Konzepte für die Modellierung und Simulation strukturvariabler Modelle

vorgelegt von
Dipl.-Ing.
Alexandra Mehlhase
geb. in Berlin

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktorin der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsauschuss:

Vorsitzender:	Prof. Dr. Rolf Niedermeier
Gutachter:	Prof. Dr. Peter Pepper
	Prof. Dr. François Cellier
	Prof. Dr. Stefan Jähnichen

Tag der wissenschaftlichen Aussprache: 3. Juni 2015

Berlin 2015

Danke

Die vorliegende Arbeit wäre nicht entstanden ohne die Hilfe von vielen lieben Menschen in meinem Leben, die mich und meine Arbeit auf die eine oder andere Art beeinflusst haben.

Zunächst geht mein Dank an meine zwei lieben Chefs Stefan Jähnichen und Peter Pepper, die mich überhaupt auf die Idee einer Promotion gebracht haben. Beide haben an mich geglaubt, mir viel beigebracht und mich immer unterstützt. Die Pepper-Jähnichen-Mehlhase Show in der MoSim Veranstaltung werde ich nie vergessen.

Einen lieben Dank auch an François Cellier, der mich mit seiner Expertise nicht nur einmal zurück auf den richtigen Weg gebracht hat.

Ein großer Dank geht auch an all meine Kollegen (aus SWT und ÜBB) mit denen ich gearbeitet, diskutiert und eine tolle Zeit verbracht habe.

Einigen Kollegen möchte ich noch besonders danken: Danke Gabi Ambach, dass du immer für ein Schwätzchen da warst und aufgepasst hast, dass ich meine Anträge richtig ausfülle. Danke Rodger Burmeister für die Organisation von Spieleabenden, lustigen und manchmal nervenaufreibenden Diskussionen :-)
Ebenso einen besonderen Dank an alle Kollegen, die mir mit konstruktiver Kritik und Hilfe stets zur Seite standen: Björn Bartels, Daniel Gomez-Esperon, Christoph Höger, Steffen Helke, Andreas Mertgen, Thomas Karbe und Alexander Rein-Jury.

Aber auch meine studentischen Kollegen möchte ich an dieser Stelle nicht vergessen. Ohne Julien Bergmann, Leif Bonorden, Amir Czwink und Daniel Steinhaus wäre wohl noch der eine oder andere Fehler unentdeckt und andere Arbeiten liegen geblieben. Es war toll mit euch zu arbeiten.

Christoph Nytsch-Geusen danke ich für seine guten Hinweise und für das Lesen meiner Arbeit. Aber auch andere externe Kollegen haben mich mit konstruktiver Kritik und Beispielen unterstützt – vielen Dank: Markus Andres, Anna Jahnke, Imke Krüger, Jens Möckel, Thomas Schmitt und Dirk Zimmer.

Danke an meine besten Freundinnen Stefanie Gutschow und Beatrice Winkel, dass ihr immer an mich geglaubt habt und da wart, wenn ich mal nicht weiter kam. Stefanie Gribat danke ich zusätzlich dafür, dass sie meine Arbeit auf ihren Flügen gelesen und hilfreiches Feedback gegeben hat. Meiner Taekwondo-Gruppe danke ich für den nötigen sportlichen Ausgleich und weil sie eine super Truppe ist.

Meiner Familie Astrid, Wolfgang und Lea Gerndt danke ich für ihre Unterstützung in allen Lebenslagen und dass sie während aller Höhen und Tiefen für mich da waren. Ohne euch hätte ich es nie geschafft.

Kurzfassung

Die Modellierung und Simulation ist ein wichtiger Bestandteil der Weiter- und Neuentwicklung technischer Systeme. In der vorliegenden Arbeit beschreibt ein Modell das vereinfachte, zeitliche Verhalten eines Systems durch mathematische Gleichungen. Das konkrete Verhalten des Modells für einen Zeitraum wird in einer Simulation ermittelt, wodurch Rückschlüsse auf das reale System möglich sind.

Während eines Betrachtungszeitraums durchlaufen technische Systeme oft mehrere Phasen: So rollt beispielsweise ein Flugzeug zunächst auf der Startbahn, begibt sich in einen Steigflug und bleibt dann in einer konstanten Flughöhe. In diesen Phasen sind oft unterschiedliche physikalische Gesetze zum Beschreiben des Verhaltens von Interesse, wodurch die Simulation mit nur einem Modell häufig nicht ausreicht.

Die vorliegende Arbeit beschäftigt sich mit Simulationen, die ebenfalls verschiedene Phasen durchlaufen und reale Phasen genauer abbilden. Die Phasen werden durch Modi repräsentiert, die wiederum Modelle sind und die Phasen mathematisch beschreiben. Zwischen diesen Modi wird während der Simulation situationsbedingt gewechselt. Modelle, die einen Phasenwechsel erlauben, werden strukturvariable Modelle genannt. In heutigen Simulationswerkzeugen werden strukturvariable Modelle nur eingeschränkt unterstützt oder erfordern einen hohen Implementierungsaufwand. Über Eigenschaften, konkrete Modellierungskonzepte und den vorteilhaften Einsatz dieser Modelle ist noch wenig bekannt.

Um die Bedeutung strukturvariabler Modelle zu untersuchen, werden zunächst die Anwendungsgebiete für diese Modelle identifiziert. Darauf folgend werden Konzepte vorgestellt, die beschreiben, wie strukturvariable Modelle aufgebaut sein sollten, um vorteilhaft in diesen Anwendungsgebieten eingesetzt zu werden. Dabei steht der hierarchische und modulare Aufbau der Modelle im Vordergrund, damit diese möglichst vielseitig einsetzbar und wiederverwendbar sind.

Es wird ein formales Modell vorgestellt, das die Semantik strukturvariabler Modelle präzise beschreibt und als Grundlage zur Beschreibung und Simulation dieser Modelle dient. Darauf basierend wird ein prototypisches Framework *DySMo* (**Dy**namic **S**tructure **Mo**deling) implementiert. Mit diesem wird gezeigt, dass strukturvariable Modelle durch die Anwendung der diskutierten Modellierungskonzepte vorteilhaft eingesetzt werden können.

Es werden somit Konzepte für Modellierer bereitgestellt, damit sie strukturvariable Modelle vorteilhaft für ihre Modellierungsvorhaben erstellen und einsetzten können.

Abstract

Modeling and simulation are an important part of the development and refinement of technical systems. For the purpose of this thesis, models describe the simplified behavior of a system using mathematical equations. A simulation determines the specific behavior of the model for a given time period, which allows conclusions to be drawn concerning the behavior of the real system.

During an observation period, technical systems often go through several phases. For example, an airplane moves on the runway, then takes off, and then remains at a constant altitude. The equations of interest describing the behavior differ across these phases, meaning simulation with a single model is often insufficient.

This thesis deals with models that go through different phases during simulation and thus can represent the real phases in more detail. The phases are represented by modes that are themselves models and describe each phase mathematically. The model can switch between these modes depending on circumstances. A model that consists of multiple modes and allows for transitions between them is called a variable-structure model. In today's simulation tools, the handling of variable-structure models is still limited. The properties, modeling concepts, and advantageous uses of these models are currently not well known.

In order to investigate the benefits of variable-structure models, the application areas for these models are identified. Concepts for building variable-structure models are introduced, along with ways to use them in the application areas advantageously. Hierarchical and modular modelling is considered in detail, to ensure the models are versatile and reusable.

A formal model describes the semantics of variable-structure models and serves as a basis for their description and simulation. Based on this formalism, a prototypical framework *DySMo* (**Dy**namic **S**tructure **Mo**deling) is implemented. It serves to demonstrate the beneficial applications of variable-structure modeling by employing the concepts discussed in this thesis.

Thus, this thesis provides concepts for modelers to create and use variable-structure models advantageous for their current modeling task.

Inhaltsverzeichnis

I

Strukturvariable Modelle

1 Einleitung

So einfach wie möglich. Aber nicht einfacher.

(Albert Einstein)

1.1. Motivation

Ein Grundsatz, wie der von Einstein genannte „So einfach wie möglich. Aber nicht einfacher.“[1], ist auf viele Bereiche des Lebens übertragbar, so auch auf die Ingenieurwissenschaften.

Ein wesentlicher Bereich in den Ingenieurdisziplinen befasst sich mit der Untersuchung des zeitlichen Verhaltens von technischen Systemen, um diese zu verbessern oder auch neu zu entwickeln. Da Experimente an technischen Systemen oft kostenintensiv, unmöglich oder auch gefährlich sind, werden Modelle eingesetzt, die das Systemverhalten abstrahiert abbilden. Diese Modelle werden als Ersatzsystem verwendet, um konkrete Fragestellungen zum Systemverhalten zu beantworten.

Das Modell sollte dabei so gewählt werden, dass es die wesentlichen Aspekte des realen Systems abbildet, um die zu untersuchenden Fragestellungen zu beantworten. Ein Modell ist demnach zunächst nur für die Fragestellungen, für die es entwickelt wurde, valide. Für andere Fragestellungen ist es meist unzureichend.

In der vorliegenden Arbeit werden mathematische Modelle betrachtet, die das zeitliche Verhalten von Systemen mit differential-algebraischen Gleichungen beschreiben. In einer Simulation wird das konkrete Verhalten eines Modells für vorgegebene Randbedingungen mit numerischen Lösungsverfahren bestimmt. Mit den Ergebnissen der Simulation können dann Aussagen über das reale Systemverhalten getroffen werden.

[1] Es ist nicht belegt, ob dieses Zitat tatsächlich von Einstein stammt, es wird jedoch meist mit ihm in Verbindung gebracht.

In einer Simulation wird in der Regel ein Modell mit einem festgelegten Abstraktionsgrad verwendet. Systeme verhalten sich jedoch situationsbedingt und in verschiedenen Phasen häufig unterschiedlich. Ist dies der Fall, so beschreibt ein einzelnes Modell das Systemverhalten oft ungenügend. Die Modelle befolgen den oben genannten Grundsatz der Einfachheit über den gesamten Zeitraum der Simulation. Für einzelne Abschnitte ist der Grundsatz aber oft nicht erfüllt. Eine Aufteilung in situationsabhängige Teilprobleme wäre vorteilhaft.

Die Aufteilung kann über die Verwendung verschiedener Modelle erfolgen, zwischen denen während der Simulation situationsbedingt gewechselt wird. Diese verschiedenen Modelle werden Modi genannt und repräsentieren das Systemverhalten unter Verwendung eines eigenen Gleichungssystems in einem definierten Zeitabschnitt. Während der Simulation wechselt das Modell zwischen den einzelnen Modi in Abhängigkeit von der aktuellen Situation. Auf diese Weise wird das Modell dynamisch während einer Simulation den wechselnden Anforderungen angepasst und genügt zu jedem Zeitpunkt dem Grundsatz der Einfachheit. Modelle dieser Art werden *strukturvariable Modelle* genannt.

Die Modellierung und Simulation von strukturvariablen Modellen werden in heutigen Standardwerkzeugen wie Simulink oder Dymola nur unzureichend unterstützt. In diversen experimentellen Modellierungssprachen und Prototypen können strukturvariable Modelle hingegen erstellt und simuliert werden. Die Sprachen und Prototypen unterliegen aber noch Einschränkungen und erfordern die Beschreibung der Modelle in den werkzeuginternen Sprachen. Das Wiederverwenden vorhandener Modelle aus Standardwerkzeugen ist somit nicht möglich.

In der aktuellen Literatur wird die Simulation strukturvariabler Modelle diskutiert und deren Nützlichkeit für die Modellierung beschrieben. Konkrete Modellierungskonzepte werden jedoch nicht erläutert. Es wird nicht untersucht, wie die Modelle aufgebaut sein müssen, um sie vorteilhaft im Vergleich zu konventionellen Modellen einzusetzen, in welchen Anwendungsgebieten sie eingesetzt werden sollten und welche neuen Konzepte für deren Modellierung erforderlich sind.

Bei der strukturvariablen Modellierung sind viele neue Aspekte zu berücksichtigen, so müssen beispielsweise die Wechsel zwischen den Modi betrachtet und modelliert werden. Bei diesen Wechseln müssen unter anderem die Initialisierung des nächsten Modus und das Finden von adäquaten Wechselbedingungen berücksichtigt werden.

Konventionelle Modelle bestehen meist aus mehreren Komponenten, die

miteinander interagieren, und Komponenten können weitere Komponenten enthalten. Dieses Konzept sollte für strukturvariable Modelle ebenfalls vorgesehen werden, damit die erstellten Modi und deren Moduswechsel wiederverwendet werden können. Dies bedarf wiederum der Beachtung von Kompatibilitäten zwischen Komponenten und deren Modi. Die strukturvariable Modellierung bringt neue Herausforderungen im Vergleich zur konventionellen Modellierung mit sich und bedarf einer detaillierten Betrachtung, um diese Modelle vorteilhaft einsetzen zu können.

1.2. Ziele der Arbeit

Die vorliegende Arbeit verfolgt das Ziel, den Umgang mit strukturvariablen Modellen zu untersuchen und Konzepte zur werkzeugunabhängigen Modellierung bereitzustellen. Mit diesen neuen Konzepten sollen Modelle zukünftig während der Simulation den Gegebenheiten angepasst werden können und damit dem Grundsatz der Einfachheit besser entsprechen. Die Arbeit hat folgende Teilziele:

1. **Es sollen mögliche Anwendungsgebiete für strukturvariable Modelle identifiziert werden.** Dazu soll auf die Ziele bei der Verwendung strukturvariabler Modelle in den jeweiligen Anwendungsgebieten eingegangen werden. Es ist zu zeigen, welche Herausforderungen bei der Modellierung auftreten und wie mit ihnen umgegangen werden kann, um die gewünschten Ziele zu erreichen.
2. **Die Eigenschaften strukturvariabler Modelle sollen werkzeugunabhängig und universell zusammengestellt werden.** Dazu soll untersucht werden, welche Modelleigenschaften zum Beschreiben der Modelle notwendig sind. Es sollen Modellierungskonzepte zum Entwurf der Modelle diskutiert werden, besonders im Hinblick auf deren hierarchischen und modularen Aufbau.
3. **Die Eigenschaften und die Simulation sollen präzise beschrieben und durch eine graphische Notation dargestellt werden.** Dazu sollen die Erkenntnisse der vorangegangenen Analyse verwendet und mittels der Z-Notation formalisiert werden. Um die Formalisierung verständlich zu halten, soll parallel eine graphische Notation beschrieben werden. Anhand der Formalisierung sollen die Anforderungen an ein Simulationswerkzeug spezifiziert werden, damit dieses für strukturvariable Modelle verwendet werden kann.

4. **Es soll ein Framework entwickelt werden, mit dem strukturvariable Modelle möglichst einfach beschrieben und simuliert werden können.** Dazu soll auf Basis der Formalisierung ein Framework entworfen und implementiert werden. Dieses soll mehrere Simulationswerkzeuge einbinden und sie für die Simulation der Modi verwenden.
5. **Es sollen realitätsnahe Modelle aus den identifizierten Anwendungsgebieten implementiert und evaluiert werden.** Dazu sollen die Modelle mit den vorgestellten Modellierungskonzepten aufgebaut werden. Die Modelle sollen daraufhin im Framework implementiert und simuliert werden. Anschließend soll anhand der Ergebnisse erörtert werden, ob die Umsetzung als strukturvariables Modell im Vergleich zu einem konventionellen Modell vorteilhaft ist.

1.3. Aufbau der Arbeit

Die vorliegende Arbeit beginnt in Kapitel 2 mit den Grundlagen der Modellierung und Simulation. Anhand eines Beispiels wird erläutert, wie Modelle in heutigen Standardwerkzeugen aufgebaut und simuliert werden und warum neue Konzepte in der Modellierung notwendig sind.

In Kapitel 3 folgt die Einführung strukturvariabler Modelle. Es wird auf die verschiedenen Terminologien der Literatur eingegangen und eine Definition für die vorliegende Arbeit gegeben. Ebenso wird die Relevanz strukturvariabler Modelle erläutert und ein kurzer Abriss über die Herausforderungen gegeben, um darzustellen, warum diese Modelle heute kaum verwendet werden. Es folgt eine Zusammenfassung bereits bekannter Ansätze für die Modellierung, Notation und Simulation strukturvariabler Modelle.

In Kapitel 4 werden Anwendungsgebiete für strukturvariable Modelle identifiziert. Dabei wird auf die Herausforderungen, die im jeweiligen Anwendungsgebiet bei der Modellierung auftreten, eingegangen. Ebenso werden die positiven und negativen Aspekte bei der Verwendung von strukturvariablen Modellen im jeweiligen Anwendungsgebiet erörtert.

In Kapitel 5 werden die Eigenschaften strukturvariabler Modelle untersucht. Dazu wird der Aufbau strukturvariabler Modelle detailliert betrachtet und analysiert, welche Eigenschaften notwendig sind, um die bereits identifizierten Anwendungsgebiete zu modellieren.

Die Eigenschaften strukturvariabler Modelle werden in Kapitel 6 formal beschrieben. Zum Beschreiben der Modelle wird eine graphische Notation eingeführt. Anhand der Formalisierung werden die Anforderungen spezifiziert, die ein Simulationswerkzeug erfüllen muss, um mit strukturvariablen Modellen umzugehen.

In Kapitel 7 wird das prototypische Framework *DySMo* vorgestellt. Dieses integriert mehrere Simulationswerkzeuge, um diese für die Simulation der Modi eines strukturvariablen Modells einzusetzen. Dabei werden der Softwareentwurf und die Implementierung auf Basis der vorangegangenen Formalisierung vorgestellt.

Eine Fallstudie wird in Kapitel 8 behandelt. In dieser werden für die in Kapitel 4 identifizierten Anwendungsgebiete Modelle vorgestellt, welche mit den vorhandenen Konzepten modelliert und in DySMo simuliert werden. Die Vor- und Nachteile bei der Modellierung mit Strukturwechseln werden diskutiert.

Damit ist der Hauptteil der Arbeit abgeschlossen und es folgt in Kapitel 9 die kritische Betrachtung der Modellierung, Formalisierung/Notation und Simulation strukturvariabler Modelle.

Kapitel 10 fasst die Beiträge der vorliegenden Arbeit zusammen und gibt einen Ausblick auf zukünftige Themen der strukturvariablen Modellierung und Simulation.

2 Grundlagen der Modellierung und Simulation

Ziel der Modellierung und Simulation ist es, das Verhalten von realen Systemen am Rechner nachzubilden, um damit Aussagen über das Systemverhalten zu treffen. Als Basis für diese Arbeit werden zunächst die Grundbegriffe für die Modellierung und Simulation eingeführt.

Darauf folgt ein kurzer Abriss über die historische Entwicklung der Modellierung und Simulation.

Das Vorgehen der Modellierung technischer Systeme mit differential-algebraischen Gleichungen wird anhand eines einfachen Satellitenbeispiels erklärt. Das entworfene Satellitenmodell wird daraufhin in verschiedenen Modellierungswerkzeugen bzw. Modellierungssprachen implementiert, um die Unterschiede der verschiedenen Vorgehensweisen aufzuzeigen.

Nach der Modellierung kann die Simulation in einem Simulationswerkzeug stattfinden. Die notwendigen Schritte einer solchen Simulation werden vorgestellt.

Am Ende dieses Kapitels wird auf die Nachteile der heutigen Vorgehensweise eingegangen und erläutert, warum zukünftig neue Konzepte bei der Modellierung und Simulation notwendig sind.

2.1. Hintergründe und Geschichte

Grundbegriffe

Ein grundlegender, wenn auch sehr allgemeiner und abstrakter Begriff ist der des „Systems“. In der Literatur sind viele Definitionen zu finden, wie beispielsweise:

- „A system is characterized by the fact that we can say what belongs to it and what does not, and by the fact that we can specify how it interacts with its environment. (...) Another property of a system is the fact that it can be controlled and observed.“ [13]

- "Ein System ist dadurch gekennzeichnet, dass es von seiner Umgebung abgegrenzt ist, wobei die Verbindungen zur Umgebung – die Eingangs- und Ausgangsgrößen – von der Systemgrenze geschnitten werden." [65]
- „A system is an object or collection of objects whose properties we want to study." [31]

Aus diesen Definitionen können die wichtigsten Eigenschaften abgeleitet werden.

System: *Ein System muss von seiner Umwelt abgegrenzt werden. Diese Grenze wird Systemgrenze genannt und sollte nur die den Benutzer interessierenden Aspekte umschließen. Innerhalb der Systemgrenze hat ein System einen inneren Zustand $\underline{z}(t)$ und kann über Eingangsgrößen $\underline{x}(t)$ und Ausgangsgrößen $\underline{y}(t)$ mit seiner Umwelt interagieren.*

Systeme können aus verschiedenen Perspektiven betrachtet werden. Ein Ball kann beispielsweise nur auf seine Materialeigenschaften oder auch auf seine Bewegung nach einem Anstoß untersucht werden. Die Systemgrenze legt demnach nicht nur fest, welches System betrachtet wird, sondern auch, welche Aspekte von Interesse sind.

Komponente (System): *Systeme sind häufig aus mehreren Komponenten aufgebaut, die ebenfalls eigene Systeme darstellen. Jede Komponente kann wiederum aus weiteren Komponenten bestehen. Solch ein Aufbau wird als modularer und hierarchischer Aufbau bezeichnet.*

Können die inneren Zustände des Systems nicht durch die Eingangsgrößen beeinflusst werden, ist das System *nicht vollständig steuerbar*. Können die inneren Zustände nicht aus Messungen der Ausgangsgrößen ermittelt werden, so ist das System *nicht vollständig beobachtbar*.

Experiment: *Mit Experimenten wird das Systemverhalten unter dem Einfluss gewählter Eingangsgrößen untersucht. Dazu sollte das System (mindestens partiell) beobachtbar sein. Sollen Experimente mit verschiedenen Stimuli durchgeführt werden, so muss das System zusätzlich steuerbar sein.*

Experimente sind oft teuer und aufwendig. Häufig sind die Systeme nicht vollständig steuer- und beobachtbar, was deren Untersuchung erschwert. Des Weiteren können Experimente zu gefährlich sein und manche

Systeme existieren noch nicht (bspw. neue Motorkonzepte), wodurch ein Experiment am realen System unmöglich ist.

Um diese Probleme zu umgehen, werden Modelle eingesetzt. Nach [79] ist ein Modell durch mindestens drei Merkmale gekennzeichnet:[1]

1. Abbildung – Ein Modell ist stets eine Abbildung oder Repräsentation eines natürlichen oder künstlichen Originals, das selbst wieder ein Modell sein kann.

2. Verkürzung – Ein Modell erfasst im Allgemeinen nicht alle Attribute des Originals, sondern nur diejenigen, die relevant sind.

3. Pragmatismus – Modelle sind ihren Originalen nicht per se eindeutig zugeordnet. Sie erfüllen ihre Ersetzungsfunktion
   a) für bestimmte Subjekte,
   b) innerhalb bestimmter Zeitintervalle und
   c) unter Einschränkung auf bestimmte gedankliche oder tatsächliche Operationen.

Beispiel 2.1 (System vs. Modell): *Wird das reale System 'Auto' betrachtet, so kann von diesem ein vereinfachtes und/oder verkleinertes Holzmodell erstellt werden. Dies wird als physikalisches Modell bezeichnet. Solch ein Holzmodell kann für aerodynamische Untersuchungen verwendet werden. Dieses Holzmodell ist aber wiederum ein reales System, von dem ein mathematisches Modell, also ein Modell, das auf physikalischen und mathematischen Gleichungen basiert, erstellt werden kann.*

Modell: *Ein Modell wird als eine vereinfachte, mathematische Abstraktion eines Originals (Systems) angenommen und genügt den Merkmalen 1–3. Ein Modell kann, wie auch ein System, durch Eingangsgrößen $\underline{x}(t)$, Ausgangsgrößen $\underline{y}(t)$ und interne Zustandsvariablen $\underline{z}(t)$ beschrieben werden. Das Verhalten der Zustandsvariablen und Ausgangsgrößen wird durch eine Menge von Gleichungen beschrieben. Das Verhalten des Modells ist durch seine Zustandsvariablen eindeutig bestimmt. Diese Zustandsvariablen können nicht ineinander umgerechnet werden und jede Zustandsvariable ist notwendig, um das Modellverhalten zu beschreiben [34].*

[1] frei nach [79] zitiert

Das zeitliche Systemverhalten kann mathematisch durch Differential- ($\underline{\dot{v}}(t) = \underline{f_d}(\underline{v}(t), t)$) und algebraische Gleichungen ($0 = f_a(v(t), t)$) beschrieben werden.

Dabei beschreiben die Differentialgleichungen meist die Zustandsvariablen, während die algebraischen Gleichungen oft zur Berechnung der Ausgangsgrößen dienen. Manchmal hängt die Lösung der Differentialgleichungen von den algebraischen Gleichungen ab. Dies wird als Gleichungssystem mit einem Index größer als Eins bezeichnet oder auch als Modell mit strukturellen Singularitäten.

Differential-algebraische Gleichungssysteme: *Die Gleichungssysteme, die sowohl aus differentiellen als auch aus algebraischen Gleichungen bestehen, werden differential-algebraische Gleichungssysteme (DAE - Differential-algebraic equations) genannt.*

Modelle können, wie auch die zugehörigen technischen Systeme, komponentenweise aufgebaut werden. Jede Komponente hat ihre eigenen Variablen und Gleichungen und kann über Schnittstellen mit anderen Komponenten verbunden werden. Der Verbund der Komponenten ergibt das Gesamtmodell. Komponenten können wiederum aus Komponenten bestehen, wodurch auch in Modellen ein hierarchischer Aufbau entsteht.

Komponenten (Modellierung): *Ein Modell kann aus Komponenten aufgebaut werden. Dabei ist jede Komponente ein eigenes Modell, welches weitere Komponenten enthalten kann. Modelle können also wie Systeme modular und hierarchisch aufgebaut werden. Komponenten auf einer Hierarchiebene können durch ihre Schnittstellen (Eingangs- und Ausgangsgrößen) mit anderen Komponenten verbunden werden.*

Da das Modell dazu dienen soll, das zeitliche Verhalten eines Systems unter bestimmten Bedingungen vorherzusagen, muss dieses zeitliche Verhalten berechnet werden. Dies wird allgemein als Simulation bezeichnet.

Simulation: *Wird an einem Modell ein Experiment durchgeführt, wird dies als Simulation oder virtuelles Experiment bezeichnet.*

Simulationen haben viele Vorteile gegenüber Experimenten am realen System:

- Simulationen sind unabhängig von Störgrößen (Bsp.: Wetter)
- Simulationen sind ungefährlich (Bsp.: Kernschmelze)

- Simulationen sind reproduzierbar (Bsp.: Lösung der Gleichungen liefern gleiche Ergebnisse)
- Simulationen bieten die Möglichkeit, interne Zustände sichtbar zu machen (Bsp.: Temperaturen im Verbrennungsmotor)
- Simulationen bieten die Möglichkeit, Zeitskalen zu ändern (Bsp.: Vergangenes und Zukünftiges ist simulierbar)
- Simulationen bieten die Möglichkeit, viele Szenarien zu untersuchen (Bsp.: Flugzeugparameter können geändert werden)

Die Ergebnisse einer Simulation basieren auf einem Experiment an einem Modell, welches die Realität vereinfacht abbildet. Nicht jedes Modell ist für jedes Experiment geeignet und es kann zu falschen Ergebnissen führen, wenn das Modell außerhalb seiner Gültigkeit verwendet wird (vgl. Merkmale 1–3). Beispielsweise wird Gas oft als ideales Gas behandelt, diese Annahme ist jedoch nur in definierten Temperatur- und Druckregionen gültig. Außerhalb dieser Regionen führt diese Annahme zu Fehlern. Wenn über die Gültigkeit von Simulationsergebnissen gesprochen wird, sollte laut [13] ein Tupel aus Modell und Simulation betrachtet werden. In [64] wird ein *Experimental Frame* vorgesehen, welches wie folgt definiert wird:

> An *Experimental Frame* defines a limited set of circumstances under which the system (real-world or model) is to be observed or subjected to experimentation.

Mit diesem *Experimental Frame* wird beschrieben, unter welchen Bedingungen ein Modell valide Ergebnisse für eine gewünschte Untersuchung liefert.

Entwicklung der Modellierung und Simulation

Die Modellierung von Systemen durch mathematische Zusammenhänge ist laut [73] bei den Ägyptern, Indern und Babyloniern schon seit 2000 v. Chr. bekannt. Dabei wurde die Mathematik vorwiegend zum algorithmischen Lösen von Alltagsproblemen verwendet. Es heißt, Thales legte um 600 v. Chr. die Grundsteine für die Geometrie. Um 500 v. Chr. folgte Pythagoras von Samos, der oft als Gründer der Mathematik, besonders in Bezug auf die Geometrie und Zahlentheorie, angesehen wird.

Viele weitere Mathematiker folgten. So heißt es, dass um 300 v. Chr. Euklid von Alexandria die erste Buchreihe *Stoicheia* (Elemente) schrieb, welche das in dieser Zeit bekannte mathematische Wissen zusammenfasste.

Diophantos von Alexandria, um 250 n. Chr., gilt als bedeutendster Algebraiker der Antike.

Claudius Ptolemäus entwickelte um 150 n. Chr. eine Berechnungsmethode, um die Bahnen von Planeten im Sonnensystem zu berechnen. Die Methode wurde noch bis 1619 verwendet, bis Johannes Kepler eine bessere, einfachere Variante zur Berechnung fand. Mit Verfeinerungen durch Newton und später auch Einstein gilt diese Variante heute immer noch als valide.

Das Problem war jedoch die Berechnung der Modelle, da diese auf dem Papier erfolgen musste. Laut [72] begann um 1675 mit Gottfried Wilhelm von Leibniz die Zeit, in der Modelle mit Differentialgleichungen beschrieben wurden. Isaac Newton legte den Grundstein zum Lösen dieser Differentialgleichungen gegen 1670.

Numerische Verfahren zum Lösen von Differentialgleichungen wurden durch das Newton-Verfahren (um 1670) und das Euler-Verfahren (um 1760) möglich. Diese mussten ebenso von Hand gelöst werden, was aufwendig und oft praktisch unmöglich war.

Um 1900 entwickelten Karl Heun, Martin Wilhelm Kutta und Carl Runge die Runge-Kutta-Verfahren (relevante numerische Verfahren werden in Abschnitt 2.3 näher erläutert).

Zwischen 1920 und 1950 wurden laut [2] analoge Modelle erstellt. Dafür wurde zunächst ein auf Differentialgleichungen basierendes Modell erstellt. Für dieses wurde dann ein reales Ersatzsystem entwickelt, welches das eigentliche System für Untersuchungszwecke ersetzte.

Erst mit der Einführung digitaler Computer konnten Modelle digitalisiert numerisch gelöst werden. Um 1970/80 wurden Modelle aus einfachen Gleichungssystemen mit wenigen Differentialgleichungen (laut [14] ca. ein halbes Dutzend) und einigen algebraischen Gleichungen erstellt und numerisch gelöst.

Heute zählen Modelica-Simulationsumgebungen (Modelica seit 1997 als offener Standard) und Simulink (seit 1984) zu den häufig verwendeten Werkzeugen zur Modellierung und Simulation. Sie ermöglichen es, komplizierte Modelle aus mehreren Komponenten zu erstellen. Typische Modelle bestehen heute aus mehreren Dutzend Differentialgleichungen und vielen tausend algebraischen Gleichungen [14].

Je größer und komplexer das Gleichungssystem wird, desto länger ist meist auch die Simulationszeit. Da in der heutigen Entwicklung und Erprobung technischer Systeme Zeit eine immer größere Rolle spielt, ist es wichtig, unnötige Simulationszeiten zu vermeiden. Der Grundsatz der Einfachheit gilt auch hier: Ein Modell sollte zu jeder Zeit nur so genau wie nötig sein, um die Fragestellung, für die es entwickelt wurde, zu beantworten.

Zukünftig sind Modelle zu erwarten, die standardmäßig aus hunderten von Differential- und tausenden von algebraischen Gleichungen bestehen. Es ist daher von längeren Simulationszeiten und komplizierteren Berechnungen auszugehen.

Um besser über die Zukunft der Modellierung und Simulation sprechen zu können, wird zunächst das heutige Vorgehen vorgestellt. Abbildung 2.1 zeigt ein solches Vorgehen.

Dabei findet zunächst die Problemidentifikation statt. In dieser muss festgelegt werden, unter welchem Aspekt ein System untersucht werden soll und welche Systemeigenschaften von Interesse sind. Dies führt zu einer Reihe von Anforderungen, die zur Modellierung des Systems verwendet werden. In Abschnitt 2.2 wird konkreter auf die Modellierung eingegangen. Ist das Modell erstellt, folgt die Simulation auf einem Computer, siehe Abschnitt 2.3. Nach der Simulation folgt die Auswertung der Ergebnisse. Dabei ist unter anderem zu untersuchen, ob die Ergebnisse plausibel erscheinen und ob die Anforderungen aus der Problemidentifikation erfüllt werden.

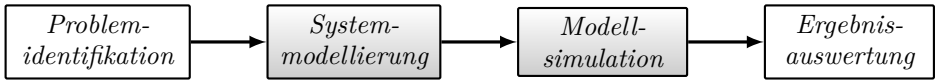


Abb. 2.1.: Grobes Vorgehen bei der Modellierung und Simulation

2.2. Systemmodellierung

In diesem Abschnitt wird das Vorgehen zur Modellierung technischer Systeme betrachtet. Es wird vorausgesetzt, dass die Problemidentifikation bereits abgeschlossen ist. Damit ist bekannt, welches System und welche Aspekte modelliert werden sollen. Die Aufgabe bei der Systemmodellierung ist es, eine geeignete Abstraktion zu finden, welche eine gegebene Anforderung erfüllt.

In [84] wird das Modellierungsproblem beschrieben als:

> The modelling problem is to construct the most simple artifact that allows an adequate answer to a given query.

Abbildung 2.2 gibt eine graphische Übersicht über die Schritte während der Modellierung.

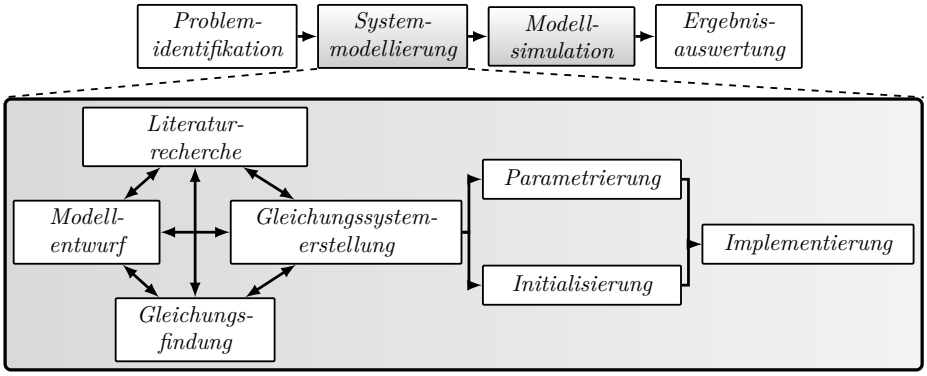


Abb. 2.2.: Grobe Schritte bei der Modellierung

Im *Modellentwurf* wird die Systemgrenze festgelegt und damit bestimmt, welche Aspekte in welchem Detaillierungsgrad berücksichtigt werden. Eine detaillierte Aufführung, was bei der Modellierung zu beachten ist, kann in [13, 84] nachgeschlagen werden.

In dieser Phase wird ebenfalls festgelegt, ob das Modell aus Komponenten zusammengesetzt wird, und falls dies der Fall ist, wie diese Komponenten miteinander interagieren. Die Wahl der Schnittstellen von einzelnen Komponenten ist dabei eine wichtige Entscheidung, da diese über die Wiederverwendbarkeit und Verständlichkeit der Komponente entscheidet.

Nachdem diese konzeptionellen Entscheidungen getroffen wurden, müssen *Gleichungen gefunden* werden, die das System- bzw. das Komponentenverhalten beschreiben. Beide vorgestellten Schritte werden von der *Literaturrecherche* begleitet. In dieser wird beispielsweise nach bereits vorhandenen Ansätzen gesucht und auch nach Gleichungen, die für das Modell und dessen Komponenten notwendig sind.

Sind diese Phasen abgeschlossen, kann das gesamte *Gleichungssystem aufgestellt* werden. Das heißt, die Komponenten werden zusammengesetzt und die Schnittstellen verbunden. Dabei können Iterationen und Schleifen in diesen drei Schritten auftreten, da neue Erkenntnisse zu Änderungen in

den vorherigen Schritten führen können. Die Problemstellung darf dabei nicht vergessen werden, um dem Grundsatz der Einfachheit zu genügen.

Vor einer Simulation eines Modells muss es *parametriert* und *initialisiert* werden. Während der Modellierung sollte daher schon gekennzeichnet werden, wie Parameter und Initialwerte gewählt werden sollten. Parameter sind in einem Modell Werte, die von Simulation zu Simulation unterschiedlich sein können, aber während der Simulation zeitlich unveränderlich sind. Initialwerte beschreiben den initialen Zustand des Modells. In Abschnitt 2.3 wird erläutert, warum eine beliebige Initialisierung nicht immer möglich ist.

Wenn alle Informationen des Modells bekannt sind, kann es in einem Modellierungswerkzeug *implementiert* werden. Anhand der vorangegangenen Modellierung kann entschieden werden, welches Werkzeug am geeignetsten ist.

Das vorgestellte Vorgehen wird im Folgenden anhand eines einfachen Beispiels gezeigt. Dazu wird zunächst das Beispiel komplett eingeführt und danach erklärt, wie das Modell erstellt wurde. Anschließend wird gezeigt, wie das Modell in unterschiedlichen Werkzeugen implementiert werden kann.

Beispiel 2.2 (Satellit): *Betrachtet wird ein einfacher Satellit, der im Orbit um einen Planeten kreist.*[2] *Die Flugbahn des Satelliten sei außerhalb der Atmosphäre, somit herrscht kein Luftwiderstand. Je nachdem, in welcher Höhe und mit welcher Geschwindigkeit der Satellit startet, werden unterschiedliche Bahnen eingenommen. Es wird folgende Problemidentifikation angenommen:*

- *Die Bahn des Satelliten um einen Planeten soll berechnet werden.*
- *Der Satellit ist antriebslos.*
- *Der Satellit soll mit verschiedenen Geschwindigkeiten und Abständen zum Planeten initialisiert werden können.*
- *Der Satellit soll um verschiedene kugelförmige Planeten fliegen können.*
- *Der Satellit soll so initialisiert werden, dass seine Flugbahn einer Kreisbahn entspricht.*

[2] Die Gleichungen und Annahmen für das Satellitenmodell sind dem Lehrbuch [52] entnommen.

Das Modell wird als Zweikörpersystem der klassischen (Newtonschen) Mechanik angenommen. Abbildung 2.3 zeigt eine abstrakte Darstellung des zu modellierenden Problems.

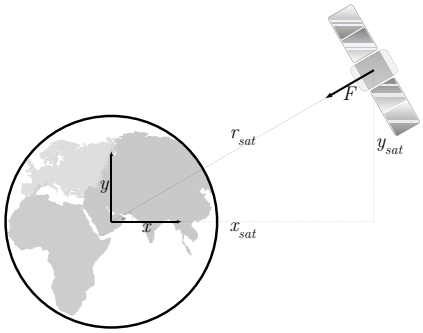


Abb. 2.3.: Angreifende Kraft F an einem Satelliten

Die Gleichungen für das Modell sind:

$$\left.\begin{aligned} F &= -\frac{M \cdot m}{r_{sat}^2} \cdot G \\ m \cdot a &= F \\ a_x &= a \cdot \frac{x_{sat}}{r_{sat}} \\ a_y &= a \cdot \frac{y_{sat}}{r_{sat}} \\ r_{sat}^2 &= x_{sat}^2 + y_{sat}^2 \end{aligned}\right\} \quad \textit{algebraisch} \tag{2.1}$$

$$\left.\begin{aligned} \dot{v}_x &= a_x \\ \dot{v}_y &= a_y \\ \dot{x}_{sat} &= v_x \\ \dot{y}_{sat} &= v_y \end{aligned}\right\} \quad \textit{differentiell} \tag{2.2}$$

Dabei ist r_{sat} der Abstand des Satelliten vom Erdmittelpunkt und x_{sat} und y_{sat} sind die jeweiligen Abstände in x- und y-Richtung. F ist die auf den Satelliten wirkende Zentripetalkraft, wobei M die Masse des Planeten und G die Gravitation darstellt. Die Variablen v_x, v_y beschreiben die Geschwindigkeiten und a_x, a_y die Beschleunigung in x- bzw. y-Richtung.

Im Folgenden wird beschrieben, wie dieses Modell durch die Systemmodellierung zustande kommt.

Modellentwurf

Wird ein Satellit betrachtet, können verschiedene Effekte berücksichtigt werden, beispielsweise das Energiemanagement, die Geschwindigkeit, die Rotation, die Beschleunigung, die Anziehungskräfte von Satelliten untereinander (da zwei Körper immer Kräfte aufeinander ausüben), die Anziehungskräfte von anderen Planeten, usw. Diese Liste wäre beliebig erweiterbar. Der Einfachheit halber wird festgelegt, nur ein Zweikörperproblem der klassischen (Newtonschen) Mechanik zu betrachten. Das bedeutet, die Satellitenbewegung wird nur von einem Planeten beeinflusst, und zwar von dem, um den der Satellit kreist.

Der Planet wird dabei als unbewegliche, kugelförmige Masse angenommen. Als Komponenten für das Modell bieten sich der Satellit und der Planet an.

Durch die gewählte Systemgrenze können jetzt die Gleichungen für die jeweiligen Komponenten erstellt werden.

Gleichungsfindung

Der Planet soll sich in diesem Modell nicht bewegen, das Modell besteht somit nur aus den unveränderlichen Parametern *Masse*, *Planetenradius* und *Gravitationskonstante*.

Der Satellit hingegen soll sich um den Planeten herum bewegen. Es sollen dabei nur die Kräfte des Planeten auf ihn wirken. Dafür kann das zweite Newtonsche Gesetz angewendet werden, welches besagt: $m \cdot a = \sum F$. Die Kraft, die auf den Satellit wirkt, ist lediglich die Gravitationskraft des Planeten. Diese ist abhängig von der Masse des Planeten, der Gravitationskonstanten und von dem Abstand des Satelliten zum Planetenmittelpunkt. Diese Kraft kann in die obige Gleichung eingesetzt werden, womit die Beschleunigung in Richtung des Planetenmittelpunkts berechnet werden kann. Diese Beschleunigung wird in ihre x- und y-Komponenten zerlegt. Der Abstand des Satelliten zum Planetenmittelpunkt r_{sat} ergibt sich durch den Satz von Pythagoras. Diese Gleichungen entsprechen den in 2.1 dargestellten.

Die Gleichungen aus 2.2 repräsentieren die Differentialgleichungen des Modells. Sie geben den Zusammenhang zwischen Weg, Geschwindigkeit und Beschleunigung in x- und y-Richtung an.

Die Koordinaten x_{sat} und y_{sat} stellen die Ausgaben des Modells dar.

Gleichungssystemerstellung

Es müssen lediglich die Parameter aus dem Planeten an den Satelliten übergeben werden. Dort werden die Werte verwendet, um die Satellitenbahn zu berechnen.

Damit sind die Komponenten, deren Variablen und Gleichungen sowie der Zusammenhang der Komponenten bekannt.

Parametrierung und Initialisierung

Im nächsten Schritt geht es um die Wahl von Parametern und Startwerten. In einem Modell ist es oft möglich, unterschiedlichen Variablen Startwerte zuzuweisen. Sei die Gleichung $u(t) = R \cdot i(t)$ für einen elektrischen Widerstand betrachtet. Für eine eindeutige Lösung müssen zwei Werte gesetzt werden. Der Parameter R sowie eine der Variablen müssen einen Wert erhalten. Unabhängig davon, welche Variable initialisiert wird, ergibt sich ein eindeutiger Zustand. Werden u und i initialisiert, kann es zu Inkonsistenzen kommen, sofern die Werte nicht der Gleichung entsprechen.

Sei nun konkret das Satellitenmodell betrachtet. Die Parameter des Modells sind M, r und G, welche vom Planeten abhängen. Als Planet für das Modell wird die Erde gewählt. Damit wird die Masse des Planeten als $M = 5,976 \cdot 10^{24}[kg]$ und der Radius $r = 6378 \cdot 10^{3}[m]$ angenommen. Die Gravitationskonstante $G = 6,670 \cdot 10^{-11}[\frac{m^3}{kg \cdot s^2}]$ ist für alle Planeten konstant.

Damit der Satellit um einen Planeten kreisen kann, muss er außerhalb der Atmosphäre platziert werden, denn nur dort gelten die verwendeten Gleichungen. Des Weiteren muss der Satellit eine Startgeschwindigkeit in mindestens eine Richtung erhalten, da er sich sonst durch die Gravitationskraft getrieben nur auf den Planeten zu bewegen kann.

Abbildung 2.4 zeigt schematisch verschiedene Umlaufbahnen eines Satelliten, der im Abstand $r_{sat}(t = 0)$ *(= r + initialHöhe)* vom Planetenmittelpunkt startet, dabei wird von $y_{sat}(t = 0) = 0$ ausgegangen. Damit ergibt sich für den initialen Zustand $x_{sat} = r_{sat}$. Die Startgeschwindigkeit wird in y-Richtung angenommen.

Der Satellit braucht abhängig vom Abstand zum Planetenmittelpunkt eine Mindestgeschwindigkeit, die auch erste kosmische Geschwindigkeit genannt wird, um sich antriebslos in einer Kreisbahn um einen Planeten zu bewegen.

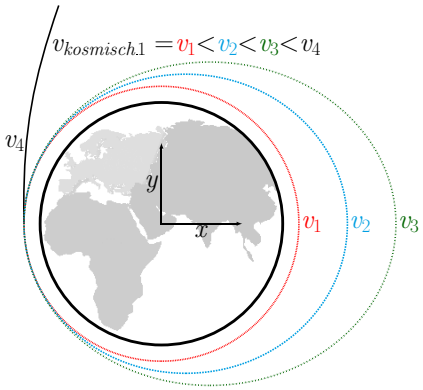


Abb. 2.4.: Schematische Darstellung der Satellitenbahnen in Abhängigkeit der Anfangsgeschwindigkeit

Diese kann wie folgt bestimmt werden:

$$v_{kosmisch1} = \sqrt{\frac{M \cdot G}{r_{sat}}} \tag{2.3}$$

Ist die Geschwindigkeit des Satelliten größer als die erste kosmische Geschwindigkeit, bewegt sich der Satellit auf einer elliptischen Bahn, wie in Abbildung 2.4 zu sehen ist. Ist die Geschwindigkeit größer als die zweite kosmische Geschwindigkeit, so entfernt sich der Satellit aus dem Gravitationsfeld des Planeten. Diese zweite kosmische Geschwindigkeit ist wie folgt definiert:

$$v_{kosmisch2} = \sqrt{\frac{2 \cdot M \cdot G}{r_{sat}}}. \tag{2.4}$$

Abbildung 2.5 zeigt, wie sich der Satellit in Abhängigkeit des initialen Abstands zum Erdmittelpunkt bei gleicher Startgeschwindigkeit bewegt.

Für das vorgestellte Modell sollen verschiedene kreisförmige Umlaufbahnen modelliert werden. Somit müssen die gewählten Initialwerte der Gleichung 2.3 für die erste kosmische Geschwindigkeit genügen.

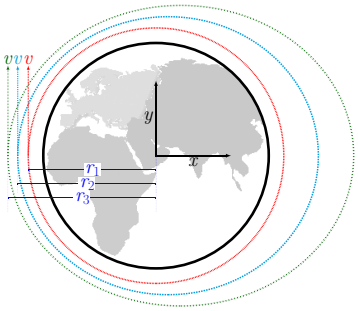


Abb. 2.5.: Schematische Darstellung der Satellitenbahnen in Abhängigkeit der initialen Entfernung zum Planeten

Implementierung

Das Modell wird in Simulink[3][81] und Modelica[4][82] implementiert. Es gibt weitere Modellierungsmöglichkeiten, wie Bondgraphen [13] (auf Energieflüssen basierend), Ptolemy [43, 70] (signalflussorientiert), Matlab/-Simulink/Simscape [80] (objektorientiert). Diese Varianten werden hier nicht näher betrachtet.

Matlab/Simulink

Matlab/Simulink basiert auf einem block- bzw. signalflussorientierten Ansatz. Basis sind dabei Blöcke, die eine definierte Schnittstelle haben und die in jedem Zeitschritt das als Eingang gegebene Signal verarbeiten und ein oder mehrere neue Ausgangssignale ausgeben. Der Modellierer muss sein Gleichungssystem mit Hilfe dieser Blöcke abbilden. Dies wird im Folgenden anhand des Satellitenmodells verdeutlicht.

Zunächst werden zwei Subsysteme erstellt, die die Komponenten des Modells repräsentieren. Diese Subsysteme dienen zur Wahrung der Übersicht, haben aber keinen semantischen Einfluss auf das Modell. In Abbildung 2.6 ist das Matlab/Simulink-Modell abgebildet. In der obersten Ebene sind die beiden Komponenten zu erkennen. Darunter sind die Subsysteme mit ihren Gleichungen dargestellt. Der Planet besitzt keine Gleichungen und stellt nur seine Daten zur Verfügung. Das Satellitenmodell besteht aus mehreren Integrationen, Multiplikationen und Additionen, die durch Blöcke dargestellt sind. Diese repräsentieren die Gleichungen 2.1.

[3]Wenn Simulink im Text verwendet wird, ist immer Matlab/Simulink® gemeint.
[4]Wenn Modelica im Text verwendet wird, ist immer Modelica® gemeint.

Um die Gleichungen zu extrahieren, muss der Signalfluss nachvollzogen werden.

Wird eine Differentialgleichung betrachtet, so kann diese analytisch wie folgt gelöst werden:

$$z(t) = \quad Z_0 + \int_{t_0}^{t} \underbrace{f(z(t), x(t), t)}_{\dot{z}(t)} \, dt \tag{2.5}$$

Für Simulink wird der Integrator verwendet, dessen Eingangssignal $\dot{z}(t)$ ist. Der Startwert Z_0 gibt den initialen Wert zum Zeitpunkt t_0 an.

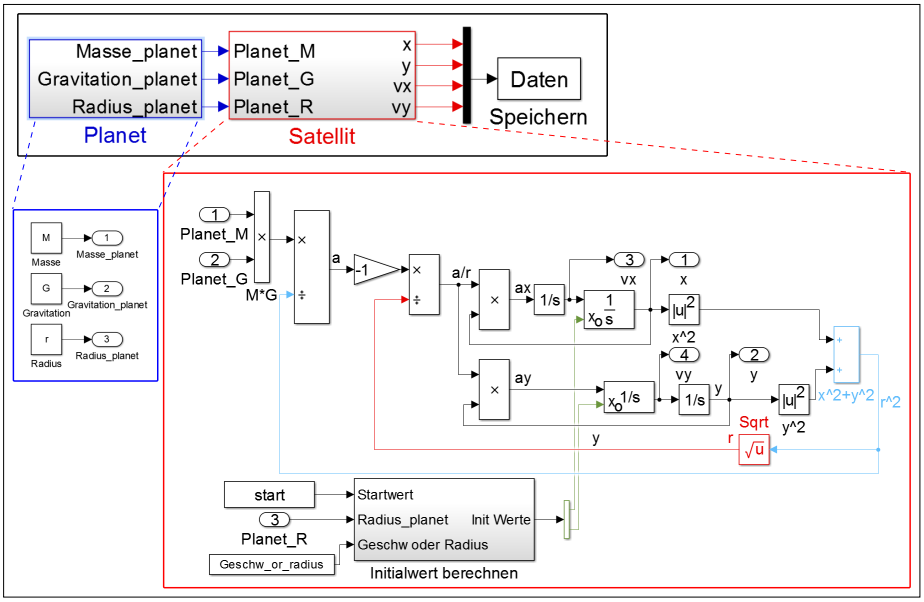


Abb. 2.6.: Satellitenmodell in Matlab/Simulink

Interessant an dieser Stelle ist die Modellierung der Startwerte. Wie bereits erläutert, soll entweder der Abstand oder die Geschwindigkeit des Satelliten vorgegeben werden. Um dies in Simulink zu realisieren, wurde im Satellitenmodell ein zusätzliches Subsystem integriert: *Initialwert berechnen*. Abbildung 2.7 zeigt dieses System. In diesem wird abhängig von dem als Eingabe übergebenen Parameter (*Geschw oder Radius*) entweder die Geschwindigkeit oder der Abstand des Satelliten vom Erdmittelpunkt berechnet. Es wurden zwei Berechnungswege implementiert, obwohl es sich um dieselbe Gleichung handelt, vgl. Gleichung 2.3. Dieses Umstellen nach den gesuchten Variablen wird Kausalisieren genannt.

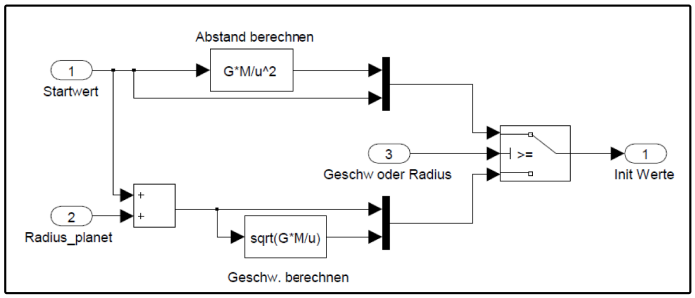


Abb. 2.7.: Initialisierungs-Subsystem des Satellitenmodells

Da bei der Modellierung mit Simulink alle Gleichungen nach den gesuchten Variablen umgestellt sein müssen, wird von kausaler Modellierung gesprochen.

Der signalflussorientierte Ansatz ist vor allem in der Regelungstechnik sehr bekannt und wird dort vorwiegend eingesetzt.

Für die Modellierung von physikalischen Systemen ist Matlab/Simulink gut verwendbar, solange die Gleichungssysteme kausal aufgeschrieben werden können. Bei großen Modellen werden Matlab/Simulink-Modelle schnell unübersichtlich, da die vielen Blöcke und Linien oft schwer überschaubar sind. Durch die geeignete Wahl von Subsystemen kann jedoch eine gute Struktur erreicht werden, was allerdings einige Übung erfordert.

Modelica

Modelica [53] ist eine objektorientierte, gleichungsbasierte Modellierungssprache, in der Modelle durch Klassen beschrieben werden können. Durch die Instanziierung einer Klasse wird eine konkrete Komponente erzeugt (Objekt). Bei der Instanziierung können Parameter und Startwerte gesetzt werden.

Eine Instanz einer Klasse kann über Schnittstellen mit anderen Objekten verbunden werden, diese repräsentieren dann den Kontext der Instanz. Wird zum Beispiel eine Klasse *Widerstand* erstellt, so kann diese in einem Modell mehrfach instanziiert werden. Jede dieser Instanzen kann anders parametriert und kausalisiert sein.

Beispiel 2.3 (Akausale Modellierung): *Eine Gleichung für einen Widerstand $u(t) = R \cdot i(t)$ ist in Modelica keine Zuweisung für u, sondern stellt den Zusammenhang zwischen den Variablen u und i (R sei ein Parameter) her und kann nach jeder der Variablen aufgelöst werden, abhängig vom Kontext, in dem die Instanz eingebettet ist. Dies wird auch als gleichungsbasierte oder akausale Modellierung bezeichnet. Das Simulationswerkzeug übernimmt dabei die Kausalisierung der Gleichungen.*

Diese Kausalisierung macht Modelica so flexibel bei der Beschreibung von physikalischen Modellen. Die benötigten Modellgleichungen können der Fachliteratur entnommen und ohne manuelles Kausalisieren implementiert werden. Detaillierte Einführungen in Modelica sind in [32, 83] zu finden. Eine kurze Einführung in die Grundkonzepte von Modelica wird im Anhang B gegeben.

Modelica hat das Ziel, unterschiedliche physikalische Domänen zu unterstützen. Ein Modelica-Modell kann in einem normalen Texteditor geschrieben und in unterschiedlichen Werkzeugen simuliert werden. Die Idee dabei ist, werkzeugunabhängig zu sein und eine Austauschbarkeit zu erreichen. Ein heute gängiges Werkzeug für Modelica ist das kommerzielle Dymola von Dassault Systems [18]. Weitere kommerzielle Werkzeuge sind MapleSim [47], SimulationX [77] und Wolfram SystemModeler [90]. Freie Werkzeuge sind beispielsweie OpenModelica [62] und JModelica [1].

Das Satellitenmodell wird im Folgenden in Modelica implementiert und kann in jedem der oben genannten Werkzeuge simuliert werden.

Auch hier sollen wieder die Komponenten *Planet* und *Satellit* verwendet werden. In Modelica werden diese Komponenten jeweils in einer Klasse beschrieben. Listing 2.1 zeigt das in Modelica implementierte Satellitenmodell. In den Zeilen 2–3 stehen die Parameter, die vom Planeten stammen. In den darauf folgenden Zeilen sind die Variablen deklariert. Zeile 14 stellt die Gleichung 2.3 für die erste kosmische Geschwindigkeit dar. Diese muss während der Initialisierung gelten. Die Gleichung muss in Modelica im Gegensatz zu Simulink nicht nach dem gesuchten Wert umgestellt werden. Es kann entweder die Geschwindigkeit v_y oder die Starthöhe x des Satelliten angeben werden, die Umstellung übernimmt das Simulationswerkzeug. Es wird absichtlich nicht r initialisiert, da in diesem Fall x und y Werte ungleich Null annehmen könnten. In Modelica haben alle Variablen den initialen Startwert Null, wobei die Zustandsvariablen eine höhere Priorität haben als andere Variablen.

Sowohl x als auch y sind Zustandsvariablen, während r keine ist. Wird kein Wert für r und y gegeben, so wird durch die Priorisierung $y = 0$ verwendet, während $r = x$ (bzw. $r = -x$) gilt.

Es folgen ab Zeile 15 die Gleichungen des Modells, welche exakt die Gleichungen 2.1 und 2.2 sind.

```
1  model Satellit
2    parameter Real M;
3    parameter Real G;
4    Real F;    // Zentripetalkraft
5    Real a;    // Beschleunigung
6    Real ax;   // Beschleunigung in x-Richtung
7    Real ay;   // Beschleunigung in y-Richtung
8    Real vx;   // Geschwindigkeit in x-Richtung
9    Real vy;   // Geschwindigkeit in y-Richtung
10   Real x;    // x-Koordinate
11   Real y;    // y-Koordinate
12   Real r;    // Abstand vom Erdmittelpunkt
13 initial equation
14   vy^2 = G*M/x;
15 equation
16   F = -G*M/r^2;
17   a = F;
18   ax = a*x/r;
19   ay = a*y/r;
20   der(vx) = ax; // Ableitung von vx nach der Zeit
21   der(vy) = ay; // Ableitung von vy nach der Zeit
22   der(x) = vx; // Ableitung von x nach der Zeit
23   der(y) = vy; // Ableitung von y nach der Zeit
24   r^2 = x^2 + y^2;
25 end Satellit;
```

Listing 2.1: Modelica-Klasse des Satelliten

Die Klasse des Planeten in Modelica ist in Listing 2.2 gezeigt. Dieses Modell ist in einer *record*-Klasse implementiert, die einen Spezialfall einer normalen Modelica-Klasse darstellt und keine Gleichungen enthalten darf.

```
1  record  Planet
2    constant Real G = 6.670*10^(-11);
3    parameter Real M;
4    parameter Real r;
5  end Planet;
```

Listing 2.2: Modelica-Klasse des Planeten

```
1 model PlanetSatellit
2   Planet erde (M = 5.976e24, r = 6378e3);
3   Satellit sat1(x(start=erde.r+400000),
4                 G = erde.G,     M = erde.M);
5   Satellit sat2(vy(start=7668),
6                 G = erde.G,     M = erde.M);
7 end PlanetSatellit;
```

Listing 2.3: Modelica-Klasse des Gesamtsystems mit Planet und zwei Satelliten

In Listing 2.3 ist ein Modell dargestellt, das zwei Instanzen der Klasse *satellit* und eine Instanz der Klasse *planet* enthält. Die beiden Satelliten wurden integriert, damit die Initialisierung mit den unterschiedlichen Werten gezeigt werden kann. Die Satelliten selbst beeinflussen sich nicht. Die Instanz aus Zeile 3 wird mit einem Abstand zum Planetenmittelpunkt instanziiert und die Instanz aus Zeile 5 durch eine Startgeschwindigkeit in y-Richtung.

Beide Satelliten erhalten bei der Instanziierung die Daten des Planeten, sie müssen deshalb nicht nochmal explizit mit dem Planeten verbunden werden.

Die Modelica-Simulationsumgebungen und das Werkzeug Matlab/Simulink sind Standardwerkzeuge für die Modellierung mit differential-algebraischen Gleichungen. Vorteile von Modelica gegenüber Matlab/Simulink sind der akausale Ansatz, mit dem auf ein manuelles Umstellen der Gleichungen verzichtet werden kann, und die Möglichkeit der objektorientierten Modellierung.

2.3. Modellsimulation

Der nächste Schritt ist die Simulation, um mithilfe der Ergebnisse Informationen über das System zu erlangen. Dafür wird das Modell zunächst für das numerische Lösungsverfahren vorverarbeitet. Es wird im Folgenden nur ein Überblick gegeben. Für detaillierte Informationen sei auf die Fachliteratur verwiesen [15].

Abbildung 2.8 zeigt die Schritte, die zu einer Simulation notwendig sind.

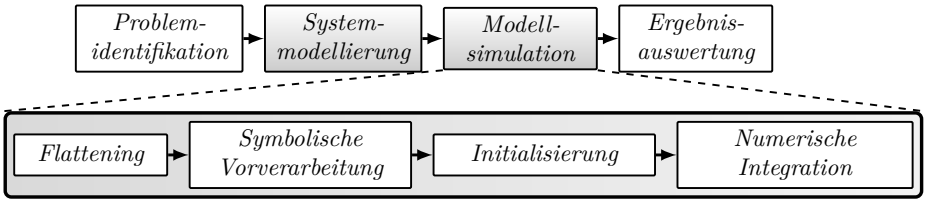


Abb. 2.8.: Grobe Schritte bei der Simulation

Flattening

In Matlab/Simulink und in allen Modelica-Werkzeugen ist es üblich, die vom Modellierer erstellte Struktur, wie Subsysteme und Objekte, aufzulösen und alle Gleichungen zu einem großen Gleichungssystem zusammenzufassen (*flattening*). Die Variablen erhalten dabei einen eindeutigen, der ursprünglichen Hierarchie entsprechenden Namen, wie *sat1.G* usw.

Symbolische Vorverarbeitung

Während der symbolischen Vorverarbeitung wird das Gleichungssystem des Modells kausalisiert.

Kausalisieren: *Das Kausalisieren dient dazu, die Gleichungen in eine vom Computer lösbare Reihenfolge zu bringen. Dabei wird jede Gleichung zur Berechnung einer Variablen verwendet und jede Variable von genau einer Gleichung bestimmt.*

In Simulink sind die Gleichungen bereits durch den signalflussorientierten Ansatz kausalisiert, der Modellierer muss dies bei der Implementierung manuell vornehmen.

Werden Modelica-Modelle betrachtet, so ist nach dem Flattening keine Berechnungsvorschrift gegeben. Die Gleichungen sind ungeordnet und es ist nicht bekannt, welche Variable aus welcher Gleichung bestimmt wird. Das Gleichungssystem muss kausalisiert werden. Ein Algorithmus zum Sortieren ist der Tarjan-Algorithmus, welcher die Gleichungen horizontal und vertikal sortiert.

Tarjan: *Der Tarjan-Algorithmus ordnet die Gleichungen in einer Reihenfolge an, in der sie gelöst werden können (vertikal). Dabei wird jede der Gleichungen nach der Variablen umgestellt, die sie berechnen soll (horizontal).*

Beim Kausalisieren können Schwierigkeiten auftreten, wie zum Beispiel algebraische Schleifen; solche Modelle mit algebraischen Schleifen werden Modelle mit Index-1 genannt. Hierbei wird während des Kausalisierens ein Punkt erreicht, an dem jede Gleichung mindestens zwei unbekannte Variablen hat und jede dieser Variablen mindestens in zwei Gleichungen vorkommt. Als sehr einfaches Beispiel dient folgendes Gleichungssystem:

$$x = y + 2$$
$$y = 5 \cdot x$$

Sind x und y unbekannt, so kann der Tarjan-Algorithmus keine der Gleichungen nach einer Unbekannten auflösen, da die andere enthaltene Variable ebenso unbekannt ist.

Damit ist das Gleichungssystem nicht mit dem Tarjan-Algorithmus weiter kausalisierbar. Das Tearing-Verfahren ist eine Möglichkeit, mit dieser Problematik umzugehen.

Tearing-Verfahren: *Es wird eine der Gleichungen ausgesucht (Residual-Gleichung) und diese nach einer der Unbekannten umgestellt (Tearing-Variable). Die Tearing-Variable wird daraufhin als bekannt angenommen. Somit ist für die zweite Gleichung nur noch eine Unbekannte in der Gleichung und das Gleichungssystem kann weiter gelöst werden. Am Ende des Verfahrens wird die Tearing-Variable durch bekannte Größen bestimmt oder durch das Newton-Verfahren berechnet.*

Hat ein Modell Zwangsbedingungen durch algebraische Gleichungen, so hat es einen höheren Index als Eins. In einem solchen Fall wird auch von einem Gleichungssystem mit struktureller Singularität gesprochen. Das Gleichungssystem kann dann ebenfalls nicht mit dem Tarjan-Algorithmus kausalisiert werden. Ein gängiges Verfahren zur Indexreduktion ist das Pantelides-Verfahren.

Pantelides-Verfahren: *Es wird die Gleichung der Zwangsbedingung differenziert und dem Gleichungssystem hinzugefügt. Zusätzlich wird eine weitere Variable hinzugefügt, indem die Ableitung einer Variablen durch eine neue, unbekannte Variable ersetzt wird. Mit jeder Anwendung dieses Verfahrens wird der Index um Eins reduziert. Bei einigen Modellen kann von einem Index-2-System direkt auf ein Index-0-System übergegangen werden. Meist bleibt ein Index-1-System übrig, das dann noch mit dem Tearing-Verfahren gelöst werden muss.*

Details zu diesen Verfahren können entsprechender Fachliteratur entnommen werden [15].

Initialisierung

Der zeitliche Verlauf der Variablen wird üblicherweise mit Hilfe von numerischen Verfahren bestimmt, welche im nächsten Abschnitt diskutiert werden. Bevor solch eine numerische Lösung stattfinden kann, müssen die Variablen initialisiert werden. Das bedeutet, jeder Variablen und auch jedem Parameter muss ein eindeutiger Wert zugewiesen werden. Wie aus Abschnitt 2.2 bereits bekannt, kann der Modellierer Startwerte für die Simulation angeben. Die gesetzten Werte sollten zu einem konsistenten, also widerspruchsfreien, initialen Zustand für das Modell führen. Dies ist, besonders bei komplexen Gleichungssystemen, kompliziert und oft sind nur Schätzwerte bekannt.

Konsistenz: Die vom Modellierer gewünschten Startwerte müssen den Gleichungen des DAE-Systems sowie den benutzerdefinierten algebraischen Bedingungen genügen (beispielsweise aus dem *initial equation*-Teil von Modelica). Es wird ein initiales Gleichungssystem erstellt, welches zur Berechnung der Startwerte verwendet wird. Viele Werkzeuge, besonders im Modelica-Bereich, erlauben es, Schätzwerte anzugeben. Diese werden soweit wie möglich verwendet, können jedoch vom Simulationswerkzeug überschrieben werden, falls sie zu Inkonsistenzen führen.

Das initiale Gleichungssystem ist meist ein nichtlineares Gleichungssystem. Bei einer Variante zum Lösen dieses Gleichungssystems wird ein impliziter Euler-Schritt vorgenommen und mit einem gedämpften Newton-Verfahren wird das Gleichungssystem gelöst [8].

Des Weiteren gibt es noch rigorose Methoden und direkte Methoden [89], welche das Finden konsistenter Startwerte ermöglichen.

Alle genannten Methoden führen nicht zwangsläufig zu einem konsistenten, gewünschten Startzustand, wie beispielsweise in [11, 76] diskutiert wird. Zur robusteren und sichereren Initialisierung wird dort die Homotopie vorgeschlagen. Bei der Homotopie wird ein vereinfachtes Modell verwendet, welches durch Deformation in das eigentliche Modell überführt wird, um Startwerte für die Simulation zu ermitteln.

Stabilität bei der Initialisierung: Startwerte sollten nicht nur zu einem konsistenten Initialzustand, sondern auch zu einer stabilen Lösung führen.

Stabilität: *Ein Modell ist stabil, wenn sich die Ausgangsgrößen des Modells bei beschränkten Eingangssignalen ebenfalls in beschränkten Bereichen befinden.*

Um die Stabilität eines Gleichungssystems zu bestimmen, kann die Zustandsraumdarstellung verwendet werden. Für zeitvariante, lineare, autonome Systeme sieht diese wie folgt aus: $\dot{z}(t) = A \cdot z(t)$. *Dabei ist A die Systemmatrix.*

Haben alle Eigenwerte λ *der Systemmatrix A einen Realteil, der kleiner als Null ist, so ist das Modell stabil. Dies wird als „bounded input bounded output" (bibo) Stabilität bezeichnet [15].*

Prinzipiell ist zwar das Verhalten des Modells durch die differential-algebraischen Gleichungen deterministisch beschrieben, die Startwerte beeinflussen jedoch den Werteverlauf signifikant. Einige Modelle sind besonders startwertsensitiv, wie beispielsweise chaotische Modelle. Bei chaotischen Modellen reichen bereits leichte Abweichungen der Startwerte, um ein anderes Gesamtverhalten des Modells zu erhalten [71]. Dabei kann es sich hier bereits um scheinbar einfache Modelle handeln, wie zum Beispiel ein Doppelpendelmodell. Je nach Startposition und Geschwindigkeit ergeben sich unterschiedliche Verhaltensweisen.[5]

Ungünstig gewählte Startwerte können zu ungenauen Ergebnissen oder auch zu komplett falschem Verhalten des Modells führen. Das Setzen guter Startwerte ist somit von immenser Bedeutung für die Simulation.

Numerische Integration

Das Gleichungssystem liegt nach der symbolischen Vorverarbeitung in einem lösbaren Zustand vor. Durch die Initialisierung sind konsistente Startwerte für das Gleichungssystem bekannt. Das Gleichungssystem kann nun mit einem numerischen Approximationsverfahren gelöst werden. Es werden hier nur die Grundlagen zum Lösen von Differentialgleichungen mit numerischen Verfahren betrachtet. Für weitere Informationen sei wieder auf die Fachliteratur verwiesen [15]. Es wird ein Gleichungssystem betrachtet, das wie folgt aussieht:

$$\dot{\underline{z}}(t) = f(t, \underline{z}(t)) \tag{2.6}$$

mit den Anfangsbedingungen:

$$\underline{z}(t_0) = \underline{Z}_0. \tag{2.7}$$

Solch ein Gleichungssystem kann häufig nicht analytisch gelöst werden, da die Stammfunktion nicht ermittelt werden kann.

[5] http://home.rotfl.org/extras/chaos/

Die Idee der numerischen Verfahren besteht darin, diese Stammfunktion mit diskreten, kleinen Zeitschritten zu approximieren. Als einfachstes, numerisches Verfahren wird das Euler-Verfahren betrachtet. Die Ableitung aus obiger Gleichung beschreibt zu jedem Zeitpunkt die Steigung der gesuchten Funktion. Dadurch entsteht ein Richtungsfeld für die Steigung der gesuchten Kurve, siehe Abbildung 2.9.

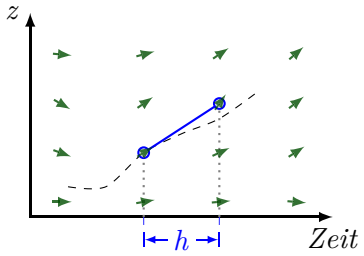


Abb. 2.9.: Richtungsfeld für die Steigung $\dot{z}(t)$, eine Beispielfunktion $z(t)$ und ein Euler-vorwärts-Schritt der Länge h

Euler-vorwärts-Verfahren: *Für das Euler-vorwärts-Verfahren wird die Steigung $\dot{z}(t)$ zu einer Zeit t und einem Wert $z(t)$ aus der Differentialgleichung 2.6 bestimmt. Diese Steigung wird mit der Schrittweite h multipliziert und auf den aktuellen Wert $z(t)$ addiert. Dies ergibt den neuen Wert $z(t+h)$.*

Gleichung 2.8 zeigt die Formel zur Berechnung des nächsten Werts mit dem Euler-vorwärts-Verfahren, wobei davon ausgegangen wird, dass $z(t)$ ein korrekter Wert auf der gesuchten Kurve ist.

$$z(t+h) \approx z(t) + h \cdot \underbrace{f(t, z(t))}_{\dot{z}(t)} = u(t+h) \tag{2.8}$$

Abbildung 2.10 zeigt eine Funktion $z(t)$ und ihre durch das Euler-vorwärts-Verfahren approximierte Kurve $u(t)$. Hierbei gilt Gleichung 2.8 nur im ersten Schritt, da nur in diesem Fall $z(t)$ auf der realen Kurve liegt. In allen weiteren Schritten ist $z(t)$ mit dem approximierten Wert $u(t)$ zu ersetzen: $u(t+h) = u(t) + h \cdot f(t, u(t))$. Da die reale Kurve nur angenähert wird, wird auch oft von Näherungsverfahren gesprochen.

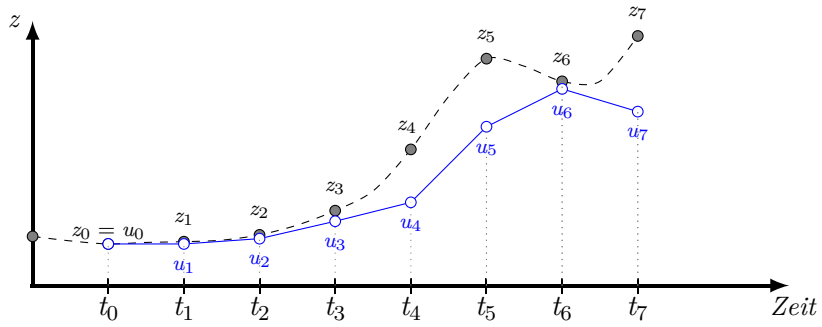


Abb. 2.10.: Graphische Darstellung des Euler-vorwärts-Verfahrens. Dabei repräsentiert z(t) die Kurve der Stammfunktion und u(t) die mit dem Euler-Verfahren approximierte Kurve.

Das Euler-Verfahren ist ein einfaches Verfahren und ist wenig berechnungsintensiv. Das Verfahren hat jedoch nur einen kleinen Stabilitätsbereich. Um die Stabilität eines Modells mit dem Euler-vorwärts-Verfahren zu überprüfen, muss Gleichung 2.9 betrachtet werden.

$$x(t+h) \approx (I^n + \underbrace{A \cdot h}_{A_{new}}) \cdot x(t) \tag{2.9}$$

In diesem Fall sind die Eigenwerte der Matrix A_{new} relevant, welche wiederum kleiner als Null sein müssen.

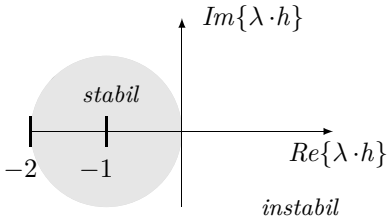


Abb. 2.11.: Stabilitätsbereich des Euler-vorwärts-Verfahrens in Abhängigkeit von den Eigenwerten (λ) der Systemmatrix (A) und der Schrittweite (h)

Der Stabilitätsbereich für das Euler-vorwärts-Verfahren ist in Abbildung 2.11 gezeigt. Dieser ist von der gewählten Schrittweite abhängig. Je kleiner die Schrittweite, desto stabiler ist das Euler-vorwärts-Verfahren. Dies bringt jedoch auch eine erhöhte Rechenzeit mit sich.

Da bei diesem Verfahren die neuen Werte $z(t_{n+1})$ nur von bereits bekannten Werten ($z(t_n)$) abhängen, wird dieses Verfahren explizites Verfahren genannt.

Im Gegensatz dazu gibt es das Euler-rückwärts-Verfahren, welches ein implizites Verfahren ist. Bei diesem werden die neuen Werte $z(t_{n+1})$ mit Hilfe der Steigung an genau diesem Punkt bestimmt, siehe Gleichung 2.10. Zum Lösen dieser Gleichungen muss bei jedem Schritt ein lineares Gleichungssystem gelöst werden, wodurch der Rechenaufwand kubisch mit der Anzahl der Variablen ansteigt. Das Euler-rückwärts-Verfahren hat aber einen wesentlich größeren Stabilitätsbereich als das Euler-vorwärts-Verfahren, siehe Abbildung 2.12.

$$z_{n+1}(t_{n+1}) = z_i(t_n) + h \cdot f(t_{n+1}, z_{n+1}) \tag{2.10}$$

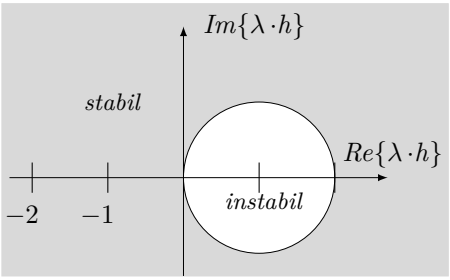


Abb. 2.12.: Stabilitätsbereich des Euler-rückwärts-Verfahrens in Abhängigkeit von den Eigenwerten (λ) der Systemmatrix (A) und der Schrittweite (h)

Es gibt noch diverse weitere Verfahren, wie das Heun-Verfahren, das Trapez-Verfahren und die Familie der Runge-Kutta-Verfahren. Diese zählen alle zu den Einschrittverfahren, da sie nur einen alten Wert zur Bestimmung des neuen Wertes nutzen. Die Gruppe der Mehrschrittverfahren verwendet zusätzlich noch weiter zurückliegende Werte, um den neuen Wert zu bestimmen.

Die oben diskutierten Verfahren sind alle Verfahren, die über die Zeit diskretisieren. Das *Quantized State Systems* Verfahren (QSS Verfahren) stellt eine weitere Möglichkeit dar, das zeitliche Verhalten zu bestimmen. Dabei wird über die Änderung jeder Variable diskretisiert und jede Variable hat ihren eigenen, individuellen Zeitvektor und Integrator.

Für mehr Informationen über numerische Lösungsverfahren sei auf die Fachliteratur verwiesen [15].

Die Simulationswerkzeuge arbeiten prinzipiell wie vorgestellt. Das bedeutet, dass für jede Simulation ein Gleichungssystem verwendet wird, welches durch den obigen Prozess vorverarbeitet und dann simuliert wird. Es ist in den aktuell verfügbaren Werkzeugen nicht vorgesehen, die symbolische Vorverarbeitung während der Simulation eines Modells zu wiederholen, falls sich Gleichungen ändern. Eine erneute symbolische Vorverarbeitung ist jedoch notwendig, sollte sich die Anzahl der Variablen, die Kausalität oder der Index des Modells ändern.

2.4. Von konventionellen zu strukturvariablen Modellen

Wie am Anfang des Kapitels erläutert, wird durch ein Modell ein reales System vereinfacht abgebildet. Dabei soll es möglichst alles beschreiben, was für den Modellierer von Interesse ist.

Das heutige Vorgehen der Werkzeuge setzt während der Simulation genau ein Gleichungssystem voraus. Damit ist der Modellierer auf genau ein Modell beschränkt, welches alle notwendigen Aspekte abdecken muss.

Ein Modell zu finden, welches alle Phasen eines Systems mit einer gewünschten Genauigkeit abbildet, ist häufig unmöglich oder führt zu nicht simulierbaren Modellen.

Es ist somit notwendig, Modelle zu betrachten, die verschiedene Gleichungssysteme und Variablen während der Simulation verwenden, um so während der gesamten Simulation dem Grundsatz der Einfachheit zu entsprechen. Die Idee besteht darin, mehrere Modelle sequentiell hintereinander zu verwenden. Jedes dieser Modelle repräsentiert das Systemverhalten für einen gewissen Zeitraum mit dem notwendigen Abstraktionsgrad. Solche Modelle werden als strukturvariable Modelle bezeichnet.

Strukturvariable Modelle stellen neue Anforderungen an die Modellierung und Simulation, welche in der vorliegenden Arbeit untersucht werden. Konzepte zum Umgang mit diesen Modellen werden detailliert diskutiert, sodass strukturvariable Modelle in der Zukunft vorteilhaft eingesetzt werden können.

3 Strukturvariable Modelle

Strukturvariable Modelle werden in der Literatur unterschiedlich definiert. Die häufigsten Definitionen werden einander hier gegenübergestellt, um dann eine für diese Arbeit gültige Definition anzugeben.

Es wird dargelegt, weshalb strukturvariable Modelle relevant für die Modellierung und Simulation von technischen Systemen sind, und ebenso, warum sie noch nicht Stand der Technik sind.

Darauf folgend werden bereits existierende Ansätze zur Modellierung, Notation und Simulation von strukturvariablen Modellen vorgestellt sowie deren Defizite zusammengefasst und erläutert.

3.1. Verschiedene Bezeichnungen für strukturvariable Modelle

Die in der Literatur verwendeten Begriffe für strukturvariable Modelle sind sehr unterschiedlich, einige häufig zu findende Begriffe werden in der Tabelle A.1 aus dem Anhang A.1 in der Reihenfolge ihres Erscheinens aufgeführt. Bereits 1979 beschäftigte sich F. Cellier in seiner Dissertation mit *variable-structure simulation* [12]. Diese beschreibt er als kombinierte Simulation, in der sich Differential- und/oder Differenzen-Gleichungen mit der Zeit ändern.[1]

Um 1987 beschäftigte sich T. Ören mit Modellen, die mehrere Modi aufweisen [63]. Diese nannte er *Multimodels* und beschrieb sie als Modelle, die aus mehreren Submodellen bestehen, wobei jeweils ein Modell aktiv ist und durch Transitionen ein anderes Modell aktiviert werden kann. Der Begriff der *Multimodels* ist in der heutigen Literatur kaum noch zu finden.

Ein sehr verbreiteter Begriff ist der der *hybriden Systeme* bzw. der *hybriden Modelle*. Dieser Begriff ist mit vielen Bedeutungen belegt. So wird ein System/Modell auch als hybrid bezeichnet, wenn es ein Hybrid aus

[1] frei übersetzt aus der genannten Dissertation

verschiedenen physikalischen Bereichen ist. Im Automobilbereich wird beispielsweise von Hybriden gesprochen, wenn der Antrieb auf einem Elektromotor und einem weiteren Energiewandler basiert. Solche Modelle werden oft auch als *Multiphysics models* bezeichnet. Des Weiteren werden Modelle, die kontinuierliche und diskrete Anteile haben, als hybrid bezeichnet [56]. Viele der heute gängigen Simulationswerkzeuge unterstützen kontinuierliche Berechnungen und diskrete Ereignisse in einem Modell.

Da Strukturänderung oft bei kontinuierlichen Modellen beim Eintreten eines bestimmten Ereignisses auftreten und diese Änderungen diskret behandelt werden, werden Modelle mit Strukturänderungen manchmal auch als hybride Modelle bezeichnet oder als Unterart dieser [19, 60]. Der verwendete Begriff sagt noch nicht aus, was genau gemeint ist. Erst aus dem Kontext der Literatur ist zu erkennen, um welche Art es sich handelt.

In der späteren Literatur sind eher Namensabwandlungen von *variable-structure* zu finden. Auch dieser Name hat weitere Bedeutungen. So wird dieser Begriff in der Regelungstechnik oft auch als *sliding mode control* oder *variable structure control* bezeichnet [20, 5]. Dabei geht es darum, einen Regler möglichst robust auszulegen, damit er wenig durch Modellierungsungenauigkeiten beeinflusst wird. Bei einer solchen Regelung ändert sich die Regelstrategie abhängig vom aktuellen Systemzustand, beispielsweise bei einer Schlupfüberwachung von Reifen.

Abwandlungen von *variable-structure* sind beispielsweise die Begriffe *structural dynamism* [33], *hybrid dynamic models of variable structure* [88] und *structural dynamical systems* [7]. Dabei sind mit diesen Begriffen allgemein tatsächlich Modelle gemeint, bei denen sich das Gleichungssystem beim Eintreten eines Ereignisses ändert. Die genauen Definitionen unterscheiden sich manchmal leicht. Allen gemein ist, dass ein diskretes Ereignis eintritt, welches einen Strukturwechsel des kontinuierlichen Modells einleitet. Bei manchen geht es um Strukturänderungen, bei denen eine Gleichung ausgetauscht wird, bei anderen werden ganze Gleichungssysteme ausgetauscht.

In deutschsprachigen Arbeiten wird oft von Strukturdynamik-Modellen gesprochen. Beispielsweise sind damit in [60] Modelle gemeint, die ihre Gleichungen während der Simulation ändern. In der Mechanik hingegen wird mit Strukturdynamik die mathematische Beschreibung des räumlichen und zeitlichen Verhaltens von mechanischen Systemen verstanden [30]. Dabei werden in der Mechanik die Schwingungen von Gegenständen und auch deren Materialverformung betrachtet. In englischen Arbeiten wird in diesem Fall von *dynamic structure* gesprochen.

Da die Begriffsvielfalt zu Verwirrungen führen kann, wird in dieser Arbeit von *strukturvariablen Modellen* bzw. *variable-structure models* gesprochen. Der Begriff *System* wird hier absichtlich nicht verwendet, da verdeutlicht werden soll, dass es um eine mathematische Abstraktion eines realen Systems geht.

3.2. Definitionen für diese Arbeit

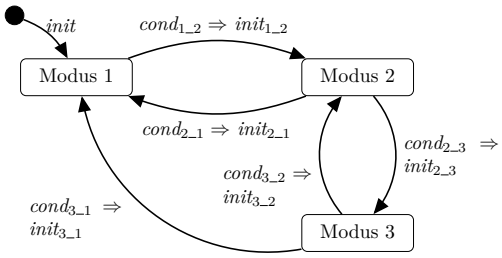


Abb. 3.1.: Graphische Darstellung eines Modells mit drei Modi

Um im Weiteren mit einheitlichen Begriffen zu arbeiten, wird Folgendes definiert:

Definition 3.1 (Strukturvariables Modell): *Ein Modell, das aus mehreren Sets von Gleichungen und Variablen besteht, zwischen denen während der Simulation gewechselt werden kann, wird strukturvariables Modell genannt. Dabei ist immer ein Set von Gleichungen und Variablen aktiv.*

Definition 3.2 (Modus bzw. Modi): *Modelle, die während eines Zeitintervalls in einem strukturvariablen Modell aktiv sind, werden als Modi bezeichnet. Modi können auf der obersten Modellebene oder innerhalb von Komponenten definiert werden. Ein Modus repräsentiert das Modell oder die Komponente, während er aktiv ist.*

Inwieweit sich die Modi unterscheiden, ist in der vorliegenden Definition nicht gesagt. Es ist möglich, nur eine Gleichung oder das komplette Gleichungssystem auszutauschen. Ändert sich das Modell signifikant, wenn sich beispielsweise die Anzahl der Variablen, der Index oder die Kausalität ändern, ist ein erneutes Kausalisieren notwendig.

Um von einem Modus zum nächsten zu wechseln, sind Transitionen notwendig.

Definition 3.3 (Transition): *Eine Transition beschreibt den Übergang von einem Modus zu einem anderen. In einer Transition muss angegeben werden, zu welchem Modus unter welcher Bedingung gewechselt wird. Zusätzlich muss angegeben werden, wie der neu zu aktivierende Modus initialisiert wird.*

Ein allgemeines Beispiel eines strukturvariablen Modells ist in Abbildung 3.1 in einer Statechart-ähnlichen Notation gezeigt.

Beispiel 3.1 (Satellitenstart – Fortsetzung von Beispiel 2.2): *Als Beispiel wird nochmals der Satellit aus dem Grundlagenkapitel betrachtet. Es wird nun aber nicht nur der Satellit im Orbit, sondern auch der Start der Rakete, die den Satellit in den Orbit bringt, betrachtet. Abbildung 3.2 zeigt eine abstrakte Sicht auf das Modell.*

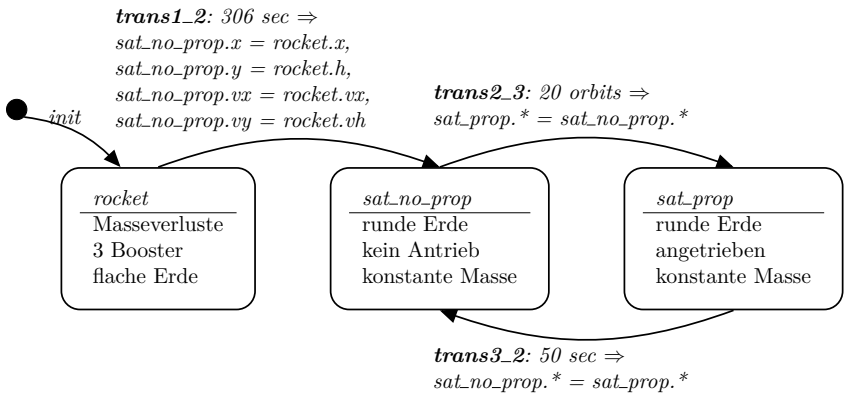


Abb. 3.2.: Abstrakte Sicht auf die Modi und Annahmen an das erweiterte Satellitenmodell, wobei der Satellit mit einer Rakete in den Orbit gebracht wird

Die betrachtete Rakete (rocket) verliert durch die Verbrennung von Kraftstoff Masse und besitzt drei Booster. Die Booster werden abgeworfen, sobald sie leer sind. Zusätzlich wird die Erde als flach angenommen. Die hier getroffenen Annahmen sind dem Lehrbuch [52] entnommen, in welchem diese als valide für einfache Raketenstarts und Satellitenbewegungen beschrieben werden. Wenn der Treibstoff der Rakete aufgebraucht ist (Vorgabe: hier nach 306 Sekunden), wird zum Satellitenmodell ohne Antrieb gewechselt (sat_no_prop).

Dieses entspricht dem schon bekannten Modell (Beispiel 2.2), jedoch wird diesmal keine Kreisbahn für den Flug des Satelliten gefordert. Die Erde wird als rund angenommen und es findet keine Beschleunigung statt.

Die Position und Geschwindigkeit des Satelliten (sat_no_prop.x, sat_no_prop.y, etc.) werden mit dem Endzustand der Rakete (rocket.x, rocket.y) initialisiert. Nachdem der Satellit 20 Mal um die Erde geflogen ist, soll er 50 Sekunden lang angetrieben werden, um ihn in eine andere Umlaufbahn zu bringen (sat_prop). Dies wird mit dem dritten Modus abgebildet, in dem zur Vereinfachung kein Masseverlust berücksichtigt wird. Zur Initialisierung werden alle Daten des einen Modells dem anderen übergeben, sofern die Namen identisch sind (sat.prop.∗ = sat_no_prop.∗). Danach wird der Antrieb wieder abgeschaltet. Abbildung 3.3 zeigt schematisch die Flugbahn des Satelliten ab dem Start auf der Erde.

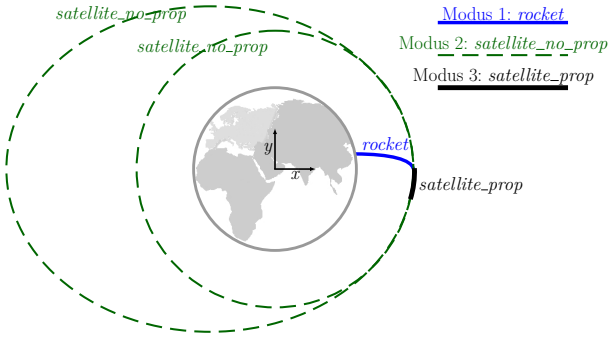


Abb. 3.3.: Flugbahn der Rakete und des Satelliten

Das Beispiel der Rakete wäre auch ohne Strukturänderungen umsetzbar, dann müsste das Modell jedoch zu jeder Zeit alle Eigenschaften enthalten und ausreichend detailliert sein. Die Strukturänderung ist somit oft nur notwendig, weil eine Abstraktion durchgeführt wurde, die das Verhalten des realen Systems nicht ausreichend für alle Eventualitäten beschreibt. Dies wiederum bedeutet: Wenn nur ausreichend komplex modelliert wird, kann auf Strukturänderungen weitestgehend verzichtet werden.

Die Frage ist nun: Warum ist die Modellierung mit Strukturänderungen trotzdem sinnvoll?

3.3. Relevanz von strukturvariablen Modellen

Modelle sollen jeweils alles Notwendige berücksichtigen, allerdings sind detaillierte Modelle meist aufwendig zu modellieren und zu simulieren. Daher sollten Modelle zu jeder Zeit so einfach wie möglich sein und dennoch alles Notwendige abdecken.

Viele technische Systeme durchlaufen verschiedene Phasen, die in konventionellen Modellen in einem Gleichungssystem abgebildet werden müssen. Es gibt grob drei Vorgehensweisen für den Entwurf solcher Modelle:

1. Verwendung eines Modells, welches während der gesamten Simulation alles Notwendige betrachtet: Würden die unterschiedlichen Phasen alle in ein großes Modell integriert werden, würde das Modell unter Umständen sehr kompliziert, hätte einen hohen Index und/oder könnte nicht simuliert werden. Wird mehr in einer Phase betrachtet als notwendig, kann dies sogar negative Auswirkungen auf die Simulationsergebnisse haben.

2. Verwendung eines einfachen Modells für die gesamte Simulation, um möglichst schnell zu simulieren: Dies führt zu kurzen Simulationszeiten, abstrahiert aber über verschiedene Phasen. Im Extremfall werden die Phasen nicht unterschieden und so die Simulationsergebnisse verfälscht.

3. Verwendung eines Modells, das für alle Phasen so einfach wie möglich und so detailliert wie notwendig für die gesamte Simulation ist: Dies ist unter den gegebenen Umständen eine sinnvolle Entscheidung. Jedoch werden dadurch einzelne Phasen zu manchen Zeitpunkten eventuell genauer oder ungenauer als gerade notwendig betrachtet.

Diese drei Varianten zeigen, dass es möglich ist, ohne Strukturänderungen in der Modellierung auszukommen. Jedoch führt dieses Vorgehen zu Modellen, die nicht dem Grundsatz der Einfachheit zu jeder Zeit entsprechen. Für eine gute Simulation sollte es möglich sein, das Gleichungssystem während der Simulation zu ändern und damit jede Phase genau in dem notwendigen Detaillierungsgrad zu betrachten.

Strukturvariable Modelle ermöglichen genau solch ein Vorgehen. Ein Modell wird in Teilmodelle (Modi) zerlegt, welche für verschiedene Phasen die gültigen mathematischen Gleichungen enthalten.

Zwischen diesen Phasen wird situationsbedingt gewechselt, sodass beispielsweise unterschiedliche Prozessphasen oder Detaillierungsgrade abgebildet werden. Ebenso trägt eine Aufteilung der Modelle zu einer besseren Strukturierung bei.

Jeder Modus wird so einfach wie möglich gehalten und betrachtet alles Notwendige für eine Phase. Da nun nicht mehr alles in ein Modell integriert sein muss, sondern jede Phase in ihrem entsprechenden Detailgrad betrachtet wird, kann die Simulationszeit verkürzt oder auch die Genauigkeit der Simulation gesteigert werden.

3.4. Herausforderungen gegenüber der konventionellen Modellierung

Wenn strukturvariable Modelle sinnvoll sind, stellt sich die Frage, warum sie heute noch nicht in allen Werkzeugen unterstützt werden.

Ein Grund ist die aufwendigere Modellierung, da mehrere Modelle erstellt werden müssen. Dazu muss entschieden werden, welche Modi für welche Phasen vorgesehen werden und wann zwischen ihnen gewechselt wird. Diese Entscheidungen haben wiederum Auswirkungen auf die Initialisierung des nächsten Modus. Wird ein ungünstiger Zeitpunkt gewählt oder stellt der vorherige Modus nicht ausreichend Informationen bereit, kann der nächste Modus nicht korrekt initialisiert werden. Selbst wenn alle Informationen vorhanden sind, muss die Initialisierung des nächsten Modus beschrieben werden. Die Initialisierung stellt wiederum eine Abstraktion dar, in der aus vergangenen Daten Initialwerte für einen neuen Modus bestimmt werden. Diese kann fehlerbehaftet, ungenau oder schlecht gewählt sein.

Durch die aufwendigere Modellierung sollte die Wiederverwendung von bereits modellierten Komponenten mit Strukturänderungen gefördert werden. Solche Komponenten mit Modi können in verschiedenen Kontexten eingesetzt werden, wodurch solche Modelle flexibel anpassbar sind. Dabei muss hier auf die Kompatibilität von Komponenten, Modi und Transitionen geachtet werden. Ebenso müssen gleichzeitige Moduswechsel und konsistente Initialisierungen berücksichtigt werden.

Auch die Simulation des resultierenden strukturvariablen Modells stellt neue Anforderungen an die Simulationswerkzeuge. So müssen die unterschiedlichen Modi des Modells entweder vor der Simulation oder bei einem Moduswechsel kausalisiert werden.

Auch muss während eines Moduswechsels die Initialisierung des neuen Modus durchgeführt werden. Diese Vorgehensweisen sind in heutigen Werkzeugen noch nicht ausreichend vorgesehen. In Abschnitt 3.5 wird näher erläutert, welche Ansätze existieren.

Bei strukturvariablen Modellen müssen mehrere Modi hintereinander simuliert werden, die in einer definierten Relation zueinander stehen. Die Wechselbedingungen von Transitionen müssen erkannt und die entsprechende Transition aktiviert werden. Die modellierte Initialisierung muss die Initialwerte des nächsten Modus setzen und eine erneute Initialwertberechnung auslösen.

Diese neuen Anforderungen setzen neue Konzepte zur Modellierung und Simulation strukturvariabler Modelle voraus, welche in der vorliegenden Arbeit vorgestellt werden.

3.5. Bekannte Ansätze für strukturvariable Modelle

In diesem Abschnitt werden existierende und für diese Arbeit relevante Ansätze zur Modellierung, Formalisierung/Notation und Simulation strukturvariabler Modelle diskutiert. Die drei Bereiche sind nicht unabhängig voneinander, werden jedoch zum Zweck der Verständlichkeit separat aufgeführt.

3.5.1. Modellierung

In [38] wurden verschiedene Benchmark-Beispiele vorgestellt. Diese sind meist mit wenigen Gleichungen zu implementieren und werden verwendet, um Implementierungen für die Simulation strukturvariabler Modelle zu testen.

Ein weiteres Beispiel für strukturvariable Modelle ist in [92] zu finden, wo das Modellieren von sozialem Verhalten in Abhängigkeit von äußeren Umständen betrachtet wird. Dabei wird das Modell den Umgebungsinformationen und den Umständen des sozialen Umfelds angepasst.

In [28, 29] werden die dort *Multimodels* genannten Modelle als Modellierungsansatz für künstliche Intelligenz angesehen. Dabei soll für eine gegebene Aufgabenstellung ein Modell mit dem passenden Detaillierungsgrad zusammengesetzt werden.

Die Arbeit [96] betrachtet für strukturvariable Modelle das Auswechseln von Komponenten durch andere Komponenten und erstellt dafür eine Simulationsumgebung.

In den Arbeiten von [61] wird auf Detaillierungsgradänderungen im Gebäude-Kontext eingegangen und erläutert, welche Vorteile durch strukturvariable Modelle zu erzielen sind.

Andere Veröffentlichungen betrachten einzelne Aspekte der Modellierung. So werden in [60] verschiedene Sichten auf strukturvariable Modelle präsentiert. Dabei wird die Initialisierung der Modi anhand von Energie- und Massenbilanzen angesprochen.

Auch in [56] werden Aspekte für eine gute Neuinitialisierung eines Modells betrachtet. Dabei geht es jedoch um das erneute Initialisieren eines gleichbleibenden Zustandsvektors.

Fazit: Die oben genannten Arbeiten stellen eine Reihe von Anwendungsbeispielen für strukturvariable Modelle vor, gehen aber kaum auf allgemeine Konzepte bei der Modellierung ein. Es wird nicht im Detail erläutert, wie die Modi des Modells und auch die Transitionen zwischen ihnen gewählt sein sollten. Ebenso wird der modulare und hierarchische Aufbau strukturvariabler Modelle nicht im Hinblick auf Wiederverwendbarkeit und Konsistenz der Modi hin berücksichtigt.

Eine Identifikation der Anforderungen zum Umgang mit strukturvariablen Modellen findet nur unzureichend statt. Es wird in den Arbeiten auf die Schwierigkeiten bei der Initialisierung hingewiesen, aber es findet keine ganzheitliche Betrachtung der Modellierung statt, in der die Modi, Transitionen und deren Eigenschaften inbegriffen sind. Jeder Moduswechsel ist eine Abstraktion, welche im gewissen Maße fehlerbehaftet ist, was zurzeit noch unzureichend betrachtet wurde.

3.5.2. Formalisierung und Notation

Strukturvariable Modelle können mit Phasen-Transition-Systemen [45, 46], Petri-Netzen [58] oder auch mit hybriden Automaten (Statechart-ähnlich [36]) dargestellt [25, 39] werden. In den genannten Arbeiten werden zu den Notationen ebenso Formalisierungen angegeben, welche die Notationen formal beschreiben.

Da es in dieser Arbeit vornehmlich um die Modellierung geht und dabei die softwaretechnische Seite betrachtet wird, werden die Statechart-ähnlichen Schreibweisen bevorzugt. Auch in [25] wird der Einsatz von hybriden Statecharts als sinnvolle Notation für strukturvariable Modelle betrachtet. Diese Schreibweise ermöglicht es, die Modi als Zustände und die Moduswechsel als Zustandsübergänge zu definieren.

Parallele Zustände sind in der Modellierung miteinander verbundene Komponenten, die gemeinsam simuliert werden. Komponenten können wiederum aus weiteren Komponenten bestehen. Diese Komposition ergibt den Gesamtzustand des Modells. Dieses Prinzip kann durch die hierarchische Komposition der Statecharts abgebildet werden.

In [23, 60] geht es um die softwaretechnische Sicht auf strukturvariable Modelle. Dabei wird als Notation und Formalisierung ebenfalls eine Statechart-ähnliche Sichtweise verwendet, wobei die Modellierung durch eine Spezifikation in der formalen Sprache *SimOO* unterstützt wird.

Zur Formalisierung sei als Referenz [68] genannt, welche als Grundlage für die vorliegende Arbeit dient. Dort wird eine formale Spezifikation für die kompositionelle Semantik für strukturvariable Modelle angegeben, wobei die graphische Darstellung ebenfalls denen der Statecharts ähnelt.

Fazit: In dieser Arbeit wird eine vorhandene Notation verwendet und um die Eigenschaften erweitert, die relevant für diese Arbeit sind. Es wird eine Statechart-ähnliche Notation verwendet, wie sie auch in [60, 68] benutzt wurde, da diese einfach verständlich ist und sich die notwendigen Eigenschaften strukturvariabler Modelle darstellen lassen.

Für die Formalisierung wird die Z-Notation [78] verwendet, wobei die Formalisierung die Idee der kompositionellen Semantik aus [68] erweitert.

3.5.3. Simulation

Im Folgenden werden einige der relevanten Ansätze vorgestellt.

Maximal State Space

Sollen weder neue Sprachen noch neue Simulationsalgorithmen verwendet werden, kann die Simulation über den *Maximal State Space* geschehen [6]. Bei diesem Ansatz werden alle Modi in ein Modell integriert und durch geschickte If-Statements werden die Gleichungen ausgetauscht [22, 26].

In [88] wird ein Algorithmus vorgestellt, der ein definiertes strukturvariables Modell erhält und ein *Maximal State Space* Modell zurückgibt.

Fazit: Der Nachteil an der *Maximal State Space*-Methode ist, dass die entstehenden Modelle alle Variablen aller Modi enthalten müssen. Dies führt in vielen Fällen zu längeren Laufzeiten und bei großen Modellen wird der Ansatz schnell unübersichtlich. Auch können die Wechselbedingungen

kompliziert werden (vgl. [48]). Das Modell ist nach dem Überführen in ein *Maximal State Space* Modell auch kein strukturvariables Modell mehr, da es nur einen Satz von Gleichungen und Variablen gibt.

Neue Sprache mit Simulationsumgebung

Eine andere Variante besteht darin, eine neue Sprache und eine Simulationsumgebung zu entwickeln, welche die strukturvariable Modellierung und Simulation unterstützt.

Vorhandene Ansätze sind beispielsweise MOSILAB [54, 61], SOL [96], Hydra [55], DEVS [67, 86], Switched Bond Graphs [19], Ptolemy [70], FUJABA [59], KeYmaera [69] und SimEvents [16]. Im Folgenden werden einige davon näher betrachtet:

MOSILAB ist ein Werkzeug, dessen Modellierungsprache (Mosila) ähnlich zu Modelica ist. Die Beschreibung strukturvariabler Modelle ist mit einer Statechart-ähnlichen Semantik möglich. Strukturwechsel können in MOSILAB auf der obersten Modellebene definiert werden. Da keine symbolische Vorverarbeitung für Höhere-Index-Modelle vorgesehen ist, können nur Modelle mit einem Index-0 simuliert werden.

SOL ist eine experimentelle Sprache, mit der die Modellierung mit Strukturwechseln möglich ist. Dabei können zum einen Modelle mit höherem Index sowie Modelle mit Modi innerhalb von Komponenten simuliert werden. Ein weiterer Vorteil von SOL ist die Kausalisierung bei einem Moduswechsel, da nur die notwendigen Teile eines Modells neu kausalisiert werden.

Hydra stellt eine auf funktionalen Programmiersprachen basierende Sprache dar, die ebenfalls die Modellierung von Strukturwechseln erlaubt. Dabei können beliebige Moduswechsel spezifiziert und Komponenten hinzugefügt oder entfernt werden. Die Sprache und der dahinterliegende Compiler sind auf den Umgang mit strukturvariablen Modellen spezialisiert. Eine Eigenschaft, welche zurzeit noch nicht in anderen Werkzeugen integriert ist, ist die *just-in-time-compilation* von Hydra, die es erlaubt, während der Laufzeit die notwendigen Komponenten und Modelle zu kompilieren.

DEVS (Discrete-Event-Simulation) ist ein Formalismus zum Modellieren und Analysieren von ereignisbasierten Modellen. Der Formalismus wurde um die Modellierung und Simulation strukturvariabler Modelle erweitert.

Zum Hinzufügen oder Entfernen können die Kopplungen von Komponenten angepasst werden. In Matlab ist eine Implementierung dieses Ansatzes verfügbar [66].

Fazit: Die Ansätze bieten Möglichkeiten zur Simulation strukturvariabler Modelle und Notationen zum Beschreiben der Modelle. Dabei haben die unterschiedlichen Umsetzungen in verschiedenen Bereichen ihre Stärken. Bei der einen ist es die *just-in-time compilation*, bei anderen die Möglichkeit Modelle mit hohem Index zu simulieren. Für alle Ansätze müssen die Modelle in der jeweiligen Sprache implementiert sein. Dies schränkt deren Verwendbarkeit für große Modelle ein, da vorhandene Modelle (bspw. aus Bibliotheken) nicht ohne eine erneute Implementierung getestet werden können.

Hybride Dekomposition

Eine weitere Variante zur Simulation strukturvariabler Modelle ist der Ansatz der *hybriden Dekomposition*, wie er in [6] vorgestellt wird. Hier werden die Modi des Modells in einem Standard-Simulationswerkzeug simuliert und die Wechsel zwischen den Modi auf einer Meta-Ebene durchgeführt. Diese koordiniert die Simulationen. Diese Variante setzt kein erneutes Implementieren der Modelle voraus, die Simulation kann in einem Standard-Simulationswerkzeug stattfinden. Die verwendeten Simulationswerkzeuge und deren Solver sind bei diesem Vorgehen nicht auf strukturvariable Modelle spezialisiert.

3.5.4. Auswertung

Wie aus den Beispielen zu erkennen ist, gibt es eine Vielzahl von Ansätzen für strukturvariable Modelle. Der Ansatz der *Maximal State Space* Variante führt zu großen Modellen, in denen zur Simulationszeit oft mehr als notwendig berechnet wird, was damit nicht mehr dem Ziel der strukturvariablen Modellierung entspricht.

Die eigenen Sprachen und Werkzeuge haben gute Eigenschaften, setzen allerdings voraus, dass die Modelle in der entsprechenden Sprache neu implementiert werden. Dies schränkt die Möglichkeiten zur Untersuchung großer, realistischer Modelle ein, die in vorhandenen Werkzeugen existieren.

Der Ansatz der hybriden Komposition stellt eine Möglichkeit dar, vorhandene Modelle wiederzuverwenden und eine Meta-Ebene zur Steuerung der Moduswechsel zu verwenden. Diese Variante erlaubt, vorhandene Modelle als Modi in strukturvariablen Modellen zu verwenden und somit Untersuchungen durchzuführen.

3.6. Zusammenfassung

Strukturvariable Modelle sind für die Zukunft der Modellierung wichtig, können aber in heutigen Standardwerkzeugen noch nicht hinreichend umgesetzt werden.

Zur Simulation der Modelle existieren verschiedene Ansätze, welche zurzeit jedoch noch Prototypen sind. Sie unterliegen daher noch Einschränkungen, sodass eher akademische Beispiele mit ihnen umsetzbar sind. Daher ist noch wenig über den sinnvollen Einsatz strukturvariabler Modelle bekannt.

Allgemeine werkzeug- und sprachunabhängige Modellierungskonzepte, in denen die Wahl der Modi, der Wechselzeitpunkte und der Initialisierung berücksichtigt wird, sind in der heutigen Literatur kaum zu finden. Auch sind noch keine Konzepte für die Modellierung von wiederverwendbaren Komponenten mit Modi vorhanden, da bisher noch nicht auf die Kompatibilität von verschiedenen Modi geachtet wurde.

In dieser Arbeit soll es um die neuen Anforderungen und Modellierungskonzepte zum Umgang mit strukturvariablen Modellen gehen. Das Ziel ist, die verschiedenen Anwendungsgebiete und Aspekte der strukturvariablen Modellierung zu untersuchen und Modellierungskonzepte zu erläutern, um diese Art von Modellen vorteilhaft in der Praxis einzusetzen.

II

Umgang mit strukturvariablen Modellen

4 Anwendungsgebiete

Strukturvariable Modelle sind in verschiedenen Anwendungsgebieten der Modellierung einsetzbar. So kann beispielsweise zwischen Modellen, die ihren Detaillierungsgrad ändern, und denen, die ihr Verhalten ändern, unterschieden werden.

Ziel jeder Anwendung strukturvariabler Modelle sollte es sein, Simulationszeit zu sparen oder genauere Simulationsergebnisse zu erhalten. Für die verschiedenen Anwendungsgebiete wird diskutiert, wie dieses Ziel erreicht werden kann und welche Herausforderungen bei der Modellierung zu lösen sind. Für die anwendungsspezifischen Herausforderungen werden Modellierungskonzepte diskutiert, die aufzeigen, wie mit den Herausforderungen umgegangen werden kann.

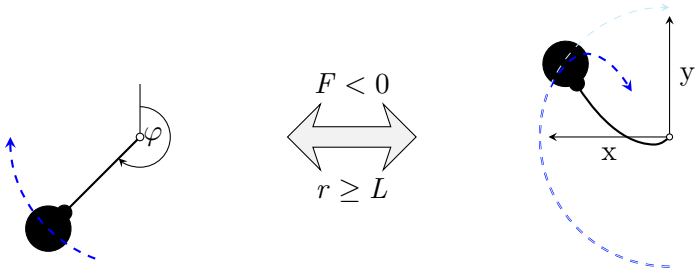
4.1. Verhaltensänderungen

Viele Systeme durchlaufen verschiedene Phasen, die in einem Modell durch verschiedene Gleichungssysteme abgebildet werden sollten. In der konventionellen Modellierung ist jedoch nur ein Gleichungssystem möglich. Daher müssen entweder alle Gleichungen der Phasen integriert werden, was zu komplizierten, nicht oder nur langsam simulierbaren Modellen führen kann. Können nicht alle Phasen integriert werden, so wird oft über die Phasen abstrahiert. Die Modelle sind in diesem Fall oft nicht so genau wie gewünscht oder zu genau, um sie zu berechnen. Werden strukturvariable Modelle verwendet, können die Phasen in Modi ausgelagert werden, welche nur das aktuell relevante Verhalten berücksichtigen. Die einzelnen Modi werden dadurch einfacher und das gewünschte Verhalten kann zu jeder Zeit detailliert berechnet werden.

Das bekannte Satellitenmodell ist ein solches Beispiel, in dem einmal die Rakete und einmal der Satellit berechnet werden. Beide Modi haben ihre eigenen Annahmen und Gleichungen. Jede der Phasen kann einzeln modelliert werden und jeweils nur das Notwendige berücksichtigen.

Das Erstellen der Modi ist einfacher und die Simulationszeit meist reduzierbar. Um solche Arten von Modellen und deren Herausforderungen bei der Modellierung besser zu verstehen, sei ein Pendelbeispiel betrachtet.

Beispiel 4.1 (Pendel): *Betrachtet wird ein Seilpendel, welches zu einer fallenden Masse wird, sobald die Zentrifugalkraft nicht mehr ausreicht, um das Pendel auf der Kreisbahn zu halten. Sobald die Masse wieder in das Seil fällt, wird zum normalen Pendel gewechselt. Abbildung 4.1 veranschaulicht dieses Verhalten.*



$$\ddot{\varphi} = -\frac{g}{L} \cdot sin(\varphi) - D \cdot \frac{g}{L} \cdot \dot{\varphi}$$

$$F = m \cdot g \cdot cos(\varphi) + m \cdot L \cdot \dot{\varphi}^2$$

$$m \cdot \dot{vx} = 0$$

$$m \cdot \dot{vy} = -g \cdot m$$

$$r = \sqrt{x^2 + y^2}$$

Abb. 4.1.: Bewegung und Gleichungen des Pendelmodells mit Verhaltensänderung

Würde das Modell erweitert werden, sodass es alle Informationen beider Phasen abdeckt, so wäre nur ein Modell notwendig. Dabei müsste das Modell Polar- sowie kartesische Koordinaten und die beiden Gleichungssysteme enthalten.

Auch ein Modell mit nur kartesischen Koordinaten wäre denkbar. Aber auch dann würden sich die Gleichungen der beiden Phasen in ihrer Kausalität und ihrem Index unterscheiden [48]. Dies würde das Gesamtmodell komplizierter als notwendig machen, da nicht alle Gleichungen gleichzeitig erforderlich sind. Auch würden die Abfragen, welche Gleichungen aktiv sind, komplex werden. Eine Aufteilung in zwei Modelle führt zu zwei übersichtlichen Modellen, die das Systemverhalten in der jeweiligen Phase detailliert beschreiben.

Bei Verhaltensänderungen können sich die Zustandsvariablen der einzelnen Modi grundlegend unterscheiden, wie in diesem Beispiel. Das Seilpendel wird mit Polarkoordinaten beschrieben, die Zustandsvariablen sind dabei der Winkel (φ) und die Winkelgeschwindigkeit ($\dot{\varphi}$). Die Berechnung für den freien Fall findet mit den kartesischen Koordinaten x, y statt. Findet ein Moduswechsel statt, müssen die Variablen ineinander umgerechnet werden.

Allerdings ist nicht nur die Initialisierung relevant, sondern auch noch weitere Faktoren sind zu beachten:

1. Werden die Wechselbedingungen wie oben beschrieben gewählt, so ergibt sich $F \leq 0$ für den Übergang vom Pendel zur fallenden Masse und für den entgegengesetzten Übergang $x^2 + y^2 > L^2$. Diese zweite Bedingung ist bereits erfüllt, sobald der Moduswechsel zur fallenden Masse stattfindet, da sich die Pendelkugel im Abstand L vom Mittelpunkt befindet. Je nach verwendetem Werkzeug kann dies zu einem sofortigen Wechsel führen, jedoch soll an dieser Stelle noch nicht gewechselt werden.
   Ein solches falsches Zurückschalten zum Pendel kann wie folgt vermieden werden:

   - Zum einen könnte die Wechselbedingung von der fallenden Masse zum Pendel so angepasst werden, dass sie erst eintreten darf, wenn eine definierte Zeit verstrichen ist. Dabei ist die zu verstreichende Zeit von großer Bedeutung, da die Bedingung nicht verpasst werden soll, die gewählte Zeit jedoch lang genug sein muss, damit sich die Pendelkugel bewegt hat.
   - Ebenfalls könnte die Übergangsbedingung so angepasst werden, dass nicht bei $x^2+y^2 > L^2$ zurück gewechselt wird, sondern erst bei $x^2+y^2 > L^2+\epsilon$. Das Modell der fallenden Masse würde dann länger als notwendig simuliert werden. Ein solches Vorgehen ist sinnvoll, wenn eine Zeitangabe nur schwer zu realisieren ist. Das Ergebnis der Simulation wird jedoch verfälscht. Abhängig von der vorzunehmenden Untersuchung muss entschieden werden, ob eine solche Abweichung akzeptabel ist.
   - Ebenfalls kann verlangt werden, dass der Zustand $x^2 + y^2 > L^2$ einmal verlassen wurde, bevor er als Bedingung zum Moduswechsel verwendet werden darf. Dies stellt die Variante dar, in der die Ergebnisse am wenigsten verfälscht werden.

2. Ebenso ist zu entscheiden, wie genau der Moduswechsel modelliert wird. Beim Übergang vom Pendel zur fallenden Masse können die Polarkoordinaten in kartesische Koordinaten umgerechnet werden. Bei dieser Berechnung entsteht ein Rundungsfehler, ansonsten ist der Übergang jedoch nahe an der Realität. Interessanter ist der Übergang in die andere Richtung. Hier fällt die Pendelkugel aus dem freien Fall in ein Seil.
   In der Realität wäre das Seil elastisch (wenn auch vermutlich nur wenig), es würde sich also verlängern und wieder zusammenziehen. Es würde sich kurzzeitig wie ein Feder-/Dämpfer-System verhalten, bis es eingeschwungen ist. Bei diesem Vorgang würde Energie verloren gehen. Eine einfache Umrechnung der Geschwindigkeiten (vx, vy) zur Winkelgeschwindigkeit ($\dot{\varphi}$) ist somit nicht realistisch. Es muss geklärt werden, ob diese Effekte relevant sind und welche Auswirkungen sie auf das Gesamtverhalten haben.
   In vielen Fällen müssen nicht alle Effekte bei einem Übergang betrachtet werden, genauso wenig wie alle Effekte in einem konventionellen Modell beachtet werden müssen. Ein Strukturwechsel ist wieder eine Abstraktion, für die zur konventionellen Modellierung ähnliche Entscheidungen getroffen werden müssen. Eine ungünstig gewählte Abstraktion kann zu Fehlern während der Simulation führen.
   Sind die Seileffekte relevant, so könnte ein Zwischenmodus eingefügt werden, welcher diese Effekte betrachtet. Dieser Zwischenmodus würde dann genau das Übergangsverhalten modellieren, und erst nachdem der Übergang abgeschlossen wäre, dürfte in das normale Pendelmodell gewechselt werden.

Schon an diesem sehr einfachen Beispiel ist zu erkennen, wie viel bei einem Moduswechsel zu beachten ist. Je größer und komplexer die Modelle werden, umso mehr muss auf die Auslegung der Modi, der korrekten Wechselbedingungen und der Initialisierung geachtet werden. Dies sind Modellierungsentscheidungen, die sich für verschiedene Modelle und deren Zweck unterscheiden. Phasen, die mit unterschiedlichen Gleichungssystemen einfacher abzubilden wären als mit einem einzigen, sollten in Modi ausgelagert werden. Dabei sollten sich die Modi so weit unterscheiden, dass sie zu Vorteilen bei der Simulation führen und einfach zu modellieren sind. Andererseits sollten sie sich insoweit ähneln, dass Initialwerte für den jeweils anderen Modus bestimmt werden können.

4.2. Detaillierungsgradänderungen

Bei der Veränderung des Detaillierungsgrades geht es darum, das gleiche Verhalten mit unterschiedlichem Detaillierungsgrad zu modellieren. Es wäre möglich, immer ein sehr detailliertes Modell für die gesamte Simulation zu verwenden. Der Nachteil liegt in den oft langen Simulationszeiten, da das Modell aus vielen Gleichungen besteht oder auch ein steifes Gleichungssystem gelöst werden muss.

Ein Modell wird als *steif* bezeichnet, wenn sich einige Zustandsgrößen im Vergleich zu den restlichen schnell ändern. Dies kann beispielsweise anhand der Eigenwerte der Jacobi-Matrix des Modells festgestellt werden. Liegen die Eigenwerte weit auseinander, handelt es sich um ein steifes Modell [12]. Wird ein Solver für steife Modelle verwendet, so sind lange Rechenzeiten während der Einschwingphase notwendig. Auch ist ein impliziter Solver erforderlich, was bedeutet, dass die Rechenzeit kubisch mit der Anzahl der Variablen im Gleichungssystem ansteigt.

Oft ist es jedoch nicht notwendig, ein sehr detailliertes Modell für die gesamte Simulation zu verwenden. In vielen Anwendungsbereichen genügt ein weniger rechenintensives Modell für den Großteil der Simulation. Nur in kritischen Bereichen sollte zum detaillierteren Modell gewechselt werden. Dies kann Vorteile in der Laufzeit bringen oder zu genaueren Ergebnissen führen, da ein Modell dadurch zu jeder Zeit so genau wie nötig und so einfach wie möglich ist. In einigen Fällen führt dieser Ansatz sogar zu genaueren Ergebnissen bei schnellerer Simulationszeit. Dies tritt beispielsweise ein, wenn durch den Wechsel des Detaillierungsgrades die Steifheit eines Modells aufgehoben wird. Ebenfalls können beide Vorteile erzielt werden, falls ein Modellierer sonst zu einem Modell zurückgreifen müsste, das weder so schnell noch so genau wie gewünscht simuliert. Ist der Detaillierungsgrad anpassbar, ist ein solcher Kompromiss oft nicht mehr erforderlich.

Beispiele für solche Modelle sind:

- Start eines Motors $\rightarrow$ konstanter Betrieb eines Motors
- Normalbetrieb eines Motors $\leftrightarrow$ klopfender Betrieb
- Normalbetrieb einer Windmühle $\leftrightarrow$ Betrieb bei hohen Windstärken (eventuell starke Biegung der Rotorblätter)

Ein Beispiel sei näher betrachtet.

Beispiel 4.2 (Motorsimulation): *Es sei ein Otto-Motor mit Drosselklappe, Sammler und Zylinder betrachtet. Dabei wird der Drosselklappenwinkel je nach Leistungsbedarf angepasst. Wenn sich der Winkel der Drosselklappe ändert, verändert sich auch der Druck im Sammler. Wenn der Drosselklappenwinkel eine Zeit lang konstant gehalten wird, so ist auch der Druck im Sammler und damit auch der Durchfluss konstant. In diesem Fall muss die Durchflussgleichung in der Drosselklappe nicht mehr berechnet werden. Die Drosselklappe kann durch einen konstanten Massenstrom ersetzt werden. Abbildung 4.2 zeigt schematisch den Aufbau der beiden Modi.*

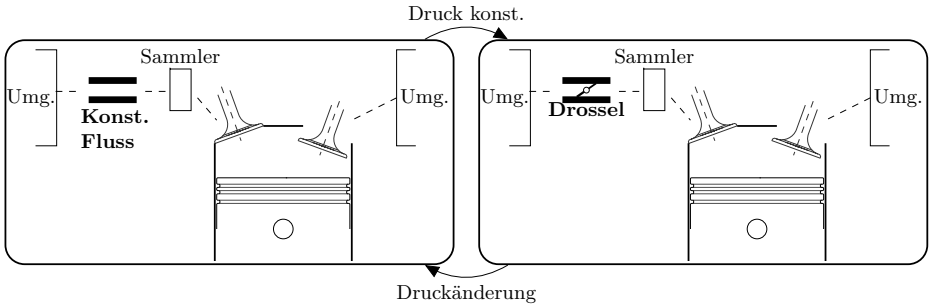


Abb. 4.2.: Schematische Darstellung eines Detaillierungsgradwechsels für einen Verbrennungsmotor

Das Modell wurde zu Testzwecken in Modelica implementiert und mit Dymola simuliert.

Für eine Simulation von 20 Sekunden werden 19 Sekunden ohne Moduswechsel, mit sieben Moduswechseln nur noch 12 Sekunden benötigt. Dabei ist zu berücksichtigen, dass beide Modi kompiliert werden müssen und der Moduswechsel ebenso Zeit in Anspruch nimmt. Dies ist in den 12 Sekunden bereits berücksichtigt.

Da die Genauigkeit des Modells durch die Strukturwechsel nicht negativ beeinflusst werden soll, wurden die Simulationsergebnisse des konventionellen und des strukturvariablen Ansatzes verglichen. Abbildung 4.3 zeigt einen Ausschnitt aus dem Temperaturverlauf. Die Ergebnisse unterscheiden sich nicht wesentlich. Im Vergleich zu realen Messdaten haben beide Simulationsergebnisse ähnliche Abweichungen.

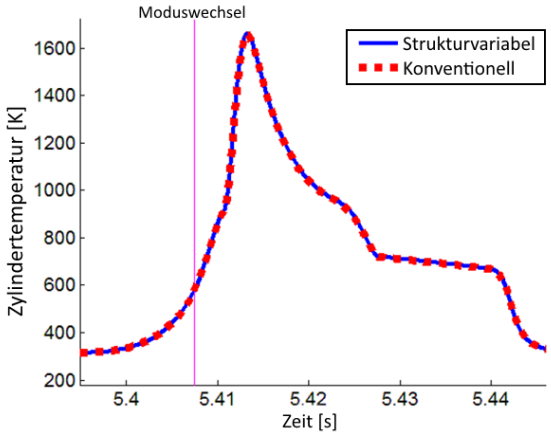


Abb. 4.3.: Vergleich der simulierten Zylindertemperatur von einem konventionellen und einem strukturvariablen Modell

Die Zielsetzung bei dieser Anwendung ist ein Genauigkeits- und/oder Zeitgewinn durch Strukturwechsel. Beide Aspekte gehören zusammen, da sie ineinander umformbar sind, je nachdem aus welcher Sicht das Problem betrachtet wird.

- Wurde zuvor für die gesamte Simulation ein genaueres Modell mit langer Laufzeit verwendet, so kann mit einer zeitweisen Vereinfachung des Modells während der Simulation Zeit gespart werden.
- Wurde zuvor für die gesamte Simulation ein einfacheres Modell mit kurzer Laufzeit verwendet, so kann mit einer zeitweisen genaueren Betrachtung während der Simulation die Genauigkeit erhöht werden.

Das Endmodell mag dasselbe strukturvariable Modell sein, aber abhängig vom originalen Modell oder Problem ist die Sichtweise eine andere.

Bei Detaillierungsgradänderungen ändert sich oft nicht das ganze Modell, sondern die Änderungen betreffen nur einzelne Komponenten eines Modells, die je nach Zustand vereinfacht oder detaillierter modelliert werden. Deshalb sind oft viele identische Variablen in den Modi enthalten. Dabei müssen gleich benannte Variablen jedoch nicht zwangsläufig dasselbe repräsentieren. Manche Modi können auch wesentlich mehr Variablen haben als andere, die bei einem Moduswechsel initialisiert werden müssen.

Folgende Fälle sind dabei zu beachten:

1. **Sinnvolle Moduswechsel:**
   - **Vom detaillierten zum groben Modus:** In vielen Fällen würde hier die Genauigkeit nicht leiden, wenn das genauere Modell länger gültig ist als notwendig. Jedoch würde dies meist auch die Simulationszeit verlängern. Findet ein zu früher Wechsel statt, können wichtige Informationen verloren gehen.
   - **Vom groben zum detaillierten Modus:** Der grobe Modus muss genügend Informationen liefern, um den detaillierten Modus zu initialisieren. Ebenso sollte versucht werden, den Wechsel einzuleiten, bevor ein kritischer Moment eintritt, da der grobe Modus für den kritischen Bereich zu abstrakt sein kann. In manchen Fällen wird der kritische Punkt im abstrakten Modell zu spät erkannt. Würde an dieser Stelle ein Moduswechsel eingeleitet werden, kann dies zu falschen Ergebnissen führen. In einem solchen Fall sollte der Moduswechsel den nächsten Modus in der Vergangenheit starten, in der die Daten des abstrakten Modus noch gültig waren. Mehr zum Zurücksetzten in die Vergangenheit ist in Abschnitt 4.4 zu finden.
   - **Fließender Übergang:** An einigen Stellen ist es sinnvoll einen fließenden Übergang vorzusehen, beispielsweise indem die Modi der beiden Detaillierungsstufen parallel simuliert werden und mit einer vorgegebenen, sich ändernden Gewichtung zum anderen Modus gewechselt wird. Dies erhöht den Berechnungsaufwand, kann aber zu stetigen Übergängen führen.
   - **Zwischenmodus:** Es wird ein Modus vorgesehen, der den Übergang von einem Modus zum nächsten beschreibt. Ein Moduswechsel wird dann durch zwei Moduswechsel und einen Modus ersetzt, der den Übergang modelliert.

2. **Die Wahl der Modi:** Hier ist zu entscheiden, wie viele und welche Modi verwendet und wie diese zur Initialisierung der anderen Modi verwendet werden. Jeder Modus muss modelliert werden und dieser Aufwand sollte durch positive Effekte während der Simulation gerechtfertigt werden.

   Sind sich die Modi zu ähnlich, sind die positiven Effekte eher gering. Sind sie zu unterschiedlich, so ist die Initialisierung während eines

Moduswechsels eventuell sehr kompliziert zu beschreiben. Ein Kompromiss zwischen Aufwand und positiven Effekten ist notwendig.

3. **Vorsehen von Modi in Komponenten:** Es ist zu entscheiden, ob Modi in Komponenten auftreten. Mit Modi in Komponenten ergeben sich verschiedene Konstellationsmöglichkeiten und damit eine hohe Flexibilität des Modells. Ebenfalls können gut ausgelegte Komponenten in anderen Modellen wiederverwendet werden, wodurch sie dort ebenfalls positive Effekte erzielen können, ohne erneuten Mehraufwand zu produzieren.

4. **Chattering vermeiden:** Es soll verhindert werden, dass zwischen zwei Modi ständig hin und her gewechselt wird. Aus diesem Grund sollte bei einem Moduswechsel eine Hysterese vorgesehen werden.

Das Ziel eines solchen Wechsels sollte immer im Vordergrund stehen. Wird weder die Genauigkeit gesteigert noch die Simulationszeit verkürzt, ist ein Moduswechsel nicht sinnvoll.

4.3. Hinzufügen/Entfernen von Modellkomponenten

Es sollen während der Simulation nur Komponenten betrachtet werden, welche das Systemverhalten im aktuellen Zustand beeinflussen.

Dabei sind grob zwei Varianten zu betrachten: zum einen die Änderung der Menge von gleichartigen Komponenten und zum anderen das Hinzufügen/Entfernen von Modellteilen.

4.3.1. Ändern der Anzahl von gleichen Komponenten

In einigen Modellen ist es sinnvoll, eine Menge identischer Komponenten zu haben und deren Anzahl während der Simulation den Gegebenheiten anzupassen.

Beispiele hierfür sind:

- Simulation von Populationen und deren Interaktion: Dabei werden Organismen simuliert, die miteinander interagieren und hinzukommen oder wegfallen können [86].

- Verkehrssimulation an einer Kreuzung: Dabei werden Fahrzeuge in Abhängigkeit davon, ob sie auf der Kreuzung sind, hinzugefügt oder entfernt.
- Flughafensimulation mit an- und abfliegenden Flugzeugen: Dabei werden nur Flugzeuge in der Nähe des Flughafens betrachtet. Flugzeuge, die weiter weg sind, können entweder weniger detailliert oder auch gar nicht simuliert werden.
- Rohrleitungen aus mehreren Rohrelementen: Dabei wird die Anzahl der Rohrelemente den Gegebenheiten angepasst (Gridänderung).

Als allgemeines Beispiel seien fallende Dominosteine betrachtet (vgl. [12]).

Beispiel 4.3 (Dominosteine): *Betrachtet sei eine Menge von aufgestellten Dominosteinen, die in einer Reihe in einem vorgegebenen Abstand aufgestellt sind. Jeder einzelne Dominostein hat das gleiche physikalische Verhalten, befindet sich jedoch jeweils in einem anderen Zustand. Entweder er fällt, ist gefallen oder ist noch nicht angestoßen worden.*

Um in der Simulation nur das Notwendige zu berücksichtigen, sollen nur fallende Steine simuliert werden. Ein neuer Stein beginnt zu fallen, wenn er von einem fallenden Stein angestoßen wird. Beim Auftreffen auf einen noch stehenden Stein wird ein Teil der Bewegungsenergie an den neuen Stein übergeben und dieser setzt sich ebenfalls in Bewegung. Fällt ein fallender Stein auf den Boden, so soll dieser aus dem Modell entfernt werden.

Ein Stein gilt als gefallen, sobald er um 90 Grad gekippt ist. Das Aufeinanderliegen von Steinen wird nicht betrachtet. Der Einfachheit halber wird ebenso angenommen, dass ein Stein nicht um 90 Grad fallen kann, bevor der Stein davor um 90 Grad gefallen ist.

Solche Modelle können in objektorientierten Sprachen durch das Anpassen von Arraygrößen oder durch das Hinzufügen/Löschen von Teilmodellen in anderen Simulationsumgebungen realisiert werden. Das Gleichungssystem des Gesamtmodells ändert sich dementsprechend.

Ein solches Modell kann mit nur einem Modus dargestellt werden, es führen jedoch mehrere Transitionen aus dem Modus wieder in sich selbst zurück, siehe Abbildung 4.4. Je nachdem, ob ein Objekt hinzugefügt oder entfernt werden soll, muss die Transition die gewünschten Änderungen einleiten. Eine Transition sollte daher durch die Wechselbedingung eindeutig identifiziert werden können.

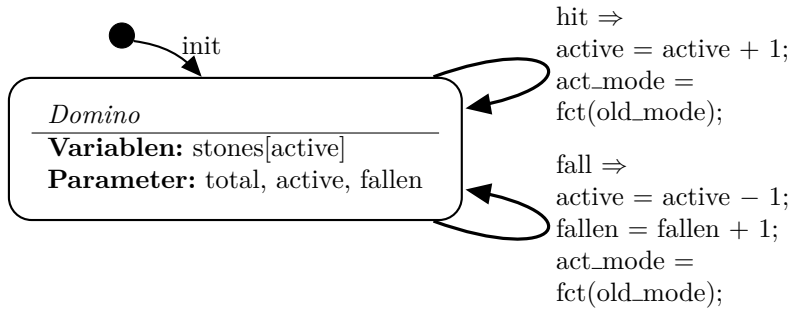


Abb. 4.4.: Strukturvariables Modell von Dominosteinen

Eine weitere Schwierigkeit betrifft die Identitäten der einzelnen Objekte. Abbildung 4.5 zeigt einen Ausschnitt, wie ein Array von variabler Länge während der Simulation aussehen könnte.

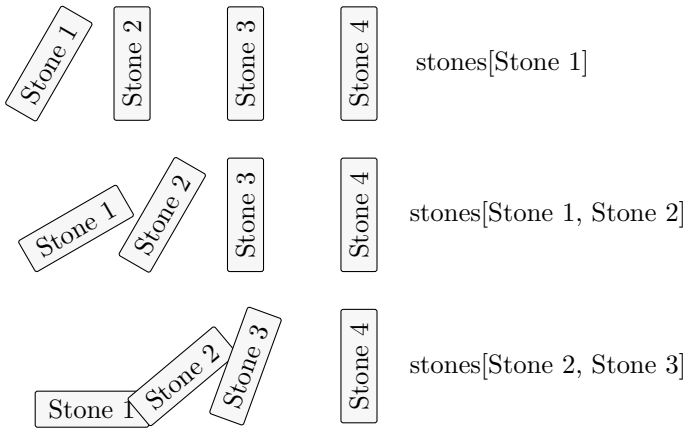


Abb. 4.5.: Beispiel einiger Modi des Dominosteinmodells

Je nach aktuellem Zustand stehen an einer Position im Array verschiedene Dominosteine. Somit ist anhand der Position im Array der exakte Dominostein nicht eindeutig identifizierbar.

Kommt ein neuer Dominostein zum Modell hinzu, so müssen die Kraft und der Winkel, in dem er angestoßen wird, aus dem ihn treffenden Dominostein abgeleitet werden. Die Winkelgeschwindigkeit des anstoßenden Dominosteins muss gleichzeitig reduziert werden. Wird ein Stein entfernt, müssen die noch fallenden Steine im Array um je Eins nach vorne gesetzt werden.

Für diese Art von Modellen sollten Funktionen während eines Moduswechsels vorgesehen werden. Mithilfe der Berechnungen können Initialwerte aus alten Daten bestimmt werden, während in Funktionen notwendige Initialwerte beliebig definiert werden können, eventuell sogar unabhängig vom Vorzustand des Modells.

Dabei kommen Komponenten hinzu oder fallen weg, wenn sie die Systemgrenze überschreiten. Je nachdem wie die Grenze gewählt ist, sind verschieden viele Komponenten enthalten. Die Komponenten an sich beschreiben jedoch jeweils ein eigenes, reales Objekt.

Ein etwas anderer Fall ist die Gridänderung. Bei dieser wird die Anzahl der Komponenten geändert, um ein reales Objekt mehr oder weniger feingranular zu betrachten. Neue Komponenten müssen die Eigenschaften vorheriger Komponenten abbilden und dementsprechend initialisiert werden. Kommen neue Komponenten hinzu, müssen Daten generiert werden, sodass beispielsweise die Masse und Energie des Objekts identisch bleibt.

Der umgekehrte Fall tritt ein, wenn mehrere Komponenten zu einer zusammengefasst werden sollen. Hier müssen die Daten reduziert werden, wozu beispielsweise eine Mittelwertbildung (bspw. Temperaturen) oder Addition (bspw. Massen) verwendet werden kann. Dabei sollten ebenfalls die Massen und Energiebilanzen beachtet werden.

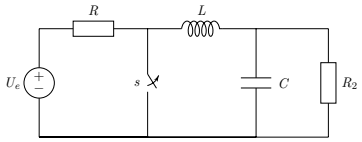
4.3.2. Komponenten hinzufügen/entfernen

Das Hinzufügen oder Entfernen von Komponenten kann zu Zeitvorteilen bei der Simulation führen, aber auch das Modellverhalten beeinflussen. Einfache Beispiele sind in der Elektrotechnik zu finden. So kann beispielsweise in einer elektrischen Schaltung ein Teil der Schaltung durch einen Schalter vom restlichen Stromkreis getrennt werden.

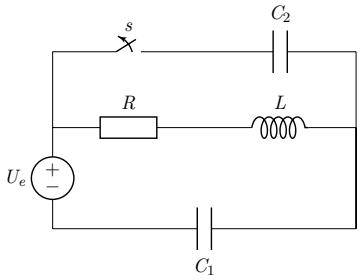
Abbildung 4.6 zeigt einige Schaltkreise, in denen Komponenten abhängig von der Schalterstellung hinzugenommen oder entfernt werden.

Ein Beispiel aus der Mechanik ist in Abbildung 4.7 gezeigt.[1] Hier wird ein Wagen mit einer Kugel modelliert, der eine Rampe hinunter fährt. Kommt der Wagen am Ende der Rampe an, wird er gestoppt. Die Kugel fällt vom Wagen und muss weiter berechnet werden, während der Wagen entfernt wird, da er sich nicht mehr bewegt. Soll das exakte Stoppen des Wagens berechnet werden, können beide Modelle kurzzeitig gemeinsam, aber unabhängig voneinander berechnet werden.

[1] Das Beispiel stammt aus [60].



(a) Bei **geöffnetem** Schalter kommen Spule L, Kondensator C und Widerstand R_2 hinzu.



(b) Bei **geschlossenem** Schalter kommt der Kondensator C_2 hinzu.

Abb. 4.6.: Beispiele elektrischer Schaltkreise, bei denen Teile des Schaltkreises durch den Schalter abgetrennt werden können

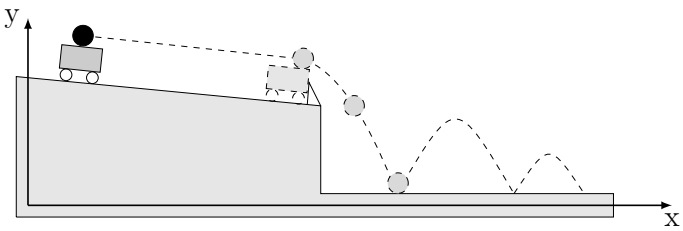


Abb. 4.7.: Ein Wagen mit Ball, bei dem der Wagen gestoppt werden kann und der Ball vom Wagen fällt

Beim Hinzufügen von Komponenten ist die Initialisierung von großer Bedeutung. Der Modus ohne die Komponenten wird keine oder wenige Informationen über die Komponente haben. Bei einem Moduswechsel müssen jedoch Daten für die Initialisierung bereitgestellt werden.

Dies fällt besonders auf, wenn eine speichernde Komponente, wie beispielsweise ein Kondensator oder Behälter, zunächst simuliert und dann entfernt wird. Wird diese Komponente wieder hinzugefügt, muss sie initialisiert werden. Hierzu ist es notwendig Funktionen vorzusehen, die das Setzen erforderlicher Startwerte erlauben.

Ebenso müssen die Verbindungen zwischen den Komponenten betrachtet werden, da sie beim Hinzufügen oder Entfernen einer Komponente angepasst werden müssen. Wird beim elektrischen Schaltkreis (Abbildung 4.6(b)) der Kondensator hinzugenommen, so muss dieser in die Gleichungen des Schaltkreises integriert werden. Wird er wieder entfernt, so muss die Verbindung zum restlichen Modell wieder entfernt werden.

4.4. Zurücksetzen in die Vergangenheit

Die Möglichkeit, bei einer Simulation in die Vergangenheit zurückzugehen und dort eine andere Strategie oder ein anderes Modell anzusetzen, ist ein großer Vorteil in der Modellierung. In der Realität ist es im Normalfall nicht möglich, die exakten Bedingungen eines beliebigen vergangenen Zeitpunkts wiederherzustellen. Beispiel 4.4 präsentiert ein Modell, für das ein Zurückgehen in die Vergangenheit vorteilhaft ist.

Beispiel 4.4 (Motorklopfen): *Betrachtet wird eine Motorsimulation, in der ein Motor ins Klopfen gerät. Das Klopfen eines Motors ist ein komplizierter Vorgang und würde das Modell verkomplizieren, falls es die ganze Zeit über simuliert wird. Für einen Großteil der Simulation kann jedoch ein vereinfachtes Modell verwendet werden, in dem kein Klopfen enthalten ist. Dieses wird verwendet, bis das Klopfen berücksichtigt werden muss, danach wird das detailliertere Modelle verwendet. Allerdings haben sich die Simulationsergebnisse an diesem Punkt eventuell schon aus dem Standardbereich entfernt und das weniger detaillierte Modell ist in diesem Grenzbereich nur bedingt gültig.*

Es ist dann von Vorteil, das detaillierte Modell in der Vergangenheit zu starten. So findet der Wechsel statt, während sich das weniger detaillierte Modell noch in einem gültigen Bereich befand.

Ein weiterer Anwendungsfall ist das Ändern einer Strategie während einer Simulation, beispielsweise der Regelung in Beispiel 4.5.

Beispiel 4.5 (Anpassen der Regelstrategie): *Beim Testen einer Regelung kann es vorkommen, dass die Regelung falsch reagiert. Bei großen Modellen ist es oft unerwünscht, die komplette Simulation erneut zu starten. Alternativ könnte die Simulation zu einem Zeitpunkt t_x in die Vergangenheit gehen und den kritischen Punkt mit einer adaptierten Regelstrategie erneut simulieren. Dies funktioniert nur, wenn sich die adaptierte Regelung bis zum Zeitpunkt t_x identisch zur alten Strategie verhält. Unterscheiden sich die Strategien schon vorher, würde der Zustand eventuell nie erreicht werden, womit der Ansatz ungültig wäre.*

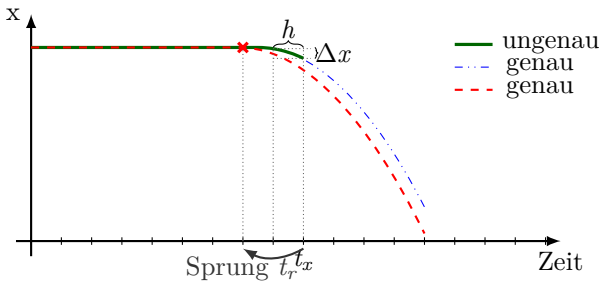


Abb. 4.8.: Transition in die Vergangenheit

Abbildung 4.8 zeigt ein abstraktes Beispiel, in dem sich eine Zustandsgröße x kaum ändert, aber dann zu einer Zeit t_x um ein Δx abfällt. Dieses Delta kann als Ereignis detektiert werden. Eventuell ist die Schrittweite aber viel zu groß oder der Modus zu ungenau und das Delta ist schon zu groß geworden. In einem solchen Fall kann eine vordefinierte Zeit t_r in die Vergangenheit gegangen werden, um in einem unkritischen Bereich in den anderen Modus zu wechseln und den kritischen Bereich erneut zu berechnen.

Um solche Wechsel zu realisieren, müssen nicht nur Enddaten einer Simulation, sondern mindestens alle Daten der Simulation des letzten Modus oder sogar die Simulationsdaten aller Modi für größere Zeitsprünge zugänglich sein.

4.5. Simulation in verschiedenen Simulationsumgebungen

Es kann notwendig oder sinnvoll sein, nicht nur das Modell, sondern auch das Simulationswerkzeug zu ändern. Mit einem solchen Wechsel des Werkzeugs kann in jeder Phase das geeignetste Werkzeug verwendet werden.

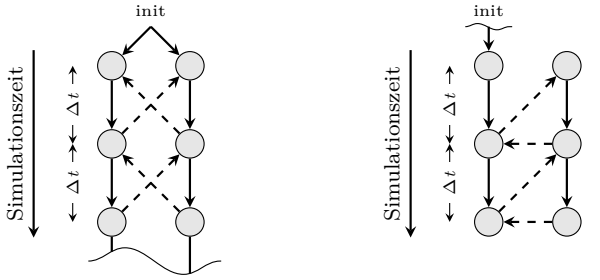
Um von einem Simulationswerkzeug zu einem anderen überzugehen, muss eine Schnittstelle vorhanden sein, über die Daten ausgetauscht werden können. Des Weiteren ist die Kompatibilität der Modelle, besonders die Übergabe von geeigneten Startwerten, relevant. Sind Modi in verschiedenen Werkzeugen realisiert, so unterscheiden sich diese oft wesentlich. Dies kann zu Schwierigkeiten bei der Bestimmung der Startwerte führen, da Variablen eventuell komplett anders heißen oder unterschiedliche Bedeutungen haben.

4.6. Co-Simulation

Eine weitere Variante, das Simulationswerkzeug während der Simulation zu ändern, ist die Co-Simulation. Im Gegensatz zu der oben besprochenen Änderung des Werkzeugs ist die Idee bei der Co-Simulation, Modelle parallel zu simulieren. Alle Modelle sollen über die gesamte Zeit und nicht abwechselnd für verschiedene Zeitabschnitte simuliert werden.

Bei der Co-Simulation werden die Modelle gekoppelt und beeinflussen sich gegenseitig. Dabei werden meist mehrere Werkzeuge gekoppelt. Modelle können aber auch im gleichen Werkzeug in Teilmodelle unterteilt werden, um diese parallel zu simulieren. So können unter anderem steife Modelle geteilt werden: Ein Teil enthält die schnell veränderlichen Größen und der andere die sich langsam ändernden Größen. Die Steifheit des Problems kann somit reduziert werden.

Zur Kopplung der Simulationen gibt es mehrere Varianten, zwei davon sind die feste Kopplung (*strong coupling*) und die lose Kopplung (*loose coupling*), siehe Abbildung 4.9 [85]. Bei der festen Kopplung werden Modelle parallel iterativ simuliert. Das bedeutet, es kann auch in die Vergangenheit zurückgegangen werden. Dies geschieht, bis ein Konvergenzkriterium erreicht ist. Erst dann wird der nächste Zeitschritt simuliert.



(a) Feste Kopplung (b) Lose Kopplung

Abb. 4.9.: Datenaustausch-Varianten bei der Co-Simulation

Bei der losen Kopplung (*sequential staggered solution* [24]) werden die Simulationen sequentiell ausgeführt. Dabei wird Modus A zunächst für eine Zeit t_1 bis t_2 simuliert. Die Enddaten der Simulation zum Zeitpunkt t_2 werden zur Initialisierung von Modus B zur Zeit t_1 verwendet. Ebenfalls können die gesamten Daten der Simulation von Modus A als Eingangsdaten für die Simulation von Modus B verwendet werden, zum Beispiel als *Lookup-Table*.

Wurde Modus B ebenfalls bis zum Zeitpunkt t_2 simuliert, werden dessen Daten zur Initialisierung von Modus A zur Zeit t_2 verwendet. Dies wird fortgesetzt, bis die Endzeit der Simulation erreicht ist.

Bei der Betrachtung der losen Kopplung fällt auf, dass dies prinzipiell einem strukturvariablen Modell entspricht, da zwei unterschiedliche Modelle sequentiell simuliert und Daten zwischen ihnen ausgetauscht werden.

Im Gegensatz zu strukturvariablen Modellen werden bei der Co-Simulation die Modellkomponenten parallel simuliert, wobei die ausgetauschten Daten oft Parameter betreffen, welche während der Simulation der parallelen Komponente konstant bleiben. Die Komponenten haben je ihre eigene Dynamik und werden parallel berechnet. Es sind daher für beide Modelle Ergebnisse für den ganzen Simulationszeitraum vorhanden, wenn auch eventuell nicht mit denselben Schrittweiten. Bei strukturvariablen Modellen wird die Dynamik in den verschiedenen Modi jeweils fortgesetzt und nicht parallel berechnet. Die Ergebnisse für den gesamten Simulationszeitraum setzen sich aus den Ergebnissen der einzelnen Simulationen zusammen.

Beispiel 4.6 (Co-Simulation): *Ein Beispiel für die Co-Simulation ist ein Verbrennungsmotor, der mit einer Motorsteuerung gekoppelt ist. Jedes Modell ist einzeln und eventuell sogar in unterschiedlichen Werkzeugen implementiert (beispielsweise Modelica für den Motor und Simulink für die Steuerung). Um sinnvolle Ergebnisse zu erhalten, werden die beiden Modelle parallel simuliert und synchronisieren sich, beispielsweise über die lose Kopplung. Bei diesem Beispiel wird bei jedem zweiten Wechsel in die Vergangenheit gegangen und zwar zurück zu dem Zeitpunkt, bei dem der vorherige Modus gestartet wurde. Jeder Zeitraum wird somit zwei Mal simuliert, einmal mit dem Modell des Verbrennungsmotors und einmal mit dem der Motorsteuerung.*

4.7. Konsistente Initialisierung

Zum Lösen des initialen Gleichungssystems können verschiedene Verfahren angewendet werden. Es ist möglich, Algorithmen [8], Optimierungsverfahren [10] oder auch Homotopie-Ansätze [11] zu verwenden, um nur einige Beispiele zu nennen.

In allen Verfahren wird vor der Simulation das initiale Gleichungssystem gelöst, welches aus den Gleichungen des Modells und benutzerdefinierten Vorgaben besteht. Die Verfahren finden im Normalfall im Hintergrund eines Simulationswerkzeugs statt. Prinzipiell entspricht dieses Vorgehen einem strukturvariablen Ansatz, da ein Gleichungssystem vor einem anderen gelöst wird und die Startwerte auf definierte Weise übergeben werden. Jedoch findet der Wechsel nicht während der Laufzeit statt, sondern davor.

Wird in Dymola beispielsweise als Solver *DASSL* verwendet, so ist laut [8] ein Verfahren implementiert, welches während der Initialisierung ein implizites Euler- und Newton-Verfahren verwendet. Es findet also bereits während der Initialisierung eine Simulation statt.

4.8. Zusammenfassung

Für die strukturvariable Modellierung gibt es viele Anwendungsgebiete. Dabei verfolgen diese das Ziel, genauere Simulationsergebnisse zu erhalten oder die Simulationszeit zu verkürzen. Dabei unterscheidet sich für die Anwendungsgebiete, wie dieses Ziel erreicht werden soll:

- **Verhaltensänderungen:** Durch die Abbildung verschiedener Phasen wird zu jeder Zeit nur das Notwendige betrachtet, wodurch die einzelnen Phasen häufig detaillierter als in einem konventionellen Modell betrachtet werden können.
- **Detaillierungsgradänderungen:** In jeder Phase wird das System nur so detailliert wie notwendig betrachtet. Größen, die unnötig sind, werden entfernt oder vereinfacht abgebildet.
- **Hinzufügen/Entfernen von Komponenten:** Es werden nur die notwendigen Komponenten im Modell betrachtet. Beeinflussen Komponenten das Systemverhalten nicht signifikant, so werden sie entfernt. Bei Gridänderungen wird ein Modell nur so feingranular wie notwendig betrachtet.
- **Zurücksetzen in die Vergangenheit:** Die Genauigkeit kann gesteigert werden, wenn durch das Zurücksetzen in die Vergangenheit genauere Initialwerte gesetzt werden.
- **Simulation in verschiedenen Simulationsumgebungen:** Für jeden Modus wird das Simulationswerkzeug verwendet, welches für ihn am besten geeignet ist; damit kann Zeit gespart und die Genauigkeit gesteigert werden.
- **Co-Simulation:** Die Co-Simulation ist ein Sonderfall, wobei das Ziel die parallele und nicht die sequentielle Simulation ist. Strukturvariable Modelle können die lose Kopplung der Co-Simulation abbilden.
- **Konsistente Initialisierung:** Vor der Simulation des eigentlichen Modells wird ein weiteres Modell simuliert, dessen Ziel es ist, konsistente Startwerte für das eigentliche Modell zu liefern. Die konsistente Initialisierung führt zu genaueren Ergebnissen, wenn bessere Startwerte gefunden werden.

Alle Anwendungsgebiete setzten voraus, dass die Modi und Transitionen sinnvoll gewählt wurden. Werden diese ungünstig gewählt, sind keine Vorteile durch den Einsatz strukturvariabler Modelle zu erwarten.

5 Modellierung

Im folgenden Kapitel wird untersucht, welche Eigenschaften für die Beschreibung strukturvariabler Modelle notwendig sind, um diese für die bereits vorgestellten Anwendungsgebiete zu verwenden. Dazu werden unter anderem die Auslegung der Modi und Transitionen detailliert betrachtet sowie der hierarchische und modulare Aufbau der Modelle. Die unterschiedlichen Aspekte und Entscheidungen, die für die strukturvariable Modellierung erforderlich sind, werden diskutiert. Daraus ergeben sich die notwendigen Eigenschaften und Konzepte zum Entwerfen strukturvariabler Modelle.

Die in diesem Kapitel verwendete Notation ist eine Mischung aus Statecharts [36, 37] und Klassendiagrammen der UML 2 [87]. Sie wird anhand von Beispielen eingeführt. In Kapitel 6.1 wird die verwendete Notation formal hinterlegt.

Die für die Modellierung zu treffenden Entscheidungen beeinflussen sich gegenseitig. Um die Erklärung möglichst einfach zu halten, erfolgt die Betrachtung dennoch einzeln.

5.1. Wahl der Modi und Transitionen

Eine der ersten Entscheidungen ist, welche und wie viele Modi verwendet werden sollen und auf welcher Modellebene diese spezifiziert werden. Dabei sind ähnliche Überlegungen notwendig wie bei der konventionellen Modellierung (Detaillierungsgrad, Initialisierung, etc.). Aber anstatt genau ein Modell auszulegen, welches das gesamte Verhalten beschreibt, müssen jetzt mehrere Modi ausgelegt werden. Jeder dieser Modi besteht aus Gleichungen und Variablen und muss ein numerisch lösbares Gleichungssystem ergeben.

Werden viele Modi gewählt, so müssen diese alle implementiert werden, was Aufwand erzeugt. Verschiedene Phasen während der Simulation sind durch viele Modi sehr feingranular abgebildet.

Bei vielen Modi sind sich die einzelnen Modi oft ähnlich, was eine Berechnung der Initialisierung des nächsten Modus vereinfacht. Allerdings bedeutet jeder Moduswechsel auch eine potenzielle Fehlerquelle, da der Solver gestoppt werden und eine erneute Initialisierung stattfinden muss. Diese Initialisierung ist wieder eine Abstraktion und erfordert das Lösen eines Initialwertproblems, was Zeit kostet und Berechnungsungenauigkeiten enthalten kann. Wenn einige Modi sich sehr ähneln, ist es besser diese zusammenzufassen, um Mehraufwand und Fehlerquellen zu vermeiden.

Werden nur wenige Modi verwendet, so sind diese oft sehr unterschiedlich, was das Finden der Initialwerte bei Moduswechseln erschweren kann. Da weniger Modi und Transitionen ausgelegt werden müssen, ist der Modellierungsaufwand meist geringer, die verschiedenen Phasen eines Systems sind jedoch nur grob abgebildet.

Auch die Entscheidung, ob ein Modell mit mehreren Modi abgebildet werden sollte, obwohl es eventuell durch Bedingungen und ein etwas komplizierteres Modell in nur einem Modus abbildbar wäre, ist eine wichtige Fragestellung. Es gibt Problemstellungen, bei denen der Aufwand strukturvariabler Modelle nicht gerechtfertigt ist und ein konventionelles Modell verwendet werden sollte.

Es ist zu entscheiden, ob die Modi auf der obersten Modellebene oder in Komponenten beschrieben werden. Werden Modi bereits in Komponenten vorgesehen, so sind die Modelle flexibel während der Simulation, da sich eine Vielzahl von Kombinationsmöglichkeiten ergibt. Ebenso können Komponenten mit ihren Modi in unterschiedlichen Kontexten verwendet werden. Die Modell müssen dafür entsprechend aufgebau werden.

Im Folgenden werden verschiedene Beispiele von Modellen mit Modi auf unterschiedlichen Ebenen gezeigt. Dabei wird erläutert, was bei deren Auslegung zu berücksichtigen ist und welche Eigenschaften notwendig sind, um konsistente, wiederverwendbare Komponenten zu erstellen.[1]

5.1.1. Strukturänderungen auf oberster Modellebene

Zunächst seien Modelle betrachtet, bei denen Modi nur auf der obersten Modellebene auftreten. Jeder Modus ist eine Komponente und besteht aus
A1 **Gleichungen und Variablen**.

[1] Die Nummern am Rand markieren die Diskussionen und Eigenschaften, welche die Grundlage der Formalisierung aus Kapitel 6.1 bilden.

Diese Gleichungen beschreiben das physikalische Verhalten. Da hierarchische Modelle betrachtet werden, kann ein Modus wiederum aus mehreren Komponenten bestehen, die miteinander in Relation stehen. Im Folgenden wird die Hierarchie noch nicht betrachtet, diese wird erst in Abschnitt 5.1.2 eingeführt. Hier wird zunächst davon ausgegangen, dass jeder Modus eine Komponente ist.

Einer der Modi ist der **initiale Modus**, der zu Beginn der Simulation verwendet wird. Dieser kann durch Startwerte **initialisiert** werden. A2 A3

Jeder Modus auf der obersten Modellebene kann einen eigenen Solver haben und in einem eigenen Simulationswerkzeug simuliert werden. Dazu muss jeder Modus ein eigenständiges, simulierbares Modell bilden.

Einfachstes Beispiel

In dem in Abbildung 5.1 gezeigten Beispiel gibt es zwei Modi (A, B), zwischen denen gewechselt werden kann. Die Modi sind in Rechtecken mit abgerundeten Ecken dargestellt. Die **Transition** zwischen den Modi ist mit einem Pfeil dargestellt und **enthält die Wechselbedingung und die Initialisierung**. Im Beispiel wird von A nach B gewechselt, sobald die Bedingung a_c eintritt. A4

Dabei hängt die **Bedingung vom Zustand des gerade simulierenden Modus A ab**. Tritt die Bedingung ein, so wird der Modus B aktiviert und mit B_i initialisiert. Diese Initialisierung ist eine **beliebige Initialisierung** und kann vom einfachen Gleichsetzten von Variablen bis hin zu komplizierten Funktionen alles enthalten. In allen folgenden Beispielen wird der Index c verwendet, um eine Bedingung zu kennzeichnen, und der Index i, um eine Initialisierung zu kennzeichnen. A5 A6

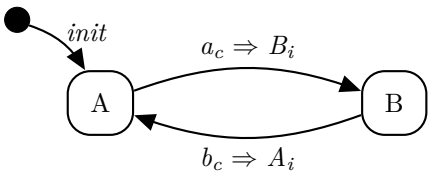


Abb. 5.1.: Zwei Modi mit eindeutigen Bedingungen

Nicht eindeutige Bedingungen

Im Beispiel aus Abbildung 5.2 ist ein weiterer Modus C hinzugekommen. In diesen Modus kann aus B gewechselt werden. Dabei ist ebenfalls die Bedingung b_c gewählt. Dies führt zu Problemen, da beim Eintreten der Bedingung b_c nicht eindeutig ist, in welchen Modus gewechselt werden soll.

A7 Um deterministisches Verhalten zu erreichen, müssen die **Wechselbedingungen** der Transitionen, die aus einem Modus hinausführen, **eindeutig** sein.

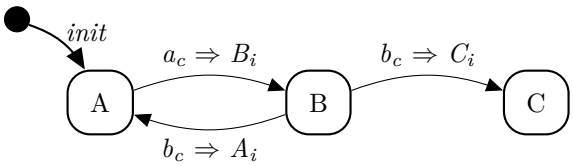


Abb. 5.2.: Drei Modi mit nicht eindeutigen Bedingungen

Zwei Transitionen zwischen zwei Modi

Es kann notwendig sein zwei Transitionen von einem Modus A nach B vorzusehen, wie in Abbildung 5.3 dargestellt. Die Wechselbedingungen müssen dafür unterschiedlich sein. Die Wechselbedingungen treten bei unterschiedlichen Zuständen des Modus ein, wodurch unterschiedliche Initialisierungen für den Modus B notwendig sein können. Daher sollte die

A8 **Initialisierung** bei einem Moduswechsel immer an die **Transition geknüpft** sein und nicht an den Modus. Sind die Initialisierungen identisch, ist nur eine Transition notwendig, welche die Wechselbedingungen disjunkt verknüpft.

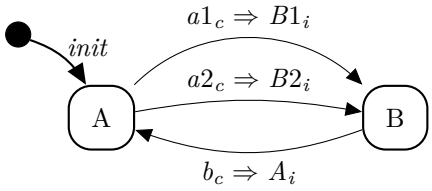


Abb. 5.3.: Mehrere Transitionen zwischen zwei Modi

Unterschiedliche Initialisierungen

Abbildung 5.4 zeigt ein Beispiel, in dem von zwei unterschiedlichen Modi (A und C) in einen Modus B gewechselt werden kann. Dabei kann die Initialisierung unterschiedlich sein (dargestellt durch $B1_i, B2_i$). Beispielsweise sind die Variablen, die zur Initialisierung von B benötigt werden, in A und C unterschiedlich benannt, oder die Informationen sind anders dargestellt. Die zu initialisierenden Variablen können die gleichen sein. Um die Modellierung zu vereinfachen, ist es sinnvoll in einem Modus die **verschiedenen Varianten zur Initialisierung anzugeben**. A9

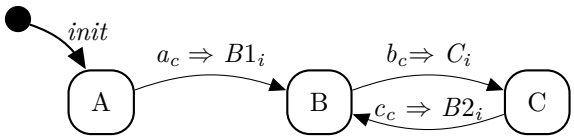


Abb. 5.4.: Mehrere Transitionen in einen Modus

5.1.2. Strukturänderungen in Komponenten

Modelle sind oft modular und hierarchisch aufgebaut. Sie bestehen also aus Komponenten, die miteinander interagieren, und Komponenten können wiederum Komponenten enthalten. Komponenten sollen nun Modi und Transitionen enthalten, wodurch hierarchische und modulare strukturvariable Modelle entstehen. Einer der Vorteile ist die hohe Flexibilität des Modells durch die unterschiedlichen Kombinationsmöglichkeiten der Modi während der Simulation. Ebenso können Komponenten mit ihren Modi spezifiziert und in Modellbibliotheken abgelegt werden. Die beschriebenen Komponenten der Bibliothek können dann in unterschiedlichen Modellen instanziiert werden. Während dieser Instanziierung können Defaultwerte der Komponente, wie beispielsweise Parameterwerte, angepasst werden. Dieses Vorgehen ist in Modelica durch die Objektorientierung vorgesehen, während in Simulink parametrierbare Subsysteme in Bibliotheken abgelegt werden können.

Eine Komponente K kann somit aus mehreren Modi bestehen. Jeder Modus ist dabei selbst eine Komponente und repräsentiert die Komponente K für den Zeitraum, in dem der Modus aktiv ist. Die Komponente K ist mit anderen Komponenten über ihre Schnittstellen verbunden.

Abbildung 5.5 zeigt ein Modell, welches zwei Instanzen von Komponenten ($a : A$, $b : B$) ohne Modi enthält. Der Übersicht halber werden die Instanzen nur als A, B dargestellt und nicht benannt.

A10 Die **Komponenten sind miteinander über ihre Schnittstellen** verbunden. Die Komponenten sind mit einer Doppellinie umrandet und deren Schnittstellen sind durch Kreise an der Doppellinie dargestellt. Unterschiedliche Muster der Kreise deuten auf verschiedene Schnittstellen hin. Der Zusammenhang der Komponenten ist dabei beliebig, d.h. es kann sich um einfaches Gleichsetzten von Variablen oder auch um komplexere Zusammenhänge handeln, die durch Gleichungen beschrieben werden.

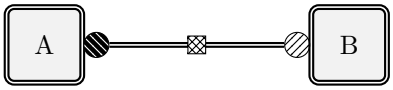


Abb. 5.5.: Modell bestehend aus zwei Komponenten, welche über eine Verbindung miteinander gekoppelt sind

Die Verbindungsbeschreibung ist abstrakt dargestellt, da sie nicht auf eine bestimmte Sprache beschränkt ist und für sämtliche Sprachen verwendet werden kann.

In Modelica gibt es zum Beispiel die *Connectoren*, während Matlab/Simulink ein einfaches Übertragen von Werten erlaubt.

In den hier betrachteten Verbindungen können beliebige Funktionen beschrieben sein, welche die Schnittstellen in Beziehung setzten. Aus der

A11 **Verbindungsbeschreibung resultieren die Verbindungsgleichungen**, welche dem Gleichungssystem des Modells hinzugefügt werden. Diese Verbindungsbeschreibung wird als Rechteck mit Muster auf der Verbindungslinie dargestellt. Abbildung 5.6 zeigt ein Beispiel, in dem mehrere Komponenten über eine Verbindungsbeschreibung gekoppelt sind.

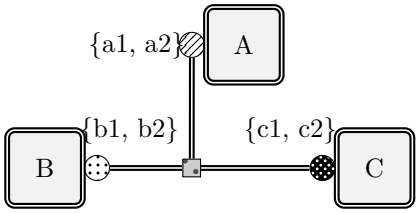


Abb. 5.6.: Modell bestehend aus drei Komponenten, welche über eine Verbindungsbeschreibung gekoppelt sind

Jede der Komponenten hat eine Schnittstelle, wobei jede Schnittstelle zwei Variablen enthält. Die Verbindungsbeschreibung könnte die Schnittstellen beispielsweise wie folgt verbinden:

- $A.a1 = B.b1$
- $A.a2 + B.b2 + C.c2 = 0$
- $B.b1 = C.c1 \cdot 10$

Zwei Modi mit gleicher Schnittstelle in einer Komponente

Abbildung 5.7 zeigt ein Modell mit zwei Komponenten. Dabei hat die Komponente A zwei Modi. Zwischen diesen Modi kann das Modell abhängig von den gewählten Bedingungen wechseln. Die Instanz von A wird im Modell dabei jeweils vom gerade aktiven Modus repräsentiert. Auch die Transitionen im Modell sind Objekte, werden aber im weiteren Verlauf nicht benannt, damit die Notation übersichtlich bleibt.

Von besonderer Relevanz sind die Schnittstellen, da diese so gewählt sein müssen, dass das Gesamtmodell lauffähig ist. Die Schnittstellen der Modi in diesem Beispiel sind identisch.

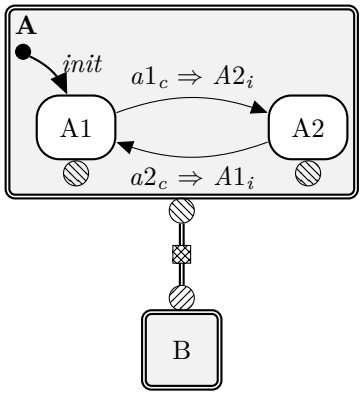


Abb. 5.7.: Modell mit zwei Komponenten, wobei eine Komponente zwei Modi hat

Während der Simulation muss zu jedem Zeitpunkt ein gültiges und lösbares Gleichungssystem bestehen. Daher muss der numerische Solver für dieses Beispiel mit zwei Modi arbeiten ($A1 - B$ und $A2 - B$). Das Modell kann bzw. muss vom Compiler so umgeformt werden, dass sich die
A12 **Wechsel auf oberster Modellebene befinden**.

Diese Umformung kann entweder vor der Simulation stattfinden oder erst dann, wenn der jeweilige Modus benötigt wird. Abbildung 5.8 zeigt das umgeformte Modell aus Abbildung 5.7, in dem die Moduswechsel nun auf der obersten Modellebene definiert sind. Dies wird ab jetzt ausfaktorisiertes Modell genannt.

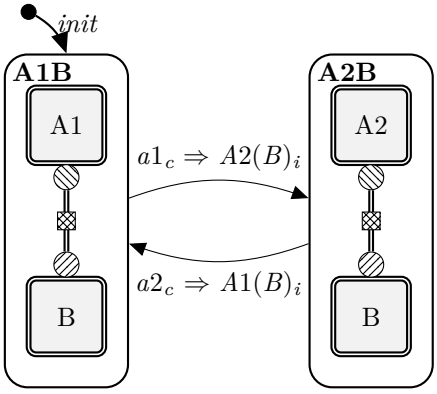


Abb. 5.8.: Ausfaktorisiertes Modell aus Abbildung 5.7

Modi mit gleichen Schnittstellen in Komponenten sind nicht zwangsläufig kompatibel mit anderen Komponenten. In [9] wird die Lösbarkeit von objektorientierten Modellen untersucht und ein *Constrained Delta* C_Δ angegeben. Dieses gibt für eine Modellkomponente die Differenz der Anzahl von Gleichungen zu Variablen an. Für ein simulierbares Modell muss diese Differenz in heutigen Simulationswerkzeugen null sein. Dabei ergibt sich das C_Δ eines Modells aus der Kopplung der einzelnen Komponenten. Die einzelne Komponenten können ein C_Δ unterschiedlich von null haben. Modi innerhalb einer Komponente sollten jedoch das gleiche C_Δ aufweisen, damit die Modi im Kontext der Komponente verwendet werden können. Daraus lässt sich ebenso die Forderung ableiten, dass sich die Anzahl der Ein- und Ausgangsgrößen der Modi in einer Komponente nicht unterscheiden sollte.

In akausalen Sprachen wie Modelica sollte das C_Δ in den Modi beachtet werden. In signalflussorientierten Sprachen wie Simulink ist es ausrei-

chend, die Ein- und Ausgangsgrößen zu betrachten, da diese direkt spezifiziert sind. Sind die Schnittstellen der Modi identisch, so sind sie auch kompatibel zum Kontext.

Eine Ausnahme der Regel mit dem identischen C_Δ der Modi besteht, wenn gleichzeitig ein Moduswechsel in einer anderen Komponente ausgelöst und dadurch wieder $C_\Delta = 0$ im Modell erreicht wird (vgl. Abschnitt 5.2.1 (Mehrere Wechsel einleiten)).

Die Idee des modularen Aufbaus ist, das Verhalten zu kapseln. **Eine lokale Transition sollte daher nicht den Kontext der Komponente initialisieren können.** Jedoch ist es bei einem Moduswechsel notwendig, auch B zu initialisieren. Aus diesem Grund wird die Initialisierung von B beim ausfaktorisierten Modell angedeutet. Diese Problematik wird in 5.2.3 detailliert behandelt. A13

Zwei Modi mit unterschiedlichen Schnittstellen in einer Komponente

In Abbildung 5.9 wird ein ähnliches Beispiel betrachtet, nur diesmal sind die Schnittstellen der Modi unterschiedlich. Sobald ein Wechsel vom Modus $A1$ nach $A2$ eintritt, muss die Verbindung zur Komponente B geändert werden.

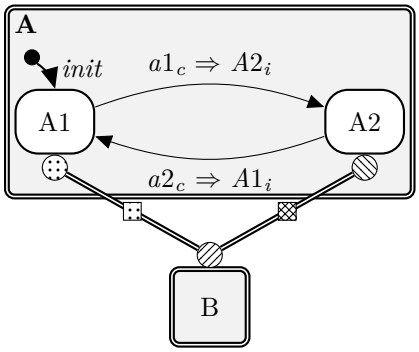


Abb. 5.9.: Modell mit zwei Modi in einer Komponente mit unterschiedlichen Schnittstellen

Es ergeben sich zwei Möglichkeiten, mit dieser Problematik umzugehen:

1. Die Schnittstellen bleiben unterschiedlich und die Gleichungen auf der oberen Modellebene müssen bei einem Moduswechsel angepasst werden.

2. Die Schnittstellen der Modi werden angepasst, damit sie über die gleiche Verbindungsbeschreibung mit B verbunden werden können.

Im ersten Fall könnte das Gesamtmodell ausfaktorisiert aussehen wie in Abbildung 5.10 dargestellt.

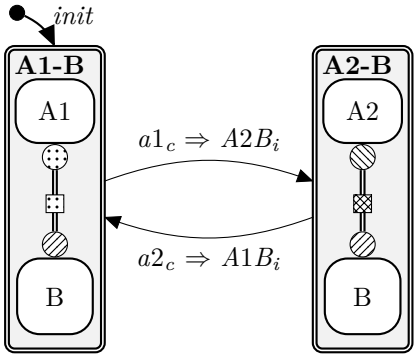


Abb. 5.10.: Ausfaktorisiertes Modell aus Abbildung 5.9

Dies sieht nach einer einfachen Variante aus, birgt jedoch einige Schwierigkeiten. Wird das Modell in Modelica ähnlichem Pseudocode beschrieben, indem mit if/else Anweisungen Moduswechsel beschrieben werden, könnte das Modell wie in Listing 5.1 dargestellt aussehen. Eine solche Beschreibung ist im heutigen Modelica so noch nicht umgesetzt und sie dient an dieser Stelle nur der Illustration.

```
1  model ModusWechselUnterschiedlicheSchnittstellen
2     Int aktModus(start =1);
3     if aktModus == 1 then A1 a; else  A2 a; end;
4     B b;
5   equations
6    if aktModus == 1 then
7      f1(a.interface, b.interface);
8    else
9      f2(a.interface, b.interface);
10   end
11   when aktModus == 1 and a1c then
12     aktModus = 2;
13     A2Bi;
14   end;
15   when aktModus == 2 and a2c then
16     aktModus = 1;
17     A1Bi;
18   end;
19 end ModusWechselUnterschiedlicheSchnittstellen;
```

Listing 5.1: Pseudocode zum Beschreiben eines Moduswechsels auf Komponentenebene mit unterschiedlichen Schnittstellen

In der Variable aus Zeile 2 wird der aktive Modus gespeichert. Abhängig vom aktiven Modus wird in Zeile 3 für die Variable a der Typ des aktuell aktiven Modus deklariert. Ab Zeile 6 werden die Verbindungsbeschreibungen angepasst, welche durch die Funktionen $f1$ und $f2$ repräsentiert werden. Die unterschiedlichen Verbindungsbeschreibungen resultieren aus den unterschiedlichen Schnittstellen der Komponenten $A1$ und $A2$.

Ab Zeile 11 wird der Moduswechsel abstrakt dargestellt, indem der aktive Modus geändert wird und eine Initialisierung stattfindet. Wie eine solche Initialisierung konkret aussieht, wird in Abschnitt 5.2.2 behandelt.

Die Moduswechsel können in einem solchen Fall nicht lokal beschrieben werden, da Änderungen auf einer höheren Modellebene notwendig sind.

In der zweiten Variante werden die Schnittstellen innerhalb von A **angepasst**, damit beide Modi die **gleiche Schnittstelle** haben. A14

Dazu wird jeder Modus in eine neue Komponente eingebettet, welche die neuen Schnittstellen hat. Die Schnittstellen des alten Modus und die Schnittstellen des neuen Modus werden über die alte Verbindungsbeschreibung verbunden. Beide Modi in A haben daraufhin genau die Schnittstellen, die B benötigt. Es muss damit auf der obersten Modellebene nur ein Gleichsetzen der Variablen in der Verbindungsbeschreibung beschrieben werden.

Wird solch ein Vorgehen in Modelica durchgeführt, kann eine neue Komponente $A1\_a$ mit den gewünschten Schnittstellen erstellt werden. Diese Komponente wird zum neuen Modus und enthält den alten Modus $A1$ als Komponente (Komposition); $A1$ ist nun selbst kein Modus mehr, sondern eine Komponente innerhalb eines Modus. Die alten Schnittstellen von $A1$ werden durch die alten Verbindungsbeschreibungen (von $A1$ nach B) auf die neuen Schnittstellen von $A1\_a$ abgebildet. Auf die gleiche Weise wird der Modus $A2\_a$ erstellt.

Es entsteht ein Modell, wie es in Abbildung 5.11 gezeigt ist. Dieses sieht auf den ersten Blick komplizierter aus als das Modell aus Abbildung 5.10. In diesem Fall finden die Moduswechsel jedoch lokal in der Komponente statt. In die Komponente können weitere Modi integriert werden, ohne das globale Modell anpassen zu müssen.

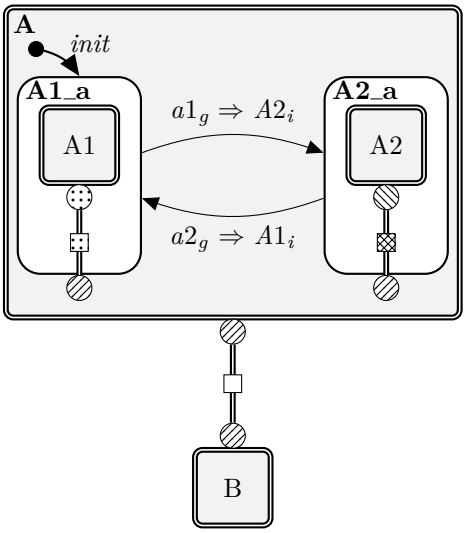


Abb. 5.11.: Modell mit zwei Modi in einer Komponente mit unterschiedlichen Schnittstellen, die Schnittstellen wurden angepasst

Ungleiche Schnittstellen in Komponenten

Das oben vorgestellte Vorgehen zum Anpassen von Schnittstellen dient dazu, Moduswechsel lokal zu halten. Haben mehrere Komponenten interne Modi, können zeitgleiche Wechsel in verschiedenen Komponenten auftreten. Wie ein zeitgleicher Wechsel geschieht, wird in Abschnitt 5.2.1 (Mehrere Wechsel einleiten) erläutert. Treten nur gleichzeitige Wechsel ein, sodass nur die Kombinationen $A1 - B1$ und $A2 - B2$ möglich sind,
A15 müssen die **Schnittstellen nicht angepasst werden**, sofern die Verbindungsbeschreibung die jeweiligen Schnittstellen in Beziehung setzen kann. Dies ist beispielsweise möglich, wenn in der Verbindungsbeschreibung alle Werte einer Schnittstelle zur anderen übergeben werden, unabhängig von deren Länge.

Dies gilt nur, solange die Moduswechsel zeitgleich eintreten und die Konstellation $A1 - B2$ nicht eintreten kann. Träte sie ein, wäre dies eine ungültige Konstellation für das Modell.

Bei solch einem Modell sollte es in Betracht gezogen werden, das Modell bereits beim Erstellen anders zu beschreiben und die Modi $A1 - B1$ und $A2 - B2$ direkt vorzusehen.

Es ist somit nicht notwendig gleiche Schnittstellen der Modi in Komponenten zu fordern. Gleiche Schnittstellen führen auch nicht unbedingt zur

Kompatibilität von Komponenten. Die Kompatibilität hängt vom Kontext, den gewählten Wechselbedingungen, der Art der Verbindungen und der Kausalität ab. Gleiche Schnittstellen vereinfachen jedoch die Überprüfung, da zumindest die Verbindungsbeschreibung wiederverwendet werden kann. Ob diese Verbindungsbeschreibung zu einem konsistenten Gesamtzustand führt, ist damit aber noch nicht sichergestellt.

Hinzufügen/Entfernen von Komponenten

Sollen Komponenten aus einem Modell entfernt werden oder hinzukommen, ist das auf lokaler Ebene der Komponente kein Moduswechsel. Sei dazu Abbildung 5.7 noch einmal betrachtet und davon ausgegangen, dass die Komponente B entfernt werden kann. Dies ist innerhalb von B nicht darstellbar, da eine Transition nicht definiert werden kann, wenn sie nicht zu einem Modus führt.

Eine solche Änderung muss eine Ebene darüber definiert werden, siehe dazu Abbildung 5.12.

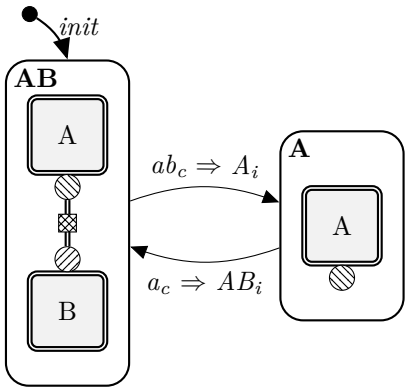


Abb. 5.12.: Modell, bei dem eine Komponente entfernt wird

Die Betrachtung der Schnittstellen ist an dieser Stelle besonders relevant. Wird die Komponente B entfernt, so ist die Schnittstelle von A nicht mehr verbunden. Benötigt die Komponente A jedoch Eingangsdaten, so ist das Modell ohne B nicht mehr simulierbar. Enthält die Schnittstelle von A nur Ausgabedaten, so wäre das Modell noch simulierbar, die Verbindung zu B muss trotzdem entfernt werden.

Sollen Komponenten hinzukommen/entfernt werden, so muss eine Änderung der Verbindungen zu den anderen Komponenten vorgesehen werden. Daher muss der Moduswechsel mindestens auf der Ebene definiert sein, in der die Verbindungen definiert sind.

Soll der Wechsel lokal definiert werden, sollte die zu entfernende Komponente vereinfacht abgebildet werden. Dies hat zusätzlich den Vorteil, dass der Modus noch Daten über die Schnittstelle bereitstellen kann, sofern dies notwendig ist. Nimmt er nur Ausgangsdaten der anderen Komponenten entgegen, so ist es ein reiner *Dummy*-Modus. Das Modell würde dann analog zu Abbildung 5.7 aussehen. Kommt die Komponente wieder hinzu, so wird wieder zum eigentlichen Modus gewechselt.

Kommen neue Komponenten hinzu, muss zusätzlich entschieden werden, wie diese initialisiert werden. Sie können entweder mit benutzerdefinierten Daten initialisiert werden oder auch durch vergangene Simulationsdaten, siehe hierzu auch Beispiel 5.2.

Ungültige Konstellationen von Modi

A16 Ein **Moduswechsel innerhalb einer Komponente sollte einen Moduswechsel in einer anderen Komponente hervorrufen können**, falls andernfalls eine Inkonsistenz entsteht. Beispiel 5.1 erläutert diesen Fall.

Beispiel 5.1 (Flugzeug): *Sei ein Flugzeugmodell betrachtet, welches aus dem Flugkörper mit Antrieb und einer Komponente, welche den Widerstand bestimmt (Reibung, Luftwiderstand, etc.), besteht. Zu Beginn befindet sich das Flugzeug auf dem Rollfeld und kann sich nur in x- und y-Richtung bewegen. Hebt das Flugzeug vom Boden ab, hat der Flugkörper zusätzlich eine Flughöhe z. Diese zusätzliche Variable kann durch einen Moduswechsel hinzukommen. An dieser Stelle ist es sinnvoll, die Reibung der Reifen am Boden nicht mehr zu betrachten. Ein fliegendes Objekt mit Bodenreibung ist eine ungültige Konstellation.*

In einem Modus wäre es gut anzugeben, welche Modi aus unterschiedlichen Komponenten zueinander kompatibel sind. Diese Kompatibilität kann logische/physikalische Gründe haben.

Wie bekannt, können Inkompatibilitäten ebenso durch unterschiedliche Schnittstellen oder Kausalitäten hervorgerufen werden. Je komplexer die Modelle werden, desto schwerer ist es, die Kompatibilität sicher zu stellen.

Zustandsexplosion

Im Beispiel aus Abbildung 5.13 sind verschiedene Modus-Kombinationen möglich, abhängig von den Schalterstellungen innerhalb des elektrischen Schaltkreises. Bei diesem Schaltkreis mit drei Schaltern ergeben sich acht mögliche Kombinationen ($2^{(n=3)}$). Wird dieses Beispiel um weitere Elemente erweitert, ergeben sich exponentiell wachsende Kombinationsmöglichkeiten. Dies wird in der Statechart-Semantik als Zustandsexplosion bezeichnet.

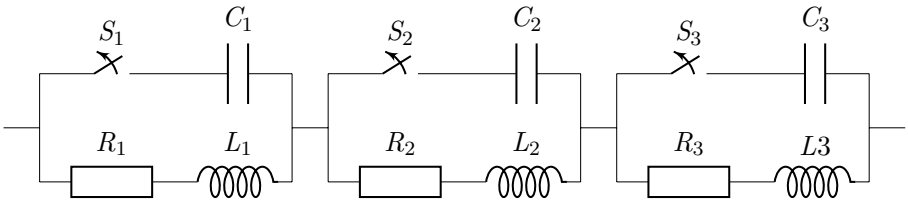


Abb. 5.13.: Elektrischer Schaltkreis mit mehreren Schaltern

Ähnliches tritt auf, wenn ein Modell aus mehreren Komponenten besteht und jede der Komponenten mehrere Modi enthält. Besonders in solchen Fällen müssen die lokalen Modi und deren Transitionen so gewählt sein, dass keine ungültigen Kombinationen auftreten. Auch besteht die Gefahr, viele schnell hintereinander auftretende Moduswechsel zu erzeugen (siehe auch Abschnitt 5.2.1).

5.2. Auslegung der Transitionen

Eine Transition beschreibt den Wechsel von einem Modus zu einem anderen. Sie gibt die Bedingung an, wann in einen anderen Modus und in welchen gewechselt werden soll.

Eine Transition ist auf der Modellebene definiert, in der auch die Modi definiert sind. Da eine Transition immer zwei Modi in Relation setzt, kann die Transition auf die Daten dieser beiden Modi zugreifen. Werden Modi nur auf der obersten Modellebene betrachtet, kann die Transition alle Daten der beiden Modi verwenden.

Ein Ziel bei der Betrachtung von Modi in Komponenten ist die lokale Kapselung. Diese Kapselung ermöglicht es, bereits definierte Komponenten in verschiedenen Kontexten einzusetzen.

Eine Voraussetzung für diese Wiederverwendung ist, dass die Komponenten nur über ihre Schnittstellen mit anderen Komponenten interagieren. Durch diese lokale Kapselung und das Ziel der Wiederverwendung der
A17 Komponenten haben die Modi und auch **Transitionen in Komponenten einen lokalen Scope**. Eine Transition innerhalb einer Komponente kann somit nur Daten aus den beiden lokalen Modi verwenden, die durch sie verbunden werden.

Aktionen in einer Transition beschreiben, was bei einem Moduswechsel passieren muss. In den meisten Fällen ist dies die Initialisierung. Während einer Aktion kann aber zusätzlich ein erneutes Kompilieren, ein Zurücksetzen in die Vergangenheit oder ein erneutes Starten des alten Modus mit anderen Parametern gefordert sein.

Die verschiedenen Aspekte einer Transition werden im Folgenden näher betrachtet. Dabei wird davon ausgegangen, dass sich der Modellierer für eine Menge von Modi entschieden hat.

5.2.1. Auslegung der Wechselbedingungen

Die Wechselbedingungen als Teil der Transition sind wichtig, da sie die Moduswechsel einleiten. Im Folgenden wird erläutert, was bei deren Auslegung zu beachten ist und welche Schwierigkeiten auftreten können.

Eindeutigkeit

Wie schon aus den vorangegangenen Beispielen bekannt, sollten die Wechselbedingungen innerhalb eines Modus eindeutig sein, damit sich das Modell deterministisch verhält. Dabei befindet sich eine Transition entweder auf der obersten Modellebene oder löst Moduswechsel innerhalb einer Komponente aus. Die Bedingungen und Aktionen können dabei Daten aus dem Scope der Transition verwenden.

Chattering vermeiden

Ein ungewolltes Hin- und Herschalten zwischen Modi sollte durch die Wahl der Bedingungen vermieden werden. Beispielsweise soll eine Transition bei einer Bedingung $T > 20$ schalten und bei $T < 20$ zurückschalten. Dies kann zu vielen unnötigen Schaltvorgängen führen, wenn sich T in diesem Größenbereich aufhält. Es sollte eine Hysterese vorgesehen werden, um solche unnötigen Schaltvorgänge zu vermeiden.

Ein weiteres Problem beim Chattering oder auch beim Eintreten mehrerer Wechsel ist die Initialisierung. Jede Initialwertberechnung führt zu Fehlern, da auch sie eine Abstraktion darstellt. Daher sollten Moduswechsel nicht direkt hintereinander stattfinden, ohne die eigentlichem Modi zu simulieren. Die Fehler können sich im schlimmsten Fall akkumulieren und zu falschen Simulationsergebnissen führen.

Es sollte bei der Auslegung der Wechselbedingungen darauf geachtet werden, dass die Simulation eines Modus zumindest den Einschwingvorgang über andauert, bevor ein neues Ereignis eintritt.

Treten zwei Moduswechsel sehr dicht hintereinander ein, kann es sinnvoll sein, beim zweiten Wechsel in die Vergangenheit zum Zeitpunkt des ersten Moduswechsels zu gehen. So kann dort direkt die Simulation des korrekten Modus gestartet werden.

Welches Vorgehen das richtige ist, ist von der jeweiligen Aufgabenstellung und den gewählten Modi abhängig und kann nicht pauschal geklärt werden. Beim Entwurf des Modells sollte die Gefahr des Chatterings jedoch nicht vernachlässigt und entsprechende Vorkehrungen getroffen werden.

Komplexität der Wechselbedingung

Die gewählten **Wechselbedingungen sind abhängig vom Zustand des aktuellen Modus** und können **beliebig komplex** sein. A18

Die Wechselbedingung muss sich aus den vorhandenen Variablen und Parametern des aktuellen Modus angeben lassen. In ihr dürfen nur Daten abgefragt werden, welche im Scope der Transition liegen.

Die Wechselbedingungen können beliebig verknüpft werden, so kann beispielsweise $(x > 5 \land y < 3) \lor z \geq 10$ gefordert werden.

Zeitpunkt des Wechsels

Wird zwischen einem sehr detaillierten und einem weniger detaillierten Modell gewechselt, kann es sinnvoll sein, nicht so früh wie möglich zu dem weniger detaillierten Modell zu wechseln bzw. nicht so spät wie möglich zum detaillierten. Teilweise sollte das detaillierte Modell zum Zweck der Genauigkeit länger verwendet werden.

Ein guter Zeitpunkt, einen Moduswechsel einzuleiten, ist, wenn sich das Modell in einem gutartigen Zustand aufhält, also einem Zustand, in dem der Modus noch für die Aufgabenstellung valide ist und sich sein Zustand möglichst wenig ändert.

Ändern sich die Daten wenig, so können neue Initialwerte gut berechnet werden. Bei dynamischem Verhalten ist es schwieriger, Startwerte für einen neuen Modus zu finden.

Wenn der Modellierer keinen geeigneten Zeitpunkt beschreiben kann, sollte mit Hilfe von Experimenten ein geeigneter Zeitpunkt gesucht werden.

Mehrere Wechsel einleiten

A19 Haben mehrere Komponenten Modi, so können **gleichzeitige Wechsel** in verschiedenen Komponenten auftreten. Dies ist zum Beispiel notwendig, wenn nur manche Modi kompatibel sind.

Erkennt das Simulationswerkzeug den gleichzeitigen Moduswechsel, werden beide Modi lokal initialisiert. Dies kann zu einem inkonsistenten Gesamtzustand führen (siehe hierzu auch 5.2.3).

Eventuell ist das Simulationswerkzeug nicht in der Lage, beide Moduswechsel zu detektieren. Dadurch findet ein Wechsel statt, der zu einem fehlerhaften Modellzustand führt.

Soll sichergestellt werden, dass zum einen beide Transitionen eintreten und zum anderen die Initialisierungen zusammenpassen, sollte eine
A20 **globale Transition definiert** werden, welche den Moduswechsel auf dem ausfaktorisierten Modell beschreibt. Die lokalen Transitionen sollten entfernt werden.

Ein anderer Fall tritt ein, wenn die Wechselbedingungen in den verschiedenen Komponenten nicht komplett identisch sind, jedoch sehr dicht aneinander liegen. In diesem Fall wird es sich kaum vermeiden lassen, mehrere Wechsel dicht hintereinander eintreten zu lassen. Wird dies bei der Simulation festgestellt, sollten die Wechselbedingungen noch einmal betrachtet und gegebenenfalls gleichzeitig eingeleitet werden.

5.2.2. Allgemeine Initialisierung

Wie aus Abschnitt 2.3 bekannt, ist die Initialisierung bereits bei der konventionellen Simulation nicht trivial. Das Finden konsistenter Startwerte ist nicht einfach und die Startwerte beeinflussen das Verhalten des Modells oft essenziell. Schlecht gewählte Startwerte können ungenaue Simulationsergebnisse oder instabiles Verhalten nach sich ziehen. Deshalb ist besonders bei der Betrachtung von Moduswechseln die Initialisierung wichtig, da nicht mehr nur am Anfang konsistente Startwerte notwendig sind, sondern bei jedem Moduswechsel.

Initialisieren eines Modus

Findet ein Moduswechsel statt, muss angegeben werden, wie der neue Modus anhand von bekannten Daten initialisiert wird. Für ein Modell (damit auch für einen Modus) gibt es meist nicht nur genau eine Möglichkeit, eine konsistente und vollständige Initialisierung zu erreichen. Oft gibt es **mehrere Varianten, ein Modell zu initialisieren**. A21

Zur Initialisierung bei einem Moduswechsel muss ein Set von Variablen initialisiert werden. Dazu können Werte aus alten Simulationen **übernommen werden oder Berechnungen notwendig sein**. Berechnungen sind erforderlich, wenn ein Übergeben von Daten nicht ausreicht, da Mittelwerte gebildet oder Daten ineinander umgerechnet werden müssen (vgl. Abschnitt 4.1). A22

Oft ist eine automatische Initialisierung nicht möglich, da die Variablen in den verschiedenen Modi unterschiedlich benannt sind. Selbst wenn alle Variablen gleich benannt sind, unterscheiden sich die Gleichungssysteme der Modi, wodurch nicht immer ein identischer Zustand der Variablen vorausgesetzt werden kann.

Sei angenommen, zwei Modi haben die Variablen x und y, wobei der erste Modus die Gleichung $x = y$ enthält und der zweite Modus die Gleichung $x = 1,1 \cdot y$. Nur für $x = 0$ und $y = 0$ können die beiden Variablen den gleichen Zustand haben. Für alle anderen Werte muss ein Anpassen der Startwerte vorgenommen werden.

Einige Werte werden einen Sprung machen, auch wenn dies nicht immer mit der Realität übereinstimmt. An diesem Punkt muss entschieden werden, ob ein solcher Sprung erlaubt ist oder die Startwerte angepasst werden müssen. In den meisten Fällen sollten Sprünge in Zustandsgrößen vermieden werden, da diese durch Differenzialgleichungen beschrieben werden und daher nicht sprungfähig sind. Werte von anderen Variablen dürfen eventuell Sprünge aufweisen, wenn sie für die jeweilige Untersuchung weniger relevant sind.

Modelle parallel simulieren

Soll ein möglichst stetiger Übergang zwischen zwei Modi erreicht werden, können beide Modi kurzzeitig parallel, mit unterschiedlicher Gewichtung der relevanten Variablen, simuliert werden. Dies entspricht einem dritten Modus. Dabei wird die Gewichtung mit der Zeit angepasst. Sobald der neue Modus 100% der Gewichtung hat, wird der alte Modus entfernt.

Dies setzt voraus, dass die Modi sich genug ähneln, was für Detaillierungsgradänderungen meist gegeben ist.

Dies entspricht der Grundidee der Homotopie, in der ein Modell in ein anderes zur Initialisierung überführt wird [76]. Nachteilig können sich die zwei Moduswechsel und drei Modi auswirken, da das Modell komplizierter wird.

Zwischenmodus vorsehen

Ist der Übergang zwischen zwei Modi kompliziert und nicht mit einer Berechnung an genau einem Zeitpunkt möglich, kann ein Zwischenmodus vorgesehen werden, welcher den Übergang beschreibt. Dieser beschreibt über einen definierten Zeitraum den Übergang von einem Modus zu einem anderen.

Mit diesem Vorgehen können Modi ineinander überführt werden. Der Moduswechsel wird also mit einem zusätzlichen Modus beschrieben. Ein solches Vorgehen kann bei Verhaltensänderungen eingesetzt werden, wenn detailliertere Moduswechsel erforderlich sind.

Nachteilig an diesem Vorgehen ist, dass zwei Moduswechsel stattfinden, welche Berechnungsungenauigkeiten mit sich bringen können. Auch müssen drei Modi beschrieben werden, was einen Mehraufwand darstellt. Ein Zwischenmodus sollte daher nur vorgesehen werden, wenn dadurch die Moduswechsel und auch die Ergebnisse der gesamten Simulation genauer werden.

Zurücksetzen in die Vergangenheit

A23 Es kann wünschenswert sein, **während eines Moduswechsels die Simulation des nächsten Modus in der Vergangenheit zu starten** (siehe ebenfalls Abschnitt 4.4), beispielsweise, wenn dort noch ein stationärer Zustand herrschte.

Dabei muss angegeben werden, wie weit in die Vergangenheit gegangen werden muss, und die vergangenen Daten müssen zur Verfügung stehen.

Der genaue Wechselzeitpunkt ist eventuell nicht in den numerischen Daten enthalten, wodurch eine Interpolation oder Näherung erforderlich sein kann.

Nutzen von vergangenen Werten

Hängt die Initialisierung nicht nur vom alten Modus ab, sondern von dem **Zustand, in dem sich der Modus bei einer vorherigen Simulation** befand, so muss auf diese Daten zugegriffen werden können. Um dies zu verdeutlichen, dient das Beispiel 5.2. A24

Beispiel 5.2 (Rohrleitungen und Behälter - Datenspeicherung):
Es soll ein Rohrleitungssystem modelliert werden. Dabei gibt es ein Rohrleitungssystem und mehrere Ventile, die zu Behältern führen. Die Behälter sollen nur simuliert werden, wenn diese befüllt oder entleert werden. Die Befüllung bzw. Entleerung findet über Ventile statt.

Sei ein Behälter betrachtet, welcher zu Beginn der Simulation befüllt wird. Nachdem der Behälter befüllt ist, wird das Einlassventil geschlossen. Der Füllstand des Behälters ist dann konstant und muss nicht mehr berechnet werden.

Nach einer Zeit t_{aus} wird für eben diesen Behälter das Auslassventil geöffnet und der Behälter entleert sich. Der Füllstand soll nun wieder berechnet werden. Die Füllmenge im Behälter muss dazu initialisiert werden. Diese Information ist nicht zwangsläufig im Rohrmodell enthalten.

Es gibt drei Möglichkeiten, mit dieser Problematik umzugehen:

- Die Werte, welche konstant bleiben sollen (hier Füllmenge des Behälters), werden durch eine zusätzliche Variable im Modell mitgeführt, welche deren Zustand speichert. Bei großen Modellen mit vielen Variablen ist dieses Verfahren nachteilig, da unnötige Daten mitgeführt werden. Der Modellierer muss sicherstellen, dass diese Werte tatsächlich konstant bleiben, da es sonst zu falschen Simulationsergebnissen kommen kann.
- Alle vergangenen Simulationsdaten werden gespeichert und können bei einem Moduswechsel ausgelesen und verwendet werden. Auch hier sollte sichergestellt werden, dass die Annahme einer Konstanten gerechtfertigt ist.
- Entspricht der neue Zustand nicht exakt dem alten Zustand der Komponente, muss entschieden werden, wie der neue Startwert bestimmt wird:
  - Die Komponente wird vereinfacht im Modell mitberechnet.
  - Während der Initialisierung wird ein Zusammenhang angegeben, mit dem die neuen Daten berechnet werden.

Die Initialisierung hinzukommender Komponenten ist keine triviale Aufgabe und erfordert hohe Flexibilität vom Werkzeug und ein hohes Maß an Wissen vom Modellierer für sein Modell. Nur so kann der Modellierer entscheiden, welche der oben genannten Varianten angewendet werden soll.

5.2.3. Initialisierung für Modi in Komponenten

Bei der Modellierung von Komponenten mit Modi und Transitionen ist die Idee, diese Komponenten in verschiedenen Kontexten zu instanziieren. Es kann jedoch vorkommen, dass die Komponente nicht immer in den gewünschten Kontext passt.

Im Verlauf der nächsten Abschnitte wird erläutert, wie dies zustande kommt und wie mit dieser Problematik umgegangen werden kann.

Inkonsistente Initialisierung

Bei einem Moduswechsel in einer Komponente kann es zu Inkonsistenzen oder zu unerwünschten Zuständen im Kontext der Komponente kommen. Am Beispiel 5.3 wird dieser Fall näher betrachtet.

Beispiel 5.3 (Rohrleitung mit Drosselstelle): *Betrachtet sei die Abbildung 5.14, in der eine Komponente mit Modi mit einer anderen Komponente verbunden ist. Dabei handelt es sich um ein stark vereinfachtes Modell. Die Rohrleitung ist einmal in 100 einzelne Teilstücke unterteilt und im anderen Modus in zehn (Gridänderung). Wenn die Zustände der vielen Teilstücke ausgeglichen sind, wird in den Modus mit weniger Rohrteilen gewechselt. Zur Initialisierung der Zustände der zehn Rohrteile wird für den Druck und die Temperatur jeweils der Mittelwert gebildet und die Masse der Rohrteile addiert.*

Wie Masse in die Rohrleitung gelangt, sei irrelevant. Auf einer Seite sei die Rohrleitung mit einer Drosselstelle verbunden. Die Schnittstelle besteht aus einem Massenstrom mflow, einem Druck p und einer Temperatur T. Der Massenstrom wird in der Drossel, abhängig von der Temperatur, dem Druck und einem Parameter P, berechnet. Der Druck und die Temperatur werden in der Rohrleitung bestimmt.

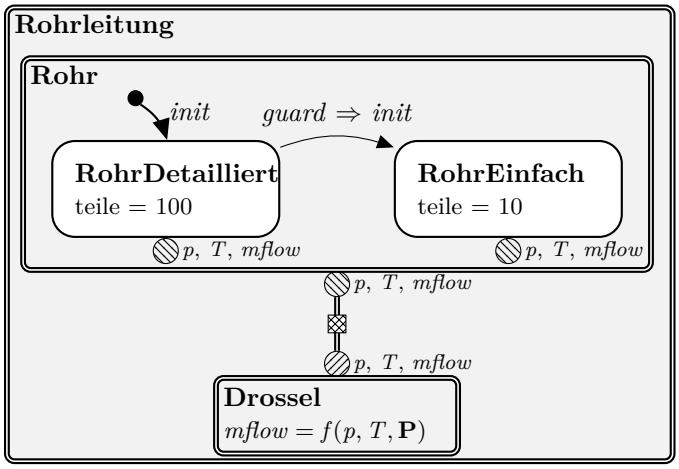


Abb. 5.14.: Rohrleitung mit zwei Modi, verbunden mit einer Drossel

Bei einem Moduswechsel im Rohrmodell können sich der Druck und die Temperatur am Ende der Rohrleitung durch die Berechnung der Startwerte ändern. Der Unterschied zum alten Wert ist dabei abhängig von der Wechselbedingung und den Zuständen der Rohrteile zum Wechselzeitpunkt.

In der zweiten Komponente (Drossel) sollen sich die Werte bei einem Wechsel nicht ändern. Bleibt der Wert des Parameters P identisch, so muss sich bei einer Änderung von T und p der Massenstrom ändern. Es stellt sich die Frage, ob dies das gewünschte Verhalten ist. Vielleicht wäre es an dieser Stelle besser, den Massenstrom konstant zu halten und dafür den Parameter P anzupassen, da dieser meist sowieso einer Abstraktion für das Durchflussverhalten entspricht.

Dies ist jedoch im vorgegebenen Beispiel nicht spezifiziert und es ist nicht bekannt, wie die Drossel bei einem Moduswechsel initialisiert wird. Hier wird auch der Scope einer Transition deutlich. Da die Transition keinen Zugriff auf ihren globalen Kontext hat, kann in ihr auch nichts für die Drossel spezifiziert werden.

Prinzipiell sollten sich die Werte in Komponenten ohne Moduswechsel nicht ändern, wenn in einer anderen Komponente ein lokaler Moduswechsel stattfindet. Ein lokaler Moduswechsel ist möglich, solange sich die Werte von Ausgangsgrößen während eines Moduswechsels nicht ändern, da sich für den Kontext nichts ändert. Sobald sich die Werte von Ausgangsgrößen ändern, kann dies zu Inkonsistenzen bei der Initialisierung führen.

Nun stellt sich die Frage, ob dies schlechter Modellierungsstil ist und bei der Initialisierung einfach keine Ausgangsgrößen verändert werden dürfen. Bei kausalen Modellierungssprachen sollte dies gefordert sein. Eine solche Forderung ist jedoch bei akausalen Modellierungssprachen schwer umsetzbar, da bei der Modellierung von einzelnen Komponenten und Modi nicht immer bekannt ist, was die Ausgangsgrößen sind. Aus diesem Grund muss betrachtet werden, wie mit einer inkonsistenten Initialisierung umgegangen werden sollte.

Da die einzelnen Komponenten in einem Modell Instanzen sind, sollten noch Änderungen an den Transitionen während der Instanziierung erlaubt sein. Passt eine lokal definierte Transition nicht zum Kontext der
A25 Komponente, so sollte diese **Transition entfernt** werden können. Ebenso sollten neue **Transitionen für die ausfaktorisierte Komponente hinzugefügt** werden können. Diese neuen Transitionen haben den Scope der ausfaktorisierten Komponente.

Im Modell der Rohrleitung aus Abbildung 5.14 könnte die lokale Transition entfernt und eine neue Transition hinzugefügt werden, welche von der Komponente $Rohr - Drossel$ zu der Komponente $RohrEinfach - Drossel$ wechselt. Da die Modelle zur Simulation immer ausfaktorisiert werden müssen, sollte ein solches Hinzufügen auf den höheren Hierarchieebenen erlaubt sein.

In der Formalisierung in Abschnitt 6.1 wird die Semantik dieses Vorgehens beschrieben.

Zwei Moduswechsel dicht hintereinander

Es sei angenommen, dass sich das Modell aus Abbildung 5.15 in Modus $A1B2$ befindet, die Bedingung $a1_c$ eintritt und dabei $A2$ und $B2$ initialisiert werden. Nun wurde eventuell gerade die Bedingung $b2_C$ dadurch überbrückt. Das bedeutet, vor dem Wechsel war die Bedingung noch *false*, während sie nach der Initialisierung zu *true* evaluiert. Diese Bedingung wird dann nicht mehr als Ereignis detektiert und der notwendige Moduswechsel von $B2$ nach $B1$ findet nicht statt.

Um dies zu verhindern, kann entweder während der Initialwertberechnung überprüft werden, ob eine Wechselbedingung vom alten zum neuen Zustand eintrat, oder der neue Modus prüft bei der Initialisierung, ob er für die angegebenen Werte gültig ist (siehe auch Abschnitt 5.5).

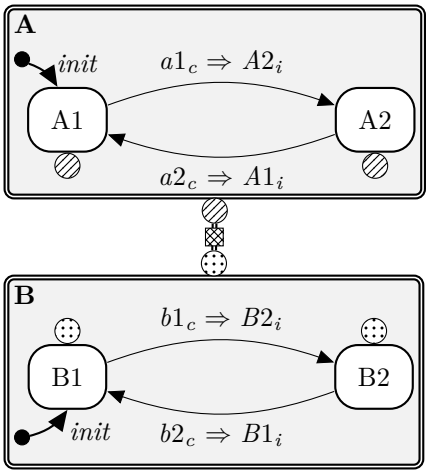


Abb. 5.15.: Modell mit zwei Komponenten mit unterschiedlichen Modi, die Schnittstellen seien kompatibel

5.3. Festlegen von globalen Daten

Hat ein Modell mehrere Modi, so beschreibt es in den meisten Fällen immer noch das gleiche System. Es gibt somit Parameter, Variablen und Gleichungen, die für alle Modi oder einen Großteil der Modi gelten.

Betrachtet man die objektorientierte Modellierung, so können die einzelnen Modi die globalen Daten erben. Da aber nicht nur objektorientierte Sprachen betrachtet werden, können globale Daten ganz allgemein in einer Komponente dargestellt werden und das restliche Modell ist mit diesen globalen Daten verbunden.

Bei jedem Moduswechsel sollten diese globalen Daten beachtet und deren Initialisierung vorgesehen werden, falls sie nicht identisch bleiben können.

5.4. Speichern von Simulationsdaten

Um alle Daten der gesamten Simulation analysieren und während der Initialisierung nutzen zu können, sollten die **Simulationsdaten nach jeder Simulation eines Modus gespeichert** werden. A26

Bei der Simulation strukturvariabler Modelle existieren mehrere Simulationsergebnisse, wobei nicht in allen Simulationen alle Daten enthalten sind oder identisch heißen. Es ist beispielsweise nicht gegeben, dass die Flughöhe einer Rakete aus unterschiedlichen Modi immer mit h benannt ist. Am Ende einer Simulation wäre es jedoch wünschenswert, sich die Flughöhe der Rakete aus allen Simulationen anzeigen zu lassen.

A27 Daher sollte ein **Beobachter definiert** werden, der Informationen aus den verschiedenen Modi sammelt und diese zu einem Gesamtergebnis zusammenfasst.

A28 Dazu müssen **Synonyme** angegeben werden, mit denen beschrieben wird, welche Variable aus einem Modus die Variable aus dem Beobachter repräsentiert.

A29 **Nicht immer sind alle Variablen, die von Interesse sind, auch in allen Modi enthalten**. Ein Beispiel dafür ist das Dominomodell, in dem Daten eines Dominosteins nur enthalten sind, wenn er simuliert wurde.

5.5. Zusätzliche Modellinformationen

Hier werden einige zusätzliche Modellinformationen diskutiert, die in der heutigen Modellierung noch nicht üblich sind, aber besonders in Bezug auf strukturvariable Modelle Vorteile haben. Diese Informationen entsprechen prinzipiell der Idee eines *Experimental Frames* [95]. Jede Komponente erhält Informationen über Bedingungen, die für die Simulation gelten müssen, damit das Modell valide Ergebnisse liefert.

5.5.1. Initialdaten

Wie bereits diskutiert, ist es sinnvoll in Komponenten anzugeben, welche Variablen initialisiert werden sollten, um einen konsistenten Zustand zu erhalten. In gleichungsbasierten Sprachen wie Modelica sind meist unterschiedliche Konstellationen möglich, während in Simulink nur ein Set von Variablen für eine Komponente erlaubt ist.

Wird bereits in Komponenten festgehalten, welche Variablen initialisiert werden können, so können bei Moduswechseln Hinweise für die Initialisierung gegeben werden. Ebenso können damit Konsistenzprüfungen eingeführt werden.

5.5.2. Solver

Auch Informationen über den **Solver können bereits in Komponenten vorgesehen werden**. So kann dort angegeben werden, welcher Solver für die Komponente erlaubt ist. A30

Dies hilft, um bei der späteren Modellierung von Moduswechseln und bei der Verbindung von Komponenten gültige Solver zu wählen.

Eine Möglichkeit, lokale Solver vorzusehen, ist in [95] vorgestellt. Diese Variante wird auch in der Formalisierung aus Abschnitt 6.1 verwendet (vgl. Definition 6.5).

5.5.3. Gültigkeiten

Da alle Modelle eine Abstraktion durch mathematische Gleichungen darstellen, sind die Modelle nur für bestimmte Randbedingungen gültig. Werden Modelle außerhalb ihres Gültigkeitsbereichs simuliert, ergeben sich falsche Werteverläufe. Die Aussagen des Modells sind somit für diese Bereiche nicht verwendbar. Nun können die Modelle entweder so erweitert werden, dass sie in größeren Bereichen gültig sind, oder es kann ein neues Modell verwendet werden, sobald ein Modell seinen Gültigkeitsbereich verlässt. Dies führt wieder zu strukturvariablen Modellen.

Zurzeit ist es in konventionellen Werkzeugen nur bedingt vorgesehen, Gültigkeiten in die Modelle einzubinden. Sie können als Kommentar angegeben werden, sie beeinflussen die Simulation dann aber nicht. Wird ohne Hinweis weiter simuliert, werden eventuell Daten analysiert, die nicht den realen Daten entsprechen, da sie mit einem ungültigen Modell ermittelt wurden.

In einer Modellierungssprache sollten Gültigkeiten angegeben werden können, damit Warnungen ausgegeben und zukünftig, eventuell sogar automatisiert, Moduswechsel eingeleitet werden können.

Gültigkeiten können verschiedene Aspekte betreffen:

- Wertebereiche
- Größenordnung der Änderungen
- Berechnung aus anderen Werten (Bsp.: Der Gültigkeitsbereich einer Variablen ist von anderen Parametern abhängig.)
- Initialwerte
- Solver
- Gültigkeiten ändern sich, abhängig vom aktuellen Zustand.

Die Gültigkeiten sind dabei vielseitig und es ist daher nicht trivial, diese in einer Modellierungssprache unterzubringen. Gültigkeiten werden im Folgenden nicht berücksichtigt, sollten aber für die Zukunft vorgesehen werden.

5.5.4. Genauigkeiten

In einem Modell oder auch in Komponenten die Genauigkeit festzuhalten, ist ebenfalls eine Information, die für strukturvariable Modelle wichtig ist. So kann beispielsweise festgehalten werden, ob zwei Komponenten kompatibel sind. Dabei ist hier nicht Kompatibilität im Sinne von Schnittstellen gemeint. Es geht darum, dass eine sehr detaillierte Komponente mit einer sehr abstrakten Komponente verbunden wird. Die detaillierte Komponente beeinflusst die Ergebnisse eventuell nicht positiv und verlangsamt die Simulation. Eine abstrakte Kompetente, die eine schnellere Simulation ermöglicht, wäre eine Alternative.

Im Extremfall können die Ergebnisse mit einer sehr detaillierten und einer sehr abstrakten Komponente sogar schlechter sein als mit zwei abstrakten Komponenten. Dies geschieht, wenn die detaillierte Komponente genauere Daten benötigt, als die abstrakte Komponente liefern kann.

Dabei müsste zusätzlich zu den Genauigkeiten in den Komponenten angegeben werden, welche Genauigkeiten kompatibel sind, also wie viel Genauigkeitsunterschied noch vertretbar ist und ab wann die Ergebnisse negativ beeinflusst werden. Auch Genauigkeiten werden im Folgenden nicht berücksichtigt, aber auch sie sollten für die Zukunft in der Modellierung vorgesehen werden.

5.6. Zusammenfassung der Modelleigenschaften

Ein Modell ist hierarchisch und modular aus Komponenten aufgebaut. Komponenten bestehen aus Gleichungen und Variablen (A1) und können durch Schnittstellen über Verbindungen (A11) mit anderen Komponenten verbunden werden (A10). Komponenten können mehrere Modi enthalten, wobei Modi wiederum Komponenten enthalten können. Für ein Modell wird angegeben, welcher Modus der initiale Modus ist (A2), und dieser lässt sich initialisieren (A3). Die Initialisierung ist durch verschiedene Sets von Variablen möglich (A21), diese Varianten sollten in den Komponenten vermerkt sein (A9). Komponenten haben einen lokalen Solver (A30).

Während der Modellierung können Modi auf Komponentenebene oder auf oberster Modellebene definiert sein. Zu jeder Zeit während der Simulation ist eine Komponentenkonstellation gültig und die Wechsel befinden sich auf oberster Modellebene (A12). Die Schnittstellen der Modi müssen gegebenenfalls angepasst werden (A14). Jedoch können bei gleichzeitigen Moduswechseln in verschiedenen Komponenten auch unterschiedliche Schnittstellen in Komponenten auftreten (A15).

Transitionen bestehen aus einer Übergangsbedingung und einer Initialisierung (A4, A8). Der Scope der Transition ist auf die Ebene, in der sie definiert ist, beschränkt (A17). Sprünge in die Vergangenheit sollten bei Moduswechseln vorgesehen werden (A23).

Die Wechselbedingung wird durch eine Bedingung beschrieben, die sich aus dem Zustand des aktuellen Modus ableitet und beliebig komplex sein kann (A5, A18). Wechselbedingungen von Transitionen, die aus einem Modus hinausführen, müssen eindeutig sein (A7).

Die Initialisierung in einer Transition kann beliebig angegeben werden (A6). Es können einfache Mappings oder kompliziertere Funktionen verwendet werden (A22), welche von den Simulationsdaten des vorherigen Modus oder alten Daten des zu simulierenden Modus abhängen (A24). Transitionen müssen entfernt oder hinzugefügt werden können, um einen konsistenten Startzustand zu erreichen (A13, A20, A25). Es kann ebenfalls erforderlich sein, lokale Transitionen in anderen Komponenten auszulösen (A16, A19).

Zur Datenauswertung sollte ein Beobachter vorgesehen werden (A27), in dem Werteverläufe von Variablen gespeichert werden. Für jeden Modus müssen diese Variablen angegeben werden (A28, A29). Zusätzlich sollten alle Daten der einzelnen Simulationen gespeichert werden, um alle Daten zugänglich zu haben (A26).

6 Formalisierung und Simulation

Die im vorherigen Kapitel zusammengetragenen Eigenschaften strukturvariabler Modelle werden in diesem Kapitel in einer Formalisierung zusammengefasst.[1] Die Formalisierung stellt damit eine werkzeugunabhängige, formale Basis zur Beschreibung der Modelle dar und konkretisiert die bereits bekannte graphische Notation. Sie enthält alle notwendigen Eigenschaften zum modularen und hierarchischen Aufbau strukturvariabler Modelle und definiert deren Semantik. Des Weiteren wird spezifiziert, wie die Moduswechsel während der Simulation realisiert werden. Es wird somit eine Basis für die Implementierung eines Simulationswerkzeugs gelegt.

Im zweiten Abschnitt des Kapitels werden anhand der Formalisierung und der bekannten Eigenschaften strukturvariabler Modelle die Anforderungen beschrieben, die ein Simulationswerkzeug erfüllen muss.

6.1. Formalisierung und graphische Notation strukturvariabler Modelle

Die folgende Formalisierung basiert auf derjenigen aus [68] und erweitert diese um die in der vorliegenden Arbeit identifizierten Eigenschaften strukturvariabler Modelle. Bei der Formalisierung wird zwischen Modellierung und Simulation unterschieden.

Bei der Modellierung geht es um die formale Beschreibung der Eigenschaften und deren Zusammenhänge als Invarianten für modular und hierarchisch aufgebaute strukturvariable Modelle. Ein strukturvariables Modell, welches die in diesem Kapitel beschriebenen Eigenschaften aufweist, enthält alle notwendigen Informationen, um in einem Werkzeug simuliert zu werden.

[1] Dabei wird auf die Nummern im Randbereich des vorherigen Kapitels verwiesen. Die Diskussion zur Notwendigkeit bestimmter Eigenschaften kann dort nachgelesen werden und wird in diesem Kapitel nicht erneut behandelt.

Bei der Spezifikation wird ein Bottom-up-Verfahren angewendet und bei einfachen Komponenten begonnen, welche um Informationen angereichert werden. Am Ende der Beschreibung wird zu erkennen sein, dass die einfachen Komponenten Spezialfälle der anderen darstellen.

Nach der Spezifikation der Modelleigenschaften wird die Simulation betrachtet. In dieser werden die dynamischen Vorgänge formalisiert, die für die Simulation des vorher definierten Modells notwendig sind. Dabei werden die Initialisierung, die Aktivierung von Transitionen und das Ende der Simulation als Funktionen beschrieben.

Als Spezifikationssprache wird die Z-Notation [78, 91] verwendet. Die bereits im vorherigen Kapitel verwendete graphische Notation wird durch die Formalisierung konkretisiert.

6.1.1. Statische Modelleigenschaften

Im Folgenden werden zunächst die Eigenschaften von Komponenten und deren Komposition betrachtet.

Sei zunächst mit den Basisinformationen begonnen, die für die späteren Formalisierungen benötigt werden. Es seien folgende Basistypen definiert:

Definition 6.1 (BASISTYPEN):

$VARNAME$	:	*Variablenbezeichner*
$VALUE$	:	*Wert einer Variablen*
$TIME$	:	*Zeit*
EQ	:	*Gleichung (differentiell, algebraisch, etc.)*
$CONNECT$	:	*Verbindungsbeschreibung*
$SOLVER$	:	*Solver*
ID	:	*Bezeichner*

Abgeleitete Basistypen

`Mehrere Gleichungen`

$EQS \mathrel{\widehat=} \mathbb{P}\, EQ$

`Zustand von Variablen`

$STATE \mathrel{\widehat=} VARNAME \pfun (TIME \pfun VALUE)$

`Mehrere Variablen`

$VARNAMESET \mathrel{\widehat=} \mathbb{P}\, VARNAME$

`Schnittstelle`

$INTERFACE \mathrel{\widehat=} VARNAMESET$

Zur Beschreibung von Modellen werden zunächst Variablenbezeichner (*VARNAME*), Werte (*VALUE*) und die Zeit (*TIME*) benötigt. Damit nicht zwischen Parametern und Variablen unterschieden werden muss, werden die Parameter (konstante Werte) als Variablen angenommen, die sich über die Zeit nicht ändern. Dazu kann beispielsweise für die Variable p die Gleichung $\dot{p} = 0$ angegeben werden.

Eine Gleichung (*EQ*) setzt Variablen in Beziehung. Es seien folgende Funktionen definiert:

Definition 6.2 (Variablen der Gleichungen):
$EQ2VAR : EQ \rightarrow VARNAMESET$ sei eine Funktion, die alle enthaltenen Variablen ($vars : VARNAMESET$) einer Gleichung ($eq : EQ$) ausgibt ($vars \mathrel{\widehat{=}} EQ2VAR(eq)$).

$EQS2VARS(eqs) \mathrel{\widehat{=}} \{v : VARNAME \mid \exists\, e : eqs \bullet v \in EQ2VAR(e)\}$
Die Funktion ($EQS2VARS(eqs)$) gibt eine Menge aller Variablen des Gleichungssystems ($eqs : EQS$) zurück.

Der Basistyp *SOLVER* beschreibt einen Solver. Es wird nicht spezifiziert, um was für eine Art Solver es sich handelt.

Eine Verbindungsbeschreibung (*CONNECT*) beschreibt den Zusammenhang von Schnittstellen (*INTERFACE*) durch eine Menge von Gleichungen (*EQS*). Eine Schnittstelle besteht aus einer Menge von Variablenbezeichnern (*VARNAMESET*). Die Verbindungsbeschreibungen werden allgemein gehalten, damit sie verschiedene Verbindungstypen darstellen können. So können auch die *connect*-Gleichungen von Modelica abgebildet werden. Ebenso sollten sich diese Verbindungsbeschreibungen bei Moduswechseln nicht ändern müssen, auch wenn die Schnittstellen sich unterscheiden. Folgende Funktion gibt die Gleichungen der Verbindungsbeschreibung zurück (A11):

Definition 6.3 (Gleichungen der Verbindungsbeschreibung):
$CON2EQS : CONNECT \rightarrow EQS$ sei eine Funktion, die zu einer gegebenen Verbindungsbeschreibung ($connect : CONNECT$) alle zugehörigen Gleichungen ausgibt ($eqs : EQS$).

Der *STATE* beschreibt den Zustand von Variablen zu bestimmten Zeitpunkten. Jeder enthaltene *VARNAME* wird dabei auf eine Menge von Tupeln abgebildet, wobei jedes Tupel aus einer Zeit und einem zugehörigen Wert besteht.

Ein $s : STATE$ könnte beispielsweise wie folgt aussehen:

$$s = \left\{ \begin{array}{l} (x \mapsto \{(t_1 \mapsto v_1), (t_2 \mapsto v_2)\}), \\ (y \mapsto \{(t_1 \mapsto v_3), (t_2 \mapsto v_4)\}) \end{array} \right\}$$

Jede Variable eines Modells ist im *STATE* enthalten. In der Beschreibung der Eigenschaften wird zunächst nur die Datenstruktur für den Zustand vorgesehen. Die Daten, die in einem *STATE* stehen, entstehen während der Simulation und werden vom Solver bestimmt, siehe Abschnitt 6.1.4.

Für die Beschreibung von Komponenten werden Klassen erstellt, welche in anderen Komponenten instanziiert werden können. Die Attribute der Komponenten und deren Invarianten werden in den Klassen beschrieben. Dabei beschreiben Attribute die Eigenschaften, welche die jeweilige Komponente bei der Instanziierung hat. Diese Eigenschaften werden durch Invarianten eingeschränkt. Werden somit bei der Instanziierung Werte von Attributen gesetzt, so müssen die angegebenen Werte immer den Invarianten genügen.

Diese Invarianten werden durch einen waagerechten Strich von den Eigenschaften getrennt. Die einzelnen Zeilen in diesem unteren Abschnitt sind implizit durch ein logisches *UND* verknüpft, welches der Übersicht halber weggelassen wird.

Basiskomponente

Eine Basiskomponente beschreibt die allgemeinste Form einer Modellkomponente, sie ist die Basis für alle folgenden Definitionen. Definition 6.4 zeigt die formale Repräsentation einer solchen Komponente.

Eigenschaften: Eine Basiskomponente hat einen Bezeichner zur Identifikation (*id*).

Die Variablenbezeichner (*vars*) repräsentieren die Variablen der Komponente.

Die Gleichungen (*eqs*) in der Komponente stellen den Zusammenhang zwischen den Variablen her und der Zustand (*state*) repräsentiert die Werte der Zustandsvariablen (A1).

Eine Komponente kann mehrere Schnittstellen (*interface*) haben, wobei jede der Schnittstellen durch eine Menge von Variablenbezeichnern beschrieben wird (A10).

Für die Simulation der Komponente werden bereits die Initialisierung und der Solver vorgesehen. Damit wird eine Art *Experimental Frame* beschrieben, in dem für eine Komponente lokal angegeben wird, unter welchen Bedingungen sie in einer Simulation gültig ist.

Zur Initialisierung einer Komponente sind oft mehrere Startwertkombinationen (*start*) möglich, so zum Beispiel in Modelica. Diese Kombinationen werden durch eine Menge von Variablenbezeichner-Mengen repräsentiert (A9, A21). Für ein Simulink-Modell hingegen muss eine vordefinierte Menge von Variablen initialisiert werden. Dies bedeutet, dass für Simulink-Modelle das Attribut *start* eine einelementige Menge ist.

Für jede Komponente wird ein Solver (*solver*) vorgesehen. Dieser ist für das Lösen des Gleichungssystems der Komponente zuständig (A30).

Definition 6.4 (BASE_COMPONENT):

BASE_COMPONENT

`Bezeichner`
id: ID

`Menge von Variablenbezeichnern`
$vars : VARNAMESET$

`Menge von Gleichungen`
$eqs : EQS$

`Zustand der Komponente`
$state : STATE$

`Mehrere Schnittstellen`
$interface : \mathbb{P}\, INTERFACE$

`Mehrere Mengen von Variablenbezeichnern`
$start : \mathbb{P}\, VARNAMESET$

`Solver`
$solver : SOLVER$

`Alle Variablen aus start sind in vars enthalten`
$\forall\, v\_start : start \bullet v\_start \subseteq vars$

`Alle Variablen aus interface sind in vars enthalten`
$\forall\, v\_interface : interface \bullet v\_interface \subseteq vars$

`Variablen aus start sind nicht im interface`
$(\bigcup start \cap \bigcup interface) = \varnothing$

`Gleichungen enthalten nur Variablen aus vars`
$EQS2VARS(eqs) \subseteq vars$

`Jede Variable hat einen Zustand`
$\mathbf{dom}(state) = vars$

Invarianten: Für eine Basiskomponente müssen alle Variablen aus der Initialisierung und der Schnittstelle in den Variablen *vars* enthalten sein. Damit beschreibt *vars* den kompletten Zustandsraum einer Komponente. In *vars* können mehr Variablen enthalten sein als in den Schnittstellen und Startwerten.

Zusätzlich sollen die zu initialisierenden Variablen nicht in den Schnittstellen enthalten sein. Diese Annahme wurde getroffen, da sich die Formalisierung bei der Komposition von Komponenten wesentlich vereinfacht. Da immer zusätzliche Variablen zur Initialisierung integriert werden können, welche mit einer Variablen der Schnittstelle gleichgesetzt werden, ist das keine Modelleinschränkung.

In den Gleichungen dürfen nur Variablen der Komponente verwendet werden, da andere Variablen nicht bekannt sind (lokaler Scope).

Jede Variable aus *vars* hat einen Zustand im *state*. Dieser muss nicht vom Modellierer angegeben werden, sondern ergibt sich implizit durch die Variablen der Komponente.

Beispiel 6.1 zeigt eine solche Komponente mit konkreten Wertebelegungen in graphischer Form.

Beispiel 6.1 (BASE_COMPONENT):

Betrachtet sei eine Basiskomponente, wie sie in Abbildung 6.1 dargestellt ist. Oben sind alle Eigenschaften der Komponente mit ihren Wertebelegungen dargestellt, welche die Invarianten erfüllen müssen. Darunter ist die graphische Repräsentation dargestellt, in der die ID und die Schnittstellen aus Gründen der Übersichtlichkeit graphisch dargestellt sind. Diese Darstellung wird im Folgenden weiterverwendet.

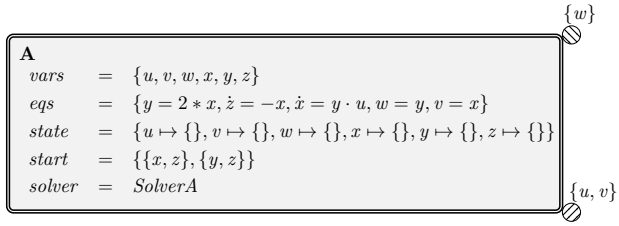
Die Komponente besteht aus sechs Variablen und fünf Gleichungen. Um ein lösbares Gleichungssystem zu erhalten, muss eine der Variablen aus den Schnittstellen eine Eingangsgröße sein.

Als Startwerte können entweder die Variablen x, y oder y, z angegeben werden. Beide Varianten ermöglichen eine eindeutige Initialisierung der Komponente. Wird x initialisiert, wird implizit auch v aus der Schnittstelle initialisiert.

Der state enthält noch keine Daten, da hier der Zustand vor der Simulation dargestellt ist.

$$
\begin{array}{lcl}
id & = & A \\
vars & = & \{u, v, w, x, y, z\} \\
eqs & = & \{y = 2 * x, \dot{z} = -x, \dot{x} = y \cdot u, w = y, v = x\} \\
state & = & \{u \mapsto \{\}, v \mapsto \{\}, w \mapsto \{\}, x \mapsto \{\}, y \mapsto \{\}, z \mapsto \{\}\} \\
interface & = & \{\{u, v\}, \{w\}\} \\
start & = & \{\{x, z\}, \{y, z\}\} \\
solver & = & SolverA
\end{array}
$$

(a) Formale Beschreibung



(b) Graphische Notation

Abb. 6.1.: Graphische Darstellung einer Basiskomponente

Komposition von Komponenten

Modelle bestehen nicht nur aus einer Komponente, sondern aus einer Menge von Komponenten, die miteinander in Verbindung stehen. Die enthaltenen Komponenten entsprechen dabei Instanzen von anderen Komponenten. Komponenten können ineinander geschachtelt werden, wodurch sich die Modellhierarchie ergibt.

Um dies zu beschreiben, wird ein Bottom-up-Verfahren verwendet. Das bedeutet, es wird bei der tiefsten Hierarchieebene begonnen und die Komponenten dieser Ebene werden komponiert. Aus der Komposition ergibt sich eine „neue“ Komponente.

Diese entspricht in ihrer Außenansicht einer normalen Basiskomponente und kann mit anderen Komponenten komponiert werden. Auf der obersten Ebene ergibt sich somit als Außenansicht eine einzelne Komponente. Dieses Vorgehen entspricht einem Flattening, da nach außen die modularen und hierarchischen Eigenschaften nicht mehr erkennbar sind.

Definition 6.5 zeigt die Formalisierung einer solchen Komposition.

Eigenschaften: Eine komponierte Komponente hat die Eigenschaften und Invarianten einer Basiskomponente (erbt von *BASE_COMPONENT*). Alle weiteren Eigenschaften sind die lokalen Eigenschaften. Diese sind nach außen nicht sichtbar. So wird ähnlich wie zu [95] sichergestellt, dass komponierte Komponenten mit anderen Komponenten komponiert werden können und dafür keine extra Semantik definiert werden muss.

Die komponierte Komponente setzt sich aus enthaltenen Instanzen von Komponenten (*children*) zusammen. Diese sind durch mehrere Verbindungsbeschreibungen (*connect*) verbunden. Die enthaltenden Komponenten können vom Typ *BASE_COMPONENT* oder von Typen, welche von diesem erben, sein.

Die Solver in komponierten Komponenten werden identisch zu denen aus [95] behandelt. Dort hat jede atomare Klasse (*BASE_COMPONENT*) einen Solver. Gekoppelte Klassen (*COMPOSIT_COMPONENT*) haben einen *Koordinator* (*coor*). Dieser koordiniert die Solver der atomaren Klassen in der gekoppelten Klasse. Dazu erhält er Informationen über lokale Ereignisse der Komponenten und propagiert diese zu den davon abhängigen Komponenten. Dieses Propagieren ist ein Senden von externen Ereignissen zu den Komponenten, die dann wiederum darauf reagieren können. Ereignisse können sowohl Werteänderungen von Variablen als auch tatsächliche diskrete Ereignisse betreffen. Durch die lokalen Solver ist die vorgestellte Formalisierung für unterschiedlichste Solver verwendbar und enthält bereits lokal Informationen zum *Experimental Frame* der Komponente. In [95] wird ebenso gezeigt, dass der *Koordinator* der gekoppelten Klasse nach außen wiederum wie ein normaler Solver aussieht. Die *COMPOSIT_COMPONENT* enthält somit auch einen Koordinator (*coor* : *COORDINATOR*). Dieser wird als Spezialfall eines Solvers angesehen.

Invarianten: Die Variablen (*vars*) der komponierten Komponente setzen sich aus den Variablen der Komponenten und Schnittstellen der neuen Komponente zusammen.

Die Gleichungen (*eqs*) setzten sich aus den Gleichungen der enthaltenen Komponenten und den Gleichungen, die sich aus der Verbindungsbeschreibung ergeben, zusammen. Dies ist ein ähnliches Vorgehen wie beim Flattening von Modelica-Modellen.

In den Gleichungen, die sich aus der Verbindungsbeschreibung (*connect*) ergeben, dürfen nur Variablen aus den Schnittstellen enthalten sein.

Definition 6.5 (COMPOSIT_COMPONENT):

COMPOSIT_COMPONENT

Erbt alles aus BASE_COMPONENT (Außenansicht)

BASE_COMPONENT

Lokale Eigenschaften

Enthaltene Komponenten

$children : \mathbb{P} \downarrow BASE\_COMPONENT$

Verbindungsbeschreibung

$connect : \mathbb{P}\ CONNECT$

Koordinator der Solver (Spezialfall eines Sovlers)

$coor : COORDINATOR$

---

vars besteht aus allen Variablen der Kinder und Schnittstellen

$vars = (\bigcup c : children \bullet c.vars) \cup (\bigcup interface)$

eqs besteht aus allen eqs der enthaltenen Komponenten und der Verbindungen

$eqs = (\bigcup c : children \bullet c.eqs) \cup (\bigcup c : connect \bullet \boldsymbol{CON2EQS}(c))$

Nur Schnittstellenvariablen sind in Verbindungen erlaubt

$\forall\, c : connect \bullet \forall\, es : \boldsymbol{CON2EQS}(c) \bullet \boldsymbol{EQS2VARS}(es)$
$\quad \subseteq ((\bigcup interface) \cup (\bigcup c : children \bullet c.interface))$

start setzt sich aus allen Kombinationen der starts der Kinder zusammen

$start = \boldsymbol{COMP\_START}(children)$

IDs der children sind eindeutig

$\forall\, c_1, c_2 : children | c_1 \neq c_2 \bullet c_1.id \neq c_2.id$

Koordinator ist der Solver in Außensicht

$coor = solver$

Die Initialisierung der neuen Komponente besteht aus allen Konstellationen der Initialisierungen der enthaltenen Komponenten. Dafür wird die Funktion *COMP_START* verwendet (siehe Definition 6.6).

Definition 6.6 (Komposition von *start*):

COMP_START $: \mathbb{P} \downarrow BASE\_COMPONENT \rightarrow \mathbb{P}\ VARNAMESET$ *sei eine Funktion, die aus allen Startkombinationen der enthaltenen Komponenten (*$children : \mathbb{P} \downarrow BASE\_COMPONENT$*) eine neue Menge von Startkombinationen (*$start_n = \mathbb{P}\ VARNAMESET$*) erstellt.*
Beispiel: Gegeben seien die Komponenten A und B, wobei die Startkombinationen $A.start = \{\{x\}, \{y\}\}$ *und* $B.start = \{\{a, b\}, \{a, c\}\}$ *angenommen werden.*
Dann ergibt $COMP\_START(\{A, B\}) \mathrel{\widehat{=}}$
$\{\{x_A, a_B, b_B\}, \{x_A, a_B, c_B\}, \{y_A, a_B, b_B\}, \{y_A, a_B, c_B\}\}$.
Im Beispiel 6.2 ist dies graphisch erklärt.

Die Bezeichner (*ids*) der enthaltenen Komponenten (*children*) müssen innerhalb der Komponente eindeutig sein.

Variablen aus den Komponenten müssen eindeutig sein. Aus diesem Grund wird angenommen, dass diese durch einen Index ihrer vorherigen Komponente zugeordnet werden können. Dieser Index ist der eindeutige Bezeichner der Komponente. Dies ist ebenfalls in Beispiel 6.2 graphisch dargestellt.

Der Koordinator (*coor*) ist der Solver (*solver*) für die komponierte Komponente. In der Außensicht ist der Koordinator nicht mehr erkennbar, sondern nur der Solver.

Beispiel 6.2 (COMPOSIT_COMPONENT):

Abbildung 6.2(a) zeigt zwei miteinander verbundene Komponenten (A, B). Diese bilden zusammen eine neue Komponente C, Abbildung 6.2(b) . Der state wird graphisch nicht dargestellt, da er sich aus den enthaltenen Variablen ergibt. Die Gleichungen der Komponenten werden hier abstrakt betrachtet und als gX beschrieben. Diese können beliebige Zusammenhänge darstellen.

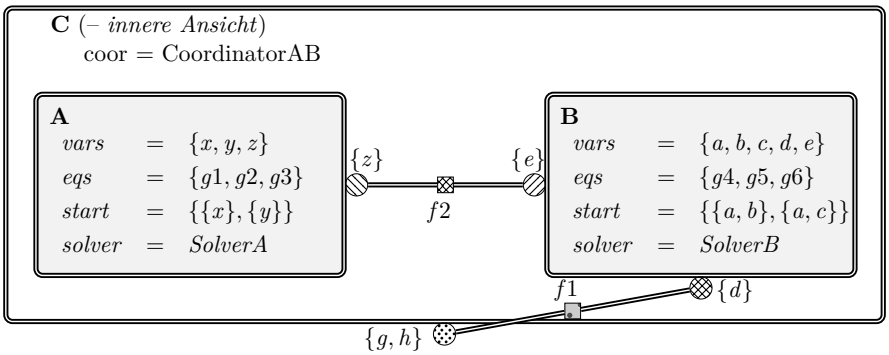
Die Schnittstellen werden durch Verbindungsbeschreibungen kombiniert (f_1, f_2). Die Komponente C hat eine Schnittstelle, diese ist nach außen sichtbar und kann mit den internen Schnittstellen verbunden werden. Der Koordinator (CoordinatorAB) der Komponente C ist für die Koordination der beiden lokalen Solver erforderlich.

Die Außenansicht der Komponente C entspricht wieder genau einer Basiskomponente. Diese hat eine Schnittstelle ($\{g, h\}$). Der Koordinator (CoordinatorAB) der beiden lokalen Solver (SolverA, SolverB) ist in der Außensicht nur noch ein Solver (SolverAB).

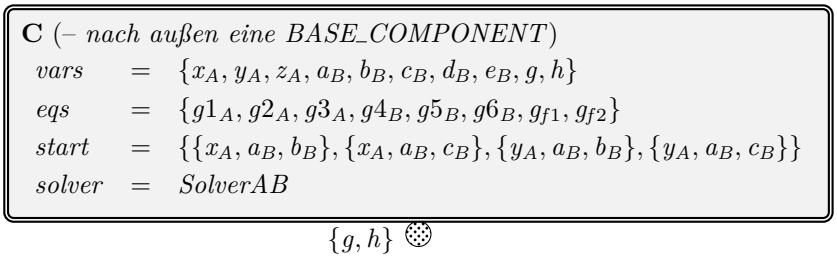
Die Verbindungsbeschreibungen $f1$ und $f2$ führen zu konkreten Gleichungen g_{f1} und g_{f2} in der komponierten Komponente. Dass Komponenten aus weiteren Komponenten bestehen können, ist nach außen nicht mehr relevant. Die neue Komponente C kann nun in einer anderen Komponente instanziiert und als Basiskomponente behandelt werden.

Die Darstellung ist nicht direkt in der Lage, die in Modelica möglichen Strukturierungsmittel darzustellen, da in Modelica ein Objekt zwar aus einer Komposition von Komponenten besteht, aber zusätzlich auch eigene Variablen enthalten kann. So können in Modelica globale Daten dargestellt werden, welche für mehrere Komponenten gelten. In der hier gewählten Darstellung muss eine Komponente vor-

gesehen werden, welche die globalen Daten bereitstellt. Beide Darstellungen sind legitim und ineinander umformbar. Die hier gewählte Form vereinfacht die Formalisierung, da die Komponenten nicht um weitere Daten erweitert werden müssen. Die in Modelica gewählte Darstellung ist für die Implementierung und Beschreibung von Modellen intuitiver, da sie etwas handlichere Scoping-Regeln hat.



(a) Verbindung von zwei Komponenten



(b) Außenansicht der komponierten Komponente

Abb. 6.2.: Graphische Darstellung einer komponierten Komponente

6.1.2. Dynamische Modelleigenschaften

Bisher wurden nur normale Modelle betrachtet, wie sie bereits in Simulationswerkzeugen modelliert und simuliert werden können. Im Folgenden werden die strukturändernden Eigenschaften formal beschrieben. Dazu müssen die Eigenschaften der Transitionen und Modi betrachtet werden. Es entsteht ein komplettes strukturvariables Modell, sodass in Abschnitt 6.1.4 die Dynamik der Simulation und der Strukturwechsel beschrieben werden kann.

Transition

Seien zunächst die Transitionen betrachtet. Abbildung 6.3 zeigt schematisch eine Instanz einer Transition (t).

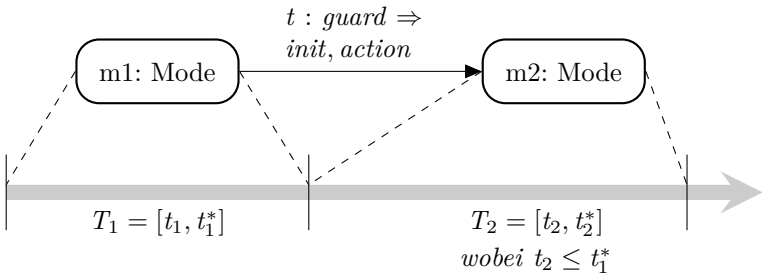


Abb. 6.3.: Aufbau einer Transition

Eine Transition stellt den Zusammenhang zwischen zwei Modi ($m1$, $m2$) her und beschreibt den Wechsel von einem Modus zum anderen. Die Transition wird durch eine Wechselbedingung (*guard*) aktiviert und führt eine Aktion durch (A4). In der Formalisierung wird zwischen Initialisierung und Aktion unterschieden. Die Initialisierung (*init*) betrifft nur das Setzen von Startwerten, während die Aktion (*action*) beliebige Vorgänge beschreiben kann. Auf die Aktionen wird später näher eingegangen. Ein Modus ist immer für ein Zeitintervall (T_x) gültig und der nächste Modus startet entweder zeitlich direkt nach dem anderen oder geht in die Vergangenheit zurück. Die Spezifikation einer allgemeinen Transition ist in Definition 6.7 gezeigt.

Eigenschaften: Eine Transition besteht aus einer Komponente, welche den vorherigen Modus repräsentiert (*pre*), und einer Komponente, welche den nachfolgenden Modus repräsentiert (*post*). Die Modi sind vom Typ *BASE_COMPONENT* oder vom Typ der von *BASE_COMPONENT* erbenden Klassen.

Die Wechselbedingung (*guard*) ist abstrakt als Funktion modelliert. Diese Funktion weist jedem einnehmbaren Zustand eines Modells einen Wahrheitswert zu. Zu jeder Zeit kann überprüft werden, ob der aktuelle Zustand zu einem *True* führt und damit die Transition aktiviert wird. Diese Art der Darstellung wurde gewählt, da für die Formalisierung keine eigene Ereigniserkennung (bspw. *zero-crossing-detection*) definiert und trotzdem ein Moduswechsel detektiert werden soll. Wenn alle möglichen Zustände in der Wechselbedingung enthalten sind, ist keine Ereignisdetektion notwendig. Es müssen lediglich der aktuelle Zustand gesucht und der zugehörige Wahrheitswert ausgelesen werden. In der Modellierung soll der Modellierer z.B. $x > 5$ schreiben können. Als graphische Notation wird für die Wechselbedingung eine Funktion $g(state)$ vorgesehen, welche vom Zustand ($state$) abhängt. Diese gibt ein Prädikat zurück (A18).

Die Initialisierung *init* beschreibt eine Funktion, welche den Zustand des neuen Modus $state_{pre\_old}$ und eventuell bereits simulierte Daten des neuen Modus $state_{post\_old}$ erhält und auf einen Zustand $state_{post\_new}$ abbildet (A6, A8, A22).

Diese Funktion kann als $state_{post\_new} = f(state_{pre\_old}, state_{post\_old})$ verstanden werden. Diese Schreibweise wird auch in der graphischen Notation verwendet.

Die Aktion ($action : ACTION$) stellt eine beliebig definierbare Aktion dar. Diese dient dazu Spezialfälle zu behandeln, wenn beispielsweise weitere Transitionen ausgelöst werden oder ein Modus neu kompiliert werden muss (A16, A19). Dabei kennt sie die im Scope der Transition enthaltenen Informationen, siehe Beispiel 6.5 weiter unten. Die Aktion wird nicht weiter spezifiziert, da sie einen Sonderfall darstellt und theoretisch beliebige Anpassungen während der Transition zulässt. Aus diesem Grund sollte solch eine Aktion auch nur in Ausnahmen verwendet werden, da keine Konsistenzprüfungen durchgeführt werden können und der Modellierer durch fehlerhafte Aktionen die Simulationsergebnisse verfälschen kann.

Definition 6.7 (TRANSITION):

TRANSITION

`Komponente, aus der die Transition hinausführt`

$pre :\downarrow BASE\_COMPONENT$

`Komponente, zu der die Transition führt`

$post :\downarrow BASE\_COMPONENT$

`Wechselbedingung`

$guard : STATE \nrightarrow \mathbb{B}$

`Auszuführende Initialisierung bei Aktivierung der Transition`

$init : (STATE \times STATE) \nrightarrow STATE$

`Auszuführende Aktion bei Aktivierung der Transition - beliebige Funktion`

$action : ACTION$

`Guard kann nur Variablenbezeichner verwenden, die in pre enthalten sind`

$\forall s : \mathbf{dom}(guard) \bullet \mathbf{dom}(s) \subseteq pre.vars$

`Alle zu initialisierenden Variablen aus` *init* `sind Element aus` *start* `von post`

$\forall s : \mathbf{ran}(init) \bullet \mathbf{dom}(s) \in post.start$

`Alle Initialdaten haben genau einen Wert zur gleichen Zeit`

$\forall s : \mathbf{ran}(init) \bullet \forall v1, v2 : \mathbf{dom}(s) \bullet \mathbf{dom}(s(v1)) = \mathbf{dom}(s(v2))$
$\wedge \#s(v1) = 1 \wedge s \in \mathbb{F}(VARNAME \times (TIME \times VALUE))$

`Daten zum Berechnen aus vorherigem Modus oder Vorzustand des neuen Modus`

$\forall s : \mathbf{dom}(init) \bullet \mathbf{first}(s) \subseteq pre.state \wedge \mathbf{second}(s) \subseteq post.state$

Invarianten: Die Wechselbedingung darf nur von den Daten des vorherigen Modus abhängen (A5).

In der Initialisierung wird für den zu aktivierenden Modus der initiale Zustand bestimmt. Der zu bestimmende Zustand darf nur Variablen aus den zu initialisierenden Variablen (*start*) des Modus enthalten (A9).

Für jede Variable darf nur ein Wert angegeben werden und die Zeiten für die Werte müssen für alle Variablen identisch sein. Die hier gewählte Zeit wird später bei der Ausführung der Transition als Startzeit für die nächste Simulation verwendet, siehe Definition 6.15. Ein Zurückgehen in die Vergangenheit ist mit der gewählten Initialisierung möglich, da für den initialen Zustand eine beliebige Zeit angegeben werden kann. Diese Zeit muss kleiner oder gleich der Zeit des Eintretens der Wechselbedingung sein, nur so ist sichergestellt, dass Daten zur Initialisierung vorhanden sind (A23). Diese Überprüfung wird in Definition 6.15 spezifiziert, welche beschreibt, was beim Eintreten eines Ereignisses geschieht.

Zur Initialisierung kann der Zustand des vorherigen Modus und auch der alte Zustand des neuen Modus (A24) verwendet werden.

Dynamische Komponente

Es seien nun dynamische Komponenten betrachtet, siehe Definition 6.8. Dynamische Komponenten bestehen aus mehreren Modi, zwischen denen sie wechseln können.

Eigenschaften: Eine solche dynamische Komponente besitzt alle Eigenschaften einer *BASE_COMPONENT*.

Zusätzlich hat sie Modi (*modes*), welche wiederum Basiskomponenten sind oder von dieser erben. Ein Modus kann somit ebenfalls eine dynamische Komponente sein.

Der bei der Initialisierung aktive Modus wird in *initMode* angegeben (A2). Dass dieser Modus zu Beginn auch aktiv ist, wird erst bei der Beschreibung der Simulation des Modells angegeben (Abschnitt 6.1.4).

Die Transitionen zwischen den Modi sind im Attribut *trans* enthalten.

Invarianten: Der aktive Modus und der initiale Modus müssen in der Menge der Modi enthalten sein (A2).

Der Scope einer Transition ist auf die Modi beschränkt, mit denen sie verbunden ist (A13, A17). Die enthaltenen Transitionen müssen eindeutige Wechselbedingungen haben, sofern sie aus demselben Modus hinaus führen (A7).

Die Eigenschaften der Komponente (*vars*, *eqs*, etc.) hängen vom gerade aktiven Modus ab und werden durch diesen repräsentiert.

Der *state* ist nicht der *act.state*, da nicht alle vergangenen Daten im *state* der dynamischen Komponente enthalten sind. Der *state* ergibt sich aus den Invarianten der Basiskomponente und enthält die Daten der Variablen des gerade aktiven Modus (mehr dazu in der Definition des Simulationsmodells 6.13).

Es wird keine Invariante für die Schnittstellen der Modi in der Komponente verlangt, da diese nicht identisch sein müssen (A15). Passen die Schnittstellen einer Komposition nicht über die Verbindungsbeschreibungen zusammen, so werden die Invarianten verletzt. In diesem Fall sollten die Schnittstellen angepasst werden (A14).

Die Bezeichner (*id*s) der Modi müssen innerhalb der Komponente eindeutig sein.

Hat eine dynamische Komponente genau einen Modus, so entspricht sie vom Verhalten her einer Basiskomponente. Damit ist eine Basiskomponente ein Spezialfall einer dynamischen Komponente und nicht umgekehrt. Eine dynamische Komponente enthält zwar exakt die Eigenschaften einer Basiskomponente, aber die dynamische Komponente kann als allgemeinster Fall angenommen werden. Im Folgenden werden alle Komponenten daher als dynamische Komponente angenommen.

Beispiel 6.3 zeigt eine solche dynamische Komponente.

Definition 6.8 (DYNAMIC_COMPONENT):

DYNAMIC_COMPONENT

Erbt alles aus BASE_COMPONENT (Außenansicht)

BASE_COMPONENT

Lokale Eigenschaften

Alle enthaltenen Modi

$modes : \mathbb{P} \downarrow BASE\_COMPONENT$

Enthaltene Transitionen

$trans : \mathbb{P}\ TRANSITION$

Initialer Modus

$initMode :\downarrow BASE\_COMPONENT$

Aktuell aktive Komponente

$act :\downarrow BASE\_COMPONENT$

Aktuelle Komponente und initialer Modus sind in modes enthalten

$$act \in modes \land initMode \in modes$$

Modi aus Transitionen sind in modes enthalten

$$\forall\, t : trans \bullet t.pre \in modes \land t.post \in modes$$

Zwei Transitionen dürfen keine guards haben, die gleichzeitig true sind

$$\forall\, t_1, t_2 : trans | t_1 \neq t_2 \land t_1.pre = t_2.pre \bullet \forall\, s : \mathbf{dom}(t_1.guard) | \\ s \in \mathbf{dom}(t_2.guard) \bullet \neg(t_1.guard(s) \land t_2.guard(s))$$

Eigenschaften der Komponente entsprechen denen des aktuellen Modus

$$vars = act.vars \land eqs = act.eqs \land start = act.start$$
$$solver = act.solver \land interface = act.interface$$

Die IDs der Modi sind eindeutig

$$\forall\, c_1, c_2 : modes | c_1 \neq c_2 \bullet c_1.id \neq c_2.id$$

Beispiel 6.3 (Dynamische Komponente): *Betrachtet wird die dynamische Komponente mit zwei Modi aus Abbildung 6.4. Die Komponente A1 ist die initiale Komponente und wird durch start initialisiert. Diese Initialisierung wird in der Formalisierung erst im Simulationsmodell betrachtet, siehe Definition 6.13. Graphisch wird sie jedoch bereits angedeutet.*

Beide Modi haben unterschiedliche Variablen und Gleichungen, aber die gleiche Schnittstelle.

Es sind zwei Transitionen vorhanden, die jeweils aus einer Übergangsbedingung und einer Initialisierung bestehen. Die Übergangsbedingung hängt dabei vom Vorzustand ab und die Initialisierung ist eine Funktion der Zustände beider Komponenten.

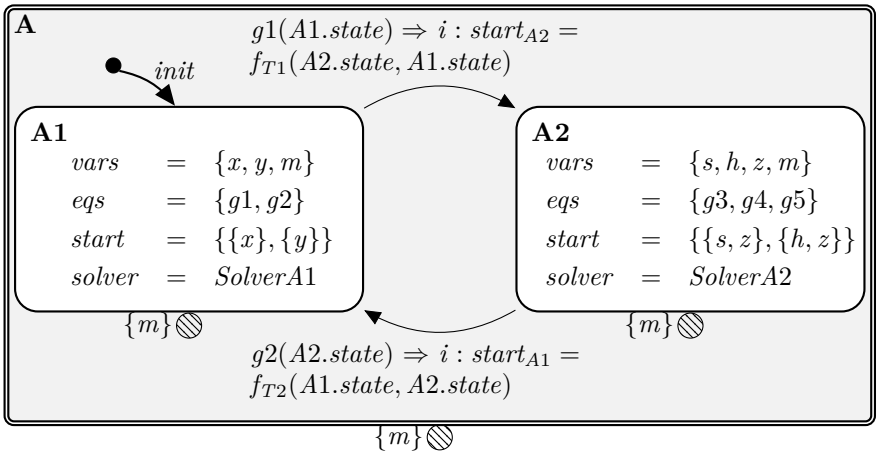


Abb. 6.4.: Dynamische Komponente

Komposition dynamischer Komponenten

Dynamische Komponenten sollen auch mit anderen Komponenten verbunden werden können. Dies soll im Folgenden anhand von einer Komposition von zwei dynamischen Komponenten gezeigt werden.

Zunächst wird dies anhand des Beispiels 6.4 vorgestellt, damit die Formalisierung leichter zu verstehen ist.

Beispiel 6.4 (Komposition zweier dynamischer Komponenten): *Es sei Abbildung 6.5 betrachtet. Das Modell besteht aus zwei Komponenten, wobei die Komponente A zwei Modi mit gleichen Schnittstellen hat und die Komponente B aus einem Modus besteht. Die Verbindungsbeschreibung f1 verbindet die beiden Komponenten. Der Koordinator (coor) in der Komponente (AB) koordiniert die Solver der gerade aktiven Komposition von Komponenten (A1B bzw. A2B).*

Zu Beginn einer Simulation müssen die initialen Modi miteinander komponiert werden. Wie bekannt wird aus komponierten Komponenten eine neue Komponente (vgl. Definition 6.5). Es ergibt sich der erste ausfaktorisierte Modus A1B aus der Komposition von A1 und B.

Die Transition führt aus diesem Modus hinaus. Wird sie aktiviert, muss der nächste Modus komponiert werden. Dieser ergibt sich zu A2B. Für jede komponierte Komponente dürfen die Verbindungen nur Variablen aus den Schnittstellen verwenden (vgl. Definition 6.5). Da die Verbindungsbeschreibungen als nicht änderbar angenommen werden, müssen sich die Schnittstellen über diese Beschreibung verbinden lassen. Dies bedeutet nicht, dass die Schnittstellen zwangsläufig identisch sein müssen (A15).

Es entsteht eine neue dynamische Komponente mit zwei Modi. Die Modi bestehen dabei je aus mehreren Komponenten, was auf der neuen Modellebene jedoch nicht mehr relevant ist. Der alte Koordinator (CoorinatorAB) ist nach außen nur noch ein Solver im jeweiligen Modus (SolverA1B, SolverA2B). Abbildung 6.6 zeigt die neue, ausfaktorisierte dynamische Komponente.

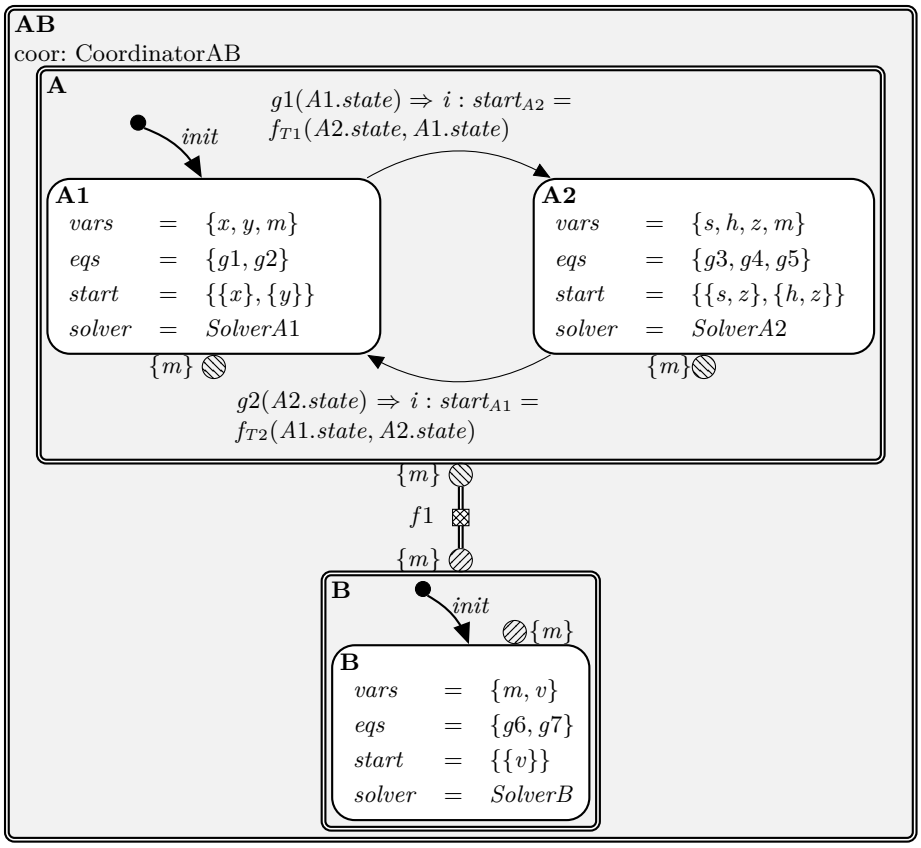


Abb. 6.5.: Dynamische Komponente (A) komponiert mit einer anderen Komponente (B)

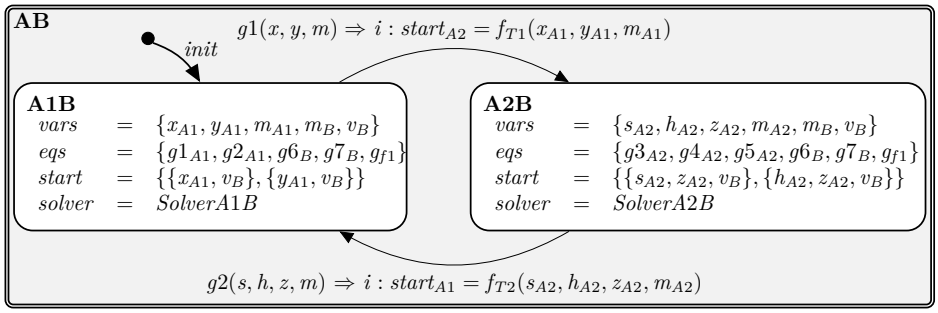


Abb. 6.6.: Ausfaktorisiertes Modell zur Abbildung 6.5 – Außensicht: neue dynamische Komponente (AB) mit zwei Modi

Im Folgenden seien die Wechselbedingungen und Initialisierungen näher betrachtet. Da die Transitionen eine Ebene nach oben gezogen wurden, stellt sich die Frage, wie die Variablen initialisiert werden müssen, für die keine Angaben gemacht wurden. Es wird vorgegangen wie in Abschnitt 5.2.2 beschrieben. Alle Werte von Variablen, deren Werte in der Initialisierung nicht spezifiziert wurden, bleiben gleich. Lokale Transitionen können, falls notwendig, entfernt werden und neue Transitionen können auf der neuen Hierarchieebene hinzugefügt werden (A25). Beispiel 6.5 zeigt solch ein Entfernen und Hinzufügen von Transitionen (A19, A20).

Beispiel 6.5 (Entfernen und Hinzufügen von Transitionen):
Abbildung 6.7 zeigt eine Erweiterung des Beispiels 6.4.

Wie bereits in Abschnitt 5.2.3 erläutert, sollten lokale Transitionen aus Komponenten entfernt werden können, wenn durch sie Inkonsistenzen entstehen. Ebenso sollten in der neuen komponierten Komponente neue Transitionen hinzugefügt werden können. Diese Transitionen können komplett neue Moduswechsel einleiten oder als Ersatz für eine entfernte Transition dienen. Dazu muss in der komponierten Komponente angegeben werden, welche Transitionen entfernt werden und welche hinzukommen. In der Außensicht der komponierten Komponente sind die Transitionen dann dementsprechend angepasst.

Exemplarisch ist dies in Abbildung 6.7 gezeigt. Dabei wird die Transition $t1$ entfernt (delete) und es wird eine neue Transition hinzugefügt (add).[2] *Diese ist eine normale Transition, welche die beiden Modi, zwischen denen diese Transition wechselt, einen guard und ein init erhält.*

Abbildung 6.8 zeigt die neue, komponierte dynamische Komponente mit ihren ausfaktorisierten Modi und der neuen Transition (t_{new}). Die Transition $t2$ wurde nur eine Ebene nach oben gezogen. Der Scope von $t2$ ist noch identisch und die Wechselbedingung und Initialisierung hängt nur von den Variablen von $A1$ bzw. $A2$ ab.

Die Komposition dynamischer Komponenten und das Hinzufügen/Entfernen von Transitionen sollen nicht nur anhand eines Beispiels vorgestellt werden, sondern auch formalisiert werden. Um die formale Darstellung zu vereinfachen, werden die Hilfsdefinitionen 6.9 und 6.10 verwendet. Die Formalisierung der Komposition ist in Definition 6.11 gezeigt.

[2] Hier wurde die Instanz der Transition mit $t1$ benannt, um diese zu identifizieren.

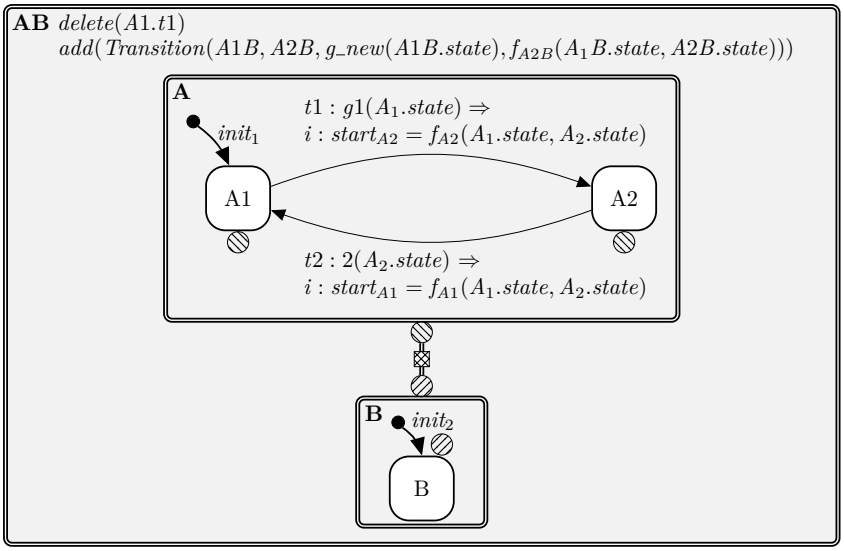


Abb. 6.7.: Modell AB mit Entfernen und Hinzufügen einer Transition, um Inkonsistenzen bei einem Moduswechsel zu vermeiden

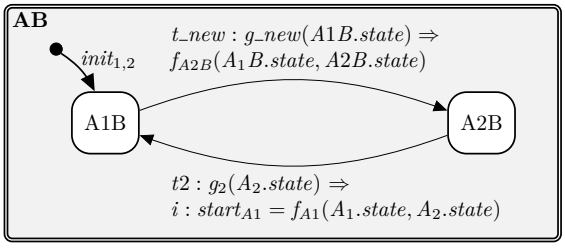


Abb. 6.8.: Ausfaktorisiertes Modell zur Abbildung 6.7 – Außensicht: neue dynamische Komponente mit zwei Modi und geänderten Transitionen

Definition 6.9 (Komposition dynamischer Komponenten):
COMP_MODES : ($\mathbb{P}$ *DYNAMIC_COMPONENT*, *CONNECT*)
$\rightarrow \mathbb{P}$ *DYNAMIC_COMPONENT*
sei eine Funktion, die zu einer gegebenen Menge von dynamischen Komponenten (dyn_comp : DYNAMIC_COMPONENT) und deren Verbindungsbeschreibungen (connect : CONNECT) eine neue Menge von dynamischen Komponenten nach Definition 6.5 komponiert. Es entsteht eine Menge von dynamischen Komponenten, welche die ausfaktorisierten Modi repräsentieren.

Beispiel: Gegeben seien zwei dynamische Komponenten (A, B) mit je zwei Modi ($A.modes = \{A_1, A_2\}$, $B.modes = \{B_1, B_2\}$). Die beiden Komponenten seien mit einer Verbindungsbeschreibung (connect : CONNECT) verbunden. Dann kann die Funktion COMP_MODES wie folgt verwendet werden:
$COMP\_MODES(\{A, B\}, connect) \mathrel{\widehat{=}} \{A_1B_1, A_1B_2, A_2B_1, A_2B_2\}$

Definition 6.10 (Komposition von Transitionen):
COMP_TRANS : ($\mathbb{P}$ *DYNAMIC_COMPONENT*, $\mathbb{P}$ *TRANSITION*, $\mathbb{P}$ *TRANSITION*) $\rightarrow \mathbb{P}$ *TRANSITION*
sei eine Funktion, die zu einer gegebenen Menge von dynamischen Komponenten (dyn_comp : $\mathbb{P}$ DYNAMIC_COMPONENT), neu definierten Transitionen (new_trans : $\mathbb{P}$ TRANSITION) und zu entfernenden Transitionen (del_trans : $\mathbb{P}$ TRANSITION) eine Menge von neuen Transitionen ausgibt.

Die lokalen Transitionen werden dann eine Ebene nach oben gezogen, so wie es auch bei der Ausfaktorisierung von Zustandsautomaten bekannt ist. Der Scope der Transition ändert sich dadurch nicht. Die zu entfernenden Transitionen werden aus der gerade erstellten Menge von Transitionen entfernt, während die neuen Transitionen hinzugefügt werden.

Eigenschaften: Die komponierte dynamische Komponente besitzt alle Eigenschaften einer *DYNAMIC_COMPONENT* (Außensicht). Sie besteht zusätzlich aus einer Menge dynamischer Komponenten (*children*) und den Verbindungsbeschreibungen zwischen diesen Komponenten (*connect*).

Neue Transitionen können in *new_trans* definiert werden, während zu löschende Transitionen in *del_trans* angegeben werden.

Definition 6.11 (COMP_DYNAMIC):

COMP_DYNAMIC

Erbt alles aus DYNAMIC_COMPONENT (Außensicht)

DYNAMIC_COMPONENT

Lokale Eigenschaften

Dynamische Komponenten

$children : \mathbb{P}\, DYNAMIC\_COMPONENT$

Verbindungsbeschreibungen

$connect$: $CONNECT$

Neue Transitionen

$new\_trans : \mathbb{P}\, TRANSITION$

Zu löschende Transitionen

$del\_trans : \mathbb{P}\, TRANSITION$

---

modes sind alle Konstellationen der dynamischen Komponenten, siehe Def. 6.9

$modes = \boldsymbol{COMP\_MODES}(children, connect)$

del_trans sind lokale Transitionen

$\forall\, t : del\_trans \bullet \exists\, c : children \bullet t \in c.trans$

trans sind alle Transitionen auf der neuen Hierarchiebene, siehe Def. 6.10

$trans = \boldsymbol{COMP\_TRANS}(children, new\_trans, del\_trans)$

Initialer Modus setzt sich aus allen initialen Modi zusammen

$\forall\, c : children.initMode \bullet \exists\, c2 : children \bullet c2.initMode = c$

Invarianten: Die neuen Modi in *modes* ergeben sich aus der Komposition aller Modi.

Die zu löschenden Transitionen müssen lokale Transitionen der enthaltenden Komponenten (*children*) sein. Die neuen Transitionen der komponierten dynamischen Komponente (*trans*) ergeben sich aus den neuen Transitionen (*new_trans*) und den Transitionen, welche nun eine Hierarchieebene hochgezogen wurden.

Der initiale Modus setzt sich aus allen initialen Modi der enthaltenden Komponenten zusammen.

Die Außensicht einer komponierten dynamischen Komponente entspricht einer normalen dynamischen Komponente (Definition 6.8). Diese kann dann wiederum mit anderen Komponenten verbunden werden, um neue dynamische Komponenten zu formen. Dies kann fortgesetzt werden, bis genau eine dynamische Komponente entsteht. Ein Ausfaktorisieren ist also immer möglich, kann jedoch zu vielen Modi und komplizierten Transitionen führen.

6.1.3. Eigenschaften des Simulationsmodells

Zur Simulation wird ein Simulationsmodell benötigt, welches aus einer dynamischen Komponente besteht und die Modi auf der obersten Hierarchieebene hat (A12).

Jeder Modus innerhalb dieser dynamischen Komponenten muss ein lauffähiges Simulationsmodell sein, welches durch das in den Grundlagen beschriebene Vorgehen aus Abschnitt 2.3 simuliert werden kann. Für ein solches Modell wird im Folgenden die Datenstruktur beschrieben.

Aktuelle Simulationsdaten

Zunächst werden in Definition 6.12 die Simulationsdaten des gerade aktiven Modus beschrieben.

Dabei gibt *num* die Nummer der aktuellen Simulation an, also wie oft bereits eine Simulation eines Modus gestartet wurde.

Das Attribut *solver* gibt den Solver für den aktiven Modus an.

start_time beschreibt die Zeit, bei der die Simulation des gerade aktiven Modus begonnen hat, und *time* die aktuelle Simulationszeit.

Definition 6.12 (Aktuelle Simulationsdaten):

CURRENT_DATA

`Nummer der aktuellen Simulation`

$num : \mathbb{N}$

`Aktueller Solver`

$solver : SOLVER$

`Startzeit der Simulation`

$start\_time : TIME$

`Aktuelle Zeit der Simulation`

$time : TIME$

Das Simulationsmodell enthält diese aktuellen Simulationsdaten. Die Definition 6.13 zeigt die Formalisierung des Simulationsmodells.

Im Simulationsmodell gibt es Eigenschaften, die während der Simulation konstant bleiben, und Eigenschaften, die sich explizit durch Methoden oder implizit durch Invarianten ändern.

Konstante Eigenschaften
Die konstanten Eigenschaften sind die Eigenschaften vor dem Δ in der Definition 6.13.

Die Initialdaten, welche den gesamten ersten Modus initialisieren, werden im *init_state* angegeben (A3).

Die Stoppzeit (*stop*) der Simulation wird ebenso angegeben.

Um die Simulationsdaten sinnvoll zu speichern, kann angegeben werden, welche Daten in jedem Modus gespeichert werden sollen und wo diese in den einzelnen Modi zu finden sind (*synonym*). Es werden nur die Daten gespeichert, die in den Modi enthalten sind. Ist für einen Modus kein Synonym einer Beobachtervariablen angegeben, so werden für diese Beobachtervariable keine Daten gespeichert (A26, A27, A28). Eine Definition der Synonyme kann für ein Modell mit zwei Modi beispielsweise wie folgt aussehen:

$$synonym = \left\{ \begin{array}{lcl} (x & \mapsto & \{(modus1 \mapsto s), (modus2 \mapsto weg)\}), \\ (y & \mapsto & \{(modus1 \mapsto h), (modus2 \mapsto hoehe)\}), \\ (v & \mapsto & \{(modus2 \mapsto geschw)\}), \end{array} \right\}$$

Für x und y sind für beide Modi Synonyme angegeben, für v gibt es jedoch nur im zweiten Modus ein Synonym (A29).

Variable Eigenschaften
Das Simulationsmodell ist eine dynamische Komponente, die Eigenschaften dieser Komponente (*vars*, *eqs*, etc.) ändern sich implizit durch den aktiven Modus.

Die aktuellen Simulationsdaten (*current*) sind vom gerade simulierten Modus abhängig und werden bei jedem Moduswechsel angepasst, siehe Definition 6.15.

Im Beobachter (*observer*) sollen die definierten Daten aus *synonym* nach jeder Simulation eines Modus gespeichert werden. In *states* wird der komplette Zustand des Modus hinter seiner Simulationsnummer gespeichert. So sind alle vergangenen Daten im Modell gespeichert (vgl. Abschnitt 6.1.4).

Invarianten: Der aktuelle Zustand *state* enthält nur Daten aus der aktuellen Simulation. Der aktuelle Zustand enthält nur Daten, deren Simulationszeit größer oder gleich der Startzeit der aktuellen Simulation ist. Dabei repräsentiert *state* einen Teilzustand aus *act.state*. In *act.state* können auch Daten von vergangenen Simulationen des Modus enthalten sein.

Der aktuelle Solver entspricht jeweils dem Solver des gerade aktiven Modus. Jeder Modus hat somit seinen eigenen Solver, der sich aus der Komposition der Komponenten ergibt.

In der Variable *observer* werden alle Daten gespeichert, die in *synonym* angegeben wurden.

Die angegebene Initialisierung (*init_state*) muss Startwerte für den initialen Modus bereitstellen.

Definition 6.13 (SIMULATION_MODEL):

SIMULATION_MODEL

Feste Daten für das Modell

Initialwerte

$init\_state : STATE$

Stoppzeit der Simulation

$stop : TIME$

Synonyme für Beobachter

$synonym : VARNAME \pfun (modes \pfun VARNAME)$

Veränderliche Variablen während der Simulation (auch durch Invarianten)

Δ

Alles aus DYNAMIC_COMPONENT

DYNAMIC_COMPONENT

Aktuelle Simulationsdaten

$current : CURRENT\_DATA$

Zusammenfassung aller Zustände

$states : \mathbb{N} \fun (ID \times STATE)$

Beobachter

$observer : STATE$

Alle Zeiten in state sind größer gleich der Startzeit der aktuellen Simulation

$$\forall\, v : \mathbf{dom}(state) \bullet \forall\, t : \mathbf{dom}(state(v)) \bullet t \geq current.start\_time \land$$

und kleiner gleich der Stoppzeit

$$t \leq stop$$

Jeder state ist Teil des states von act

$$\forall\, v : \mathbf{dom}(state) \bullet state(v) \subseteq act.state(v)$$

Der aktuelle Solver ist der Solver des aktiven Modus

$$current.solver = act.solver$$

Beobachter enthält genau die Daten aus synonym

$$\mathbf{dom}(observer) = \mathbf{dom}(synonym)$$

init_state ist eine Menge aus start

$$\mathbf{dom}(init\_state) \in initMode.start$$

6.1.4. Beschreibung der Simulation

Das vorher spezifizierte Modell soll nun simuliert werden. Dabei wird der dynamische Aspekt betrachtet, indem die Initialisierung, die Moduswechsel und das Ende der Simulation spezifiziert werden.

Während der Simulation werden die Gleichungen durch einen numerischen Solver gelöst. Die berechneten Werte werden in den Zustand (*state*) des gerade aktiven Modus geschrieben. Dies wird im Folgenden nicht weiter spezifiziert, sondern angenommen, da dies prinzipiell einer normalen Simulation entspricht.

Im Folgenden geht es um die dynamischen Vorgänge, die für strukturvariable Modelle zusätzlich notwendig sind. Diese Vorgänge basieren auf den Eigenschaften, die im Simulationsmodell beschrieben wurden.

Es werden drei Funktionen (*INIT*, *Transition_active*, *End_Simulation*) definiert, welche zum Simulationsmodell gehören und die Werte der Attribute verändern können.

Initialisierung

Die erste Funktion beschreibt die Initialisierung des Modells. Während der Initialisierung sind noch keine Simulationsergebnisse im Zustand (*states*) enthalten.

Der aktive Modus ist der initiale Modus und die Startzeit der Simulation ist gleich der aktuellen Simulationszeit.

In den Zustand *state* des Modells werden die Startwerte geschrieben. Sofern keine Daten für eine Variable angegeben sind, werden diese durch die abstrakte Funktion *INIT_CALC* initialisiert. Diese abstrahiert eine werkzeuginterne Initialwertberechnung, wie sie beispielsweise in Dymola integriert ist. Bei Simulink-Modellen hingegen müssten alle Startwerte angegeben werden, da eine solche Startwertberechnung nicht existiert.

Die aktuelle Simulationsnummer wird auf Eins gesetzt, es beginnt die Simulation des ersten Modus.

Definition 6.14 (Funktion – Initialisierung):

INIT

SIMULATION_MODEL

`Sammlung von Zuständen`

$states = \{\}$

`Erster Modus ist initialer Modus, aktuelle Zeit ist Startzeit`

$act = initMode \land current.time = current.start\_time$

`Initialdaten sind in state enthalten`

$\forall\, s : \mathbf{dom}(init\_state) \bullet state(s) = init\_state(s)$

`Initialisierung der Daten, die nicht explizit initialisiert werden`

$\forall\, v : vars | v \notin \mathbf{dom}(init\_state) \bullet$
$state(v) = \boldsymbol{INIT\_CALC}$

`Erste Simulation wird gestartet`

$current.num = 1$

Moduswechsel

Die zweite Funktion, vgl. Definition 6.15, beschreibt den Moduswechsel. Als Vorbedingung dieser Funktion muss gelten, dass eine Wechselbedingung einer Transition des aktuellen Modus *true* ist. Daraufhin wird der Modus aktualisiert. Dabei ändern sich die Eigenschaften der Komponente (*vars*, *eqs*, etc.) implizit durch die Invarianten.

Anhand der gefundenen Transition wird die Initialisierung durchgeführt. Dafür wird die Variable *state* des Modells angepasst, indem für alle Startwerte, die in der Transition definiert wurden, die Daten an die entsprechende Stelle in *state* geschrieben werden. Für alle nicht definierten Werte wird die Funktion *INIT_CALC* verwendet.

Die aktuelle Simulationszeit *current.time* und die Startzeit der nächsten Simulation *current.start_time* werden anhand der Zeit, die in der Initialisierung angeben ist, initialisiert.

Die Daten in *states* werden um die simulierten Werte erweitert und die Beobachterdaten werden im *observer* gespeichert. Dabei werden nur Daten für eine Beobachtervariable gespeichert, sofern ein Synonym angegeben wurde (A29).

Die Simulationsnummer wird um Eins erhöht und der Solver für die nächste Simulation wird gesetzt. Nach dieser Funktion wird die Aktion der Transition ausgeführt, falls eine angegeben ist. Diese kann die zuvor gesetzten Daten überschreiben. Dies wird hier nicht formalisiert. Diese Aktion kann alle Daten des Simulationsmodells verwenden und anpassen.

Definition 6.15 (Funktion – Transition):

Fct: Transition_active

SIMULATION_MODEL

$$\Delta(act, states, current\_data, state)$$

`Guard, der aus aktuellem Modus hinausführt, wird aktiv`

$$\exists\, t : trans \bullet t.pre = act \wedge t.guard(t.pre.state) \wedge$$

`Modus aktualisieren`

$$act' = t.post \wedge$$

`Initialisierung in state', Startzeit und aktuelle Zeit wird gesetzt`

$$(\textbf{let}\ s == t.init(t.pre.state, t.post.state) \bullet$$
$$\forall\, v : \textbf{dom}(s) \bullet s(v) = state'(v) \wedge$$
$$current'.start\_time \in \textbf{dom}(state'(v)) \wedge$$
$$current'.time = current'.start\_time \wedge$$

`Initialisierung der Daten, die nicht explizit initialisiert werden`

$$\forall\, v : act'.vars | v \notin \textbf{dom}(s) \bullet$$
$$state'(v) = \textbf{\textit{INIT\_CALC}})$$

`Der alte Zustand wird gespeichert`

$$states' = states \cup \{current.num \mapsto (act.id \mapsto state)\}$$

`Daten des Beobachters speichern`

$$\forall\, v : \textbf{dom}(observer) | (\exists\, v_a : vars \bullet synonym(v)(act) = v_a) \bullet$$
$$observer(v)' = observer(v) \cup state(v_a)$$
$$\forall\, v : \textbf{dom}(observer) | (\nexists v_a : vars \bullet synonym(v)(act) = v_a) \bullet$$
$$observer(v)' = observer(v)$$

`Aktuelle Simulationsnummer ändern`

$$current'.num = current.num + 1$$

Simulationsende

Die letzte Funktion betrifft das Ende der Gesamtsimulation. Die Vorbedingung dieser Transition ist erfüllt, wenn die Stoppzeit erreicht ist.

Es werden die Simulationsergebnisse des aktuellen Modus in das Attribut *states* geschrieben.

Ebenso wird der Beobachter aktualisiert. Daraufhin ist die Simulation beendet.

Definition 6.16 (Funktion – Simulationsende):

Fct: End_Simulation

SIMULATION_MODEL

`Stoppzeit erreicht`

$$current.time \geq stop$$

`Der alte Zustand wird gespeichert`

$$states' = states \cup \{current.num \mapsto (act.id \mapsto state)\}$$

`Daten des Beobachters speichern`

$$\forall\, v : \mathbf{dom}(observer) | (\exists\, v_a : vars \bullet synonym(v)(act) = v_a) \bullet observer(v)' = observer(v) \cup state(v_a)$$

$$\forall\, v : \mathbf{dom}(observer) | (\nexists v_a : vars \bullet synonym(v)(act) = v_a) \bullet observer(v)' = observer(v)$$

6.1.5. Fazit

Die vorgestellte Formalisierung konkretisiert zum einen die graphische Notation und beschreibt zum anderen formal die wichtigsten Eigenschaften strukturvariabler Modelle. Sie ist unabhängig von einer konkreten Modellierungssprache und kann für modular und hierarchisch aufgebaute strukturvariable Modelle verwendet werden. Ein Modell, das die in der Formalisierung spezifizierten Eigenschaften enthält und die Invarianten erfüllt, kann simuliert werden.

Die Simulation eines strukturvariablen Modells wurde mit Hilfe von Funktionen beschrieben. Diese beschreiben die dynamischen Abläufe während der Simulation.

Mit der Formalisierung ist eine Basis zum Beschreiben und Simulieren strukturvariabler Modelle gelegt. Sie kann damit als Grundlage für eine Implementierung eines Simulationswerkzeugs genutzt werden.

6.2. Simulation

Nachdem ein Modell entworfen und in einer geeigneten Notation dargestellt wurde, muss es simuliert werden. Dabei geht es um Modelle, wie sie in Definition 6.13 vorgestellt wurden.

Die neuen Herausforderungen, die bei der Simulation strukturvariabler Modelle auftreten, betreffen die Übergänge von einem Modus zum nächsten. Die Simulation eines einzelnen Modus ist identisch zur normalen Simulation eines Modells mit Ereigniserkennung.

Im Folgenden wird erklärt, welche Anforderungen ein Simulationswerkzeug aufweisen muss, um für strukturvariable Modelle geeignet zu sein.

Ausfaktorisieren der korrekten Modi

Ein Simulationswerkzeug für strukturvariable Modelle sollte dem Modellierer das Ausfaktorisieren der Modi, wie es in Abschnitt 6.1.2 beschrieben wurde, abnehmen. Dabei sollte ein solches Werkzeug darauf hinweisen, wenn Schnittstellen inkompatibel sind oder wenn die Initialisierung nicht eindeutig ist.

In einem Simulationswerkzeug ist es zudem nicht notwendig, zu Beginn einer Simulation das gesamte Modell auszufaktorisieren. Erst wenn eine neue Kombination von Komponenten notwendig ist, sollte diese zusammengesetzt werden. Besonders bei einer hohen Vielfalt an Kombinationen wäre sonst der initiale Overhead hoch; außerdem werden oft nicht alle Kombinationen während der Simulation benötigt.

Zur Simulationszeit sollte das Werkzeug zunächst den initialen Modus zusammensetzen, falls dieser aus verschiedenen Komponenten mit Modi besteht, und ihn kompilieren. Tritt ein Moduswechsel ein, so sollte wiederum der neue Modus aus den entsprechenden Komponenten zusammengesetzt und kompiliert werden, usw.

Handhabung von Ereignissen

Zur Simulation von strukturvariablen Modellen ist es notwendig, Ereignisse zu detektieren. Das Ereignis entspricht dabei einer Wechselbedingung einer Transition. Viele gängige Simulationsumgebungen wie Dymola und Simulink können mit Ereignissen umgehen. Jedoch ist in der Modelica-Spezifikation nicht eindeutig festgehalten, wie die Ereignisse gehandhabt werden. In unterschiedlichen Werkzeugen können Ereignisse zu unterschiedlichen Zeiten eintreten.

Um ein Ereignis zu detektieren, wird in vielen Simulationswerkzeugen eine *zero-crossing detection* vorgenommen, siehe Abbildung 6.9. Um den Zeitpunkt des Ereignisses während eines Zeitschrittes zu detektieren, kann beispielsweise das Newton-Iterationsverfahren oder die Goldene-Schnitt-Methode (golden section) [15] verwendet werden. Das Finden des genauen Zeitpunktes benötigt Zeit.

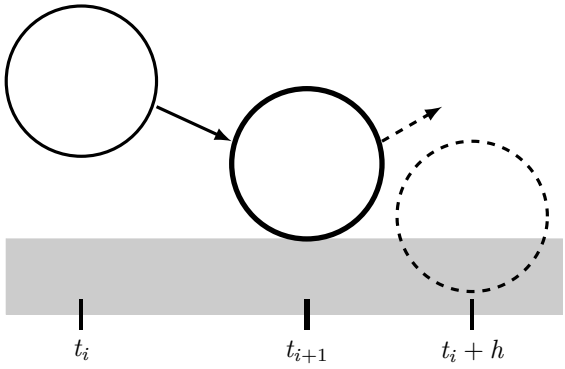


Abb. 6.9.: Finden des genauen Zeitpunkts eines Events

Werden zwei Ereignisse für einen Moduswechsel während des gleichen Zeitschrittes detektiert, was nicht bedeutet, dass sie zur gleichen Zeit eintreten, ist die Frage, wie mit ihnen verfahren werden soll. Dabei ist zwischen asynchronen und synchronen Ereignissen zu unterscheiden.

Die asynchronen Ereignisse sind die einfacheren, da sie nicht kausal zusammen hängen. Die Auswertungsreihenfolge ist damit nicht maßgeblich für die Korrektheit der Ergebnisse.

Die synchronen Ereignisse hingegen haben einen kausalen Zusammenhang und es ist relevant, in welcher Reihenfolge sie eintreten, da eine falsche Reihenfolge zu falschen Ergebnissen führt. Werden zwei Ereignisse in einem Zeitschritt detektiert, so können beide Transitionen aktiviert werden oder es wird erkannt, welches der Ereignisse das erste war, und die entsprechende Transition wird aktiviert. Daraufhin startet die Simulation erneut und kurz darauf tritt das andere Ereignis ein. Eventuell tritt das nächste Ereignis aber auch nicht mehr ein, was zu einer falschen Konstellation führen würde.

Auch verschiedene Werkzeuge handhaben das Finden mehrerer Ereignisse anders, was bei der Modellierung strukturvariabler Modelle beachtet werden sollte.

Da es sich bei der Suche nach dem exakten Ereigniseintritt und bei dem Modell nur um eine Näherung handelt, stimmen die simulierten Ereignisse eventuell nicht zeitlich mit den realen Daten überein. Die Gefahr besteht darin, einen Modus zu lange oder nicht lange genug zu verwenden und damit den nachfolgenden Modus zu einer falschen Zeit und mit falschen Werten zu starten. Im Extremfall kann sich dieser Fehler fortsetzen und die gesamte Simulation beeinflussen.

Eine gute Handhabung von Ereignissen ist aus diesem Grund dringend erforderlich. Der Modellierer sollte sich auch mit der Handhabung der Ereignisse vertraut machen, um diese korrekt zu verwenden und die Strukturwechsel entsprechend zu beschreiben.

Auch sollte im Simulationswerkzeug angegeben werden können, wie mit gleichzeitigen Moduswechseln umgegangen werden soll.

Transitionen auswerten

Wird eine Wechselbedingung innerhalb eines Modus detektiert, so muss die Transition zum neuen Modus, die Initialisierung und Aktion durchgeführt werden. Dabei kann, wie bekannt, die Aktion auch eine Aktivierung einer weiteren Transition fordern oder ein erneutes Kompilieren.

Initialisierung

Die bei der Modellierung spezifizierte Initialisierung muss angewendet werden. Da jedes Simulationswerkzeug eine Initialisierung von Modellen zulässt, kann auch bei einem Moduswechsel eine erneute Initialisierung durchgeführt werden.

Die Schwierigkeit liegt darin, dass erneut ein Set konsistenter Startwerte anhand der gewünschten Vorgaben (vgl. Abschnitt 5.2.3) gefunden werden muss. Ein Simulationswerkzeug für strukturvariable Modelle sollte bereits bei der Definition der Transitionen helfen, die korrekten Startwerte zu initialisieren. Dabei sollten direkt bei der Modellierung Informationen zur Initialisierung beschrieben werden können. Spätestens nach dem Kompilieren eines Modells ist bekannt, welche Variablen initialisiert werden sollten; diese sollte ein Werkzeug ausgeben können.

Ebenso kann das Lösen des Initialwertproblems viel Zeit in Anspruch nehmen und teilweise zu Sprüngen in der Lösung führen.

Findet der Moduswechsel in einem stationären Zustand (steady-state) statt, ist die Wahrscheinlichkeit hoch, dass das Startwertproblem gut lösbar ist. Die notwendigen Startwerte müssen dafür aus dem alten Modus extrahierbar sein.

Um ein Zurückgehen in die Vergangenheit zu realisieren, sollte das Simulationswerkzeug mindestens die Daten automatisch speichern, die in Transitionen zum Initialisieren benötigt werden. In Modelica-Werkzeugen werden meist alle Simulationsdaten gespeichert, wodurch diese Anforderung erfüllt ist. In Simulink werden Daten jedoch nur gespeichert, wenn dies definiert wurde. Wird ein solches Werkzeug für strukturvariable Modelle erweitert, sollte es automatisiert die notwendigen Daten für die Transition speichern.

Neuer Modus

Wird ein neuer Modus aktiviert, so muss dieser vorverarbeitet werden. Es ist also ein Flattening notwendig und eventuell auch eine erneute Kausalisierung und Indexreduktion. Dabei sollte das Simulationswerkzeug möglichst nur die notwendigen Teile des Modells diesem Prozess unterziehen, wie es auch in [96] vorgesehen wurde. Ebenfalls sollten Modi, die schon simuliert wurden, bei Bedarf wiederverwendet werden.

Ein weiterer Ansatz ist, die einzelne Komponente separat zu übersetzen und in ihren verschiedenen kausalisierten Zuständen zu speichern [40]. Wenn eine bestimmte Komponentenkonstellation benötigt wird, können die bereits kompilierten Varianten zusammengesetzt werden. Es muss somit bei einem Moduswechsel nicht erneut kompiliert, sondern lediglich das Modell aus entsprechenden Komponenten zusammengesetzt werden. Dies wäre ein großer Vorteil für die Simulation von strukturvariablen Modellen. Der Ansatz ist gut, um ein erneutes Kompilieren bei objektorientierten Modellierungssprachen wie Modelica zu verhindern.

Solver

Zwischen zwei Simulationen kann bei strukturvariablen Modellen der Solver geändert werden. Dies ist beispielsweise sinnvoll, wenn es sich bei einem Modus um ein steifes Modell handelt und beim anderen Modus nicht. Dies kann zu Vorteilen in der Stabilität, der Geschwindigkeit und Genauigkeit führen.

Auch wenn der Solver gleich bleibt, besteht die Schwierigkeit beim Neustart einer Simulation darin, dass der Solver ebenfalls „neu gestartet“ wird und somit vergangene Informationen verloren gehen. Dies ist relevant, wenn ein Mehrschrittverfahren verwendet wird, welches vergangene Werte bei der Berechnung berücksichtigt. Ein Mehrschrittverfahren beginnt bei einer neuen Simulation entweder mit einem Einschrittverfahren oder erhöht die Ordnung des Verfahrens in jedem Schritt, um die notwendigen Daten für die Vergangenheit zu bestimmen.

Sobald ein Moduswechsel eintritt, können die Daten aus der Vergangenheit jedoch nicht ohne weiteres weiterverwendet werden, da:

- sich das Zeitverhalten einer Variable so ändert, dass die vergangenen Werte die Zukunft eher verfälschen,
- es manche Variablen vorher gar nicht gab,
- oder die Konstellation von vergangenen Werten nicht mehr mit dem gültigen Gleichungssystem übereinstimmt.

Ebenso wäre es sinnvoll, wie auch bereits in der Formalisierung vorgesehen, die Solver bereits in den Komponenten vorzusehen. Die Beschreibung der Komponenten wäre damit allgemeiner und es könnten hilfreiche Informationen bereits lokal hinzugefügt werden.

6.3. Zusammenfassung

Die Eigenschaften eines strukturvariablen Modells wurden formal zusammengefasst, um als Basis zur Beschreibung strukturvariabler Modelle zu dienen. Dabei stellt die Formalisierung alle notwendigen Eigenschaften strukturvariabler Modelle in Beziehung und zeigt, wie ein Ausfaktorisieren von Modi in Komponenten stattfinden und was bei der Auslegung von Schnittstellen und Transitionen beachtet werden muss. Ebenso kann die Formalisierung als Basis für eine Implementierung von Simulationswerkzeugen für strukturvariable Modelle verwendet werden.

Die Anforderungen, die an ein Simulationswerkzeug für strukturvariable Modelle gestellt werden, bestehen aus praktischer Sicht im Wesentlichen aus folgenden Punkten:

Aus Benutzersicht sollte das Ausfaktorisieren und Initialisieren möglichst durch ein Werkzeug unterstützt werden. Außerdem sollte eine graphische Notation im Werkzeug enthalten sein, welche ein Beschreiben der strukturvariablen Eigenschaften zulässt.

Auf der Simulationsseite muss ein Werkzeug eine gute Ereigniserkennung mit Auswertungsmöglichkeiten für Transitionen bereitstellen. Zusätzlich wären eine *just-in-time-compilation* und die separate Übersetzung der Komponenten von Vorteil.

7 Prototypisches Framework *DySMo*

In diesem Kapitel wird ein Werkzeug zur Beschreibung und Simulation strukturvariabler Modelle vorgestellt.

Dazu wird zunächst erläutert, wie eine zentrale Ablaufsteuerung zur Integration verschiedener Simulationswerkzeuge umgesetzt werden kann. Die Vor- und Nachteile eines solchen Vorgehens werden dabei diskutiert.

Daraufhin wird eine Softwarearchitektur für ein prototypisches Werkzeug *DySMo* (**Dy**namic **S**tructure **Mo**deling) vorgestellt. Diese Softwarearchitektur basiert auf der Formalisierung des vorherigen Kapitels. DySMo stellt Schnittstellen zu gängigen Simulationswerkzeugen zur Verfügung, sodass die Modi eines strukturvariablen Modells in den Simulationswerkzeugen simuliert werden können, während DySMo die Koordination der Modi übernimmt.

Anhand eines Modells für einen Satellitenstart mit drei Modi wird erklärt, wie DySMo verwendet werden kann.

Zum Abschluss wird DySMo auf Effizienz, Erweiterbarkeit, Handhabung und Einschränkungen hin evaluiert.

7.1. Zentrale Ablaufsteuerung zur Simulation von strukturvariablen Modellen

Wie bereits erläutert, gibt es zwei grobe Vorgehensweisen für den Umgang mit strukturvariablen Modellen. Es kann eine Meta-Ebene eingeführt oder eine eigene Sprache entwickelt werden. Da es hier um die Untersuchung strukturvariabler Modelle geht und eine möglichst große Flexibilität erreicht werden soll, wird eine Meta-Ebene eingeführt. Zunächst wird auf die allgemeine Vorgehensweise für solch eine Meta-Ebene eingegangen, bevor der Softwareentwurf für DySMo erläutert wird.

Ein Simulationswerkzeug muss eine Skriptsprache zur Steuerung der Simulation unterstützen. Viele heute gängige Simulationswerkzeuge erfüllen diese Voraussetzung, wie weiter unten anhand von Beispielen gezeigt wird.

Für den vorgestellten Ansatz müssen die Modi ausfaktorisiert sein. Dies bedeutet einen initialen Mehraufwand für den Modellierer, stellt jedoch auch gleichzeitig sicher, dass die Modi alle simulierbar sind.

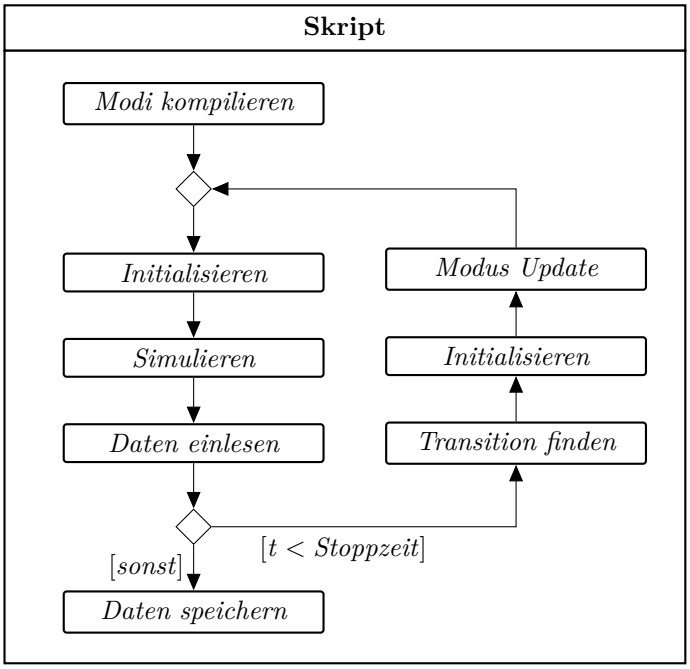


Abb. 7.1.: Ablauf eines Skriptes zur Simulation von Moduswechseln

Abbildung 7.1 zeigt einen sequentiellen Ablauf eines Skriptes, welches Strukturwechsel ermöglicht (ähnlich zu [6, 74]). Mit dieser Vorgehensweise wird ein Wechsel zwischen beliebig vielen Modi realisiert. Zu Beginn des Skriptes werden die einzelnen Modi im Simulationswerkzeug kompiliert. Dieses Kompilieren ist nicht für alle Modi notwendig, ist jedoch beispielsweise für Modelica-Modelle erforderlich. Es wird daraufhin eine *while*-Schleife betreten. In dieser Schleife werden zunächst die Startwerte des aktiven Modus gesetzt und danach die Simulation gestartet. Wenn die Simulation durch eine definierte Wechselbedingung einer Transition beendet wurde, werden die Endwerte der Simulation gelesen. Diese können für die Initialisierung des nächsten Modus verwendet werden. Der aktive Modus wird nun auf den neuen Modus gesetzt und die Schleife beginnt erneut. Ist die Stoppzeit erreicht, wird die Schleife beendet und die Simulationsergebnisse werden gespeichert.

Es ist bei diesem Ansatz irrelevant, wie komplex die Modelle der einzelnen Modi sind, solange sie im jeweiligen Simulationswerkzeug simuliert werden können. Das Skript startet die Simulation des aktuellen Modus und wartet, bis die Simulation beendet ist. Vorhandene Modelle können mit diesem Ansatz wiederverwendet werden, ohne sie grundlegend ändern zu müssen. Auch können existierende Werkzeuge zur Simulation verwendet und so deren Vorteile genutzt werden.

Im Folgenden wird dieser skriptbasierte Ansatz für mehrere Simulationswerkzeuge vorgestellt. Die vorgestellten Varianten werden am Ende verglichen und deren jeweiligen Vor- und Nachteile erörtert.

Zur Illustration wird das Modell des Satellitenstarts aus Beispiel 3.1 wiederverwendet.

Matlab/Simulink

Wird Simulink als Simulationswerkzeug verwendet, kann ein Skript zur Ablaufsteuerung des strukturvariablen Modells in der Sprache von Matlab geschrieben werden. Matlab bietet alles Notwendige, um Simulink-Modelle zu initialisieren, zu simulieren und die Simulationsdaten zu lesen und zu verarbeiten.

In Simulink müssen zunächst drei separate Modelle erstellt werden, welche die Modi aus Beispiel 3.1 repräsentieren. Dafür werden drei separate MDL-Dateien erzeugt, die jeweils einen Modus enthalten. Die Stopp-Bedingung wird durch entsprechende Simulink-Blöcke abgefragt. Das Ergebnis dieser Abfrage entscheidet, ob die Simulation durch einen *Stop-Simulation*-Block beendet wird. Dieses Beenden der Simulation ist kein fehlerhaftes Abbrechen, sondern eine Möglichkeit, die Simulation fehlerfrei zu beenden, falls keine konkrete Stoppzeit bekannt ist. Abbildung 7.2 zeigt den Modus der Rakete aufbereitet als Simulink-Modell. Die Komponenten (Subsysteme) enthalten dabei jeweils die mathematischen Gleichungen, während auf der obersten Ebene die Stoppbedingung realisiert ist.

Das Skript zur Simulation des Modells wird in Matlab implementiert, siehe Listing 7.1. Zunächst wird in Zeile 1 die Simulation des ersten Modus gestartet (*sim*). Dabei wird der Modellname und die gewünschte Stoppzeit der Simulation übergeben. Zusätzliche Informationen, wie die Startzeit und Solver-Informationen können optional übergeben werden. Werden diese nicht übergeben, werden die im Modell gesetzten Einstellungen verwendet.

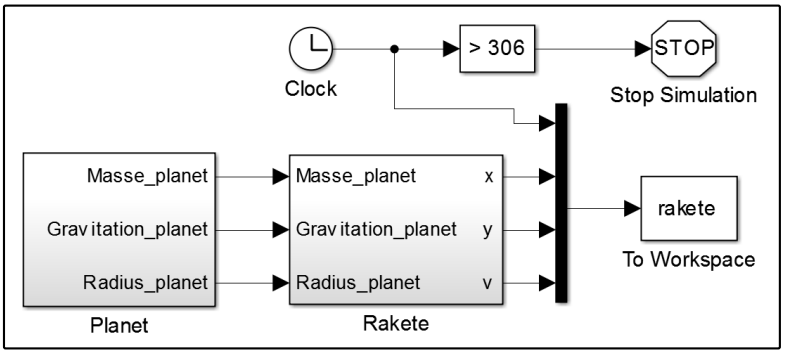


Abb. 7.2.: Simulink-Modell der Rakete

In der Variable *rakete* sind die Simulationsdaten aus dem *To Workspace-Block* gespeichert, welche in den Zeilen 2 – 5 gelesen werden. Diese Daten können zum einen zum Darstellen der Flugbahn verwendet werden und zum anderen, um den nächsten Modus zu initialisieren. In Zeile 14 wird die Simulation des Satelliten gestartet. Die Initialisierung findet dabei mit den im Workspace gespeicherten Werten von 'x', 'y' und 'v' statt. Nach dem Beenden der Simulation des Modus werden die Simulationsergebnisse gelesen und der dritte Modus gestartet.

Dieses Beispielskript zeigt, wie einfach es ist, mit dem skriptbasierten Ansatz ein strukturvariables Modell in Simulink zu realisieren.

```
1  sim('raktete', 'stopTime', '500000');
2  t = num2str(rakete(end,1));
3  x = rakete(end,2);
4  y = rakete(end,3);
5  v = rakete(end,4);
6
7  sim('satellit_no_prop','startTime',t,'stopTime','500000');
8  t = num2str(satellit(end,1));
9  x = satellit(end,2);
10 y = satellit(end,3);
11 vx = satellit(end,4);
12 vy = satellit(end,5);
13
14 sim('satellit_prop','startTime',t,'stopTime','500000');
15 ...
```

Listing 7.1: Matlab-Skript zur Simulation des Satellitenmodells in Simulink

Dymola

In Dymola gibt es zwei Varianten, ein Skript für die Simulation eines strukturvariablen Modells umzusetzen. Zum einen kann die Skriptsprache von Modelica verwendet werden. Diese wird in MOS-Dateien gespeichert. Die zweite Variante besteht darin, Dymola durch ein externes Programm zu steuern, zum Beispiel durch Matlab oder Python. Beide Varianten werden im Folgenden mit dem Satellitenstart-Beispiel vorgestellt.

Zunächst müssen die drei Modelle erstellt werden. Ein Modell für die Rakete und je ein Modell für die unterschiedlichen Satelliten. Diese Modelle müssen wiederum separat lauffähig sein. Damit die Simulation zur korrekten Zeit abgebrochen wird, werden *when-Events* benutzt. Sobald die Wechselbedingung eintritt, wird mit dem Befehl *terminate* die Simulation beendet. Die Simulationsergebnisse werden daraufhin gespeichert. Listing 7.2 zeigt das Satellitenmodell ohne Antrieb, wie es aus Listing 2.1 bekannt ist, jedoch ohne initiales Gleichungssystem. Dieses gilt hier nicht mehr, da der Satellit nicht mehr auf einer Kreisbahn fliegen soll. Listing 7.3 zeigt eine Erweiterung des Modells, das nun eine Wechselbedingung mit einem Simulationsabbruch enthält.

Das Gesamtmodell für den Satelliten setzt sich aus dem Planeten und dem Satelliten zusammen, vgl. Listing 2.3

```
1  model satellit
2   parameter Real M;
3   parameter Real G;
4   Real F;
5   Real a, ax, ay;
6   Real vx, vy;
7   Real x, y;
8   Real r;
9  equation
10  F = -G * M/(r*r);
11  a = F;
12  ax = a*x/r;
13  ay = a*y/r;
14  der(vx) = ax;
15  der(vy) = ay;
16  der(x) = vx;
17  der(y) = vy;
18  r^2 = x^2 + y^2;
19 end satellit;
```

Listing 7.2: Modelica-Satellit

```
1  model sat_no_prop
2   extends satellit
3
4   Integer loops;
5   Integer switch_to;
6
7  equation
8    when x <= 0 then
9     loops = pre(loops)+1;
10   end when;
11
12   when (loops == 20) then
13    terminate("loops");
14   end when;
15
16 end sat_no_prop;
```

Listing 7.3: Modelica-Satellit mit Stoppbedingung

MOS-Dateien sind Modelica spezifische Skripte, mit denen Modelica-Modelle simuliert werden können. Zusätzlich können Berechnungen durchgeführt werden. Dymola kann diese MOS-Dateien einlesen und die Befehle sequentiell abarbeiten. Listing 7.4 zeigt den Anfang des MOS-Skripts für den Satellitenstart.

Der Befehl *simulateModel* bzw. *simulateExtendedModel* startet die Simulation des jeweiligen Modells (Zeilen 1, 5). Dazu werden der Name des Modells, die Start- und Stopp-Zeit, die Solvereinstellungen und der vorher definierte Name der Ergebnisdatei übergeben. Bei der ersten Simulation wird *simulateModel* verwendet, da keine Startwerte gesetzt werden. Bei der zweiten Simulation sollen definierte Startwerte verwendet werden. Deshalb wird hier der Befehl *simulateExtendedModel* genutzt, bei dem Startwerte übergeben werden können.

Nach einer Simulation werden die Simulationsergebnisse ausgelesen (Zeilen 2, 3, 6, 7) und zur Initialisierung des nächsten Modus verwendet (Zeile 5). Dies wird mit den anderen Modi ebenso gehandhabt.

```
1 simulateModel("satelliten.RaketenStart", stopTime =
      500000, resultFile= "resultR", method="dassl");
2   n=readTrajectorySize("resultR.mat");
3   xy=readTrajectory("resultR.mat",
      {"rakete.height","rakete.x","rakete.v","Time"}, n);
4
5 simulateExtendedModel("satelliten.PlanetSatellit",
      startTime = xy[4,n], stopTime = 500000, resultFile =
      "resultS", method="dassl",
      initialNames={"sat.y","sat.x","sat.vx"},
      initialValues=xy[1:3,n], finalNames = {});
6   n=readTrajectorySize("resultS.mat");
7   xy=readTrajectory("resultS.mat",
      {"sat.y","sat.x","sat.vy","sat.vx","Time"}, n);
8 ...
```

Listing 7.4: MOS-Skript zur Simulation des Satellitenmodells

Vor jeder Simulation eines Modelica-Modells muss dieses kompiliert werden. Dymola bietet bisher nicht die Möglichkeit, kompilierte Modelle zu laden und erneut zu verwenden.

Es kann lediglich das letzte kompilierte Modell wiederverwendet werden. Aus diesem Grund muss bei jedem Moduswechsel das benötigte Modell neu kompiliert werden.

Wird ein Modus zwei Mal benötigt, so muss er bei diesem Ansatz auch zwei Mal kompiliert werden, obwohl sich an dem Modus nichts ändert. Dies führt zu unnötigem Mehraufwand. Es wäre von Vorteil, wenn kompilierte Modelle gespeichert und wiederverwendet werden könnten.

Matlab-Skripte können ein unnötiges Kompilieren von Modelica-Modellen vermeiden. Dymola erzeugt beim Kompilieren eines Modells eine ausführbare Datei mit dem Namen *dymosim.exe* und eine Datei, welche die Initialwerte enthält. Die *dymosim.exe* kann aus Matlab heraus gestartet und das Modell durch Manipulation der Initialdatei initialisiert werden. Die Dymola-Installation stellt eine Vielzahl von Funktionen breit, die es ermöglichen Dymola zu steuern, eine Initialdatei zu schreiben und die Ergebnisdatei einzulesen.

Ein Auszug aus dem Matlab Skript für das Modell ist in Listing 7.5 gezeigt. Die Kommunikation zu Dymola findet per DDE-Channel statt, indem mit dem Befehl *dymolaM* ein MOS-Kommando gesendet wird.

Der erste Befehl betrifft das Öffnen der Modelica-Datei, die die entsprechenden Modelle enthält (Zeile 2). Danach wird der Pfad in Dymola umgesetzt (Zeile 5), damit die *dymosim.exe* beim Kompilieren in diesem Ordner gespeichert wird (Zeile 6). Danach kann aus diesem Verzeichnis die Initialdatei eingelesen werden (Zeile 7), die Initialdaten werden dabei in lokalen Variablen gespeichert. Dies wird für alle drei Modi gemacht.

Nun muss auch Matlab zu dem Pfad wechseln, in dem die *dymosim.exe* gespeichert ist (Zeile 10), woraufhin die Simulation gestartet werden kann (Zeile 16). Dabei werden diesem Aufruf Solver-Informationen und je ein Array mit Parameter- und Startwerten übergeben. Der aufgerufene Befehl gibt nach dem Beenden der Simulation die gesamten Simulationsdaten zurück.

Diese werden in den Zeilen 19 – 21 verwendet, um die Startwerte des zweiten Modus zu setzen. Daraufhin kann der zweite Modus simuliert werden, dessen Enddaten dann ebenfalls wieder zum Start des nächsten Modus verwendet werden.

Mit diesem Ansatz ist es möglich, jedes Modell nur einmal zu kompilieren.

Als Alternative zu Matlab können auch andere Sprachen verwendet werden, so beispielsweise Scilab [75] oder auch Python. Ein Interface zu Dymola muss dort jedoch selbst implementiert werden.

```
1  % Modell öffnen
2  dymolaM('openModel("satellit.mo");');
3
4  % Modus 1 kompilieren
5  dymolaM(['cd(rakete);']);
6  dymolaM('translateModel("satellit.rakete");');
7  [p1,x01,pnames1,x0names1] = loaddsin('dsin.txt');
8
9  % Modus 2 kompilieren
10 cd(satellitPath);
11 dymolaM(['cd(satellitNoProp);']);
12 dymolaM('translateModel("satellit.PlanetSatellit");');
13 [p2,x02,pnames2,x0names2] = loaddsin('dsin.txt');
14 ...
15 cd(raketePath); % erste Modus
16 [val,names1] = dymosim([0,500000,3e-05,3e-05,8],
       x01,p1,1);
17
18 % Auslesen und Setzen der Startwerte
19      x02(1) = val(end,myStrMatch('rakete.x',names1));
20      x02(2) = val(end,myStrMatch('rakete.height',names1));
21      x02(3) = val(end,myStrMatch('rakete.v',names1));
22
23 cd(satellitPath); % zweite Modus
24 [val2,names2] =
       dymosim([val(end,1),500000,3e-05,3e-05,8],x02,p2,1);
25 ...
```

Listing 7.5: Matlab-Skript zur Simulation des Satellitenmodells mit Dymola

OpenModelica

Das Vorgehen in OpenModelica ist sehr ähnlich zu dem in Dymola. Auch hier kann ein MOS-Skript verwendet werden, welches identisch zu dem in Dymola ist. Auch könnte OpenModelica durch eine Batch-Datei, Scilab, Python oder durch Matlab-Befehle gesteuert werden. Beispielsweise kann Matlab verwendet werden, um die OpenModelica-Shell aufzurufen und dort das Kompilieren der Modelle zu initiieren. Daraufhin erzeugt auch OpenModelica eine ausführbare Datei und eine Initialdatei. Diese Initialdatei ist ab OpenModelica 1.9 eine XML-Datei, die eingelesen und verändert werden kann. Das Skript wird hier nicht gezeigt, da es dem aus Listing 7.5 sehr ähnelt.

Vergleich und Evaluation des skriptbasierten Ansatzes für verschiedene Simulationswerkzeuge

Wie an den oben gezeigten Beispielen zu sehen ist, lässt sich der skriptbasierte Ansatz gut in verschiedenen Werkzeugen umsetzen. Besonders einfach wird der Ansatz, wenn das Simulationswerkzeug eine eigene Skriptsprache bereitstellt. Sehr gut ist der Ansatz in Matlab mit Simulink-Modellen verwendbar. Die Modelle können auf den Workspace zugreifen und nutzen damit aktuelle Daten.

Die Modelica-Skriptsprache, die in den meisten Werkzeugen für Modelica verwendet werden kann, ist ebenfalls gut für den Ansatz nutzbar. Die Befehle der Skriptsprache sind jedoch umständlich und lang. Durch diverse Testläufe wurde festgestellt, dass bei vielen Moduswechseln die MOS-Skripte viel Zeit in Anspruch nehmen. Bei kleinen Modellen mit vielen Moduswechseln ist die Zeit, die das Skript benötigt, länger als die reine Simulationszeit. Da kompilierte Modelle nicht wiederverwendet werden können, ist dieser Ansatz wenig für strukturvariable Modelle geeignet.

In Matlab können kompilierte Modelica-Modelle (Dymola und OpenModelica) gestartet und initialisiert werden. Daher ist es, wenn viele Moduswechsel notwendig sind, einfacher und zeiteffizienter Matlab zu verwenden. Auch andere Sprachen sind dafür verwendbar; OpenModelica und Dymola können beispielsweise durch Scilab oder Python gesteuert werden.

Der Nachteil ist, dass zwei Werkzeuge notwendig sind, um ein Modell zu simulieren: Ein Werkzeug zum Steuern und eins zum Durchführen der Simulation.

Der große Vorteil des skriptbasierten Ansatzes ist, dass vorhandene Modelle nicht neu implementiert werden müssen und bekannte Simulationswerkzeuge zur Simulation strukturvariabler Modelle verwendet werden können. Die Solver und Stärken der jeweiligen Werkzeuge können daher genutzt werden.

Ein Nachteil des skriptbasierten Ansatzes ist, dass jeder Modus auf der obersten Modellebene definiert sein muss und keine Modi in Komponenten möglich sind. Der Modellierer muss daher alle Modi als eigenständige Modelle vorbereiten.

Wären die Strukturwechsel direkt in die Sprache implementiert, könnten Modi in Komponenten vorgesehen werden. Auch wäre eine eigene Sprache in der Lage, den Moduswechsel werkzeugintern und damit schneller und effizienter zu handhaben.

Da es in dieser Arbeit darum geht zu untersuchen, wann strukturvariable Modelle sinnvoll angewendet werden können und sollen, ist der skriptbasierte Ansatz von Vorteil. Mit den Skripten ist es möglich, strukturvariable Modelle anhand von existierenden Modellen zu testen und zu untersuchen. Der Aufwand, große und realistische Modelle zu untersuchen, ist geringer, als er bei der Verwendung neuer Sprachen wäre. Im weiteren Verlauf dieses Kapitels soll daher gezeigt werden, wie der skriptbasierte Ansatz verallgemeinert werden kann und für verschiedene Werkzeuge einheitlich nutzbar ist. Die Strukturwechsel werden ausgelagert, was bedeutet, dass ein Modell aus verschiedenen Modi bestehen kann, welche wiederum jeweils in ihrem eigenen Simulationswerkzeug mit eigenem Solver simuliert werden können.

7.2. Softwareentwurf

In diesem Abschnitt geht es um den Softwareentwurf für ein Framework zur Modellierung und Simulation strukturvariabler Modelle.

Für die vorliegende Arbeit wurde ein eigenes Framework implementiert, anstatt vorhandene Ansätze zu nutzen, da:

- mehrere Simulationswerkzeuge integriert werden sollen,
- existierende Modelle möglichst einfach wiederverwendet werden sollen,
- und genau bekannt sein soll, was wann während der Simulation geschieht.

Eine erste Version dieses Frameworks wurde in [49] vorgestellt.

Ziel des Frameworks

Das Ziel ist es, verschiedene strukturvariable Modelle einfach zu modellieren und zu simulieren. Dabei sollen existierende Modelle möglichst ohne viel Aufwand wiederverwendet werden können.

Ein Modellierer soll die Strukturwechsel definieren und das Framework führt diese aus. Dabei sollen möglichst keine Programmierkenntnisse notwendig sein. Andererseits sollen für kompliziertere Moduswechsel auch erweiterte Funktionalitäten nutzbar sein, die dann Programmierkenntnisse erfordern dürfen.

Beim Entwurf des Frameworks werden folgende Annahmen getroffen:

- Es finden nur Moduswechsel auf der obersten Modellebene statt. Die Modelle sind demnach alle ausfaktorisiert. Die Schnittstellen der Modellkomponenten müssen somit nicht gesondert betrachtet werden.

- Jeder Modus ist ein Modell, welches in einem Simulationswerkzeug lauffähig ist.

Die genauen Anforderungen an das Framework können der Tabelle A.2 aus dem Anhang A.2 entnommen werden.

Grobe Idee zum Ablauf einer Simulation

Zunächst wird ein grober Überblick über die Umsetzung des Frameworks gegeben, sodass der spätere Softwareentwurf verständlicher wird.

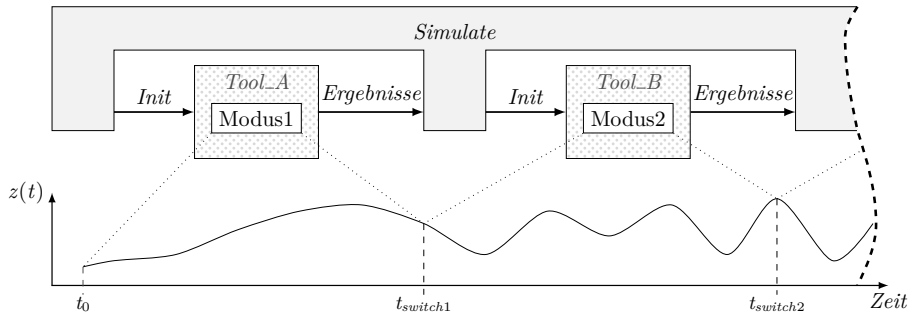


Abb. 7.3.: Ablauf einer Simulation mit zwei Modi, die in zwei unterschiedlichen Simulationswerkzeugen simuliert werden

Da jeder Modus in seinem eigenen Simulationswerkzeug simuliert und die Modellierungssprachen nicht verändert werden sollen, ist eine Ebene über den Werkzeugen notwendig. Diese soll eine beliebige Anzahl von Modi und deren Simulation in verschiedenen Simulationswerkzeugen koordinieren. Abbildung 7.3 zeigt schematisch, wie eine Simulation eines strukturvariablen Modells abläuft.

Am Anfang einer Simulation wird die Beschreibung des strukturvariablen Modells eingelesen und die Modi und Transitionen werden identifiziert. Danach wird die Simulation des initialen Modus im vorgesehenen Simulationswerkzeug gestartet. Die Kommunikation zum Simulationswerkzeug findet durch ein Interface statt. Sobald eine Übergangsbedingung während der Simulation eintritt, wird die Simulation beendet und die Simulationsergebnisse werden durch die Steuerungsebene eingelesen. Die alten Simulationsdaten werden verwendet, um die Startwerte für den nächsten Modus zu bestimmen und diesen zu simulieren. Dieser Ablauf wird fortgesetzt, bis die Endzeit der Simulation erreicht ist.

7.2.1. Objektorientierte Struktur eines strukturvariablen Modells

Strukturvariable Modelle wurden eingehend in Abschnitt 6.1 behandelt. Dort wurde festgehalten, welche Daten in einem solchem Modell notwendig sind (siehe Definition 6.13). Diese Definition wird als Basis verwendet; sofern nichts anderes beschrieben wird, entsprechen die Datentypen denen der Definition. Zur effizienteren Implementierung sind einige Anpassungen notwendig, jedoch verändern diese die geforderten Eigenschaften der strukturvariablen Modelle nicht.

Strukturvariables Modell

Abbildung 7.4 zeigt ein Klassendiagramm eines strukturvariablen Modells für die Implementierung. Die Klasse *VSM* repräsentiert die Klasse eines strukturvariablen Modells für das Framework. Es setzt dabei die Struktur der Definition 6.13 um.

Im Gegensatz zur bekannten Definition sind die Transitionen in der Implementierung den Modi zugeordnet. Tritt eine Wechselbedingung ein, muss nur in den Transitionen des gerade aktiven Modus gesucht werden.

Der *observer* ist im Vergleich zur Definition ebenfalls anders aufgebaut. In der Definition werden für jede Variable im *observer* die Zeit und die Simulationsdaten als Tupel gespeichert. Das Framework ist zurzeit jedoch auf Solver beschränkt, welche über die Zeit diskretisieren. Durch diese Einschränkung ist der Zeitvektor für alle Variablen identisch und muss nur einmal gespeichert werden. Ein Zurückgehen in die Vergangenheit ist damit ebenso vereinfacht, da vorausgesetzt werden kann, dass für einen vergangenen (berechneten) Zeitpunkt ein Wert für alle simulierten Variablen vorhanden ist.

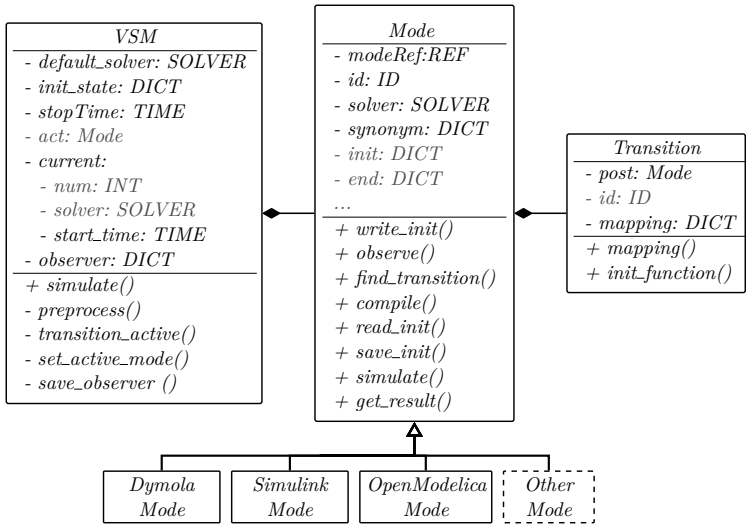


Abb. 7.4.: Objektorientierte Sicht auf ein strukturvariables Modell; dabei sind die vom Modellierer zu spezifizierenden Daten schwarz und die vom Framework generierten grau dargestellt

Im Framework wird der Datentyp *Dictonary* verwendet, der für jede Variable, die im *observer* definiert ist, einen Datenvektor speichert (in Abbildung 7.4 *DICT* genannt). Zusätzlich werden die Simulationszeit und die ID des aktuellen Modus gespeichert. Ein solcher Beobachter kann beispielsweise wie folgt aussehen:

$$observer = \left\{ \begin{array}{lcl} (time & \mapsto & \{t_1, t_2, \ldots, t_a, t_b, \ldots\}), \\ (x & \mapsto & \{x_1, x_2, \ldots, x_a, x_b, \ldots\}), \\ (y & \mapsto & \{y_1, y_2, \ldots, y_a, y_b, \ldots\}), \\ (z & \mapsto & \{z_1, z_2, \ldots, NaN, NaN, \ldots\}), \\ (modeID & \mapsto & \{1, 1, \ldots, 2, 2, \ldots\}) \end{array} \right\}$$

In diesem Beobachter gibt es drei Variablen (x, y, z). Dabei existiert für den zweiten Modus kein Synonym für z, was dazu führt, dass an allen Zeitpunkten des zweiten Modus ein NaN (Not a Number) im Datenvektor von z steht. Die Datenvektoren müssen alle dieselbe Länge haben, da sonst keine eindeutige Zuordnung zu den Simulationszeiten möglich ist.

Die Simulationsergebnisse der Modi (*state*) werden in Dateien gespeichert, wobei sich der Name der Datei aus der Simulationsnummer und der ID des Modus ergibt (vgl. *current.num* und *mode.id* aus der Formalisierung).

Die Synonyme der einzelnen Modi sind in der Definition 6.13 im Modell enthalten. Da diese Information abhängig vom Modus ist (jeweils ein Mapping von Modus zu anderen Daten), wird sie direkt im Modus gespeichert. So wird der Zugriff vereinfacht.

In der Formalisierung hat jeder Modus seinen eigenen Solver, der sich aus der Komposition der Komponenten des Modus ergibt. Im Framework hat jeder Modus genau einen Solver, der auch während der Beschreibung des strukturvariablen Modells noch geändert werden kann. Im Attribut *solver* des Modus kann der Solver des Modus angegeben werden. Im *VSM*-Objekt wird zusätzlich eine Default-Solver vorgesehen. Falls alle Modi den gleichen Solver haben, kann dieser gesetzt werden. Wird in einem Modus kein Solver angegeben, wird der Default-Solver verwendet.

Die Initialisierung des ersten Modus kann im *VSM*-Objekt im Attribut *init_state* angegeben werden.

Modus

Die Klasse *Mode* repräsentiert die Modi in einem VSM-Objekt. Da das Framework nicht die einzelnen Modi umsetzt und simuliert, ist eine Referenz auf das zu simulierende Modell vorgesehen (*modeRef*). In diesem Modell sind die Variablen, Gleichungen, Schnittstellen, Zustände und Initialinformationen enthalten.

Zusätzlich werden hier der moduseigene Solver (*solver*) und die Synonyme (*synonym*) für den Beobachter definiert. Zum Beschreiben der Synonyme wird die Datenstruktur aus der Definition 6.13 verwendet, jedoch ohne die Angabe des Modus. Dadurch kann wieder ein *Dictonary*-Datentyp verwendet werden. Dieser kann einer Variablen des Beobachters eine zugehörige Variable des Modus zuweisen.

Im Modus werden die Zustände (*state*) nicht gespeichert, da diese während der Simulation des Modells im Werkzeug intern gehalten werden.

Für die jeweiligen Werkzeuge, in denen der Modus simuliert werden soll, wird eine Vererbungshierarchie vorgesehen. So ist sichergestellt, dass jeder Modus in einem Simulationswerkzeug die notwendigen Daten enthält. Weitere Vorteile dieser Variante werden bei der Beschreibung der Simulationsmethode in Abschnitt 7.2.2 ersichtlich.

Transition

Eine Transition enthält einen *guard*, dieser wird nun jedoch innerhalb eines referenzierten Modells (Simulink-Modell, Modelica-Modell, etc.) implementiert und bei der Simulation überprüft. Daher taucht er in der implementierten Repräsentation nicht auf.

Im Modell muss beim Eintreten einer Bedingung die Transition identifiziert werden, dazu erhalten die Transitionen eine *ID*. Diese *ID* muss beim Eintreten des Ereignisses gesetzt werden und sie wird im Framework zur Identifikation der zu aktivierenden Transition verwendet.

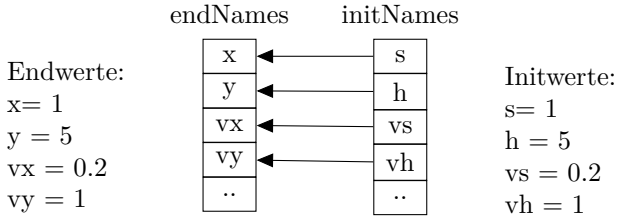


Abb. 7.5.: Eins-zu-eins-Zuweisung bei einer aktiven Transition

Die Initialisierung der Transition (*init*) ist für die Implementierung in zwei Teile geteilt:

Die einfachste Initialisierung ist in Abbildung 7.5 zu sehen, dort findet eine Eins-zu-eins-Zuweisung statt. Es wird dabei genau ein Endwert verwendet, um einen Startwert zu setzten. Dies kann durch ein Mapping dargestellt werden ($mapping : initName \nrightarrow endName$). Abbildung 7.5 zeigt ein abstraktes Beispiel. Dazu wird im Framework der Dictonary-Datentyp verwendet. Dabei bedeutet $\{'s' : 'x',\ 'h' : 'y', \ldots\}$, dass die Variable s im zu aktivierenden Modus durch den Endwert der Variable x des vorherigen Modus initialisiert wird, usw.

Eine weitere Variante zur Initialisierung ist die Initialisierungsfunktion zur Berechnung von Startwerten. In dieser können Startwerte beliebig aus bekannten Daten bestimmt werden. Für die Implementierung können auch die Abläufe der Aktionen einer Transition in dieser Funktion integriert werden, wodurch nicht nur Berechnungen möglich sind, sondern Modi auch neu kompiliert und Transitionen ausgelöst werden können.

Für die Implementierung hat diese Init-Funktion Zugriff auf das komplette VSM-Modell (da die Transition den *post*-Modus enthält und dieser eine Referenz zu dem Modell, in dem es enthalten ist). Mit der Init-Funktion sind so beliebige Manipulationen möglich.

7.2.2. Methode zur Simulation des objektorientierten, strukturvariablen Modells

Die Simulation wird als Methode (*simulate()*) der *VSM*-Klasse implementiert und übernimmt die Koordination der Simulationen der verschiedenen Modi. Da die *VSM*-Klasse alle erforderlichen Daten enthält, stehen diese auch in der Methode zur Verfügung. Abbildung 7.6 zeigt den Ablauf der Methode. Dieser ist dem Ablauf aus Abbildung 7.1 sehr ähnlich, enthält jedoch einige weitere Details.

Jeder Schritt aus Abbildung 7.6 ist eine Methode, wobei die verschiedenen Methoden an unterschiedlichen Stellen implementiert sind. Alle grau hinterlegten Methoden sind vom Modus abhängig, werden also in der *Mode*-Klasse beschrieben. Dabei wird zwischen werkzeugabhängigen und werkzeugunabhängigen Methoden unterschieden. Die Methoden in gestrichelten Kästen sind werkzeugabhängig. Diese müssen individuell für jedes Simulationswerkzeug in der entsprechenden *Mode*-Klasse implementiert werden. Alle werkzeugunabhängigen Methoden sind direkt in der allgemeinen Mode-Klasse enthalten und gelten für alle Simulationswerkzeuge. Methoden, welche nicht grau hinterlegt sind, sind allgemeine Methoden, die vom jeweiligen Modus unabhängig sind und zur *VSM*-Klasse gehören.

Die jeweiligen Methoden werden im Folgenden näher beschrieben, dabei sind im Framework bereits Dymola, OpenModelica und Simulink integriert. Bei der Beschreibung werden die Unterschiede für die jeweiligen Werkzeuge erläutert.

VSM-Objekt verarbeiten (*preprocess()*): *Das vom Modellierer beschriebene strukturvariable Modell wird eingelesen und die enthaltenen Informationen verarbeitet. Diese Methode dient dazu herauszufinden, welcher Modus der initiale ist und welche Startwerte für ihn verwendet werden.*

Kompilieren (*compile()*, in Modus, werkzeugabhängig): *Das gerade aktive Modell wird kompiliert, falls dies notwendig ist. In Simulink ist dieser Schritt nicht notwendig, daher würde für Simulink diese Methode keinen Methodenrumpf mit Funktionalität haben. Für Modelica-Modelle hingegen wird der Compiler des angegebenen Werkzeugs aufgerufen. Jeder Modus wird für eine Simulation nur einmal kompiliert. Aus diesem Grund ist diese Methode während der Schleife in Klammern dargestellt. Der erste Modus muss immer kompiliert werden.*

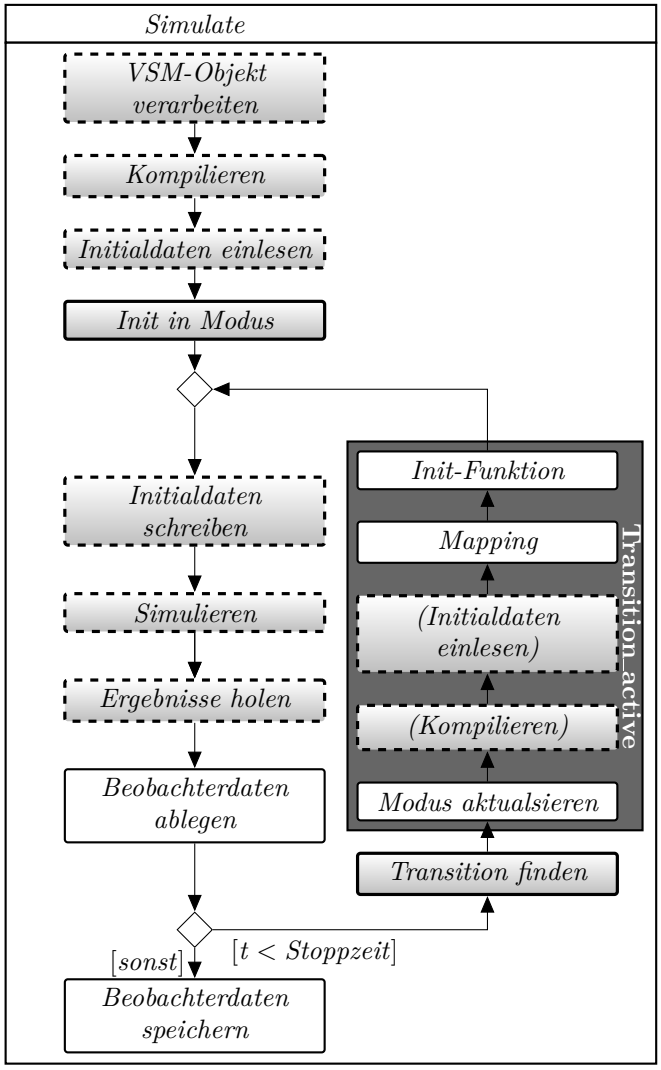


Abb. 7.6.: Ablauf in der *simulate*-Methode (gestrichelte Kästen stellen werkzeugabhängige Schritte dar)

Initaldaten einlesen (*read_init()*, in Modus, werkzeugabhängig): *Als nächstes werden alle Initialdaten des Modus eingelesen.*

In Simulink ist es üblich, Initialdaten in einer M-Datei vorzusehen. Aus diesem Grund wird auch im Framework eine solche M-Datei für die Initialisierung verlangt und eingelesen. Diese Datei muss alle Daten enthalten, die für die Initialisierung des Simulink-Modells erforderlich sind. Variablen, welche nicht in dieser Datei enthalten sind, können mit dem Framework auch nicht initialisiert werden.

Bei den Modelica-Werkzeugen wird beim Kompilieren eine Initialdatei erzeugt, die eingelesen wird. Diese Datei enthält alle Variablen und deren Initialwerte für das Modelica-Modell. Alle Variablen und deren Werte werden im Attribut (init) des Modus als Dictonary-Datentyp in der Form $\{'varname1' : value1, 'varname2' : value2\}$ *gespeichert.*

Diese Methode wird nur einmal für jeden Modus ausgeführt, wenn dieser neu kompiliert wurde.

Init in Modus (*write_init()*, in Modus): *Wurden Initialwerte für den ersten Modus angegeben, werden die übergebenen Werte an die entsprechende Stelle der init-Liste des Modus geschrieben. Diese Methode ist werkzeug***un***abhängig.**

Als nächstes wird die While-Schleife betreten, in der die Simulationen gestartet und ausgewertet werden. Zunächst werden die Initialwerte, welche im Mode-Objekt in der *init*-Liste gespeichert sind, zum Initialisieren des aktiven Modus verwendet.

Initialdaten schreiben (*save_init(), in Modus, werkzeugabhängig)*: *Diese Initialisierung ist abhängig vom verwendeten Simulationswerkzeug. In Simulink werden die Initialwerte in den Workspace von Matlab geschrieben. In Dymola und OpenModelica werden die Daten in die Initialisierungsdatei geschrieben.*

Simulieren (*simulate()*, in Modus, werkzeugabhängig): *Daraufhin wird die Simulation des derzeit aktiven Modus in dem spezifischen Werkzeug gestartet. In Simulink muss Matlab geöffnet sein und das MDL-Modell wird simuliert. Für die Modelica-Modelle ist das entsprechende Simulationswerkzeug nicht mehr notwendig, da nur die ausführbare Datei aufgerufen werden muss.*

Ergebnisse holen (*get_result()*, in Modus, werkzeugabhängig): *Sobald die Simulation im Werkzeug terminiert, werden die Simulationsdaten eingelesen.*

In Simulink werden die Daten aus dem Workspace von Matlab ausgelesen. Dafür muss im Simulink-Modell ein entsprechender Block vorgesehen werden.

In Dymola und OpenModelica wird die Ergebnisdatei eingelesen, die alle Simulationsdaten enthält.

Die Namen und Werte der Enddaten werden analog zu den Initialdaten in einer Dictonary-Liste gespeichert (end).

Die Simulationsergebnisse werden als eigene Datei mit einem individuellen Namen gespeichert. Der Name setzt sich aus der Nummer der Simulation und der Modus-ID zusammen (analog zum states aus der Formalisierung). Die Daten werden nicht im VSM-Objekt gehalten, um nicht zu viele Daten während der Simulation zu speichern. Die Daten können in einer Init-Funktion jederzeit eingelesen werden.

Beobachterdaten ablegen (*observe()*): *Die zuvor eingelesenen Daten werden für den Beobachter verwendet. Die Daten werden im vorgestellten Format im VSM-Objekt gespeichert, um am Ende der Simulation in eine Datei geschrieben zu werden. Ist kein Synonym für eine Variable angegeben, wird NaN an die entsprechende Stelle geschrieben. Diese Methode ist werkzeug***un***abhängig.*

Transition finden (*find_transition()*, in Modus): *Ist die Endzeit der Simulation nicht erreicht, findet ein Moduswechsel statt. Mit Hilfe der Enddaten der Simulation muss die zu aktivierende Transition eindeutig identifiziert werden. Dafür müssen die Transitionen des zuletzt aktiven Modus durchsucht werden. Dies geschieht über eine eindeutige Transitions-ID. Diese Methode ist werkzeug***un***abhängig.*

Ist die Transition gefunden, sind der nächste Modus und die Initialisierung bekannt. Die Transition wird aktiviert, was einer eigenen Methode im *VSM*-Modell entspricht. Diese Methode (*transition_active*) repräsentiert die Funktion aus Definition 6.15. Die Methode setzt sich aus folgenden Methoden zusammen:

Modus aktualisieren (*set_active_mode()*): *Der aktuelle Modus wird auf den neuen Modus geändert, wobei der alte Modus noch zugänglich ist. Diese Methode ist direkt in der VSM-Klasse implementiert.*

Nun kann der neue Modus kompiliert und dessen Initialdaten mit den bekannten Methoden ausgelesen werden. Dies geschieht jedoch nur, wenn der Modus noch nicht kompiliert wurde.

Mapping (*mapping()*, in Transition): *Bei der Initialisierung findet zunächst eine Eins-zu-eins-Zuweisung statt. Diese verwendet die Enddaten der alten Simulation zum Initialisieren des neuen Modus mit dem definierten Mapping der Transition.*

Init-Funktion (*init_function()*, in Transition): *Hier können beispielsweise Berechnungen oder ein erneutes Kompilieren durchgeführt werden. Diese Methode setzt Programmierkenntnisse voraus, da hier beliebige Funktionen implementiert werden können. Hier können die gegebenen Datenstrukturen verwendet und manipuliert werden. Der Modellierer ist komplett frei und kann alle Methoden des Frameworks verwenden. So kann auch die Methode transition_active aufgerufen werden, welche einen weiteren Moduswechsel einleiten kann.*

Für Simulink-Modelle können in dieser Methode beispielsweise die notwendigen Werte für die Initialisierung der vorgegebenen Variablen berechnet werden, sodass die Werte zu einem konsistenten Zustand bei der Initialisierung führen. Dies kann abhängig vom Modell sehr aufwendig sein.

In Modelica hingegen übernimmt das Werkzeug die konsistente Initialisierung und es ist nicht zwangsläufig notwendig, alle Variablen zu initialisieren.

Die While-Schleife wird wieder betreten und der nun aktive Modus wird simuliert.

Beobachterdaten speichern (*save_observer ()*): *Ist die vordefinierte Endzeit der Simulation erreicht, so werden die gespeicherten Beobachterdaten in einer MAT- oder CSV-Datei gespeichert. Diese Methode ist direkt in der VSM-Klasse implementiert.*

Der objektorientierte Aufbau des strukturvariablen Modells ermöglicht es, die Methoden, die einen Modus direkt betreffen, in der *Mode*-Klasse zu implementieren.

Der objektorientierte Aufbau bietet sich auch für die Methoden an, die sowohl modus- als auch werkzeugabhängig sind. Es werden zunächst alle notwendigen Attribute und Methoden in der allgemeinen Klasse *Mode* definiert. Die nicht vom Werkzeug abhängigen Methoden sind mit ihrem Methodenrumpf in der *Mode*-Klasse implementiert, während für die werkzeugabhängigen Methoden die Signaturen vorgegeben sind. Für jedes integrierte Werkzeug wird eine *Tool-Mode*-Klasse implementiert, die von der abstrakten Klasse erbt. In dieser werden die werkzeugspezifischen Methodenrümpfe implementiert. Jeder Modus im strukturvariablen Modell muss ein spezialisiertes *Mode*-Objekt sein.

Da die Ablaufsteuerung zu jeder Zeit den aktiven Modus als Objekt zur Verfügung hat und dieser Modus ein spezialisiertes *Mode*-Objekt ist, kann mit der Punktnotation (*act_mode.method()*) die werkzeugspezifische Methode aufgerufen werden. Die Ablaufsteuerung kennt die verschiedenen Werkzeuge nicht. Sie nutzt lediglich verschiedene *Mode*-Objekte, welche alle über die gleichen Methoden verfügen, und ruft diese Methoden auf. Ein neues Werkzeug kann integriert werden, indem eine weitere spezialisierte *Mode*-Klasse erzeugt wird. Die Ablaufsteuerung bedarf keiner Änderung.

Zum Beschreiben des Modells steht eine Vorlage einer Config-Datei zur Verfügung, in der die objektorientierte Struktur aus Abbildung 7.4 vorgegeben ist. Die internen Informationen wie *init* und *end* müssen nicht angegeben werden. Das Framework analysiert die Angaben aus der Config-Datei und erzeugt alle internen Informationen. Am Ende der Simulation können die Daten geplottet werden.

7.3. Python-Implementierung von DySMo

Im Folgenden wird die Implementierung des *DySMo* (**Dy**namic **S**tructure **Mo**deling) Frameworks erläutert. Dabei wird auf die gewählte Programmiersprache und auf die bereits integrierten Simulationswerkzeuge eingegangen.

Die Simulationswerkzeuge und die Modi müssen Anforderungen erfüllen, um im Framework verwendet zu werden. Diese Anforderungen, die für die hier vorgestellte Implementierung gelten, werden erläutert.

Wahl der Programmiersprache

Die Programmiersprache sollte in der Lage sein, verschiedene Simulationswerkzeuge zu steuern. Da die objektorientierte Darstellung von strukturvariablen Modellen wünschenswert ist, sollte die Programmiersprache ebenfalls objektorientierte Konzepte bereitstellen. Diese Anforderungen werden beispielsweise von Matlab, Java und Python erfüllt.

Für das Framework wurde Python gewählt, da es nicht kommerziell und bei vielen Ingenieuren und auch in der Modelica-Community bekannt ist. Zudem bietet Python eine Vielzahl von Bibliotheken, welche die Implementierung der notwendigen Eigenschaften erleichtern. Alles Folgende bezieht sich daher auf die Implementierung in Python.

Das Framework wurde genau wie in Abschnitt 7.2 entworfen implementiert.[1]

Integrierte Simulationswerkzeuge

In der aktuellen Version des Frameworks sind folgende drei Simulationswerkzeuge integriert: Dymola, OpenModelica und Simulink. Es wurden zwei Modelica-Simulationsumgebungen integriert. Dymola wurde gewählt, da einige Modelle der Fallstudie nur in Dymola lauffähig sind. Um jedoch nicht nur kommerzielle Werkzeuge zu integrieren, wurde auch OpenModelica integriert.

Bereits vorhandene Schnittstellen wie FMI[2] [4] oder HLA[3] (High-Level Architecture) [17] wurden für das Framework nicht verwendet.

FMI war zum Beginn der Implementierung noch im Anfangsstadium [3]. Heute ist die Version 2.0 [4] vorhanden, allerdings ist beispielsweise die Toolbox für Matlab/Simulink kostenpflichtig. Aus diesem Grund wurde die ursprüngliche Schnittstelle beibehalten. Eine Erweiterung auf FMI wäre jedoch problemlos möglich, wurde aber nicht umgesetzt, da die im Framework enthaltene Schnittstelle alle notwendigen Funktionalitäten bereitstellt.

[1] DySMo ist frei verfügbar und kann unter `https://gitlab.tubit.tu-berlin.de/amsun/dysmo` heruntergeladen werden.

[2] `https://www.fmi-standard.org/` (Sept 2014)

[3] `http://www.pitch.se/hlatutorial` (Sep 2014)

Der Standard für Simulator-Kopplung HLA ist für den hier verwendeten Ansatz ebenfalls nicht geeignet. Für Matlab/Simulink wäre eine externe Software notwendig und für die verwendeten Modelica-Werkzeuge ist eine solche Schnittstelle zurzeit nicht vorhanden.

Die Schnittstellen zu den Werkzeugen wurden mit Hilfe von Python-Bibliotheken implementiert.

Dymola 2014: *Das Framework wurde mit Dymola 2014 getestet und die Schnittstelle ist über einen DDE-Channel in Windows realisiert. Über diesen DDE-Channel wird das Kompilieren der jeweiligen Modi eingeleitet und die erzeugten ausführbaren Dateien werden nach deren Modi sortiert. Nachdem die Modelle kompiliert sind, ist eine Kommunikation mit Dymola nicht mehr notwendig und Python ruft direkt die ausführbare Simulationsdatei auf und initialisiert das Modell über MAT-Dateien. Das Auslesen von Daten findet ebenfalls über MAT-Dateien statt.*

OpenModelica 1.9.1: *Im Framework wird die Version OpenModelica 1.9.1 verwendet.*

Um OpenModelica zu verwenden, wird die OpenModelica-Shell direkt aus Python angesprochen und das Kompilieren eines Modells eingeleitet. Auch hier wird nach dem Kompilieren keine Kommunikation zu OpenModelica benötigt und nur noch mit Dateien zum Initialisieren, Auslesen von Daten und den ausführbaren Simulationsdateien gearbeitet.

Frühere Versionen von OpenModelica funktionieren im Framework nicht, da sie zum einen den terminate-Befehl und zum anderen die jetzige Initialisierungsroutine nicht unterstützen.

Simulink 2014a: *Simulink wurde in der Version 2014a verwendet. Dazu wird das Python-Modul pymat*[4] *verwendet. Dieses ruft Matlab auf und öffnet es. Durch das bereitgestellte Interface können dann normale Matlab-Befehle, wie sie auch schon aus dem skriptbasierten Ansatz bekannt sind, verwendet werden.*

[4] `http://pymat.sourceforge.net/` (Sep 2014)

Anforderungen an die Modi und die Simulationswerkzeuge

Modi

In DySMo sind Moduswechsel nur auf der obersten Hierarchieebene des Modells erlaubt und jeder Modus muss ein simulierbares Modell sein. Die integrierten Werkzeuge, die zur Simulation verwendet werden, haben keine Informationen darüber, dass es sich um ein strukturvariables Modell handelt. Das Simulationswerkzeug kennt nur ein aktives Modell, welches im strukturvariablen Modell einen Modus repräsentiert. Dieses simuliert es, bis ein vordefiniertes Ereignis die Simulation stoppt. Jeder Modus muss demnach eine Stoppbedingung enthalten, welche angibt, wie lange der aktive Modus gültig ist. Vor dem Beenden der Simulation muss die zu aktivierende Transition angegeben werden (durch ihre eindeutige Transitions-ID).

Die Initialisierung ist bei einem Moduswechsel von großer Bedeutung. Ein sinnvoller Wechsel kann nur stattfinden, wenn der alte Modus genug Daten bereitstellt, damit der neue Modus initialisiert werden kann. Dies ist besonders relevant, wenn von einem weniger detaillierten Modell in ein detaillierteres gewechselt wird. In diesem Fall sind eventuell nicht alle erforderlichen Daten im ersten Modell enthalten. Bereits bei der Wahl der Modi muss sichergestellt werden, dass auch weniger detaillierte Modelle genügend Daten bereitstellen. Ähnliches gilt auch bei Verhaltensänderungen, wenn manche Variablen in einem der Modi gar nicht existieren. Ebenso gilt dies, wenn Komponenten, die initialisiert werden müssen, neu hinzukommen.

Die Initialisierung kann vereinfacht werden, wenn in einem Modus angegeben wird, welche Variablen initialisiert werden sollen. Das bedeutet, dass das *start*-Attribut der Formalisierung (vgl. Definition 6.4) im Modell angegeben wird. In Simulink kann dazu die M-Datei, welche die Initialdaten beschreibt, verwendet werden.

In Modelica können die zu initialisierenden Variablen in der jeweiligen Initialdatei (Dymola:*dsin.txt*, OpenModelica: Model.xml) gefunden werden. In dieser sind *fixed*-Variablen und Parameter gekennzeichnet. Sie gibt somit einen Einblick in die notwendigen Startwerte.

Eine weitere gute Möglichkeit ist das Einführen von Start-Parametern. Diese können im Modell dazu verwendet werden, um die notwendigen Variablen zu initialisieren. Es kann damit bereits bei der Modellierung eine gute Kombination von Startwerten festgelegt werden.

Simulationswerkzeuge

Nicht nur die gewählten Modi, sondern auch die Werkzeuge, die in DySMo integriert werden, müssen einige Eigenschaften aufweisen. So muss das Werkzeug durch Python kontrollierbar sein.

Eine externe Initialisierung von Modellen muss ebenso möglich sein. In Modelica-Werkzeugen ist das meist durch Initialdateien (bspw. Dymola: dsin.txt, OpenModelica: Model.xml) möglich, während in Matlab Startwerte in den Matlab-Workspace geschrieben werden können. Das Werkzeug muss zudem in der Lage sein, Ereignisse in einem Modus auszuwerten und gegebenenfalls die Simulation zu stoppen. Dieses Stoppen sollte kein fehlerhaftes Anhalten sein, sondern die Simulation korrekt beenden und die Simulationsdaten zur Auswertung bereitstellen.

7.4. Verwendung von DySMo

Um zu zeigen, wie DySMo verwendet werden kann, wird das bekannte Satellitenstartmodell aus Beispiel 3.1 in DySMo umgesetzt.

In dem hier erstellten Modell sollen drei verschiedene Simulationswerkzeuge verwendet werden. Dabei ist das Raketenmodell in Simulink realisiert und die beiden Satellitenmodelle in Modelica, wobei eines in Dymola und das andere in OpenModelica simuliert wird. Abbildung 7.7 zeigt die objektorientierte Sicht auf das Gesamtmodell und die jeweiligen Werkzeuge der Modi.

Erstellen der Modi für ein strukturvariables Modell

Zunächst müssen die Modi vorbereitet werden. Dazu muss, wie beim skriptbasierten Ansatz, jeder Modus in seinem entsprechenden Simulationswerkzeug lauffähig sein. Jeder Modus braucht ebenfalls eine Stoppbedingung, welche die Simulation terminiert. Dies ist also identisch zum skriptbasierten Ansatz aus Abschnitt 7.1. Als zusätzliche Information muss jeder Modus vor dem Terminieren noch die zu aktivierende Transition identifizieren. Dies kann durch die eindeutige ID der Transition geschehen. Listing 7.6 zeigt dies exemplarisch für das Modelica-Modell des nicht angetriebenen Satelliten.

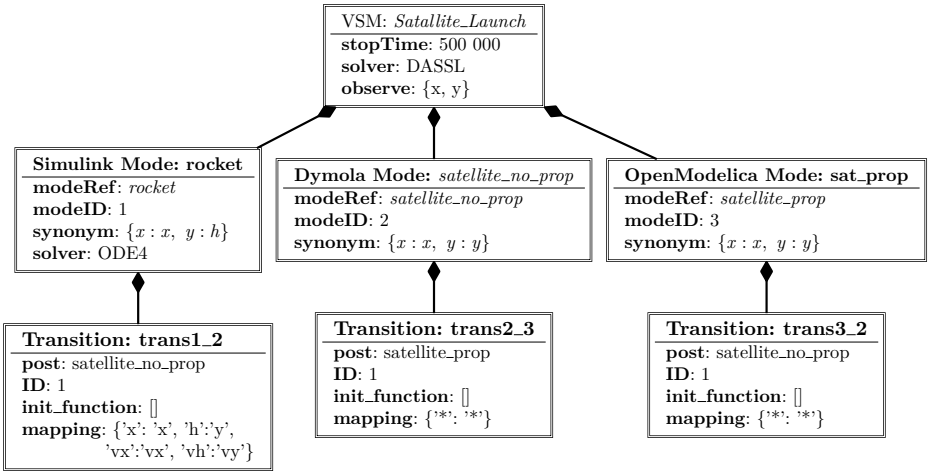


Abb. 7.7.: Objektorientierte Sicht auf das Satellitenstartmodell mit drei verschiedenen Simulationswerkzeugen

```
1  model sat_no_prop
2   extends satellit
3   Real loop;
4   Integer transitionId;
5  equation
6    when x <= 0 then
7     loops = pre(loops)+1;
8    end when;
9    when (loops == 20) then
10    transitionId = 1;
11    terminate("loops");
12   end when;
13 end sat_no_prop;
```

Listing 7.6: Satellitenmodell in Modelica mit Stoppbedingung

Der Programmcode ist fast identisch zu dem aus Listing 7.3 für den skriptbasierten Ansatz, es wird diesmal jedoch vor dem Terminieren der Simulation die Variable *transitionId* gesetzt, welche die zu aktivierende Transition identifiziert. Die IDs der Transitionen sind dabei für einen Modus eindeutig, da nur diese aus dem Modus hinausführen. Die ID entspricht dem Array-Index[5] der Transition (siehe Listing 7.7 Zeile 23).

[5]Mathematisch nummeriert: der erste Index entspricht der Eins.

Erstellen des Gesamtmodells

Um das Gesamtmodell möglichst einfach zu definieren, ohne dass Programmierkenntnisse notwendig sind, bietet DySMo eine Modellabstraktion, in der die in schwarz geschriebenen Informationen aus Abbildung 7.4 in einer Config-Datei angegeben werden. Die in der Abbildung grau geschriebenen Informationen werden vom Framework erzeugt. Ein Auszug aus der Config-Datei ist in Listing 7.7 zu finden.

Das Modell (*mode*) existiert per default und es kann ein Default-Solver durch eine Instanz vom Typ *Solver* gesetzt werden (Zeile 2). In diesem können alle Solvereinstellungen getroffen werden. Im Beispiel wird nur der DASSL gesetzt.

Dem ersten Modus werden keine Startwerte übergeben, es werden daher die im Modell gesetzten Werte verwendet (Zeile 3).

Als Nächstes wird die Stopp- und Startzeit der Simulation gesetzt (Zeilen 4, 5). Die Startzeit wird nach jedem Moduswechsel angepasst, die Stoppzeit bleibt immer identisch, außer sie wird in einer Funktion in der Transition geändert.

Der Beobachter *observe* wird nur mit den Namen der zu speichernden Variablen beschrieben, die tatsächliche Datenstruktur zum Speichern der Daten wird später generiert (Zeile 6).

Die Instanzen der Modi werden für ihre entsprechenden Werkzeuge erzeugt (Zeile 9).

Der erste Modus ist das Raketenmodell. Der Name des Modells wird in der Referenz (*mode*1.*modeRef*) gespeichert (Zeile 13).

Der im Modell beschriebene Solver kann im Modus überschieben werden. Dazu kann das Attribut *solver* im Modus gesetzt werden. Für den Simulink-Modus wird der Solver auf den ODE4 gesetzt (Zeilen 14).

Die Synonyme werden daraufhin beschrieben. Für diesen Modus werden für beide Beobachtervariablen Synonyme angegeben (Zeile 15).

Als nächstes können beliebig viele Transitionen definiert werden, die aus dem gerade definierten Modus zu einem anderen Modus führen. Dazu wird eine Instanz der Transition erzeugt (Zeile 18). In dieser wird die Instanz des *post*-Modus angegeben (Zeile 19).

Die Initialisierung wird durch eine Eins-zu-eins-Zuweisung vorgenommen (Zeile 20).

```
1  ######## DEFINE A MODEL #########
2  model.default_solver = Solver("dassl"); # solver
3  model.init = {}; # no inits
4  model.stopTime = 500000 # stop time
5  model.startTime = 0 # start time
6  model.observe = ['x', 'h'] # names of observer
7
8  ######## IINSANTIATE MODES #########
9  model.modes = [SimulinkMode(), DymolaMode(),
                  OpenModelicaMode()]
10
11 ######## DEFINE A MODE #########
12 mode1 = model.modes[0]
13 mode1.modeRef = "rocket" # reference to model
14 mode1.solver = Solver("ODE4") # instantiate solver
15 mode1.synonym = {'x':'x', 'h':'y'} # synonyms
16
17 ######## DEFINE TRANSITIONS #########
18 trans1_2 = Transition() # instantiate transition
19 trans1_2.post = model.modes[1]
20 trans1_2.mapping = {'s1.y':'rakete.height',
       'sat.x':'rakete.x',  'sat.vx':'rakete.der(x)',
       'sat.vy':'rakete.der(h)'} # init mapping
21
22 ########  ADD TRANSITIONS TO MODE ############
23 mode1.transitions = [trans1_2]
24 ...
```

Listing 7.7: Ausschnitt aus der Config-Datei zur Spezifikation von strukturvariablen Modellen, es wird nur der Modus des Raketenstarts mit seiner Transition gezeigt

Des Weiteren kann eine Funktion angegeben werden, die dem Attribut *fct* der Transition zugewiesen wird. Diese Funktion muss folgende Methodensignatur aufweisen: *def name(actMode, oldMode):*.

In den übergebenen Objekten (*actMode*, *oldMode*) sind alle notwendigen Daten enthalten. In dieser Methode sind jegliche Manipulationen der Modi möglich, welche mit Python realisiert werden können. Listing C.2 aus dem Anhang C zeigt, wie solche Init-Funktionen in DySMo verwendet werden können.

Die Transition wird dann dem Modus zugewiesen (Zeile 23).

Das Modell ist nun vollständig beschrieben und kann in DySMo simuliert werden.

7.5. Evaluation von DySMo

In diesem Abschnitt wird das Framework auf seine Performance und Einschränkungen evaluiert. Die Performance wird anhand von einfachen Modellen getestet, bei denen es sich um Extremfälle handelt, also besonders viele Moduswechsel oder besonders viele Zustandsvariablen.

Bei der Analyse der Einschränkungen wird darauf eingegangen, wie Modelle und deren Modi aufgebaut sein müssen, um in DySMo verwendet zu werden.

Die Einschränkungen werden bei der Fallstudie noch einmal aufgegriffen und anhand von konkreten Beispielen verdeutlicht.

7.5.1. Effizienz

Um die Effizienz zu untersuchen, wird ein sehr einfaches Modell eines Balls verwendet. Dieser Ball wird in einer Höhe h fallen gelassen; sobald der Ball auf den Boden trifft, wird ein Moduswechsel eingeleitet. Die aktivierte Transition führt wieder zum Ballmodell zurück und initialisiert dabei die Höhe mit dem Endwert der letzten Simulation, also eine Eins-zu-eins-Zuweisung. Die Geschwindigkeit des Balls wird in einer Funktion (f) berechnet, die den Endwert der Geschwindigkeit negiert. Dies ergibt ein Modell eines ewig springenden Balls ohne Energieverlust. Abbildung 7.8 zeigt die schematische Darstellung des Modells.

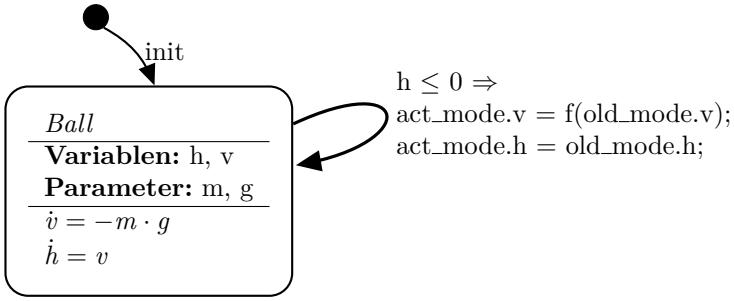


Abb. 7.8.: Schematische Darstellung eines ewig springenden Balls

Mit diesem Modell werden zwei unterschiedliche Tests gemacht. Zum einen wird es verwendet, um sehr viele Moduswechsel zu realisieren.

Der zweite Test betrifft die Größe des Modells. Dazu wird das gleiche Grundmodell verwendet und viele springende Bälle instanziiert, sodass bei einem Moduswechsel viele Initialwerte gesetzt werden müssen. Alle Tests wurden auf einem Windows 7 64bit-System mit 8 GB RAM und Intel® Core™ i7-2620M CPU 2.70GHz durchgeführt.

Variation der Anzahl der Moduswechsel

Um die Performance zu testen, wird das beschriebene Modell in Simulink, OpenModelica und Dymola implementiert. Die Simulationen werden so durchgeführt, dass 10, 100 und 1 000 Moduswechsel von DySMo durchgeführt werden (entspricht 40, 400, 4 000 Sekunden Simulationszeit). Abbildung 7.9 zeigt das Verhältnis der benötigten Zeit für die Moduswechsel (Skriptzeit) zur Gesamtsimulationszeit. Bei der Gesamtsimulationszeit wird die komplette Simulationszeit verwendet, jedoch ohne die Kompilation zu berücksichtigen. Die Annotationen in der Abbildung repräsentieren die Werte der benötigten Skriptzeit/Gesamtsimulationszeit.

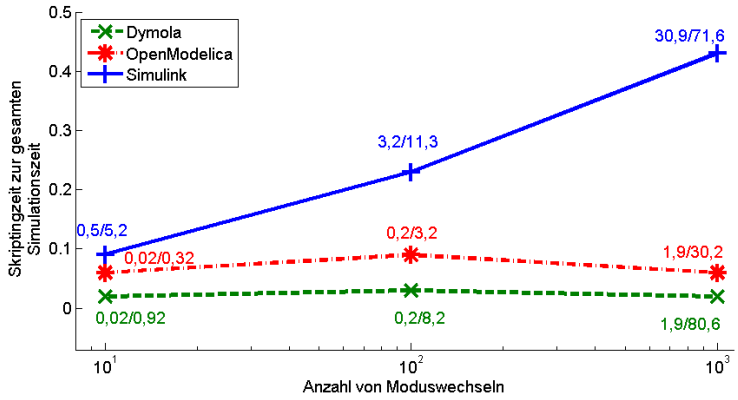


Abb. 7.9.: Verhältnis von Skriptzeit zur Gesamtsimulationszeit für unterschiedlich viele Moduswechsel

Das Verhältnis von Skriptzeit zu Gesamtsimulationszeit ist in Simulink für diesen Testfall am größten. Die langen Zeiten bei den Moduswechseln kommen durch das Auslesen der Enddaten zustande. Diese Daten werden nach jeder Simulation über das pymat-Modul aus dem Matlab-Workspace in Python eingelesen. Dies nimmt viel Zeit in Anspruch, wodurch sich der große zeitliche Overhead ergibt. Dies wäre sicher durch eine Optimierung bei der Implementierung zu beheben.

In Dymola und OpenModelica ist der zeitliche Overhead gering. Anhand der Kurven könnte man annehmen, dass die Kommunikation zu OpenModelica viel Zeit in Anspruch nimmt. Das Verhältnis ist jedoch nur höher, da die Simulation in OpenModelica weniger Zeit in Anspruch nimmt als in Dymola und damit die Gesamtsimulationszeit geringer ist.

Die Ergebnisse zeigen, dass selbst bei diesem sehr extremen Beispiel der Overhead in den Modelica-Werkzeugen zu vertreten ist. In der Realität würden solche Modelle mit sehr vielen Moduswechseln auch eher nicht mit einem strukturvariablen Modell implementiert werden, sondern nur mit Ereignissen.

Variation der Modellgröße

Der zweite Test betrifft die Größe der verwendeten Modelle und es wird untersucht, wie gut DySMo mit der Initialisierung der vielen Variablen zurechtkommt. Dazu wird wieder das Modell des springenden Balls verwendet. Diesmal werden jeweils nur zehn Moduswechsel simuliert, aber dafür wird die Anzahl der Bälle erhöht.

Es werden Modelle mit 10, 100, 1 000, 2 000 Bällen simuliert. Dies führt zu 20, 200, 2 000, bzw. 4 000 Variablen, die bei einem Moduswechsel initialisiert werden müssen. Dies ist wiederum ein Extrembeispiel, da in realen Beispielen oft nur ein Bruchteil der vorhandenen Variablen eines Modells initialisiert wird (Parameter und Zustandsvariablen).

Abbildung 7.10 zeigt wieder das Verhältnis von der Skript- zur Gesamtsimulationszeit mit den gemessenen Werten als Annotation. Auch hier ist Simulink wieder das Werkzeug, welches die längste Simulationszeit und das größte Verhältnis aufweist. Dieser große Zeitaufwand kommt wiederum durch das Auslesen der Enddaten zustande. Je mehr Daten ausgelesen werden, umso mehr Zeit wird für den Moduswechsel benötigt.

In Dymola und OpenModelica sind die benötigten Zeiten sehr ähnlich. OpenModelica benötigt etwas mehr Skriptzeit, diese kommt durch ein langsameres Auslesen der Simulationsergebnisse zustande. Die Gesamtzeit der Simulationen ist in OpenModelica erst bei größeren Modellen höher als in Dymola. Dies ist zum einen auf eine schnellere Simulation der kleinen Modelle in OpenModelica zurückzuführen und zum anderen auf die längere Skriptzeit bei größeren Modellen.

Auffällig ist, dass beide Kurven bei 100 Bällen höhere Verhältnisse aufweisen als bei 2 000 Bällen. Der Grund liegt in der längeren Simulationszeit der vielen Bälle, während die Skriptzeit sich nicht im gleichen Maße ändert. Dies zeigt auch, für welche Art von Modellen DySMo am ehesten geeignet ist: für große Modelle mit wenigen Moduswechseln. Für solche Modelle ist der Overhead des Ansatzes gering, während er für kleine Modelle und viele Wechsel zur Geltung kommt. Bei vielen Modi muss jeder Modus kompiliert werden, was bei großen Modellen ein enormer Zeitaufwand sein kann. Im vorliegenden Beispiel benötigt das Modell mit 1 000 Bällen ca. 2,5 Sekunden und das Modell mit 2 000 Bällen ca. 3,5 Sekunden zum Kompilieren in Dymola. Dabei handelt es sich noch um ein recht einfaches Modell. Der Overhead durch das Kompilieren ist daher nicht zu vernachlässigen.

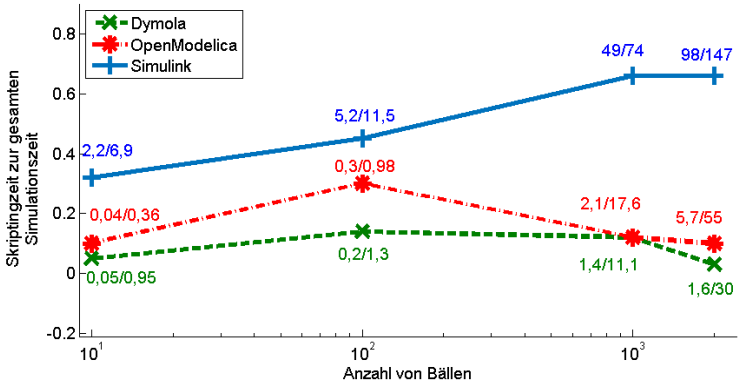


Abb. 7.10.: Verhältnis von Skriptzeit zur Gesamtsimulationszeit für unterschiedlich viele springende Bälle

Die Ergebnisse zeigen, dass selbst bei diesen sehr extremen Beispielen DySMo die Simulationen bewältigen kann. Bei großen Modellen, für die die Strukturänderungen am ehesten verwendet werden, ist die Simulationszeit meist hoch und es müssen verhältnismäßig wenig Variablen initialisiert werden. Die Zustandsvariablen und damit die zu initialisierenden Variablen liegen meist bei unter 100. Bei realistischen Modellen, wie sie in der Fallstudie in Kapitel 8 vorgestellt werden, ist das Verhältnis von der Skriptzeit zur Gesamtsimulationszeit meist unter 0,0005 und damit kaum relevant.

7.5.2. Einschränkungen

Der gewählte Ansatz für DySMo erzeugt einen Overhead bei der Simulation der Modelle, da die Initialisierung und das Auslesen von Daten aus den verschiedenen Formaten Zeit beanspruchen. Eine Simulationsumgebung wie beispielsweise SOL [96] oder MOSILAB [61] sind darauf ausgelegt, bei einem Moduswechsel bestimmte Daten zu lesen und zu schreiben. Da die integrierten Werkzeuge in DySMo diese Funktionalität zwar bieten, aber nicht speziell auf einen solchen Fall ausgelegt sind, treten hier Verzögerungen ein.

DySMo ist darauf angewiesen, dass jeder Modus ein eigenes lauffähiges Modell darstellt. Ein Modellierer muss die Modi in den entsprechenden Werkzeugen implementieren. Da die integrierten Werkzeuge keine Mittel bereitstellen, um Moduswechsel von Komponenten zu beschreiben, müssen die verschiedenen Varianten ausfaktorisiert werden. Das bedeutet: Besteht ein Modell aus zwei Komponenten, die jeweils zwei Modi haben, müssen für DySMo vier lauffähige Modelle erzeugt werden, welche die Kombinationen der Modi repräsentieren. In MOSILAB oder auch SOL kann dies direkt im Werkzeug angegeben werden.

Das Beschreiben von strukturvariablen Modellen ist in DySMo sehr einfach gehalten und es werden kaum Programmierkenntnisse gefordert. Erst wenn eigene Funktionen zur Initialisierung notwendig sind, sind Programmierkenntnisse hilfreich.

DySMo bietet zurzeit noch keine Unterstützung zur Überprüfung einer konsistenten Initialisierung. Der Modellierer ist verantwortlich für das Sicherstellen einer solchen Initialisierung. Für die Zukunft wäre es hilfreich Informationen aus den Modi zu extrahieren, um den Modellierer zu unterstützen und ggf. Vorschläge für die Initialisierung bereitzustellen.

7.6. Zusammenfassung

Mit dem vorgestellten Framework *DySMo* lassen sich strukturvariable Modelle mit existierenden Simulationswerkzeugen einfach definieren und simulieren. Bereits existierende Modelle lassen sich mit wenig Aufwand anpassen und in DySMo verwenden. Daher ist es einfach reale Beispiele als strukturvariable Modelle umzubauen. Durch den objektorientierten Aufbau ist das Erweitern von DySMo um weitere Simulationswerkzeuge einfach.

Mit der Evaluation wurde gezeigt, dass DySMo für große Modelle sowie für viele Moduswechsel geeignet ist. Dabei liegen die Stärken des Frameworks bei großen Modellen mit wenigen Wechseln, da hier am wenigsten Overhead produziert wird. Für kleine Modelle mit vielen Moduswechseln benötigt die Simulation des strukturvariablen Modells viel Zeit im Vergleich zur reinen Simulation im Werkzeug, wodurch *DySMo* für diesen Fall weniger geeignet ist als spezielle Sprachen und Werkzeuge für strukturvariable Modelle. DySMo unterstützt zurzeit Moduswechsel auf der obersten Modellebene. Sollen Modi auch in Komponenten erlaubt sein, so kann DySMo um die in Abschnitt 6.1 beschriebene Ausfaktorisierung erweitert werden.

Die Anforderungen und Ziele von DySMo wurden erfolgreich umgesetzt und es wird ermöglicht, einfach und effizient vorhandene Modelle wiederzuverwenden und diese in strukturvariable Modelle umzuwandeln.

8

Fallstudie zur Modellierung und Simulation

In diesem Kapitel werden verschiedene strukturvariable Modelle aus den unterschiedlichen Anwendungsgebieten modelliert und mit DySMo simuliert. Die Modelle wurden so gewählt, dass sie zum einen verschiedene Anwendungsgebiete repräsentieren und zum anderen unterschiedliche Aspekte der strukturvariablen Modellierung betrachten. Tabelle 8.1 zeigt die Aspekte und inwieweit diese in den Fallbeispielen betrachtet werden. Dabei werden folgende Aspekte unterschieden:

- **Funktionen in Transitionen**: Es sind Funktionen in den Transitionen enthalten (+). Es sind keine Funktionen, sondern nur Mappings notwendig (−).
- **Vergangene Daten zur Initialisierung**: Ein Modus muss unter Berücksichtigung von vorangegangenen Simulationen initialisiert werden (+). Die Daten des vorherigen Modus sind ausreichend (−).
- **Zurückgehen in die Vergangenheit**: Während eines Moduswechsels wird in die Vergangenheit gesprungen (+). Es ist kein Zurückgehen in die Vergangenheit notwendig (−).
- **Zwischenmodus**: Ein eigener Modus wird vorgesehen, um den Moduswechsel durchzuführen (+). Es gibt keine Zwischenmodi (−).
- **Modi in Komponenten**: Es gibt Komponenten mit Modi (+). Modi befinden sich nur auf der obersten Modellebene (−).
  - **Schnittstellen betrachten (bei Modi in Komponenten)**: Schnittstellen müssen bei einem Moduswechsel geändert werden (+). Schnittstellen können bei einem Moduswechsel identisch bleiben (∘).

- **Simulationswerkzeug geändert**: Modi werden in verschiedenen Simulationswerkzeugen simuliert (+). Es wird nur ein Simulationswerkzeug für alle Modi verwendet (−).
- **Zeit gespart**: Das Modell wurde im Vergleich zu einem konventionellen Ansatz schneller simuliert (+). Es war die gleiche Zeit notwendig (○). Es war mehr Zeit notwendig (−).
- **Genauigkeit erhöht**: Die Genauigkeit der Ergebnisse konnte im Vergleich zu einem konventionellen Ansatz erhöht werden (+). Die Genauigkeit blieb ähnlich (○). Die Genauigkeit verschlechterte sich (−).
- **Realitätsnähe**: Das Modell ist kein rein akademisches Beispiel (+). Das Modell ist ein akademisches Beispiel (−).

	Honigmann-Prozess	*Automatik-getriebe*	*Rakete*	*Thermische-Elemente*	*Kühl-kreislauf*	*Satellit*
Funktionen in Transitionen	−	−	+	+	+	−
Vergangene Daten zur Initialisierung	−	−	−	+	+	−
Zurücksetzen in die Vergangenheit	−	−	−	+	−	−
Zwischenmodus	−	+	−	−	−	−
Modi in Komponenten	−	+	+	−	+	−
Schnittstellen betrachten		○	○		+	
Simulationswerkzeug geändert	−	−	−	−	−	+
Zeit gespart	+	○	+	○	+	○
Genauigkeit erhöht	○	○	○	○	+	○
Realitätsnähe	+	+	+	−	+	−

Tabelle 8.1.: Betrachtete Aspekte in den Fallbeispielen

Die Beispiele werden anhand der besprochenen Konzepte modelliert. Die Modelle werden daraufhin in DySMo implementiert und simuliert.

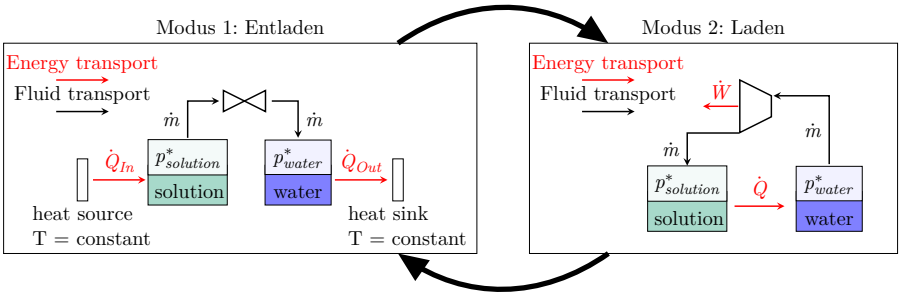


Abb. 8.1.: Schematische Darstellung des Honigmann-Prozesses mit den zwei Modi *Entladen* und *Laden*

8.1. Verhaltensänderungen

Für die Verhaltensänderungen werden zwei Beispiele betrachtet.

Das erste Beispiel ist der Honigmann-Prozess, der von Kollegen entwickelt wurde [41, 42]. Dieser Prozess durchläuft mehrere Phasen, welche durch unterschiedliche Modi dargestellt werden.

Als zweites Beispiel wird ein Automatikgetriebe betrachtet, bei dem mehrere Gänge durch unterschiedliche Modi repräsentiert und die Moduswechseln mit Zwischenmodi durchgeführt werden.[1]

8.1.1. Unterschiedliche Phasen: Honigmann-Prozess

Der Honigmann-Prozess beschreibt das Verhalten eines thermochemischen Energiespeichers, für den Wärme oder mechanische Arbeit zum Laden verwendet werden kann. Die Energie kann später als Kälteleistung oder Strom zur Verfügung gestellt werden.

Der Honigmann-Prozess besteht aus zwei wesentlichen Phasen: Laden und Entladen, welche in Abbildung 8.1[2] gezeigt sind.

Der beladene Speicher kann die Energie durch eine Turbine freisetzen. Die Turbine wird durch einen Massenstrom betrieben, der durch die Druckdifferenz zwischen den Behältern der Salzlösung und des Wassers zustande kommt. Die freigesetzte Wärme der Salzlösung wird dem Wasser zugeführt, was zur Erhöhung des Massenstroms führt. Modus 1 aus Abbildung 8.1 repräsentiert diesen Prozessschritt.

[1] Vorgestellt auf der MSO-TOOLS 2012 in Berlin

[2] Die Abbildung ist angelehnt an die der Autoren aus [42] und wird hier mit deren Erlaubnis verwendet.

Zum Laden des Energiespeichers wird Wärme zum Evaporieren von Wasser aus der Salzlösung verwendet. Der Modus 2 in Abbildung 8.1 zeigt diesen Vorgang schematisch.

Das Modell wurde von den Kollegen des Institutes *Maschinen- und Energieanlagentechnik* der TU Berlin entwickelt, in Modelica implementiert und mittels Dymola simuliert.

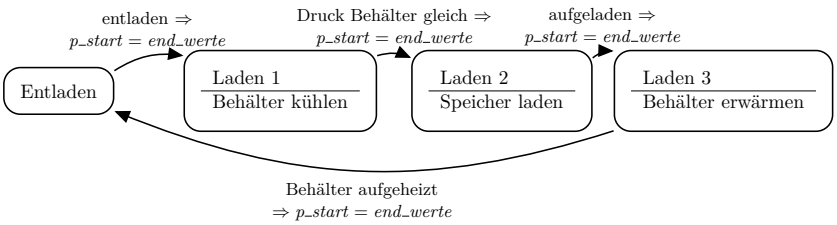


Abb. 8.2.: Modi des Honigmann-Prozesses mit Wechselbedingungen

Umsetzung

Der Ladevorgang wurde in weitere drei Modi unterteilt; die sich ergebenden vier Modi sind in Abbildung 8.2 gezeigt.

Die Moduswechsel finden nur auf der obersten Modellebene statt. Die Kollegen hatten diese Modi bereits in Dymola implementiert und wechselten manuell von einem Modus zum nächsten. Dazu wurde ein Modus gestartet und nach dem Beenden der Simulation dieses Modus wurden die Enddaten der Simulation per Copy und Paste zur Initialisierung des nächsten Modus verwendet. Dies funktionierte, war aber sehr zeitaufwendig und fehleranfällig. Solche Phasenwechsel können durch ein strukturvariables Modell beschrieben werden. In Kooperation mit den Kollegen wurde das Modell als solches aufgebaut.

Um das Modell als strukturvariables Modell zu verwenden, wurden die Modi des Modells erweitert, indem eine Variable *transitionsId* eingeführt wurde, welche die ID der zu aktivierenden Transition bei einem Moduswechsel angibt. In den Modi waren bereits Stoppbedingungen integriert, da ein Stoppen der Simulation schon in der alten Version notwendig war. Die Bedingungen sind in Abbildung 8.2 schematisch beschrieben.

Jeder Modus ist mit Parametern versehen, welche für die Initialisierung des Modells verwendet werden (als *p_start* abstrahiert). Diese Parameter waren bereits im originalen Modell vorhanden und repräsentieren das *start*-Attribut der Formalisierung.

Diese Parameter werden durch ein Mapping anhand der Enddaten des vorher simulierten Modus initialisiert.

Da das Modell nur Modi auf der obersten Modellebene hat und die Modi alle bereits implementiert waren, war der Aufwand zur Erstellung der Modi, die in DySMo verwendet werden können, sehr gering.

Das Modell wurde daraufhin in einer Config-Datei für DySMo beschrieben und mit Dymola als Simulationswerkzeug simuliert.

Auswertung

Durch die automatisierten Moduswechsel wurde zwar die reine Simulationszeit nicht verringert, aber da keine manuelle Bearbeitung der Daten mehr notwendig ist, benötigt die Simulation nur noch wenige Sekunden. Des Weiteren ist die Simulation weniger fehleranfällig, da keine Fehler bei der manuellen Übertragung von Initialwerten mehr auftreten.

Einen weiteren Vorteil bietet der Beobachter, da er die interessierenden Daten in einem einfach nutzbaren Format speichert. In dem vorherigen Modell waren vier Dateien mit Enddaten vorhanden, aus denen die Ergebnisse extrahiert werden mussten.

Die Simulationsergebnisse sind identisch zum originalen Modell, da die Modi und auch die vorher manuell durchgeführte Initialisierung bei den Moduswechseln übernommen wurden.

Das strukturvariable Modell wird seither von den Kollegen für ihre Untersuchungen verwendet.

8.1.2. Zwischenmodus: Automatikgetriebe

Betrachtet wird ein elektronisch gesteuertes 5-Gang-Automatikgetriebe, das von der Daimler-Chrysler AG entwickelt wurde und seit 1996 verbaut wird. Das Getriebe besteht aus drei Planetensätzen, drei Kupplungen, drei Bremsen, zwei Freiläufen und sechs Wellenknotenpunkten.

Der schematische Aufbau des Modells ist in Abbildung 8.3 gezeigt. In diesem Modell gibt es mehrere Komponenten mit Modi. So kann eine Kupplung bzw. Bremse offen, geschlossen oder gleitend sein. Ein Freilauf kann offen oder geschlossen sein. Abbildung 8.4 zeigt exemplarisch die drei Modi einer Kupplung.

In diesem Modell sind Zwischenmodi vorgesehen, damit beispielsweise die Kupplung nicht direkt vom geöffneten Zustand zum geschlossenen übergeht. Der Zwischenmodus gewährleistet den gleitenden Übergang von einem Modus zum nächsten.

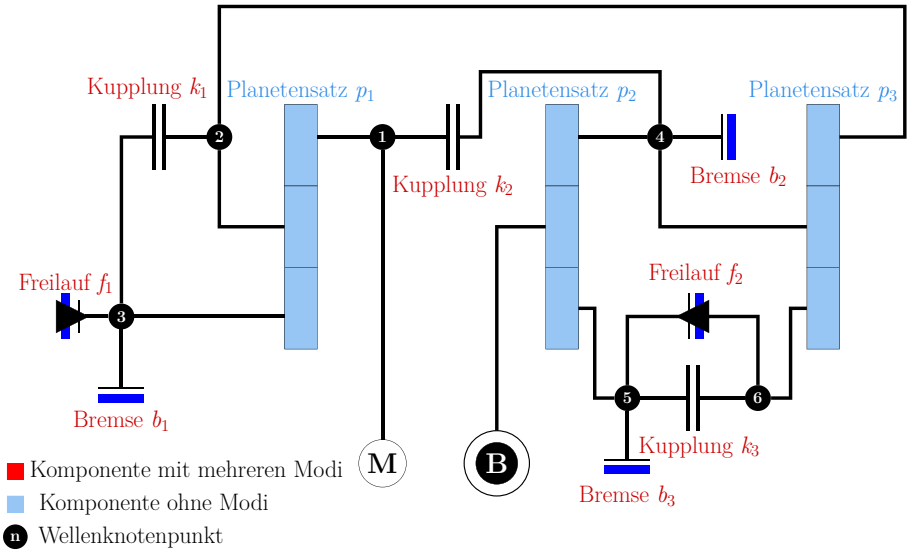
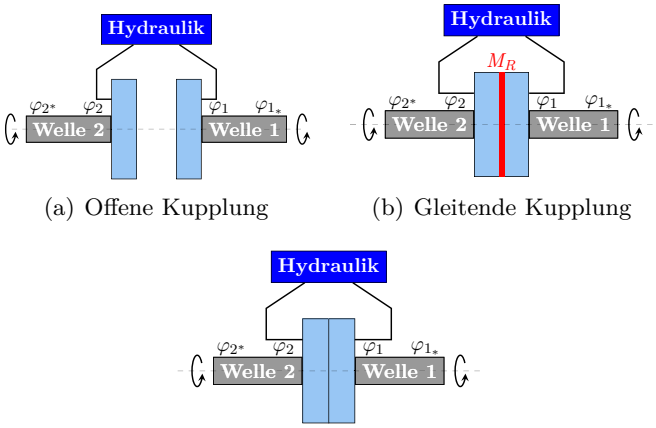


Abb. 8.3.: Schematische Darstellung des Automatikgetriebes, basierend auf [35]



(a) Offene Kupplung (b) Gleitende Kupplung

(c) Geschlossene Kupplung

Abb. 8.4.: Verschiedene Modi der Kupplung (offen, gleitend, geschlossen)

Umsetzung

In der Arbeit [35] wurde das Modell als strukturvariables Modell in FORTRAN implementiert. Dazu wurden die verschiedenen Modi und ein eigener Solver für die Strukturwechsel implementiert. Das Modell wurde daraufhin mit drei Modi (Gang 1, 2, 3) und drei Zwischenmodi (Gang 12, 23, 32) simuliert. Abbildung 8.5 zeigt die Schaltabfolge während der Simulation.

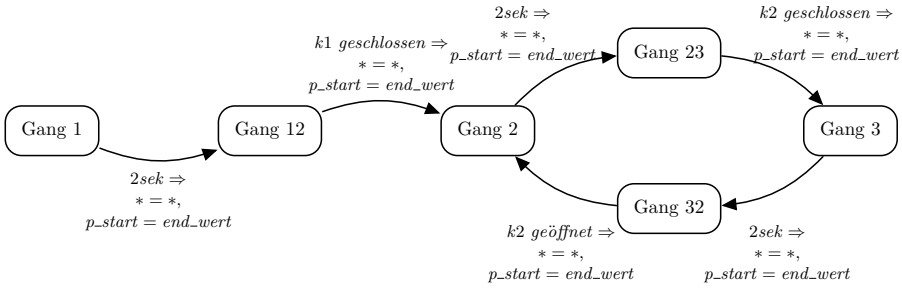


Abb. 8.5.: Schaltvorgänge im Getriebe mit den dazugehörigen Modi

Der Nachteil des FORTRAN-Modells ist, dass die Modi mit Hilfe von Anweisungen beschrieben werden müssen und ein Aufbau aus Komponenten nicht möglich ist; die Wiederverwendbarkeit ist somit eingeschränkt. Auch musste ein spezialisierter Solver für das Modell geschrieben werden. Eine allgemeine Umsetzung für strukturvariable Modelle war mit diesem Ansatz nicht vorgesehen.

Um diese Nachteile zu umgehen, wurde das Modell 2012 im Rahmen einer Studienarbeit in Modelica implementiert [21]. Für jede Komponente mit Modi wurde eine allgemeine Klasse entworfen, welche die Variablen, Gleichungen und Schnittstellen für deren Modi definiert. Die einzelnen Modi erben von dieser Klasse und werden um das spezialisierte Verhalten erweitert. Abbildung 8.6 zeigt solch einen Entwurf für die Kupplung und deren drei Modi. Bei einem Moduswechsel innerhalb einer Komponente ist somit die Kompatibilität der globalen Daten und Schnittstellen sichergestellt.

Die Variablen, die dasselbe repräsentieren, wurden in den Modi identisch benannt. Daher können deren Werte bei der Initialisierung automatisch übertragen werden. Zur Initialisierung zusätzlicher Variablen wurden Parameter vorgesehen, welche bei einem Moduswechsel gesetzt werden (p_start).

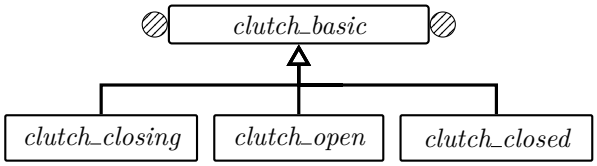


Abb. 8.6.: Vererbungshierarchie für die Modi der Kupplung

Diese Parameter setzen im Modell die entsprechenden Startwerte, somit wird auch hier das *start*-Attribut der Formalisierung abgebildet.

Da DySMo Moduswechsel nur auf oberster Modellebene unterstützt, wurden die sechs Modi als eigene Modelle in Modelica aus den Komponenten zusammengesetzt. Durch den Aufbau der Komponenten und durch die kompatiblen Schnittstellen müssen lediglich die Instanzen der Kupplungen, Bremsen und Freiläufe ausgetauscht werden. Die Verbindungsbeschreibungen bleiben identisch. Die Wechsel zwischen den sechs Modi wurden daraufhin in der Config-Datei von DySMo beschrieben.

Auswertung

In der FORTRAN-Implementierung [35] wurden die Simulationsergebnisse mit Ergebnissen einer Simulation in ASIM[3] verglichen, in der eine vereinfachte Version des Getriebes implementiert war. Es wurde in [35] auf die starken Vereinfachungen der ASIM-Implementierung hingewiesen, wodurch die Ergebnisse nur eingeschränkt zum Vergleich dienen. Messdaten stehen nicht zur Verfügung, um die Simulationsergebnisse zu vergleichen.

Abbildung 8.7 zeigt die Ergebnisse der drei Simulationen. Dabei wurden die Daten der ASIM- und FORTRAN-Simulation der zugrundeliegenden Arbeit entnommen. Die Ergebnisse der DySMo- und FORTRAN-Implementierung ähneln sich, die Ergebnisse der ASIM-Simulation hingegen zeigen größere Abweichungen. Die Abweichung betrifft jedoch im Wesentlichen den zeitlichen Versatz der Winkelgeschwindigkeit und nicht den Verlauf an sich.

Da Referenzdaten fehlen, ist nicht eindeutig, welche Simulation realistischere Daten liefert.

[3] ASIM ist ein in der Forschung entwickeltes Simulationswerkzeug von Daimler-Chrysler.

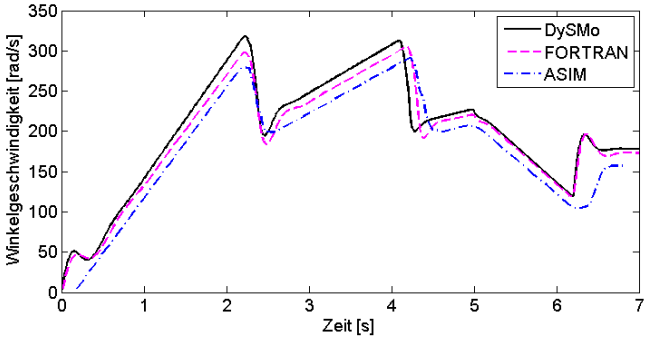


Abb. 8.7.: Winkelgeschwindigkeit der Getriebeeingangswelle für die drei Simulationen: ASIM, FORTRAN, DySMo

Die Simulationszeit des FORTRAN-Modells beträgt $0,9$ Sekunden, während in DySMo $1,1$ Sekunden benötigt werden. Die längere Zeit kommt durch die Kommunikation mit Dymola zustande. Zusätzlich kommt bei der DySMo-Simulation die Zeit zum Kompilieren der Modi hinzu, welche für jeden Modus ca. $0,2$ Sekunden dauert. Die Moduswechsel selbst fallen bei diesem Modell nicht ins Gewicht, wodurch die gesamte Simulation in DySMo ca. $2,3$ Sekunden benötigt.

Der Modelica-Ansatz ist, trotz längerer Simulationszeiten, als positiv zu werten, da der Aufbau des Modells durch den objektorientierten Aufbau einfach und die Erweiterbarkeit und Wiederverwendbarkeit der Komponenten im Vergleich zur FORTRAN-Umsetzung hoch ist. Ebenso muss kein eigener Solver implementiert werden, da die Simulation in Standardwerkzeugen stattfindet, während DySMo die Moduswechsel übernimmt.

8.2. Detaillierungsgradänderung: Rakete

Für den Detaillierungsgradwechsel sei ein Raketenstart betrachtet [50]. Ziel dieses Modells ist es, die Flugbahn einer Rakete zum Mond zu berechnen.

Diese Rakete besteht unter anderem aus drei Boostern und drei Treibstofftanks. Der Antrieb der Rakete wird, im Gegensatz zu den ersten sehr einfachen Raketenmodellen (siehe Beispiel 3.1), anhand der Verbrennung von Treibstoff berechnet. Dazu hat jeder Booster seine eigene Brennkammer, in der die chemischen Reaktionen stattfinden.

Aus den chemischen Reaktionen ergeben sich eine Temperatur und ein Druck, mit denen eine Austrittsgeschwindigkeit aus den Boostern bestimmt wird. Diese lässt sich in Schub umrechnen.

Beim Start der Rakete wird mit dem ersten Booster und dessen Brennkammer begonnen. Sobald der erste Tank leer ist, wird dieser mit seiner Brennkammer abgeworfen und der nächste Booster verwendet.

Zusätzlich wird die Atmosphäre der Erde betrachtet, solange diese noch Einfluss auf die Rakete hat.

Umsetzung als nicht strukturvariables Modell

Das Modell wurde in Modelica implementiert und in Dymola simuliert. Das Modell wurde mit einem Booster abstrahiert, welcher einmal pro Stufe verwendet wird. Dieser Booster enthält die chemischen Reaktionen und berechnet diese während der gesamten Simulation. Wird ein eineinhalbtägiger Flug in Dymola simuliert, ergibt sich eine Simulationszeit von 50 Sekunden und ca. $1,5$ GB an gespeicherten Daten. Diese lange Zeit und große Menge an Daten ergeben sich durch die kleinen Schrittweiten, die durch die Berechnung der chemischen Reaktionen notwendig sind. Bei einem Modell mit ca. 300 Gleichungen ist solch eine lange Simulationszeit und der große Speicherbedarf jedoch unerwünscht.

Umsetzung als strukturvariables Modell

Abbildung 8.8 zeigt den berechneten Schub des Boosters. Dieser ändert sich nur während der Startphase des Boosters wesentlich und ist danach nahezu konstant.

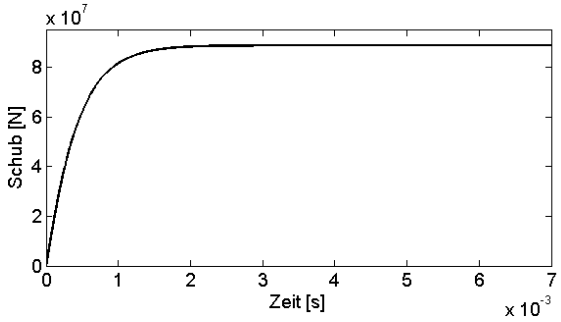


Abb. 8.8.: Schub des ersten Boosters

Aus diesem Grund kann die Verbrennungsrate und der Schub nach der Startphase durch konstante Werte abstrahiert werden. Erst wenn der Tank leer ist und ein neuer Booster aktiviert werden muss, müssen die chemischen Reaktionen wieder berücksichtigt werden. Abbildung 8.9 zeigt die Komponenten des Modells.[4]

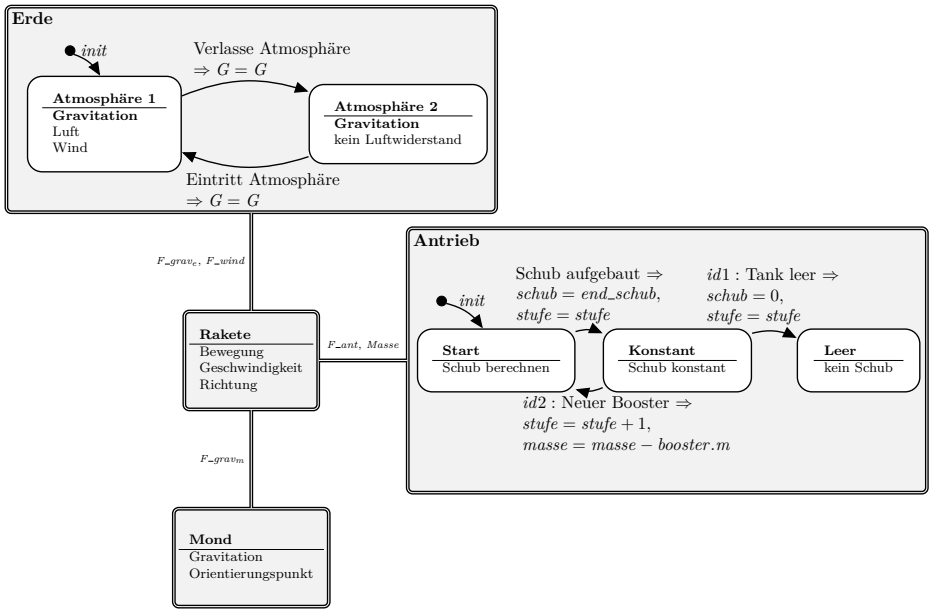


Abb. 8.9.: Schematische Sicht auf die Komponenten und Modi des Raketenmodells

In der *Erde*-Komponente werden die Gravitation (F_grav_e), welche auf die Rakete wirkt, sowie die Einflüsse der Atmosphäre (F_wind) bestimmt. Diese Komponente besteht aus zwei Modi. In dem einen Modus hat die Atmosphäre einen Einfluss, in dem anderen nicht. Die Gravitation (G) soll bei einem Moduswechsel identisch bleiben.

In der *Antrieb*-Komponente werden die chemischen Reaktionen berechnet, welche den Schub (F_ant) für den Antrieb der Rakete liefern. Dabei ist der Antrieb in drei Modi aufgeteilt. Der initiale Modus startet den Booster; ist der Schub aufgebaut, wird in den zweiten Modus gewechselt.

[4]Der Übersicht halber sind die Schnittstellen und Verbindungsbeschreibungen nicht dargestellt.

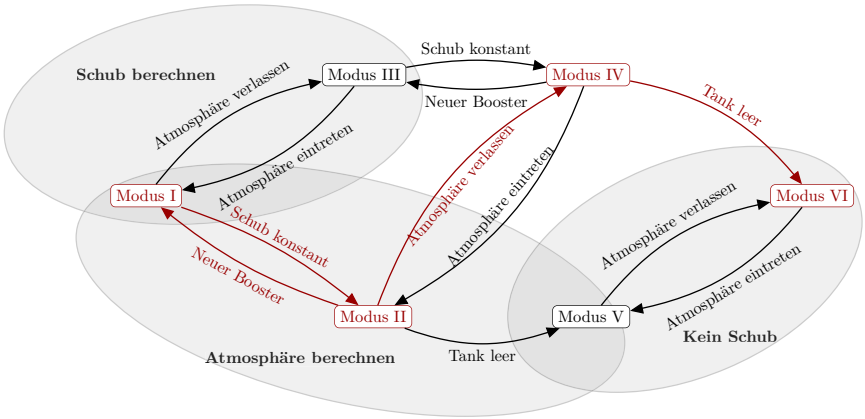
Ist der Booster leer und gibt es noch einen weiteren, so wird der Booster erneut gestartet. Dabei wird die Stufe angepasst, die angibt, welcher Booster aktiv ist. Es werden nicht drei separate Booster mit je drei Modi erzeugt, da dies für dieses Modell mehr Implementierungsaufwand bedeutet und jeweils nur ein Booster aktiv sein kann.

Zusätzlich wird die Masse des Antriebs bei einem Moduswechsel angepasst, da der Booster samt Tank abgeworfen wird. Dieses Ändern der Masse bewirkt bei einem Moduswechsel auch ein Ändern der Raketenmasse. Ist der Tank leer und kein Booster übrig, wird in den Modus ohne Antrieb gewechselt, wobei auch hier die Masse des Antriebs durch den Boosterabwurf angepasst wird.

Die *Mond*-Komponente liefert die Gravitationskraft (F_grav_m) des Mondes auf die Rakete. Zusätzlich wird diese Komponente als Orientierungspunkt verwendet, da der Mond das Ziel der Rakete ist.

Die Bewegung der Rakete wird in der Komponente *Rakete* berechnet. Diese verwendet die Kräfte aus den anderen Komponenten und bestimmt die Flugbahn der Rakete.

Wird das Modell ausfaktorisiert, ergeben sich sechs Modi. Diese sind in Abbildung 8.10 mit ihren Wechselbedingungen dargestellt.



Moduswechsel: I -> II -> I -> II -> I -> II -> IV -> VI

Abb. 8.10.: Modi des Raketenmodells mit Wechselbedingungen

Die Modi in den Komponenten werden mit identischen Schnittstellen implementiert. Dabei werden die Schnittstellen der Modi *Antrieb.Start* und *Atmosphäre.Atmosphäre 1* verwendet, da diese bereits Komponenten des konventionellen Modells waren.

Die Verbindungsbeschreibungen, *connect*-Gleichungen in Modelica, müssen nicht angepasst werden. In den jeweiligen ausfaktorisierten Modi werden nur die Instanzen der sich ändernden Komponente angepasst. Das Erstellen der sechs Modi stellt daher wenig Aufwand dar.

Die Daten, die während eines Moduswechsels zu initialisieren sind, sind in Abbildung 8.9 angedeutet. Diese werden durch Mappings bzw. eine Funktion initialisiert. Alle nicht angegebenen Daten, die in beiden Modi einer Transition enthalten sind, werden mit den Enddaten der vorherigen Simulation initialisiert.

Auswertung

Das beschriebene Modell wird für eineinhalb Tage simuliert. Mit den gewählten Parametern und Startwerten des Modells ergeben sich die Moduswechsel, wie sie in Abbildung 8.10 dargestellt sind.

Durch die Umsetzung als strukturvariables Modell kann die Anzahl der Gleichungen bei konstantem Schub um ca. 100 Gleichungen reduziert werden, wodurch von 24 Zustandsvariablen nur noch 12 übrig bleiben. Die Simulationszeit konnte wesentlich reduziert werden, da die Gleichungen der chemischen Reaktionen entfernt wurden und damit die Schrittweite des Solvers vergrößert werden konnte. Die benötigte Simulationszeit für die Startphase der Rakete ist in Abbildung 8.11 dargestellt.

Die Simulation dauert ca. $0,15$ Sekunden, was im Vergleich zu den vorherigen 50 Sekunden eine erhebliche Zeitersparnis ist.

Zu der reinen Simulationszeit kommen noch die Zeiten zum Kompilieren der Modi und die der Moduswechsel hinzu. Die Gesamtzeit mit diesem Overhead beträgt $3,55$ Sekunden. Dabei sind 3,1 Sekunden für das Kompilieren und $0,3$ Sekunden für die Moduswechsel notwendig. Die kompilierten Modi 1 und 2 wurden mehrere Mal verwendet. Trotz dieses Overheads benötigt die Simulation nur 8% der vorherigen Simulationszeit.

Die meiste Zeit wird gespart, wenn der Treibstofftank leer ist. Aus diesem Grund wurde ein weiteres Modell erstellt, welches nur zwei Modi hat: einen Modus, der die verschiedenen Stufen simuliert, und einen Modus, der erst im Flug im Weltall aktiviert wird, nachdem der Treibstofftank leer ist.

Auch diese Zeit ist in Abbildung 8.11 aufgetragen. Die reine Simulationszeit ist höher als bei vier Modi. Da jedoch nur noch zwei Kompilationen notwendig sind, sinkt die Gesamtsimulationszeit auf $2,5$ Sekunden.

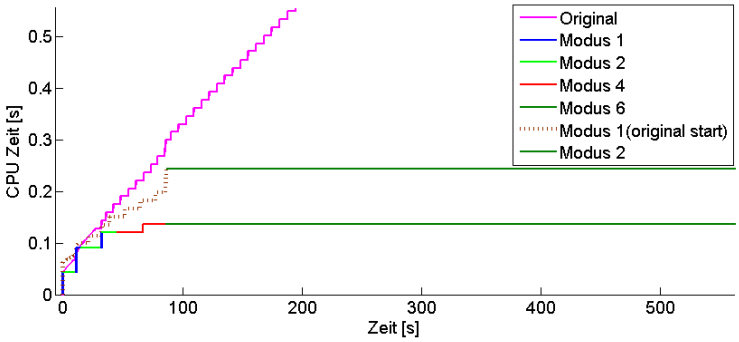


Abb. 8.11.: Simulationszeit der drei Raketenmodelle: Ein Modus, Vier Modi und zwei Modi

Die Ergebnisse sollten durch die Moduswechsel nicht negativ beeinflusst werden. Abbildung 8.12 zeigt die Entfernung der Rakete von der Erde für das Modell mit Strukturwechseln (vier Modi[5]) und ohne.

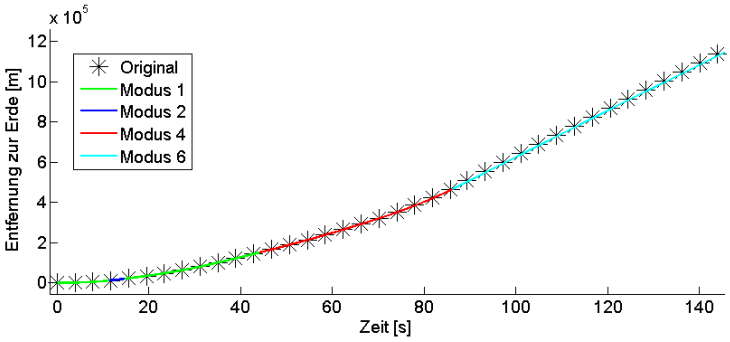


Abb. 8.12.: Abstand der Rakete zur Erde für die Modelle: Ein Modus, vier Modi

In Abbildung 8.13 ist die prozentuale Abweichung der Flughöhe gezeigt. Die Abweichungen der Simulationen sind vernachlässigbar gering. Mit den vorgenommenen Änderungen konnte somit die Simulation beschleunigt werden, während die Ergebnisse nicht negativ beeinflusst wurden.

[5]Die Ergebnisse des Modells mit zwei Modi sind identisch und werden der Übersicht halber nicht mit dargestellt.

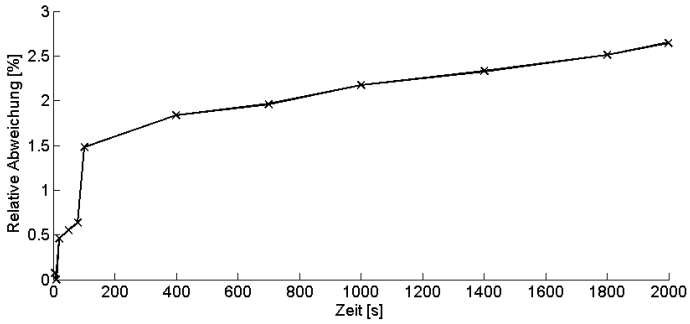


Abb. 8.13.: Relative Abweichung des Simulationsergebnisses für die Höhe der Rakete

Ein weiterer Nachteil des Modells ohne Strukturänderungen war die Datenmenge von 1.5 GB. Mit dem strukturvariablen Modell und sechs Modi konnte dieser Speicherbedarf auf ca. 1MB reduziert werden. Das Modell mit zwei Modi benötigt ca. 30 MB.

Aus diesem Beispiel ist zu erkennen, dass beim Entwurf von strukturvariablen Modellen nicht so viele Modi wie möglich erstellt werden sollten, sondern für jeden Anwendungsfall individuell entschieden werden sollte, welche Modi vorteilhaft sind.

8.3. Hinzufügen/Entfernen von Modellkomponenten

Beim Hinzufügen/Entfernen von Komponenten können zwei Varianten unterschieden werden. In der ersten Variante ändert sich die Anzahl von gleichen Komponenten (Gridänderung). Zur Demonstration wird ein thermischer Leiter betrachtet, der in mehrere Elemente geteilt wird.

Die zweite Variante betrifft das Hinzufügen/Entfernen von beliebigen Komponenten im Modell. Anhand eines Klimamodells, welches von Kollegen der TU Hamburg-Harburg erstellt wurde, wird dies gezeigt [51].

8.3.1. Anzahl von gleichartigen Komponenten / Schalten in die Vergangenheit: Thermische Elemente

Es wird ein thermischer Leiter betrachtet, der aus mehreren thermischen Elementen aufgebaut ist. Betrachtet seien thermische Elemente, welche am linken Ende entweder erwärmt oder gekühlt werden, siehe Abbildung 8.14.

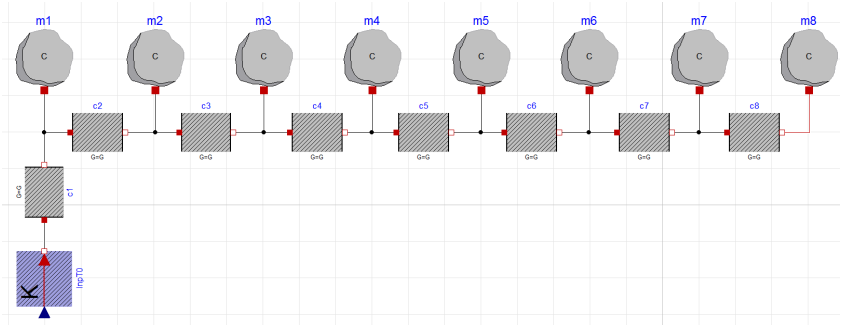


Abb. 8.14.: Modell für die thermischen Elemente in Dymola

Die Temperatur in den Elementen ändert sich und strebt einen Gleichgewichtszustand an. Ist dieser erreicht, so haben die Elemente annähernd die gleiche Temperatur.

Es wird zweimal erwärmt und einmal gekühlt und im Original mit acht Elementen gerechnet. Abbildung 8.15 zeigt die Temperaturverläufe der acht Elemente für eine 22-stündige Simulation. Dabei ist der Gleichgewichtszustand nach einer Heiz- bzw. Kühlperiode zu erkennen.

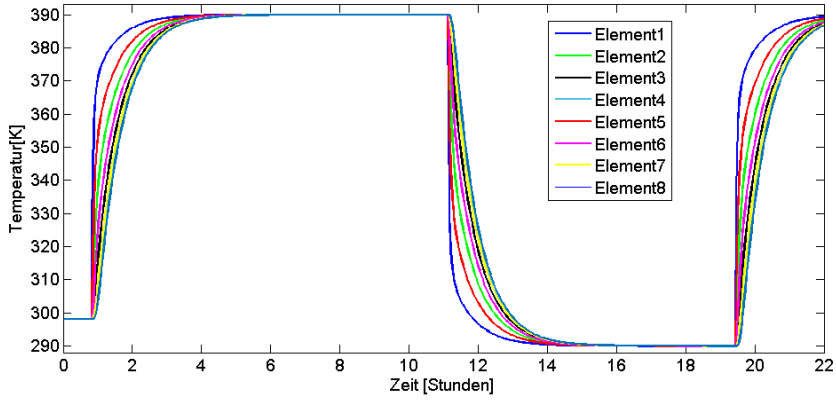


Abb. 8.15.: Temperaturverlauf der acht Elemente bei einer Simulation von 22 Stunden

Umsetzung

Das Modell soll mit einer unterschiedlichen Anzahl von Elementen simuliert werden. Die Idee besteht darin, nur zwei Elemente zu verwenden, wenn die Temperaturen in den Elementen (fast) identisch sind. Ansonsten sollen acht Elemente verwendet werden.

Das Modell besteht aus einem Modus, wobei es zwei Transitionen aus dem Modus in sich selbst zurück gibt, siehe Abbildung 8.16. Bei der ersten Transition wird die Anzahl der Elemente erhöht. Bei der Initialisierung wird die Temperatur T_1 für die ersten vier Elemente und T_2 für die hinteren vier Elemente verwendet.

Bei der zweiten Transition wird die Anzahl der Elemente verringert. Dabei wird der Mittelwert der ersten vier Elemente zur Initialisierung von T_1 verwendet und der Mittelwert der hinteren vier für T_2.

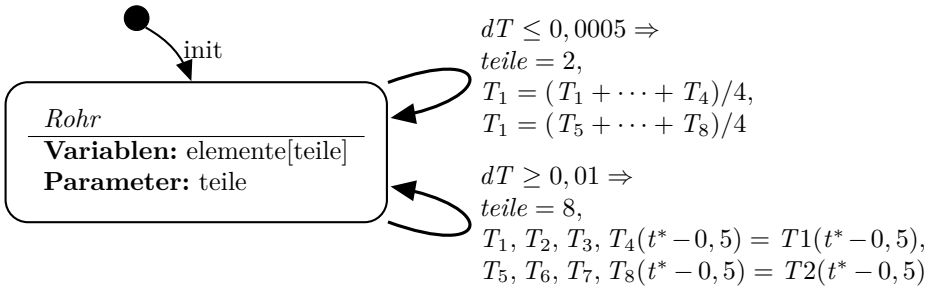


Abb. 8.16.: Strukturvariables Modell der Rohrleitung

Da bei der Änderung der Anzahl der Elemente ein erneutes Kompilieren des Modus notwendig ist, ergeben sich zwei kompilierte Modi für das Modell. Modus 1 hat zwei Elemente, während Modus 2 die acht Elemente beschreibt.

Die Simulation beginnt mit dem ersten Modus, da von einem ausgeglichenen Zustand ausgegangen wird. Nach einer Sekunde wird die linke Seite erwärmt, wodurch sich die Temperatur im ersten Element zu ändern beginnt. Diese Änderung wird im Modus 1 als Ereignis detektiert, welches eine Transition auslöst.

Da die Elemente in Modus 1 größer sind als die Elemente aus Modus 2, verändern sich die Temperaturen langsamer. Ein Ereignis, welches die Temperaturänderung betrifft, wird daher später detektiert. Zusätzlich hat sich die Temperatur im ganzen ersten Element geändert.

Wird nun zum Modus 2 gewechselt, muss diese Temperatur auf vier Elemente projiziert werden. Allerdings sollte in diesen vier Elementen nicht die gleiche Temperatur herrschen. Aus diesem Grund wird ein Schalten um ca. 5 Minuten in die Vergangenheit vorgesehen, sodass der zweite Modus mit ausgeglichenen Temperaturen initialisiert werden kann und bereits der Beginn der Temperaturänderung im detaillierteren Modus berechnet wird.

Die Simulation wird nun fortgesetzt, bis keine wesentlichen Temperaturänderungen in den Elementen detektiert werden. Daraufhin wird wieder in den Modus 1 gewechselt. Dies setzt sich für alle Heiz- und Kühlperioden fort.

Auswertung

Abbildung 8.17 zeigt die Simulationsergebnisse der Temperatur am linken Ende des Rohres für 22 Stunden.

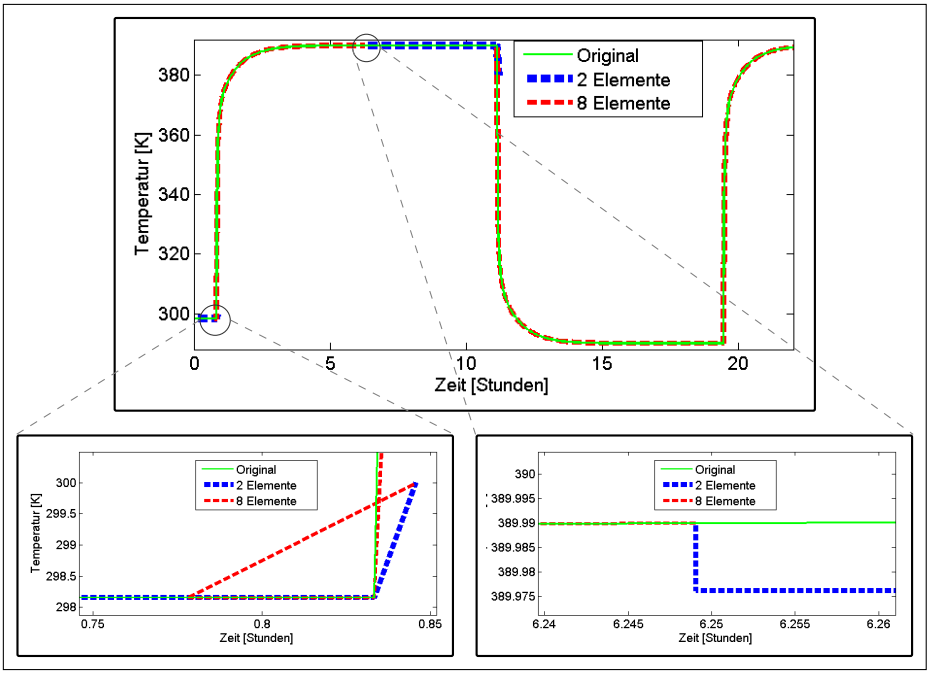


Abb. 8.17.: Simulationsergebnisse der Temperatur für das strukturvariable Modell

Größer dargestellt ist der erste Moduswechsel 1–2. Hier ist das Zurückgehen in die Vergangenheit zu sehen. Erst ca. 1 Minute nachdem sich die Temperatur ändert, wird in dem vereinfachten Modell das Ereignis detektiert und der Verlauf ist ein anderer als im originalen Modell.

Es findet ein Zurücksetzen in die Vergangenheit statt und der Modus 2 wird mit der noch unveränderten Temperatur initialisiert. In der erneuten Simulation findet die Temperaturänderung dann analog zum originalen Modell statt.

Ebenso ist der zweite Moduswechsel 2–1 dargestellt. In diesem ist der Sprung der Temperatur durch die Mittelwertbildung zu sehen. Der Fehler ist gering und beeinflusst die Simulationsergebnisse nicht negativ (y-Achsenbeschriftung beachten), führt aber einen geringen Fehler ein.

Die entstehenden Fehler bei der Mittelwertbildung führen zu abweichenden Ergebnissen zum Original. Jedoch ist dies bei Gridänderungen nicht zu vermeiden, da in den seltensten Fällen die Werte in allen Elementen identisch sind. Der Vorteil eines solchen Vorgehens ist, dass weniger Variablen im Modell vorhanden sind und damit in vielen Fällen Simulationszeit gespart werden kann. In dem hier gezeigten Beispiel trat dieser Vorteil nicht ein, da das Modell auf Grund seiner Einfachheit wenig Zeit für die Simulation beansprucht.

8.3.2. Komponenten hinzufügen/löschen: Kühlkreislauf

In diesem Beispiel geht es um ein Kühlkreislaufmodell zum Kühlen von HEV-Batterien (Hybrid Electrical Vehicle). Dabei soll eine Kühlung nur erfolgen, wenn die Batterie zu heiß wird.

Das Modell wurde mit der *AirCondition-Library* von den Kollegen der TU Hamburg-Harburg in Dymola implementiert und simuliert. In Zusammenarbeit mit den Kollegen wurde das Modell als strukturvariables Modell aufgebaut [51].

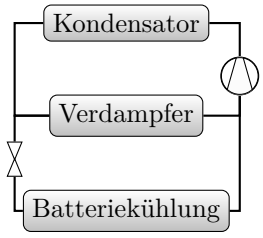


Abb. 8.18.: Kühlkreislauf in einem Elektrofahrzeug mit Batteriekühlung

Der schematische Aufbau des Kühlkreislaufs ist in Abbildung 8.18 dargestellt. Ist ein Kühlen der Batterie erforderlich, so wird das Ventil für den Kühlkreislauf geöffnet. Es wird wieder geschlossen, wenn keine Kühlung mehr erforderlich ist.

Das Modell wird mit geöffnetem Ventil gestartet. Nach 30 Sekunden wird das Ventil geschlossen und ist bei 40 Sekunden vollkommen geschlossen. Das Ventil bleibt bis zum Zeitpunkt von 160 Sekunden geschlossen und fängt dann an sich zu öffnen, bis es bei 170 Sekunden wieder vollständig geöffnet ist. Abbildung 8.19 zeigt den Durchflussfaktor (kv) für das Ventil.

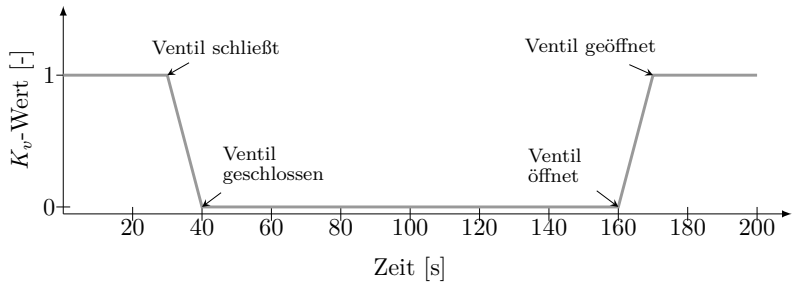


Abb. 8.19.: Verlauf des kv-Wertes für die Simulation des Kühlkreislaufs

Die Schwierigkeit bei diesem Modell ist die Simulation bei geschlossenem Ventil, da der Massenstrom durch das Ventil null wird und dies beim Lösen des Gleichungssystems zu Problemen führt. Für die numerische Berechnung bleibt ein minimaler Massenstrom übrig. Dieser minimale Massenstrom führt zu kleinen Schrittweiten und zu Schwankungen im Massenstrom, die bei geschlossenem Ventil nicht auftreten dürften.

Umsetzung als strukturvariables Modell

Um die oben beschriebenen Probleme zu umgehen, wurde das Modell als strukturvariables Modell aufgebaut. Prinzipiell hat der wegfallende Strang zwei Modi. Wie bereits in Abschnitt 4.3.2 diskutiert ist es in einem solchen Fall jedoch einfacher, die Moduswechsel auf einer höheren Ebene vorzusehen. Es muss somit für den wegfallenden Teil kein leerer Modus vorgesehen werden. Fallen die Komponenten des unteren Strangs weg, so muss die Schnittstelle des statischen Teils des Modells angepasst werden, damit keine Verbindungsbeschreibungen übrig bleiben, die auf einer Seite nicht verbunden sind.

Das Modell besteht somit aus zwei Modi: einem Modus mit geöffnetem Ventil und einem mit geschlossenem Ventil, in dem der Zweig für die Batteriekühlung nicht mehr enthalten ist. Abbildung 8.20 zeigt die beiden in Modelica implementierten Modi. Die Schnittstelle der *Verteiler*, an die der untere Zweig gekoppelt war, wurde in Modus 2 entfernt.

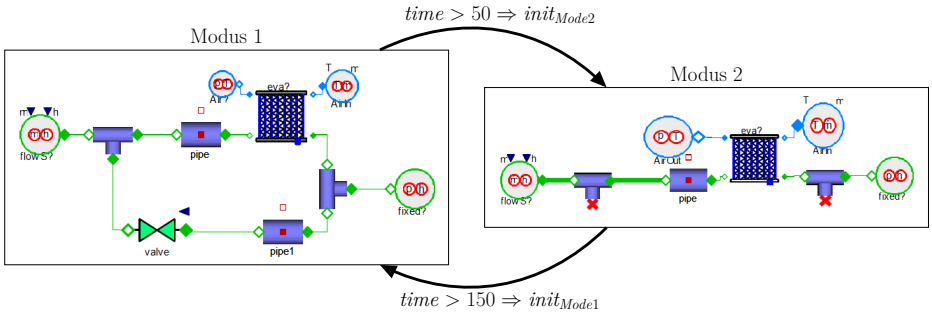


Abb. 8.20.: Modell des Kühlkreislaufs mit zwei Modi, dargestellt mit den Dymola-Komponenten

Die Moduswechsel werden in diesem Modell abhängig von der Zeit eingeleitet. Dabei wurde darauf geachtet, die Moduswechsel in einem stabilen Zustand des Modells einzuleiten. Die überflüssigen Komponenten wurden erst entfernt, nachdem das Ventil bereits einige Sekunden geschlossen war. Ebenso wurden die Komponenten hinzugefügt, bevor das Ventil geöffnet wurde. So kann sichergestellt werden, dass in einem unkritischen Bereich gewechselt wird.

Zur Initialisierung der Modi werden alle gleich benannten Variablen auf dieselben Werte gesetzt. Bei diesem Modell sind bei der Initialisierung zwei Dinge interessant:

1. In der *AirCondition-Library* wird mit Initialgleichungen gearbeitet. Wird jedoch nach einem Strukturwechsel initialisiert, sind diese Zusammenhänge nicht immer gültig. Die Bibliothek bietet für solche Fälle die Möglichkeit, die Initialgleichungen und damit die zusätzlichen algebraischen Bedingungen bei der Initialisierung zu deaktivieren. Werden sie nicht deaktiviert, ist eine gewünschte Initialisierung oft nicht möglich.

2. Wird in den ersten Modus zurück gewechselt, stellt sich die Frage, wie die nun hinzugefügten Komponenten initialisiert werden sollen. In diesem Beispiel wurden die Enddaten der alten Simulation des ersten Modus verwendet. Es wurde somit angenommen, dass die Werte konstant bleiben. Zum Auslesen der alten Daten wurde eine Init-Funktion verwendet, welche die Daten für die Initialisierung der hinzukommenden Komponente verwendet.

Auswertung

Die benötigte Zeit für eine Simulation von 200 Sekunden ist in Abbildung 8.21 dargestellt.

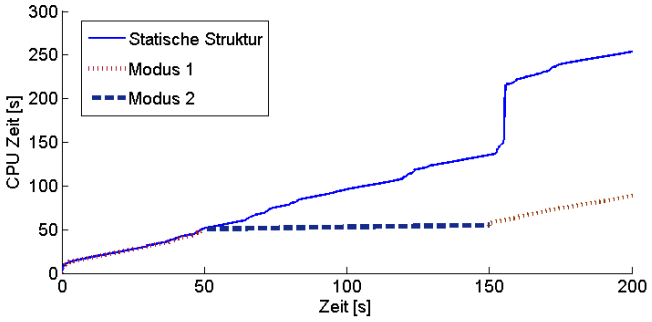


Abb. 8.21.: Benötigte Simulationszeit für die Modelle des Kühlkreislaufs

Die Simulationszeit mit Strukturänderungen ist wesentlich geringer im Vergleich zum Modell mit statischer Struktur. Allerdings müssen beim strukturvariablen Modell zwei Modi kompiliert werden im Vergleich zu einem einzelnen. Dies führt zu einer Gesamtzeit der Simulation von:

- Original: 266 Sekunden
- Strukturvariabel: 113 Sekunden

Trotz des Mehraufwands beim Kompilieren der Modi wurde durch den strukturvariablen Ansatz mehr als die Hälfte der Simulationszeit gespart.

Auch hier sollten die Simulationsergebnisse durch die Moduswechsel nicht negativ beeinflusst werden. Abbildung 8.22 zeigt den Druck am Einlass des Verdampfers für beide Modelle. Die Ergebnisse unterscheiden sich nicht wesentlich voneinander. Bei der Simulation ohne Strukturänderungen sind leichte Druckschwankungen zu erkennen, während das Ventil geschlossen ist. Diese treten durch den minimalen Massenstrom durch das

Ventil auf. Diese Schwankungen treten bei dem Modell mit Strukturänderungen nicht mehr auf. Die Simulation wurde somit nicht nur schneller, sondern sogar genauer.

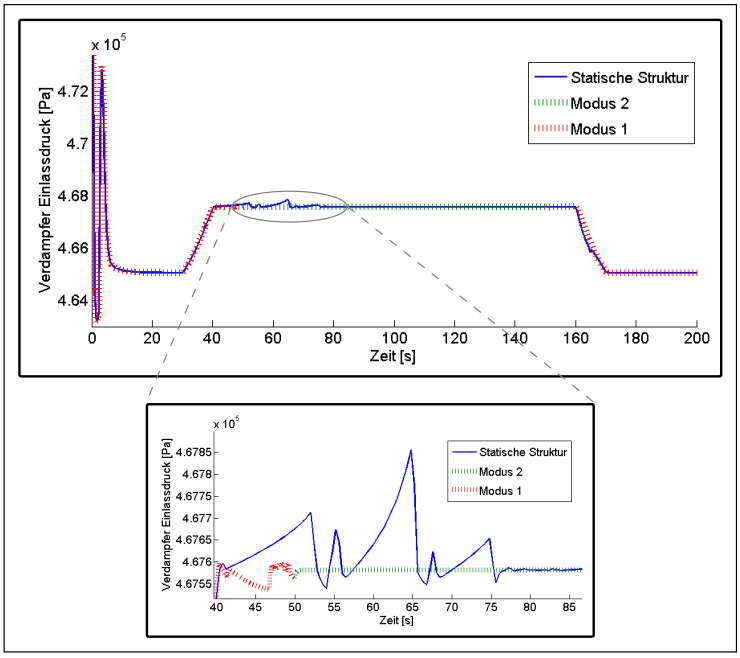


Abb. 8.22.: Simulationsergebnisse für den Druck am Einlass des Verdampfers

8.4. Verschiedene Simulationsumgebungen: Satellit/Rakete

Im Framework ist es möglich, für verschiedene Modi verschiedene Simulationswerkzeuge zu verwenden. Dies ist beispielsweise bei Detaillierungsgradänderungen sinnvoll, wenn von einem Modelica-Modell zu einem FEM-Modell gewechselt werden soll. Zur Zeit ist noch kein FEM-Werkzeug in DySMo integriert, aus diesem Grund wird für die Detaillierungsgradänderung kein separates Beispiel angegeben.

Ein anderer Anwendungsfall ist das Verwenden von komplett verschiedenen Modellen, wobei diese Modelle in unterschiedlichen Werkzeugen implementiert sind. Ein Beispiel ist das bereits in Abschnitt 7.4 vorgestellte Modell, siehe auch [49]. In diesem Beispiel wird ein in Simulink implementiertes Raketenmodell verwendet, um einen in Modelica modellierten Satelliten in den Orbit zu bringen. Dabei könnte die Rakete beispielsweise beim Start durch mehrere Regelstrategien gesteuert werden, folglich wäre Simulink ein gutes Werkzeug für eine solche Simulation. Modelica hingegen ist gut für die Berechnung der Satellitenbewegung im Orbit geeignet.

Da Modelle in verschiedenen Werkzeugen oft zu verschiedenen Zeiten oder von verschiedenen Modellierern implementiert wurden, sind die Modelle wahrscheinlich sehr unterschiedlich. Meist werden hier Funktionen zur Startwertbestimmung notwendig sein. Dem Modellierer sollte durch gute Dokumentation der Modelle geholfen werden, korrekte Initialisierungsmöglichkeiten für die Modi zu finden.

8.5. Co-Simulation

Die lose sequentielle Kopplung bei der Co-Simulation kann als strukturvariables Modell angesehen werden (vgl. Abschnitt 4.6) und wäre mit DySMo umsetzbar.

Die Wechselbedingungen könnten entweder in äquidistanten Zeitschritten stattfinden oder anhand eines beliebigen Schaltkriteriums.

Da dies ein etabliertes Vorgehen ist, wird es hier nicht separat betrachtet. Es sind bereits Co-Simulationsplattformen für solche Modelle vorgesehen und DySMo ist speziell für diesen Anwendungsfall weniger geeignet. Wie in Abschnitt 7.5.1 diskutiert, liegen die Stärken von DySMo eher bei großen Modellen mit wenigen Wechseln als bei Modellen mit sehr vielen Wechseln, was in der Co-Simulation der Fall ist.

8.6. Konsistente Initialisierung

Wie in Abschnitt 4.7 erläutert, sind existierende Ansätze für die Initialisierung von DAE-Systemen ähnlich zu strukturvariablen Modellen. Dabei ist der Moduswechsel für den Benutzer meist nicht ersichtlich, da er während der Initialisierungsphase im Simulationswerkzeug stattfindet. Es sollte bedacht werden, dass verschiedene Werkzeuge oft unterschiedliche Algorithmen implementieren, was im Extremfall bei gleichen Modellen und

gleichen Initialwerten zu unterschiedlichen Simulationsergebnissen führen kann.

Eigene Initialisierungen zu implementieren und den Strukturwechsel selbst vorzunehmen, ist nur sinnvoll, wenn für das spezielle Problem eigene Optimierungen oder Modelle bessere Startwerte liefern. So kann beispielsweise ein stationäres Modell simuliert werden, um Startwerte für ein komplizierteres Modell zu finden. Soll dies mit DySMo geschehen, können Python-Algorithmen vor der Simulation oder auch eigene Modelle in einem der Simulationswerkzeuge verwendet werden.

Da die möglichen Initialisierungen und die verwendeten Algorithmen in der Fachliteratur und den Publikationen (bspw. [8, 10, 11, 15, 44, 89, 93]) erläutert werden und in dieser Arbeit keine neuen Algorithmen oder Vorgehensweisen für diesen Kontext erstellt wurden, wird an dieser Stelle kein separates Beispiel angegeben.

8.7. Zusammenfassung

Die Fallstudien präsentierten verschiedene Modelle der unterschiedlichen Anwendungsgebiete und verschiedene Modellierungskonzepte. Für die Modelle aus Projekten wurden Vorteile bei der Simulationszeit oder Genauigkeit erreicht. Die kleineren, akademischen Beispiele zeigen weniger Vorteile, da sie dazu ausgelegt waren, verschiedene Aspekte der strukturvariablen Modellierung zu zeigen. In realen, größeren Kontexten sind jedoch Vorteile zu erwarten.

Für Verhaltensänderungen, bei denen sich Prozesse ändern (Bsp. rutschende Kupplung, Reibung), sind strukturvariable Modelle oft erforderlich, solange das Modell nicht weiter abstrahiert werden soll. Gleiches gilt beim Wechsel des Simulationswerkzeugs.

Anders sieht es beim Detaillierungsgradwechsel und beim Hinzufügen und Entfernen von Komponenten aus. Modellwechsel sind bei diesen Anwendungsfällen nicht immer notwendig, da das genauere Modell oft ausreichend ist. Jedoch führt die genauere Betrachtung oft zu langen Simulationszeiten, was nicht erwünscht ist.

Co-Simulationen und konsistente Initialisierungen sind eher ein Fall für ein konventionelles Vorgehen und sollten nur in Spezialfällen mit strukturvariablen Modellen umgesetzt werden.

Strukturvariable Modelle sind für alle genannten Anwendungsgebiete einsetzbar, es sollte jedoch immer überprüft werden, ob durch ihren Einsatz positive Effekte zu erwarten sind.

III

Zusammenfassung und Ausblick

9 Kritische Betrachtung

In diesem Kapitel werden die in den vorherigen Kapiteln behandelten Themen kritisch betrachtet und zusammengefasst.

Bei der Modellierung werden die Voraussetzungen, die ein Modell erfüllen muss, um vorteilhaft im Vergleich zu konventionellen Modellen eingesetzt zu werden, zusammengefasst.

Die vorgestellte Formalisierung und die graphische Notation werden auf ihre Anwendbarkeit und ihren Nutzen in Bezug auf strukturvariable Modelle betrachtet.

Das Framework wird auf seinen praktischen Nutzen hin evaluiert. Es wird erläutert, unter welchen Voraussetzungen DySMo Vorteile in der Modellierung und Simulation von Strukturänderungen bringt.

9.1. Modellierung strukturvariabler Modelle

Die Auswahl der Modi spielt eine entscheidende Rolle, da jeder Modus das zu betrachtende System für ein Zeitintervall beschreibt. Werden viele Modi gewählt, ähneln sich diese meist und sind eventuell nicht notwendig, da der Mehraufwand nicht durch die Vorteile bei der Simulation gerechtfertigt wird.

Die Modi müssen allerdings hinreichend ähnlich sein, damit während eines Moduswechsels Initialwerte für den neuen Modus bestimmt werden können. Um die Initialisierung zu vereinfachen, sollte während der Modellierung im Modus markiert werden, welche Variablen für die Initialisierung geeignet sind.

Sind die Modi ausgelegt, müssen die Moduswechsel mit deren Initialisierung des Folgemodus und der Wechselbedingung beschrieben werden.

Bei der Initialisierungsbeschreibung können Berechnungsschritte, Daten aus alten Simulationen oder benutzerdefinierte Werte erforderlich sein. Jede Initialisierung führt zu einem neuen Initialwertproblem, welches zu Berechnungs- und Rundungsfehlern führen kann. Aus diesem Grund ist

es wichtig, die Wechselbedingung möglichst in stabilen Bereichen vorzusehen. Ist der aktuelle Modus gerade in einer dynamischen oder kritischen Phase, so sollte zu dieser Zeit nicht gewechselt werden. Kann eine Wechselbedingung erst in solch einer Phase detektiert werden, so sollte der nächste Modus in der Vergangenheit gestartet werden, sofern dort bessere Bedingungen herrschten.

Jeder Moduswechsel benötigt zudem Zeit zum eventuell notwendigen Kompilieren des neuen Modus und zur Bestimmung der neuen Startwerte. Daher sollten nicht beliebig viele Moduswechsel vorgesehen werden, wenn dies nicht zu wesentlichen Vorteilen führt.

Komponenten mit Modi

Modi und die Wechsel zwischen ihnen sollten lokal in Komponenten gekapselt werden, um diese in verschiedenen Modellen wiederzuverwenden und durch die verschiedenen Kombinationsmöglichkeiten flexibel anpassbare Modelle zu erhalten.

Die Komponenten müssen dafür möglichst lokal abgekapselt werden und kompatibel zu anderen Komponenten sein. Eine Komponente ist lokal gekapselt, wenn die Komponente nach außen hin unabhängig vom gerade aktiven Modus immer gleich erscheint, sodass die Verbindungen zu anderen Komponenten bei einem Moduswechsel nicht geändert werden müssen. Dazu müssen die Schnittstellen der Modi und die Anzahl der Eingänge und Ausgänge identisch sein (impliziert identisches C_Δ). Erfüllen die Schnittstellen diese Bedingung nicht, so muss der Kontext der Komponente bei einem Moduswechsel angepasst werden. Die Moduswechsel sind damit nicht mehr lokal gekapselt, was die Vorteile der Kapselung verringert.

Auch die Initialisierung ist wichtig und sollte lokal gekapselt vorgenommen werden. Dazu sollten sich die Werte von Ausgangsvariablen der Komponente während eines Moduswechsels nicht ändern, da dies zu Änderungen in anderen Komponenten führen kann.

Können die genannten Regeln nicht beachtet werden, so sind die Moduswechsel nicht unbedingt lokal und es müssen weitere Transitionen aktiviert oder Startwerte gesetzt werden.

Detaillierungsgradänderung

Die Modi sollten sich in den Genauigkeiten oder der Laufzeit weit genug unterscheiden, damit durch einen Moduswechsel positive Effekte erzielt werden können. Der Mehraufwand zum Erstellen der Modelle sollte durch

die gesparte Laufzeit oder die erhöhte Genauigkeit aufgewogen werden. Der Aufwand zum Erstellen des Modells ist umso besser angelegt, je öfter Komponenten mit lokalen Moduswechseln wiederverwendet werden können.

Die Genauigkeit bei sinkender Laufzeit zu steigern ist beispielsweise bei steifen Modellen möglich, wenn zeitweise ein vereinfachtes Modell verwendet werden kann, welches schneller simuliert werden kann, aber trotzdem eine hohe Genauigkeit aufweist.

Verhaltensänderung

In der konventionellen Modellierung müssen verschiedene Phasen eines Systems entweder in ein Modell integriert oder es muss über die Phasen abstrahiert werden. Mit strukturvariablen Modellen können verschiedene Phasen eines Systems in Modi aufgeteilt werden, wodurch jeder Modus nur das Notwendige für die entsprechende Phase enthält. Die verschiedenen Phasen sind dadurch häufig detaillierter simulierbar, als es mit konventionellen Methoden möglich wäre.

Die Modi sollten sich auch hier so weit unterscheiden, dass der Modellierungsaufwand durch die Vorteile des Ansatzes kompensiert wird. Oft haben die Modi unterschiedliche Zustandsvariablen, welche ineinander umgerechnet werden müssen. Dies kann zu Mehraufwand bei der Auslegung der Transitionen und zu Ungenauigkeiten während der Initialwertberechnung führen. Somit sollte auch hier versucht werden, nicht zu viele Moduswechsel einzuleiten.

Hinzufügen/Entfernen von Modellkomponenten

Ändern von gleichartigen Komponenten: Gridänderung

Bei der Gridänderung wird eine reale Komponente in viele kleine Teilelemente zerlegt (Bsp. FEM-Modell), um damit kritische Phasen detailliert und unkritische möglichst einfach zu betrachten. Das Ziel ist, wieder einen Kompromiss zwischen Genauigkeit und Laufzeit zu erhalten.

Bei der Modellierung ist die entscheidende Frage, wann wie viele Elemente verwendet werden sollen. Die Auflösung der Modi sollte sich möglichst weit unterscheiden, um einen Mehrgewinn zu erzielen, jedoch nah genug aneinander liegen, um eine sinnvolle Initialisierung bei einem Moduswechsel zu ermöglichen.

Bei der Initialisierung sind Fehler zu erwarten, da entweder Daten aus dem ungenaueren Modus generiert oder Mittelwerte berechnet werden müssen, um das ungenauere Modell zu initialisieren. Der Wechselzeitpunkt sollte daher möglichst so gewählt werden, dass sich die einzelnen Komponenten möglichst wenig unterscheiden. Ein Sprung in die Vergangenheit kann hilfreich sein, falls ein solcher Punkt überschritten wurde.

Ändern von gleichartigen Komponenten: Eigenständige Komponenten

Gleichartige Komponenten können für Systeme verwendet werden, in denen Komponenten unter vorgegebenen Voraussetzungen entfernt werden oder hinzukommen (Bsp. Ampelkreuzung, an der Autos ankommen und sich wieder entfernen). Aus Effizienzgründen sollten dabei nur die notwendigen Komponenten während der Simulation berücksichtigt werden.

Dabei ist die Wahl der Grenze relevant. Sie gibt an, wann eine Komponente betrachtet wird und wann nicht. Die Initialisierung kann dabei von externen Daten oder von bereits simulierten Daten abhängen. Auch ist zu beachten, dass sich Werte von bereits betrachteten Komponenten bei einem Moduswechsel ändern können.

Komponenten hinzufügen/entfernen

In einem Modell sind nicht immer alle Komponenten während der gesamten Laufzeit notwendig, unnötige Komponenten können zur Beschleunigung der Simulation entfernt werden. Sobald sie für die Genauigkeit notwendig sind, sollten sie wieder hinzugefügt werden.

Werden Komponenten hinzugefügt oder entfernt, so müssen deren Verbindungsbeschreibungen und gegebenenfalls die Schnittstellen der sich nicht ändernden Komponenten angepasst werden. Die Moduswechsel sind daher nicht lokal.

Soll der Moduswechsel lokal sein, so können *Dummy*-Komponenten vorgesehen werden, welche nur die notwendigen Schnittstellen enthalten und die entfernten Komponenten repräsentieren, wenn sie nicht mehr enthalten sein sollen.

Kommen neue Komponenten hinzu, so müssen diese anhand bekannter Daten initialisiert werden. Dazu können externe Daten oder Daten aus vorherigen Simulationen der Komponente verwendet werden. Dabei ist auf die Konsistenz zu den anderen Komponenten zu achten, da deren Verhalten nicht unbedingt mit den gewünschten Initialwerten kompatibel ist.

9.2. Formalisierung und Notation strukturvariabler Modelle

Die vorgestellte Formalisierung stellt einen formalen Rahmen zum werkzeugunabhängigen Beschreiben strukturvariabler Modelle zur Verfügung. Sie stellt alle notwendigen Eigenschaften strukturvariabler Modelle und deren Zusammenhänge kompakt und übersichtlich dar. Ebenso werden die Initialisierung, die Moduswechsel und das Ende der Simulation mit deren dynamischen Eigenschaften beschrieben. Die Formalisierung dient somit als Basis für Implementierungen und Sprachen, welche ebenfalls strukturvariable Modelle umsetzen sollen.

Die Formalisierung der Komponenten enthält bereits Informationen zu deren Simulation, die noch nicht explizit in Modellierungssprachen enthalten sind. Diese geben Hinweise darüber, unter welchen Bedingungen eine Komponente für eine Simulation gültig ist.

In der Formalisierung wurde bereits eine Art *Experimental Frame* vorgesehen, in dem Informationen über Startwertkombinationen *start* und Solver für Komponenten angegeben werden können. Diese geben Hinweise darüber, unter welchen Bedingungen die jeweilige Komponente für eine Simulation verwendet werden darf. In heutigen Standardwerkzeugen sind diese Angaben so jedoch noch nicht vorgesehen.

Diese Simulationsinformationen sollten noch um Gültigkeiten und Genauigkeiten ergänzt werden. Jedoch sind dazu noch weitere Untersuchungen notwendig. Aus diesem Grund sind diese Informationen noch nicht in der Formalisierung enthalten, könnten aber problemlos ergänzt werden.

Die Notation ist an die der Statecharts angelehnt und stellt eine graphische Beschreibungsweise der Formalisierung dar. Die Notation ist abstrakt, da sie an keine bestimmte Sprache gekoppelt ist.

Die Formalisierung kann um weitere Eigenschaften ergänzt und auf spezielle Sprachen angepasst werden, um so den formalen Rahmen für neue Werkzeuge bereitzustellen. Es kann damit sichergestellt werden, dass die Ausfaktorisierungen, Initialisierungen, Schnittstellen etc. korrekt behandelt werden.

9.3. Simulation mit DySMo

DySMo stellt einen Prototyp dar, der auf der Formalisierung basiert, mit dem strukturvariable Modelle aufgebaut und simuliert werden können. Zur Simulation einzelner Modi werden Standardwerkzeuge verwendet, somit können bereits existierender Modelle als Modi wiederverwendet werden. Ebenso können die Stärken der Compiler und Solver von den vorhandenen Werkzeugen genutzt werden. Durch den objektorientierten Aufbau des Frameworks können weitere Simulationswerkzeuge integriert werden. Die Steuerung der Moduswechsel findet im Framework statt.

Die Kommunikation zu den Simulationswerkzeugen nimmt Zeit in Anspruch. Auch sind die Solver der Werkzeuge nicht dafür ausgelegt, Strukturwechsel zu vollziehen. So muss jeder Modus auch bei kleinen Änderungen neu kompiliert werden, selbst wenn keine komplette erneute Kausalisierung notwendig wäre. Unter Umständen müssen die Änderungen in alle Modi manuell übertragen werden, falls die Modi nicht bereits für solche Fälle ausgelegt sind. Daher werden die Simulationszeiten für Modelle mit vielen Modi und vielen Moduswechseln mit DySMo länger sein als mit spezialisierten Werkzeugen.

Das Framework besitzt keine graphischen Notationsmöglichkeiten und kann Strukturänderungen nur auf oberster Modellebene durchführen. Der Modellierer muss die Modelle daher selbst ausfaktorisieren.

DySMo stellt also ein Testframework dar, um mit vorhandenen Mitteln in verschiedenen Werkzeugen strukturvariable Modelle zu simulieren. Es dient daher dazu, mehr über strukturvariable Modelle und deren Vorteile zu lernen.

10 Beitrag und Ausblick

Just because you can doesn't mean you should.

(Sprichwort)

10.1. Beitrag

Im Bereich der Ingenieurwissenschaften gewinnen strukturvariable Modelle immer mehr an Bedeutung. Bisher gab es jedoch weder eine ganzheitliche Untersuchung zum vorteilhaften Einsatz solcher Modelle noch (Modellierungs-)Konzepte zum Entwickeln dieser Modelle. Ebenfalls existierten für deren Simulation bisher nur Prototypen, die eine erneute Implementierung von Modellen in der jeweiligen Sprache voraussetzen.

Im Folgenden wird erläutert, welche Beiträge die vorliegende Arbeit leistet. Dazu werden die am Anfang der Arbeit beschriebenen Teilziele aufgegriffen.

1. **Klassifizierung anhand der Anwendungsgebiete:** Es wurde gezeigt, für welche Anwendungsgebiete strukturvariable Modelle geeignet sind. Dabei wurde für jedes Anwendungsgebiet erklärt, wann strukturvariable Modelle positiv im Vergleich zu klassischen Ansätzen sind und was bei deren Modellierung zu berücksichtigen ist.

   Es werden damit erste Richtlinien genannt, anhand derer ein Modellierer feststellen kann, ob für ein aktuelles Modellierungsvorhaben ein strukturvariables Modell vorteilhaft ist.

2. **Aufstellung der Modelleigenschaften:** Die während der Modellierung zu treffenden Entscheidungen bei der Entwicklung strukturvariabler Modelle wurden detailliert beschrieben. Dabei wurde gezeigt, wie strukturvariable Modelle aufgebaut sein sollten und welche Eigenschaften zu deren Beschreibung erforderlich sind. Der hierarchische und modulare Aufbau stand dabei im Vordergrund und es wurde gezeigt, wie wiederverwendbare, konsistente und simulierbare Modelle ausgelegt sein müssen.

Es werden damit die notwendigen Eigenschaften und Konzepte zum Modellieren von konsistenten, wiederverwendbaren strukturvariablen Modellen zusammengestellt.

3. **Formalisierung der Eigenschaften und Simulation strukturvariabler Modelle:**
   - **Formalisierung der Modelleigenschaften und der Simulation:** Die bereits textuell beschriebenen Eigenschaften strukturvariabler Modelle wurden formal beschrieben. Die notwendigen Konsistenzbedingungen, die für die hierarchische und modulare Modellierung notwendig sind, wurden in Form von Invarianten beschrieben. Ebenso wurden die dynamischen Vorgänge während der Simulation von strukturvariablen Modellen formalisiert.

     Es wird damit eine formale, werkzeugunabhängige Basis zum Beschreiben und Simulieren strukturvariabler Modelle bereitgestellt, welche als Grundlage für die Implementierung eines Simulationswerkzeugs dient.

   - **Graphische Notation:** Es wurde eine graphische Notation eingeführt, die eine konsistente Beschreibung strukturvariabler Modelle ermöglicht. Die Semantik der Notation wird durch die Formalisierung beschrieben.

     Es wird damit eine einfach verständliche Möglichkeit zum Beschreiben strukturvariabler Modelle mit allen notwendigen Eigenschaften bereitgestellt.

   - **Anforderungen an Simulationswerkzeuge:** Anhand der Formalisierung und der graphischen Notation wurden die Anforderungen zusammengestellt, die ein Simulationswerkzeug erfüllen muss, um strukturvariable Modelle zu handhaben.

     Es wird damit eine Zusammenfassung über die Mindestanforderungen an ein Simulationswerkzeug für strukturvariable Modelle angegeben.

4. **Werkzeug zur Modellierung und Simulation:** Ein prototypisches Framework, das auf der Formalisierung basiert, wurde implementiert. In diesem werden die Modi in konventionellen Werkzeugen simuliert, während die Strukturwechsel im Framework vorgenommen werden. Das Framework wurde in Python umgesetzt und unterstützt drei Simulationswerkzeuge: Dymola, OpenModelica und Simulink.

Es wird damit eine Möglichkeit geschaffen, strukturvariable Modelle aus vorhandenen Modellen aufzubauen und die Modi in Standardwerkzeugen zu simulieren.

5. **Fallstudie zu strukturvariablen Modellen:** Die Fallbeispiele zeigen unterschiedliche Anwendungsgebiete und Aspekte der strukturvariablen Modellierung. Dabei wird gezeigt, wie mit den in der vorliegenden Arbeit vorgestellten Modellierungskonzepten und dem Framework strukturvariable Modelle erstellt und simuliert werden können.

   Es wurde damit gezeigt, wie die Modellierung und Simulation von der Verwendung der vorgestellten Konzepte für strukturvariable Modelle profitieren kann.

Fazit

Mit der vorgestellten werkzeugunabhängigen Betrachtung und dem erlangten Wissen über strukturvariable Modelle ist ein Rahmen für den zukünftigen Umgang mit diesen Modellen geschaffen. Es wurde gezeigt, wie Modelle aufgebaut sein müssen, welche Eigenschaften sie haben und was bei deren Modellierung zu beachten ist, sodass diese in Zukunft vorteilhaft eingesetzt werden können. Die entwickelten Konzepte zum hierarchischen und modularen Aufbau von strukturvariablen Modellen führen neues und bekanntes Wissen in einem einheitlichen Formalismus zusammen und tragen somit signifikant dazu bei, strukturvariable Modelle zukünftig besser zu verstehen.

Es wurde erläutert, wann strukturvariable Modelle zu besseren Simulationsergebnissen führen und wie mit ihnen Systemverhalten simuliert werden kann, das konventionell schwer, gar nicht oder nur mit viel Zeitaufwand simulierbar ist.

Ein strukturvariables Modell sollte, wie auch ein konventionelles Modell, für den jeweiligen Untersuchungszweck ausgelegt sein. Die vorliegende Arbeit gibt einen Überblick über vorteilhafte Untersuchungszwecke strukturvariabler Modelle. Jedoch werden ebenso die Einschränkungen strukturvariabler Modelle erläutert, damit ein Modellierer entscheiden kann, ob für das aktuelle Vorhaben ein strukturvariables Modell geeignet ist. Strukturvariable Modelle sind nicht immer vorteilhaft und es sollte für jeden Untersuchungszweck individuell entschieden werden, ob solch ein Modell eingesetzt werden sollte: „Just because you can doesn't mean you should."

10.2. Ausblick

Zum Umgang mit strukturvariablen Modellen gibt es viele weitere Themen, die untersucht werden sollten. Die vorliegende Arbeit betrachtet die Basis zum Umgang mit strukturvariablen Modellen, um deren Eigenschaften sowie Vor- und Nachteile zu verstehen. Je mehr strukturvariable Modelle in den Fokus der Modellierung gelangen, desto mehr Untersuchungen sind notwendig, um den Modellierer bei der Erstellung der Modelle zu unterstützen und die Simulationen zu verbessern.

Im Folgenden wird ein Überblick über mögliche Folgethemen gegeben:

Erweiterung um Gültigkeiten und Genauigkeiten
Für die zukünftige Modellierung sollten Informationen zu Genauigkeiten und Gültigkeiten in die Modelle aufgenommen werden, damit Inkonsistenzen schneller erkannt werden und die Basis für automatische Moduswechsel gelegt wird. Wie bereits bekannt, ist es nicht trivial, automatische Modellwechsel zu vollziehen. Jedoch ist es durchaus möglich, Werte von Variablen innerhalb einer Simulation automatisiert konstant zu setzen oder vereinfacht zu berechnen. Um jedoch festzulegen, wann eine Komponente vereinfacht werden kann, sollten Gültigkeiten und Genauigkeiten beachtet werden.

Automatisiertes Ausfaktorisieren
Die Modellierung mit Strukturänderungen erfordert bisher noch manuellen Aufwand, da die Modelle ausfaktorisiert werden müssen. Dies sollte automatisiert werden.

Dafür sollte eine Notation entworfen oder die in dieser Arbeit vorgestellte erweitert werden, um die Modi und Transitionen in der gewünschten Modellierungssprache abzubilden. Die Formalisierung sollte an diese Sprache angepasst werden, sodass beispielsweise die Verbindungsbeschreibung zu der Modellierungssprache passt. Ist eine solche Notation entworfen, so kann das Ausfaktorisieren der Modi analog zur Formalisierung automatisiert werden.

Spezialisiertes Simulationswerkzeug
Ist eine Notation und Formalisierung für eine spezifische Sprache vorhanden, so müssen die Modelle simuliert werden. Das vorgestellte DySMo-Framework ist gut für die Simulation und Untersuchung strukturvariabler Modelle geeignet und könnte als Backend für ein Modellierungswerkzeug

verwendet werden. Durch die langen Kommunikationszeiten mit den Simulationswerkzeugen und dem Kompilieren aller Modi hat es eher experimentellen Charakter und wurde noch nicht auf Performanz hin optimiert. Je mehr mit strukturvariablen Modellen gearbeitet wird, desto besser sollte auch das Simulationswerkzeug für diese Modelle ausgelegt sein, um bessere Performanz bei der Simulation zu erreichen.

Compiler
Um die Modelle in ausführbaren Code umzuwandeln, sind Interpreter oder Compiler notwendig. Besonders in Bezug auf Modi innerhalb von Komponenten sollten Compiler verwendet werden, welche den Aufwand beim Erstellen der verschiedenen Modi reduzieren. Ein möglicher Ansatz dafür ist die *separate compilation*, in der Komponenten unabhängig von ihrem Kontext in verschiedenen Kausalisierungen kompiliert werden. Ein Modell kann dann aus Komponenten zusammengesetzt werden und die bereits kompilierten Modelle wiederverwenden. Ein komplettes Kompilieren des Modells ist dann nicht mehr notwendig. Ebenso ist die *just in time compilation* von Vorteil, in der erst kompiliert wird, wenn dies notwendig wird.

Skriptsprache zur Simulationsbeschreibung
Ein Vorteil des jetzigen DySMo-Frameworks sind die Schnittstellen zu den verschiedenen Werkzeugen. Würde eine gut ausgelegten DSL (**D**omain **S**pecific **L**anguage) für Modellbeschreibungen entworfen werden, so könnte diese zum Erstellen von Skripten für Simulationsbeschreibungen verwendet werden. Da es zurzeit noch keine einheitliche Skriptsprache für Simulationen gibt, eine solche jedoch für Parametervariationen nützlich wäre, könnte das Framework dahingehend erweitert werden.

Model pruning
Das automatische Vereinfachen von komplexen Modellen ist unter dem Begriff *Model pruning* bekannt. Mit Hilfe dieses Verfahrens können automatisiert vereinfachte Versionen von komplexen Modellen erstellt werden. Diese vereinfachten Modelle könnten dann beispielsweise als Modi in strukturvariablen Modellen verwendet werden. Das Erstellen von Modellen, die ihren Detaillierungsgrad ändern, wäre damit vereinfacht.

IV

Anhang

A

Tabellen

A.1. Definitionen für strukturvariable Modelle aus der Literatur

Tabelle A.1.: Auszug aus Papieren mit Definitionen für strukturvariable Modelle.

[12] (1979)	Variable-structure simulation	Eine Simulation, in der sich die Anzahl der Differential- oder Differenzen-Gleichungen mit der Zeit ändern
[63] (1987)	Multimodel	Ein Modell, welches aus mehreren Submodellen besteht: Dabei ist immer nur eins der Submodelle aktiv. Ein neues Submodell kann unter vordefinierten Bedingungen aktiviert werden. Es findet dann ein Submodellwechel statt.
[27] (1988), [20] (2006)	variable-structure system/-control, sliging mode	Reglerentwurfsmethode, mit der ein robuster Regler entworfen werden kann, welcher sich während der Regelung dem Zustand des zu regelnden Systems anpasst
[5] (1995)	Hybrid systems, continuous-variable control	Modellwechsel werden hier in Bezug auf die Regelungstechnik untersucht. Dabei geht es hier um ähnliche Prinzipien wie in [27]. Hier sollen Reglermodelle den Gegebenheiten angepasst werden. Auch hier werden die Modelle als Untergruppe von hybriden Systemen angesehen.
[19] (1996)	Mode switching models/systems	Untergruppe von hybriden Modellen, da sie kontinuierliche und diskontinuierliche Anteile aufweisen: Modelle sind kontinuierlich, bis ein Ereignis eintritt.

[56] (1997), [57] (1999)	Hybrid system, hybrid modeling, variable structure system	Modelle, die kontinuierliche und diskrete Anteile haben: Modelle können unterschiedliche Zustandsvektoren haben, zwischen denen gewechselt werden kann, um den Detaillierungsgrad anzupassen.
[23] (2000)	Modelle variabler Modellierungstiefe	Mehrere Teilmodelle, zwischen denen gewechselt werden kann, als Teilbereich der hybriden Modellierung: Dabei geht es hier um das Anpassen des Detaillierungsgrades während der Laufzeit.
[60] (2003)	Hybride Systeme, Strukturdynamik	Strukturdynamik ist ein Teil der hybriden Systeme. Hybride Systeme teilen sich in vier Aspekte auf: statische Struktur, kontinuierliches Verhalten, diskretes Verhalten, Strukturdynamik. Modelle können ihr Gleichungssystem während der Laufzeit anpassen.
[88] (2003)	variable-structure model, hybrid dynamic model of variable structure	In den verschiedenen Modi kann die Anzahl der Zustands- und algebraischen Variablen unterschiedlich sein. Zwischen den Modi kann gewechselt werden.
[7, 38, 94] (ab 2007)	structural dynamical systems	Zwischen Modellen mit unterschiedlich vielen Gleichungen und Variablen kann gewechselt werden.
[33] (2009)	structurally dynamic systems, structural dynamism	Modelle sind eine allgemeinere Form von hybriden Systemen. Das Modell kann aus Modi bestehen, zwischen denen gewechselt wird.

A.2. Anforderungen an das DySMo-Framework

Die Anforderungen an das Framework sind in Tabelle A.2 festgehalten.

Tabelle A.2.: Anforderungstabelle

Req1	**Allgemeine Anforderungen**
Req1.1	Verwendete Programmiersprache soll frei verfügbar sein.
Req1.2	Framework soll mehrere gängige Simulationswerkzeuge integrieren.
Req1.3	Framework soll leicht auf andere Simulationswerkzeuge erweiterbar sein.
Req1.4	Modellierer soll nichts am Ablauf im Framework ändern müssen, um neues Simulationswerkzeug zu integrieren.
Req1.5	Falls Kompilieren notwendig ist, sollte dies so selten wie möglich geschehen.
Req2	**Verwendung des Frameworks**
Req2.1	Modell soll skriptbasiert beschrieben werden.
Req2.2	Programmierkenntnisse sollen für grundlegende Funktionalitäten nicht notwendig sein.
Req2.3	Fortgeschrittene Funktionalitäten dürfen Programmierkenntnisse erfordern.
Req3	**Modellierung des strukturvariablen Modells**
Req3.1	Alle notwendigen Informationen zur Spezifikation des Modells können angegeben werden (siehe Abschnitt 6.1).
Req3.2	Beliebig viele Modi können beschrieben werden.
Req3.3	Modi sind eindeutig einem Simulationswerkzeug zugewiesen.
Req3.4	Beliebig viele Transitionen können beschrieben werden.
Req3.5	Modi und Transitionen müssen eindeutig identifizierbar sein.
Req3.6	Folgemodus wird in Transition angegeben.
Req3.7	Mapping-Initialisierung in Transition vorsehen.
Req3.8	Funktionen zum Initialisieren vorsehen.
Req3.9	In Funktionen dürfen Methoden des Frameworks und Python-Methoden verwendet werden (bspw. Daten auslesen, erneutes Kompilieren, etc.).
Req3.10	Alte Simulationsdaten sind für die Initialisierung verwendbar.

Req3.11	Schalten in die Vergangenheit soll möglich sein (in Funktion).
Req3.12	Globaler Solver kann spezifiziert werden.
Req3.13	Jeder Modus kann eigenen Solver haben.
Req4	**Speicherung von Simulationsdaten**
Req4.1	Beobachter mit Synonym-Beschreibung für jeden Modus muss angegeben werden können.
Req4.2	Nicht für jeden Modus muss Synonym angegeben werden.
Req4.3	Daten jeder Simulation sollen unter eindeutigem Namen gespeichert werden.
Req4.4	Daten sollen in einem gut weiterverwendbaren Format gespeichert werden.

B

Modelica Einführung

Modelica ist ein offener Standard und wird von der Modelica Association weiterentwickelt und gepflegt [82]. Da die vorliegende Arbeit einige Modelica Beispiele zeigt, werden grundlegende Konzepte der Sprache erläutert. Da es in dieser Arbeit nicht alleine um Modelica geht, wird nur ein grober Überblick über die wichtigsten Konzepte gegeben. Für nähere Informationen sei auf [32, 83] verwiesen.

Modelica ist eine deklarative Modellierungssprache zum Beschreiben von mathematischen Modellen, die auf differential-algebraischen Gleichungen basieren.

Reale Systeme bestehen in den meisten Fällen aus Komponenten, die miteinander in Relation stehen. Dabei wird versucht Komponenten so auszulegen, dass sie auch in anderen oder bauähnlichen Systemen wiederverwendet werden können.

Modelica greift dieses Konzept auf, indem Modelle ebenfalls aus Komponenten aufgebaut werden können. Dabei wird die Komponente als Klasse beschrieben (ähnlich einem Bauplan in der Realität) und kann in einem Modell mit gewünschten Parametern instanziiert werden (ähnlich zum Bauen der Komponente anhand des Bauplans mit konkreten Maßen, Material, etc.). Komponenten können weitere Komponenten enthalten und ebenso als Klassen beschrieben werden. Es können so Komponentengruppen vordefiniert werden, die wiederum in Komponenten einsetzbar sind. Modelica unterstützt somit das Kompositionsprinzip der objektorientierten Programmierung.

Die Komponenten können mit *connect*-Gleichungen verbunden werden. Modelica unterstützt somit die Möglichkeit, sehr nah an der Realität zu modellieren.

Der Zusammenschluss von Komponenten und die Beschreibung der Klassen ist einfach, da Modelica akausale Modellierung unterstützt. In einem Modelica-Modell werden nicht eine Reihe von Befehlen hintereinander abgearbeitet, sondern es werden alle Gleichungen aller Komponenten und

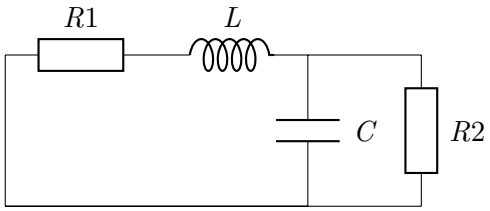


Abb. B.1.: Einfacher elektrischer Schaltkreis

deren Verbindungsgleichungen gesammelt, wodurch sich ein großes Gleichungssystem ergibt. Physikalische Gleichungen können somit auch als Gleichungen verwendet und müssen nicht zu Anweisungen umgeschrieben werden.

Modelica bietet eine große Bibliothek mit Standardkomponenten, welche für die eigenen Modelle verwendet werden können.

Sei ein einfaches Standardbeispiel eines elektrischen Schaltkreises betrachtet, welcher aus einem Widerstand, einem Kondensator und einer Spule besteht, siehe Abbildung B.1.

Die Gleichungen für den Schaltkreis ergeben sich aus den Gleichungen der Bauteile:

$$\begin{aligned} u_{R1} &= R1 \cdot i_{R1} \\ u_{R2} &= R2 \cdot i_{R2} \\ i_C &= C \cdot \dot{u}_C \\ u_L &= L \cdot \dot{i}_L \end{aligned}$$

und den Gleichungen der Maschen- und Knotenregel:

$$\begin{aligned} u_{R1} - u_L - u_C &= 0 \\ u_C - u_{R2} &= 0 \\ i_L - i_C - i_{R2} &= 0 \\ i_L &= i_{R1} \\ i_{R1} &= i. \end{aligned}$$

Wird das Modell in Simulink aufgebaut, so müssen die Gleichungen kausalisiert werden und können dann als Blockdiagramm beschrieben werden. Abbildung B.2 zeigt den obigen Schaltkreis als Simulink Modell. Kommen neue Elemente zum Schaltkreis hinzu, so muss in vielen Fällen das komplette Simulink-Modell erneuert werden.

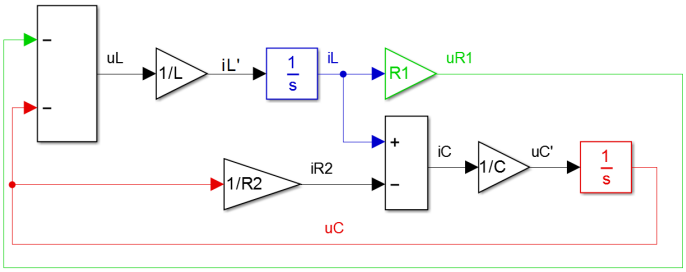


Abb. B.2.: Implementierung eines elektrischen Schwingkreises in Simulink

In Modelica kann der obige Schaltkreis direkt als Gleichungssystem angegeben werden, ohne die Gleichungen umzustellen, wie es in Listing B.1 gezeigt ist.

```
1  model elektro
2    Real uR1, uR2, uC, uL, iL, iR1, iR2, iC;
3    Real uR1, uR2, uL, iL, iR1, iR2, iC;
4    Real uC(start = 10);
5    parameter Real L = 5, C = 0.000005, R1 = 1, R2 = 10000;
6  equation
7    // Bauteile
8    uR1 = R1 * iR1;
9    uR2 = R2 * iR2;
10   iC = C * der(uC);
11   uL = L * der(iL);
12
13   // Maschen und Knotenregel
14   uR1 + uC + uL = 0;
15   uC - uR2 = 0;
16   iL - iR2 - iC = 0;
17   iL = iR1;
18 end elektro;
```

Listing B.1: Modelica-Modell für einen elektrischen Schwingkreis

Da der Schwingkreis keine Spannungsquelle enthält, müssen Initialwerte vorgesehen werden. Dazu bietet sich der Kondensator als speicherndes Element an, er wird mit einer Spannung von $10\,V$ initialisiert (Zeile 4).

In diesem Beispiel setzt sich das Modell noch nicht aus Komponenten zusammen, sondern beschreibt nur die obigen Gleichungen. Daher müssen beispielsweise zwei Gleichungen für die Widerstände vorgesehen werden und auch Änderungen am Schaltkreis müssen durch das Anpassen von Gleichungen umgesetzt werden.

Um die Vorteile von Modelica zu nutzen, sollten Modelle möglichst aus Komponenten aufgebaut werden, welche miteinander in Relation stehen. Listing B.2 zeigt eine Klasse eines Widerstands ähnlich zu dem aus der Standard-Bibliothek von Modelica.[1]

```
1 model Resistor
2  extends Modelica.Electrical.Analog.Interfaces.OnePort;
3  parameter Modelica.SIunits.Resistance R(start=1)
4 equation
5  v = R*i;
6 end Resistor;
```

Listing B.2: Modelica-Klasse eines Widerstands

Im Gleichungsteil (nach Zeile 4) ist die Bauteilgleichung zu erkennen (v steht in der Modelica-Bibliothek für die Spannung). Die Klasse erbt in Zeile 2 von der Klasse *OnePort*, welche zwei Ports (*Connector*) jeweils mit v, i und den Zustand der Spannung v und des Stroms i bereitstellt. Die entsprechende Klasse ist in Listing B.3 gezeigt. Erbt eine Komponente von einer Klasse K, enthält sie alle Eigenschaften dieser Klasse K. Im Widerstand kann daher direkt auf v und i zugegriffen werden.

```
1  partial model OnePort
2   SI.Voltage v;
3   SI.Current i;
4   PositivePin p;
5   NegativePin n;
6  equation
7   v = p.v - n.v;
8   0 = p.i + n.i;
9   i = p.i;
10 end OnePort;
```

Listing B.3: Modelica-Klasse für die Schnittstellen von elektrischen Bauteilen

Werden weitere elektrische Komponenten benötigt, so erben diese ebenso von *OnePort* und besitzen damit automatisch dessen Eigenschaften. Für die Spule und den Kondensator werden zwei weitere Klassen erzeugt, wobei nur deren bauteilspezifische Gleichung in den *equation*-Teil der Klasse geschrieben werden muss.

Um Ports (*Connector*s) von Bauteilen zu verbinden, können in Modelica *connect*-Gleichungen verwendet werden.

[1] Zum Zweck der Übersichtlichkeit werden die gezeigten Klassen auf das Wesentliche gekürzt.

Dazu sei der *Connector* des positiven Pins aus Listing B.4 betrachtet, welcher in der Klasse *OnePort* verwendet wird. Dieser enthält zwei Größen, einmal die Spannung und einmal den Strom. Beim Strom ist das Präfix *flow* angegeben, dieses gibt an, dass es sich um eine Flussvariable handelt und beim Verbinden mit anderen solchen Variablen die Summe Null ergeben muss.

```
1  connector PositivePin
2   Modelica.SIunits.Voltage v;
3   flow Modelica.SIunits.Current i;
4  end PositivePin;
```

Listing B.4: Modelica-Connector des positiven Pins

Zum besseren Verständnis sei das Modelica Modell des Schaltkreises, der nun aus den Komponenten zusammengesetzt wurde, aus Listing B.5 betrachtet.

```
1  model elektro2
2    Modelica.Electrical.Analog.Basic.Resistor R1(R=1);
3    Modelica.Electrical.Analog.Basic.Inductor L(L=5);
4    Modelica.Electrical.Analog.Basic.Capacitor
         C(C=0.000005, v(start=10));
5    Modelica.Electrical.Analog.Basic.Resistor R2(R=10000);
6    Modelica.Electrical.Analog.Basic.Ground ground;
7  equation
8    connect(R1.n, L.p) ;
9    connect(L.n, C.p);
10   connect(R2.p, C.p);
11   connect(R2.n, C.n);
12   connect(C.n, R1.p);
13   connect(ground.p, R1.p);
14 end elektro2;
```

Listing B.5: Modelica-Klasse für die Schnittstellen von elektrischen Bauteilen

Im Deklarationsteil der Klasse *elektro2* werden die Bauteile mit ihren spezifischen Eigenschaften instanziiert. Im *equation*-Teil werden jeweils zwei Schnittstellen miteinander verbunden. Für die Variablen, welche kein *flow*-Präfix haben, bedeutet dies, dass die Werte gleich sind. Sei der Knoten zwischen den Komponenten L, C und $R2$ betrachtet, welcher in den Zeilen 9 und 10 definiert ist.

Aus diesen Zeilen ergeben sich die Gleichungen:

$$\begin{aligned} L.n.v &= C.p.v \\ R2.n.v &= C.p.v \\ L.n.i + C.p.i + R2.p.i &= 0. \end{aligned}$$

Die ersten beiden Gleichungen beschreiben die Maschenregel, während die dritte Gleichung der Knotenregel entspricht. In der letzten Gleichung wird eine Summe gebildet und die negativen Vorzeichen, welche vorher notwendig waren, tauchen nicht mehr auf. Die korrekten Vorzeichen ergeben sich durch die Werte der Variablen.

Wird dieser Schaltkreis um weitere Komponenten erweitert, so können diese einfach hinzugefügt werden. Modelica ermöglicht eine einfache Modellierung mit Komponenten und deren Beziehungen untereinander durch die objektorientierte Struktur.

C Dominosteinmodell in DySMo

In diesem Abschnitt wird vorgestellt, wie das Beispiel 4.3 der Dominosteine aus Abschnitt 4.3.1 in DySMo umgesetzt wurde.

Dieses Modell hat einen Modus, wobei zwei Transitionen aus diesem hinausführen. Die einzelnen Dominosteine sind in Modelica in einem Array enthalten, wobei sich die Anzahl der Elemente im Array abhängig von den aktuell fallenden Steinen ändert.

Das Modelica-Modell des Dominobeispiels ist in Listing C.1 gezeigt. Die beiden Transitionen repräsentieren die beiden bekannten Fälle, in denen ein neuer Stein angestoßen wird oder ein Stein um 90 Grad gefallen ist.

```
1  model stones
2    parameter Real D=0.011; // Abstand der Steine
3    parameter Integer active=1;// Aktive Steine
4    parameter Integer fallen=0; // Gefallene Steine
5    parameter Integer gesamt=5; // Alle Steine
6    parameter Integer rest=gesamt - fallen - active;
7    Dominostein stones[active]; // Array aktive Steine
8    Integer transitionId; // Transitions ID
9  algorithm
10   // Stein wird angestoßen
11   when stones[active].x > D and rest > 0 then
12     transitionId := 1;
13     terminate("crash");
14   end when;
15   // Stein ist gefallen
16   when stones[1].phi_deg > 90 and active > 1 then
17     transitionId := 2;
18     terminate("ground");
19   end when;
20 end stones;
```

Listing C.1: Modelica-Modell der Dominosteine

Listing C.2 zeigt die DySMo Config-Datei des Dominosteinmodells. Darin werden zunächst die zwei Funktionen definiert und danach der Modus und dessen Transitionen.

```
# Dominostein trifft anderen Stein
def crash(act, old):
  import math
  active =  int(old.get_endValue('active') +1)
  fallen= int(old.get_endValue('fallen'))s
  D_val = old.get_endValue('D')
  Z_val = old.get_endValue('stones[1].Z')

  phipush = math.asin(D_val/Z_val);
  KO = (1+math.cos(2*phipush))/2;
  KR = 1 -math.cos(phipush);

  o = []
  p = []
  for i in range(1,active):
    omega = old.get_endValue('stones['+str(i)+'].omega')
    phi = old.get_endValue('stones['+str(i)+'].phi')
    p.append(phi)
    if i == active-1:
      o.append(omega*KR)
    else:
      o.append(omega)
  act.set_parameters({'active':active,'fallen':fallen})
  act.compile()
  act.read_init()

  for i in range(1,active):
    act.set_initialValue('stones['+str(i)+'].omega',
        o[i-1]);
    act.set_initialValue('stones['+str(i)+'].phi',
        p[i-1]);
  act.set_initialValue('stones['+str(active)+'].omega',
      o[-1]*KO/KR);
  act.set_initialValue('stones['+str(active)+'].phi', 0);

# Dominostein gefallen
def fall(act, old):
  active =  int(old.get_endValue('active'))-1
  fallen =  int(old.get_endValue('fallen')+1)

  o = []
  p = []
  for i in range(2,active+2):
    omega = old.get_endValue('stones['+str(i)+'].omega')
    phi = old.get_endValue('stones['+str(i)+'].phi')
    o.append(omega)
    p.append(phi)
```

```
45    act.set_parameters({'active':active,'fallen':fallen})
46    act.compile()
47    act.read_init()
48
49    for i in range(1,active+1):
50      act.set_initialValue('stones['+str(i)+'].omega',
          o[i-1]);
51      act.set_initialValue('stones['+str(i)+'].phi',
          p[i-1]);
52
53
54  # MODEL BESCHREIBUNG
55  model.translate = True;
56  model.default_solver = Solver("dassl") # Solver
57  model.init = {'stones[1].omega': 0.1} # Init
58  model.stopTime = 5 # Stopzeit
59  model.startTime = 0 # Startzeit
60  model.observe = ['stones[1].phi','stones[1].omega']
61
62  mode = DymolaMode()
63  mode.files = ['dominospiel.mo'] # Modelica Dateien
64  mode.modeRef = "dominospiel.stones" # Modelica-Model
65  mode.synonym={'stones[1].phi':'stones[1].phi',
      'stones[1].omega':'stones[1].omega'}
66
67  # Stein getroffen
68  trans1 = Transition()
69  trans1.post = mode # Wechsel zu sich selbst
70  trans1.init_function = crash # Funktionsaufruf
71  trans1.mapping = {}; # Kein Mapping
72
73  # Stein gefallen
74  trans2 = Transition()
75  trans2.post = mode # Wechsel zu sich selbst
76  trans2.init_function = fall # Funktionsaufruf
77  trans2.mapping = {}; # Kein Mapping
78
79  mode.transitions = [trans1, trans2]
80
81  model.modes = [mode]
```

Listing C.2: Beschreibung des Dominobeispiels in DySMo

Die erste Funktion startet in Zeile 2 und wird in Zeile 70 der Transition zugewiesen. In der Funktion werden die Daten der Dominosteine ausgelesen und für alle außer dem anstoßenden die alten Werte übernommen. Für den anstoßenden Stein wird die Winkelgeschwindigkeit um einen vorher berechneten Faktor *KR* verringert.

In Zeile 24 wird der Modus erneut kompiliert, wobei die Anzahl der aktiven und gefallenen Steine aktualisiert wird. Für dieses neue Modell werden die Initialdaten eingelesen und entsprechend der berechneten Startwerte gesetzt. Der neue Stein wird mit einem Winkel von 0 Grad und einer Winkelgeschwindigkeit initialisiert. Die Winkelgeschwindigkeit ergibt sich aus der alten Geschwindigkeit des anstoßenden Steins und einem berechneten Faktor *KO*.

Die Funktion der zweiten Transition beginnt ab Zeile 34 und wird in Zeile 76 der zweiten Transition zugewiesen. In ihr wird ein Stein aus dem Modus entfernt. Zur Initialisierung müssen die Enddaten des vorherigen Modus ausgelesen und jeweils um einen Index nach vorne geschoben werden.

Im restlichen Teil der Vorlage werden das Modell, dessen Modi und die zwei Transitionen instanziiert. Dies geschieht analog zu dem bekannten Vorgehen aus Abschnitt 7.4.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Åkesson, J. et al. *Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problems.* Computers and Chemical Engineering, Vol. 34(11), pp. 1737–1749, 2010.

[2] Åström, K. J.; Elmqvist, H.; Mattson, S. E. *Evolution of Continuous-Time Modeling and Simulation.* In: Proceedings of the 12th European Simulation Multiconference on Simulation - Past, Present and Future, pp. 9–18, 1998.

[3] Blochwitz, T. et al. *The Functional Mockup Interface for Tool independent Exchange of Simulation Models.* In: Proceedings of the 8th International Modelica Conference, pp. 105–114, 2011.

[4] Blochwitz, T. et al. *Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models.* In: Proceedings of the 9th International Modelica Conference, pp. 173–184, 2012.

[5] Branicky, M. *Studies in Hybrid Systems: Modeling, Analysis, and Control.* Dissertation, Massachusetts Institute of Technology, 1995.

[6] Breitenecker, F. *Development of Simulation Software - from Simple ODE Modelling to Structural dynamic systems.* In: Proceedings of the 22nd European Conference on Modelling and Simulation (ECMS'08), pp. 20–37, 2008.

[7] Breitenecker, F.; Popper, N. *Structure of Simulation Systems for Structural-Dynamic Systems.* In: Proceedings of the First Asia International Conference on Modelling & Simulation, pp. 574–579, 2007.

[8] Brenan, K. E.; Campbell, S. L.; Petzold, L. R. *Numerical solution of initial-value problems in differential-algebraic equations.* Classics in applied mathematics. Society for Industrial and Applied Mathematics, 1996.

[9] Broman, D. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation.* Dissertation, Linköping University, 2010.

[10] Casella, F.; Donida, F.; Bachmann, B.; Aronsson, P. *Overdetermined Steady-State Initialization Problems in Object-Oriented Fluid System Models.* In: Proceedings of the 6th International Modelica Conference, pp. 311–317, 2008.

[11] Casella, F.; Sielemann, M.; Savoldelli, L. *Steady-state initialization of object-oriented thermo-fluid models for homotopy methods.* In: Proceedings of the 8th International Modelica Conference, pp. 86–96, 2011.

[12] Cellier, F. E. *Combined Continuous/discrete System Simulation by Use of Digital Computers: Techniques and Tools.* Dissertation, Eidgenössische Technische Hochschule ETH Zürich, 1979.

[13] Cellier, F. E. *Continuous System Modeling.* Springer Science & Business Media, 1991.

[14] Cellier, F. E. *The Complexity Crisis.* In: Proceedings of the 8th International Joint Conference on Software Technologies SIMULTECH 2013, pp. IS–5, 2013.

[15] Cellier, F. E.; Kofman, E. *Continuous System Simulation.* Springer Science & Business Media, 2006.

[16] Clune, M.; Mosterman, P.; Cassandras, C. *Discrete Event and Hybrid System Simulation with SimEvents.* In: Proceedings of the 8th International Workshop on Discrete Event Systems, pp. 386–387, 2006.

[17] Dahmann, J.; Morse, K. *High Level Architecture for simulation: An Update.* In: Proceeding of the 2nd International Workshop on Distributed Interactive Simulation and Real-Time Applications (DIS-RT '98), pp. 32–40, 1998.

[18] *Dassault Systems.* `www.dynasim.se`. Abgerufen: Januar 2015.

[19] Edström, K. *Simulation of Mode Switching Systems Using Switched Bond Graphs.* Dissertation, Linköping University, 1996.

[20] Edwards, C.; Fossas Colet, E.; Fridman, L. *Advances in Variable Structure and Sliding Mode Control.* Springer Berlin Heidelberg, 2006.

[21] Ehrich, A. *Modellierung und Simulation eines Automatikgetriebes mit Strukturdynamik.* Studienarbeit, Technische Universität Berlin, 2012.

[22] Elmqvist, H.; Cellier, F. E.; Otter, M. *Object-Oriented Modeling Of Hybrid Systems.* In: Proceedings of the European Simulation Symposium (ESS'93), Society of Computer Simulation, pp. 31–41, 1993.

[23] Ernst, T.; Klein-Robbenhaar, C.; Nordwig, A.; Schrag, T. *Modellierung und Simulation hybrider Systeme mit Smile.* Informatik, Forschung und Entwicklung, Vol. 15(1), pp. 33–50, 2000.

[24] Felippa, C. A.; Geers, T. L. *Partitioned analysis for coupled mechanical systems.* Engineering Computations, Vol. 5(2), pp. 123–133, 1988.

[25] Ferreira, J. A. *Hybrid statecharts to model continuous and discrete behavior in engineering systems.* In: Proceedings of the 4th International Conference on Applied Mathematics and Computer Science (AMCOS'05), pp. 15:1–15:6, 2005.

[26] Ferreira, J. A.; Estima de Oliveira, J. P. *Modelling hybrid systems using statecharts and Modelica.* In: Proceedings of the 7th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '99), Vol. 2, pp. 1063–1069, 1999.

[27] Filippov, A.; Arscott, F. *Differential Equations with Discontinuous Righthand Sides: Control Systems.* Mathematics and its Applications. Springer Science & Business Media, 1988.

[28] Fishwick, P. A.; Narayanan, N. H.; Sticklen, J.; Bonarini, A. *A Multimodel Approach to Reasoning and Simulation.* IEEE Transactions on Systems, Man, and Cybernetics, Vol. 24(10), pp. 1433–1449, 1994.

[29] Fishwick, P. A.; Zeigler, B. P. *A Multimodel Methodology for Qualitative Model Engineering.* ACM Transactions on Modeling and Computer Simulation (TOMACS'92), Vol. 2(1), pp. 52–81, 1992.

[30] Freymann, R. *Strukturdynamik: Ein anwendungsorientiertes Lehrbuch.* Springer Berlin Heidelberg, 2011.

[31] Fritzson, P. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley, 2004.

[32] Fritzson, P. *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica.* Wiley, 2011.

[33] Giorgidze, G.; Nilsson, H. *Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems.* In: Proceedings of the 7th International Modelica Conference, pp. 208–218, 2009.

[34] Gipser, M. *Systemdynamik und Simulation.* Vieweg+Teubner Verlag, 1999.

[35] Hamann, P. *Modellierung und Simulation von realen Planetengetrieben.* Masterarbeit, Technische Universität Berlin, 2003.

[36] Harel, D. *Statecharts: A Visual Formalism For Complex Systems.* Science of Computer Programming, Vol. 8(3), pp. 231–274, 1987.

[37] Harel, D.; Politi, M. *Modeling Reactive Systems with Statecharts: The Statemate Approach.* McGraw-Hill, 1998.

[38] Heinzl, B. et al. *BCP - A Benchmark for Teaching Structural Dynamical Systems.* In: Mathematical Modelling 7(1), pp. 896–901, 2012.

[39] Henzinger, T. A. *The theory of hybrid automata.* In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96), pp. 278–292, 1996.

[40] Höger, C. *Separate Compilation of Causalized Equations - Work in Progress.* In: Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT 2011), pp. 113–120, 2011.

[41] Jahnke, A.; Ziegler, F. *Der Honigmann-Prozess - Thermochemische Energiespeicherung für die Bereitstellung von Strom, Wärme oder Kälte.* Solarzeitalter - Politik, Kultur und Ökonomie Erneuerbarer Energien, Vol. 1, pp. 59–62, 2010.

[42] Jahnke, A.; Ziegler, F.; Karow, M. *Re-evaluation of the Honigmann-process: Thermo-chemical heat store for the supply of electricity and refrigeration.* In: Heat Powered Cycles Conference, 2009.

[43] Lee, E. A.; Tripakis, S. *Modal Models in Ptolemy.* In: Proceedings of 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT'10), pp. 11–21, 2010.

[44] Li, P.; Li, Y.; Seem, J. E. *Consistent initialization of system of differential-algebraic equations for dynamic simulation of centrifugal chillers.* Journal of Building Performance Simulation, Vol. 5(2), pp. 115–139, 2012.

[45] Maler, O.; Manna, Z.; Pnueli, A. *From Timed to Hybrid Systems.* In: REX workshop Real-Time: Theory in Practice, Lecture Notes in Computer Science 600, pp. 447–484, 1992.

[46] Manna, Z.; Pnueli, A. *Verifying Hybrid Systems.* In: Hybrid Systems, Lecture Notes in Computer Science, Vol. 736, pp. 4–35, 1993.

[47] MapleSoft. *MapleSim.* `http://www.maplesoft.com/products/maplesim/`. Abgerufen: Januar 2015.

[48] Mehlhase, A. *Varying the level of detail during simulation.* In: 21. Symposium Simulationstechnik: ASIM 2011; Grundlagen, Methoden und Anwendungen in Modellbildung und Simulation, 2011.

[49] Mehlhase, A. *A Python framework to create and simulate models with variable structure in common simulation environments.* Mathematical and Computer Modelling of Dynamical Systems, Vol. 20(6), pp. 566–583, 2013.

[50] Mehlhase, A. et al. *An example of beneficial use of variable-structure modeling to enhance an existing rocket model.* In: Proceedings of the 10th International Modelica Conference, pp. 707–713, 2014.

[51] Mehlhase, A.; Krüger, I.; Schmitz, G. *Variable Structure Modeling for Vehicle Refrigeration Applications.* In: Proceedings of the 9th International Modelica Conference, pp. 927–934, 2012.

[52] Messerschmid, E.; Fasoulas, S. *Raumfahrtsysteme: Eine Einführung mit Übungen und Lösungen.* Springer-Verlag, 2011.

[53] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.3.* `https://www.modelica.org/documents/ModelicaSpec33.pdf`. Abgerufen: Januar 2015.

[54] *MOSILAB.* `http://mosim.swt.tu-berlin.de/wiki/doku.php?id=projects:mosilab:home`. Abgerufen: Januar 2015.

[55] Mosterman, P. J. *An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages.* In: Proceedings of the Second International Workshop on Hybrid Systems: Computation and Control (HSCC '99), pp. 165–177, 1999.

[56] Mosterman, P. J.; Biswas, G. *Formal Specifications for Hybrid Dynamical Systems.* In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97), pp. 568–577, 1997.

[57] Mosterman, P. J.; Biswas, G. *A Java implementation of an environment for hybrid modelling and simulation of physical systems.* In: 1999 International Conference on Bond Graph Modeling and Simulation (ICBGM '99), pp. 157–162, 1999.

[58] Münz, E. *Identifikation und Diagnose hybrider dynamischer Systeme.* Dissertation, Institut für Regelungs- und Steuerungssysteme, Karlsruher Institut für Technologie, 2006.

[59] Nickel, U.; Niere, J.; Zündorf, A. *The FUJABA Eenvironment.* In: Proceedings of the 22nd International Conference on Software Engineering (ICSE '00), pp. 742–745, 2000.

[60] Nordwig, A. *Integration von Sichten für die objektorientierte Modellierung hybrider Systeme.* Dissertation, Fachgebiet Softwaretechnik, Technische Universität Berlin, 2003.

[61] Nytsch-Geusen, C. et al. *MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics.* In: Proceedings of the 4th International Modelica Conference, pp. 527–535, 2005.

[62] Open Source Modelica Consortium. *OpenModelica.* `https://www.openmodelica.org`. Abgerufen: Januar 2015.

[63] Ören, T. I. *Model update: A model specification formalism with a generalized view of discontinuity.* In: Proceedings of the Summer Computer Simulation Conference, pp. 689–694, 1987.

[64] Ören, T. I.; Zeigler, B. P. *Concepts for advanced simulation methodologies.* SIMULATION, Vol. 32(3), pp. 69–82, 1979.

[65] Pahl, G. *Grundlagen der Konstruktionstechnik.* In: Dubbel Taschenbuch für den Maschinenbau, pp. 433–465. Springer Berlin Heidelberg, 1995.

[66] Pawletta, T. et al. *DEVS-Based Modeling and Simulation in Scientific and Technical Computing Environments.* In: DEVS Integrative

M&S Symposium (DEVS'06) – Part of the 2006 Spring Simulation Multiconference (SpringSim'06), pp. 151–158, 2006.

[67] Pawletta, T.; Lampe, B.; Pawletta, S.; Drewelow, W. *A DEVS-Based Approach for Modeling and Simulation of Hybrid Variable Structure Systems.* In: Modelling, Analysis, and Design of Hybrid Systems, Lecture Notes in Control and Information Sciences, Vol. 279, pp. 107–129. Springer Berlin Heidelberg, 2002.

[68] Pepper, P.; Mehlhase, A.; Höger, C.; Scholz, L. *A Compositional Semantics for Modelica-style Variable-structure Modeling.* In: Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT'11), pp. 45–54, 2011.

[69] Platzer, A.; Quesel, J. D. *KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description).* In: Proceedings of the 4th international joint conference on Automated Reasoning (IJCAR '08), pp. 171–178, 2008.

[70] Ptolemaeus, C., Editor. *System Design, Modeling, and Simulation using Ptolemy II.* `Ptolemy.org`, 2014.

[71] Richter, P. *Physik zwischen Chaos und Ordnung.* In: Der Flügelschlag des Schmetterlings - ein neues Weltbild durch die Chaosforschung, pp. 25–49. Deutsche Verlags-Anstalt Stuttgart, 1993.

[72] Sasser, J. E. *History of Ordinary Differential Equations: The First Hundred Years.* In: Proceedings of the Midwest Mathematics History Society, 1992.

[73] Schichl, H. *Models and the History of Modeling.* In: Modeling Languages in Mathematical Optimization, Vol. 88, pp. 25–36. Springer US, 2004.

[74] Schwarz, P. *Simulation of systems with dynamically varying model structure.* Mathematics and Computers in Simulation, Vol. 79(4), pp. 850–863, 2008.

[75] *Scilab.* `http://www.scilab.org/products/scilab`. Abgerufen: Januar 2015.

[76] Sielemann, M.; Casella, F.; Otter, M. *Robustness of declarative modeling languages: Improvements via probability-one homotopy.* Simulation Modelling Practice and Theory, Vol. 38, pp. 38–57, 2013.

[77] *SimulationX.* `http://www.simulationx.com/de/`. Abgerufen: Januar 2015.

[78] Spivey, J. M. *The Z Notation: A Reference Manual.* Prentice-Hall, Inc., 1989.

[79] Stachowiak, H. *Allgemeine Modelltheorie.* Springer-Verlag, 1973.

[80] The MathWorks Inc. *MATLAB und Simscape 2013b.* Natick, Massachusetts, United States.

[81] The MathWorks Inc. *MATLAB und Simulink 2013b.* Natick, Massachusetts, United States.

[82] *The Modelica Association.* `https://www.modelica.org`. Abgerufen: Januar 2015.

[83] Tiller, M. *Modelica by Example.* `http://book.xogeny.com/`. Abgerufen: Januar 2015.

[84] Top, J. *Conceptual Modelling of Physical Systems.* Dissertation, University of Twente, 1993.

[85] Trčka, M.; Hensen, J.; Wetter, M. *Co-simulation for performance prediction of integrated building and HVAC systems - An analysis of solution characteristics using a two-body system.* Simulation Modelling Practice and Theory, Vol. 18(7), pp. 957–970, 2010.

[86] Uhrmacher, A. M. *Dynamic Structures in Modeling and Simulation: A Reflective Approach.* ACM Transactions on Modeling and Computer Simulation (TOMACS'01), Vol. 11(2), pp. 206–232, 2001.

[87] *UML.* `http://www.uml.org/`. Abgerufen: Januar 2015.

[88] Urquia, A.; Dormido, S. *Object-Oriented Description of Hybrid Dynamic Systems of Variable Structure.* Simulation, Vol. 79(9), pp. 485–493, 2003.

[89] Vieira, R.; Biscaia Jr, E. *Direct methods for consistent initialization of DAE systems.* Computers and Chemical Engineering, Vol. 25(9 - 10), pp. 1299–1311, 2001.

[90] Wolfram. *Wolfram System Modeler.* `http://www.wolfram.com/system-modeler`. Abgerufen: Januar 2015.

[91] Woodcock, J.; Davies, J. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Inc., 1996.

[92] Yilmaz, L.; Ören, T. I. *Dynamic Model Updating in Simulation with Multimodels: A Taxonomy and a Generic Agent-Based Architecture.* In: Proceedings of the Summer Computer Simulation Conference (SCSC'04), pp. 2–8, 2004.

[93] Ylikiiskilä, J.; Akesson, J.; Führer, C. *Improving Newton's method for initialization of Modelica models.* In: Proceedings of the 8th International Modelica Conference, pp. 97–104, 2011.

[94] Zauner, G.; Leitner, D.; Breitenecker, F. *Modeling Structural - Dynamics Systems in MODELICA/Dymola, MODELICA/Mosilab and AnyLogic.* In: Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT'07), pp. 99–110, 2007.

[95] Zeigler, B. P.; Praehofer, H.; Kim, T. G. *Theory of Modeling and Simulation, Second Edition.* Academic Press, 2000.

[96] Zimmer, D. *Equation-based modeling of variable-structure systems.* Dissertation, Eidgenössische Technische Hochschule ETH Zürich, 2010.