von dynamischen Kopplungsstrukturen. Der Aufbau von neuen Koppelstrukturen zur Laufzeit erfolgt mit

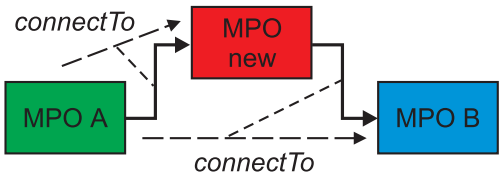


Abbildung 4.9: Dynamische Verkopplung

Hilfe der Methode `connectTo`. Ein entsprechendes Szenario ist in Abbildung

4.9 zu finden. Gegeben sind zwei MPO A und B. Über eine weiteres, neu erzeugtes MPO *new* will MPO A eine Verbindung zum MPO B aufbauen. Dazu führt es zweimal die Methode `connectTo` aus, wobei jeweils die zu verbindenden Aus- und Eingänge einschließlich des Quell- und Ziel-MPO angegeben werden.

4.3.6 Ermittlung der Auswertereihenfolge

Blockorientierte Modellierungsformen beschreiben das Systemverhalten in Form von gerichteten Ein-/Ausgangsbeziehungen. Bei der Systemsimulation ist dafür zu sorgen, dass keine algebraischen Schleifen auftreten, die gerichteten Beziehungen über alle Systemverkopplungen hinweg also zyklusfrei sind, anderenfalls gibt es keine korrekte Auswertereihenfolge der Systemgleichungen. Je nach Verkopplung der einzelnen Teilsysteme ergeben sich unterschiedliche Abhängigkeiten und damit auch jeweils andere Auswertereihenfolgen.

calcEvaluationOrder

Die freie Wahl einer beliebigen Auswertereihenfolge stellt im Besonderen an den kompilierten Code einige Anforderungen. Einzelne Berechnungspfade innerhalb eines MPO müssen getrennt voneinander ausführbar sein. Aus Effizienzgründen, insbesondere auch für Echtzeitanwendungen, sollte die Auswertereihenfolge für jedes MPO während der Simulation statisch bestimmt sein und nicht bei jedem Simulationsschritt erst datengetrieben ermittelt werden.

Die MPO-Repräsentation des Systemverhaltens als kompilierter Code stellt die einzelnen Berechnungspfade über einen Sprungverteiler zur Verfügung, so dass ein sehr schneller Zugriff auf den auszuführenden Code gewährleistet ist. Mit Hilfe der Methode `calcEvaluationOrder` wird in einem Initialisierungsschritt vor dem Start der Simulation und eventuell auch bei einem Wechsel des Kontrollmodus eine korrekte Auswertereihenfolge ermittelt und für jedes MPO eine lokale Schedule-Tabelle aufgebaut, die auf die einzelnen Einträge im Sprungverteiler verweist. Diese Tabelle kann sehr effizient abgearbeitet werden, der Performanzverlust gegenüber einer festcodierten Auswertereihenfolge ist sehr gering.

4.3.7 Linearisierung

Für die Anwendung von Verfahren der Linearen Systemtheorie ist es erforderlich, dass das Systemverhalten in Form der linearen Zustandsraumdarstellung 1. Ordnung vorliegt. Die dazu vom Mechatronic Processing Object bereitgestellte Methode soll im Folgenden kurz vorgestellt werden. Detaillierte Informationen finden sich im Kapitel 7.

linearize

Die Methode `linearize` ermittelt die linearen Systemmatrizen eines Mechatronic Processing Objects. Dies kann sowohl auf Basis der internen, interpretativen Datenstruktur als auch auf Basis des kompilierten Maschinencodes geschehen. Dabei werden die in Kapitel 7 beschriebenen Algorithmen angewendet, einschließlich der Ermittlung der linearen Darstellung für gekoppelte Systeme.

4.4 Implementierung

4.4.1 Definition, Erzeugung und Modifikation von Datenstrukturen

Der zentrale Bestandteil eines Mechatronic Processing Objects ist die Datenstruktur, die die verarbeitungsrelevanten Informationen des mechatronischen Teilsystems enthält. Diese Datenstruktur kann als eine Zwischensprache angesehen werden, sie bildet die Schnittstelle zwischen dem mathematischen Modell und der Weiterverarbeitung auf unterschiedlichsten Rechnerarchitekturen.

Speziell für die Definition und Implementierung dieser Zwischensprache wurde eine eigene Spezifikationssprache entwickelt, die *Intermediate Representation Language 2* (IRL2). IRL2 als Nachfolger von IRL (**I**ntermediate **R**epresentation **L**anguage [Hom93]) ist eine universelle Spezifikationssprache zur Beschreibung komplexer Datenstrukturen. Als Grundelemente stehen Strukturen, Alternativen und Sequenzen zur Verfügung. Zur Abbildung von Objektbeziehungen können relative und absolute Objektreferenzen verwendet werden. Durch die Möglichkeit der Einführung unterschiedlicher Sichtbarkeitsbereiche von Objekten ist dabei auch das Einbeziehen von externen Objekten möglich. Zur Unterstützung der Behandlung von Objekttypen und Objektinstanzen können sowohl typ- als auch instanzspezifische Attribute angelegt werden.

Zur Umsetzung von in IRL2 spezifizierten Zwischensprachen wurden mit Hilfe der Übersetzerentwicklungsumgebung Eli [KPJ98] Werkzeuge erstellt, die eine IRL2-Spezifikation automatisch in eine C-Implementierung der Zwischensprache umsetzen. Dabei steht ein umfangreicher Satz von Funktionen zum Datenzugriff und zur Datenmodifikation bereit — einschließlich dem Import und Export von Daten mit Hilfe einer plattformunabhängigen ASCII-Notation.

Der in den Mechatronic Processing Objects als Zwischensprache eingesetzte Dynamic System Code (DSC) ist vollständig in IRL2 spezifiziert. Die entsprechende C-Implementierung des MPO-Kerns sowie eine Grammatik zur Beschreibung der textuellen Modellrepräsentation können daraus automatisch mit Hilfe des IRL2-Übersetzers erzeugt werden.

Im Folgenden sollen nun die wesentlichen Aspekte von IRL2 vorgestellt werden. Dabei werden zunächst die einzelnen Spezifikationselemente vorgestellt. Im späteren Verlauf werden dann die Umsetzung in eine C-Implementierung der Zwischensprache und die textuelle Repräsentation erörtert.

Strukturen

Strukturen (*records*) bilden das Grundelement in IRL2. Eine Struktur kann beliebig viele Attribute besitzen. Diese Attribute sind jeweils wieder IRL2-Strukturen. Die Definition einer Struktur besteht aus der Angabe ihres Namens und der Aufzählung der Gesamtmenge ihrer Attribute, jeweils mit Angabe der Attributstruktur. Eine Struktur `MyStructure`, die ein Attribut `AttributeOne` der Struktur `StructureOne` und ein Attribut `AttributeTwo` der Struktur `StructureTwo` haben soll, muss dann folgendermaßen definiert werden:

```
STRUCTURE MyStructure IS
  AttributeOne : StructureOne;
  AttributeTwo : StructureTwo;
END
```

Die Strukturen `StructureOne` und `StructureTwo` müssen in der gleichen Datei definiert werden. Die Attributnamen innerhalb einer Struktur müssen unterschiedlich sein. Dabei ist zu beachten, dass IRL2 zwischen Groß- und Kleinschreibung nicht unterscheidet.

Als elementare, vordefinierte Strukturen stehen in IRL2 `INTEGER`, `REAL` und `STRING` zur Verfügung.

Varianten

IRL2 ermöglicht die Angabe von Attributen mit *alternativer* Struktur. Die Strukturen, die das Attribut einnehmen kann, werden durch das Zeichen '`|`' getrennt. Eine Zwischensprache mit der Struktur eines Binärbaums kann dann z. B. folgendermaßen beschrieben werden:

```
STRUCTURE BinaryTree IS
  Root         : InnerNode | Leaf;
END

STRUCTURE InnerNode IS
  Info         : Information;
  LeftSubtree  : BinaryTree;
  RightSubtree : BinaryTree;
END

STRUCTURE Leaf IS
```

```
  Info         : Information;
END
```

Sequenzen

Viele Strukturen beinhalten *Sequenzen* von gleichartigen Elementen. Dieser Tatsache wurde in IRL2 durch das Konstrukt `SEQ OF` Rechnung getragen. Ein allgemeiner Baum lässt sich mit diesem Konstrukt wie folgt beschreiben:

```
STRUCTURE GeneralTree IS
  Root     : Information;
  Subtrees : SEQ OF GeneralTree;
END
```

Der allgemeine Baum `GeneralTree` besteht aus einer Wurzel `Root` und einer Sequenz von Unterbäumen, die auch leer sein kann.

Als Implementierung der Sequenzen werden sowohl verkettete Listen als auch Arrays unterstützt.

Objektreferenzen

Zur Repräsentation komplexer Sachverhalte werden zusätzlich zur Definition der eigentlichen Objekte Informationen über die Beziehungen der Objekte untereinander benötigt. Zu diesem Zweck wurden in IRL2 die Objektreferenzen neu eingeführt.

Eine Referenz in IRL2 ist ein Verweis auf ein durch eine IRL2-Struktur definiertes Objekt. Im einfachen Fall handelt es sich dabei um eine einseitig gerichtete Assoziation. Tritt eine zu referenzierende Struktur innerhalb einer Zwischensprache mehrfach als Attributtyp auf, so muss die Eindeutigkeit der Verweise gewährleistet sein. Dazu stehen in IRL2 *Scopes* zur Verfügung: Objektdefinitionen und Objektverweise können unterschiedlichen Sichtbarkeitsbereichen zugeordnet werden. Das erlaubt auch die Angabe von relativen und externen Verweisen.

Als Beispiel für relative, lokale Referenzen soll ein Ausschnitt aus der IRL2-Spezifikation von DSC dienen. Die Struktur `SpecificationTree` beschreibt hierarchisch die Systemgrößen. Eine mögliche Größe ist dabei ein Systemeingang, beschrieben durch die Struktur `Input`, die sich in einem Teilbaum der Struktur `Variables` befindet. Referenzen auf Objekte der Struktur `Input` werden durch Objekte vom Typ `U` dargestellt:

```
STRUCTURE SpecificationTree IS NEW SCOPE
  SystemName    : SEQ OF STRING;
  SrcClass      : SrcClassSpecification;
  Specification : Variables | Instantiation | IsExtern;
  Subsystems    : SEQ OF SpecificationTree;
END
```

```
STRUCTURE Variables IS
  TimeVariables : SEQ OF Time;
  Parameters    : SEQ OF Param;
  Inputs        : SEQ OF Input;
  Outputs       : SEQ OF Output;
  ExternFunc    : SEQ OF Func;
  States        : SEQ OF State;
  Auxiliars     : SEQ OF Aux;
END

STRUCTURE Input IS REFERENCED
  Name          : STRING;
  Unit          : STRING;
  MinValue      : SEQ OF REAL;
  MaxValue      : SEQ OF REAL;
  StartValue    : REAL;
  DebugInfo     : References | Nil;
END

U IS REFERENCE TO Input
END
```

Jedes Objekt der Struktur `SpecificationTree` repräsentiert die Schnittstelle eines Subsystems und erfordert somit auch einen eigenen Sichtbarkeitsbereich für Systemgrößen. Die Spezifikation eines neuen Sichtbarkeitsbereichs geschieht in IRL2 durch das Schlüsselwort `NEW SCOPE`. Jede Struktur, die referenzierbar sein soll, darf innerhalb eines Sichtbarkeitsbereichs höchstens einmal als Attributtyp verwendet werden, entweder in Form eines einfaches Strukturattributs, als Sequenz oder in Form einer Variante.

Zur Unterstützung von hierarchischen Strukturen und relativen Referenzen erlaubt IRL2 das Schachteln von Sichtbarkeitsbereichen, wobei jeder Sichtbarkeitsbereich aber für sich abgeschlossen ist, Objekte über- sowie untergeordneter Sichtbarkeitsbereiche also nicht direkt zugreifbar sind.

Die Darstellung der Objektreferenzen erfolgt in IRL2 durch Positionsangabe des zu referenzierenden Objekts. Eine Referenz auf eine Eingangsgröße eines Systems S hat dann in DSC z. B. folgende Form:

U < $s_1 s_2 \ldots s_m$ > l	mit	< $s_1 s_2 \ldots s_m$ >	Position des Systems S im Hierarchiebaum
		m	Systemebene von S
		l	Position der Eingangsgröße

Die Positionsnummerierung beginnt dabei jeweils bei 0. Die konkrete Codie-

rung einer Objektreferenz in IRL2 ergibt sich aus dem folgenden Algorithmus:

```
theReferenceCoding        = crEmptyList();
theCurrentStructure       = theReferencedStructure;
maybeEndOfRecursion       = True;
maybeBeginningOfRecursion = False;

while (getParentStructure(theCurrentStructure) != theRoot) {
  if (maybeEndOfRecursion) {
    if (isRecursiveStructure(theCurrentStructure)) {
      theReferenceCoding        = append(theReferenceCoding, '>');
      mayBeEndOfRecursion       = False;
      mayBeBeginningOfRecursion = True;
    }
  }

  if (isSequenceAttributeInParentStructure(theCurrentStructure)) {
    theReferenceCoding
     = append(getSequencePositionInAttributeOfParentStructure(
                theCurrentStructure),
              theReferenceCoding);
  }
  else if (isAlternativeAttributeInParentStructure(
             theCurrentStructure)) {
    theReferenceCoding
     = append(getStructureKind(theCurrentStructure),
              theReferenceCoding);
  }

  theParentStructure = getParentStructure(theCurrentStructure);
  if (maybeBeginningOfRecursion) {
    if (   isRecursiveStructure(theCurrentStructure)
        && getStructureKind(theCurrentStructure)
             != getStructureKind(theParentStructure) {
      theReferenceCoding        = append(theReferenceCoding, '<');
      mayBeEndOfRecursion       = True;
      mayBeBeginningOfRecursion = False;
    }
  }
  theCurrentStructure = theParentStructure;
}
```

Ausgangspunkt für die im obigen Algorithmus beschriebene Berechnung eines eindeutigen Pfades eines Objekts der Struktur S ist die Wurzel des kleinsten Teilbaums, der alle Objekte der Struktur S und alle Referenzen auf Objekte der Struktur S enthält. Die Struktur dieser Wurzel soll im Folgenden als *Wurzelstruktur* von S bezeichnet werden. Zu jeder referenzierbaren Struktur S kann die Wurzelstruktur von S statisch anhand der IRL2-Spezifikation bestimmt werden.

Die Bestimmung der Wurzelstrukturen kann z. B. durch den IRL2-Compiler erfolgen. Dazu wird für jede spezifizierte Struktur S eine Liste geführt, die jeweils alle Strukturen enthält, die Attribute vom Typ S enthalten. Zusätzlich wird für jedes Auftreten als Attribut die Tiefe eingetragen. Die Tiefe ergibt sich aus der minimalen Anzahl der Ebenen, die das Attribut vom Typ S unterhalb der Startstruktur auftritt. Eine analoge Liste wird für die Referenz auf Objekte der Struktur S geführt. Anhand dieser Listen kann nun die Wurzelstruktur von S bestimmt werden. Zunächst werden dazu alle in der Struktur- und der Referenzen-Liste eingetragenen übergeordneten Strukturen bis zur maximalen gemeinsamen Tiefe verfolgt. Anschließend sind die übergeordneten Strukturen Ebene für Ebene gemeinsam zu verfolgen. Sind die übergeordneten Strukturen aller Zweige identisch, so stellt diese übergeordnete Struktur die Wurzelstruktur von S dar.

Zur Umsetzung der in IRL2 spezifizierten Referenzen in eine C-Implementierung stehen zusätzlich zu den spezifizierten Attributen drei weitere Strukturkomponenten zur Verfügung, die je nach Notwendigkeit vom IRL2-Compiler automatisch eingefügt werden:

structureKind Diese Strukturkomponente gibt den Strukturtyp des aktuellen Objekts an. Sie ist erforderlich bei rekursiven Strukturen und deren direkt übergeordneten Strukturen sowie bei Strukturen, die als Attributvarianten auftreten.

parentRef Diese Strukturkomponente gibt das übergeordnete Objekt an, in dem das aktuelle Objekt als Attribut enthalten ist. Diese Strukturkomponente ist bei allen Strukturen erforderlich, die auf dem Pfad von einer referenzierbaren Struktur S zu der Wurzelstruktur von S liegen.

seqPos Diese Strukturkomponente gibt die Position des aktuellen Objekts in einer Sequenz an. Sie ist erforderlich, wenn die Struktur als Sequenz auftreten kann.

Diese drei zusätzlichen Strukturkomponenten der C-Realisierung einer in IRL2 spezifizierten Zwischensprache sind im Wesentlichen für die Ausgabe der Zwischensprache als ASCII-Notation erforderlich. Für das Einlesen der Zwischensprache als ASCII-Notation werden sie nicht benötigt. Die durch den IRL2-Compiler automatisch generierte C-Realisierung setzt alle Referenzen in effiziente C-Pointer-Konstrukte um. Ein Beispiel der Umsetzung einer IRL2-Spezifikation in eine C-Datenstruktur ist in Abbildung 4.10 dargestellt.

Neben der vorgestellten Codierung einer Referenz durch Positionsindizes ist auch eine Referenzierung über Schlüssel möglich. Als Schlüssel kann dabei ein beliebiges Attribut einer Struktur dienen. Hierbei kann ein Referenztyp für Objekte unterschiedlicher Strukturen verwendet werden. Für jede Struktur ist dann der entsprechende Schlüssel anzugeben, wobei die Schlüssel alle

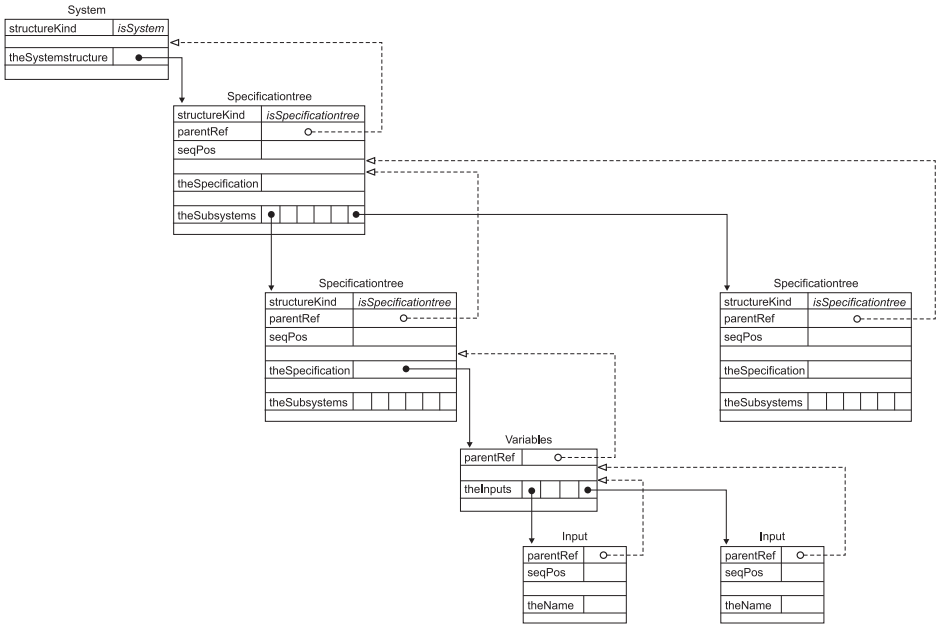


Abbildung 4.10: Referenzen in IRL2

denselben Strukturtyp haben müssen. Sollen Schlüssel zur Codierung von Referenzen verwendet werden, so ist neben der Menge der zu referenzierenden Strukturen die Menge der zugehörigen Schlüssel anzugeben. Als Beispiel ist die IRL2-Spezifikation von Referenzen auf externe Systemgrößen in DSC angegeben:

```
E IS REFERENCE TO {Param, Input, Output, State} IN SCOPE DSC
  KEY {Param.Name, Input.Name, Output.Name, State.Name}
END
```

Neben den bisher vorgestellten lokalen und relativen Referenzen unterstützt IRL2 auch die Referenzierung von externen Objekten. Durch die explizite Bezugnahme auf einen anderen Sichtbarkeitsbereich (*Scope*) können externe Objekte referenziert werden. Erfolgt eine Zusammenführung der Objekte in einem gemeinsamen Adressraum, so stehen Mechanismen zur Umwandlung in lokale Referenzen zur Verfügung. Ein Beispiel für die Anwendung externer Referenzen in DSC ist der *MPONameServer*, bei dem sich alle MPOs registrieren müssen:

```
REPRESENTED STRUCTURE MPONameServer IS NEW SCOPE
  RegisteredMPOs : SEQ OF DSC;
END

MPO IS REFERENCE TO DSC IN SCOPE MPONameServer
  KEY DSC.SystemName
```

```
END
```

Der *MPONameServer* enthält eine Liste der DSC-Darstellungen aller registrierten MPOs. Ein MPO selbst ist als Referenz auf eine im *MPONameServer* abgelegte DSC-Darstellung spezifiziert. Als eindeutiger Schlüssel zur Zuordnung dient dabei der Systemname. Als Bezugsbereich für den Schlüssel ist der *MPONameServer* explizit angegeben.

Bei den bisher vorgestellten Objektreferenzen handelt es sich um unidirektionale Verweise. Insbesondere wird vorausgesetzt, dass ein Objekt definiert worden sein muss, bevor es referenziert werden kann. Ein vorwärts gerichtetes Verwenden einer Referenz vor der eigentlichen Definition des referenzierten Objekts ist — mit Ausnahme von externen Objekten — so nicht möglich.

Zur Lösung dieser Problematik bietet IRL2 zusätzlich *bidirektionale Referenzen* an. Soll eine Referenz bidirektionalen Charakter haben, so ist bei der Definition der Strukturreferenz zusätzlich das Schlüsselwort `BIDIRECTIONAL` anzugeben. Ein Beispiel in DSC für eine bidirektionale Verlinkung von Objekten sind die Strukturen `SpecificationTree` und `CoupleTree`:

```
STRUCTURE SpecificationTree IS NEW SCOPE
  SystemName    : SEQ OF STRING;
  SrcClass      : SrcClassSpecification;
  Specification : Variables | Instantiation | IsExtern;
  Subsystems    : SEQ OF SpecificationTree;
END

SysSpecBiRef IS BIDIRECTIONAL REFERENCE TO SpecificationTree
END

STRUCTURE CoupleTree IS
  Couplings         : SEQ OF Assign;
  CoupledSubsystems : SEQ OF CoupleTree;
  SysSpec           : SysSpecBiRef;
END
```

Im obigen Beispiel wird die Struktur `SysSpecBiRef` als bidirektionale Referenz auf Objekte der Struktur `SpecificationTree` spezifiziert. Soll nun die Struktur `CoupleTree` eine bidirektionale Verlinkung mit der Struktur `Specificationtree` eingehen, so muss `CoupleTree` ein Attribut der bidirektionalen Strukturreferenz `SysSpecBiRef` enthalten, das explizit definiert werden muss. Für die Struktur `SpecificationTree` darf kein entsprechendes Referenzattribut in der IRL2-Spezifikation angegeben werden, es wird implizit vom IRL2-Compiler angelegt. Beim Setzen des Attributs `SysSpec` der Struktur `CoupleTree` wird dann in der entsprechenden *set*-Methode

automatisch der entsprechende Verweis in dem referenzierten Objekt der Struktur `SpecificationTree` gesetzt.

Bei der Spezifikation einer bidirektionalen Verlinkung zwischen zwei Strukturen ist darauf zu achten, dass die explizite Attributdefinition der bidirektionalen Referenz stets bei der Struktur geschieht, die bei einer Tiefensuche durch die aufgebauten Datenstrukturen erst an zweiter Stelle folgen würde. So ist beim Erstellen der bidirektionalen Referenz jederzeit gewährleistet, dass die andere Struktur bereits existiert und ihr implizit vorhandenes Referenzattribut korrekt besetzt werden kann.

Kopien und Instanzen

Bei einem mechatronischen System treten häufig Komponenten desselben Typs mehrfach auf. Die effiziente Abbildung von Systemtypen und Instanzen ist daher eine wichtige Anforderung an eine verarbeitungsorientierte Modellrepräsentation.

Zur effizienten Unterstützung von Kopien und Instanzen bietet IRL2 einen intelligenten Kopiermechanismus an. Für jedes Attribut, das instanzspezifische Werte enthält, wird an der Quellstruktur ein Array angelegt. Über einen Instanzzähler wird dann auf den zur jeweiligen Instanz gehörenden Wert zugegriffen. Die Kennzeichnung der instanzspezifischen Attribute erfolgt durch das Schlüsselwort `COPY OF`.

Dieser Mechanismus erfordert, dass auf jede Struktur über ein Adressen-Tupel, bestehend aus der Quellstruktur und dem Instanzindex, zugegriffen werden muss. Dies gilt sowohl für alle benutzerspezifizierten Attribute als auch für die implizit angelegten Attribute wie z. B. `parentRef`.
Ein Beispiel für die rechnerinterne Darstellung von Kopien und Instanzen in IRL2 zeigt Abbildung 4.11. Schematisch dargestellt sind drei Strukturen *a*, *b* und *c*. Jede der Strukturen besitzt ein Attribut `commonAttr`, das bei einer möglichen Kopie nicht dupliziert werden soll, d. h. nicht instanzspezifisch ist. Des Weiteren besitzt jede Struktur ein Attribut `copiedAttr`, das beim Anfertigen einer Strukturkopie dupliziert werden soll, also instanzspezifisch ist.

Bei der Struktur *a* handelt es sich um eine Struktur ohne Kopien. Für jedes Attribut gibt es nur einen Eintrag. Das Attribut `commonAttr` von *a* ist eine Instanz der Struktur *c* mit dem Instanzindex 2, dargestellt durch einen Zeiger mit dem annotierten Instanzindex 2. Alle für *a.commonAttr* instanzspezifischen Werte finden sich in den entsprechenden Instanzarrays an der Position 2. An dieser Position ist auch in der generierten Strukturkomponente `parentRef` der Verweis auf *a* als das übergeordnete Objekt zu *a.commonAttr* abgelegt. Der Instanzindex an diesem Zeiger ist 0, da er auf eine Quellstruktur und nicht auf eine Kopie verweist.

Bei *b* handelt es sich um eine Struktur, zu der drei weitere Kopien existieren. Bei der dritten Kopie von *b* ist der Wert des instanzspezifischen Attri-

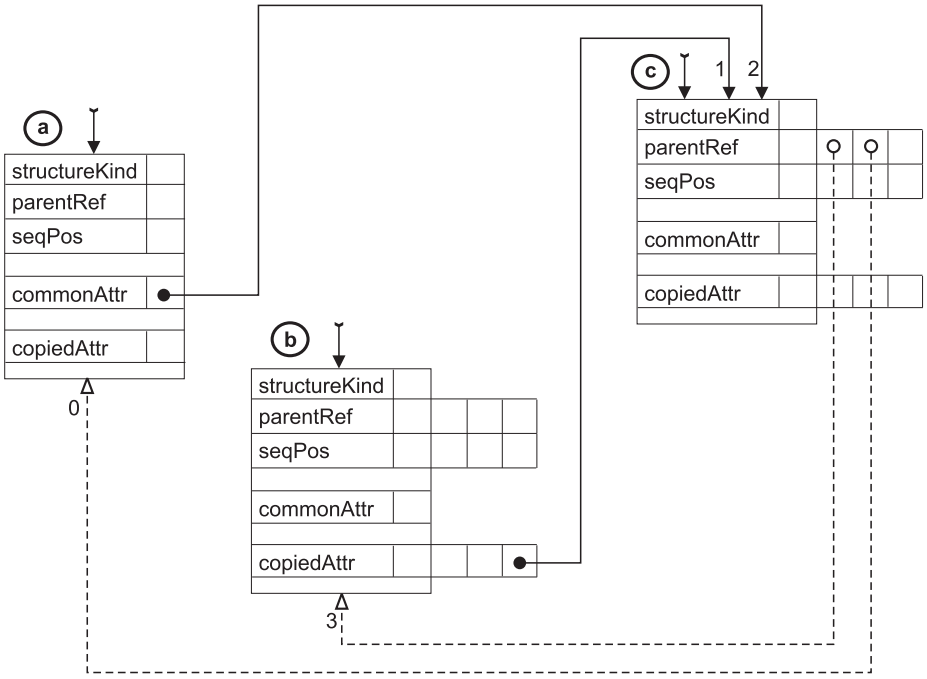


Abbildung 4.11: Kopien in IRL2

buts `copiedAttr` eine Kopie von *c* mit Instanzindex 1. Bei der Strukturkomponente `parentRef` muss damit an Position 1 ein Zeiger auf die Struktur *b* eingetragen werden. Als Instanzindex muss 3 angegeben werden, da die erste Kopie von *c* ein instanzspezifisches Attribut der dritten Instanz von *b* ist.

Für jede spezifizierte Struktur werden vom IRL2-Compiler entsprechende Kopierfunktionen automatisch erzeugt. Zusätzlich werden *DeepCopy*-Funktionen erzeugt, die eine Struktur einschließlich aller Unterstrukturen vollständig kopieren.

IRL2 im Einsatz: Der Bindevorgang eines hierarchischen MPO

Als Beispiel für den Einsatz von IRL2 zur Realisierung der Mechatronic Processing Objects soll die Implementierung der Methode *bind* näher betrachtet werden. Diese Methode soll dafür sorgen, dass für das aktuelle MPO alle Referenzen zu externen MPOs aufgelöst werden und diese wie interne Subsysteme in die Datenstruktur integriert werden, um ein homogenes Gesamtsystem zu erhalten.

Ist ein Subsystem eine Instanz eines externen Systemtyps, so sind die folgenden Objekte des Systemtyps instanzspezifisch:

- Initialwerte der Systemgrößen

- Verkopplung auf Ebene der Basissysteme
- Auswertereihenfolge der Systemgleichungen

Die IRL2-Spezifikation dieser Datenstrukturen ist im Folgenden auszugsweise wiedergegeben:

```
STRUCTURE Input IS REFERENCED
  Name       : STRING;
  Unit       : STRING;
  MinValue   : SEQ OF REAL;
  MaxValue   : SEQ OF REAL;
  StartValue : COPY OF REAL;
  DebugInfo  : References | Nil;
END

STRUCTURE Couples IS
  Hierarchy           : CoupleTree;
  BasicBlockCouplings : COPY OF SEQ OF Assign;
END

STRUCTURE HierarchicalClusterNode IS
  ...
  DirectLinkEvalOrder : COPY OF SEQ OF ClusterDirectLinkPart;
  ...
END
```

Die Struktur `INPUT` dient als Beispiel für die Spezifikation einer Systemgröße. Die Komponente `StartValue` ist durch das Schlüsselwort `COPY OF` als instanzspezifisches Attribut definiert. Die Werte aller übrigen Attribute sind für alle Instanzen des Systemtyps identisch und müssen daher nur einmal pro Systemtyp abgelegt werden.

Systemkopplungen werden in der Struktur `Couples` beschrieben. Die hierarchische Verkopplung, repräsentiert durch das Attribut `Hierarchy`, ist dabei pro Systemtyp definiert, muss also nicht instanzspezifisch abgelegt werden. Die Verkopplung auf Ebene der Basissysteme ist jedoch vom Kontext des Gesamtsystems abhängig, sie muss daher instanzspezifisch abgelegt werden. Dazu dient das Attribut `BasicBlockCouplings`.

Die Auswertereihenfolge der Systemgleichungen muss ebenfalls instanzspezifisch ermittelt werden. Zur Ablage dieser Informationen dient das Attribut `DirectLinkEvalOrder` der Struktur `HierarchicalClusterNode`. Damit kann für jeden Cluster eine individuelle Auswertereihenfolge festgelegt werden.

Die Umsetzung des eigentlichen Bindevorgangs durch den Einsatz von IRL2 soll nun im Folgenden in Auszügen vorgestellt werden. Als Grundlage dient dabei das in Abbildung 4.6 vorgestellte Szenario: Gegeben sei ein

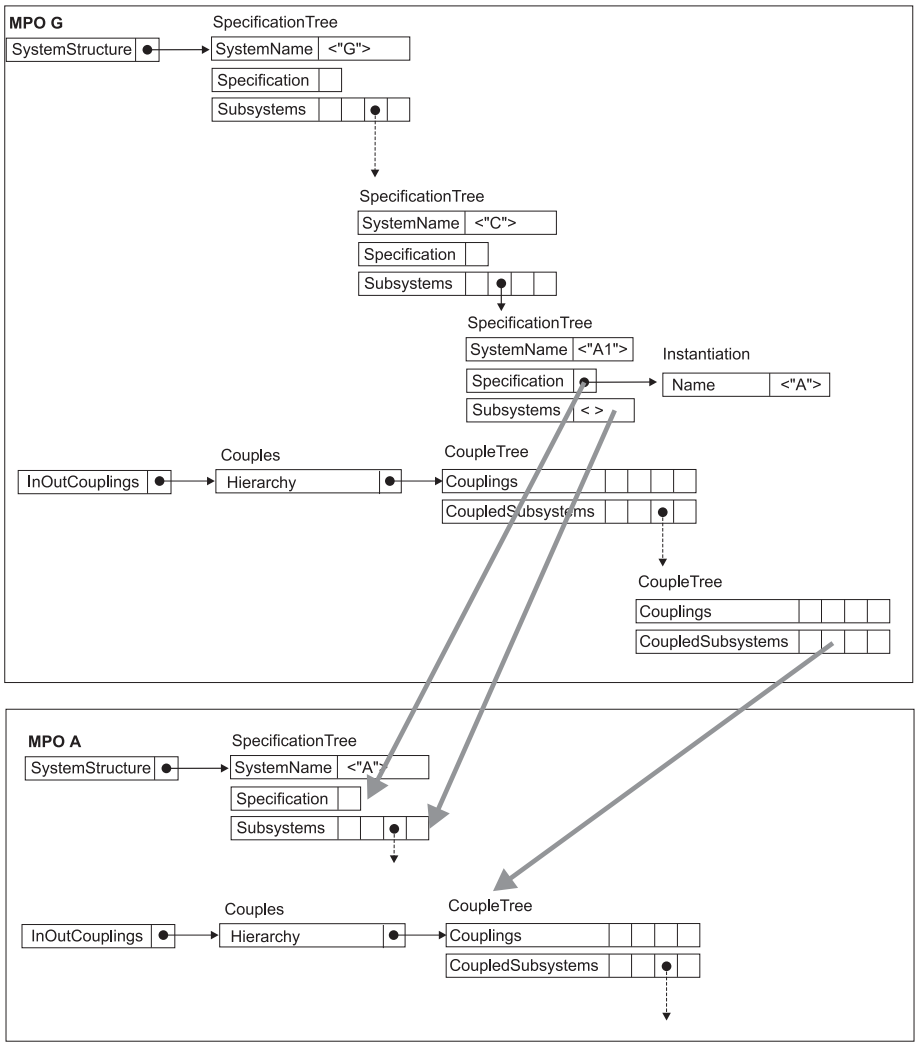


Abbildung 4.12: Bindevorgang eines hierarchischen MPO

hierarchisches MPO G, das unter anderem eine Komponente $A1$ als Instanz eines Systemtyps A enthalte. Die entsprechenden DSC-Datenstrukturen der MPOs G und A zu diesem Szenario sind auszugsweise in Abbildung 4.12 dargestellt. Die Komponente $A1$ sei dabei ein direktes Subsystem des hierarchisch höherwertigen Teilsystems C. Die eigentliche Aufgabe des Bindevorgangs, die Einbindung von Instanzen extern definierter Systemtypen, kann dann nach folgendem Algorithmus erfolgen:

```
if (getStructureKind(A1_SpecTree.Specification)
     == isInstantiation) {
  ### A1 ist Instanz eines Systemtyps ###
  Instantiation := A1_SpecTree.Specification;
```

```
  # Erzeugen einer Instanz
  A  := getRegisteredMPO(NameServer, Instantiation.Name);
  A1 := copyDSC(A);

  # Einhängen des SpecificationTree von A1
  #  als i-tes Subsystem von C
  C_SpecTree.Subsystems[i] := A1.SystemStructure;

  # Einhängen der hierarchischen Kopplungen
  A1_CoupleTree := A1.InOutCouples.CoupleTree;
  C_CoupleTree  := C_SpecTree.SysSpecBiRef;
  C_CoupleTree.CoupledSubsystems[i] := A1_CoupleTree;
}
```

Der obige Algorithmus ist rekursiv in Form einer Tiefensuche auf alle Objekte der Struktur `SpecificationTree` in G anzuwenden. `A1_SpecTree` gibt dabei jeweils die Struktur des aktuell betrachteten Teilsystems an, `C_SpecTree` die Struktur des direkt übergeordneten Subsystems. Der Index `i` gibt jeweils eine Ordnung der Unterkomponenten des gekoppelten Subsystems C an.

4.4.2 Abbildung von Nebenläufigkeit

Ein zentrales Anliegen der Mechatronic Processing Objects ist die Unterstützung verteilter Systeme. Für eine effiziente Unterstützung der Parallelverarbeitung ist eine hohe Skalierbarkeit erforderlich. Es muss ohne großen Konfigurationsaufwand möglich sein, eine verteilte Anwendung unverändert sowohl auf einem Einprozessorrechner, auf einem Multiprozessorsystem mit gemeinsamem Speicher als auch auf einem Rechnernetzwerk ausführen zu können.

Zur Erreichung einer hohen Skalierbarkeit und problemlosen Portierbarkeit von verteilten Anwendungen stellen die Mechatronic Processing Objects und der MPONameServer ihre Funktionalität zunächst in Form von Funktionen mit gemeinsam genutzten Daten bereit. In Abhängigkeit von der Rechnerarchitektur stehen dann entsprechende Frameworks zu Verfügung, die die einzelnen Objekte dann z. B. in dynamisch ladbare Bibliotheken, in Threads oder in Betriebssystemprozesse einbetten.

Einen prinzipieller Überblick über verteilte Implementierungen findet sich in Abbildung 4.13. Hat man nur einen Einprozessorrechner ohne Multitasking zur Verfügung, so wird das MPO des Gesamtsystems G durch eine ausführbare Datei repräsentiert, die den MPONameServer sowie weitere eingebundene MPOs in Form von dynamisch ladbaren Bibliotheken enthält. Auf einem Multiprozessorsystem mit gemeinsamem Speicher wird

jedes MPO in einem leichtgewichtigen Prozess (Thread) gekapselt. Die Kommunikation der einzelnen MPOs geschieht weiterhin sehr effizient über den gemeinsamen Speicher. Bei der Implementierung auf einem Rechnernetzwerk kann jedes MPO in Form einer ausführbaren Datei auftreten, die über die Ein- und Ausgangsgrößen mit Hilfe von Nachrichten mit den übrigen MPOs Werte austauscht.

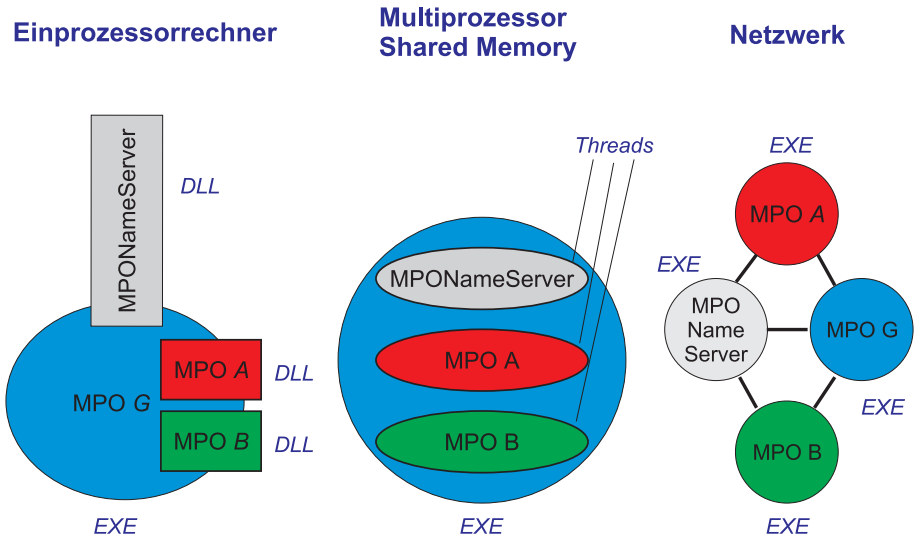


Abbildung 4.13: Abbildung von Nebenläufigkeit

Kapitel 5

Simulation

Bei dem rechnergestützten Entwurf eines mechatronischen Systems spielt die Simulation des zu entwerfenden Systems eine zentrale Rolle. So wird z. B. während der Modellbildung durch den Einsatz der nichtlinearen Simulation das modellierte Systemverhalten überprüft. Beim späteren, schrittweisen Übergang vom vollständig im Rechner abgelegten Modell zum realen Prototypen werden Hardware-in-the-Loop-Simulationen durchgeführt. Dabei existieren einige Komponenten des mechatronischen Systems als reale Hardware, alle übrigen Komponenten werden auf leistungsfähiger Echtzeithardware simuliert.

Das vorliegende Kapitel beschäftigt sich mit dem Einsatz der Mechatronic Processing Objects bei der Simulation mechatronischer Systeme. Dabei wird zunächst ein Simulationskonzept für hybride Systeme vorgestellt, das sowohl den Anforderungen einer Offline- als auch einer Echtzeit-Simulation genügt.

Um den Verlauf einer Simulation fortschreiben zu können, muss das Zeitverhalten des modellierten Systems berechnet werden. Dabei gibt es zwei Alternativen zur Auswertung des Zeitverhaltens. Diesen beiden Alternativen der interpretierenden Modellauswertung und der Codeerzeugung sind die folgenden zwei Unterkapitel gewidmet. Zum Abschluss des Kapitels wird die Simulationsumgebung SIMBA präsentiert, in der die zuvor vorgestellten Ansätze umgesetzt wurden.

5.1 Konzept

Um eine nichtlineare Simulation effizient sowohl offline als auch unter Echtzeitbedingungen für mechatronische Systeme einsetzen zu können, muss der zugrundeliegende Simulationsalgorithmus die folgenden zwei Hauptaufgaben erfüllen:

1. Berechnung des kontinuierlichen Systemverhaltens unter Minimierung der Totzeit

Für eine erfolgreiche Simulation mechatronischer Systeme unter Echtzeitbedingungen ist es von entscheidender Bedeutung, dass insbesondere bei Teilsystemen mit direkt auf den Ausgang durchgreifenden Eingangsgrößen die Totzeit zwischen dem Einlesen der Sensorgrößen und dem Ausgeben der Aktorgrößen möglichst gering gehalten wird. Anderenfalls kommt es zu vermeidbaren, zusätzlichen Phasenverschiebungen, die die Stabilität des Systems unnötig gefährden.

2. Schnelle Reaktion auf das Auftreten von diskreten Ereignissen
   Mit zunehmender Intelligenz mechatronischer Systeme steigt der Anteil der diskreten, ereignisgesteuerten Systemanteile. Hier muss der Simulationsalgorithmus Mechanismen zur Ereignisverarbeitung bereitstellen, die eine kurze Reaktionszeit gewährleisten. Andererseits muss aber auch darauf geachtet werden, dass der Algorithmus zur Ereignisdetektion die Berechnungen des kontinuierlichen Systemverhaltens möglichst wenig verzögert.

In den folgenden Abschnitten wird zunächst auf die Simulation des kontinuierlichen Systemverhaltens eingegangen. Aufgrund der hohen Bedeutung der Totzeitminimierung wird zunächst eine Klassifikation der Ein- und Ausgangsgrößen nach ihrem Durchgriffsverhalten vorgenommen. Anschließend wird der allgemeine Ablauf eines kontinuierlichen Simulationsschritts vorgestellt. Ein weiterer Abschnitt ist der Ermittlung und Festlegung der Auswertereihenfolge von verkoppelten Systemen gewidmet. Danach wird eine allgemeine Schnittstelle vorgestellt, mit der sich beliebige numerische Integrationsverfahren in das Simulationskonzept einbinden lassen. Der letzte Abschnitt beschäftigt sich dann mit der Einbettung der Verarbeitung von diskreten Ereignissen.

5.1.1 Klassifikation von Ein- und Ausgangsgrößen

Ein Mechatronic Processing Object als verarbeitungsorientierte Ausprägung einer blockorientierten Modellierungsform beschreibt das Systemverhalten in Form von gerichteten Ein-/Ausgangsbeziehungen. Bei der Systemsimulation ist dafür zu sorgen, dass keine algebraischen Schleifen auftreten. Die gerichteten Beziehungen über alle Systemverkopplungen hinweg müssen zyklusfrei sein, anderenfalls gibt es keine korrekte Auswertereihenfolge der Systemgleichungen. Je nach Verkopplung der einzelnen Teilsysteme ergeben sich unterschiedliche Abhängigkeiten und damit auch jeweils andere Auswertereihenfolgen. Um die vollständige Modularität der Repräsentation eines Teilsystems zu erhalten, darf die Auswertereihenfolge nicht starr codiert sein, sondern sie muss in Abhängigkeit von der Struktur des Gesamtsystems anpassbar sein.

Um die Anforderungen einer modularen, verkopplungssensitiven, aber dennoch laufzeiteffizienten Simulationsumgebung zu erfüllen, unterteilt ein

MPO seine kontinuierlichen Ein- und Ausgangsgrößen in drei Kategorien:

Nichtdurchgriffsgrößen (u_{ND}, y_{ND}) Zu dieser Kategorie zählen alle Ausgangsgrößen, die nicht direkt von Eingangsgrößen abhängen, das heisst, ihr Wert zu einem Zeitpunkt t lässt sich allein aus den Zustandswerten zum Zeitpunkt t ermitteln. Eine Veränderung der Eingangsgrößen greift hier nicht direkt auf den Systemausgang durch. Die zugehörigen Gleichungen können zu Beginn eines jeden Simulationsschritts sofort ausgewertet und die Nichtdurchgriffsausgänge unmittelbar ausgegeben werden. Der diesen Größen zugeordnete Teil eines MPO ist für die Gewährleistung einer korrekten Auswertereihenfolge unproblematisch.

Durchgriffsgrößen (u_D, y_D) Zu dieser Kategorie gehören alle Eingangsgrößen, die ohne Zeitverzug direkt auf mindestens eine Ausgangsgröße wirken. Die jeweiligen Ausgangsgrößen sind ebenfalls hier einzuordnen. Für die korrekte Abarbeitung der zugehörigen Gleichungen muss eine Auswertereihenfolge ermittelt werden. Dazu stößt ein MPO die Methode *calcEvaluationOrder* an, über die miteinander vernetzten MPOs wird eine verteilte, umgekehrte topologische Sortierung gestartet. Pro MPO wird dabei der gesamte den Durchgriffsgrößen zugeordnete Code als ein Block betrachtet. Die Durchgriffsgrößen des MPO werden also pro Auswertung einmal als ganzer Vektor eingelesen bzw. ausgegeben. Wie die Implementierung von DSL [Sch94] zeigt, lässt sich mit der Zweiteilung des Codes zur Bestimmung der Ausgangsgrößen in einen Durchgriffs- und einen Nichtdurchgriffsteil bei einer Vielzahl von Systemen eine korrekte Auswertereihenfolge ermitteln.

White-box-Größen (u_W, y_W) Repräsentiert ein MPO jedoch ein komplexes Teilsystem mit vielen Subsystemen, so lässt sich mit der vektoriellen Zusammenfassung aller Durchgriffseingänge bzw. -ausgänge pro MPO keine Auswertereihenfolge mehr ermitteln. Für solche Fälle bietet ein MPO die White-box-Größen an. Diese Größen können jederzeit „zwischendurch“ kommunizieren. Für problematische Durchgriffsgrößen kann ein Wechsel in diese Kategorie vorgenommen werden, so dass sich spätestens auf dieser Ebene eine Auswertereihenfolge finden lässt.

Zu jeder der oben aufgeführten Kategorien existiert innerhalb eines MPO ein separater Codeblock, in dem die zugehörigen Ausgangsgrößen berechnet werden.

5.1.2 Ablauf eines Simulationsschritts

Wurde im vorangegangen Abschnitt die Klassifikation der kontinuierlichen Ein- und Ausgangsgrößen eines MPO vorgenommen, so sollen nun die dazugehörigen Codeblöcke vorgestellt werden, in denen die einzelnen System-

größen berechnet werden. Die Berechnung der Zustands- und Ausgangsgrößen eines MPO ist in die folgenden drei Codeblöcke unterteilt:

Nichtdurchgriffscode (ND-Code) Der Nichtdurchgriffscode dient zur Ermittlung der Nichtdurchgriffsausgänge $\underline{y}_{ND}$ und der dazu erforderlichen Hilfsgrößen. Zur Berechnung stehen dabei die aktuellen Zustandsgrößen $\underline{x}$ sowie daraus abgeleitete Hilfsgrößen zur Verfügung.

Durchgriffscode (D-Code) Der Durchgriffscode dient zur Ermittlung der Durchgriffsausgänge $\underline{y}_D$ und der dazu erforderlichen Hilfsgrößen. Dabei können Hilfsgrößen, deren Wert bereits im ND-Code berechnet wurde, unverändert übernommen werden. Der D-Code eines Teilsystems kann erst abgearbeitet werden, nachdem der Wert der zugehörigen Durchgriffseingänge $\underline{u}_D$ berechnet wurde. Wird während eines Simulationsschritts zuerst der gesamte ND-Code gerechnet und dann der D-Code in der entsprechenden Reihenfolge, so ist immer eine korrekte Auswertung gewährleistet.

Zustandscode (S-Code) Der Zustandscode dient zur Bestimmung der aktuellen Zustandsableitungen. Die Berechnung der einzelnen Ableitungen kann erfolgen, sobald alle Eingänge, von denen die Ableitung abhängt, mit dem dem aktuellen Zeitschritt entsprechenden Wert vorliegen. Dies ist immer gewährleistet, wenn zuvor der gesamte ND- und D-Code abgearbeitet wurde. Im Zustandscode müssen nur noch die Hilfsgrößen neu berechnet werden, die nicht bereits im ND- oder D-Code berechnet worden sind.

Zum Zweck einer weitestmöglichen Trennung der Fortschreibung des inneren Systemzustands von der Berechnung des Ausgangsverhaltens stellt ein MPO für jeden dieser Codeblöcke jeweils zwei Methoden zur Verfügung:

eval*Codeblock* Diese Methode dient zur Berechnung des inneren Systemverhaltens. Hier werden alle Systemgleichungen zusammengefasst, die zur Fortschreibung des inneren Systemzustands erforderlich sind. Sind bei einzelnen Auswertungen des Systemverhaltens die Werte der Ausgangsgrößen nicht relevant, so genügt ein ausschließlicher Aufruf dieser Methode.

outputs*Codeblock* Diese Methode dient zur Berechnung der Ausgangsgrößen des Codeblocks. Handelt es sich bei dem MPO um ein Basissystem, so enthält diese Methode die Berechnung der eigentlichen Ausgangsgleichungen. Des Weiteren werden hier auch die Hilfsgrößen berechnet, die ausschließlich für die Berechnung der Ausgangsgrößen dieses Codeblocks benötigt werden. Bei einem hierarchischen System enthält diese Methode die Auswertung der Ausgangskoppelgleichungen.

Methode	Variablen, die in der Methode berechnet werden
evalND	$\mathcal{A}_{DX} \cap \mathcal{A}_{\mathcal{Y}_{ND}}$
evalD	$(\mathcal{A}_{DX} \cap \mathcal{A}_{\mathcal{Y}_D}) \setminus \mathcal{A}_{evalND}$
evalS	$\mathcal{A}_{DX} \setminus (\mathcal{A}_{evalND} \cup \mathcal{A}_{evalD})$
outputsND	$(\mathcal{A}_{\mathcal{Y}_{ND}} \setminus \mathcal{A}_{evalND}) \cup \mathcal{Y}_{ND}$
outputsD	$(\mathcal{A}_{\mathcal{Y}_D} \setminus (\mathcal{A}_{evalND} \cup \mathcal{A}_{evalD})) \cup \mathcal{Y}_D$

Definitionen:

$\mathcal{Y}_{ND}$	Menge aller Variablen, die Nichtdurchgriffsausgänge repräsentieren
$\mathcal{Y}_D$	Menge aller Variablen, die Nichtdurchgriffsausgänge repräsentieren
$\mathcal{A}_{\mathcal{Y}_{ND}}$	Menge aller Hilfsvariablen, die zur Berechnung von $\mathcal{Y}_{ND}$ erforderlich sind
$\mathcal{A}_{\mathcal{Y}_D}$	Menge aller Hilfsvariablen, die zur Berechnung von $\mathcal{Y}_D$ erforderlich sind
$\mathcal{A}_{DX}$	Menge aller Hilfsvariablen, die zur Berechnung der Zustandsableitungen $\underline{\dot{x}}$ erforderlich sind

Tabelle 5.1: Zuordnung der Berechnungsgleichungen

Eine detaillierte Zuordnung der Berechnungsgleichungen zu den einzelnen Methoden ist in Tabelle 5.1 dargestellt. Mit Hilfe dieser Methoden ergibt sich für die kontinuierliche Simulation eines MPO der folgende Algorithmus:

1. Initialisierungsschritt vor dem Zeitpunkt t_0:
   (a) *evalND(t_0)*
   (b) *outputsND(t_0)*
   (c) Ausgabe von $\underline{y}_{ND}(t_0)$
2. Kontinuierlicher Zeitschritt zum Zeitpunkt t_k:
   (a) Einlesen von $\underline{u}_D(t_k)$
   (b) *evalD(t_k)*
   (c) *outputsD(t_k)*
   (d) Ausgabe von $\underline{y}_D(t_k)$
   (e) Einlesen von $\underline{u}_{ND}(t_k)$
   (f) *evalS(t_k)*
   (g) Berechnung des Nachfolgezustands $\underline{x}_{k+1}$ durch Integration
   (h) *evalND(t_{k+1})*
   (i) *outputsND(t_{k+1})*

(j) Ausgabe von $\underline{y}_{ND}(t_{k+1})$

Zur Minimierung der Totzeit wird der bereits in [Eng95] beschriebene Ansatz verwendet, die rein zustandsabhängigen Ausgangsgrößen $\underline{y}_{ND}$ schon am Ende des vorangegangenen Simulationsschritts aus den dann ja bereits vorliegenden Zustandsgrößen zu berechnen. Damit sind zum Zeitpunkt t_k die aktuellen Werte der Ausgangsgrößen $\underline{y}_{ND}(t_k)$ sofort verfügbar. Die Ermittlung der Werte für den Startzeitpunkt t_0 erfolgt bereits in der Initialisierungsphase.

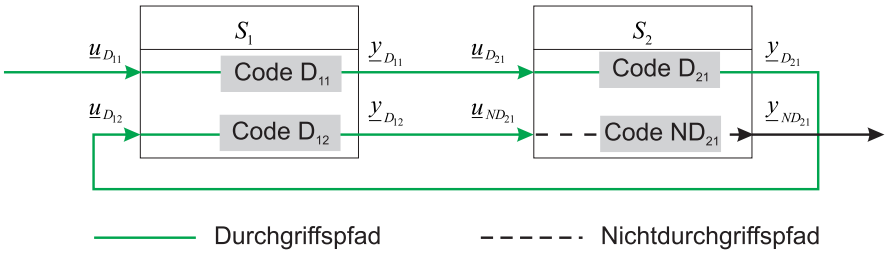
5.1.3 Ermittlung und Festlegung der Auswertereihenfolge

Werden mehrere Teilsysteme gekoppelt, so muss bei der Simulation des Gesamtsystems besondere Rücksicht auf die Reihenfolge der Berechnung der einzelnen D-Code-Anteile genommen werden. Um ein möglichst stabiles und genaues Simulationsverhalten zu erzielen, ist es erforderlich, dass zu jedem Zeitpunkt t_k für die Berechnung der Durchgriffsausgänge $\underline{y}_D(t_k)$ die aktuellen Werte aller dafür benötigten Durchgriffseingänge $\underline{u}_D(t_k)$ vorliegen.

Zur Veranschaulichung dieser Reihenfolgeproblematik ist in Abbildung 5.1 ein Beispiel eines gekoppelten Systems mit Durchgriffspfad zu sehen. Unter einem Durchgriffspfad ist hier ein Ein-/Ausgangspaar $(\underline{u}_{D_{mi}}, \underline{y}_{D_{mj}})$ eines Subsystems S_m zu verstehen, bei dem zur Berechnung von $\underline{y}_{D_{mj}}(t_k)$ der Wert von $\underline{u}_{D_{mi}}(t_k)$ benötigt wird. Im Beispiel ist der D-Code des Subsystems S_1 zweigeteilt in die Codeblöcke D_{11} und D_{12}. Als explizite Berechnungsvorschrift zur Ermittlung des Systemverhaltens ergibt sich in diesem Fall der angegebene Algorithmus.

Wählt man den Ansatz, nur einen D-Code-Block pro Subsystem zu verwenden, so ergibt sich bei dem gleichen System die in Abbildung 5.2 dargestellte Situation. Der Codeblock D_1 kann erst ausgeführt werden, wenn der vollständige Eingangsvektor $\underline{u}_{D_1}$ anliegt. Für die Ermittlung von $\underline{u}_{D_1}$ muss jedoch vorher der Codeblock D_{21} ausgeführt worden sein, der wiederum von der Ausführung vom Codeblock D_1 abhängig ist. Damit ergibt sich eine algebraische Schleife, die nur durch implizite Gleichungslöser aufgelöst werden kann. Aufgrund des hohen Rechenaufwands sind implizite Verfahren jedoch für den Einsatz unter Echtzeitbedingungen nicht geeignet.

Wie die aufgezeigten Beispiele zeigen, haben die unterschiedlichen Möglichkeiten der Zusammenfassung von D-Code-Anteilen Einfluss auf die Ermittlung einer korrekten Auswertereihenfolge der Systemgleichungen. Im Fall höchster Granularität besteht ein D-Code-Block nur aus der Berechnung einer einzigen skalaren Ausgangsgröße. Die Ermittlung der Auswertereihenfolge kann dann auf Gleichungsebene durchgeführt werden. Im anderen Extremfall wird der D-Code eines gesamten Subsystems zu einem einzigen Codeblock zusammengefasst. Die Auswertereihenfolge wird hier auf Blockebene ermittelt. Ein Block wird als untrennbare Einheit betrachtet, die Ausgänge



Auswertealgorithmus für den k-ten Zeitschritt t_k:

1. $\underline{y}_{ND_{21},k} = g_{ND_{21}}(\underline{x}_{2,k}, t_k)$
2. (a) $\underline{y}_{D_{11},k} = g_{D_{11}}(\underline{x}_{1,k}, \underline{u}_{D_{11},k}, t_k)$
   $\underline{u}_{D_{21},k} = \underline{y}_{D_{11},k}$
   (b) i. $\underline{y}_{D_{21},k} = g_{D_{21}}(\underline{x}_{1,k}, \underline{u}_{D_{21},k}, t_k)$
   $\underline{u}_{D_{12},k} = \underline{y}_{D_{21},k}$
   ii. $\underline{y}_{D_{12},k} = g_{D_{12}}(\underline{x}_{1,k}, \underline{u}_{D_{12},k}, t_k)$
   $\underline{u}_{D_{22},k} = \underline{y}_{D_{11},k}$
3. (a) $\dot{\underline{x}}_{1,k} = \underline{f}_1(\underline{x}_{1,k}, \underline{u}_{1,k}, t_k)$
   $\dot{\underline{x}}_{2,k} = \underline{f}_2(\underline{x}_{2,k}, \underline{u}_{2,k}, t_k)$
   (b) $\underline{x}_{1,k+1} = \underline{x}_{1,k} + \underline{h}_1(\dot{\underline{x}}_{1,k+1}, \underline{x}_{1,k}, \dot{\underline{x}}_{1,k}, \underline{x}_{1,k-1}, \dot{\underline{x}}_{1,k-1}, \ldots)$
   $\underline{x}_{2,k+1} = \underline{x}_{2,k} + \underline{h}_2(\dot{\underline{x}}_{2,k+1}, \underline{x}_{2,k}, \dot{\underline{x}}_{2,k}, \underline{x}_{2,k-1}, \dot{\underline{x}}_{2,k-1}, \ldots)$

Abbildung 5.1: Gekoppeltes System mit Durchgriffspfad

können erst ermittelt werden, nachdem alle Eingangswerte des Blocks vorliegen. Durch die Zusammenfassung von mehreren Systemgleichungen steigt die Wahrscheinlichkeit, dass algebraische Schleifen entstehen, die auf Gleichungsebene nicht vorhanden sind.

Das Simulationskonzept der Mechatronic Processing Objects unterstützt sowohl eine Auswertereihenfolge auf Gleichungsebene als auch auf Blockebene. Die einzelnen Durchgriffseingänge und Durchgriffsausgänge eines Blocks können jeweils zu einer endlichen Anzahl von Eingangs- und Ausgangsvektoren zusammengefasst werden.

Zur Ermittlung der Auswertereihenfolge des D-Codes wird zunächst ein gerichteter Graph zur Darstellung der Durchgriffspfade aufgebaut. Die Knoten des Graphen bilden dabei die Eingangsvektoren $\underline{u}_{D_{m_i}}$ und die Ausgangsvektoren $\underline{y}_{D_{m_i}}$, die jeweils eine Zusammenfassung mehrerer Durchgriffseingänge bzw. Durchgriffsausgänge sein können. Jeder Durchgriffspfad wird

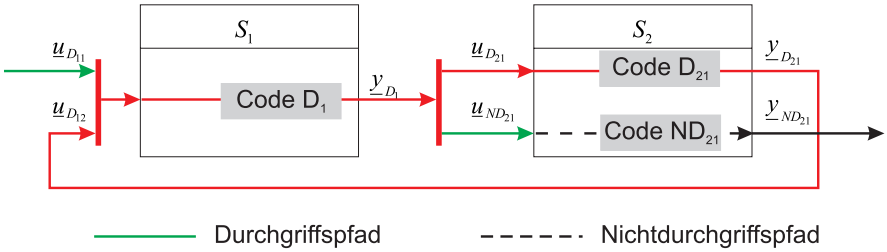


Abbildung 5.2: System mit algebraischer Schleife

durch eine gerichtete Kante vom Eingang zum Durchgriffsausgang dargestellt. Außerdem werden von den Ausgangsknoten jeweils gerichtete Kanten zu allen Durchgriffseingängen gezogen, auf die sie gekoppelt sind. Damit sind in dem Graphen alle Durchgriffspfade des Systems abgebildet. Führt man nun auf diesem Graphen eine *topologische Sortierung* aus, so erhält man einen Systempfad, der alle Durchgriffspfade umfasst. Wertet man die den jeweiligen Ausgangsvektoren zugeordneten D-Codes entlang dieses Systempfades aus, so ist eine korrekte Auswertereihenfolge für den Durchgriffscode sichergestellt. Ein konkreter Algorithmus zur Berechnung einer topologischen Sortierung ist z. B. in [Sed92] zu finden.

Zur Umsetzung des beschriebenen Verfahrens zur Ermittlung der Auswertereihenfolge stellt ein MPO mehrere Datenstrukturen bereit. Zunächst hält jedes MPO eine Liste seiner internen Durchgriffspfade im Attribut *DirectLinks* der Struktur *System*. Zur Repräsentation eines Durchgriffs dient die Struktur *DirectLink*, die jeweils eine Referenz auf eine Ausgangs- und eine Eingangsgröße enthält:

```
STRUCTURE System IS
  ...
  DirectLinks       : SEQ OF DirectLink;
  ...
  InOutCouplings    : Couples;
  ...
  EquationEvalOrder : SEQ OF EquationDirectLinkPart;
  ...
END

STRUCTURE DirectLink IS
  Output : Ident;
  Input  : Ident;
END
```

Des Weiteren enthält jedes MPO eine Liste aller Verkopplungen auf Basisblockebene, die als Attribut *BasicBlockCouplings* in der Struktur *Couples* abgelegt ist:

```
STRUCTURE Couples IS
  Hierarchy           : CoupleTree;
  BasicBlockCouplings : COPY OF SEQ OF BasicCoupleEquation;
END

STRUCTURE BasicCoupleEquation IS
  Equation            : Assign;
END

BasCplEqn IS REFERENCE TO BasicCoupleEquation
END
```

Bei den Verkopplungen auf Basisblockebene sind eventuelle hierarchische Verkopplungen bereits aufgelöst, sie geben direkt die Kopplung zwischen Ein- und Ausgängen auf unterster Ebene der Systemgleichungen an. Mit diesen Informationen lässt sich bereits der Graph zur Bestimmung der Auswertereihenfolge auf Gleichungsebene aufbauen. Das Ergebnis der topologischen Sortierung wird im Attribut *EquationEvalOrder* der Struktur *System* abgelegt, das eine Sequenz von Objekten der Struktur *EquationDirectLinkPart* enthält:

```
STRUCTURE EquationDirectLinkPart IS
  AuxiliarComp : SEQ OF AuxEqn;
  OutputComp   : SEQ OF OutEqn;
  CoupleComp   : SEQ OF BasCplEqn;
END
```

Ein Objekt der Struktur *EquationDirectLinkPart* besteht jeweils aus einer geordneten Menge von Referenzen auf Gleichungen zur Berechnung von Hilfs- und Ausgangsgrößen sowie Basissystemverkopplungen.

Zusätzlich zur Abbildung der Auswertereihenfolge auf Gleichungsebene bietet das Konzept der *Cluster* auch die Möglichkeit der Festlegung einer Auswertereihenfolge auf Blockebene. Bei einem Cluster wird eine Menge von Basisblöcken zu einer Kommunikationseinheit zusammengefasst. Eine detaillierte Vorstellung der Cluster findet sich im Kapitel 6, hier soll nur das Durchgriffsverhalten der Cluster betrachtet werden. Ein Cluster, dargestellt durch die Struktur *HierarchicalClusterNode*, enthält gemäß der im Unterkapitel 5.1.1 vorgestellten Klassifizierung drei Arten von Ein- und Ausgangsgrößen sowie ein Attribut *DirectLinkEvalOrder* zur Repräsentation der Auswertereihenfolge auf Blockebene:

```
STRUCTURE HierarchicalClusterNode IS
  ...
  IndirectInputs      : SEQ OF Ident;
  DirectInputs        : SEQ OF Ident;
```

```
  WhiteboxInputs      : SEQ OF ReceiveDataRef;
  IndirectOutputs     : SEQ OF Ident;
  DirectOutputs       : SEQ OF Ident;
  WhiteboxOutputs     : SEQ OF SendDataRef;
  ...
  DirectLinkEvalOrder : COPY OF SEQ OF ClusterDirectLinkPart;
END

STRUCTURE ClusterDirectLinkPart IS
  RecvInputs   : SEQ OF ReceiveDataRef;
  SetInputs    : SEQ OF ClusterCouplingRef;
  CalcOutputs  : SEQ OF ClusterRef;
  WriteOutputs : SEQ OF ClusterCouplingRef;
  SendOutputs  : SEQ OF SendDataRef;
END
```

Zur Repräsentation der Auswertung des D-Codes eines Subclusters dient die Struktur *ClusterDirectLinkPart*. Das Attribut *RecvInputs* gibt die zu empfangenden Datenpakete an. Im Attribut *SetInputs* wird dann die Zuordnung der einzelnen Elemente der Datenpakete auf die Eingangsgrößen des Subclusters festgehalten. Anschließend kann der entsprechende D-Code gerechnet werden, hier repräsentiert durch das Attribut *CalcOutputs*. Analog zu den Eingängen müssen dann wieder die Ausgangsgrößen beschrieben werden (Attribut *WriteOutputs*) und den zu sendenden Datenpaketen zugeordnet werden (Attribut *SendOutputs*).

5.1.4 Zusammenspiel Modell und Integrationsverfahren

Die vorangegangenen Unterkapitel haben sich im Wesentlichen mit der Berechnung der Ausgangsgrößen beschäftigt. Neben der Ermittlung der Ausgänge muss jedoch auch der innere Systemzustand fortgeschrieben werden, um das Zeitverhalten des Systems simulieren zu können. Dabei soll zunächst nur die Fortschreibung der kontinuierlichen Zustandsgrößen betrachtet werden. Auf die diskreten Zustände wird im folgenden Unterkapitel eingegangen.

In Abbildung 5.3 ist die Softwarearchitektur dargestellt, mit der die Anwendung eines Integrationsverfahrens auf ein Modell bei einem Mechatronic Processing Object umgesetzt wird. Die Architektur zur Umsetzung der nichtlinearen Simulation besteht aus fünf voneinander unabhängigen Softwarekomponenten, die je nach Anwendung in jeweils unterschiedlichsten Ausprägungen miteinander kombiniert werden können.

Zur Bereitstellung der Eingangsdaten des MPO steht die Komponente *InputInterface* zur Verfügung. Der Zugriff auf die Eingangsgrößen erfolgt mit Hilfe der Methoden *inputValuesD* und *inputValuesND*. Diese Methoden

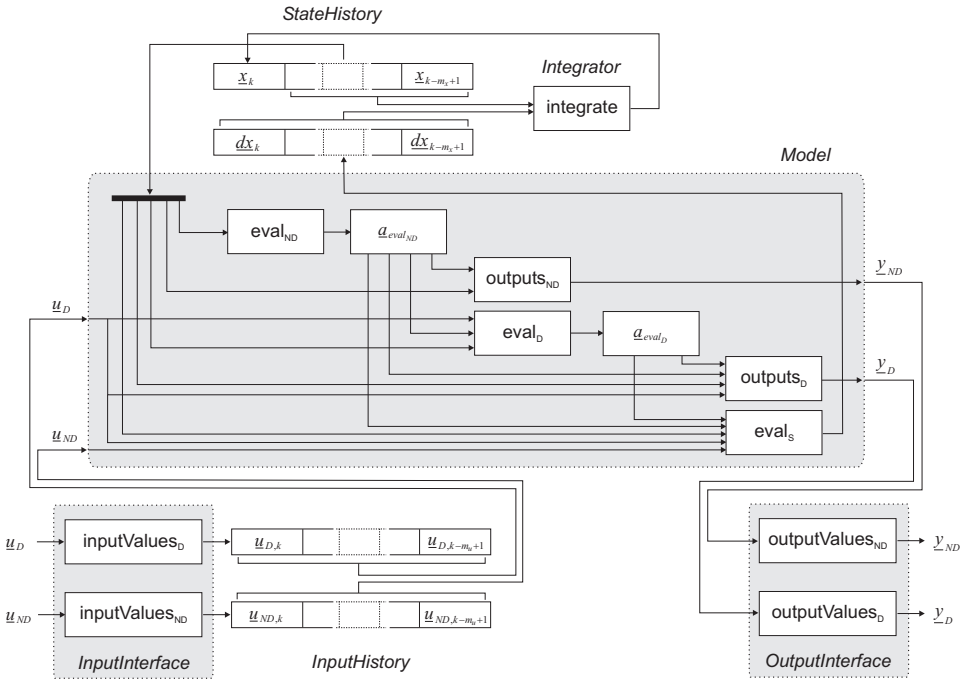


Abbildung 5.3: Zusammenspiel Modell und Integrationsverfahren

liefern für einen vorgegebenen Zeitpunkt t die gültigen Werte der entsprechenden Eingangsgrößen. Je nach Anwendung kann eine Komponente vom Typ *InputInterface* den Hardwarezugriff auf Sensordaten, den Empfang von Kommunikationsdaten oder auch das einfache Auslesen von bereits existierenden Variablen beinhalten. Zur Entkopplung der Auswertung des Modellverhaltens vom direkten Zugriff auf die Sensor- und Kommunikationsdaten stehen Felder zur Speicherung der Werte der Eingangsgrößen zur Verfügung. Die Methoden *inputValuesD* bzw. *inputValuesND* greifen zunächst immer auf diese Felder zu. Nur wenn für den aktuellen Zeitpunkt die Eingangswerte noch nicht abgefragt wurden, findet ein tatsächlicher Zugriff auf die Sensor- bzw. Kommunikationsdaten statt.

Für die Bereitstellung der Ausgangsdaten des MPO ist die Komponente *OutputInterface* zuständig. Sie stellt die Methoden *outputValuesD* und *outputValuesND* zur Verfügung, die vom Integrationsverfahren aufgerufen werden und die berechneten Werte der Ausgangsgrößen an diese Schnittstellenkomponente übergeben. Die Umsetzung dieser Werte in Stellgrößensignale für den angebundenen Prozess oder aber die Weitergabe an andere MPOs oder Visualisierungsprozesse ist Aufgabe der Komponente *OutputInterface*.

Die Gleichungen zur Beschreibung des Modellverhalten sind in der Komponente *Model* gekapselt. Dabei stehen die Methoden *evalND*, *evalD* und *evalS* zur Ermittlung der Zustandsableitungen sowie die Methoden *out-*

putsND und *outputsD* zur Ermittlung der Ausgangsgrößen zur Verfügung.

Die Komponente *Integrator* dient zur Kapselung des verwendeten numerischen Integrationsverfahrens. Als Schnittstelle dieser Komponente stehen dabei die folgenden Methoden und Informationen zur Verfügung:

- Integrationsmethode *integrate*
- Anzahl der betrachteten Zustandsschritte m_x
- Anzahl der betrachteten Zustandsableitungen m_{dx}
- Anzahl der betrachteten Eingangsschritte m_u

Eingangs- und Zustandshistorie werden getrennt vom Modellverhalten und Integrationsverfahren in jeweils eigenen Komponenten verwaltet. Zur Unterstützung von mehrstufigen Integrationsverfahren erlauben diese Komponenten die Ablage einer frei wählbaren Anzahl von Zwischenstufen. Die Zustandshistorie wird in der Komponente *StateHistory* in folgender Form gehalten:

- m_x zurückliegende Zustandsvektoren mit den dazugehörigen Zeitpunkten
- m_{dx} Vektoren der Zustandsableitungen mit den dazugehörigen Zeitpunkten

Die Historie der Eingangsgrößen wird in der Komponente *InputHistory* getrennt nach Durchgriffs- und Nichtdurchgriffseingängen gehalten.

5.1.5 Eventverarbeitung

Lag der Schwerpunkt der bisherigen Ausführungen auf der Simulation des kontinuierlichen Systemverhaltens, so soll in diesem Unterkapitel die Simulation von diskreten Ereignissen näher betrachtet werden.

Tritt bei der Simulation eines hybriden Systems ein diskretes Ereignis auf, so sind zwei Aspekte zu beachten: Zum einen muss das diskrete Systemverhalten in Form der Verarbeitung von Events einschließlich des Wechsels des diskreten Zustands simuliert werden. Darüber hinaus stellt jedes diskrete Ereignis auch eine Diskontinuität für den kontinuierlichen Anteil des Systemverhaltens dar. Diese Diskontinuitäten müssen bei der Simulation des kontinuierlichen Systemanteils berücksichtigt werden. Nach dem in [And94] beschriebenen Simulationsalgorithmus für hybride Modelle ergibt sich für die Eventverarbeitung von Mechatronic Processing Objects der nachfolgend aufgeführte Algorithmus:

```
initialize();
while (t <= t_end) {
```

```
  while ((t <= t_end) and noEventOccured()) {
    execContinuousStep();
  }
  if (eventOccured()) {
    detectRiseOfFirstEvent();
    while (getEvent()) {
      execTransition();
    }
  }
}
```

Ist während dem Fortschreiten des kontinuierlichen Systemverhaltens ein diskretes Ereignis aufgetreten, so wird der kontinuierliche Simulationsschritt unterbrochen. Zur Vermeidung von numerischen Problemen und zum Erreichen der geforderten Genauigkeit muss zunächst das erste Auftreten einer Unstetigkeit für diesen Simulationsschritt innerhalb einer vorgegebenen Genauigkeit ermittelt werden. Dies kann durch Intervallhalbierung nach dem in [Lef96] beschriebenen Verfahren durchgeführt werden. Der Simulationsalgorithmus für Mechatronic Processing Objects behandelt auch Unstetigkeiten als Events. Die Eventverarbeitung für ein MPO umfasst die Behandlung der folgenden Ereignisse:

1. Eintreffen einer Message v
2. Auftreten eines internen Events e

Über die Behandlung dieser beiden Ereignisarten sind die auftretenden Unstetigkeiten bei der Simulation eines MPO vollständig abgedeckt. Der Wechsel eines diskreten Zustands kann ausschließlich durch eine Message oder einen internen Event ausgelöst werden. Ebenso kann eine Wertänderung an einer diskreten Speichergröße s nur durch eine Message oder einen internen Event ausgelöst werden.

Sind Teile eines Systemverhaltens in Form von bedingten Anweisungen (*if-then-else* Strukturen) modelliert, so wird in der MPO-Repräsentation für jede Bedingung eine eigene diskrete Speichervariable eingeführt, die den Wert der Bedingung aus dem letzten Zeitschritt repräsentiert. Weicht der aktuelle Wert der Bedingung von dem Wert der entsprechenden Speichervariablen ab, so wird ein interner Event zur Signalisierung der durch den Wechsel der Bedingung verursachten Unstetigkeit ausgelöst.

Bei der Simulation von mechatronischen Systemen unter Echtzeitbedingungen ist aufgrund der kurzen Zykluszeiten eine iterative Ereignisdetektion nicht möglich, da dafür nicht genügend Rechenzeit zur Verfügung steht. In diesem Fall reduziert sich die Eventverarbeitung auf die Durchführung der Kontrollwechsel, die direkt in einen kontinuierlichen Integrationsschritt eingebettet werden können. Der nachfolgende Codeausschnitt zeigt eine Umsetzung am Beispiel des Integrationsverfahrens *Euler*:

```
/* Processing of D-code */
U.inputValuesD(U.u[0], t);
M->evalD(X.x[0], U.u[0], t);
if (outputValuesD != NULL) {
  M->outputsD(y, X.x[0], U.u[0], t);
  outputValuesD(y, t);
}

/* Getting input for ND-code */
U.inputValuesND(U.u[0], t);

/* Event processing and control mode switch */
M->evalDiscrete();
outputEvents();

/* Processing of S-code */
M->evalS(X.dx[0], X.x[0], U.u[0], t);

/* Calculation of next continuous state */
for (i = 0; i < M->n_x; ++i) {
  X.x[0][i] = X.x[0][i] + h * X.dx[0][i];
}
X.t_x[0] = X.t_dx[0] = t;

/* Calculation of ND-code for next time step */
M->evalND(X.x[0], U.u[0], t + h);
if (outputValuesND != NULL) {
  M->outputsND(y, X.x[0], U.u[0], t + h);
  outputValuesND(y, t);
}
```

5.2 Interpretierende Modellauswertung

Die Beschreibung des Systemverhaltens eines Mechatronic Processing Object erfolgt in Form von Modellgleichungen und Automatentransitionen. In jedem Simulationsschritt muss diese Beschreibung numerisch ausgewertet werden. Einen sehr flexiblen Ansatz für die Modellauswertung stellt dabei ein Interpretierer da:

> Ein Interpretierer ist ein Programm, das eine abstrakte Sprachmaschine implementiert. Seine Eingaben sind das Quellprogramm und dessen Eingabedaten. Die Operationen des Programms werden schrittweise ausgeführt (interpretiert) und Ergebnisse als Ausgabe produziert. [Kas90]

Vorteil einer interpretierenden Modellauswertung ist, dass bei Modifikation der Modellrepräsentation keine aufwendigen Transformationsschritte wie Codeerzeugung und Kompilierung nachfolgen müssen. Die Änderungen an der Modellrepräsentation sind direkt für den Simulationsprozess verfügbar. Die interpretierende Modellauswertung ist auch sehr gut für das Modell-Debugging geeignet.

Für den Anwendungsfall einer interpretierenden Modellauswertung kann ein MPO ein optionales Interpretierer-Modul enthalten (s. Abbildung 4.1). Das Interpretierer-Modul besteht aus dem *Interpreter Adaptor* und der *DSC Data Structure*. Der *Interpreter Adaptor* implementiert eine abstrakte Sprachmaschine für die verarbeitungsorientierte Beschreibungssprache DSC. Die DSC-Repräsentation des MPO wird als interne Datenstruktur *DSC Data Structure* in den Speicher geladen. Auf dieser internen Datenstruktur findet dann die schrittweise Abarbeitung der Systemgleichungen statt. Als Schnittstelle zum Integrationsverfahren stellt der Interpreter Adaptor die Auswertung der Systemgleichungen in Form der Methoden *evalD*, *evalND*, *evalS*, *outputsD*, *outputsND* und *evalDiscrete* zur Verfügung.

Speziell für Zwecke des Modell-Debugging bietet der Interpreter Adapter noch weitere Eingriffsmöglichkeiten in die Abarbeitung der Systemgleichungen, dazu zählen unter anderem:

- Modifikation von Koppelgleichungen und Umschaltung zwischen hierarchischer Verkopplung und Kopplung auf Basissystemebene
- Änderung und Überprüfung von Wertebereichen für Systemgrößen
- Anzeige und Modifikation beliebiger Systemgrößen und Hilfsvariablen
- Auswertung von beliebigen Ausdrücken in DSC
- Unterbrechung der Simulation in Abhängigkeit von beliebigen DSC-Ausdrücken

Für eine weitergehende Betrachtung des Einsatzes der interpretierenden Modellauswertung für das Modell-Debugging sei an dieser Stelle auf das Kapitel 8 verwiesen.

5.3 Codeerzeugung

Möchte man längere Offline-Simulationsläufe oder Hardware-in-the-Loop-Simulationen durchführen, so wird die Ausführungszeit der Modellauswertung zu einem sehr entscheidenden Kriterium. Liegt das Systemverhalten eines mechatronischen Systems in Form von kompiliertem Maschinencode vor, so lässt sich damit erfahrungsgemäß die Laufzeit der Simulation gegenüber einem interpretierenden Ansatz ungefähr halbieren. Ein üblicher Ansatz ist die Generierung von Programmcode in der Programmiersprache

C, der dann durch handelsübliche Compiler in sehr effizienten Maschinencode für unterschiedlichste Prozessorplattformen übersetzt wird.

Zur Codeerzeugung für Mechatronic Processing Objects steht ein Codegenerator zur Verfügung, der aus der DSC-Repräsentation eines MPO C-Programmcode erzeugt. Dieser C-Code kann dann mit einem C-Compiler übersetzt und als dynamisch ladbares Objekt vom MPO nachgeladen werden. Für diesen Anwendungsfall kann ein MPO einen optionalen *Compiled Model Adaptor* enthalten (s. Abbildung 4.1).

5.3.1 C-Datenstruktur *SimModel*

Das Systemverhalten wird in Form der normierten C-Datenstruktur *SimModel* bereitgestellt. Dabei wird ein objektorientierter Ansatz verfolgt, die Methoden *evalD*, *evalND*, *evalS*, *outputsD*, *outputsND* und *evalDiscrete* zur Auswertung der Systemgleichungen sind in Form von Funktionszeigern Bestandteil der Datenstruktur. Die einzelnen Komponenten der Datenstruktur sind nachfolgend aufgeführt.

Name des MPO und Dimensionen der Systemgrößen

Der Name des MPO sowie die Dimensionen der Systemgrößen werden durch die in Tabelle 5.2 aufgeführten Datenstrukturkomponenten repräsentiert.

`char *name`	Name des MPO
`int n_p`	$dim(\underline{p})$
`int n_uND`	$dim(\underline{u}_{ND})$
`int n_uD`	$dim(\underline{u}_{D})$
`int n_yND`	$dim(\underline{y}_{ND})$
`int n_yD`	$dim(\underline{y}_{D})$
`int n_v`	$dim(\underline{v})$
`int n_z`	$dim(\underline{z})$
`int n_q`	$dim(\underline{q})$, entspricht der Anzahl der im MPO enthaltenen Basisblöcke
`int n_s`	$dim(\underline{s})$
`int n_x`	$dim(\underline{x})$

Tabelle 5.2: C-Datenstruktur *SimModel*

void (*init)(void)

Diese Funktion dient zur Initialisierung des MPO.

void (*init_chan)(void)

Diese Funktion dient zur Initialisierung der Kommunikationskanäle.

void (*init_p)(RealT *)

Diese Funktion dient zur Bereitstellung der Initialwerte der Systemparameter $\underline{p}$. Als Ergebnis des Aufrufs `init_p(p)` ist das Array p mit den modellierten Parameter-Initialwerten besetzt.

void (*set_p)(RealT *)

Diese Funktion dient zum Setzen der Systemparameter $\underline{p}$. Als Ergebnis des Aufrufs `set_p(p)` werden die Systemparameter des MPO mit den Werten aus dem Array p besetzt.

void (*init_s)(RealT *)

Diese Funktion dient zur Bereitstellung der Initialwerte der diskreten Speichergrößen $\underline{s}$. Als Ergebnis des Aufrufs `init_s(s)` ist das Array s mit den modellierten Initialwerten besetzt.

void (*get_s)(RealT *)

Diese Funktion dient zur Bereitstellung der aktuellen Werte der diskreten Speichergrößen $\underline{s}$. Als Ergebnis des Aufrufs `get_s(s)` werden die aktuellen Werte der diskreten Speichergrößen in das Array s geschrieben.

void (*set_s)(RealT *)

Diese Funktion dient zum Setzen der diskreten Speichergrößen $\underline{s}$. Als Ergebnis des Aufrufs `set_s(s)` werden die diskreten Speichergrößen des MPO mit den Werten aus dem Array s besetzt.

void (*init_q)(IntT *)

Diese Funktion dient zur Bereitstellung des diskreten Startzustands $\underline{q}_0$. Als Ergebnis des Aufrufs `init_q(q)` ist das Array q mit den modellierten diskreten Startzuständen besetzt.

void (*get_q)(IntT *)

Diese Funktion dient zur Bereitstellung des aktuellen diskreten Systemzustands $\underline{q}$. Als Ergebnis des Aufrufs `get_s(q)` werden die aktuellen diskreten Zustände der einzelnen Basisblöcke in das Array q geschrieben.

void (*set_q)(IntT *)

Diese Funktion dient zum Setzen des diskreten Zustands $\underline{q}$. Als Ergebnis des Aufrufs `set_q(q)` werden die diskreten Zustände der einzelnen Basisblöcke des MPO mit den Werten aus dem Array q besetzt.

void (*init_x)(RealT *)

Diese Funktion dient zur Bereitstellung der Initialwerte der diskreten Speichergrößen $\underline{s}$. Als Ergebnis des Aufrufs `init_s(s)` ist das Array s mit den modellierten Initialwerten besetzt.

void (*evalND)(RealT *, RealT)

Diese Funktion dient zur Berechnung der Hilfsvariablen, die sowohl zur Berechnung der Zustandsableitungen $\underline{\dot{x}}$ als auch zur Berechnung der Nichtdurchgriffsausgänge erforderlich sind (s. Tabelle 5.1). Bei dem Aufruf `evalND(x, t)` werden die aktuellen Werte der kontinuierlichen Zustandsgrößen zum Zeitpunkt t als Array x übergeben. Die daraus ermittelten Werte der Hilfsvariablen werden modulintern abgelegt.

void (*evalD)(RealT *, RealT *, RealT)

Diese Funktion dient zur Berechnung der Hilfsvariablen, die sowohl zur Berechnung der Zustandsableitungen $\underline{\dot{x}}$ als auch zur Berechnung der Durchgriffsausgänge erforderlich sind (s. Tabelle 5.1). Bei dem Aufruf `evalD(x, u_D, t)` werden die aktuellen Werte der kontinuierlichen Zustandsgrößen zum Zeitpunkt t als Array x und die aktuellen Werte der Durchgriffseingänge als Array u_D übergeben. Die daraus ermittelten Werte der Hilfsvariablen werden modulintern abgelegt.

void (*evalS)(RealT *, RealT *, RealT *, RealT)

Diese Funktion dient zur Berechnung der Zustandsableitungen $\underline{\dot{x}}$. Als Ergebnis des Aufrufs `evalS(dx, x, u_ND, t)` ist das Array dx mit den neu berechneten Zustandsableitungen besetzt. Eingangsparameter sind die aktuellen Werte der kontinuierlichen Zustandsgrößen als Array x, die aktuelle Systemzeit t und die aktuellen Werte der Nichtdurchgriffseingänge als Array u_ND.

void (*outputsND)(RealT *, RealT *, RealT)

Diese Funktion dient zur Berechnung der Nichtdurchgriffsausgänge $\underline{y}_{ND}$. Als Ergebnis des Aufrufs `outputsND(y_ND, x, t)` ist das Array y_ND mit den berechneten Werten der Nichtdurchgriffsausgänge besetzt. Eingangsparameter sind die aktuellen Werte der kontinuierlichen Zustandsgrößen als Array x und die aktuelle Systemzeit t.

void (*outputsD)(RealT *, RealT *, RealT *, RealT)

Diese Funktion dient zur Berechnung der Durchgriffsausgänge $\underline{y}_D$. Als Ergebnis des Aufrufs `outputsD(y_D, x, u_D, t)` ist das Array y_D mit den

berechneten Werten der Durchgriffsausgänge besetzt. Eingangsparameter sind die aktuellen Werte der kontinuierlichen Zustandsgrößen als Array x, die aktuelle Systemzeit t und die aktuellen Werte der Durchgriffseingänge als Array u_D.

int (*evalDiscrete)(void)

Diese Funktion implementiert die Transitionsfunktion δ. Da sich die diskreten Größen eines mechatronischen Systems im Vergleich zu den kontinuierlichen Systemgrößen nur selten ändern, werden die diskreten Systemgrößen aus Effizienzgründen nicht als vektorielle Funktionsparameter übergeben, sondern über interne, statische Variablen implementiert. Ist ein expliziter Zugriff auf die diskreten Speichergrößen oder die diskreten Zustände erforderlich, so stehen dafür die separaten Funktionen `get_s`, `set_s`, `get_q` und `set_q` zur Verfügung. Ist der Rückgabewert des Aufrufs `evalDiscrete()` Null, so hat kein Wechsel des Kontrollmodus stattgefunden. Der Rückgabewert zeigt die Detektion einer Unstetigkeit an und kann bei der Simulation des kontinuierlichen Systemverhaltens entsprechend berücksichtigt werden.

SimModelInfo I

In der Struktur `SimModelInfo` werden die für die Interaktion mit dem Anwender erforderlichen Informationen über die einzelnen Systemgrößen gehalten:

```
typedef struct {
   int                 m;       /* number of blocks */
   struct BlockInfo    *blocks;
   struct VariableInfo *params,
                       *inputs,
                       *outputs,
                       *states,
   ...
} SimModelInfo;
```

Jedes Subsystem einschließlich des obersten Systems ist durch ein Objekt vom Typ `struct BlockInfo` repräsentiert. Die gesamte Systemhierarchie ist als lineares Array in dem Strukturelement `blocks` abgelegt. Die Informationen über die einzelnen Systemgrößen sind ebenfalls in Form von jeweils einem Array pro Systemgrößentyp abgelegt. Pro Subsystem wird im `struct BlockInfo` angegeben, welcher Ausschnitt aus den jeweiligen Systemgrößen-Arrays dem jeweiligen Subsystem zugeordnet ist:

```
struct BlockInfo {
   char *blockName;
   int  noOfParams,
```

```
        globParamOffset;
   int  noOfInputs,
        globInputOffset;
   int  noOfOutputs,
        globOutputOffset;
   int  noOfStates,
        globStateOffset;
   ...
   int  noOfSubsystems,
        globSystemOffset;
};
```

Pro Systemgröße gespeichert sind der ursprüngliche Name aus dem Modellierungswerkzeug sowie die Einheit. Darüber hinaus ist ein Zeiger auf die Variable im C-Code abgelegt, über den der aktuelle Wert der Systemgröße ausgelesen oder modifiziert werden kann. Die Bereitstellung der Informationen über die einzelnen Systemgrößen geschieht durch den Typen `struct VariableInfo`:

```
struct VariableInfo {
   char  *name;
   char  *unit;
   RealT *ref;
};
```

5.3.2 Auswertung der Systemgleichungen

Die Auswertung der Systemgleichungen erfolgt in den Funktionen *evalD*, *evalND*, *evalS*, *outputsD*, *outputsND* und *evalDiscrete*. Nachfolgend wird der Code für die Auswertung der Systemgleichungen exemplarisch anhand der Funktionen *evalS* und *evalDiscrete* näher erläutert.

Auswertung der Zustandsgleichung f

Die Berechnung des Zustandscode (S-Code) ist in der Funktion *evalS* umgesetzt. Der Code hat die folgende Struktur:

```
void evalS(RealT *dx, RealT *x, RealT *u_ND, RealT t)
{
  switch (Q0) {
  case 0:
    /* S-Code */
    break
  case 1:
    /* S-Code */
    break;
```

```
  ...
  default:
    /* unknown control mode */
  }

  switch (Q1) {
  case 0:
    /* S-Code */
    break
  case 1:
    /* S-Code */
    break;
  ...
  default:
    /* unknown control mode */
  }

  ...
}
```

Für jeden Basisblock ist der zu berechnende S-Code in Form einer switch-Anweisung umgesetzt. Abhängig vom aktuellen Kontrollmodus eines jeden Basisblocks wird der für den jeweiligen Kontrollmodus gültige Zustandscode berechnet. Die Kontrollmodi der einzelnen Basisblöcke werden durch die Variablen `Q0` bis `Q`n_q repräsentiert.

Die vorgestellte Struktur gilt analog für die Funktionen *evalD*, *evalND*, *outputsD* und *outputsND*.

Auswertung der Transitionsfunktion δ

Die Auswertung der Transitionsfunktion δ ist in der Funktion *evalDiscrete* umgesetzt. Der prinzipielle Aufbau des C-Codes dieser Funktion ist nachfolgend dargestellt:

```
int evalDiscrete()
{
  if (pendingMessageOrEventInBlock(0)) {
    ev = getMessageOrEventFromBlock(0, &isMessage);
    if (isMessage) {
      switch (ev) {
      case V0_0:
        switch (Q0) {
        case 3:
          ...
          break;
```

```
        case 7:
          ...
          break;
        default:
          ...
        }
        break;
      case V0_1:
        ...
      }
    }
    else /* internal event */ {
      switch (ev) {
      case E0_0:
        ...
      }
    }
  }

  if (pendingMessageOrEventInBlock(1)) {
    ev = getMessageOrEventFromBlock(1);
    if (isMessage) {
      switch (ev) {
      case V1_0:
      ...
}
```

Die Umsetzung des Codes für die Transitionsfunktion δ erfolgt basisblockweise. Pro Basisblock existiert eine Queue, die alle Messages und internen Events des Basisblocks hält.

Mit Hilfe der Funktion `pendingMessageOrEventInBlock` wird abgefragt, ob eine Message oder ein internes Event für den Block ansteht, der relevante Block wird dabei als Indexparameter übergeben. Die Entnahme der Message bzw. des internen Events mit der höchsten Priorität geschieht über die Funktion `getMessageOrEventFromBlock`. Die Alphabete der möglichen Messages und internen Events werden über Konstanten `V...` bzw. `E...` abgebildet. Abhängig von der aktuellen Message bzw. dem aktuellen internen Event wird über eine switch-Anweisung der auszuführende Transitionscode ausgewählt, der wiederum von dem aktuellen Kontrollmodus des Basisblocks abhängig ist.

5.4 SIMBA

In diesem Unterkapitel soll die Simulationsumgebung SIMBA präsentiert werden, in der die im Vorangegangenen beschriebenen Konzepte umgesetzt sind. SIMBA (SIMulation BAckend) [Hom97] ist am MLaP als Nachfolger des Simulationswerkzeugs SIMEX [Lef96] im Rahmen des vom BMBF geförderten Projekts METEOR (Method and Tool Development for Micro-Systems Design) entwickelt worden [Hom96].

Die offene Simulationsumgebung SIMBA besteht aus einem Satz von kooperativen Softwarekomponenten in Form von kompakten, eigenständigen UNIX-Programmen, die über Interprozess-Kommunikation zusammenarbeiten. Jede der kooperativen Softwarekomponenten läuft dabei in einem eigenen UNIX-Prozess. Mit Hilfe der Scriptsprache Tcl [Ous94] ist es möglich, Simulationsexperimente zu beschreiben. Dabei stehen auch simulationsspezifische Spracherweiterungen zur Verfügung. Basierend auf der standardmäßigen Interprozess-Kommunikation von UNIX stellt SIMBA eine offene Schnittstelle für Simulatorkopplungen zur Verfügung. Eine Simulatorkopplung von SIMBA mit dem 3D-Magnetfeld-Simulationswerkzeug PROFI [PRO96] ist im Rahmen des Projekts METEOR erfolgreich umgesetzt worden.

Die Architektur von SIMBA orientiert sich an den zentralen Funktionen eines Simulationswerkzeugs: Bedienschnittstelle, Berechnung der Simulationsdaten und graphische Visualisierung. Aus dieser funktionalen Dreiteilung ergibt sich die in Abbildung 5.4 dargestellte Umsetzung von SIMBA in Form von drei eigenständigen UNIX-Prozessen: Bedienprozess (user interaction process), Simulationsprozess (simulation process) und Visualisierungsprozess (visualization process).

Der Bedienprozess startet den Simulations- und den Visualisierungsprozess über den UNIX-Pipe-Mechanismus. Damit steht jeweils ein bidirektionaler Datenkanal zum Nachrichtenaustausch mit dem Bedienprozess zur Verfügung. Fehlermeldungen werden zum Bedienprozess umgeleitet, wo sie durch einen Automatismus in Form eines Fehlerdialogs sofort angezeigt werden. Für den Transport der Simulationsdaten ist ein hoher Durchsatz erforderlich, der Simulations- und der Visualisierungsprozess sind daher untereinander direkt durch eine FIFO/Socket-Verbindung verbunden.

Für die Umsetzung des Bedienprozesses wurde der GUI-Interpreter Interact/X am MLaP entwickelt. Interact/X basiert auf X-Windows, OSF/Motif, Motif Tools [Fla94] und Tcl. Beim Einsatz dieses GUI-Interpreters ist die Spezifikation der Benutzerschnittstelle zweigeteilt: Die statische Struktur des GUI, d. h. Struktur, Anordnung und Erscheinungsbild der Oberflächenelemente wird in Form von X-Resource-Dateien beschrieben. Die dynamischen Aspekte wie z. B. Callbacks werden in der einfach erlernbaren Scriptsprache Tcl codiert. Ein Modul für Interprozess-Kommunikation ist integriert. Mit Interact/X steht damit ein GUI-Interpreter zur Verfügung,

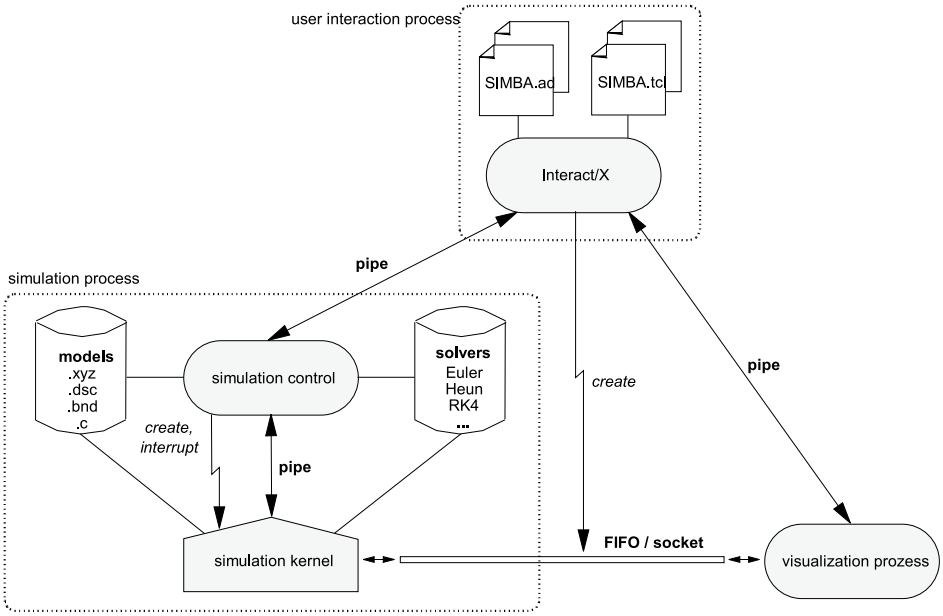


Abbildung 5.4: Architektur des Offline-Simulationswerkzeugs SIMBA

mit dem sich der Bedienprozess der offenen Simulationsumgebung SIMBA komfortabel an den individuellen Einsatz anpassen lässt.

Der Simulationsprozess besteht aus zwei nebenläufigen Einheiten, dem Steuerungsprozess (simulation control) und dem eigentlichen Simulationskern (simulation kernel), die über den UNIX-Pipe-Mechanismus kommunizieren. Der Steuerungsprozess ist für die Kommunikation mit dem Bedienprozess zuständig, wertet die Kommandos aus und steuert den Simulationskern.

Die Berechnung der Simulationsdaten erfolgt im Simulationskern. Zentraler Bestandteil des Simulationskerns sind die an der Simulation beteiligten Mechatronic Processing Objects. Im Fall einer verteilten Simulation kann der Simulationskern aus mehreren UNIX-Prozessen bestehen, auf denen unterschiedliche Mechatronic Processing Objects gerechnet werden. Der Steuerungsprozess ist u. a. dafür zuständig, dass Prozesse für neue MPOs erzeugt und wieder terminiert werden. Darüber hinaus steuert er die Codegenerierung und Übersetzung für den Fall, dass das Systemverhalten eines MPO als kompilierter Maschinencode zur Verfügung stehen soll. Die für eine Simulatorkopplung erforderliche Koordination geschieht ebenfalls über den Steuerungsprozess.

Beim Visualisierungsprozess sind zwei Varianten verfügbar: Ein zweidimensionales Plot-Fenster für die Darstellung von Signalverläufen und eine auf der Graphikbibliothek OpenInventor [Wer94] basierende, dreidimensionale Animationsanwendung. Die Visualisierungsvarianten können sowohl alternativ als auch beide gleichzeitig verwendet werden.

Kapitel 6

Verteilte Informationsverarbeitung und autonome Systeme

Komplexe, mechatronische Systeme enthalten eine Vielzahl von lokalen Reglern, die jeweils für die korrekte Funktion einer Teilkomponente sorgen. Durch hierarchische Organisation dieser Regler und zusätzliche, übergeordnete Reglerstrukturen wird die gewünschte Funktionalität des Gesamtsystems hergestellt. Dieses Kapitel behandelt die Unterstützung der verteilten Informationsverarbeitung in Entwurf und Realisierung durch das Konzept der Mechatronic Processing Objects. Dazu dient zunächst das Ordnungsprinzip der Cluster, das im ersten Unterkapitel vorgestellt wird. Ein weiterer wichtiger Aspekt ist die Erzeugung neuer Teilsysteme und die dynamische Verkopplung zur Laufzeit. Mit derartigen dynamischen Systemstrukturen beschäftigt sich das zweite Unterkapitel.

6.1 Cluster

Beim Entwurf eines mechatronischen Systems wird zunächst unter Anwendung der Subsystemtechnik ein hierarchisches Modell aus einzelnen bereits im Rahmen eines Modellkatalogs verfügbaren oder auch neu zu entwerfenden Komponenten erstellt. Diese Subsystemhierarchie wird in DSC mit der Struktur *SpecificationTree* dargestellt.

Die Einführung der informationstechnischen Klassifizierung von Systemen in MFM, AMS und VMS macht eine Erweiterung der Strukturierungsmöglichkeiten in DSC erforderlich. Zusätzlich zur spezifizierten Subsystemhierarchie können zur Abbildung der tatsächlichen informationstechnischen Struktur hierarchische Cluster gebildet werden. Dazu wird die DSC-Struktur *ClusterNode* eingeführt:

```
STRUCTURE System IS
  ...
  Clustering : ClusterNode;
END

STRUCTURE ClusterNode IS
  Content : HierarchicalClusterNode | BlockHierarchy;
END

STRUCTURE HierarchicalClusterNode IS
  Name                : STRING;
  Type                : STRING;
  IndirectInputs      : SEQ OF Ident;
  DirectInputs        : SEQ OF Ident;
  WhiteboxInputs      : SEQ OF ReceiveDataRef;
  IndirectOutputs     : SEQ OF Ident;
  DirectOutputs       : SEQ OF Ident;
  WhiteboxOutputs     : SEQ OF SendDataRef;
  Subclusters         : SEQ OF ClusterNode;
  Couplings           : SEQ OF ClusterCoupling;
  Level               : CommunicationLevel;
  EvaluationTimes     : INTEGER;
  MappedTo            : INTEGER;
  DirectLinkEvalOrder : COPY OF SEQ OF ClusterDirectLinkPart;
END

STRUCTURE CommunicationLevel IS
  Level : INTEGER;
END

STRUCTURE ClusterDirectLinkPart IS
  RecvInputs   : SEQ OF ReceiveDataRef;
  SetInputs    : SEQ OF ClusterCouplingRef;
  CalcOutputs  : SEQ OF ClusterRef;
  WriteOutputs : SEQ OF ClusterCouplingRef;
  SendOutputs  : SEQ OF SendDataRef;
END

STRUCTURE ClusterCoupling IS
  Target  : DirectInputRef  | IndirectInputRef  | SendDataRef;
  Source  : DirectOutputRef | IndirectOutputRef | ReceiveDataRef;
  Mapping : SEQ OF IndexMap;
END
```

Zunächst ist ein Clusterknoten einfach eine Ansammlung von Subsystemen, die über eine gemeinsame informationstechnische Anbindung mit den restlichen Komponenten des Systems verbunden sind. Die einzelnen Clusterknoten können wieder zu neuen Clustern zusammengefasst werden. Es ergibt sich eine hierarchische Struktur, die die Informationsverarbeitung widerspiegelt. Die einzelnen Hierarchieebenen können dann als Prozesse, Prozessoren, MFMs, AMS bzw. VMS interpretiert werden, die Verwendung weiterer Hierarchieebenen ist ebenfalls möglich.

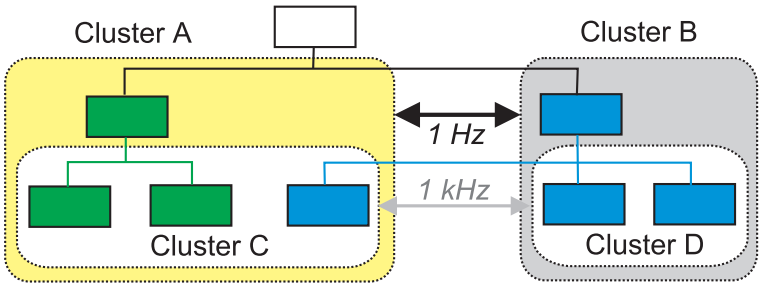


Abbildung 6.1: Strukturierung der Informationsverarbeitung mit Hilfe von Clustern

Die Clusterhierarchie und die in der Modellbildung spezifizierte Systemhierarchie sind zunächst voneinander völlig unabhängig. So können z. B. einige Komponenten eines Teilsystems einem anderen Cluster zugeordnet sein (s. Abbildung 6.1). Dies erlaubt eine hohe Flexibilität beim Entwurf einer geeigneten informationstechnischen Struktur. Eine Veränderung der Informationsverarbeitung erfordert zunächst keine Modifikationen am ursprünglichen Modell, damit lassen sich ohne großen Aufwand unterschiedlichste informationstechnische Strukturen modellieren. Hat man eine geeignete, modulare Struktur der Informationsverarbeitung gefunden, so stellt diese auch einen Anhaltspunkt für einen sinnvollen Aufbau der Subsystemhierarchie dar. Aufgrund dieser Erkenntnisse sollte dann auch die Subsystemhierarchie entsprechend angepasst werden, sinnvollerweise mit entsprechender Werkzeugunterstützung.

6.2 Dynamische Systemstrukturen

Die im vorangegangenen Unterkapitel vorgestellten hierarchischen Cluster stellen zunächst eine statische Strukturierung des Systems dar. Betrachtet man jedoch autonome Systeme, so ist auch die Unterstützung von über der Laufzeit veränderlichen Systemstrukturen erforderlich. Situationsabhängig treten autonome Systeme in neue Organisationsstrukturen ein oder verlassen die bisherige Teilstruktur. Für die Umsetzung autonomer Systeme ist es daher erforderlich, zur Laufzeit neue Instanzen eines Teilsystems erzeugen

und in das Gesamtsystem einbetten als auch wieder entfernen zu können. Ein typischer Anwendungsfall für dynamische Systemstrukturen ist die Abbildung von Verkehrsszenarien (s. auch Kapitel 9.2). Nachfolgend soll anhand der Definition der einzelnen Objekte in DSC die Repräsentation von dynamischen Systemstrukturen mit Mechatronic Processing Objects erläutert werden.

Zur Repräsentation eines strukturdynamischen Systems dient die optionale Struktur *DynamicStructure*. Sie umfasst eine Menge von Referenzvariablen (*SystemRef*) auf Teilsysteme sowie eine Liste der möglichen dynamischen Konfigurationen, die das System einnehmen kann:

```
STRUCTURE System IS
  ...
  DynamicStructures : SEQ OF DynamicStructure;
  ...
END

STRUCTURE DynamicStructure IS
  SystemRefs            : SEQ OF SystemRef;
  DynamicConfigurations : SEQ OF DynamicConfiguration;
END
```

Eine dynamische Konfiguration, dargestellt durch die Struktur *DynamicConfiguration*, repräsentiert einen Zustand des Gesamtsystems mit fester, laufzeitinvarianter Topologie. Solange sich das Gesamtsystem innerhalb derselben dynamischen Konfiguration befindet, finden keine dynamischen Veränderungen an der Systemstruktur statt. Ein Konfigurationswechsel (*ConfigurationChange*) kann nur durch ein diskretes Ereignis, d. h. durch eine *Message* oder einen *externen Event*, ausgelöst werden. Die zulässigen Konfigurationswechsel sind pro dynamischer Konfiguration angegeben:

```
STRUCTURE DynamicConfiguration IS
  Name                : STRING;
  ConfigurationChange : SEQ OF ConfigurationChange;
END

STRUCTURE ConfigurationChange IS
  TriggerEvent          : V;
  DynamicInstantiations : SEQ OF DynamicInstantiation;
  DynamicDeletions      : SEQ OF DynamicDeletion;
  NextConfiguration     : DynamicConfigurationRef;
END
```

Im Rahmen eines Konfigurationswechsels ist sowohl das Erzeugen neuer Teilsysteme (*DynamicInstantiation*) als auch das Entfernen von vorhandenen

Teilsystemen (*DynamicDeletion*) möglich. Dabei können jeweils Systemreferenzen gesetzt als auch Verkopplungen vorgenommen werden:

```
STRUCTURE DynamicInstantiation IS
  Instantiation    : Instantiation;
  SystemRefAssigns : SEQ OF SystemRefAssign;
  Couplings        : SEQ OF Assign;
END

STRUCTURE DynamicDeletion IS
  Instance         : SystemRef;
  SystemRefAssigns : SEQ OF SystemRefAssign;
  Couplings        : SEQ OF Assign;
END
```

Kapitel 7

Lineare Analyse und Synthese

Beim Entwicklungsprozess eines mechatronischen Systems ist der Entwurf der Informationsverarbeitung von besonderer Wichtigkeit. Die Qualität eines mechatronischen Produkts wird im Wesentlichen durch die Güte der eingesetzten Regelungen, Steuerungen und Signalaufbereitungen bestimmt. Zum Entwurf der Informationsverarbeitung kommen insbesondere Verfahren der linearen Systemtheorie zum Einsatz. Sie ermöglichen die Analyse der in der Modellbildung gewonnenen Streckenmodellansätze und die Identifikation von Systemparametern. Darauf aufbauend kann dann die Reglersynthese stattfinden, bei der eine Optimierung der Reglerstruktur und Reglerparameter durchgeführt wird.

In diesem Kapitel soll der Einsatz eines MPO in der linearen Systemanalyse und -synthese vorgestellt werden. Im ersten Abschnitt wird das zugrundeliegende Konzept erläutert und eine Einordnung in den Gesamtentwurfsprozess eines mechatronischen Systems gegeben. Ein Großteil der Systemmodelle liegt zunächst nur in nichtlinearer Form vor. Daher beschäftigt sich der zweite Abschnitt mit der Linearisierung nichtlinearer Systeme, der Ableitung von linearen Modellen aus nichtlinearen Beschreibungsformen. Im dritten Abschnitt wird schließlich die Darstellung linearer und linearisierter Systeme in DSC vorgestellt. Auf dieser Grundlage kann eine effiziente Auswertung erfolgen, die im letzten Abschnitt beschrieben wird.

7.1 Konzept

Der Reglerentwurfsprozess spielt bei der Entwicklung eines mechatronischen Systems eine zentrale Rolle. Die Ausstattung eines mechatronischen Systems mit regelungs- und steuerungstechnischen Komponenten ermöglicht es, ein der Produktspezifikation entsprechendes dynamisches Verhalten kostengünstig ohne hohen Aufwand bei der mechanischen Konstruktion zu erhalten.

Beim Reglerentwurf kommen insbesondere Verfahren der linearen Systemanalyse und -synthese zur Anwendung. Im Folgenden sollen daher die Besonderheiten im Umgang mit linearen Systemmodellen und die Auswirkungen auf eine verarbeitungsnahe Modellrepräsentation näher betrachtet werden.

Zentrale mathematische Darstellung für die Verfahren der linearen Systemanalyse und -synthese ist eine lineare Beschreibung des Systemverhaltens. Dazu wird das zu untersuchende System in der *linearen Zustandsraumdarstellung erster Ordnung* formuliert:

$$\begin{aligned} \dot{\underline{x}} &= \underline{A} \cdot \underline{x} + \underline{B} \cdot \underline{u} \\ \underline{y} &= \underline{C} \cdot \underline{x} + \underline{D} \cdot \underline{u} \end{aligned} \quad \text{mit}$$

n	Anzahl der Zustandsgrößen
p	Anzahl der Eingangsgrößen
q	Anzahl der Ausgangsgrößen
$\underline{u}(p)$	Eingangsvektor
$\underline{y}(q)$	Ausgangsvektor
$\underline{x}(n)$	Zustandsvektor
$\underline{A}(n,n)$	Dynamikmatrix (Systemmatrix)
$\underline{B}(n,p)$	Eingangsmatrix
$\underline{C}(q,n)$	Ausgangsmatrix
$\underline{D}(q,p)$	Durchgangsmatrix

Das Systemverhalten wird durch vier Matrizen charakterisiert. Dabei werden die Systemdynamik und das Ausgangsverhalten in Form linearer Abhängigkeiten vom aktuellen Zustands- und Eingangsvektor beschrieben.

Der Einsatz linearer Systemmodelle beim mechatronischen Entwurf ist in Abbildung 7.1 dargestellt. Zunächst steht das in der Modellbildung aufgestellte Modell zur Verfügung, das im Allgemeinen nichtlinear ist. Daraus muss für die relevanten Betriebspunkte des Systems ein lineares Systemmodell abgeleitet werden. Unter Angabe des jeweiligen Betriebspunkts, der sich aus den Eingangs- und Zustandsgrößen um den Betriebspunkt zusammensetzt, wird eine symbolische oder numerische Linearisierung des nichtlinearen Systems vorgenommen. Dies geschieht durch partielle Ableitung der Systemgleichungen nach den Zustands- und Eingangsgrößen. Als Endergebnis erhält man unter Berücksichtigung der aktuellen Modellparameter vier numerische Matrizen $\underline{A}$, $\underline{B}$, $\underline{C}$ und $\underline{D}$, die das Systemverhalten am entsprechenden Betriebspunkt beschreiben. Die Linearisierung ist zulässig, da es ohnehin Aufgabe der zu entwerfenden Regelung ist, das System im entsprechenden Betriebspunkt zu halten. Basierend auf den vier numerischen Matrizen können nun die Verfahren der linearen Analyse und Synthese aufsetzen. So können Eigenwerte, Streuungen und Frequenzgänge zur Reglerauslegung berechnet werden.

Die Ergebnisse der linearen Analyse können darüber hinaus auch zu einer automatischen Regleroptimierung verwendet werden. In einem nachgeschalteten Optimierer werden die gewonnen Kenngrößen ausgewertet und anhand einer geeigneten Zielfunktion gewichtet. Daraus wird ein neuer Satz

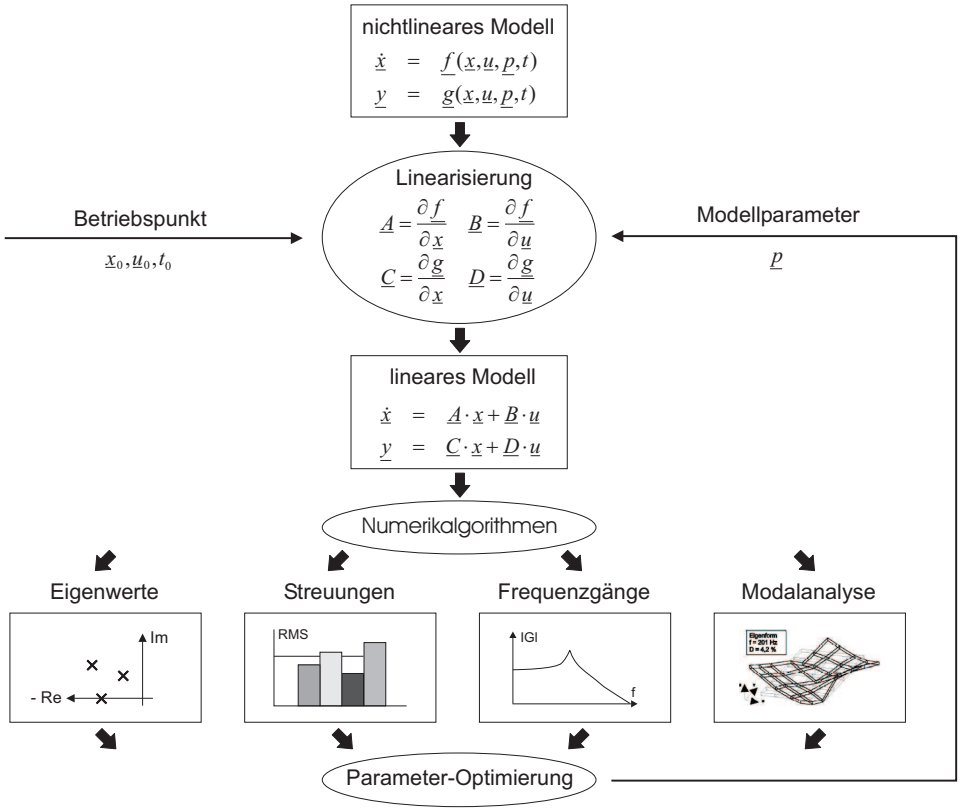


Abbildung 7.1: Einsatz linearer Modelle im mechatronischen Entwurfsprozess

von Modellparametern abgeleitet und die Ermittlung der numerischen Matrizen $\underline{A}$, $\underline{B}$, $\underline{C}$ und $\underline{D}$ mit anschließender Analyse erneut durchlaufen. Hat die Zielfunktion ein zuvor definiertes Optimum erreicht, so ist die Optimierung beendet.

Eine weiteres Mittel zur Untersuchung des Systemverhaltens, das auf einem linearen Systemmodell beruht, ist die Modalanalyse. Sie ermöglicht die Ermittlung von Eigenschwingungen des Systems, die sehr anschaulich in Form einer 3D-Animation visualisiert werden können. Die Modalanalyse kann insbesondere auch schon in frühen Phasen der Modellbildung einen Beitrag zum Verständnis der Systemdynamik leisten.

7.2 Linearisierung nichtlinearer Systeme

Als Ergebnis der Modellbildung hat der Ingenieur üblicherweise ein nichtlineares Modell des zu entwickelnden mechatronischen Systems vorliegen. Dieses muss für die Verfahren der linearen Analyse und Synthese in eine lineare Darstellung überführt werden. Dies ist die Aufgabe der Linearisierung,

die sowohl numerisch als auch symbolisch erfolgen kann.

Hat man in der Modellbildung ein Systemmodell in Form nichtlinearer Zustandsgleichungen 1. Ordnung aufgestellt, so erhält man nach der Methode von Newton-Kantorowitsch durch Anwendung einer Taylorentwicklung für einen Betriebspunkt $(\underline{x}_0, \underline{u}_0)$ folgendes lineares Modell [BS89b]:

$$\begin{aligned} \dot{\underline{x}} &= \underline{A}(\underline{x}-\underline{x}_0)+\underline{B}(\underline{u}-\underline{u}_0)+\underline{f}(\underline{x}_0,\underline{u}_0) \quad \text{mit} \quad A = \left.\frac{\partial(f_1,f_2,\ldots,f_n)}{\partial(x_1,x_2,\ldots,x_n)}\right|_{x=x_0} \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad B = \left.\frac{\partial(f_1,f_2,\ldots,f_m)}{\partial(u_1,u_2,\ldots,u_m)}\right|_{u=u_0} \\ \underline{y} &= \underline{C}(\underline{x}-\underline{x}_0)+\underline{D}(\underline{u}-\underline{u}_0)+\underline{g}(\underline{x}_0,\underline{u}_0) \quad \text{mit} \quad C = \left.\frac{\partial(g_1,g_2,\ldots,g_n)}{\partial(x_1,x_2,\ldots,x_n)}\right|_{x=x_0} \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad D = \left.\frac{\partial(g_1,g_2,\ldots,g_m)}{\partial(u_1,u_2,\ldots,u_m)}\right|_{u=u_0} \end{aligned}$$

Als Voraussetzung müssen $\underline{f}$ und $\underline{g}$ mindestens einmal stetig differenzierbar sein. Da die Modellparameter als Konstanten anzusehen sind, sind sie aus Gründen der Übersichtlichkeit nicht explizit mit aufgeführt.

Zentraler Punkt bei der Ableitung des linearen Systemmodells ist die Bestimmung der Matrizen $\underline{A}$, $\underline{B}$, $\underline{C}$ und $\underline{D}$ als den partiellen Ableitungen der Funktionen $\underline{f}$ und $\underline{g}$ nach den Zustandsvariablen $\underline{x}$ bzw. den Eingangsvariablen $\underline{u}$. Die Bestimmung dieser partiellen Ableitungen kann entweder numerisch oder aber symbolisch erfolgen.

7.2.1 Numerische Linearisierung

Viele regelungstechnische Entwurfswerkzeuge wie z. B. MATLAB/Simulink verwenden eine numerische Linearisierung. Die einzelnen Zustands- bzw. Eingangsgrößen werden dabei um einen kleinen Betrag von h um den Zustands- bzw. Eingangswert variiert und daraus wird ein symmetrischer Differenzenquotient ermittelt. Die Formel für den skalaren Fall lautet folgendermaßen [BS89a]:

$$\left(\frac{df}{dx}\right)_{x_0} = \frac{f(x_0+h)-f(x_0-h)}{2h} + O(h^2)$$

Angewendet auf die lineare Zustandsraumdarstellung erhält man damit folgende Zustandsgleichung:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & & \vdots \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \cdots$$

Für die einzelnen Einträge der Matrix $\underline{A}$ gilt:

$$\left(\frac{\partial f_i}{\partial x_j}\right)_{x_{j0}} = \frac{f_i(x_{00}, x_{10}, \ldots, x_{j0}+h, \ldots, x_{n0}))}{2h}$$

$$- \quad \frac{f_i(x_{00}, x_{10}, \ldots, x_{j0} - h, \ldots, x_{n0})}{2h} + O(h^2)$$

Die Bestimmung der weiteren Matrizen erfolgt analog.

Wie aus den Formeln ersichtlich, ist für die Durchführung einer numerischen Linearisierung nur die numerische Auswertung der Zustands- und Ausgangsgleichungen mit entsprechenden Zustands- und Eingangsvektoren erforderlich. Damit kann sie zunächst auf jedes nichtlineare Modell angewendet werden. Auch hierarchisch verkoppelte Systeme bereiten keine Schwierigkeiten, die Auswertung der Systemgleichungen kann von der nichtlinearen Simulation übernommen werden.

Allerdings birgt die numerische Linearisierung auch einige Probleme in sich. So können z. B. Unstetigkeiten allein aus den zahlenmäßigen Ergebnissen der Auswertung der Systemgleichungen nicht ohne Weiteres detektiert werden. Eine Linearisierung an einer Unstetigkeitsstelle kann zu völlig verfälschten Ergebnissen führen. Des Weiteren bringt die Ersetzung des Differentialquotienten durch den Differenzenquotienten numerische Ungenauigkeiten mit sich. Da sich das Systemverhalten üblicherweise nicht für jeden Zustand oder Ausgang getrennt berechnen lässt, muss z. B. für jeden Eintrag der Matrix $\underline{A}$ zweimal die Auswertung der gesamten Zustandsgleichungen ausgeführt werden. Bei n Zustandsgrößen und m Eingangsgrößen ergeben sich damit $2nm$ Auswertungen der Systemgleichungen.

7.2.2 Symbolische Linearisierung

Stehen die das Systemverhalten beschreibenden mathematischen Gleichungen zur Verfügung, so kann die Berechnung der partiellen Ableitungen der Zustands- und Ausgangsfunktionen auch symbolisch erfolgen. Der Ablauf einer symbolischen Linearisierung ist in Abbildung 7.2 schematisch dargestellt.

Zunächst muss für jedes Teilsystem eine linearisierte, symbolische Darstellung ermittelt werden. Dazu werden alle Zustands- und Ausgangsgleichungen des betrachteten Blocks symbolisch nach den Eingangs- und Zustandsgrößen differenziert. Dabei müssen im Besonderen auch Hilfsgrößen berücksichtigt werden. Viele Beschreibungssprachen auf der Ebene der mathematischen Modellbeschreibung ermöglichen den Einsatz von Hilfsgrößen, die in einem separaten Gleichungsblock berechnet werden und dann in den Zustands- und Ausgangsgleichungen verwendet werden können. Diese Hilfsgrößen sind ebenfalls symbolisch nach den Eingangs- und Zustandsgrößen abzuleiten und müssen in den Hauptgleichungen dann entsprechend beachtet werden. Ergebnis dieses ersten Schritts ist ein Satz von Matrizen $\underline{A}$, $\underline{B}$, $\underline{C}$ und $\underline{D}$, deren Einträge $\underline{x}$, $\underline{u}$, $\underline{p}$ und t noch als Variablen beinhalten, des Weiteren ein Satz von symbolischen Gleichungen zur Berechnung der abgeleiteten Hilfsgrößen. Damit liegt pro Teilsystem ein mathematisch exaktes, lineares Modell vor, das interpretativ ausgewertet werden kann.

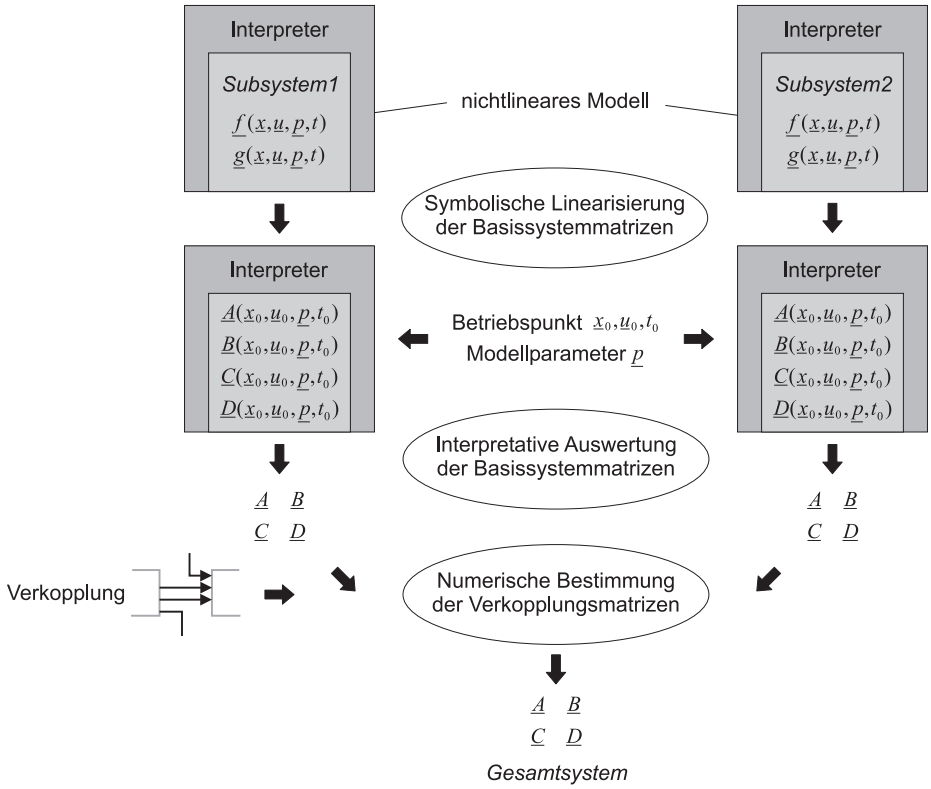


Abbildung 7.2: Symbolische Linearisierung

Im nächsten Schritt erfolgt nun die Berechnung der einzelnen Teilmatrizen durch Besetzen der Systemparameter und des Betriebspunkts und anschließender interpretativer Auswertung der Matrixeinträge. Dies braucht pro Parametersatz und Betriebspunkt für jedes Teilsystem jeweils nur einmal durchgeführt werden.

Abschließend müssen aus den Matrizen der Teilsysteme noch die Matrizen des Gesamtsystems berechnet werden. Dies kann nach dem in [Sch94] beschriebenen Verfahren geschehen: Ausgehend von den Einzelmatrizen verkoppelter Teilsysteme wird durch die Verknüpfung mit einer Koppelmatrix eine geschlossene lineare Darstellung des Teilsystems ermittelt. Dazu ist lediglich die Durchführung von Matrizen-Additionen und -Multiplikationen erforderlich. Enthält ein Teilsystem Rückführungen, so wird zusätzlich eine Matrizen-Inversion durchgeführt. Dieses Verfahren kann beginnend bei den Basissystemen auf unterster Ebene bis zur Ebene des Gesamtsystems auf jeder Hierarchieebene angewendet werden, so dass schließlich eine geschlossene lineare Darstellung des Gesamtsystems zur Verfügung steht. Die auf den einzelnen Hierarchieebenen verwendeten Koppelmatrizen können direkt aus den hierarchischen Systemverkopplungen abgeleitet werden.

7.3 Darstellung linearer und linearisierter Systeme in DSC

Aus den Darstellungen der vorausgegangenen Abschnitte ergibt sich der folgende modellspezifische Informationsgehalt, der für die Verfahren der linearen Analyse und Synthese benötigt wird:

1. Code zur Berechnung der Matrizen $\underline{A}$, $\underline{B}$, $\underline{C}$, $\underline{D}$ eines Basissystems
   - Gleichungen zur Berechnung der Hilfsgrößen des ursprünglichen (nichtlinearen) Systems
   - Ableitungen der Hilfsgrößen nach dem Zustand für die Berechnung der Matrizen $\underline{A}$ und $\underline{C}$
   - Ableitungen der Hilfsgrößen nach dem Eingang für die Berechnung der Matrizen $\underline{B}$ und $\underline{D}$
   - Symbolische Einträge der Matrizen $\underline{A}$, $\underline{B}$, $\underline{C}$, $\underline{D}$
2. Verkopplungsinformationen zum Aufbau der Koppelmatrizen

Zur Ablage der Matrizen eines Basissystems wird in DSC in der Struktur `BasicBlock` ein Attribut `LinearRepresentation` eingeführt, das entweder leer oder vom Typ `LinearCode` ist:

```
STRUCTURE BasicBlock IS
  InitCode             : SEQ OF Assign;
  IndirectLinkCode     : LinkCode;
  DirectLinkCode       : SEQ OF LinkCode;
  StateCode            : NoLinkCode;
  LinearRepresentation : LinearCode | Nil;
END
```

Die Struktur `LinearCode` enthält die Informationen, die speziell für die Auswertung der linearen Matrizen eines Basissystems erforderlich sind:

```
STRUCTURE LinearCode IS
  StateDerivedAux : SEQ OF Assign;
  InputDerivedAux : SEQ OF Assign;
  A               : SEQ OF MatrixEntry;
  B               : SEQ OF MatrixEntry;
  C               : SEQ OF MatrixEntry;
  D               : SEQ OF MatrixEntry;
END
```

Die Attribute `StateDerivedAux` und `InputDerivedAux` enthalten jeweils die Gleichungen zur Berechnung der Hilfsgrößen, abgeleitet nach den Zustands- bzw. Eingangsgrößen. Die Gleichungen zur Berechnung der Hilfsgrößen selbst

sind schon im nichtlinearen Teil des `BasicBlock` abgelegt und können daraus entnommen werden.

Die eigentlichen Matrizen sind als Listen von Einträgen der Form $(i, j, e_{i,j})$ für $e_{i,j} \neq 0$ abgelegt. Zur Repräsentation eines von Null verschiedenen Matrixeintrags dient die Struktur `MatrixEntry`:

```
STRUCTURE MatrixEntry IS
  Row        : INTEGER;
  Column     : INTEGER;
  Expression : Expr;
END
```

Damit liegen alle Informationen zur Auswertung der linearen Darstellung eines Basissystems vor. Die Koppelmatrizen lassen sich direkt aus den hierarchischen Verkopplungsinformationen ableiten. Diese sind bereits in Form der Struktur `CoupleTree` in der allgemeinen DSC-Repräsentation vorhanden.

Als Beispiel für die symbolisch linearisierte Darstellung eines Systems sei an dieser Stelle auf [Sch94] verwiesen. Dort wird exemplarisch am nichtlinearen Stoßschwinger die nichtlineare und die symbolisch linearisierte Modellbeschreibung in Form von Pseudocode vorgestellt.

7.4 Auswertung linearer Systeme

Nachdem im vorangegangenen Abschnitt die Beschreibung linearer und linearisierter Systeme in DSC vorgestellt wurde, soll nun eine Übersicht über die effiziente Auswertung linearer Systeme durch MPOs gegeben werden.

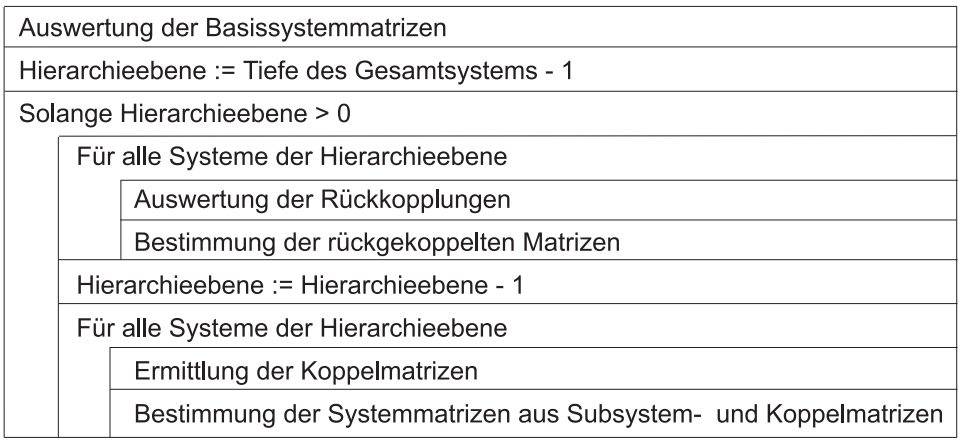


Abbildung 7.3: Ablaufstruktur der Auswertung linearer Systeme

Die grobe Struktur für die Auswertung linearer Systeme ist in Abbildung 7.3 dargestellt. Zunächst werden für alle Basissysteme die Systemmatrizen

ausgewertet. Dann erfolgt “bottom-up“ pro Hierarchieebene die Umsetzung der Koppelbedingungen. Dies geschieht in zwei Schritten. Zunächst werden für jedes Subsystem der Ebene die Rückkopplungen aufgelöst. Im zweiten Schritt werden dann die Systemmatrizen des hierarchischen Systems aus den Verkopplungen der einzelnen Subsysteme untereinander erstellt.

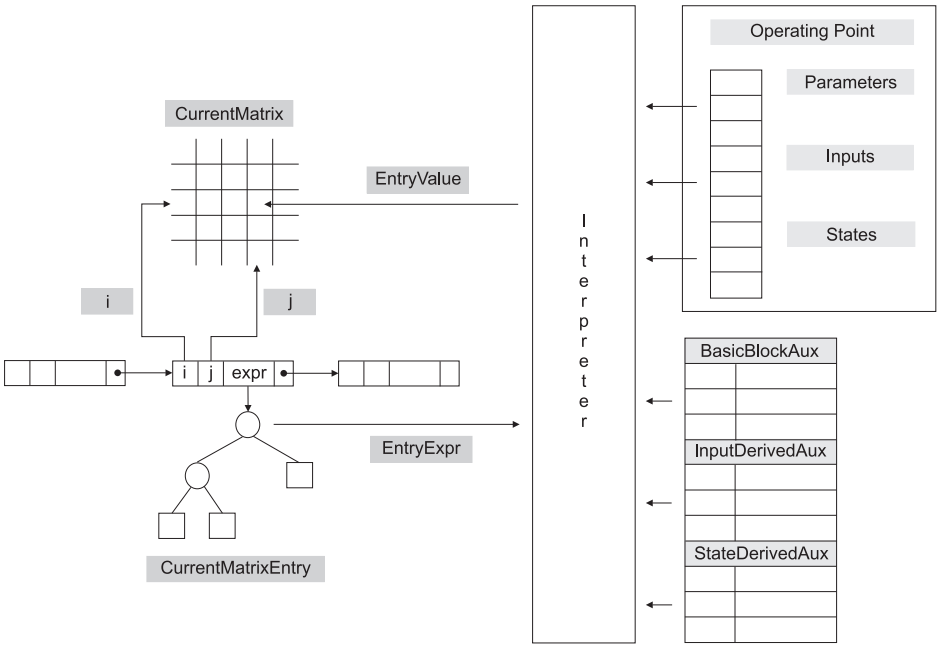


Abbildung 7.4: Auswertung der Basismatrizen eines linearen Systems

Das Prinzip der Auswertung der Basissystemmatrizen ist in Abbildung 7.4 dargestellt. Wie im vorangegangenen Unterkapitel beschrieben, sind die einzelnen symbolischen Matrizen eines Basissystems als Adjazenzlisten abgelegt. Für die Matrizen $\underline{A}$, $\underline{B}$, $\underline{C}$ und $\underline{D}$ werden diese Listen nacheinander vom DSC-Interpretierer-Kern abgearbeitet. Der Zeiger *CurrentMatrixEntry* zeigt dabei auf das jeweils aktuelle Matrixelement. Zur Berechnung des aktuellen Werts wird der einem jeden Matrixeintrag vom Typ `MatrixEntry` in Form der Komponente `Expression` angeheftete Ausdrucksbaum vom DSC-Interpretierer ausgewertet. Zur Besetzung der Blätter des Ausdrucksbaums *EntryExpr* greift der Interpretierer auf den aktuellen Betriebspunkt und die aktuellen Werte der Hilfsgrößen zurück. Der durch Auswertung des Ausdrucksbaums ermittelte aktuelle Wert *EntryValue* des Matrixeintrags kann dann an der entsprechenden Position der Wertematrix abgespeichert werden.

Der Betriebspunkt liegt in einer separaten Tabelle vor, die beim Aufruf der Auswertung eines linearen Systems entsprechend besetzt wird. Die Berechnungsvorschriften für die Hilfsvariablen liegen als Listen von Zuweisungen vor. Diese müssen vor der Auswertung der Matrixeinträge berechnet

werden. Dies kann nach dem gleichen Verfahren wie bei der nichtlinearen Simulation geschehen. Zunächst werden die in der zugehörigen DSC-Struktur `BasicBlock` enthaltenen Hilfsgrößen (*BasicBlockAux*) ausgewertet und die aktuellen Werte abgelegt. Dann kann die Auswertung der abgeleiteten Hilfsgrößen (*InputDerivedAux* und *StateDerivedAux*) erfolgen.

Zur Auflösung der Rückkopplungen wird folgender Algorithmus angewendet:

1. Gegeben:
   (a) Matrizen $\underline{A}_0, \underline{B}_0, \underline{C}_0, \underline{D}_0$ des unverkoppelten Systems S_0
   (b) Menge R der Rückkopplungen des Subsystems S_0 innerhalb des gekoppelten Systems S:
   $$R = \{r = (i,j) \in [1 \ldots dim(\underline{u})] \times [1 \ldots dim(\underline{y})] | u_i = y_j\}$$
2. Bestimmung der Rückkoppelbedingung $\underline{\hat{u}} = \underline{\hat{y}}$
   (a) $dim(\underline{\hat{u}}) = dim(\underline{\hat{y}}) = dim(R) = \#$ Rückkopplungen
   (b) Abbildung von $\underline{\hat{u}}$ auf $\underline{u}$:
   $$\text{map}_u(i) = j \quad \equiv \quad \hat{u}_i = u_j$$
   (c) Abbildung von $\underline{\hat{y}}$ auf $\underline{y}$:
   $$\text{map}_y(i) = k \quad \equiv \quad \hat{y}_i = y_k$$
3. Aufstellung der Gleichung $\underline{\hat{y}} = \underline{C}_{C0}\underline{x} + \underline{D}_{C0}\underline{u} + \underline{D}_{CC}\underline{\hat{u}}$ mit
   (a) $\underline{C}_{(C0)}$
   $$c_{(C0)i,j} = c_{map_y(i),j}$$
   (b) $\underline{D}_{(C0)}$
   $$d_{(C0)i,j} = d_{map_y(i),j}$$
   (c) $\underline{D}_{(CC)}$
   $$d_{(CC)i,j} = d_{map_y(i),map_u(j)}$$
4. Bestimmung von $\underline{K}_C = \underline{I} - \underline{D}_{CC}$
5. Bestimmung von $\underline{C}_C = \underline{K}_C^{-1} \cdot \underline{C}_{C0}$
6. Bestimmung von $\underline{D}_C = \underline{K}_C^{-1} \cdot \underline{D}_{C0}$
7. Aufstellung der Gleichung $\underline{\dot{x}} = \underline{A}_0\underline{x} + \underline{B}_0\underline{u} + \underline{B}_C\underline{\hat{y}}$ mit
   $$\underline{B}_C : b_{(C)i,j} = b_{i,map_u(j)}$$

8. Aufstellung der Gleichung $\underline{y} = \underline{C}_0\underline{x} + \underline{D}_0\underline{u} + \underline{D}_{0C}\hat{\underline{y}}$ mit

$$\underline{D}_{0C} : d_{(0C)i,j} = d_{i,map_u(j)}$$

9. Aufstellung der Matrizen des rückgekoppelten Systems:

   (a) $\underline{A} = \underline{A}_0 + \underline{B}_C\underline{C}_C$
   (b) $\underline{B} = \underline{B}_0 + \underline{B}_C\underline{D}_C$
   (c) $\underline{C} = \underline{C}_0 + \underline{D}_{0C}\underline{C}_C$
   (d) $\underline{D} = \underline{D}_0 + \underline{D}_{0C}\underline{D}_C$

Kapitel 8

Modell-Debugging

Das Fenster in Modell und Rechner

Joachim Lückel

Die Erstellung eines geeigneten Modells nimmt einen großen Teil der Entwicklungszeit eines mechatronischen Systems ein. Aufgrund der hohen Systemkomplexität und der vielen Rückkopplungen im Modell gestaltet sich dabei die Suche nach Fehlern im Modell schwierig. In diesem Kapitel wird vorgestellt, wie auf Basis der verarbeitungsorientierten Modellbeschreibungsform DSC der Ingenieur werkzeugunterstützt besseren Einblick in das logische als auch das numerische Verhalten des Modells erhalten kann. Der hier vorgestellte DSC-Debugger bietet dem Ingenieur ein „Fenster in das Modell und den Rechner“, mit dem er Einblick in das modellierte Systemverhalten einschließlich der numerischen Aspekte erhält.

8.1 Anforderungsprofil

Unter *Debugging* versteht man allgemein den Prozess des Aufspürens und Eliminierens von Fehlern in Computerprogrammen [Wis94]. Dieser Begriff aus der Softwaretechnik lässt sich auch auf die Fehlersuche bei Modellen mechatronischer Systeme anwenden. Die im Kontext der Mechatronikentwicklung auftretenden Fehler lassen sich den nachfolgenden drei Kategorien zuordnen:

Generator-Fehler Die vom Ingenieur auf physikalisch-topologischer Ebene modellierten Systemkomponenten werden typischerweise unter Einsatz von Generatoren in eine verarbeitungsorientierte Beschreibungsform übersetzt. Diese Übersetzungsschritte können Fehler beinhalten. Beispiele hierfür sind eine fehlerhafte Erzeugung der Verkopplung auf

Basisblockebene oder eine falsche Berechnung der Auswertereihenfolge.

Logische Fehler Logische Fehler sind Fehler, die der Ingenieur beim Erstellen des Modells macht, indem er falsche Modellierungskonstrukte verwendet. Beispiele für Fehler logischer Natur sind die Angabe eines falschen Terms in einer Gleichung oder die Verknüpfung falscher Ein- und Ausgangssignale.

Numerische Fehler Die Auswertung mechatronischer Modelle erfolgt in der Regel numerisch. Abhängig von den Systemeigenschaften und den gewählten numerischen Verfahren kann die Auswertung eines Systems zu verfälschten bis völlig falschen Ergebnissen führen. Beispiele für numerische Fehler sind Wertebereichsunter- und überschreitungen oder eine Division durch Null.

Aus den aufgeführten Fehlerkategorien lassen sich die folgenden Anforderungen an einen Modell-Debugger für mechatronische Systeme ableiten:

Fachspezifische Sicht Aus Anwendersicht ist die fachspezifische/physikalische Struktur oder die mathematische Modellbeschreibung die Modellsicht, auf der er die Fehlersuche durchführen möchte.

Zugriff auf Gesamtsystem Bei der Fehlersuche besteht jederzeit Zugriff auf alle Komponenten des Systems. Aufgrund der Rückkopplungen in mechatronischen Systemen ist eine auf einzelne Subsysteme begrenzte Fehlersuche nicht ausreichend.

Granularität Die schrittweise Ausführung auf Ebene einzelner Variablen ist möglich.

Haltepunkte *(breakpoints)* Die Modellauswertung wird an einer vorher definierten Stelle und/oder beim Eintreten einer spezifizierten Bedingung unterbrochen.

Überwachung von Variablen *(watchpoints)* Die Modellauswertung wird unterbrochen, wenn auf vorher definierte Variablen zugegriffen wird.

Überprüfungspunkte *(checkpoints)* An bestimmten Stellen der Modellauswertung wird der komplette Systemzustand gespeichert. Zu einem späteren Zeitpunkt kann das System dann direkt in diesen Zustand zurückgeführt werden.

Manipulation zur Laufzeit Systemgrößen und Gleichungen sind zur Laufzeit vom Anwender veränderbar. So können Auswirkungen einer Modellmodifikation auf das dynamische Systemverhalten sofort nachvollzogen werden.

8.2 Herstellung von Quellcodereferenzen

Bei der Fehlersuche in mechatronischen Modellen ist der Bezug zur originalen fachspezifisch/topologischen oder mathematischen Modellbeschreibung von zentraler Bedeutung. Eine verarbeitungsorientierte Beschreibungsform muss daher Mechanismen bieten, die für jede Anweisung auf verarbeitungsorientierter Beschreibungsebene einen Bezug zu dem entsprechenden Konstrukt auf den höheren Beschreibungsebenen herstellt.

Mechatronic Processing Objects stellen den Bezug zur fachspezifisch/topologischen oder mathematischen Modellbeschreibung durch Verwendung von annotierten Referenzen in der verarbeitungsorientierte Modellbeschreibungsform DSC her. Dazu enthält die DSC-Spezifikation für jede Variable ein Attribut *DebugInfo* vom Strukturtyp *References*:

```
STRUCTURE Param IS REFERENCED
  ...
  DebugInfo : References | Nil;     -- Debugging-Informationen
END

STRUCTURE References IS
  Coordinates : SEQ OF Coordinate;  -- Quellcodekoordinaten
  Condition   : Expr;               -- Bedingung für Anzeige
  Generated   : INTEGER;            -- wahr: generierte Größe
  Origin      : SEQ OF Ident;       -- Quelle fuer gen. Größen
  Extension   : STRING;             -- Erweiterungen
END
```

Die Codierung der Quellcodekoordinaten geschieht mit Hilfe des Attributs *Coordinates* vom Strukturtyp *Coordinate*:

```
STRUCTURE Coordinate IS
  SrcFileName : STRING;             -- Name der Quellcodedatei
  StartLine   : INTEGER;            -- 1. Zeichen, Zeile
  StartColumn : INTEGER;            -- 1. Zeichen, Spalte
  NextLine    : INTEGER;            -- Nächstes Token, Zeile
  NextColumn  : INTEGER;            -- Nächstes Token, Spalte
END
```

Ist der zugehörige Quellcode auf Fragmente an verschiedenen Stellen des Quelltextes verteilt, so ist dies ebenfalls darstellbar, da eine Liste von Quellcodekoordinaten angegeben werden kann.

Da es sich bei der DSC-Repräsentation um eine Single-Assignment-Form handelt, bei der die Abhängigkeit von Variablen rein funktional abgebildet ist, müssen bei der Referenzierung des Quellcodes folgende Besonderheiten berücksichtigt werden:

- Abbildung von Mehrfachzuweisungen
  Werden im Quellcode Variablen mehrfach zugewiesen, so ist bei einer Single-Assignment-Form für jede Zuweisung eine eigene Variable erforderlich. Diese für die DSC-Repräsentation erforderlichen, zusätzlich generierten Variablen sind durch den Wert 1 des Attributs *Generated* gekennzeichnet. In diesem Fall enthält das Attribut *Origin* die zugehörige Originalgröße. Diese Sequenz darf maximal ein Element enthalten.

- Abbildung von Kontrollstrukturen
  Enthält der Quellcode eine Kontrollstruktur, so muss in DSC für die Umsetzung der Kontrollbedingung eine zusätzliche Variable generiert werden, die die Kontrollbedingung repräsentiert. In diesem Fall erhält das Attribut *Generated* ebenfalls den Wert 1.

- Bedingte Ausführung von Quellcode
  Enthält der Quellcode bedingte Anweisungen, so dürfen diese beim schrittweisen Debugging nur dann angezeigt werden, wenn die entsprechende Bedingung zur Ausführung der Anweisung erfüllt ist. Diesem Zweck dient das Attribut *Condition*. Die Berechnung der Variable wird nur dann angezeigt, wenn der im Attribut *Condition* angegebene Ausdruck wahr ist. Dieser Ausdruck enthält üblicherweise die Und-Verknüpfung der übergeordneten Ausführungsbedingungen.

8.3 Der DSC-Debugger

Wurden in den vorangegangenen Unterkapiteln die Anforderungen und Grundlagen des Debugging von Mechatronic Processing Objects beschrieben, so soll hier nun das eigentliche Werkzeug zur Unterstützung bei der Suche nach Modellfehlern vorgestellt werden [HO99].

Grundprinzip des Modell-Debuggers ist die interpretative Ausführung der DSC-Repräsentation eines MPO [Vül96]. Die Architektur ist in Abbildung 8.1 dargestellt. Sie ist viergeteilt und gliedert sich in einen Bedienprozess (*user interaction process*), den eigentlichen Debugging-Prozess (*debugging process*), einen Visualisierungsprozess (*visualization process*) zur zwei- oder dreidimensionalen Darstellung des Zeitverhaltens und eine optionale Anbindung des Modellierungswerkzeugs auf fachspezifischer oder mathematischer Modellierungsebene.

Zentraler Bestandteil des Modell-Debuggers ist der Debugging-Prozess (*debugging process*). Er besteht aus einem MPO, das üblicherweise interpretativ die Modellgleichungen auswertet, sowie einer kontrollierenden Instanz (*MPO debug control*). Die kontrollierende Instanz erzeugt das MPO und steuert dessen Ausführung. Ausserdem stellt die kontrollierende Instanz die Verbindung zum Bedienprozess her.

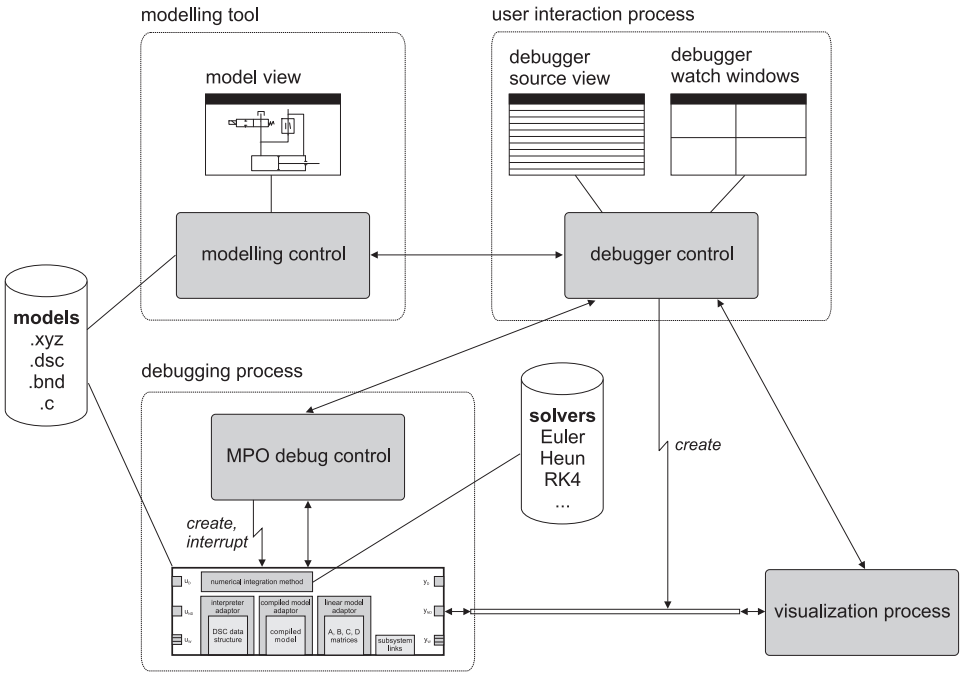


Abbildung 8.1: Architektur des DSC-Debuggers

Der Bedienprozess (*user interaction process*) ist für die Interaktion mit dem Benutzer zuständig. Er ist dreigeteilt und besteht aus einer Visualisierungseinheit für die Modellbeschreibungsquellen (*debugger source view*), einer Anzeigemöglichkeit für den Systemzustand (*debugger watch windows*) sowie einer kontrollierenden Instanz (*debugger control*). Die kontrollierende Instanz des Bedienprozesses ist sowohl für die Koordination der zu visualisierenden Quellen und Daten als auch für die Steuerung des eigentlichen Debugging-Prozesses zuständig.

Die Visualisierungseinheit für die Modellbeschreibungsquellen ist schematisch in Abbildung 8.2 dargestellt. Sie bietet dem Ingenieur drei unterschiedliche Sichtweisen auf das mechatronische System:

DSC-Sicht Der Modell-Debugger arbeitet auf der Ebene der DSC-Beschreibung. Es wird das jeweils bearbeitete Fragment des DSC-Codes angezeigt. Der Modell-Debugger bietet hier die Möglichkeit, die interpretative Auswertung der Koppelgleichungen sowohl hierarchisch als auch auf Ebene der Basisblockkopplungen durchzuführen.

Meta-Sicht Im DSC-Code erscheinen die Systemnamen und Systemgrößen in codierter Form. In der Meta-Sicht werden die DSC-Bezeichner durch ihre Originalnamen aus der ursprünglichen Modellbeschreibung ersetzt. Dies ermöglicht eine bessere Lesbarkeit der verarbeitungsorientierten Beschreibungsform für den Anwender. Ansonsten ist diese

Sichtweise identisch mit der DSC-Sicht.

Quell-Sicht Der Modell-Debugger arbeitet auf der Ebene der ursprünglichen Modellbeschreibung. Bei der interpretativen Auswertung des DSC-Codes werden die korrespondierenden Fragmente des Quellcodes angezeigt.

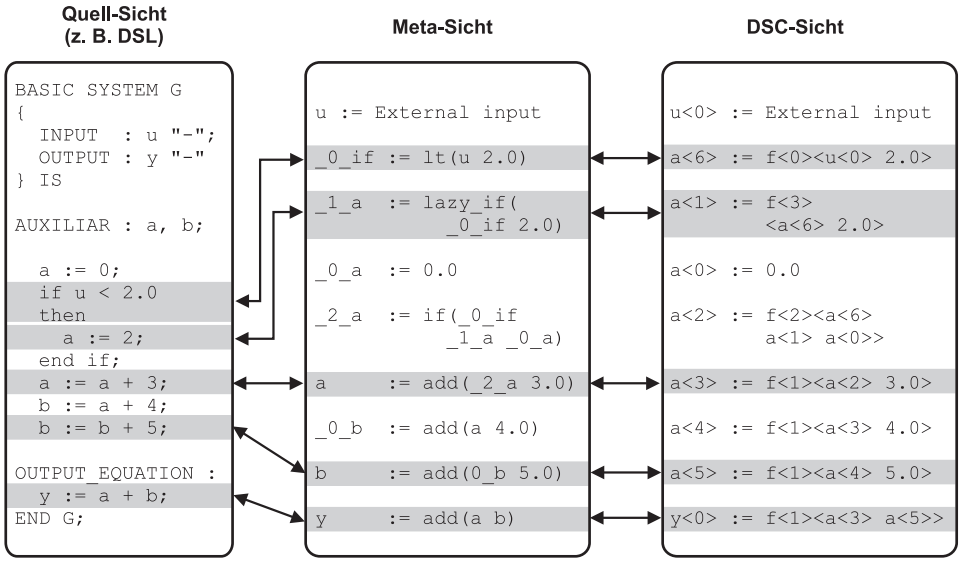


Abbildung 8.2: Sichtweisen des DSC-Debuggers

Für eine optimale Visualisierung auf Ebene der fachspezifisch/topologischen oder mathematischen Modellbeschreibung bietet der Modell-Debugger zusätzlich eine offene Schnittstelle an, mit der optional Modellierungswerkzeuge angebunden werden können. Dazu reicht die kontrollierende Instanz des Bedienprozesses die vom Debugging-Prozess erhaltenen Informationen über die aktuell auszuwertende Systemgleichung an das Modellierungswerkzeug weiter. Diese Debug-Informationen werden in Form der DSC-Struktur *References* übermittelt, die sowohl die Quellcodekoordinaten als auch werkzeugspezifische, erweiterte Informationen enthalten kann. Mit Hilfe dieser Debug-Informationen kann das Modellierungswerkzeug dann das korrespondierende Modellfragment bzw. den entsprechenden Modellzustand anzeigen.

Die Visualisierung des zeitlichen Verhaltens des Systems ist Aufgabe des Visualisierungsprozesses (*visualization process*). Dabei kann es sich sowohl um eine zweidimensionale Darstellung der zu betrachtenden Größen in Form von Zeitschrieben als auch um eine dreidimensionale Animation des modellierten Systems z. B. bei mechanischen Mehrkörpersystemen handeln. Die Übermittlung der Zeitdaten vom MPO zum Visualisierungsprozess geschieht über eine spezielle Kommunikationsverbindung, die von der kontrollierenden Instanz des Bedienprozesses angelegt wird.

Die vorgestellte Architektur des Modell-Debuggers ist generisch und lässt sich auf unterschiedlichste Betriebssysteme abbilden. Unter UNIX-Betriebssystemen können alle Prozesse als gewöhnliche UNIX-Prozesse implementiert werden, die über UNIX-Pipes kommunizieren. Die Anbindung der Visualisierung des Zeitverhaltens an das MPO kann über eine UNIX-Socket-Verbindung mit FIFO-Charakter realisiert werden.

Bei der Implementierung des Modell-Debuggers unter Windows ist das MPO innerhalb des Debugging-Prozesses als separater Thread implementiert. Die Nebenläufigkeit des Bedienprozesses und des Visualisierungsprozesses ist durch die Ausnutzung der Event- und Callback-Mechanismen von Windows gewährleistet. Ein Beispiel für die Benutzeroberfläche des DSC-Debuggers unter Windows ist in Abbildung 8.3 dargestellt.

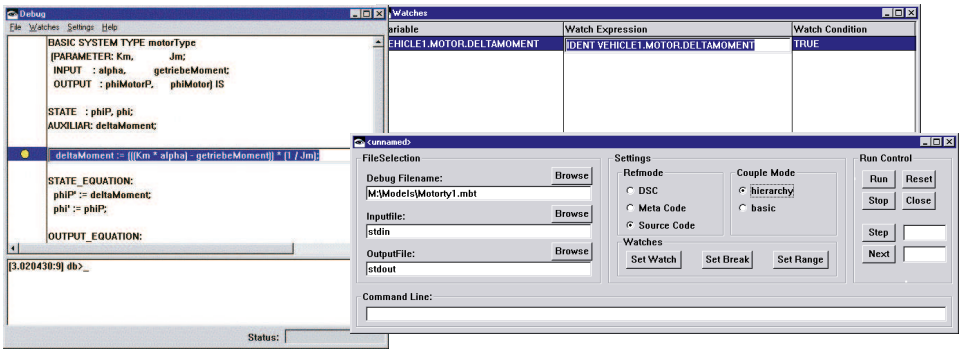


Abbildung 8.3: Benutzeroberfläche des DSC-Debuggers

Kapitel 9

Anwendungsbeispiele

Wurden in den vorangegangenen Kapiteln die Grundlagen und Werkzeugumgebungen für Mechatronic Processing Objects als verarbeitungsorientierte Modellrepräsentation mechatronischer Systeme beschrieben, so soll in diesem Kapitel der Nachweis für die Einsatzfähigkeit des hier vorgestellten Konzepts in der Praxis erbracht werden. Im Folgenden werden drei Anwendungsbeispiele für komplexe mechatronische Systeme vorgestellt, deren Entwicklung unter Einsatz der Mechatronic Processing Objects und der verarbeitungsorientierten Beschreibungsform DSC durchgeführt wurde. An allen drei Projekten war das Mechatronik Laboratorium Paderborn maßgeblich beteiligt.

Erstes Anwendungsbeispiel ist ein serieller Hybridantrieb, dessen Entwicklung vom rechnergestützten Entwurf bis zur Hardware-in-the-Loop-Realisierung auf verteilter Echtzeit-Hardware durchgeführt wurde. Schwerpunktaspekt dieses Anwendungsbeispiels ist die Einbindung von Teilmodellen aus unterschiedlichsten Modellierungswerkzeugen.

Zweites Anwendungsbeispiel ist ein dezentrales Kreuzungsmanagement. Es dient zur Veranschaulichung der Modellierung und Verarbeitung vernetzter intelligenter mechatronischer Systeme. Hauptaugenmerk liegt bei diesem Beispiel auf der Abbildung gemischt diskret-kontinuierlicher Systeme und auf der Umsetzung von dynamischen Systemstrukturen.

Drittes Anwendungsbeispiel ist der Educational Truck (EduTruck), ein komplexes Modell eines Nutzfahrzeugs. Das komplexe Mehrkörpermodell bildet die Ausgangsbasis für die Analyse der Auswirkungen von konstruktiven Änderungen auf das Fahrverhalten des Fahrzeugs. Schwerpunkt der Betrachtung dieses Beispiels ist die verteilte Offline-Simulation komplexer Modelle unter Einsatz automatischer Lastverteilungsverfahren.

Eine Übersicht über die im Rahmen der einzelnen Anwendungsbeispiele validierten Anforderungen an eine verarbeitungsorientierte Modellrepräsentation aus der Aufstellung in Kap. 3.1 ist in Tabelle 9.1 dargestellt. In der Tabelle ist auch festgehalten, bis zu welchem Grad die einzelnen Anwendun-

gen in die Praxis umgesetzt wurden.

		9.1 Serieller Hybridantrieb	9.2 Dezentrales Kreuzungsmanagement	9.3 Educational Truck
Validierte Anforderungen (s. Kap. 3.1)				
MOD.1	Unterstützung modular-hierarchischer Strukturen	+	+	+
MOD.2	Homogene Integration unterschiedlicher Fachdisziplinen	+		
MOD.3	Transparente Anknüpfung an ursprüngliche Modellspezifikation	+		
MOD.4	Schnelle Umsetzbarkeit von Modelländerungen	+	+	+
MOD.5	Unterstützung bei der Suche nach Modellierungsfehlern	+	+	+
SYN.1	Abbildung linearer Systeme als Matrizen			+
SYN.2	Verkopplung symbolischer, linearer Teilsysteme			+
REA.1	Schlanke und portable Software-Architektur	+		
REA.2	Echtzeitfähigkeit	+		
REA.3	Codeeffizienz	+		+
REA.4	Geringe Hardwareanforderungen	+		
REA.5	Unterstützung der Parallelverarbeitung	+		+
REA.6	Automatische Codegenerierung	+	+	+
REA.7	Anbindung von Sensorik/Aktorik	+		
SWT.1	Domänenübergreifende Systemkomposition	+		
SWT.2	Modulare Erweiterbarkeit	+	+	
SWT.3	Schnittstelle zwischen Modellbeschreibung und Weiterverarbeitung	+	+	+
Praxisbezug				
Einsatz MPO/DSC-basierter Entwurfswerkzeuge		+	+	+
Einsatz MPO/DSC am HIL-Prüfstand		+		
Anwendungsbeispiel als Produkt verfügbar				+

Tabelle 9.1: Übersicht Anwendungsbeispiele

9.1 Serieller Hybridantrieb

Der Antrieb eines Kraftfahrzeugs wird als Hybridantrieb bezeichnet, wenn er eine Kombination von mindestens zwei Antriebsarten darstellt. Das hier vorgestellte Anwendungsbeispiel des seriellen Hybridantriebs [Wä01] ist im Rahmen des Forschungsprojektes METRO (*Einsatz massiv paralleler Rechner für den Entwurf und die Realisierung komplexer mechatronischer Systeme*) , das vom Bundesministerium für Bildung und Forschung (BMBF) gefördert wurde, entstanden. Besonderheit dieses Anwendungsbeispiels ist die Tatsache, dass von den unterschiedlichen Projektpartnern Teilmodel-

le des Hybridfahrzeugs in unterschiedlichsten Modellierungswerkzeugen erstellt wurden und diese für die Offline-Simulation, die Systemoptimierung und die Hardware-in-the-Loop-Simulation zu einem Gesamtsystem verknüpft werden mussten.

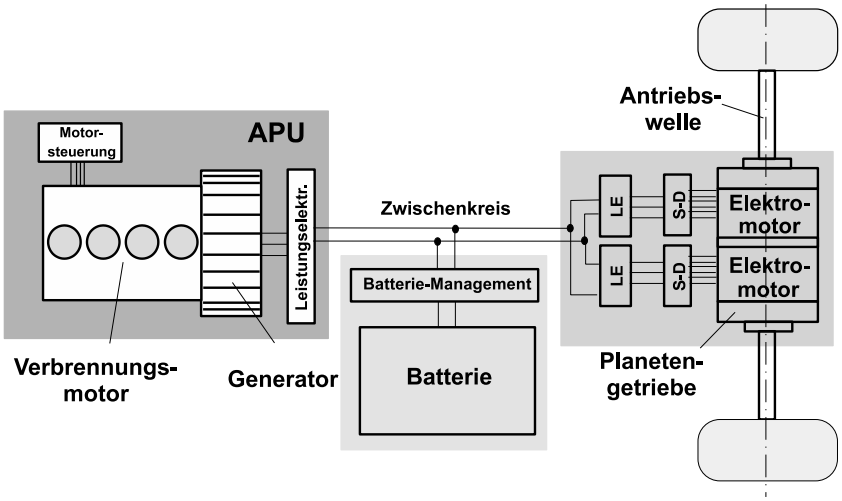


Abbildung 9.1: Struktur eines seriellen Hybridantriebs

Die Struktur des seriellen Hybridantriebs ist in Abbildung 9.1 dargestellt. Ein serieller Hybridantrieb besteht im Wesentlichen aus den Komponenten *Traktionsantrieb*, *Speicher* und der *Einheit zur Erzeugung der elektrischen Energie (APU)*. Der *Traktionsantrieb* wird durch einen sogenannten Sachs-Tandemmotor realisiert. Der Tandemmotor besteht aus zwei elektrisch und antriebstechnisch völlig getrennten Maschinen, die an einer gemeinsamen Halterung befestigt sind und jeweils ein Rad antreiben. Die *APU* wird durch einen Verbrennungsmotor, gekoppelt mit einem Generator, umgesetzt. Als *Speicher* für die von der APU erzeugten Energie fungiert eine Batterie.

Der wesentliche Vorteil des seriellen Hybridantriebs ist die mechanische Entkopplung des Traktionsantriebs von der Energieerzeugungseinheit. Der Verbrennungsmotor kann mit konstanter Drehzahl mit optimalem Wirkungsgrad unabhängig von der gerade geforderten Leistung arbeiten. Überschüssige Energie wird in der Batterie gespeichert, bei hoher Leistungsanforderung kann zusätzliche Energie aus der Batterie an die Elektromotoren abgegeben werden. Bei Bedarf kann in emissionsarmen Zonen (z. B. im Innenstadtbereich) mit reinem Batteriebetrieb gefahren werden. Des Weiteren kann über einen geregelten Einzelradantrieb das Fahrverhalten verbessert werden.

Zum effizienten, umweltschonenden und komfortablen Betrieb des seriellen Hybridantriebs bedarf es eines an die Fahrerwünsche angepassten Fahrzeugmanagementsystems und optimierter Regelungsstrukturen. Aufgrund

der Komplexität ist hier eine modellgestützte Systemanalyse und Optimierung einschließlich Hardware-in-the-Loop-Simulation zur Verifizierung der Ergebnisse zwingend erforderlich.

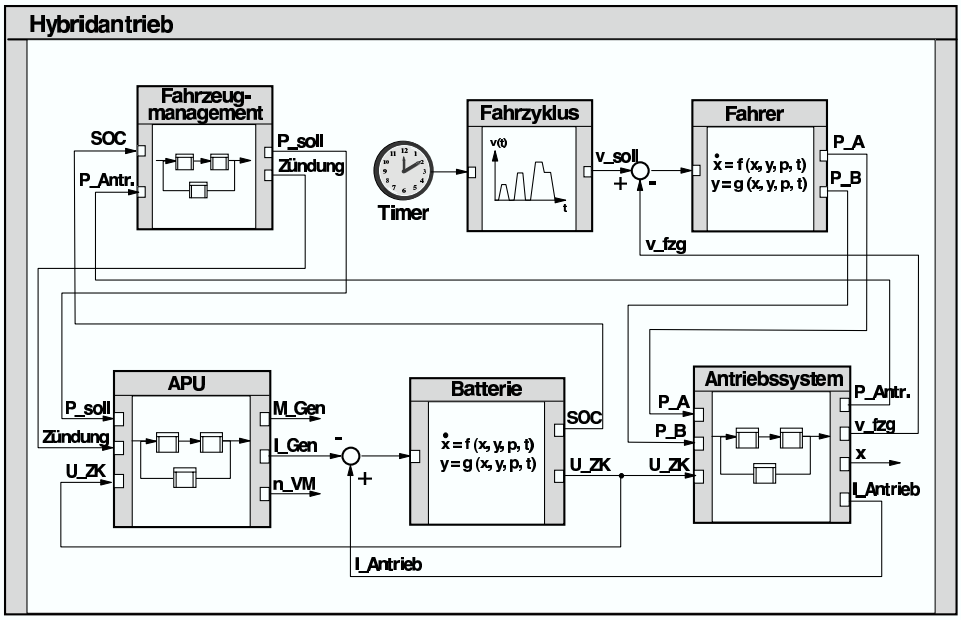


Abbildung 9.2: Struktur des Gesamtsystems Hybridantrieb

Die Struktur des Modells für das Gesamtsystem Hybridantrieb ist in Abbildung 9.2 dargestellt. Die Modelle für die einzelnen Komponenten sind dabei von unterschiedlichen Projektpartnern in verschiedenen Modellierungswerkzeugen erstellt worden (s. Abbildung 9.3). Die zentralen Bestandteile des Verbrennungsmotors (*drehmoment, kraftstoffPfad, luftpfad*) sind in Matrix/X modelliert. Die Modelle für Generator, Batterie und Tandemmotor sind mit ASCET-SD erstellt worden. Die Fahrzeugdynamik ist mit dem Modellierungswerkzeug alaska für Mehrkörpersysteme beschrieben worden. Zur Beschreibung des Fahrzeugmanagements wurden erweiterte Prädikat-Transitionsnetze verwendet, modelliert im Werkzeug SEA. Die Gesamtsystemtopologie sowie die übrigen Komponenten sind in Objective-DSS spezifiziert.

Zur Arbeit mit dem Gesamtsystem wurde die Entwurfsumgebung CAMeL-View eingesetzt. Die nicht in Objective-DSS modellierten Komponenten werden über eine spezielle Objective-DSS-Schnittstellenbeschreibung in das Gesamtmodell eingebunden. Aus CAMEL-View heraus wird dann das Gesamtsystem einschließlich aller Komponenten über eine mathematische Modellrepräsentation in die verarbeitungsorientierte Modellbeschreibungsform DSC transformiert. Dabei wird, abhängig vom jeweiligen Modellierungswerkzeug, die entsprechende DSC-Generierung angestoßen.

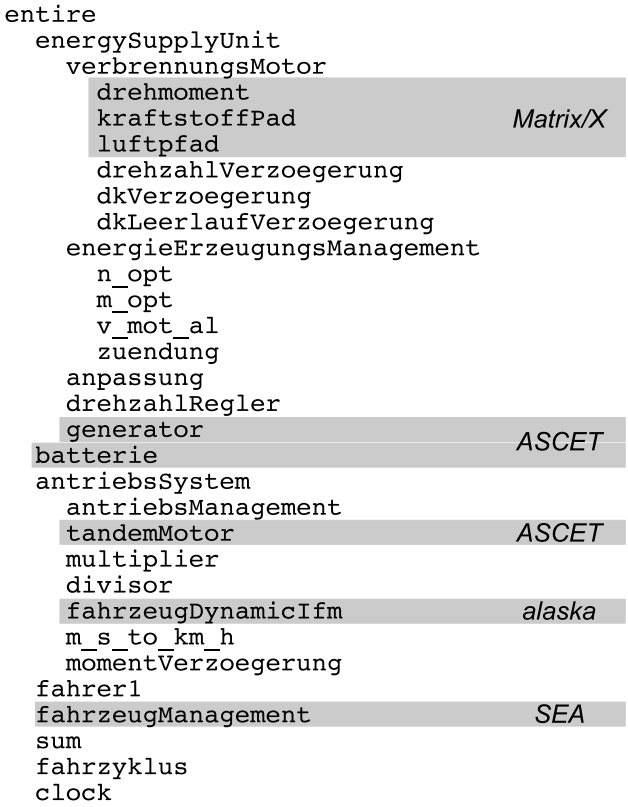


Abbildung 9.3: Hierarchische Komponentenstruktur des seriellen Hybridantriebs

Da für die Modellierungswerkzeuge MatrixX, ASCET-SD, alaska und SEA eine direkte Generierung des Systemverhaltens als DSC-Code nicht zur Verfügung stand, musste auf bereits existierende Exportmöglichkeiten dieser Modellierungswerkzeuge zurückgegriffen werden. Da alle Modellierungswerkzeuge den Export des Systemverhaltens in Form von C-Programmcode anbieten, wurde der Ansatz gewählt, die Systemgleichungen direkt als C-Code-Fragmente einzubinden. Nachfolgend soll die Vorgehensweise beispielhaft anhand der in ASCET-SD modellierten Komponente des Tandemmotors aufgezeigt werden.

Der von ASCET-SD generierte modellspezifische C-Code besteht für ein Teilsystem mit dem Namen *Modellname* aus den folgenden Unterprogrammen:

- **int ascInitModel*Modellname*(double dT)**
  Dieses Unterprogramm dient zur Initialisierung des Modells. Als Eingangsgröße wird die Schrittweite dT übergeben.

- **int ascComputeOutputEquations*Modellname*(double *in,**

double *out)
Diese Prozedur dient zur Berechnung der Ausgangsgrößen in Abhängigkeit von der aktuellen internen Systemzeit und dem Eingangsvektor *in*. Innerhalb der Prozedur erfolgt die Besetzung des Ausgangsvektors *out* mit den aktuellen Werten der Ausgangsgrößen. Ein Weiterschalten der internen Systemzeit findet beim Aufruf dieser Prozedur nicht statt.

- **int ascComputeStateEquations*Modellname*(double *in)**
  Durch den Aufruf dieses Unterprogramms wird die interne Systemzeit um dT weitergeschaltet und der interne Systemzustand durch Auswerten der Zustandsgleichungen und durch numerische Integration aktualisiert.

Um den von ASCET-SD generierten C-Code unter DSC einbinden zu können, ist noch die Bereitstellung eines C-Rahmens für die entsprechende Komponente erforderlich. Der C-Rahmen stellt die folgenden Funktionen bereit:

- **int step*Modellname*(int timeStep)**
  Diese Funktion wertet die Zustandsgleichungen aus und schaltet die interne Systemzeit um dT weiter durch Aufruf der ASCET-SD-Funktion *ascComputeStateEquations*. Über den Parameter *timeStep* wird dabei sichergestellt, dass bei Mehrschritt-Integrationsverfahren die interne Systemzeit nur einmal weitergeschaltet wird. Außerdem wird beim Zeitschritt 0 anstatt einer Zustandsweiterschaltung die ASCET-SD-Funktion *ascInitModel* zur Initialisierung der Komponente aufgerufen. Des Weiteren dient diese Funktion zu Berechnung der Nichtdurchgriffsausgänge.

- **int setDirectInputOf*Modellname*(int timeStep, *direct inputs*)**
  Diese Funktion dient zum Setzen der Durchgriffseingänge der Komponente. Der Einsatz zusätzlicher Subroutinen zum Setzen von Eingangsgrößen ist erforderlich, da ASCET-SD bezüglich eines Subsystems keine Unterscheidung zwischen Durchgriffs- und Nichtdurchgriffsgrößen anbietet.

- **int setIndirectInputOf*Modellname*(int timeStep, *indirect inputs*)**
  Diese Funktion dient zum Setzen der Nichtdurchgriffseingänge einer Komponente.

- **RealT getOutputOf*Modellname*(int timeStep, int i)**
  Diese Funktion dient zum Zugriff auf einzelne Ausgangsgrößen. Sie stellt den Zugang zu einzelnen Größen des ASCET-SD-Ausgangsvektors über den Index i dar.

Sowohl der C-Rahmen als auch die DSC-Spezifikation des ASCET-SD-Modells des Tandemmotors können aus der nachfolgenden Objective-DSS-

Schnittstellenbeschreibung für ASCET-SD-Teilmodelle automatisch generiert werden:

```
CAscetOdss named: TandemType.
  input:            #( U_Bat
                       w_Wheel
                       M_DriveTarget ) on: ScalarOdss;
  output:           #( I_Drive
                       M_Drive
                       EfficyDrive
                       T_Cu ) on: ScalarOdss;
  directLinks:      #( (I_Drive U_Bat)
                       (I_Drive w_Wheel)
                       (M_Drive U_Bat)
                       (M_Drive w_Wheel)
                       (EfficyDrive U_Bat)
                       (EfficyDrive w_Wheel) );
  headerFiles:      prot.h;
                    asc_proc.h;
                    asc_macs.h;
                    asccsym.h;
                    ccg.h;
                    ccg_proc.h;
                    ccg_macs.h;
  codeFiles:        asc_proc.c;
                    ccg_proc.c;
  modelHeaderFile:  tandem.h;
  modelCodeFile:    tandem.c;
  initFunction:     ascInitModelTandem;
  evalFunction:     ascComputeStateEquationsTandem;
  outputFunction:   ascComputeOutputEquationsTandem;
end.
```

Neben der Spezifikation der Schnittstellengrößen, der C-Subroutinen sowie der erforderlichen Programmdateien enthält die Objective-DSS-Beschreibung eine explizite Aufzählung der Durchgriffspfade. Aus diesen Informationen kann die zugehörige DSC-Spezifikation automatisch erzeugt werden. Für das Beispiel des Tandemmotors hat der DSC-Code dann die folgende Struktur, dargestellt in der Notation der Meta-Sicht des DSC-Debuggers:

```
...
-- Functions
<
  "stepTandem"
  "setDirectInputOfTandem"
  "setIndirectInputOfTandem"
  "getOutputOfTandem"
  "ADD"
```

```
>
<>
<
  "TIME_STEP"           INTEGER -1
  "DIRECT_INPUT_SET"    INTEGER  0
  "INDIRECT_INPUT_SET"  INTEGER  0
>
<>
-- DirectLinks
<
  I_DRIVE      U_BAT
  M_DRIVE      U_BAT
  EFFICYDRIVE  U_BAT
  I_DRIVE      W_WHEEL
  M_DRIVE      W_WHEEL
  EFFICYDRIVE  W_WHEEL
>
...
-- BasicBlocks
<
  <>
  <
    TIME_STEP  stepTandem<ADD<TIME_STEP0 1>>
  ><
    T_CU       getOutputOfTandem<TIME_STEP 3>
  ><
    <
      DIRECT_INPUT_SET  setDirectInputOfTandem<
                         TIME_STEP U_BAT W_WHEEL>
    ><
      I_DRIVE           getOutputOfTandem<DIRECT_INPUT_SET 0>
      M_DRIVE           getOutputOfTandem<DIRECT_INPUT_SET 1>
      EFFICYDRIVE       getOutputOfTandem<DIRECT_INPUT_SET 2>
    >
  >
  <
    INDIRECT_INPUT_SET  setIndirectInputOfTandem<
                         TIME_STEP M_DRIVETARGET>
  >
  <>
>
```

Abschließend bleibt bei diesem Anwendungsbeispiel noch anzumerken, dass die dargestellte Einbindung von anderen Modellierungswerkzeugen unter Zuhilfenahme von generierten C-Code-Fragmenten ein gangbarer Weg ist, der nur sehr geringen Aufwand erfordert. Allerdings lassen sich bei diesem Ansatz die Vorteile einer verarbeitungsorientierten Modellrepräsentation, die einen transparenten Zugriff auf das Gesamtsystem bis hinunter zu den einzelnen Systemgleichungen der beteiligten Komponenten ermöglicht,

nur zum Teil ausnutzen. Die eingebundenen C-Code-Fragmente haben den Charakter einer „Black Box“. Innerhalb des C-Codes ist weder eine vom Verkopplungskontext abhängige Auswertereihenfolge noch ein Modell-Debugging möglich. Die vorgestellte Modellierungsumgebung kann in der Hinsicht noch optimiert werden, dass die Modellierungswerkzeuge MatrixX, ASCET-SD, alaska und SEA jeweils um ein DSC-Backend erweitert werden, das auch die Beschreibung des inneren Systemverhaltens als reine DSC-Spezifikation erzeugt.

9.2 Dezentrales Kreuzungsmanagement

Dezentrales Kreuzungsmanagement ist ein Anwendungsbeispiel für die Modellierung und Verarbeitung vernetzter intelligenter mechatronischer Systeme. Das hier beschriebene Beispiel ist im Rahmen des Sonderforschungsbereichs 376 „Massive Parallelität — Algorithmen, Entwurfsmethoden, Anwendungen“ an der Universität Paderborn entstanden.

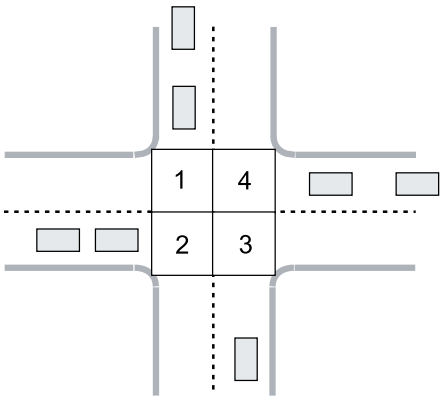


Abbildung 9.4: Kreuzungsmanagement

Hauptaugenmerk bei dem hier vorgestellten Modell ist die Abbildung gemischt diskret-kontinuierlicher Systeme und die Umsetzung von dynamischen Systemstrukturen. Dazu wird zur besseren Verständlichkeit im Folgenden ein stark vereinfachtes Modell verwendet. Im Rahmen des Sonderforschungsbereichs sind noch weit komplexere Modelle entwickelt worden.

Grundprinzip des hier vorgestellten dezentralen Kreuzungsmanagements sind autonome Fahrzeuge, die selbstorganisierend Kolonnen bilden und die Kreuzung überqueren. Optimierungsziel kann dabei sowohl maximaler Durchsatz als auch minimale Wartezeit der Fahrzeuge sein. Wie in Abbildung 9.4 dargestellt, wird zur Regelung des Verkehrs der Kreuzungsbereich dazu in vier Sektoren unterteilt. Jeder dieser Sektoren darf zu einem Zeitpunkt von maximal einem Fahrzeug belegt sein.

Die Struktur des Modells der verkehrenden Fahrzeuge ist in Abbildung 9.5 dargestellt. Die für dieses Anwendungsszenario relevanten Komponenten sind dabei die Fahrzeugdynamik, die Ermittlung der Sollgeschwindigkeit, das Geschwindigkeitsmanagement und das Freigabemanagement der Kreuzungssektoren.

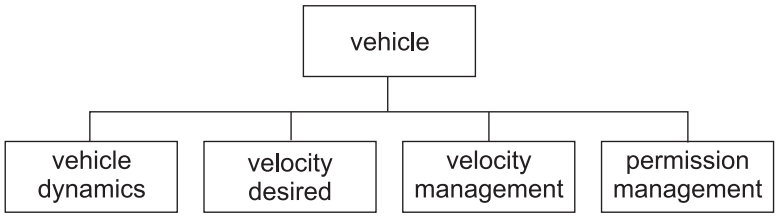


Abbildung 9.5: Fahrzeugmodell beim Kreuzungsmanagement

Als Beispiel für die Abbildung diskreter, ereignisbasierter Systemanteile ist das Freigabemanagement der Kreuzungssektoren als Objective-DSS-Modell `permissionManagementType` nachfolgend aufgelistet. Erreicht ein Fahrzeug die Kreuzung, so fordert es über den externen Event `req_token` ein Freigabetoken für das Passieren der einzelnen Kreuzungssektoren an. Das Freigabetoken wird über alle am Kreuzungsverkehr beteiligten Fahrzeuge durchgereicht. Das Token enthält für jeden der vier Kreuzungssektoren einen Semaphor, der die Belegung des entsprechenden Sektors repräsentiert. Sind alle für das Passieren der Kreuzung erforderlichen Sektoren frei, so reserviert das Fahrzeug die einzelnen Sektoren im Freigabetoken und gibt das modifizierte Token über den externen Event `rel_token` weiter. Das Fahrzeug kann nun die Kreuzung kollisionsfrei überqueren. Sind nicht alle erforderlichen Sektoren frei, so wird das Token unverändert weitergegeben. Sobald ein Fahrzeug einen Sektor verlassen hat, fordert es erneut das Freigabetoken an, gibt den entsprechenden Sektor im Token wieder frei und reicht das modifizierte Token weiter.

```
1  DiscreteBeOdss named: permissionManagementType.
2    input:   #(in_lane)          on: ScalarOdss;
3             #(out_lane)         on: ScalarOdss;
4
5    message: #(crossing_reached) on: EventOdss;
6             #(sector1_exited)   on: EventOdss;
7             #(sector2_exited)   on: EventOdss;
8             #(sector3_exited)   on: EventOdss;
9             #(sector4_exited)   on: EventOdss;
10            #(recv_token)       on: EventOdss
11                                args: #(lane)    on: IntegerOdss;
12                                      #(sector1) on: IntegerOdss;
13                                      #(sector2) on: IntegerOdss;
14                                      #(sector3) on: IntegerOdss;
15                                      #(sector4) on: IntegerOdss;
```

```
16
17    externalEvent: #(req_token)  on: EventOdss
18                                 args: #(lane) on: IntegerOdss;
19                   #(rel_token)  on: EventOdss
20                                 args: #(sector1) on: IntegerOdss;
21                                       #(sector2) on: IntegerOdss;
22                                       #(sector3) on: IntegerOdss;
23                                       #(sector4) on: IntegerOdss;
24
25    discreteMemory: #(sector1_needed) on: IntegerOdss;
26                    #(sector2_needed) on: IntegerOdss;
27                    #(sector3_needed) on: IntegerOdss;
28                    #(sector4_needed) on: IntegerOdss;
29
30    controlMode: #(mode) on: ControlModeOdss
31                         modes: #(normal wait_for_token crossing);
32
33    stateTransition:
34      mode modeIs: ( normal ) and: [crossing_reached isTriggered]
35        then: [sector1_needed := route_needs_sector(1, in_lane, out_lane);
36               sector2_needed := route_needs_sector(2, in_lane, out_lane);
37               sector3_needed := route_needs_sector(3, in_lane, out_lane);
38               sector4_needed := route_needs_sector(4, in_lane, out_lane);
39               req_token lane:in_lane;
40               mode nextMode: wait_for_token;];
41
42      mode modeIs: ( wait_for_token ) and: [recv_token isTriggered]
43        then: [((recv_token lane) = in_lane)
44                  and: (sector1_needed + (recv_token sector1) <= 1)
45                  and: (sector2_needed + (recv_token sector2) <= 1)
46                  and: (sector3_needed + (recv_token sector3) <= 1
47                  and: (sector4_needed + (recv_token sector4) <= 1)
48                  then: [rel_token
49                          sector1:(sector1_needed + (recv_token sector1))
50                          sector2:(sector2_needed + (recv_token sector2))
51                          sector3:(sector3_needed + (recv_token sector3))
52                          sector4:(sector4_needed + (recv_token sector4));
53                         mode nextMode: crossing]
54                  else: [rel_token sector1:(recv_token sector1)
55                                   sector2:(recv_token sector2)
56                                   sector3:(recv_token sector3)
57                                   sector4:(recv_token sector4)]];
58
59      mode modeIs: ( crossing ) and: [recv_token isTriggered]
60        then: [((recv_token lane) = in_lane)
61                  then: [rel_token
62                          sector1:(sector1_needed + (recv_token sector1))
63                          sector2:(sector2_needed + (recv_token sector2))
64                          sector3:(sector3_needed + (recv_token sector3))
65                          sector4:(sector4_needed + (recv_token sector4));
66                         (sector1_needed + sector2_needed
67                          + sector3_needed + sector4_needed = 0)
68                           then: [mode nextMode: normal]]];
69
```

```
70      mode modeIs: ( crossing ) and: [sector1_exited isTriggered]
71        then: [sector1_needed = 0;
72               req_token lane:in_lane];
73
74      mode modeIs: ( crossing ) and: [sector2_exited isTriggered]
75        then: [sector2_needed = 0;
76               req_token lane:in_lane];
77
78      mode modeIs: ( crossing ) and: [sector3_exited isTriggered]
79        then: [sector3_needed = 0;
80               req_token lane:in_lane];
81
82      mode modeIs: ( crossing ) and: [sector4_exited isTriggered]
83        then: [sector4_needed = 0;
84               req_token lane:in_lane];
85
86  end.
```

Als Beispiel für die Umsetzung dynamischer Systemstrukturen ist die Modellierung des Verkehrs an einer Kreuzung als Objective-DSS-Modell `crossingTrafficType` dargestellt. Das Ereignis des Annäherns eines Fahrzeugs an die Kreuzung wird durch den diskreten Eingang (Message) `new_vehicle` modelliert. Der Parameter `lane` gibt dabei die Spur an, auf der das Fahrzeug eintrifft. Bei Eintreten der Message `new_vehicle` wird ein neues MPO vom Typ `vehicleType` angelegt. Entsprechend der Spur wird das Fahrzeug am Ende der Kolonne eingereiht. Dabei wird das MPO mit dem MPO des direkten Vorgängers in der Kolonne verkoppelt, so dass dem neu hinzugekommenen Fahrzeug die für die Abstandsregelung erforderliche Position des Vorgängerfahrzeugs zur Verfügung steht.

```
1   DynamicHcsOdss named: crossingTrafficType.
2     message: #(new_vehicle) on: EventOdss args: #(lane) on: IntegerOdss;
3
4     partDefinition:
5       tokenManager is: tokenManagerType;
6       vehiclesOnLane1 isList: vehicleType;
7       firstVehicleOnLane1 isRef: vehicleType;
8       lastVehicleOnLane1 isRef: vehicleType;
9       vehiclesOnLane2 isList: vehicleType;
10      firstVehicleOnLane2 isRef: vehicleType;
11      lastVehicleOnLane2 isRef: vehicleType;
12      vehiclesOnLane3 isList: vehicleType;
13      firstVehicleOnLane3 isRef: vehicleType;
14      lastVehicleOnLane3 isRef: vehicleType;
15      vehiclesOnLane4 isList: vehicleType;
16      firstVehicleOnLane4 isRef: vehicleType;
17      lastVehicleOnLane4 isRef: vehicleType;
18
19    coupleCondition:
20      (new_vehicle isTriggered)
21        then: [newVehicle new: vehicleType;
```

```
22              newVehicle.lane := (new_vehicle lane);
23              ((new_vehicle lane) = 1)
24                then: [(FirstVehicleOnLane1 = nil)
25                          then: [firstVehicleOnLane1 := newVehicle;
26                                 lastVehicleOnLane1  := newVehicle]
27                          else: [lastVehicleOnLane1 at: pos connectTo:
28                                   newVehicle at: predecessor_pos]]
29                else: [((new_vehicle lane) = 2)
30                then: [(FirstVehicleOnLane2 = nil)
31                          then: [firstVehicleOnLane2 := newVehicle;
32                                 lastVehicleOnLane2  := newVehicle]
33                          else: [lastVehicleOnLane2 at: pos connectTo:
34                                   newVehicle at: predecessor_pos]]
35                else: [((new_vehicle lane) = 3)
36                then: [(FirstVehicleOnLane3 = nil)
37                          then: [firstVehicleOnLane3 := newVehicle;
38                                 lastVehicleOnLane3  := newVehicle]
39                          else: [lastVehicleOnLane3 at: pos connectTo:
40                                   newVehicle at: predecessor_pos]]
41                else: [((new_vehicle lane) = 4)
42                then: [(FirstVehicleOnLane4 = nil)
43                          then: [firstVehicleOnLane4 := newVehicle;
44                                 lastVehicleOnLane4  := newVehicle]
45                          else: [lastVehicleOnLane4 at: pos connectTo:
46                                   newVehicle at: predecessor_pos]]]]];
47
48              tokenManager at: send_token connectTo:
49                newVehicle at: recv_token;
50
51              newVehicle at: req_token connectTo:
52                tokenManager at: req_token;
53
54              newVehicle at: rel_token connectTo:
55                tokenManager at: rel_token];
56
57  end.
```

Bei dem hier vorgestellten Beispiel werden die Kolonnen jeweils vereinfacht als zentrale Warteschlangen pro Kreuzung und Spur modelliert. Ebenso ist die Weitergabe der Freigabetoken vereinfacht über einen zentralen Tokenmanager pro Kreuzung modelliert. Für Simulationszwecke ist diese vereinfachte, noch nicht vollständig dezentralisierte Organisation ausreichend. Ersetzt man den Tokenmanager durch entsprechende Kommunikationskomponenten pro Fahrzeug sowie die Kolonnen-Warteschlangen durch entsprechende Sensoren und Detektionsmechanismen pro Fahrzeug, so lässt sich der hier vorgestellte Ansatz zu einer vollständig dezentralen Lösung ausbauen.

9.3 Educational Truck (EduTruck)

Der Educational Truck (EduTruck) ist ein komplexes Mehrkörpermodell eines Nutzfahrzeugs [Hah99]. Im Rahmen der modellgestützten Analyse sollen an diesem Modell die Auswirkungen von konstruktiven Änderungen auf das Fahrverhalten des Fahrzeugs studiert werden.

Schwerpunkt der Betrachtung dieses Beispiels ist die verteilte Offline-Simulation komplexer Modelle unter Einsatz automatischer Lastverteilungsverfahren. Dazu wurde das Lastverteilungswerkzeug LoDiT2 (Load Distribution Tool 2) [Hee99] am MLaP entwickelt, das auf der Graphpartitionierungsbibliothek PARTY [Uni98] basiert.

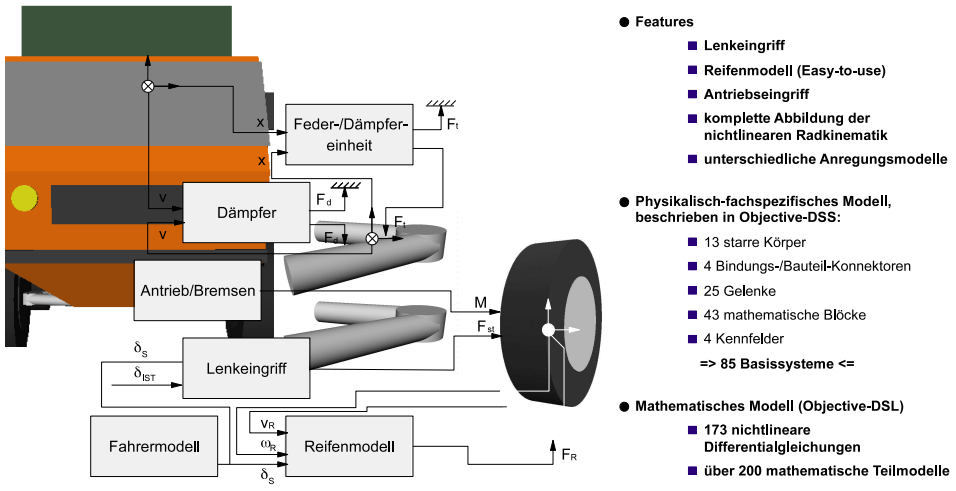


Abbildung 9.6: Modellierte Effekte und Komplexität des mathematischen Modells des EduTruck, dargestellt an einer der Vorderradaufhängungen [Hah99]

Das Gesamtfahrzeugmodell des Educational Truck besteht aus den folgenden Baugruppen:

- Fahrzeugaufbau
- 4 Einzelradaufhängungen in Doppelquerlenkerbauweise mit Radträger, Reifen, Felgen und Aktoren für die Federung
- Easy-to-use-Reifenmodell
- Antriebs-, Brems- und Lenkeingriff
- Anregungsmodell für unterschiedliche Fahrmanöver
- Bewertungsmodell

Abbildung 9.6 gibt einen Überblick über die modellierten Effekte und die Komplexität des mathematischen Modells. Die kinematische Struktur des EduTruck ist in Abbildung 9.7 dargestellt. Die Fünfteilung des Systems in den Chassisaufbau und die vier Halbachsen ist hier sehr gut erkennbar.

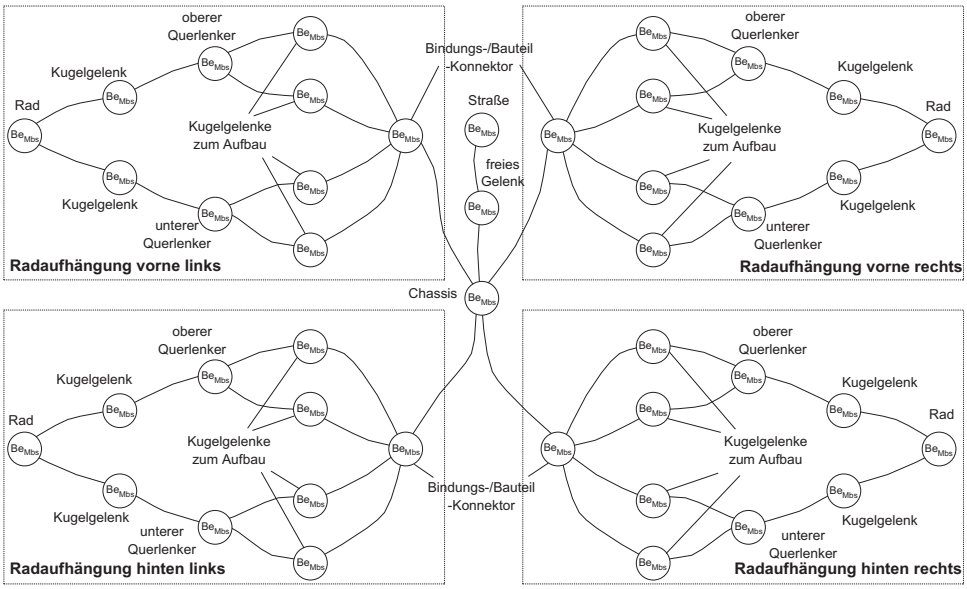


Abbildung 9.7: Kinematische Struktur des EduTruck-Modells [Hah99]

Bei einer Parallelisierung des Modells auf Basisblockebene erhält man einen Prozessgraphen mit einer Größe von 891 Knoten und 1204 Kanten [Hee99]. Führt man für diesen Prozessgraphen eine automatische Lastverteilung mit LoDiT2 durch, so ergibt sich für ein Netzwerk aus vier Transputern TM800 ein abgeschätzter Speedup bei der Laufzeit im Bereich von $1,21$ bis $3,66$ (s. Abbildung 9.8). Das beste Lastverteilungsergebnis bezüglich des Laufzeit-Speedups liefert die Clustering-Methode PARTY mit Modulo Mapping und Kommunikationszusammenfassung sowohl vor der eigentlichen Lastverteilung als auch nach dem Mapping und Scheduling.

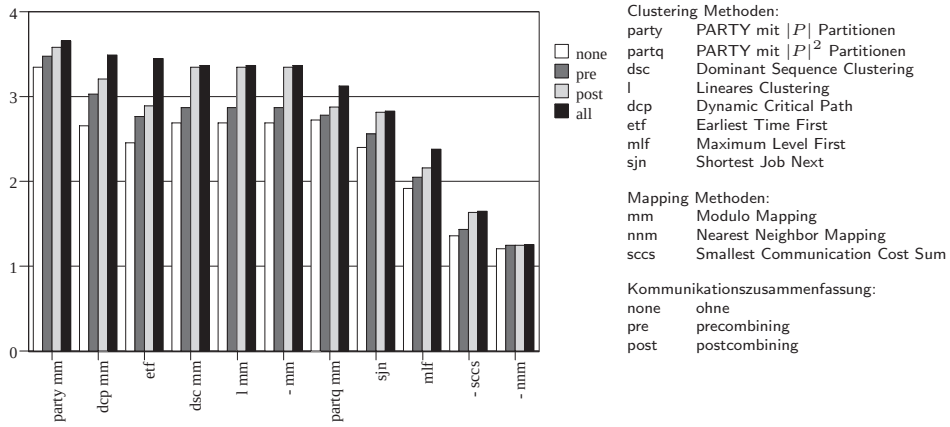


Abbildung 9.8: Speedup bei der Laufzeitvorhersage des EduTruck auf vier Prozessoren [Hee99]

Kapitel 10

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine verarbeitungsorientierte Modellrepräsentation als Basis einer universellen, offenen Entwurfsumgebung für mechatronische Systeme entwickelt und in Form einer Reihe von Softwarewerkzeugen realisiert. Die verarbeitungsorientierte Modellrepräsentation verkörpert im Wesentlichen einen kompaktenWerkzeuge, flachen Graphen, der das Systemverhalten des gesamten Modells in Form von rein funktionalen Zusammenhängen beschreibt. Das Modell des Systems liegt hier als eine Menge von rechnernahen Anweisungen vor. Für die auf dieser Ebene aufsetzenden Simulations-, Analyse- und Synthesewerkzeuge reduziert sich die Aufbereitung des Modells für die Berechnung auf ein Minimum, was die Entwicklung und Wartung dieser Werkzeuge sehr vereinfacht. Als textuelle Ausprägung in Form der Modellbeschreibungssprache DSC bietet sie ein offenes Austauschformat, das einfach maschinell weiterverarbeitet werden kann und so eine problemlose Anbindung weiterer Werkzeuge an die Entwicklungsumgebung ermöglicht.

Die besondere Herausforderung bei der Entwicklung mechatronischer Systeme, die Integration von Teilmodellen unterschiedlicher Fachdisziplinen, modelliert in unterschiedlichen, fachspezifischen Modellierungssprachen, wird durch kleine Modellierungs-Frontends gelöst, die als einfache Übersetzer die jeweilige fachspezifische Modellierungssprache in die verarbeitungsnahe Modellbeschreibungssprache DSC transformieren. Als Resultat hat man das gesamte System einheitlich und transparent in der verarbeitungsorientierten Modellbeschreibungsform vorliegen. Die weitere Auslegung des Systems kann vorgenommen werden, ohne durch Teilsystemgrenzen eingeschränkt zu sein. Auch hat man nicht mit den bei Simulatorkopplungen häufig auftretenden numerischen Problemen zu kämpfen.

Die verarbeitungsorientierte Modellrepräsentation erlaubt die Abbildung von modular-hierarchischen, gemischt diskret-kontinuierlichen Modellen.

Weiterhin ermöglicht sie die Abbildung von dynamischen Systemstrukturen, bei denen zur Laufzeit neue Systeme instanziiert bzw. wieder entfernt werden. Damit lassen sich alle Arten mechatronischer Systeme bis hin zu komplexen, autonomen Systemstrukturen behandeln.

Es wurde aufgezeigt, wie die einzelnen Entwurfsschritte eines mechatronischen Systems unter Einsatz der verarbeitungsorientierten Modellrepräsentation umgesetzt werden können. Für die Simulation ist immer eine totzeitoptimierte Ermittlung und Festlegung der Auswertereihenfolge der Systemgleichungen über das gesamte System hinweg möglich. Die Auswertung kann sowohl interpretierend als auch mit Hilfe von generiertem, effizientem C-Code geschehen. Clustermechanismen unterstützen verteilte Informationsverarbeitung, insbesondere auch für Echtzeitanwendungen. Auch die Anwendung von linearen Verfahren zur Systemanalyse und Synthese ist natürlich möglich, insbesondere auch eine symbolische Linearisierung gekoppelter Systeme. Für die Fehlersuche in den Modellen mechatronischer Systeme wurde ein Konzept für einen Modell-Debugger vorgestellt, der einen transparenten Zugang auf alle Teilmodelle des gesamten Systems ermöglicht.

Durch den Einsatz der Mechatronic Processing Objects und der zugrundeliegenden verarbeitungsorientierten Modellbeschreibungssprache DSC in einer Reihe von umfangreichen Projekten wurde der beschriebene Leistungsumfang und die dargestellte Offenheit der Entwicklungsumgebung nachgewiesen und, getrieben durch praxisrelevante Anwenderforderungen, weiter abgerundet. Ein Großteil der vorgestellten Ansätze ist mittlerweile in dem Werkzeug CAMeL-View kommerziell verfügbar.

Abschließend soll noch ein **Ausblick** auf weiterführende Arbeiten gegeben werden. Da wäre zunächst die Erweiterung der Entwicklungsumgebung um weitere Modellierungs-Frontends zu nennen. Ein wichtiger Kandidat ist hier VHDL-AMS (Analog and Mixed-Signal Extensions) [VHD07, Her06], eine Weiterentwicklung der Hardwarebeschreibungssprache VHDL [VHD08], die die Modellierung von analogen und gemischten (d. h. analogen und digitalen) Systemen ermöglicht. Die Beschreibungssprache wird primär in der Elektronik verwendet, lässt sich aber auch für viele andere Fachdisziplinen sinnvoll einsetzen.

In Hinsicht auf die verteilte Echtzeitverarbeitung laufen am MLaP weiterführende Arbeiten, die auf der verarbeitungsorientierten Modellbeschreibungssprache DSC basieren. Hier ist insbesondere die modulare Rapid-Prototyping-Plattform RABBIT zu nennen [LGZ+01]. Soll eine Simulation auf Basis von generiertem C-Code durchgeführt werden, so wird standardmäßig vor der Codegenerierung eine Auswertereihenfolge ermittelt und diese dann fest in dem generierten Code umgesetzt. Hier gibt es weiterführende Arbeiten, die eine Konfiguration der Auswertereihenfolge auch noch nach erfolgter Codegenerierung ermöglichen sollen [GKMV06].

Bezüglich der Echtzeithardware für die Reglerrealisierung hat sich diese Arbeit auf Rechnerarchitekturen konzentriert, deren einzelne Rechenkerne

nach dem Prinzip des von-Neumann-Rechners operieren. Die verarbeitungsorientierte Modellbeschreibungssprache DSC kann aber auch für das Rapid Control Prototyping mit anderen Rechnerarchitekturen wie z. B. FPGAs oder Datenfluss-Architekturen [Ung93] eingesetzt werden. In dem Werkzeug CAMeL-View gibt es bereits die Möglichkeit, FPGA-in-the-Loop-Simulationen durchzuführen [MGP+08]. Der Ansatz geht allerdings über den Umweg, zunächst eine Transformation in ein anderes mathematisches Modell — in diesem Fall ein Simulink-Modell — vorzunehmen und aus diesem Simulink-Modell dann über den Simulink HDL Coder [MAT10c] eine VHDL-Beschreibung zu generieren, mit der der FPGA programmiert wird. Hier ließe sich mit der Entwicklung eines einfachen Backends, das aus der verarbeitungsorientierten Repräsentation des Controller-Modells in CAMeL-View direkt VHDL generiert, eine von MATLAB/Simulink unabhängige Werkzeugkette schaffen. Damit kann man die oft problematische Transformation in eine andere mathematische Modellbeschreibung, die häufig zu Verlusten in der Modellsemantik führt, vollständig vermeiden. Bezüglich des Einsatzes von DSC auf Datenfluss-Architekturen gibt es erste Ansätze mit dem FLYSIG-Prozessor [HKR00].

Für das Werkzeug CAMeL-View existiert weiterhin eine Tool-Kopplung mit der Fujaba Real-Time Tool Suite [BGH+07], die auch im Rahmen des Sonderforschungsbereichs 614 “Selbstoptimierende Systeme des Maschinenbaus“ [SFB11] eingesetzt wird. Teilmodelle können als Code oder per Simulatorkopplung gegenseitig integriert werden. Auch hier ist es aus den in dieser Arbeit aufgeführten Gründen sinnvoll, ein Modellierungs-Frontend für Fujaba einzusetzen, dass die entsprechenden Teilmodelle in der verarbeitungsorientierten Beschreibungssprache DSC bereitstellt.

Literaturverzeichnis

[ACHH93] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems I*, Lecture Notes in Computer Science 736, pages 209–229. Springer-Verlag, 1993.

[AG97] M. Antoniotti and A. Göllü. SHIFT and SmartAHS: A Language for Hybrid Systems Engineering, Modeling, and Simulation. In *Proceedings of the USENIX Conference of Domain Specific Languages*, Santa Barbara, CA, U.S.A., October 1997.

[AM10] Karine Altisen and Matthieu Moy. ac2lus: Bringing SMT-solving and abstract interpretation techniques to real-time calculus through the synchronous language Lustre. In *22nd Euromicro Conference on Real-Time Systems (ECRTS)*, Brussels, Belgium, July 2010.

[And94] Mats Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1994.

[BB09] Torsten Blochwitz and Thomas Beutlich. Real-Time Simulation of Modelica-based Models. In *Proceedings 7th Modelica Conference*, pages 386–392, 20 - 22 September 2009.

[BDL09] Marco Bonvini, Filippo Donida, and Alberto Leva. Modelica as a design tool for hardware-in-the-loop simulation. In *Proceedings 7th Modelica Conference*, pages 378–385, 20 - 22 September 2009.

[BGBK08] S. Barner, M. Geisinger, C. Buckl, and A. Knoll. EasyLab: Model-based Development of Software of Mechatronic Systems. In *IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, pages 540–545, 2008.

[BGH+07] Sven Burmester, Holger Giese, Stefan Henkler, Martin Hirsch, Matthias Tichy, Alfonso Gambuzza, Eckehard Münch, and Henner Vöcking. Tool Support for Developing Advanced Mechatronic Systems: Integrating the Fujaba Real-Time Tool Suite with CAMeL-View. In *29th International Conference on Software Engineering (ICSE)*, pages 801–804, Minneapolis, Minnesota, USA, May 2007.

[BGH+10] Simon Barner, Michael Geisinger, Jia Huang, Alois Knoll, Holger Bönicke, Christoph Ament, Jochen Mades, Reinhard Pittschellis, Gerd Bauer. EasyKit — Eine allgemeine Methodik für die Entwicklung von Steuerungskomponenten. In Jürgen Gausemeier, Franz Rammig, Wilhelm Schäfer, Ansgar Trächtler (Hrsg.), *Entwurf mechatronischer Systeme*, *HNI-Verlagsschriftenreihe*, Band 272, S. 23–26. Heinz Nixdorf Institut, Universität Paderborn, Paderborn, 18 - 19 März 2010.

[BS89a] I. N. Bronstein, K. A. Semendjajew. *Näherungsweise Differentation*, Kap. 7.1.2.8, S. 767–768. Verlag Harri Deutsch, 1989.

[BS89b] I. N. Bronstein, K. A. Semendjajew. *Nichtlineare Gleichungssysteme*, Kap. 7.1.2.4, S. 747. Verlag Harri Deutsch, 1989.

[BTC10] BTC-ES, Oldenburg, Germany. *Embedded Validator, Release 3.4*, 2010.

[CCJ+10] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltev. *NuSMV 2.5 User Manual*. FBK-irst, Povo (Trento), Italy, 2010.

[CS09] Jürgen Crepin and Thomas Schmerler. Multipower für LABCAR — Multicore-Anwendungen für LABCAR-RTPC. *RealTimes*, (1), 2009.

[Das10] Dassault Systèmes, Lund, Sweden (Dynasim). *Dymola Version 7.4*, 2010.

[Dep11] Department of Information Technology, Uppsala University (UPP), Sweden, and Department of Computer Science at Aalborg University (AAL), Denmark. UPPAAL 4.0. `http://www.uppaal.org/`, April 2011.

[dSP10a] dSPACE GmbH, Paderborn, Germany. *RTI and RTI-MP Implementation Guide, Release 7.0*, November 2010.

[dSP10b] dSPACE GmbH, Paderborn, Germany. *TargetLink Production Code Generation Guide, Release 7.0*, November 2010.

[DSS+03] W. Damm, M. Segelken, C. Schulte, H. Wittke, U.Higgen, M. Eckrich. Formale Verifikation von ASCET Modellen im Rahmen der Entwicklung der Aktivlenkung. In Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.), *Informatik 2003 - Innovative Informatikanwendungen Band 1, Beiträge der 33. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, *GI-Edition - Lecture Notes in Informatics (LNI)*, Reihe P-34, S. 340–344, Frankfurt am Main, 29.9.–2.10. 2003.

[Ecl11] The Eclipse Foundation, http://www.eclipse.org. *Eclipse Helios (3.6) Documentation*, 2011.

[Eng95] Andreas Engelke. *Transient — Ein Werkzeug zur Simulation mechatronischer Systeme unter Echtzeitbedingungen.* Fortschr.-Ber. VDI Reihe 20 Nr. 153. VDI Verlag, Düsseldorf, 1995.

[Est11] Esterel Technologies S.A., Elancourt, France. SCADE Suite. `http://www.esterel-technologies.com/products/scade-suite/`, April 2011.

[ETA11] ETAS GmbH, Stuttgart, Germany. ASCET Software-Produkte. `http://www.etas.com/de/products/ascet_software_products.php`, April 2011.

[Fla94] D. Flanagan. *Motif Tools: Streamlined GUI Design and Programming with the Xmt Library.* O'Reilly & Associates, Sebastopol, CA, 1994.

[GKMV06] Alfonso Gambuzza, Dirk Koert, Eckehard Münch, Henner Vöcking. Automatische Codegenerierung für verteilte Informationsverarbeitung in mechatronischen Systemen. In *4. Paderborner Workshop - Entwurf mechatronischer Systeme*, Paderborn, 2006.

[GL00] Jürgen Gausemeier, Joachim Lückel. *Entwicklungsumgebungen Mechatronik - Methoden und Werkzeuge zur Entwicklung Mechatronischer Systeme*, *HNI-Verlagschriftenreihe*, Band 80. Heinz Nixdorf Institut, Universität Paderborn, Paderborn, 2000.

[Hah91] Martin Hahn. Ein graphischer Ansatz zur topologischen Beschreibung und Kinematikanalyse von Mehrkörpersystemen. Diplomarbeit, MLaP, FB 10, Universität-GH Paderborn, 1991.

[Hah99] Martin Hahn. *OMD - Ein Objektmodell für den Mechatronikentwurf.* Fortschr.-Ber. VDI Reihe 20 Nr. 299. VDI Verlag, Düsseldorf, 1999.

[Hee99] Karsten Hees. Kommunikationseffiziente Lastverteilungsverfahren zur Simulation mechatronischer Systeme. Diplomarbeit, Universität-GH Paderborn, FB 17 - Informatik, 1999.

[Hen96] Thomas A. Henzinger. The Theory of Hybrid Automata. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkely, 1996.

[Her06] Yannick Hervé. *VHDL-AMS: Anwendungen und industrieller Einsatz*. Oldenbourg, München, Wien, 2006.

[HHR94] M. Hahn, C. Homburg, J. Richert. DSS-DSL-DSC. Die drei Ebenen einer Modellbeschreibungssprache für mechatronische Systeme. In *9. Symposium Simulationstechnik*, Stuttgart, 10.-13. Oktober 1994.

[HKR00] W. Hardt, B. Kleinjohann, and A. Rettberg. The FLYSIG prototyping approach. In *11th International Workshop on Rapid System Prototyping*, Paris, 2000.

[HO99] C. Homburg, O. Oberschelp. Modell-Debugging mechatronischer Systeme. In *4. Magdeburger Maschinenbau-Tage*, S. 241–248, Otto-von-Guericke-Universität Magdeburg, 22.-23. September 1999.

[Hom93] C. Homburg. Entwurf und Implementierung einer standardisierten Beschreibungsform mechatronischer Systeme. Diplomarbeit, Universität Paderborn, 1993.

[Hom96] C. Homburg. SIMBA – eine offene Simulationsumgebung für mechatronische Systeme auf der Basis von DSC. In *4. Workshop "Methoden- und Werkzeugentwicklung für den Mikrosystementwurf", 4. Statusseminar zum BMBF-Verbundprojekt METEOR*, S. 489–502, Karlsruhe, 18.-19. November 1996.

[Hom97] C. Homburg. SIMBA - Increasing Efficiency in the Simulation of Heterogeneously Modelled Mechatronic Systems. In *9th European Simulation Symposium, Simulation in Industry*, pages 713–717, Passau, Oktober 1997.

[Int94] Integrated Systems Inc., Sunnyvale, CA. *SystemBuild User's Guide, Version 4.0*, 1994.

[Jae91] Karl-Peter Jaeker. *Entwicklung realisierbarer hierarchischer Kompensatorstrukturen für lineare Mehrgrößensysteme mittels CAD*. Fortschr.-Ber. VDI Reihe 8 Nr. 243. VDI Verlag, Düsseldorf, 1991.

[JKL+91] K.-P. Jäker, P. Klingebiel, U. Lefarth, J. Lückel., J. Richert, and R. Rutz. Tool Integration by Way of a Computer-Aided Mechatronics Laboratory (CAMeL). In *CADCS 91, 5th IFAC/IMACS Symposium on Computer-Aided Design in Control Systems*, Swansea, Wales, 15th–17th July 1991.

[Kas90] Uwe Kastens. *Übersetzerbau*, *Handbuch der Informatik*, Band 3.3. R. Oldenbourg Verlag, München Wien, 1990.

[Kas92] Roland Kasper. *Entwicklung und Erprobung eines instrumentellen Verfahrens zum Entwurf von Mehrgrößensystemen*. Fortschr.-Ber. VDI Reihe 8 Nr. 90. VDI Verlag, Düsseldorf, 1992.

[KPJ98] Uwe Kastens, Peter Pfahler, and Matthias Jung. The Eli System. In Kai Koskimies, editor, *Proceedings 7th International Conference on Compiler Construction CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 294–297. Springer Verlag, March 1998.

[Kro93] B. H. Krogh. Condition/Event Signal Interfaces for Block Diagram Modeling and Analysis of Hybrid Systems. In *Proceedings of the 1993 International Symposium on Intelligent Control*, pages 180–185, Chicago, Illinois, USA, August 1993.

[LE09] U. Lauff and M. Elbs. PC-basierte Tools für Test und Entwicklung. *Hanser automotive*, (11):10–13, 2009.

[Lef96] Ulrich Lefarth. *SIMEX - eine offene Simulationsumgebung zum rechnergestützten Entwurf mechatronischer Systeme*. Fortschr.-Ber. VDI Reihe 20 Nr. 223. VDI Verlag, Düsseldorf, 1996.

[LGZ+01] Thomas Lehmann, R. Gielow, Mauro C. Zanella, M. Robrecht, Volker Horsthemke, and A. Fransisco de Freitas. Rabbit – a modular rapid-prototyping platform for distributed mechatroinc systems. In *SBCCI 2001*, Pirenópolis, Brazil, 2001.

[MAT92] MATHWORKS Inc., South Natick, MA. *SIMULINK – User's Manual*, 1992.

[Mat10a] The MathWorks, Inc., Natick, MA. *Real-Time Workshop Embedded Coder User's Guide, Version 5.6 (Release 2010b)*, 2010.

[Mat10b] The MathWorks, Inc., Natick, MA. *Real-Time Workshop User's Guide, Version 7.6 (Release 2010b)*, 2010.

[MAT10c] MATHWORKS Inc., South Natick, MA. *SIMULINK HDL Coder – User's Manual*, 2010.

[Mat10d] The MathWorks, Inc., Natick, MA. *Simulink User's Guide, Version 7.6 (Release 2010b)*, 2010.

[Mat10e] The MathWorks, Inc., Natick, MA. *Stateflow User's Guide, Version 7.6 (Release 2010b)*, 2010.

[MGP+08] Eckehard Münch, Alfonso Gambuzza, Carlos Paiz, Christopher Pohl, and Mario Porrmann. Fpga-in-the-loop simulations with camel-view. In Jürgen Gausemeier, Franz Josef Rammig, and Schäfer Wilhelm, editors, *Proceedings of the 7th International Heinz Nixdorf Symposium: Self-optimzing Mechatronic Systems*, ALB-HNI-Verlagsschriftenreihe. Heinz Nixdorf Institut, Heinz Nixdorf Institut, 20 - 21 February 2008.

[Mod10] Modelica Association. *Modelica, A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification, Version 3.2*, 24 March 2010.

[Nat11] National Instruments Corporation, Austin, Texas. LabVIEW. `http://www.ni.com/labview/`, April 2011.

[Obj05] Object Management Group, Inc., http://www.omg.org/spec/SPTP/1.1/PDF. *UML Profile for Schedulability, Performance and Time Specification, Version 1.1*, January 2005.

[Obj09] Object Management Group, Inc., http://www.omg.org/spec/MARTE/1.0/PDF. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.0*, November 2009.

[Obj10] Object Management Group, Inc., http://www.omg.org/spec/SysML/1.2/PDF. *OMG Systems Modeling Language (OMG SysML), Version 1.2*, June 2010.

[OE95] M. Otter and H. Elmquist. The DSblock model interface for exchanging model components. In F. Breitenecker and I. Husinsky, editors, *EUROSIM 95*, pages 505–510. North-Holland, Amsterdam, 1995.

[OME+09] Martin Otter, Martin Malmheden, Hilding Elmqvist, Sven Erik Mattson, and Charlotta Johnsson. A New Formalism for Modeling of Reactive and Hybrid Systems. In *Proceedings 7th Modelica Conference*, pages 364–377, 20 - 22 September 2009.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, 1994.

[PBB+10] C.J.J. Paredis, Y. Bernard, R.M. Burkhart, H.-P. de Koning, S. Friedenthal, P. Fritzson, N.F. Rouquette, and W. Schamai. An Overview of the SysML-Modelica Transformation Specification. In *Proceedings of the 2010 INCOSE International Symposium*, Chicago, IL, 12 - 15 July 2010.

[PRO96] PROFI ENGINEERING SYSTEMS GmbH, Darmstadt. *PROFI 2000 User's Guide and Reference Manual*, 1996.

[Pro11] Prover Technology AB, Stockholm, Sweden. Prover Plug-In. `http://www.prover.com/products/prover_plugin/`, April 2011.

[Ric96] Jobst Richert. Integration of Mechatronic Design Tools with CAMeL — Exemplified by Vehicle Convoy Control Design. In *IEEE International Symposium on CACSD*, Dearborn, Michigan, USA, September 15–18 1996.

[Ric98] Jobst Richert. *CAMeL — Eine Integrationsplattform für kooperative Entwurfswerkzeuge der Mechatronik*. Fortschr.-Ber. VDI Reihe 20 Nr. 273. VDI Verlag, Düsseldorf, 1998.

[Sch94] Joachim Schröer. *Eine Modellbeschreibungssprache zur Unterstützung der Simulation und Optimierung von nichtlineraren und linearisierten hierarchischen Systemen*. Fortschr.-Ber. VDI Reihe 20 Nr. 128. VDI Verlag, Düsseldorf, 1994.

[Sed92] Robert Sedgewick. *Algorithmen in C*. Addison-Wesley, 1992.

[SFB11] Internetseite des Sonderforschungsbereichs 614. `http://sfb614.de`, April 2011.

[SK91] R. S. Sreenivas and B. H. Krogh. On Condition/Event Systems with Discrete State Realizations. *Discrete Event Dynamic Systems: Theory and Applications*, 1(2):209–236, 1991.

[THB+10] Matthias Tichy, Martin Hirsch, Christopher Brink, Wilhelm Schäfer, Christopher Gerking, Martin Hahn. Integration hybrider Modellierungstechniken in CAMeL-View. In Jürgen Gausemeier, Franz Rammig, Wilhelm Schäfer, and Ansgar Trächtler (Hrsg.), *Entwurf mechatronischer Systeme*, *HNI-Verlagsschriftenreihe*, Band 272, S. 235–251. Heinz Nixdorf Institut, Universität Paderborn, Paderborn, 18 - 19 März 2010.

[Ung93] Theo Ungerer. *Datenflußrechner*. Teubner-Verlag, 1993.

[Uni92] Universität-Gesamthochschule Paderborn, Automatisierungstechnik — Prof. Dr.-Ing. Joachim Lückel. *LINTOOL Benutzerhandbuch*, Version 1.2, Februar 1992.

[Uni96] Universität-Gesamthochschule Paderborn, Automatisierungstechnik — Prof. Dr.-Ing. Joachim Lückel. *OPIDEX Benutzerhandbuch*, 1996.

[Uni98] Universität Paderborn, Paderborn. *The Party Graphpartitioning-Library, User Manual - Version 1.99*, 1998.

[VHD07] IEEE 1076.1-2007 Standard VHDL Analog and Mixed-Signal Extensions. Technical report, IEEE, 2007.

[VHD08] IEEE 1076-2008 Standard VHDL Language Reference Manual. Technical report, IEEE, 2008.

[Vül96] Dieter Vüllers. Entwurf und Implementierung eines DSC-Debuggers für mechatronische Systeme. Diplomarbeit, Universität-Gesamthochschule Paderborn, Fachbereich 17 - Mathematik/Informatik, April 1996.

[Wä01] Peter Wältermann. *Der serielle Hybridantrieb - Vom rechnergestützten Entwurf bis zur Hardware-in-the-Loop-Realisierung.* Fortschr.-Ber. VDI Reihe 12 Nr. 447. VDI Verlag, Düsseldorf, 2001.

[Wer94] J. Wernecke. *Inventor Mentor: Programming Object oriented 3D Graphics with OpenInventor.* Addision-Wesley, Reading, MA, 2 edition, 1994.

[Wis94] R. Wismüller. *Quellsprachorientiertes Debugging von optimierten Programmen.* Dissertation, Technische Universität München, München, Germany, Dezember 1994.

Anhang A

IRL2-Syntax

Speziell für die Definition und Implementierung der den Mechatronic Processing Objects zugrundeliegenden verarbeitungsnahen Systembeschreibungssprache DSC wurde eine eigene Spezifikationssprache entwickelt, die *Intermediate Representation Language 2* (IRL2). IRL2 ist als universelle Spezifikationssprache für die Beschreibung von komplexen Datenstrukturen angelegt.

Als Grundelemente stehen in IRL2 Strukturen, Alternativen und Sequenzen zur Verfügung. Zur Abbildung von Objektbeziehungen können relative und absolute Objektreferenzen verwendet werden. Das Konzept unterschiedlicher Sichtbarkeitsbereiche von Objekten ermöglicht auch das Einbeziehen von externen Objekten. Zur Unterstützung der Behandlung von Objekttypen und Objektinstanzen können sowohl typ- als auch instanzspezifische Attribute angelegt werden. Die wesentlichen Sprachelemente sind in der nachfolgenden Tabelle aufgelistet:

StructureDef	Definition einer Datenstruktur mit dem Bezeichner *IdDef* Inhalt des Strukturtyps ist eine Liste von Attributen *SeqOfAttribute*.
Attribute	Definition eines Attributs einer Datenstruktur
Structure	Datentyp eines Attributs Hierbei kann es sich um eine Alternative *Alternatives*, eine Sequenz, den vordefinierten Typen **INTEGER**, **REAL** oder **STRING** oder um den Bezeichner einer anderen in IRL2 spezifizierten Datenstruktur handeln.

ReferenceDef	Definition eines Datentyps mit dem Bezeichner *IdDef*, mit dem sich Objektreferenzen darstellen lassen Mit diesem Datentyp lassen sich Objekte aller unter *RefStructs* aufgelisteten Datenstrukturen referenzieren.
ReferenceOption	Option, die bei der Definition einer Datenstruktur angibt, ob Objekte dieser Struktur referenziert werden können
NewScopeOption	Option, die bei der Definition einer Datenstruktur angibt, ob für die Objekte dieser Struktur ein neuer Sichtbarkeitsbereich angelegt werden soll
CopyOption	Option, die bei der Definition eines Attributs angibt, wie der Inhalt des Attributs bei Objektkopien behandelt werden soll
BidirectionalOption	Option, die bei der Definition eines Objektreferenz-Typen angibt, ob die Objektreferenz bidirektional sein soll
InScopeOption	Option, die bei der Definition eines Objektreferenz-Typen angibt, welcher Sichtbarkeitsbereich bei Referenzen über Schlüsselattribute gelten soll
KeyOption	Option, die bei der Definition eines Objektreferenz-Typen angibt, welches Attribut als Schlüssel für die Auflösung von Referenzen verwendet werden soll

Der komplette Sprachumfang von IRL2 ist in der nachfolgenden Grammatik dargestellt. Weitere Details bezüglich der Modellierung von Datenstrukturen mit IRL2 sowie veranschaulichende Beispiele sind in Kapitel 4.4.1 aufgeführt.

IntermediateRepresentation $\longrightarrow$ DefinitionBlock +.

DefinitionBlock $\longrightarrow$ RepresentedStructureDef
SeqOfStructureOrRefDef.

RepresentedStructureDef $\longrightarrow$ '**REPRESENTED**' StructureDef.

SeqOfStructureOrRefDef $\longrightarrow$ SeqOfStructureOrRefDef StructureOrRefDef
| ϵ.

StructureOrRefDef $\longrightarrow$ StructureDef
| ReferenceDef.

StructureDef ⟶ '**STRUCTURE**' IdDef '**IS**'
ReferenceOption NewScopeOption
SeqOfAttribute
'**END**'.

ReferenceOption ⟶ '**REFERENCED**'
| ϵ.

NewScopeOption ⟶ '**NEW**' '**SCOPE**'
| ϵ.

SeqOfAttribute ⟶ SeqOfAttribute Attribute
| Attribute.

Attribute ⟶ IdDef '**:**' CopyOption Structure '**;**'.

CopyOption ⟶ '**COPY**' '**OF**'
| ϵ.

Structure ⟶ Alternatives
| '**SEQ**' '**OF**' Simple.

Alternatives ⟶ Alternatives '**|**' Simple
| Simple.

Simple ⟶ IdUse
| '**INTEGER**'
| '**REAL**'
| '**STRING**'.

ReferenceDef ⟶ IdDef '**IS**' BidirectionalOption
'**REFERENCE**' '**TO**' RefStructs
InScopeOption KeyOption '**END**'.

BidirectionalOption ⟶ '**BIDIRECTIONAL**'
| ϵ.

InScopeOption ⟶ '**IN**' '**SCOPE**' ScopeStruct
| ϵ.

KeyOption ⟶ '**KEY**' Keys
| ϵ.

RefStructs ⟶ RefStructs '**,**' RefStruct
| RefStruct.

Keys ⟶ Keys '**,**' Key
| Key.

Key	$\longrightarrow$	Ident '**.**' Ident.
IdDef	$\longrightarrow$	Ident.
IdUse	$\longrightarrow$	Ident.
RefStruct	$\longrightarrow$	Ident.
ScopeStruct	$\longrightarrow$	Ident.

Anhang B

DSC-Definition

Dieser Anhang enthält die vollständige Spezifikation der verarbeitungsnahen Systembeschreibungssprache Dynamic System Code (DSC). DSC beschreibt in Form einer Zwischensprache die zentrale Datenstruktur des Mechatronic Processing Objects.

DSC ist vollständig in IRL2 spezifiziert, einer Spezifikationssprache zur Beschreibung komplexer Datenstrukturen. Eine Beschreibung der hier verwendeten Syntax und Semantik ist in Anhang A zu finden.

```
REPRESENTED STRUCTURE DSC IS REFERENCED
  Object : System | SystemType;
END

STRUCTURE System IS
  Class             : Linear | PartialLinear | NonLinear;
  SystemStructure   : SpecificationTree;
  DirectLinks       : SEQ OF DirectLink;
  ParamCouplings    : Couples;
  InOutCouplings    : Couples;
  BasicBlocks       : SEQ OF BasicBlock;
  DynamicStructures : SEQ OF DynamicStructure;
  EquationEvalOrder : SEQ OF EquationDirectLinkPart;
  Communication     : SEQ OF CommunicationData;
  Clustering        : ClusterNode;
END

STRUCTURE SystemType IS
  Description : System;
END
```

```
STRUCTURE Linear IS
  NoInformation : SEQ OF STRING;
END

STRUCTURE PartialLinear IS
  NoInformation : SEQ OF STRING;
END

STRUCTURE NonLinear IS
  NoInformation : SEQ OF STRING;
END

STRUCTURE SpecificationTree IS REFERENCED NEW SCOPE
  SystemName    : SEQ OF STRING;
  SrcClass      : SrcClassSpecification;
  Specification : Variables | Instantiation | IsExtern;
  Subsystems    : SEQ OF SpecificationTree;
END

SysSpecBiRef IS BIDIRECTIONAL REFERENCE TO SpecificationTree
END

STRUCTURE SrcClassSpecification IS
  Specification : STRING;
END

STRUCTURE Variables IS
  TimeVariables : SEQ OF Time;
  Parameters    : SEQ OF Param;
  Inputs        : SEQ OF Input;
  Outputs       : SEQ OF Output;
  ExternFunc    : SEQ OF Func;
  States        : SEQ OF State;
  Auxiliars     : SEQ OF Aux;
  EventParams   : SEQ OF EventParam;
  Messages      : SEQ OF Message;
  ExtEvents     : SEQ OF ExternalEvent;
  IntEvents     : SEQ OF InternalEvent;
  DiscrMemories : SEQ OF DiscreteMemory;
  CtrlModes     : SEQ OF ControlMode;
END
```

```
STRUCTURE Time IS REFERENCED
  Name : STRING;
END

STRUCTURE Param IS REFERENCED
  Name       : STRING;
  Unit       : STRING;
  MinValue   : SEQ OF REAL;
  MaxValue   : SEQ OF REAL;
-- start value optional
  StartValue : REAL | Nil;
  DebugInfo  : References | Nil;
END

STRUCTURE Input IS REFERENCED
  Name       : STRING;
  Unit       : STRING;
  MinValue   : SEQ OF REAL;
  MaxValue   : SEQ OF REAL;
  StartValue : REAL;
  DebugInfo  : References | Nil;
END

STRUCTURE Output IS REFERENCED
  Name      : STRING;
  Unit      : STRING;
  MinValue  : SEQ OF REAL;
  MaxValue  : SEQ OF REAL;
  DebugInfo : References | Nil;
END

STRUCTURE Func IS REFERENCED
  Name : STRING;
END

STRUCTURE State IS REFERENCED
  Name       : STRING;
  Unit       : STRING;
  MinValue   : SEQ OF REAL;
  MaxValue   : SEQ OF REAL;
  StartValue : REAL;
  DebugInfo  : References | Nil;
END
```

```
STRUCTURE Aux IS REFERENCED
  Name       : STRING;
  StartValue : REAL | INTEGER;
  DebugInfo  : References | Nil;
END

STRUCTURE EventParam IS REFERENCED
  Name         : STRING;
  Unit         : STRING;
  MinValue     : SEQ OF REAL;
  MaxValue     : SEQ OF REAL;
  DefaultValue : REAL | INTEGER;
  DebugInfo    : References | Nil;
END

EP is BIDIRECTIONAL REFERENCE TO EventParam
END

STRUCTURE Message IS REFERENCED
  Name      : STRING;
  Params    : SEQ OF EP;
  DebugInfo : References | Nil;
END

STRUCTURE ExternalEvent IS REFERENCED
  Name      : STRING;
  Params    : SEQ OF EP;
  DebugInfo : References | Nil;
END

STRUCTURE InternalEvent IS REFERENCED
  Name      : STRING;
  Params    : SEQ OF EP;
  DebugInfo : References | Nil;
END

STRUCTURE DiscreteMemory IS REFERENCED
  Name       : STRING;
  Unit       : STRING;
  MinValue   : SEQ OF REAL;
  MaxValue   : SEQ OF REAL;
  StartValue : REAL | INTEGER;
  DebugInfo  : References | Nil;
END
```

```
STRUCTURE ControlMode IS
  Name : STRING;
END

STRUCTURE References IS
  Coordinates : SEQ OF Coordinate;
  Condition   : Expr;
  Generated   : INTEGER;
  Origin      : SEQ OF Ident;
  Extension   : STRING;
END

STRUCTURE Coordinate IS
  SrcFileName : STRING;
  StartLine   : INTEGER;
  StartColumn : INTEGER;
  NextLine    : INTEGER;
  NextColumn  : INTEGER;
END

STRUCTURE Instantiation IS
  Name      : SEQ OF STRING;
  Redefined : SEQ OF Assign;
END

STRUCTURE IsExtern IS
  NoInformation : SEQ OF STRING;
END

STRUCTURE DirectLink IS
  Output : Ident;
  Input  : Ident;
END

STRUCTURE Couples IS
  Hierarchy           : CoupleTree;
  BasicBlockCouplings : COPY OF SEQ OF BasicCoupleEquation;
END
```

```
STRUCTURE CoupleTree IS
  Couplings         : SEQ OF Assign;
  CoupledSubsystems : SEQ OF CoupleTree;
  SysSpec           : SysSpecBiRef;
END

STRUCTURE BasicCoupleEquation IS
  Equation : Assign;
END

BasCplEqn IS REFERENCE TO BasicCoupleEquation
END

STRUCTURE BasicBlock IS
  InitCode             : SEQ OF Assign;
  IndirectLinkCode     : LinkCode;
  DirectLinkCode       : SEQ OF LinkCode;
  StateCode            : NoLinkCode;
  LinearRepresentation : LinearCode | Nil;
END

STRUCTURE LinkCode IS
  AuxiliarComp : SEQ OF AuxiliarEquation;
  OutputComp   : SEQ OF OutputEquation;
END

STRUCTURE NoLinkCode IS
  AuxiliarComp : SEQ OF Assign;
  StateComp    : SEQ OF Assign;
END

STRUCTURE AuxiliarEquation IS
  Equation : Assign;
END

AuxEqn IS REFERENCE TO AuxiliarEquation
END

STRUCTURE OutputEquation IS
  Equation : Assign;
END

OutEqn IS REFERENCE TO OutputEquation
END
```

```
STRUCTURE LinearCode IS
  StateDerivedAux : SEQ OF Assign;
  InputDerivedAux : SEQ OF Assign;
  A               : SEQ OF MatrixEntry;
  B               : SEQ OF MatrixEntry;
  C               : SEQ OF MatrixEntry;
  D               : SEQ OF MatrixEntry;
END

STRUCTURE MatrixEntry IS
  Row        : INTEGER;
  Column     : INTEGER;
  Expression : Expr;
END

STRUCTURE DynamicStructure IS
  SystemRefs            : SEQ OF SystemRef;
  DynamicConfigurations : SEQ OF DynamicConfiguration;
END

STRUCTURE DynamicConfiguration IS
  Name                : STRING;
  ConfigurationChange : SEQ OF ConfigurationChange;
END

STRUCTURE ConfigurationChange IS
  TriggerEvent          : V;
  DynamicInstantiations : SEQ OF DynamicInstantiation;
  DynamicDeletions      : SEQ OF DynamicDeletion;
  NextConfiguration     : DynamicConfigurationRef;
END

STRUCTURE DynamicInstantiation IS
  Instantiation    : Instantiation;
  SystemRefAssigns : SEQ OF SystemRefAssign;
  Couplings        : SEQ OF Assign;
END

STRUCTURE DynamicDeletion IS
  Instance         : SystemRef;
  SystemRefAssigns : SEQ OF SystemRefAssign;
  Couplings        : SEQ OF Assign;
END
```

```
STRUCTURE EquationDirectLinkPart IS
  AuxiliarComp : SEQ OF AuxEqn;
  OutputComp   : SEQ OF OutEqn;
  CoupleComp   : SEQ OF BasCplEqn;
END

STRUCTURE CommunicationData IS
  Idents : SEQ OF Ident;
END

STRUCTURE CommunicationDataRef IS
  Subcluster : SEQ OF INTEGER;
  Index      : INTEGER;
END

STRUCTURE SendDataRef IS
  CommDataRef : CommunicationDataRef;
END

STRUCTURE ReceiveDataRef IS
  CommDataRef : CommunicationDataRef;
END

STRUCTURE ClusterNode IS
  Content : HierarchicalClusterNode | BlockHierarchy;
END

STRUCTURE HierarchicalClusterNode IS
  Name                : STRING;
  Type                : STRING;
  IndirectInputs      : SEQ OF Ident;
  DirectInputs        : SEQ OF Ident;
  WhiteboxInputs      : SEQ OF ReceiveDataRef;
  IndirectOutputs     : SEQ OF Ident;
  DirectOutputs       : SEQ OF Ident;
  WhiteboxOutputs     : SEQ OF SendDataRef;
  Subclusters         : SEQ OF ClusterNode;
  Couplings           : SEQ OF ClusterCoupling;
  Level               : CommunicationLevel;
  EvaluationTimes     : INTEGER;
  MappedTo            : INTEGER;
  DirectLinkEvalOrder : COPY OF SEQ OF ClusterDirectLinkPart;
END
```

```
STRUCTURE CommunicationLevel IS
  Level : INTEGER;
END

STRUCTURE ClusterDirectLinkPart IS
  RecvInputs   : SEQ OF ReceiveDataRef;
  SetInputs    : SEQ OF ClusterCouplingRef;
  CalcOutputs  : SEQ OF ClusterRef;
  WriteOutputs : SEQ OF ClusterCouplingRef;
  SendOutputs  : SEQ OF SendDataRef;
END

STRUCTURE BlockHierarchy IS
  Identity    : Block;
  Basicblocks : SEQ OF Block;
END

STRUCTURE Block IS
  Subsystem : SEQ OF INTEGER;
END

STRUCTURE ClusterCoupling IS
  Target  : DirectInputRef  | IndirectInputRef  | SendDataRef;
  Source  : DirectOutputRef | IndirectOutputRef | ReceiveDataRef;
  Mapping : SEQ OF IndexMap;
END

STRUCTURE ClusterCouplingRef IS
  Index : INTEGER;
END

STRUCTURE DirectInputRef IS
  Cluster : ClusterRef;
END

STRUCTURE IndirectInputRef IS
  Cluster : ClusterRef;
END

STRUCTURE DirectOutputRef IS
  Cluster : ClusterRef;
END
```

```
STRUCTURE IndirectOutputRef IS
  Cluster : ClusterRef;
END

STRUCTURE ClusterRef IS
  Subcluster : SEQ OF INTEGER;
END

STRUCTURE IndexMap IS
  TargetIndex : INTEGER;
  SourceIndex : INTEGER;
END

STRUCTURE Assign IS
  Variable   : Ident;
  Expression : Expr;
END

STRUCTURE Expr IS
  Kind : Function | Ident | REAL | INTEGER | STRING;
END

STRUCTURE Function IS
  Name      : Ident;
  Arguments : SEQ OF Expr;
END

STRUCTURE Ident IS
  Kind : T | P | U | Y | X | dX | A | A0 | F | E;
END

T IS REFERENCE TO Time
END

P IS REFERENCE TO Param
END

U IS REFERENCE TO Input
END

Y IS REFERENCE TO Output
END
```

```
X IS REFERENCE TO State
END

dX IS REFERENCE TO State
END

A IS REFERENCE TO Aux
END

A0 IS REFERENCE TO Aux
END

F IS REFERENCE TO Func
END

E IS REFERENCE TO Param, Input, Output, State IN SCOPE DSC
  KEY Param.Name, Input.Name, Output.Name, State.Name
END

STRUCTURE AllP IS
  NoInformation : SEQ OF STRING;
END

STRUCTURE AllU IS
  NoInformation : SEQ OF STRING;
END

STRUCTURE AllY IS
  NoInformation : SEQ OF STRING;
END

STRUCTURE Nil IS
  NoInformation : SEQ OF STRING;
END

REPRESENTED STRUCTURE MPONameServer IS NEW SCOPE
  RegisteredMPOs : SEQ OF DSC;
END

MPO IS REFERENCE TO DSC IN SCOPE MPONameServer
  KEY DSC.SystemName
END
```