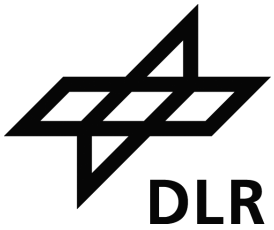
Diplomarbeit

Gekoppelte Ausführung von Simulink- und Modelica-Modellen mittels SMP2

Alexander Röhnsch

14. Dezember 2009



Prof. Dr. Joachim Fischer
Lehrstuhl Systemanalyse
Institut für Informatik
Mathematisch-Naturwissenschaftliche
Fakultät II
Humboldt-Universität zu Berlin

Dr. Andreas Gerndt
Deutsches Zentrum für Luft- und
Raumfahrt (DLR)
Simulations- und Softwaretechnik
(SISTEC)
Lilienthalplatz 7
38108 Braunschweig

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Ein Simulationsmodell lässt in einem einzigen Programm erstellen. Finden Modellierung und Berechnung vollständig in einem Programm statt, nennt man diese Simulation geschlossen. Der Ansatz bietet sich vor allem bei übersichtlichen Modellen an, die von wenigen Personen bearbeitet werden.

Übersteigt die Größe eines Simulationsmodells die Möglichkeiten eines Modellierers, müssen mehrere Modellierer zusammen daran arbeiten. Dazu wird das Modell aufgeteilt und modularisiert. Modellierer tauschen Teilmodelle untereinander aus. Dabei stoßen sie schnell auf Probleme. Benutzen sie unterschiedliche Versionen desselben Programms, wird ein Teilmodell unterschiedlich gespeichert. Der direkte Austausch wird kompliziert oder gar unmöglich. Sehr spezielle Aspekte eines Modells können nicht mit jedem Programm modelliert werden. Teilmodelle könnten dann mit Spezialwerkzeugen modelliert werden. Auch hierbei können Modelle nicht einfach ausgetauscht werden.

Eine Lösung ist die direkte Transformation eines Modells in die Modellrepräsentation eines anderen Programms. Die Entwickler müssen dann für jede Kombination von zwei Programmen, zwischen denen Modelle ausgetauscht werden sollen, eine oder sogar zwei Transformationen parat haben. Allerdings stoßen sie wieder auf Probleme. Spezialprogramme benutzen spezielle, an den jeweiligen Zweck angepasste Paradigmen zur Modellierung. Diese können nicht uneingeschränkt in andere Repräsentationen übersetzt werden. Übersteigt die Größe eines Simulationsprojekts die Möglichkeiten einer ganzen Einrichtung, muss die Entwicklung in Teilen sogar an externe Unternehmen abgegeben werden. Zwischen Unternehmen in der Industrie spielt der Schutz geistigen Eigentums eine Rolle. Der Austausch von Modellen wird dieser Anforderung nicht gerecht.

Eine andere Lösung ist der Austausch von Simulationscode, auch (Simulator-) Kopplung oder Co-Simulation genannt ([Dro04], [VGvdS+99] und [GKL06]). Dabei wird die Berechnung eines Teilmodells in Quellcode einer Hochsprache ausgedrückt. Der Quellcode eines Teilmodells kapselt seine vollständige Simulationsberechnung inklusive numerischer Lösungsmethoden; er formt ein eigenständiges Programm.

Die Teilmodelle sollen gemeinsam simuliert werden. Dazu müssen die aus ihnen entstandenen Teilmodell-Programme parallel ausgeführt werden. Sie müssen in ihrer Ausführung synchronisiert werden und untereinander Ergebnisse austauschen können.

Dieser Ansatz ermöglicht die Verwendung von Modellen beliebiger Modellierwerkzeuge in einer Simulation, sofern die Modellierwerkzeuge den Export der Simulationsberechnung als Quellcode unterstützen. Teilmodell-Programme können danach unabhängig von der Verfügbarkeit des Modellierwerkzeugs ausgeführt werden. Sie können als Programm auch leicht ausgetauscht werden. Geistiges Eigentum kann gewahrt werden, indem ein Teilmodell-Programm als Binärdatei weitergegeben wird.

Die Synchronisierung von Ausführung und Ergebnissen erfordert die Festlegung einer Schnittstelle. Diese regelt, wie Teilmodell-Programme miteinander interagieren und wie sie auszuführen sind. Der zur Synchronisation verwendete Mechanismus beeinflusst

die Ergebnisse der Gesamtsimulation. Genaue Untersuchungen der Ergebnisse für Co-Simulationen mit einer Vielzahl beteiligter Programme sind nicht bekannt. Mit der Synchronisierung führt die Co-Simulation jedenfalls einen erhöhten Simulationsaufwand ein. Programme müssen auf andere Programme warten. Werte müssen ausgetauscht werden. Werkzeugspezifische Modelloptimierungen können nur modell-lokal, nicht aber simulationsweit angewendet werden.

1.1. Problemstellung

Im DLR-Projekt *Virtueller Satellit* [SBMR08] wird untersucht, wie eine Bibliothek unterschiedlicher Teilmodelle erstellt werden kann. Sie soll Teilmodelle verschiedenster Disziplinen, Modelliersprachen, Entwickler und Institutionen beherbergen. Für eine Simulation sollen diese dann wie Komponenten eines physikalischen Systems einfach hergenommen und zu einem virtuellen Satelliten kombiniert werden können. Ingenieure können so zur Modellierung der Teilmodelle die Werkzeuge ihrer Wahl nutzen. Ein Teilmodell wird in der Bibliothek als Komponente in verschiedenen Repräsentationen abgelegt, auch als ausführbare Binärdatei. Zu einer Simulation kombinierte Komponenten werden mit Co-Simulation gemeinsam ausgeführt.

Zur Co-Simulation der Komponenten wird der Simulationsstandard *SMP2* (Simulation Model Portability 2.0) genutzt. SMP2 definiert die Rollen SMP2-Modell und SMP2-Simulator. Der SMP2-Simulator ist die Ausführungsschicht. Er führt SMP2-Modelle als parallele Prozesse aus. Die Komponenten der Bibliothek sollen als SMP2-Modelle ausgeführt werden. Dazu wird der Code mit ihren isolierten Simulationsberechnungen in ein SMP2-Modell eingebettet. Der SMP2-Simulator lädt das kompilierte SMP2-Modell dann dynamisch, führt es aus und lässt es mit anderen SMP2-Modellen kommunizieren. Dazu definiert SMP2 eigene Kommunikationsmechanismen.

Ein SMP2-Modell kann mit Mechanismen des Standards ausgeführt werden, indem zu wiederkehrenden, durch einen festen Zeitabstand definierten Ereignissen eine Methode des SMP2-Modells ausgeführt wird. Dieses Ausführungsschema wird für erste Untersuchungen der Co-Simulation im Projekt *Virtueller Satellit* benutzt. Es wird sich zeigen, wann komplexere Schemen notwendig werden.

Zur Modellierung von Teilmodellen werden das Modellier- und Simulationswerkzeug Simulink und die Modelliersprache Modelica unterstützt. *Simulink* ist zur Modellierung im Raumfahrtbereich das am weitesten verbreitete Werkzeug. Ingenieuren ist es aus Ausbildung und Praxis im Allgemeinen umfassend bekannt. Viele im DLR bereits erstellte Modelle liegen zudem bereits unter Simulink vor. Zur SMP2-Einbettung von Simulink-Modellen existiert zudem mit dem Programm MOSAIC bereits eine Lösung. MOSAIC ist momentan aber nur manuell zu bedienen. Es erzeugt SMP2-Modelle, sieht aber keine Kopplung vor. Zwischen den Modellen ist ohne weiteren Programmieraufwand keine Kommunikation möglich.

Modelica ist eine relativ junge Sprache zur Modellierung komplexer physikalischer Systeme verschiedener Domänen. Sie wurde unter maßgeblicher Beteiligung des DLR ent-

wickelt. Mit ihrer Unterstützung im Projekt Virtueller Satellit kann die Co-Simulation von Komponenten unterschiedlicher Implementierung demonstriert werden. Die SMP2-Einbettung von Modelica-Modellen ist bisher noch nicht umgesetzt, sondern lediglich ihre Möglichkeit untersucht worden [Rö08].

Im Virtuellen Satelliten sollen Komponenten wie von Simulink und Modelica bekannt über Signalkanäle miteinander kommunizieren. Dazu kann eine Komponente nach außen Ein- und Ausgänge definieren. Zwischen Komponenten können auf SMP2-Ebene Ein- und Ausgänge miteinander verknüpft werden, um sie so Zustände synchronisieren zu lassen.

1.2. Zielsetzung

Das Ziel dieser Diplomarbeit ist die Entwicklung eines Software-Prozesses, der ein Simulink- oder Modelica-Modell in ein SMP2-Modell einbettet. Die Einbettung soll automatisch ohne manuelle Eingriffe ablaufen. Dazu wird die allgemeine Funktionsweise einer SMP2-Einbettung untersucht. Darauf aufbauend wird eine Schnittstelle zur Kopplung von SMP2-eingebetteten Modellen festgelegt.

Für Simulink-Modelle soll die Ausführung der zur Einbettung zu verwendenden Programme Real-Time Workshop und MOSAIC automatisiert werden. Die durch Anwendung der Programme resultierenden SMP2-Modelle müssen dann gemäß der Kopplungs-Schnittstelle koppelbar gemacht werden, so dass Komponenten einfach miteinander kommunizieren können. Die Beschränkungen der Einbettung auf die Modellierung werden festgehalten.

Für Modelica-Modelle wird die Einbettung in SMP2-Modelle neu entwickelt. Erforderliche Schritte wurden in der vorangegangenen Studienarbeit untersucht [Rö08]. Die resultierenden SMP2-Modelle müssen ebenfalls gemäß der Schnittstelle koppelbar sein. Die Entwicklung der Einbettung von Modelica-Modellen dient in dieser Arbeit als Beispiel, wie Simulationscode generell zur Co-Simulation in ein SMP2-Modell eingebettet werden kann. Auch für Modelica-Modelle werden die durch die Einbettung eingeführten Einschränkungen festgehalten.

Mit den entwickelten Einbettungen werden gekoppelte Testmodelle simuliert. Dabei wird untersucht, wie der Einbettungsmechanismus und die Kopplung auf SMP2-Ebene die Simulationsergebnisse beeinflussen. Insbesondere ist der Einfluss der festen Schrittweite bei der synchronisierten Ausführung von Interesse; besonders auch der Einfluss auf das komplexe Verhalten eines dynamischen Systems. Zu Untersuchungen mit dieser Fragestellung ist keine Literatur bekannt.

1.3. Struktur der Arbeit

Die Hintergründe von SMP2 sind für das Verständnis der folgenden Arbeit besonders wichtig. Deshalb wird der Standard zuerst in seinen relevanten Teilen in Kapitel 2 vorgestellt.

Im Kapitel 3 werden Grundlagen der Einbettung von Simulationscode allgemein behandelt. Der generelle Aufbau von Simulationscode wird beleuchtet und daraus ein Ansatz für eine Einbettung in SMP2 entwickelt. Zu dem Ansatz wird ein allgemeiner Einbettungsprozess beschrieben. Die Folgekapitel bauen sowohl auf diesem Ansatz als auch auf dem allgemeinen Einbettungsprozess auf. Schließlich werden noch die Werkzeuge vorgestellt, mit denen die Entwicklung durchgeführt wurde.

Kapitel 4 und 5 stellen jeweils die Entwicklung der Einbettung von Simulink-Modellen bzw. von Modelica-Modellen dar. Benötigte Programme werden vorgestellt. Anschließend werden die Prinzipien erklärt, mit denen Modelle der jeweiligen Sprache in SMP2 eingebettet werden. Der Abschnitt Umsetzung erklärt dann, wie diese Prinzipien konkret implementiert wurden. Beide Kapitel schließen mit einer Beschreibung, wie erstellte SMP2-Modelle genutzt werden können und welchen Einschränkungen sie unterliegen.

Daraufhin werden mit den entwickelten Software-Prozessen Modelle in SMP2 simuliert und ihre Ergebnisse untersucht. Dazu werden in Kapitel 6 zwei Testmodelle vorgestellt, verschiedene Kopplungsszenarien angewendet und die Eigenschaften der Ergebnisse ausgewertet.

Kapitel 7 fasst die Entwicklung der SMP2-Einbettungen und die Auswertung der gekoppelten Simulationen zusammen. Das Kapitel schließt mit einem Ausblick. Darin werden Empfehlungen in Hinsicht auf Verbesserung der Einbettung und Verbesserung der Simulationsergebnisse gegeben.

2. SMP2

Simulation Model Portability 2.0 (SMP2) [ESO05c] ist ein unter Führung der Europäischen Raumfahrtbehörde (ESA) entwickelter Simulationsstandard. Er soll europäischen Raumfahrtprojekten und -unternehmen ermöglichen, Simulationsmodelle und -anwendungen auszutauschen und Entwicklungskosten zu reduzieren.

Im Kern steht die Idee, Simulationsmodelle unabhängig voneinander auszuführen und über plattformunabhängige Mechanismen miteinander kommunizieren zu lassen. Dazu definiert SMP2 eine Schnittstelle, die die Interaktion zwischen Modellen und einer ausführenden Instanz, dem so genannten *SMP2-Simulator* definiert. SMP2 definiert weiter Schnittstellen, die die Interaktion der Modelle untereinander beschreiben.

Diese Schnittstellen sind allgemein gehalten, um verwendbare Modelle so wenig wie möglich einzuschränken. Daher sind SMP2-Modelle aus Sicht des Simulators einfache Prozesse, die parallel ausgeführt werden. SMP2 könnte so auch für Prozesse eingesetzt werden, die keine Simulationsaufgaben realisieren. Lediglich der von SMP2 zur Verfügung gestellte Dienst *TimeKeeper* nimmt simulationsspezifische Aufgaben wahr. Er verwaltet verschiedene Zeitskalen, wie z.B. die Simulationszeit, die Systemzeit und eine Missionszeit, deren Verwaltung bei Simulationen häufig relevant ist.

Da in dieser Arbeit SMP2-Modelle erstellt werden sollen, ist im Weiteren auch nur die Modellschnittstelle von Interesse. Sie unterteilt sich in zwei Aspekte: wie die Struktur eines Modells definiert wird, und wie das Modell mit dem Simulator interagiert, um diese Struktur konsistent darzustellen. Diese beiden Aspekte werden im Folgenden genauer vorgestellt. Es schließt sich eine Beschreibung an, wie aus SMP2-Modellen Quelltext generiert werden kann. Schließlich wird der SMP2-Simulator SIMSAT vorgestellt.

2.1. Struktur von SMP2-Modellen

Die Modellstruktur wird mit Elementen des SMP2-Metamodells [ESO05d] definiert. Es definiert eine Vielzahl von Elementen wie Funktionen, Variablen, die Möglichkeit auf Ereignisse zu reagieren, andere Modelle zu beinhalten u.v.m. Die für diese Arbeit relevanten Konzepte sind *Entrypoints* und *Field-Variablen*.

Ein *Entrypoint* ist eine vom Modell definierte Funktion ohne Parameter und ohne Rückgabewerte. Entrypoints können dem Simulator bekannt gemacht werden, damit er sie aufruft. So lässt sich der Simulator anweisen, zu bestimmten Zeiten eine bestimmte Funktionalität des Modells auszulösen.

Fields heißen in SMP2 definierte Variablen. Verschiedene Typen wurden in SMP2 definiert. Verfügbar sind: Integer, Float, Array, String, DateTime, Duration und viele weitere. Um zwei Fields einfach zu synchronisieren, definiert SMP2 den Field-Link. Er wird in Abschnitt 3.4 näher erläutert.

Die Struktur eines SMP2-Modells wird in einem SMP2-Catalogue-Dokument abgelegt. Das ist eine XML-Datei, für die eine XML-Schema-Definition (XSD) vorliegt. Die XSD

definiert das formale Metamodell für Catalogues. Mit ihr können Catalogue-Dokumente validiert und geparst werden. Ein Beispiel findet sich im Anhang A.2. Das SMP2-Metamodell umfasst noch andere XML-Dokumente, die im Abschnitt 2.4 vorgestellt werden.

2.2. Modell-Simulator-Interaktion

Die Interaktion zwischen SMP2-Modellen und dem SMP2-Simulator ist im SMP2-Komponentenmodell [ESO05b] definiert. Das Komponentenmodell regelt dabei unter anderem, wie die Modelle ihre Struktur bekannt machen. Aus Sicht des Simulators durchläuft jedes Modell eine Folge an Modellphasen. Diese sind *Created*, *Publishing*, *Configured* und *Connected*. Durch Aufruf bestimmter Übergangsmethoden, kann der Simulator die Phase eines Modells wechseln und dem Modell so die Möglichkeit geben, die in dieser Phase erforderlichen Bekanntmachungen durchzuführen. Die Übergangsmethoden lauten `Publish`, `Configure`, und `Connect`.

Bei Erzeugung einer Modellinstanz wird zunächst ihr Konstruktor aufgerufen. Im Konstruktor können lokale Variablen instanziert und das Modell an sich initialisiert werden. Mit Beendigung des Konstruktors ist das Modell im Stadium *Created*.

Indem der Simulator die Modellfunktion `Publish` aufruft, erhält das Modell die Möglichkeit, seine Strukturelemente wie Entrypoints und Field-Variablen dem Simulator bekannt zu machen. Dies macht es wiederum über speziell definierte Simulatorfunktionen, z.B. `PublishField` oder `PublishOperation`. In dieser Phase kann das Modell auch dynamisch weitere Modelle erstellen und bekannt geben. Mit Rückgabe der Funktion `Publish` geht das Modell in die Phase *Publishing* über.

Indem der Simulator die Modellfunktion `Configure` aufruft, erhält das Modell die Möglichkeit, seine Variablen und internen Zustände zu konfigurieren. Dies kann auch über externe Komponenten erfolgen. So lädt z.B. der SMP2-Simulator Parameterwerte aus einem angegebenen Assembly-Dokument und überträgt sie in die Modellvariablen. Diese Technik wird auch in dieser Arbeit genutzt. Mit Rückkehr aus der Funktion ist das Modell in der Phase *Configured*.

Der Aufruf der Funktion `Connect` zeigt dem Modell an, dass es nun konfiguriert und mit dem Simulator verbunden ist. Es können keine weiteren Elemente bekannt gemacht oder Modelle erzeugt werden. Aber der Schritt eignet sich z.B. um letzte Initialisierungen nach dem Laden des Assemblys durchzuführen. Mit Rückkehr der Funktion ist das Modell in der Phase *Connected*. Diese Phase behält das Modell über die gesamte Simulation bei. Erst wenn bei Beendigung des Simulators die Modelle aufgelöst werden, wird diese Phase wieder verlassen.

2.3. Codegenerierung

Ein SMP2-Language-Mapping definiert, wie aus einem Catalogue-Dokument Quelltext generiert wird. Bisher definiert SMP2 das C++-Mapping [ESO05a]. Darin werden die

Elemente des SMP2-Metamodells, wie Variablentypen, Klassen, Vererbung, etc. auf Elemente der Sprache C++ abgebildet. So wird z.B. der SMP2-Typ Float64 auf den (architekturabhängigen) C++-Typ *double* abgebildet und SMP2-Interfaces auf C++-Klassen, deren Methoden alle als `public virtual abstract` deklariert sind. Als Beispiel bietet der Anhang A.2 eine C++-Header-Datei, die aus einem Catalogue generiert wurde.

Die Definitionen des Komponentenmodells werden ebenfalls berücksichtigt. Der Quelltext von SMP2-Modellen beinhaltet so gleich die Übergangsmethoden zum Wechseln der Modellphase mit einer Standardimplementierung. Dem Programmierer wird so der meiste Aufwand der Modell-Simulator-Interaktion abgenommen. Er kann sich auf die Implementierung der Entrypoints konzentrieren. Der generierte Quelltext eines SMP2-Modells ist sogar komplett in dem Sinn, dass er kompiliert und in einem SMP2-Simulator geladen werden kann.

2.4. Weitere SMP2-Dokumente

Über die Struktur von Modellen hinausgehend formuliert das SMP2-Metamodell noch weitere Aspekte einer SMP2-Simulation. In einem Assembly-Dokument werden SMP2-Modelle instanziert. Die Variablen und Parameter jeder Instanz können individuell belegt werden. Außerdem lassen sich im Assembly Field-Links definieren, also zwei Field miteinander synchronisieren. Im Assembly wird so auch angegebenen, welche Modellinstanz mit welcher anderen kommunizieren soll.

Im Schedule-Dokument kann schließlich eine eine zeitliche Taktung von Ereignissen angegeben werden. Ereignissen lassen sich Aktivitäten zuordnen. Aktivitäten sind die Ausführung einer Entrypoint-Funktion, die Aktivierung einer Field-Link-Synchronisierung oder andere, die aber hier nicht von Bedeutung sind.

2.5. SIMSAT

Das Programm *SIMSAT* ist die Referenzimplementierung eines SMP2-Simulators. SIMSAT wird unter Auftrag der ESA entwickelt. Die aktuelle Version 4 des Programms läuft ausschließlich unter Linux. Diese Version kommt auch in dieser Arbeit zum Einsatz. Damit ist die Ausführung von Simulationen gegenwärtig auf das Betriebssystem Linux festgelegt und alle ausführbaren Dateien müssen für diese Plattform kompiliert werden.

3. Einbettung von Simulationscode

Bevor die Einbettungsprozesse für Simulink- und Modelica-Modelle beleuchtet werden, sollen die zugrundeliegenden Konzepte erklärt werden. Wie kann, ganz allgemein, Simulationscode in ein SMP2-Modell eingebettet werden? Dazu wird zunächst festgehalten, wie Simulationscode zusammengesetzt ist und wie seine Teile zusammen arbeiten. Mit diesen Erkenntnissen kann die Ausführung der Berechnungen dem Simulationscode abgenommen und auf ein SMP2-Modell übertragen werden. Dazu wird eine konkrete Möglichkeit vorgeschlagen, einerseits die Berechnung mit SMP2-Mechanismen auszuführen, andererseits die SMP2-Modelle miteinander kommunizieren zu lassen. Abschließend wird der allgemeine Einbettungsprozess in seiner Gesamtheit dargestellt.

3.1. Aufbau von Simulationscode

Einige Simulationswerkzeuge generieren Simulationscode in einer Hochsprache, z.B. C/ C++. Der Code führt die zur Simulation des Modells benötigten Berechnungen aus. Er kann unabhängig von der Verfügbarkeit des Simulationswerkzeugs kompiliert und ausgeführt werden.

Für die Einbettung von Simulink- und Modelica-Modellen werden die Modellcompiler *Real-Time Workshop* und *OpenModelica Compiler* benutzt (und in den folgenden Kapiteln vorgestellt). Sie trennen erzeugten Simulationscode in Modellcode und Laufzeitbibliothek [RTW05][LF05][LP08]. Werden Laufzeitbibliothek und Modellcode kompiliert und miteinander verlinkt, entsteht eine ausführbare Datei, die in der Regel ohne weitere Dateiabhängigkeiten ausführbar ist und die Simulation so exakt wie in der Simulationsumgebung des Modells berechnet. Abbildung 1 zeigt ein Schema für diese Trennung aus der Dokumentation zum Real-Time Workshop.

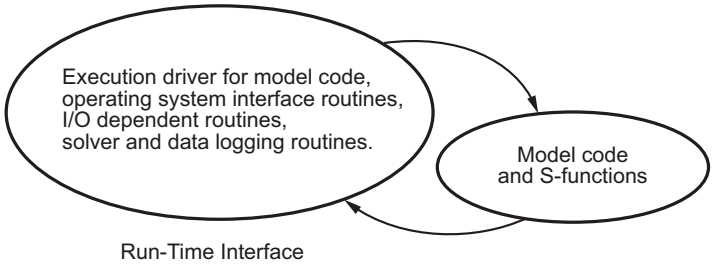


Abbildung 1: Trennung von Modellcode und Laufzeitbibliothek im Real-Time Workshop. Aus [RTW05].

Die Laufzeitbibliothek beinhaltet die Steuerung der Simulation, die verfügbaren Lösungsalgorithmen sowie allgemein benötigte Hilfsfunktionen. Ihr Code bleibt für alle übersetzten Modelle gleich. Zur Simulation ruft die Laufzeitbibliothek fest vereinbarte Schnittstellenfunktionen des Modellcodes auf, im Folgenden auch Modellfunktionen genannt. Als Beispiel für Simulink-Modellfunktionen fordert `model_Derivatives` aktu-

elle Ableitungen kontinuierlicher Zustände an, und `MdlUpdate` fordert die Aktualisierung diskreter Zustände. Abbildung 2 zeigt, wie die Real-Time-Workshop-Laufzeitbibliothek die Schnittstellenfunktionen des Modellcodes ausführt. Mit "Execution Loop" ist die Simulations-Hauptschleife angezeigt. Mit "Integration in Minor Time Steps" sind eventuell bei der Integrationsberechnung anfallende Teilschritte markiert.

Nur für das spezifische, im Modell beschriebene Verhalten wird der Modellcode erzeugt. Die Berechnung des Verhaltens wird in Form der Modellfunktionen bereitgestellt. Jedes Simulationswerkzeug definiert seine eigene spezifische Schnittstelle zwischen Modellcode und Laufzeitbibliothek. Zwar kann es Gemeinsamkeiten geben, dies ist aber nicht garantiert.

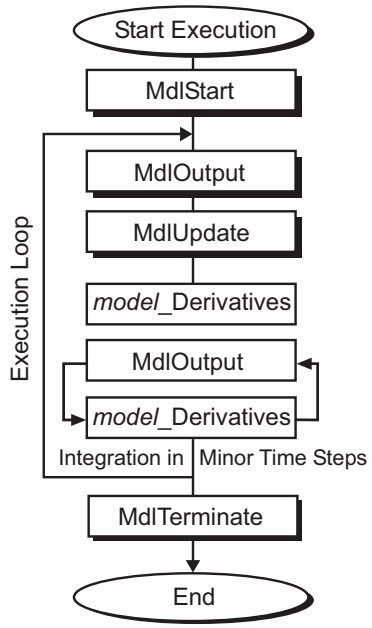


Abbildung 2: Schnittstellenfunktionen für Modellcode des Real-Time Workshops und wie sie von der Laufzeitbibliothek aufgerufen werden. Aus [RTW05].

3.2. Entnahme der Ausführung

Die Ausführung der Berechnung soll von der Laufzeitbibliothek auf das SMP2-Modell übertragen werden. Es lassen sich dazu aber nicht einfach die Modellfunktionen eines Simulationswerkzeugs nutzen. Die Schnittstelle zwischen Modellcode und Laufzeitbibliothek wird von jedem Werkzeug anders definiert. Jedes SMP2-Modell soll sich aber unabhängig von seinem Kern gleich verhalten.

Gemeinsamkeiten zwischen verschiedenen Schnittstellen lassen sich nur finden, wenn sie auf derselben Definition beruhen. Dies ist bei Real-Time Workshop und OpenModelica

nicht der Fall und deshalb unterscheiden sich beide. Zwar gibt es Modellfunktionen, die ähnliche Ziele verfolgen (z.B. Bestimmung von Nulldurchgängen einer Funktion), aber sie werden unter anderen Umständen aufgerufen, oder mit anderen Parametern. Es gibt keine festgelegt oder einvernehmlich identische Bedeutung bestimmter Funktionen, so dass sie zwischen verschiedenen Werkzeugen ausgetauscht werden könnten. Auch hängt die Definition der Modellfunktionen untrennbar mit der Ausführungslogik des Lösers in der Laufzeitbibliothek zusammen.

Es muss also ein allgemeines Prinzip jenseits der Modellfunktionen gefunden werden. Gemeinsam ist Real-Time Workshop und OpenModelica, dass sie schrittweise Berechnungen durchführen. Wird also die Ausführung jedes nächsten Simulationsschritts als gemeinsame Grundlage gesehen, kann Simulationscode beider Werkzeuge gleich ausgeführt werden, nämlich schrittweise. Es kann außerdem erwartet werden, dass weitere Werkzeuge ebenso behandelt werden können.

Alle Anweisungen, die zur Berechnung eines Simulationsschrittes gehören, werden in einer neuen Funktion übernommen. So wird die interne Abarbeitung der Modellfunktionen vollständig gekapselt. Mit dem Löser wird auch die Steuerung der Schrittweiten gekapselt. Da unterschiedliche Löser zum Einsatz kommen können, unterscheiden sich so gekapselte Komponenten immer noch in der Wahl ihrer Schrittweiten. In dieser Arbeit können die Komponenten jedoch auf eine gemeinsame, feste Schrittweite beschränkt werden. Zum einen fordern dies benutzte Programme wie MOSAIC, zum anderen ist dies eine Beschränkung von SMP2-Schedules. Die Löser müssen entsprechend konfiguriert werden.

3.3. Übertragung der Ausführung

Für die Ausführung des nächsten Simulationsschrittes wird ein SMP2-Entrypoint vereinbart. Ruft der SMP2-Simulator diesen Entrypoint auf, berechnet das Modell seine internen Zustände und seine Ausgabevariablen für den nächsten Zeitschritt.

SMP2-Entrypoints werden mit fester Schrittweite ausgeführt. Nur dieser Mechanismus wird direkt vom SMP2-Schedule unterstützt [ESO05d]. SMP2 bietet keine anderen Möglichkeiten, Einfluss auf die Simulationszeit des Simulators zu nehmen. Eine Möglichkeit zur Realisierung variabler Schrittweiten wird im Anhang A.3 gegeben. Feste Schrittweiten sind auch von MOSAIC vorgegeben. MOSAIC arbeitet mit Real-Time-Workshop-Simulationscode der auf eine Echtzeit-Plattform ausgelegt ist und deshalb mit fester Schrittweite rechnen muss. MOSAIC will damit auch Komponenten für den echtzeitfähigen SMP2-Simulator Eurosim erzeugen.

Der Simulationscode sollte also mit einem Löser mit fester Schrittweite rechnen. Ist dies nicht möglich, muss ein Löser variabler Schrittweite dazu gebracht werden, mindestens auch alle festen Zeitschritte zu berechnen.

Die Schrittweite wird bei Simulationsbeginn als Parameter festgelegt. Der Einfachheit halber wird allen Komponenten einer SMP2-Simulation zudem die gleiche feste Schrittweite zugewiesen. Unterschiedliche Schrittweiten können in SMP2 nur unzuverlässig be-

trieben werden. In einem Schedule lassen sich Ereignisse mit unterschiedlichen Taktungen definieren. Fallen zwei Ereignisse aber auf denselben Zeitpunkt, wird unter SIMSAT ihre Reihenfolge zufällig bestimmt. SMP2 sollte so geändert werden, dass bei zeitgleichen Ereignissen eine Reihenfolge definiert ist. Bis dahin ist der Schedule zur Steuerung unterschiedlich getakteter Komponenten ungeeignet.

3.4. Propagierung von Werten

SMP2-Komponenten sollen bei Ausführung untereinander Werte abgleichen können. Auch dies muss unabhängig vom zur Modellierung gewählten Werkzeug geschehen, und wird deshalb über SMP2 realisiert.

So wie Blöcke in Blockdiagrammen weist eine Komponente Eingangs- und Ausgangsvariablen auf. Sie können auf SMP2-Ebene explizit als Eingangs- bzw. Ausgangsvariable markiert werden. Paare von Ein- und Ausgangsvariablen können mittels SMP2 miteinander verknüpft werden; dann sollen ihre Werte zu gegebener Zeit synchronisiert werden. Als SMP2-Mechanismus zur Synchronisation solcher Ein-/Ausgangsvariablenpaare wird der so genannte *Field-Link* [ESO05d] genutzt.

Ein Field-Link ist eine logische Verknüpfung zwischen zwei SMP2-Variablen, auch Fields genannt. Eine Variable ist dabei die Quelle, die andere das Ziel der Übertragung (Abbildung 3). Ein Field-Link kann als Ereignis in einen SMP2-Schedule eingetragen werden. Wird das Ereignis aktiv, wird der Wert der Zielvariablen mit dem Wert der Quellvariablen überschrieben. Als Übertragungsereignis betont das Konzept Field-Link den Datenfluss zwischen Komponenten. Es ist daher wegen der Nähe zum Blockdiagramm für die vorliegende Anwendung geeignet.

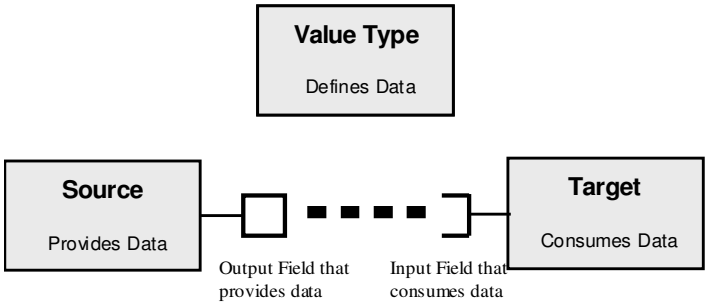


Abbildung 3: Schema des SMP2-Field-Link-Mechanismus. Aus [ESO05c].

Es sind also zur Synchronisierung von Ein- und Ausgangsvariablen in SMP2 nur entsprechende Felder im SMP2-Modell anzulegen. Der Zugriff darauf wird mit SMP2-Mechanismen ermöglicht, und ist damit unabhängig vom zugrunde liegenden Simulationscode. Field-Variablen können Referenzen auf die eigentlichen Variablen im zugrunde liegenden Simulationscode sein. Es können aber auch Duplikate sein, dann müssen die Duplikate untereinander innerhalb der Komponente geeignet synchronisiert werden.

Alternative SMP2-Kommunikationsmechanismen sind der Interface-Link, der Event-Link oder der direkte Operationsaufruf (Dynamic Invocation). Der *Interface-Link* erfordert die Definition eines Interfaces für jedes Modell, das Werte über eine Get-Funktion bereitstellen soll, oder das Werte über eine Set-Funktion erhalten soll. Es müssten unterschiedliche Interfaces verwaltet und zugeordnet werden. Das steigert den Implementierungsaufwand. Es verwirrt aber auch den Entwickler, denn Interfaces können auch an anderen Stellen eines Modells auftauchen, müssen aber nicht der Kommunikation dienen.

Beim *Event-Link* markiert eine Komponente ein Ereignis. Auf SMP2-Ebene wird festgelegt, welche Komponenten von diesem Ereignis informiert werden. Entweder definiert man hier die Bereitstellung eines neuen Ausgangswertes als Ereignis, oder die Anforderung eines neuen Eingangswertes. Auch hier wird mit der Definition von Ereignisquellen, -senken und -behandlungen der Implementierungsaufwand erhöht. Ebenso können Ereignisse mit anderen Bedeutungen verwirren.

3.5. Allgemeiner Einbettungsprozess

Abbildung 4 beschreibt die allgemeinen Schritte zur SMP2-Einbettung eines Simulationsmodells.

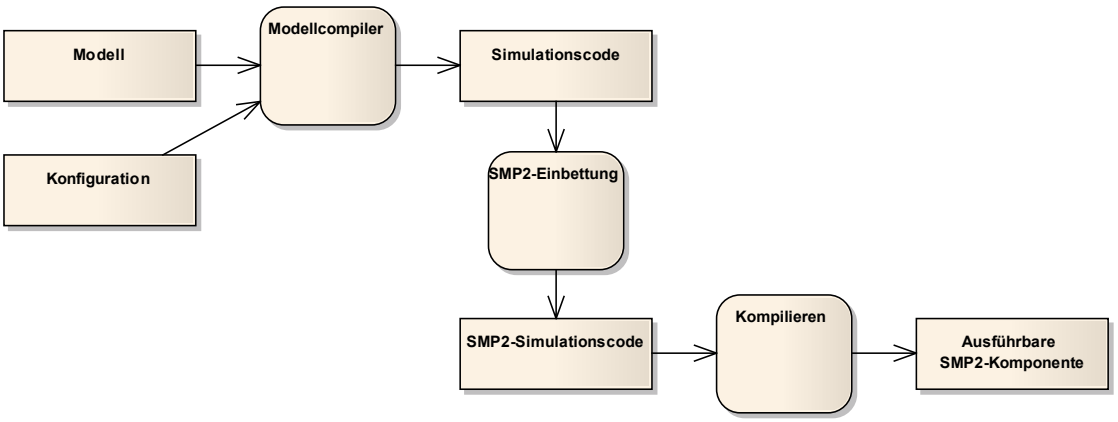


Abbildung 4: Übersicht über die allgemeinen Schritte zur SMP2-Einbettung eines Simulationsmodells. Die Aktionen Modellcompiler und SMP2-Einbettung sind für jedes verwendete Werkzeug spezifisch zu implementieren.

Das *Modell* und die *Konfiguration* eines Simulationslaufes sind vorgegeben. Ein Ingenieur hat sie vorher erstellt. Ein geeigneter *Modellcompiler* erstellt aus diesem Modell *Simulationscode*. Die hier eingesetzten Modellcompiler erstellen C-Code oder gemischten C-/C++-Code.

Bei der *SMP2-Einbettung* wird der Simulationscode in einem SMP2-Modell gekapselt. Auf Code-Ebene ist das eine Klasse, die die SMP2-Schnittstelle realisiert. Die Klasse definiert eine Entrypoint-Funktion, welche die gekapselte Berechnung eines Simulationsschrittes ausführt. Sie definiert auch relevante Variablen, insbesondere Ein- und

Ausgangszustände, Parameter und Integratorzustände auf SMP2-Ebene. So entsteht der *SMP2-Simulationscode*.

Wird der SMP2-Simulationscode kompiliert und mit allen benötigten Bibliotheken wie der evtl. modifizierten Laufzeitbibliothek verbunden, entsteht eine Bibliotheksdatei, die ein SMP2-Simulator direkt ausführen kann. Alle beschriebenen Schritte sind plattformunabhängig, mit Ausnahme der Kompilierung. Die Zielplattform der Kompilierung ist die Ausführungsplattform des SMP2-Simulators. Da zur Simulation SIMSAT 4 genutzt werden soll, ist die Zielplattform Linux.

3.6. Implementierungswerkzeuge

Zur Einbettung von SMP2-Modellen sind ganz unterschiedliche Maßnahmen erforderlich. Zum einen müssen externe Programme mit unterschiedlichen Parametern aufgerufen werden. Zum anderen müssen ihnen bestimmte Dateien zur Verfügung gestellt werden. Sie generieren aber auch Dateien, die wiederum für andere Einbettungsschritte aufbereitet werden müssen. Es müssen daher Dateien referenziert, erzeugt, gelöscht und bewegt werden. Darüber hinaus müssen manche Dateien auch direkt manipuliert, also gelesen und geändert werden. Build-Tools sind auf den Umgang mit Dateien und das Aufrufen von externen Programmen spezialisiert. Da Ausführung von Einbettungen nicht auf eine Architektur oder ein Betriebssystem beschränkt sein soll, ist ein plattformunabhängiges Build-Tool nötig. Deshalb wird zur Prozesssteuerung und Dateiverwaltung das Build-Tool *Ant* [Loughran2007] verwendet.

Die Einbettung besteht aber auch aus dem Modifizieren der in Teilschritten erzeugten Dateien. Hier ist Ant nicht geeignet. Da die Einbettung im Projekt *Virtueller Satellit* zur Anwendung kommen soll, und dort Java benutzt wird, sollten auch die Modifikationen in Java laufen. Als Sprache wurde hier *Groovy* [Koenig2007] gewählt. Groovy ist eine Skriptsprache, die direkt Java-Byte-Code erzeugen kann.

3.6.1. Apache Ant

Die besondere Stärke von *Ant* liegt in der plattformunabhängigen Ausführung externer Programme. Diese Unabhängigkeit ist erwünscht, da viele der benutzten Programme für verschiedene Plattformen verfügbar sind. So könnte die Einbettung mit MATLAB, RTW und MOSAIC vollständig unter Windows oder Linux vollzogen werden. Gegebenenfalls müssen so keine neue Lizenzen angeschafft werden.

Ant kümmert sich um die synchrone Ausführung externer Programme und um die Leerung/Befüllung aller Ein- und Ausgabepuffer. Die Ausführung externer Programme unter verschiedenen Plattformen unterscheidet sich normalerweise bei verwendeten Verzeichnistrennzeichen (“/” unter Linux, “ünter Windows) und bei Listentrennzeichen (”:ünter Linux, “;” unter Windows), aber auch bei Leerzeichen in Argumenten , bei der Bedeutung von Sonderzeichen, bei der Auflösung von Gruppierungszeichen wie den Anführungszeichen.

Befehlsargumente an externe Programme müssten auf jeder Plattform unterschiedlich zusammengestellt werden. Bei den Einbettungen sind System sind Pfade statisch abgelegt, werden aber auch zur Laufzeit als Argumente oder Parameter erwartet. Sie müssten immer erst aufwändig auf die aktuelle Plattform umgewandelt werden. Ant macht dies automatisch.

Die Kompilierung findet nur unter Linux statt, da die aktuelle SIMSAT-Version nur Linux unterstützt. Mit der Wahl von Eclipse als Grundlage ist aber zu erwarten, dass Windows gleichermaßen unterstützt werden könnte.

Zudem ist Ant leicht aus Java-Prozessen einbind- und steuerbar. Das ist für die Anbindung an den *Virtuellen Satelliten* von Bedeutung. So wird für die Anbindung lediglich eine Java-Laufzeitumgebung benötigt.

3.6.2. Groovy

Groovy-Quelltexte werden wie Java-Programme in Java-Bytecode übersetzt. Gültige Java-Programme sind auch gültige Groovy-Programme. Aber Groovy stellt zusätzlich aus Skriptsprachen bekannte Konzepte zur Verfügung. So bietet es dynamische Typisierung und erweitert die aus Java bekannte Syntax, um häufig genutzte Funktionalität, wie Maps, Listen oder so genannte Closures direkt als Sprachelemente zur Verfügung zu stellen. Im Vergleich zu äquivalenten Java-Programmen resultieren wesentlich kürzere Quelltexte. Sie sind übersichtlicher und leichter verständlich. Deshalb wird hier Groovy verwendet.

Eine generische Programmiersprache wie Groovy ist wiederum nicht für die Prozesssteuerung geeignet, wegen den erwähnten Problemen mit der Plattformunabhängigkeit, den manuell zu leerenden/befüllenden Ein- und Ausgabepuffern und der manuellen Festlegung, ob synchron oder asynchron ausgeführt wird. Deshalb werden Ant und Groovy in Kombination zur Implementierung der Einbettungsschritte genutzt.

Groovy wird an Stellen genutzt, wo XML-Dateien oder Quelltexte modifiziert werden. Beim Verarbeiten einfacher XML-Dateien werden der Groovy-eigene XML-Parser und XML-Generator verwendet, die mit eigenen Sprachelementen bedient werden können. Für komplexere Eingriffe in XML-Dateien wird der SAX-Parser der JDOM-Bibliothek[1] genutzt.

Zur Modifikation von Quelltexten der Sprache C++ werden reguläre Ausdrücke verwendet. Dieser Ansatz hat sich als ausreichend robust heraus gestellt, um auf automatisch generierte Quelltexte angewendet zu werden. Als Alternative wurde die Verwendung eines C++-Parsers erwogen. Aber selbst wenn die zu verändernden Quelltexte als abstrakter Syntaxbaum vorlägen, müssten immer noch weitreichende Annahmen über die Beschaffenheit der Codestrukturen getroffen werden, die mit einer neuen Version oder der Verwendung ungetesteter Features ungültig sein könnten. Die Verwendung regulärer Ausdrücke erlaubte eine schnellere Umsetzung von Modifikationen. Es ist aber davon

[1] http://jdom.org/ (am 9.11.2009 zuletzt eingesehen)

auszugehen, dass dieser Ansatz schwieriger zu Warten ist als eine Modifikation über Syntaxbäume.

4. Einbettung von Simulink-Modellen

Für die SMP2-Einbettung von Simulink-Modellen gibt es eine vorhandene Lösung. Wie in Abbildung 5 gezeigt, wird dafür mit dem Programm *Real-Time Workshop* Simulationscode generiert. Das Programm MOSAIC vollzieht die SMP2-Einbettung, wie sie im Wesentlichen im vorigen Kapitel beschrieben wurde. Beide Programme und ihre Rolle bei der Einbettung werden in den folgenden Abschnitten genauer vorgestellt.

Ein von MOSAIC erzeugtes SMP2-Modell kann aber noch nicht kommunizieren. Daher muss das SMP2-Modell anschließend in diesem aber auch in weiteren Punkten angepasst werden. Erst wenn der angepasste Simulationscode kompiliert und mit der Laufzeitbibliothek des Real-Time-Workshop verbunden wird, entsteht eine ausführbare SMP2-Komponente die mit anderen gekoppelt werden kann. Die erforderlichen Anpassungen werden zuerst im Zusammenhang beschrieben. Danach wird erklärt, wie sie umgesetzt wurden. Schließlich werden noch Aspekte der Nutzung so erstellter SMP2-Modelle beleuchtet.

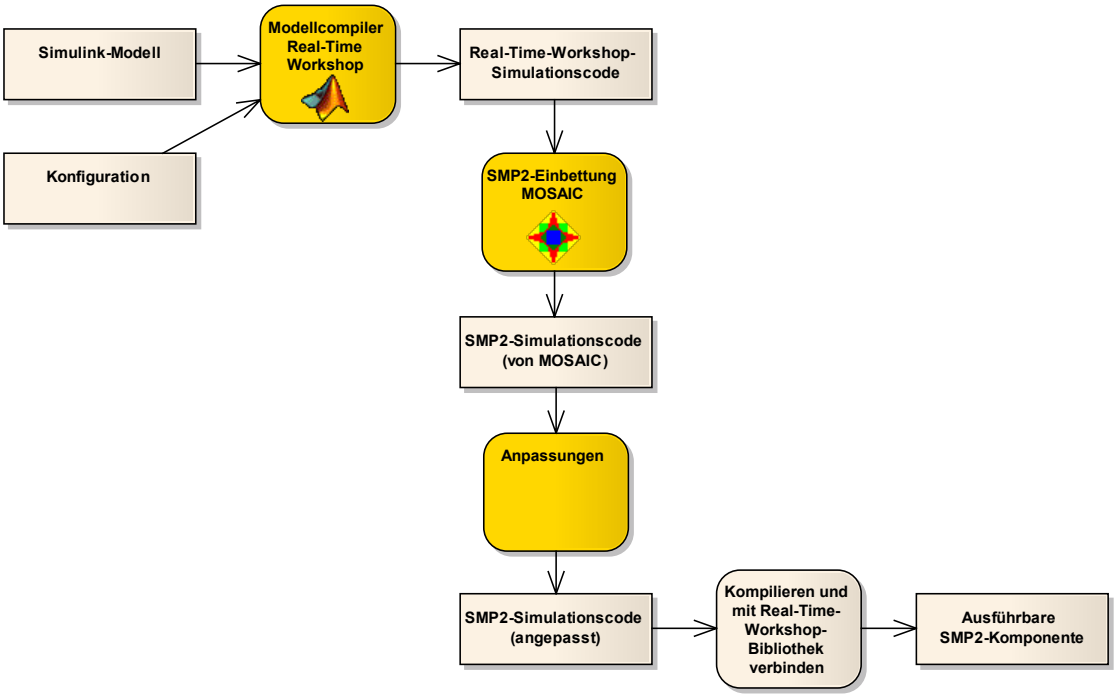


Abbildung 5: Übersicht über die speziellen Schritte zur SMP2-Einbettung eines Simulink-Modells. Markierte Schritte werden eingehend behandelt.

4.1. Modellcompiler Real-Time Workshop

Das Programm *Real-Time Workshop* (RTW) ist ein Zusatzmodul zur Modellierungs- und Simulationsumgebung Simulink. Es erzeugt für ein Simulink-Modell Code, der die

Simulationsberechnung isoliert durchführt. Der erzeugte Code berechnet die gleichen Ergebnisse wie eine Simulation in Simulink.

Real-Time Workshop kann echtzeitfähigen Code für echtzeitfähige Hardwarearchitekturen erzeugen. Der Code kann dann dort ohne MATLAB, Simulink oder Real-Time Workshop ausgeführt werden. Real-Time Workshop wird aber auch ohne die Echtzeit-Anforderung eingesetzt, um Simulink-Modelle als Prototypen für eingebettete Software auszuprobieren. In diesem Fall kann der Code sowohl Löser mit fester als auch Löser mit variabler Schrittweite einsetzen.

Die Arbeitsschritte bei der Ausführung des Real-Time Workshops bestehen aus Konfiguration und Auslösung der Codegenerierung. Beide Schritte können über die grafische Benutzeroberfläche von Simulink ausgeführt werden, oder mit MATLAB-Befehlen über die MATLAB-Shell. In dieser Arbeit wird der Real-Time Workshop automatisch über die MATLAB-Befehle gesteuert.

Real-Time Workshop, Simulink und MATLAB sind für verschiedene Betriebssysteme verfügbar, unter anderem für Windows und für Linux/UNIX. In dieser Arbeit kommt Real-Time Workshop 6.3 zum Einsatz, wie er zusammen mit Simulink 6.3 im Programmpaket "MATLAB 7.1 Release 14 Service Pack 3" ausgeliefert wird. Diese Anforderung wird vom Programm MOSAIC gestellt und weiter unten erklärt.

Anforderungen an Simulink-Modelle

Um generell Code mit dem Real-Time Workshop erstellen zu können, dürfen keine algebraischen Schleifen im Modell vorhanden sein. Eine *algebraische Schleife* ist ein gerichteter Kreis im Blockdiagramm, der ausschließlich über zustandslose Blöcke führt. Siehe Beispiel in Abbildung 6. Simulink kann mit einfachen algebraischen Schleifen wie in der Abbildung umgehen [SIM05]. Dazu führt es in jedem Simulationsschritt ein Annäherungsverfahren aus, das einen Gleichgewichtspunkt der Schleife ermittelt. Für das Beispiel $z = u - z$ ist die Lösung $z = u/2$ offensichtlich, Simulink muss aber trotzdem die Annäherung durchführen. Wenn Simulink keine Konvergenz herbeiführen kann, bricht die Berechnung mit einem Hinweis auf die algebraische Schleife ab.

Der Real-Time Workshop kann laut [LM06] nicht mit algebraischen Schleifen umgehen. Ausnahmen seien Schleifen über geschaltete Teilsysteme (triggered subsystems), Schleifen zum Reset-Port eines Integrators und weitere, die aber nicht ausgeführt werden.

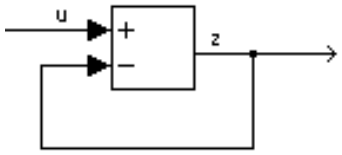


Abbildung 6: Beispiel einer algebraischen Schleife in Simulink. Aus [SIM05].

Zur Modellierung können sonst alle Blöcke benutzt werden, mit Ausnahme der Blöcke "MATLAB function" und "S-function", wenn sie M-Quelltexte aufrufen. [LM06] empfiehlt, auch keine ToWorkspace-Blöcke zu verwenden. Mit ToWorkspace-Blöcken lassen sich Si-

gnale als Variablen in den MATLAB-Workspace ausgeben. Stattdessen sollen Ausgangs-Ports verwendet werden. Um Signale in den MATLAB-Workspace auszugeben, kann man auch die Signal-Protokollierung von Simulink benutzen.

Alle Ein- und Ausgänge müssen mit Simulink-Ports explizit verknüpft sein, wenn sie im SMP2-Modell verfügbar sein sollen. Real-Time Workshop legt dann entsprechende Variablen in seinen Ein-/Ausgangs-Datenstrukturen an. MOSAIC erzeugt für diese beiden Datenstrukturen Äquivalente auf SMP2-Ebene.

Einstellungen

Um den mit Real-Time Workshop erstellten Code später per MOSAIC in SMP2 einbetten zu können, müssen folgende Einstellungen getroffen werden [LM06].

Es muss ein spezieller *Target Language Compiler* verwendet werden: Generic Real-Time Target with Dynamic Memory. Siehe dazu die MOSAIC-Anforderungen im nächsten Abschnitt.

Es muss ein Lösungsverfahren fester Schrittweite verwendet werden. [LM06] empfiehlt *ode5* [SIM05], das Standardverfahren für feste Schrittweiten in Simulink. Es ist ein Runge-Kutta-Verfahren der Ordnung 5. Das Lösungsverfahren kann noch im MOSAIC-Einbettungsschritt geändert werden. Falls das Modell keine kontinuierlichen Zustände enthält, wird vom Real-Time Workshop automatisch ein diskretes Lösungsverfahren verwendet. In diesem Fall kann das Verfahren auch in MOSAIC nicht mehr geändert werden.

4.2. SMP2-Einbettung mit MOSAIC

MOSAIC (Model-Oriented Software Automatic Interface Converter) wurde vom niederländischen Luft- und Raumfahrtlaboratorium (NLR - Nationaal Lucht- en Ruimtevaartlaboratorium) entwickelt. Aktuell ist die Version 7.1 aus dem Jahr 2005. Diese Version ist auf die damals aktuelle MATLAB-Version Release 14 ServicePack 3 ausgerichtet. Um MOSAIC betreiben zu können, müssen MATLAB, Simulink und Real-Time Workshop deshalb in dieser Version vorliegen.

Das Programm *MOSAIC* bettet Real-Time-Workshop-Simulationscode in ein SMP2-Modell ein. Dazu parst es die vom Real-Time Workshop produzierten Quelltexte und extrahiert Modell- und Laufzeitinformationen. Aus diesen Informationen generiert es unter anderem ein SMP2-Modell als C++-Quellcode. In dieser Code-Implementierung des SMP2-Modells ist die Übertragung der Berechnungsausführung auf SMP2-Mechanismen und die Bekanntmachung von Variablen in SMP2 programmiert.

4.2.1. Übertragung der Ausführung

Der Simulationscode des Real-Time Workshops (RTW) wird in den generierten SMP2-Code durch Funktionsaufrufe und Initialisierungen eingebettet (Abbildung 7). Als Beispiel eine Funktion des SMP2-Codes, die entsprechende Funktionen der Laufzeitumgebung aufruft:

```
rtModel_battery* battery_init(double step_size, int *status)
{
    rtModel_battery *S;
    [...]
    rtmSetStepSize(S, step_size);
    rtsiSetFixedStepSize(S->solverInfo, step_size);
    rtmiInitializeSizes(rtmGetRTWRTModelMethodsInfo(S));
    rtmiInitializeSampleTimes(rtmGetRTWRTModelMethodsInfo(S));
    RetVal = rt_SimInitTimingEngine(...);
    rt_ODECreateIntegrationData(rtmGetRTWSolverInfo(S));
    [...]
}
```

Diese Funktion wird dann wie im folgenden Auszug ersichtlich auf eine Entrypoint-Funktion übertragen:

```
void ::battery::battery::battery_init_Handler()
{
    [...]
    _status = 0;
    if (!this->_model) {
        this->_model = ::battery_init(0.005, &_status);
        [...]
    }
    assert(!_status);
    [...]
}
```

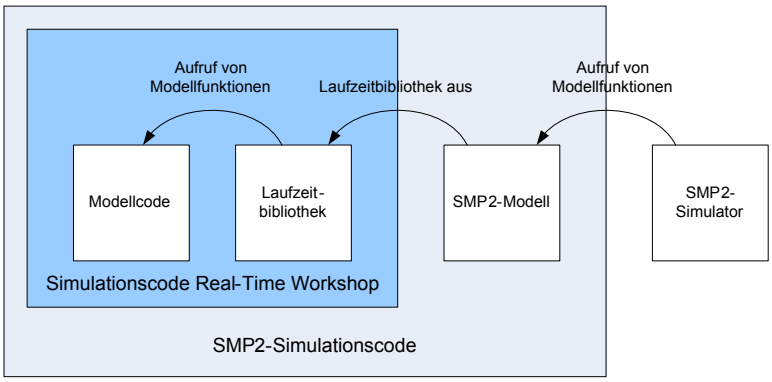


Abbildung 7: Einbettung des Simulationscodes und Aufrufreihenfolge.

Die Entrypoint-Funktion kann über SMP2-Mechanismen ausgeführt werden; etwa durch die zeitliche Taktung in einem Schedule-Dokument. So kann ein SMP2-Simulator regelmäßig einen Berechnungsschritt einer Komponente ausführen.

Die Initialisierung des Modells wird im Konstruktor der Modell-Klasse ausgeführt. Erforderliche Aufräumarbeiten werden entsprechend im Destruktor ausgeführt.

4.2.2. Referenzierung von Variablen

Das SMP2-Modell referenziert auch Variablen und Datenstrukturen vom RTW-Simulationscode (Abbildung 8). Im Simulationscode sind alle Variablen in Strukturen zusammengefasst. Deshalb deklariert auch MOSAIC äquivalente Strukturen auf SMP2-Ebene und lässt dann SMP2-Instanzen dieser Strukturen auf die RTW-Strukturen verweisen. Die definierten SMP2-Strukturen lassen sich aber nicht zur Kommunikation in SIMSAT nutzen. Dies wird im Abschnitt 4.3.3 weiter ausgeführt.

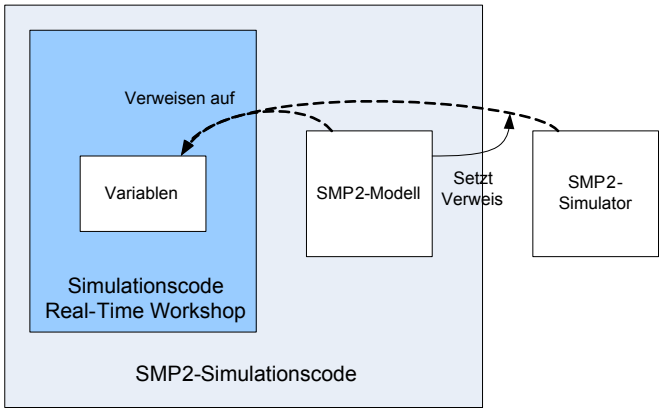


Abbildung 8: Einbettung der Variablen im Simulationscode.

MOSAIC setzt die Äquivalenz der bei der Referenzierung beteiligten Datentypen voraus. So ist z.B. der SMP2-Typ `Float64` im ebenfalls standardisierten C++-Mapping [ESO05a] auf den C++-Typ `double` festgeschrieben. Der als Äquivalent dazu genutzte RTW-Typ `real_T` lässt sich jedoch mit Compilerschaltern auf `double` ebenso wie auf `float` konfigurieren. Auf der zur Entwicklung und Simulation verwendeten Rechnerarchitektur x86 unter Windows XP und Linux entsprachen `Float64` und `real_T` einander. Bei Verwendung von MOSAIC auf anderen Rechnerarchitekturen sollte erst die Typengleichheit geklärt werden.

Die Verwendung der Strukturen sei an einem Beispiel erklärt. Darin sind alle Modellvariablen des Modells *battery*, die einen Ausgang des Blocks repräsentieren, in der Struktur `ExternalOutputs_battery` im RTW-Code gespeichert:

```
/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  real_T Out1;                         /* '<Root>/Out1' */
} ExternalOutputs_battery;
```

MOSAIC legt dafür eine SMP2-Struktur names `ExternalOutputs` an. Die vorhandene Ausgabevariable vom Typ real_T wird mit einer Variable vom Typ ::Smp::Float64 repräsentiert:

```
struct ExternalOutputs {
    ::Smp::Float64 Out1; // <Root>/Out1
    static void _Register(::Smp::Publication::ITypeRegistry* registry);
};
```

Zum Arbeiten werden Pointer auf die jeweiligen Strukturen angelegt. Die Struktur rtModel_battery enthält mit dem geschachtelten Eintrag ModelData.outputs wiederum einen Pointer auf die Struktur `ExternalOutputs_battery`:

```
struct ::rtModel_battery* _model;
struct ::battery::ExternalOutputs * _Y;
```

Während der verschiedenen Initialisierungsschritte wird für Berechnungen innerhalb des MOSAIC-Codes der Pointer `_Y` auf die RTW-Struktur gesetzt. Zur Benutzung auf SMP2-Ebene wird eine SMP2-Field-Variable mit der Adresse der RTW-Struktur publiziert:

```
this->_Y = (struct ::battery::ExternalOutputs *)
           this->_model->ModelData.outputs;
[...]
receiver->PublishField(
  "battery_Y", "Output",
  ((struct ::battery::ExternalOutputs *)(this->_model->ModelData.outputs)),
  ::battery::Uuid_ExternalOutputs, true, true, true, true);
```

4.2.3. SMP2-Dokumente

MOSAIC generiert zusätzlich zum Quelltext auch die Beschreibung der Struktur des SMP2-Modells als Catalogue-Dokument. Um das SMP2-Modell unter SIMSAT auch ausführen zu können, werden zudem noch ein Assembly- und ein Schedule-Dokument generiert. Die von MOSAIC generierten SMP2-Dokumente sind aber nicht standardkonform [Rö08]. Für die Benutzung im DLR-Projekt Virtueller Satellit werden Assembly und Schedule nicht benötigt, da die von MOSAIC erzeugte Komponente nicht einzeln simuliert werden soll. Werden im Virtuellen Satelliten Komponenten zu einer Simulation gekoppelt, werden die benötigten SMP2-Dokumente dazu neu erstellt. Das Catalogue-Dokument ist bis auf die Verletzung einiger Empfehlungen aber standardkonform.

4.2.4. Anforderungen an Simulink-Modelle

MOSAIC führt weitere Anforderungen an Simulink-Modelle ein. Alle Beschränkungen werden im Anhang A.1 zusammengefasst.

Es können keine *Scope*-Blöcke verwendet werden. Abgesehen davon, dass diese nur für die Ausführung in Simulink gedacht sind, führen sie zu einem Fehler in MOSAIC. MOSAIC erzeugt zwar erfolgreich SMP2-Simulationscode, dieser lässt sich aber wegen fehlender Strukturdefinitionen nicht kompilieren.

Der Name des Modells muss 10 Zeichen kürzer sein, als in den Real-Time-Workshop-Einstellungen in Simulink angegeben. Bei der Standardeinstellung von 31 Zeichen darf der Modellname nicht länger als 21 Zeichen sein. Diese Einschränkung ist offensichtlich ein Fehlverhalten in MOSAIC. Bei der Generierung der SMP2-Modellklasse werden Bezeichner deklariert, die aus dem Modellnamen durch Anhängen weiterer Zeichen hervorgehen. Die Zeichenbeschränkung aus MATLAB wird dann willkürlich auch auf diese Bezeichner angewendet. Z.B. wird für ein Modell "Akkumulatormodell_R14SP3" die Entrypoint-Funktion `Akkumulatormodell_R14SP3_terminate_Handler()` deklariert, an anderer Stelle aber mit `Akkumulatormodell_R14SP3_termin_Handler()` aufgerufen.

Ferner dürfen Modelle keine Namen von Standard-C++-Funktionen/Bibliotheken tragen (z.B. "random"), weil es sonst zu Namensraumüberschneidungen kommt.

4.2.5. Sonstige Voraussetzungen

MOSAIC erfordert, dass der Real-Time Workshop mit dem Target "Generic Real-Time Target with dynamic memory allocation" konfiguriert wird. Für dieses Target kann der Real-Time Workshop aber nur Lösungsalgorithmen mit fester Schrittweite benutzen. Für Targets zum so genannten "Rapid Prototyping" könnte er auch variable Schrittweiten berechnen. Aber MOSAIC ist auf feste Schrittweiten festgelegt [LM06]. Spätestens mit der Anwendung von MOSAIC wird der Lösungsalgorithmus festgelegt.

MOSAIC benötigt Quelltextdateien der RTW-Laufzeitbibliothek. Wie in [LM06] beschrieben, müssen ausgewählte Quelltextdateien in einen von MOSAIC bestimmten Ordner kopiert werden. Da die Auswahl aus Erfahrungen bestimmt wurde, ist sie vielleicht nicht vollständig und es können Fehler beim Verbinden des Codes entstehen. In diesem Fall sind die fehlenden Dateien zu ermitteln und der Auswahl hinzuzufügen.

Sowohl MOSAIC selbst als auch auch die zum Kompilieren erzeugten Makefiles sind für Windows ausgelegt. Da als Ausführungsumgebung aber SIMSAT unter Linux genutzt werden soll, können nur die reinen Quelltexte benutzt werden, die plattformunabhängig sind.

4.3. Anpassungen

Das von MOSAIC erstellte SMP2-Modell kann noch nicht einfach mit anderen Modellen kommunizieren. Bei der Entwicklung des Einbettungsprozesses sind außerdem mehrere Fehler bei der Kompilierung des von MOSAIC erstellten SMP2-Simulationscodes aufgefallen. Um die Modelle an die Anforderungen im Virtuellen Satelliten vorzubereiten, sind deshalb Anpassungen des SMP2-Modells notwendig.

4.3.1. Publikation der Entrypoints

In den Entrypoints des SMP2-Modells werden verschiedene Abschnitte der RTW-Laufzeitbibliothek aufgerufen. Sie werden dem Simulator bekannt gemacht, damit sie über SMP2-Mechanismen ausführbar sind. MOSAIC publiziert die Entrypoints des Modells explizit im Code:

```
void ::battery::battery::Publish(::Smp::IPublication* receiver) [...]
{
    [...]
    (void)receiver->PublishOperation( "battery_init_Handler",
            "Initalisation",
            ::Smp::Publication::Uuid_Void);
    (void)receiver->PublishOperation( "battery_input_Handler",
            "Input gathering",
            ::Smp::Publication::Uuid_Void);
    (void)receiver->PublishOperation( "battery_output_Handler",
            "Output generation",
            ::Smp::Publication::Uuid_Void);
    (void)receiver->PublishOperation( "battery_terminate_Handler",
            "Termination",
            ::Smp::Publication::Uuid_Void);
    (void)receiver->PublishOperation( "battery_update_Handler",
            "Timestep update of all state variables",
            ::Smp::Publication::Uuid_Void);
}
```

Zur Laufzeit bricht SIMSAT das Laden der Komponente mit einem Fehler während der Publizierung ab. Lässt man die explizit von MOSAIC vorgenommene Publizierung jedoch weg, werden die Entrypoints trotzdem ausgeführt. Anscheinend sind sie dem Simulator bereits von anderer Stelle bekannt. Es ist nicht klar, ob dieses Verhalten ein Fehler von MOSAIC oder eine Fähigkeit von SIMSAT ist. Jedenfalls werden zur Nutzung dieser MOSAIC-Version und dieser SIMSAT-Version die expliziten Publizierungen entfernt.

4.3.2. Auswahl der Entrypoints

Die von MOSAIC definierten Entrypoints sind `init_Handler`, `terminate_Handler`, sowie `input_Handler`, `update_Handler` und `output_Handler`, jeweils mit dem Modellnamen davor. Die Entrypoints `init_Handler` und `terminate_Handler` waren in den Testmodellen immer leer. Aus dem erzeugten Code geht außerdem hervor, dass sie nicht zur Ausführung im SMP2-Schedule vorgesehen sind, sondern fest im Modell-Konstruktor bzw. -Destruktor im Code aufgerufen werden. Deshalb brauchen sie nicht über SMP2-Mechanismen angesteuert werden.

Die anderen drei Entrypoints führen verschiedene Aspekte eines Berechnungsschrittes durch und sind zur Ausführung im SMP2-Schedule vorgesehen. Ihre Namen ähneln denen der RTW-Modellcode-Funktionen (Siehe Abbildung 2) und haben auch fast die gleiche Bedeutung. Die Funktion `output_Handler` führt direkt die RTW-Modellfunktion `MdlOutputs` aus. Die Funktion `update_Handler` fasst mehrere Aufrufe zusammen. Sie führt mehrere, für einen Zeitschritt nötige Anweisungen aus der Laufzeitbibliothek aus. Aus Abbildung 2 ersichtlich, sind das unter Anderem der Aufruf von `MdlUpdate` und die Integrationsschleife mit mehreren Aufrufen von `model_Derivatives` und `MdlOutput`. Die Funktion `input_Handler` ist von MOSAIC hinzugefügt und hat keine Entsprechung in RTW-Modellfunktionen. Sie bietet dem Anwender die Möglichkeit, eigene Eingangssignale bereitzustellen. Sie muss dazu aber für jedes Modell erst implementiert werden.

Die drei Funktionen können zu einem Entrypoint zusammengefasst werden, da sie immer in der gleichen Reihenfolge ausgeführt werden. Auch werden sie nicht einzeln aufgerufen, mit einer Ausnahme. Diese wird weiter unten im Abschnitt 4.3.6 beschrieben. Die Reihenfolge sollte in Anlehnung an die Ausführung der RTW-Modellfunktionen sein: `input_Handler`, `output_Handler` dann `update_Handler`.

4.3.3. Variablenduplikation

SIMSAT 4 soll als Simulator für SMP2-Komponenten genutzt werden. Als Kommunikationsmechanismus sollen SMP2-Field-Links genutzt werden, da sie die adäquate Beschreibung für den Datenfluss sind.

SIMSAT 4 kann aber keine Field-Links zwischen Strukturelementen aufbauen. Das ist ein Programmfehler und sollte in folgenden Versionen behoben werden. Um den Fehler zu umgehen, werden alle Ein- und Ausgabevariablen mittels neuer Variablen dupliziert, die nicht in Strukturen zusammengefasst sind. Es werden der Catalogue und der Code geändert. Dazu müssen in der von MOSAIC angelegten Input-Methode die Werte in den zusätzlich angelegten Input-Variablen in die Strukturen geschrieben werden, und in der Output-Methode die Werte aus den Strukturen in die speziellen Output-Variablen.

Die duplizierten Variablen müssen denselben Datentyp aufweisen wie ihre Originale aus den SMP2-Strukturen. In verwendeten Testmodellen traten die Typen Float64, UInt8 und Float64ArrayX (Arrays mit beliebiger aber fester Länge X) auf. Diese werden bei der Einbettung berücksichtigt. Weitere Typen werden momentan nicht benötigt, sind bei Bedarf aber mit geringem Aufwand hinzuzufügen.

Die Namen der duplizierten Variablen müssen innerhalb eines SMP2-Modells eindeutig sein. Darauf ist zu achten, wenn eine Eingangsvariable den gleichen Namen wie eine Ausgangsvariable trägt. Eine Lösungsmöglichkeit besteht darin, Variablennamen um eine Unterscheidung von Ein- und Ausgangsvariablen zu ergänzen. Da aber Modellierer Ein- und Ausgangsvariablen selbst häufig diese Information im Namen beigeben, sollte vor einer entsprechenden Umsetzung zuerst die Namensbildung geklärt werden.

Der Datentyp Float64ArrayX bedarf einer weiteren Anmerkung. Für Elementsamm-

lungen in SMP2-Modellen können nicht die C/C++-internen Arrays verwendet werden, denn SMP2 soll plattform- und damit sprachenunabhängig sein. Deshalb definiert SMP2 einen eigenen Datentypen für Elementsammlungen: `Array`. Damit können aber nur Arrays mit fest vorgegebener Länge definiert werden. `Float64Array3` wird beispielsweise definiert als:

```
typedef ::Smp::Mdk::Array< ::Smp::Float64, 3 > Float64Array3;
```

4.3.4. SMP2-Interface

MOSAIC definiert jeweils ein SMP2-Interface für jedes Modell. Das Interface deklariert keine Operationen oder Methoden, sondern ist leer. Vermutlich soll das Interface helfen, das Modell in einem SMP2-Container zu sammeln. SMP2-Container sind typisiert, sie können nur Modelle eines angegebenen SMP2-Typs beinhalten [ESO05d]. SMP2-Modelle, -Klassen und -Interfaces sind allesamt SMP2-Typen. Statt für jedes Modell ein eigenes Interface für die Typisierung zu definieren, könnte ein Container also auch direkt auf das Modell typisiert werden. Container können auch von den ihn zugewiesenen Typen abgeleitete Typen beinhalten.

Momentan gibt es keine einfache Möglichkeit, beliebig verschiedene Modelle in einen Container zu legen. Der Container müsste explizit auf jedes zu beinhaltende Modell typisiert werden. Stattdessen wird das von MOSAIC für jedes Modell angelegte SMP2-Interface geändert, so dass es vom in SMP2 definierten Interface *IModel* ableitet. So kann jedes mit MOSAIC erstellte Modell in generischen, auf IModel typisierten Containern enthalten sein.

4.3.5. Logging

Um die Berechnungen jedes SMP2-Modells nachvollziehen zu können, müssen die Werte der Modellvariablen in jedem Zeitschritt protokolliert werden. Dabei genügt es, nur die Ausgangsvariablen eines SMP2-Modells mitzuschreiben. Sie repräsentieren die Werte, die zwischen SMP2-Modellen ausgetauscht werden. Andere Variablen, wie interne Zustände oder interne diskrete Variablen von SMP2-Modellen könnten später problemlos zur Protokollierung hinzugefügt werden.

4.3.6. Initialisierung

Die Protokollierung der Ausgabevariablen von Testmodellen hat aufgezeigt, dass zum Zeitpunkt 0 in dem von MOSAIC erzeugten SMP2-Modell keine korrekten Startwerte vorliegen. Dieser Fehler ist bekannt und sollte mit Folgeversionen von MOSAIC behoben sein. Dennoch muss er hier behandelt werden.

Eine Betrachtung der originalen Quelltexte der RTW-Laufzeitbibliothek ergab, dass Real-Time Workshop zur Initialisierung und vor allen Zeitschritten die Modellfunktion

`MdlOutputs` aufruft. Dadurch werden noch vor dem ersten Berechnungsschritt an alle Blöcke des Modells Startwerte angelegt. Während der Initialisierung in MOSAIC-erzeugtem SMP2-Code fehlt dieser Aufruf. Nach Initialisierung der SMP2-Simulation besitzen die Modellvariablen deshalb nicht ihre Startwerte. Zwar wird dies im ersten Berechnungsschritt durch den dortigen Aufruf von `output_Handler` nachgeholt, und die Simulation rechnet mit korrekten Werten weiter; bei der Auswertung der Simulationsergebnisse müsste aber immer der Zeitschritt 0 zur Ausnahme erklärt werden. Die Funktion `output_Handler` muss bereits während der Initialisierung aufgerufen werden.

Der SMP2-Schedule ist keine geeignete Alternative zur Initialisierung. Selbst wenn ein einmaliger SMP2-Task zum Simulationszeitpunkt 0 definiert würde, wird dieser erst beim Simulationsbeginn ausgeführt, aber noch nicht bei der Modellinitialisierung. Es ist generell zu empfehlen, Initialisierung und Terminierung von Modellen streng von Ereignissen der Simulationszeit zu trennen.

4.4. Umsetzung

Die notwendigen Anpassungen betreffen sowohl die Struktur als auch die Implementierung des SMP2-Modells. Strukturelle Änderungen wie die Einführung eines Basisinterfaces, von dem das Modellinterface ableitet, müssen sowohl im Catalogue als auch im Quelltext des SMP2-Modells vorgenommen werden. Der Catalogue als XML-Dokument eignet sich auch besonders, um bestimmte Informationen auszulesen, z.B. die Liste der Ausgangsvariablen des Modells. Zunächst wird in Grundzügen vorgestellt, wie die beschriebenen Anpassungen im Catalogue und im Quelltext umgesetzt wurden. Schließlich wird noch die Umsetzung des Build-Prozesses erklärt, der die Ausführung der Einbettungsschritte automatisiert.

4.4.1. Catalogue-Modifikation

Aus der von MOSAIC erstellten Catalogue-Datei werden wichtige Informationen gelesen: Name und Typ aller Ein- und Ausgabevariablen und die von MOSAIC generierte Identifikationsnummer des Modells. Die Identifikationsnummer wird im Quelltext und zur Simulation im Assembly referenziert und sorgt dafür, dass der SMP2-Simulator die im Assembly definierten Instanzen der richtigen Klassenimplementierung im Quelltext zuordnet.

Um später Informationen für die Generierung eines Assemblys zu sammeln, ist der Catalogue wieder eine geeignete Informationsquelle. Deshalb muss auch der Catalogue modifiziert und aktuell gehalten werden, auch wenn er für die Simulation selbst ohne Bedeutung ist.

Der Catalogue wird mit dem SAX-Parser der JDOM-Bibliothek[2] in eine JDOM-Knotenstruktur eingelesen.

[2] http://jdom.org/ (am 9.11.2009 zuletzt eingesehen)

In der Knotenstruktur wird die Definition des Interfaces gesucht und um die Ableitung von `IModel` erweitert.

Für die Ermittlung der Ein- und Ausgabevariablen wird die Definition der Strukturen `ExternalInputs` bzw. `ExternalOutputs` gesucht. Diese wird jeweils gelesen und für jeden Eintrag Name und Typ gespeichert. Für jede so ermittelte Ein- bzw. Ausgabevariable wird eine neue SMP2-Field-Variable in der Knotenstruktur definiert. Die Liste der Ein- und Ausgabevariablen wird für spätere Verarbeitungsschritte zwischengespeichert.

Die von MOSAIC vergebene UUID-Identifikationsnummer des Modells wird ebenfalls gesucht und für die spätere Verarbeitung zwischengespeichert.

Die Knotenstruktur wird über die JDOM-Klasse `XML-Outputter` wieder als XML-Catalogue-Datei gespeichert.

4.4.2. Quelltext-Modifikation

Die Änderungen am Quelltext werden mit regulären Ausdrücken vorgenommen. Markante Stellen, wie eine bestimmte Funktionssignatur mit anschließenden Anweisungen wird gesucht, um dort Anweisungen einzufügen, oder bestimmte Anweisungen zu löschen.

Die Definition des Modell-Interfaces wird gesucht und so geändert, dass das Interface von `Smp::IModel` ableitet.

Für jede Ein- und Ausgabevariable, deren Name und Typ aus dem Catalogue bekannt sind, wird in der Modellklasse eine neue Variable deklariert. Die Stelle der Deklaration wird über einen regulären Ausdruck gesucht. Beispiel für die Ausgabevariable `Out1` eines Modells `battery`:

```
// Automatic insertion of wrapper field declarations.
private:
    ::Smp::Float64 Out1;
```

Um die deklarierten Variablen auf SMP2-Ebene benutzen zu können, werden sie veröffentlicht. Dazu wird die Publish-Methode der Modellklasse gesucht und um die Anweisung zur Veröffentlichung ergänzt. Im Beispiel die Veröffentlichung der Ausgangsvariablen `Out1`. Die letzten beiden Boolean-Argumente geben an, ob eine Variable Eingang und/oder Ausgang ist. Hier wird also festgehalten, dass `Out1` eine Ausgangsvariable ist:

```
// Automatic insertion for publication of wrapper fields.
receiver->PublishField("Out1", "", &Out1, true, true, false, true);
```

Wie im Abschnitt 4.3 erklärt, werden ebenfalls in der Publish-Methode alle Veröffentlichungen von Entrypoints entfernt, also z.B. die Anweisung:

```
(void)receiver->PublishOperation("battery_input_Handler",
                                 "Input gathering",
                                 ::Smp::Publication::Uuid_Void);
```

Schließlich müssen die deklarierten Variablen mit den entsprechenden Variablen in den Eingabe-/Ausgabestrukturen synchronisiert werden. Der Wert einer Eingabevariablen wird im Entrypoint `input_Handler` der entsprechenden Strukturvariablen zugewiesen. Im Entrypoint `output_Handler` wird der Wert der Strukturvariablen der Ausgabevariablen zugewiesen. Z.B.:

```
void ::battery::battery::battery_output_Handler()
{
    [...]
    // Automatic insertion of wrapper field updates.
    Out1 = _Y->Out1;
    [...]
}
```

Als Protokoll-Mechanismus wird der SMP2-Logger-Dienst in Anspruch genommen. SMP2 schreibt vor, das dieser Dienst von jedem SMP2-Simulator angeboten werden muss. Die Liste der Ausgabevariablen ist aus dem Catalogue-Dokument bekannt. Beim Anpassen des SMP2-Modells wird mit der Standard-C-Funktion `sprintf` ein String erzeugt, in dem die Werte aller Ausgabevariablen und der aktuellen Zeit stehen. Der String wird auf eine Länge von 20 Zeichen je Zahleneintrag plus 20 Zeichen allgemein initialisiert (im Beispiel 60 Zeichen für 2 Zahleneinträge). Der String wird dann als Log-Eintrag protokolliert:

```
void ::battery::battery::battery_output_Handler()
{
    try {
        _status = 0;
        ::battery_output(this->_model, &this->_t, &this->_status);
        assert(!_status);
        // Automatic insertion of wrapper field updates.
        Out1 = _Y->Out1;

        // Log output for this time instance
        char buffer[60];
        sprintf(buffer, "time: %f; Out1: %f", _t, Out1);
        this->m_simulator->GetLogger()->Log(
            this,
            buffer,
            ::Smp::Services::LMK_Information);
    } catch (...) {
```

```
        this->m_simulator->GetLogger()->Log(
            this,
            "Unhandled exception in battery_output_Handler",
            ::Smp::Services::LMK_Error);
    }
}
```

Um eine konsistente Variablenbelegung zum Zeitpunkt 0 zu gewährleisten, muss bei der Initialisierung die SMP2-Modellfunktion `output_Handler` aufgerufen werden. Allerdings müssen zu diesem Zeitpunkt schon mögliche Parameteranpassungen vollzogen sein. Die Parameteranpassungen finden während der Modellphase *Configured* statt [ESO05b]. Der Aufruf von `output_Handler` sollte also in der anschließenden Phase erfolgen, d.h. in der Methode `Connect`:

```
void ::battery::battery::Connect(::Smp::ISimulator* simulator)
        throw (::Smp::IModel::InvalidModelState)
{
    // Call output handler and let RTW perform a time 0 simulation step.
    battery_output_Handler();
    assert(simulator);
    ::Smp::Mdk::Management::ManagedModel::Connect(simulator);
}
```

4.4.3. Automatisierung und Buildprozess

Folgende Teilschritte zur Automatisierung der gesamten Einbettungsprozedur aus Abbildung 5 werden im Ant-Skript ausgeführt.

Die Codeerzeugung mit dem Real-Time Workshop wird über ein MATLAB-Skript gesteuert. Über das MATLAB-Skript lässt sich das Modell laden, die Simulation konfigurieren und die Codegenerierung auslösen. Es wird als Argument auf der Kommandozeile an MATLAB übergeben. Das Skript wird aus einem Template erzeugt, das mit dem jeweiligen Modellnamen und ggf. vorhandenen Konfigurationsdateien gebunden wird. Im Anhang A.6 ist das Template einsehbar. Darin wird das Modell geladen und verschiedene Standardeinstellungen gesetzt.

Die Schrittweite wird auf 0,005 Sekunden gesetzt. Sie kann später in der fertigen Komponente über einen Parameter geändert werden. Die Simulationsdauer wird standardmäßig auf 40 Sekunden gesetzt. Die Wirkung dieses Wertes wird aber durch die SMP2-Einbettung von MOSAIC außer Kraft gesetzt, so dass die Simulation beliebig lange läuft. Die anderen Einstellungen sind von MOSAIC vorgegeben [LM06]. Anschließend stößt das Skript die Code-Generierung des Real-Time Workshops an, schließt das Modell und beendet MATLAB.

MOSAIC kann über einen Kommandozeileninterpreter aufgerufen werden [LM06]. Der

erforderliche RTW-Code eines Modells muss in einem auf das Modell lautenden Verzeichnis im Arbeitsverzeichnis vorliegen. Als Argumente werden der Modellname und der Ablageort für den von MOSAIC erzeugten Quelltext angegeben, sowie die Festlegung auf SIMSAT als Zielsimulator und SMP2. Daraufhin generiert MOSAIC Quelltexte und SMP2-Dokumente für das SMP2-Modell. Auf den Quelltext werden dann die in den vorigen Abschnitten beschriebenen Modifikationen angewendet.

Der Quelltext soll wegen SIMSAT unter Linux kompiliert werden. Da MOSAIC nur Makefiles für Windows erzeugt, müssen noch Makefiles für die Kompilierung unter Linux hinzugefügt werden. Die Makefiles wurden aus Vorlagen in SIMSAT 4 erstellt. Darin werden die Quelltext-Dateien, bestehend aus RTW-Code und modifiziertem MOSAIC-Code, kompiliert und zusammen mit der RTW-Laufzeitbibliothek und der mit SIMSAT 4 bereitgestellten SMP2-MDK-Bibliothek zu einer dynamischen Bibliothek verbunden. Dies ist die SMP2-ausführbare Komponente. Die Dateien für die RTW-Laufzeitbibliothek wurde in Anlehnung an die bei [LM06] beschriebene Sammlung ausgewählt.

4.5. Modellbenutzung

Nach Erzeugung und Kompilierung liegt das Simulink-Modell als ausführbare SMP2-Komponente vor. Nun ist noch von Interesse, wie das SMP2-Modell konfiguriert werden kann und welche Einschränkungen es bei der Benutzung gibt.

4.5.1. Parametrisierung

Bei Generierung des RTW-Simulationscodes werden alle Parameter des Simulink-Modells ausgewertet und im Code definiert. Die Werte der Parameter werden ebenfalls aus dem Simulink-Modell entnommen und im Code fest initialisiert. Eingebettet in ein SMP2-Modell gibt es zunächst keine Möglichkeit, die Parameter neu zu definieren.

In einem Assembly-Dokument werden SMP2-Modelle instanziert. Dort gibt es auch die Möglichkeit, auf SMP2-Ebene definierte Field-Variablen jeder Modell-Instanz neu zu belegen. Damit diese Belegung in die Berechnung eingeht, muss sie auf die Variablen im eingebetteten RTW-Code übertragen werden. Dies geschieht mit der gegenwärtigen Einbettung automatisch. Beim Laden der Simulation in SIMSAT wird das Assembly gelesen, entsprechend Instanzen von SMP2-Modell-Klassen angelegt und in der SMP2-Modellphase `Configure` mit den angegebenen Parameterwerten initialisiert. Da die von MOSAIC deklarierten SMP2-Field-Variablen Referenzen auf die RTW-Variablen sind, werden mit der Assembly-Parametrisierung die Werte der RTW-Parameter überschrieben.

Parameter sind Konstanten-Blöcke, maskierte Parameter und andere blockinterne Parameter, z.B. der Startzustand eines Integrators. Alle können im Assembly neu angegeben werden, egal ob sie in Simulink mit Variablen aus dem MATLAB-Workspace oder mit konstanten Werten belegt waren. Wird ein Parameter im Assembly ausgelassen, wird stattdessen der entsprechende, im Simulink-Modell definierte Wert genommen. Das er-

möglicht es, sowohl Standardparameter eines SMP2-Modells zu definieren (im Simulink-Modell), als auch einzelne Instanzen des SMP2-Modells individuell zu parametrisieren (im Assembly).

4.5.2. Ausführung im Schedule

Durch die beschriebenen Anpassungen der Einbettung sind die SMP2-Modelle noch vor dem ersten Simulationsschritt initialisiert. In Simulink würde daraufhin die Berechnung des ersten Simulationsschrittes stattfinden. Bei einer Schrittweite von 0,005 s findet die erste Berechnung zur Simulationszeit 0,005 statt.

Im Schedule-Dokument lässt sich die Ausführung der Entrypoint-Funktionen einer SMP2-Modell-Instanz takten. Ein Schedule definiert ein Simulationszeitereignis, das eine Schrittweite und einen Startzeitpunkt hat. Analog zur Ausführung in Simulink sollte der Startzeitpunkt auf (0 + Schrittweite) gewählt werden. Für eine Simulation der Schrittweite 0,005 sollte der Startzeitpunkt des Simulationsereignisses deshalb bei 0,005 s liegen.

In jedem durch das Ereignis ausgelösten Schritt sollten die Entrypoint-Funktionen in der Reihenfolge `input_Handler`, `output_Handler` und `update_Handler` ausgeführt werden.

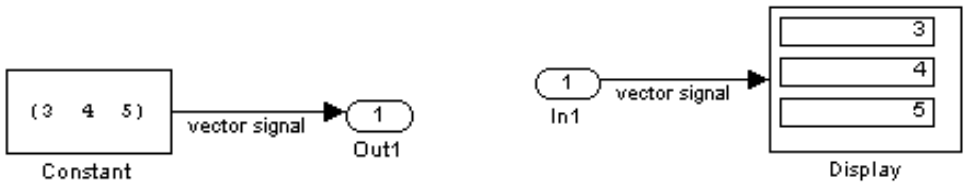
4.5.3. Verwendung von Vektorsignalen

Zwei Simulink-Teilmodelle seien durch ein Vektorsignal verbunden (Abbildung 9). Wird jedes dieser Teilmodelle in jeweils ein SMP2-Modell eingebettet, werden ihre Ein- und Ausgänge unterschiedlich typisiert. Die eigentlich zu verknüpfenden Ports sind dann inkompatibel. Um dieses Problem zu umgehen, muss der Modellierer in Simulink (Vektor-)Typ und die Dimension jedes Ports festlegen, an dem ein Vektorsignal anliegt.

Ein- oder Ausgangs-Ports ("Out1" in Abbildung 9a, bzw. "In1" in Abbildung 9b) von Simulink-Modellen, an denen nicht erkennbar oder festgelegt ist, ob sie Skalar-, Vektor- oder Matrix-Signale repräsentieren, werden bei der Codegenerierung mit dem Real-Time Workshop standardmäßig auf Skalarsignale vom Typ `double` festgelegt. Dies betrifft in jedem Fall Eingangs-Ports. Für Ausgangs-Ports findet Simulink den Typ korrekt heraus, wenn der Vektor im Modell erstellt und an den Ausgangs-Port geleitet wird.

Legt der Modellierer den Typ eines Ports explizit fest, wird damit auch ein fester Wert für die Dimension des Vektors vorgegeben. Es können keine SMP2-Komponenten erstellt werden, die an ihren Ein- oder Ausgängen Vektorgrößen mit beliebiger Dimension zulassen. Die Dimension ist mit der Komponente festgelegt.

Matrizen werden vom RTW in Vektoren konvertiert. Ports können auch nicht explizit auf Matrizen typisiert werden. Matrix-Signale unterliegen damit den gleichen Beschränkungen wie Vektorsignale. Zusätzlich entsteht aber ein Mehraufwand, da das vektorisierte Signal im Modell wieder korrekt als Matrix interpretiert werden muss. Das ist Aufgabe des Modellierers.



(a) Simulink-Block, der ein Vektorsignal bereitstellt.

(b) Simulink-Block, der ein Vektorsignal empfängt.

Abbildung 9: Simulink-Modell, das den Transport von Vektorsignalen demonstriert.

4.5.4. Auftrennung von Simulink-Modellen

Wenn ein großes Simulink-Modell in mehrere, separate Teilmodelle aufgeteilt, und jedes Teilmodell mit dem Real-Time Workshop übersetzt wird, dann wird für jedes Teilmodell der Lösungsalgorithmus neu ausgewählt. Normalerweise wird der eingestellte Löser verwendet. Enthält aber ein Modell keine kontinuierlichen Zustände, wählt der Real-Time Workshop automatisch den diskreten Löser. Wenn von diesen Teilmodellen manche kontinuierliche Zustände enthalten, manche aber nicht, werden die kontinuierlichen Teilmodelle mit dem eingestellten (kontinuerlichen) Löser berechnet, die nicht-kontinuierlichen Teilmodelle jedoch mit dem diskreten Löser.

Dies mag vielleicht den Modellierer überraschen, da in Simulink für alle Teilmodelle derselbe Löser verwendet wird. Diskrete Teilmodelle können aber ihre Ausgangsvariablen in jedem Zeitschritt selbst bereitstellen (ohne die Hilfe eines Lösungsalgorithmus zu benötigen). Die Änderung mancher Löser könnte dann ein Problem darstellen, wenn sich damit ihre Schrittweitenwahl ändert. Dann müssten zur Simulation die Teilmodelle anders ausgeführt werden. Solange alle Teilmodelle jedoch die gleiche feste Schrittweite benutzen, ist die Änderung des Lösers nicht relevant.

5. Einbettung von Modelica-Modellen

Die Einbettung eines Modelica-Modells in ein SMP2-Modell ist neu. Bisher wurde kein Werkzeug für diesen Zweck entwickelt. Aufbauend auf Ergebnissen der vorangegangenen Studienarbeit [Rö08] wurde in dieser Diplomarbeit eine Einbettung entwickelt. Sie wird in diesem Kapitel beschrieben. Eine Übersicht über die Teilschritte der Einbettung gibt Abbildung 10.

Das Programm OpenModelica wird benutzt, um Simulationscode aus einem Modelica-Modell zu generieren. Zuvor muss das Modelica-Modell jedoch erweitert werden, um es auf die Kopplung vorzubereiten. Da diese Erweiterung vor der Codegenerierung stattfindet, wird sie im folgenden auch zuerst beschrieben. Danach folgt die Vorstellung des Modellcompilers OpenModelica.

Dann wird erklärt, welche Anforderungen die Einbettung erfüllen muss, damit das resultierende SMP2-Modell die in Kapitel 3 beschriebene Schnittstelle einhält und damit allgemein koppelbar ist. Es schließt sich die Beschreibung an, wie die Einbettung umgesetzt wurde. Schließlich wird noch die Nutzung so erstellter SMP2-Modelle beleuchtet.

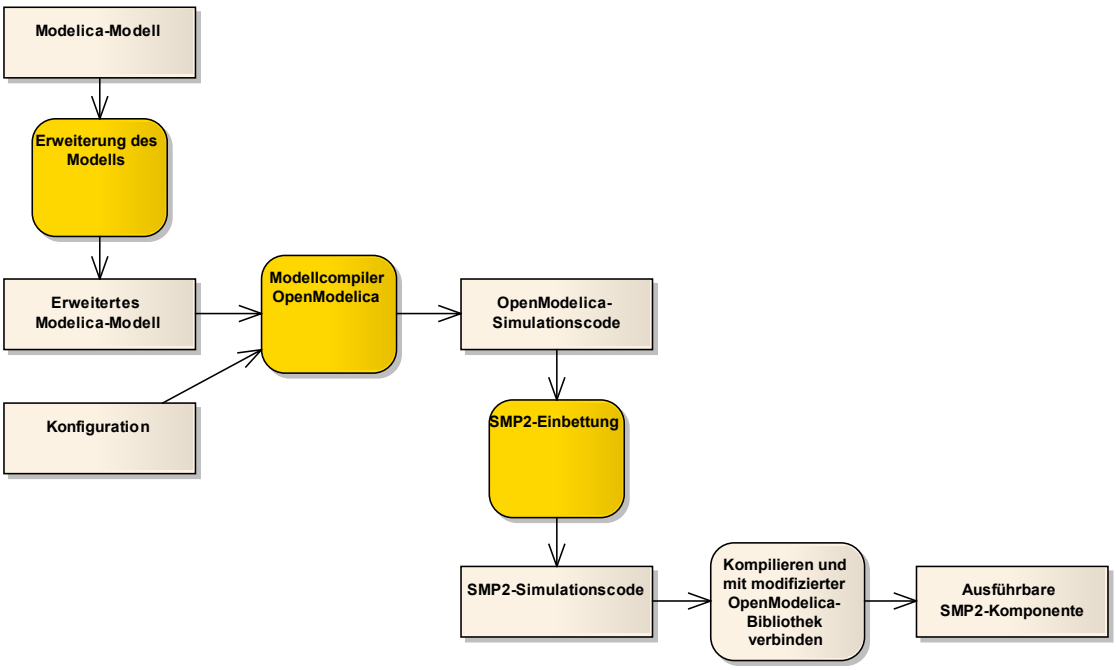


Abbildung 10: Übersicht über die speziellen Schritte zur SMP2-Einbettung eines Modelica-Modells.

5.1. Erweiterung des Modells

Kopplung von SMP2-Modellen bedeutet unter anderem, dass ein SMP2-Modell Berechnungen mit aktualisierten Werten seiner Eingangsvariablen vornimmt und damit seine Ausgangswerte aktualisiert. Die Ausgangswerte werden dann an Eingangsvariablen anderer Modelle übertragen, die wiederum Berechnungen durchführen.

Zugriff auf die Ausgangsvariablen kann ohne große Probleme auf Codeebene erfolgen und wird im Abschnitt 5.3 behandelt. Um die Werte aktualisierter SMP2-Eingangsvariablen aber in den eingebetteten Simulationscode zu übernehmen, wären sowohl Eingriffe in die Laufzeitumgebung des Modellcompilers als auch in seinen Mechanismus zur Erstellung von Simulationscode notwendig. Einfacher und zuverlässiger lässt sich die Aktualisierung über einen Modellmechanismus von Modelica selbst erreichen: Modelica sieht externe Funktionen vor, mit denen Code außerhalb des Modelica-Modells aufgerufen werden kann [Fri04]. Der Code kann in den Sprachen C, Fortran 77, der C-Untermenge der Sprache C++ oder der Fortran–77-Untermenge der Sprache Fortran 90 vorliegen.

Das Modelica-Modell kann so erweitert werden, dass für jede Auswertung einer Eingangsvariablen im Modelica-Modell zuerst eine externe Funktion aufgerufen wird, die den aktuellen Wert bereitstellt. Der Rückgabewert dieser externen Funktion wird aus dem SMP2-Modell heraus gesetzt.

Zur Erweiterung wird das ursprüngliche Modelica-Modell in ein neues Modelica-Modell eingebettet. In diesem Erweiterungsmodell wird für jede Eingangsvariable des Ursprungsmodells eine externe Funktion deklariert, und der Wert der Eingangsvariablen zu jeder Zeit als Ergebnis der externen Funktion definiert. Als Beispiel die Erweiterung eines Modelica-Modells *battery*:

```
model battery_wrapper
  battery sourcemodel;
equation
  sourcemodel.u = u_from_external(1);
end battery_wrapper;
```

Die Übergabe des konstanten Arguments "1" umgeht einen Fehler in OpenModelica 1.4.4. Diese Version unterstützt keine externen Funktionen ohne Argumente. In aktuellen Versionen von OpenModelica ist dieser Fehler behoben. Im erweiterten Modelica-Modell wird die externe Funktion `battery_u` in der C++-Bibliothek `libbattery_ext_input.o` referenziert:

```
function u_from_external
  input Real x;
  output Real y;
  external "C" y=battery_u(x)
    annotation(Library="libbattery_ext_input.o",
```

```
                Include="#include \"battery_ext_input.h\"");
end u_from_external;
```

Für das Erweiterungsmodell und damit auch das Ursprungsmodell kann mit einem Modellcompiler für Modelica-Modelle Simulationscode erzeugt werden. Wird der Simulationscode kompiliert und mit dem kompilierten Code der externen Funktionen verbunden, resultiert eine vollständige ausführbare Datei. Darin wird bei jeder Auswertung der Eingangsvariablen die entsprechende externe Funktion aufgerufen.

5.2. Modellcompiler OpenModelica

Zur Codegenerierung aus Modelica-Modellen wird *OpenModelica*[3] [FAP+06] genutzt. OpenModelica ist unter einer Open-Source-Lizenz[4] frei nutzbar. Es wird als Forschungsprojekt der Universität Linköping entwickelt. Für die Version 1.5.0 steht ein Release Candidate bereit. Damit hat es eine zur Verwendung geeignete Stabilität erreicht. Dennoch werden nicht alle Features der aktuellen Modelica-Spezifikation 3.1 unterstützt, insbesondere die Bibliothekspakete MultiBody und Media nicht. Für die Einbettung wird OpenModelica-Code verwendet, der von der früheren Version 1.4.4, Revision 3624 stammt. Diese Version war bei Arbeitsanfang aktuell. Soll eine neue Version unterstützt werden, müssten auch der verwendete Code aktualisiert werden.

Zu OpenModelica gibt es wenige alternative Modelica-Compiler. Die Simulationsumgebung *Dymola*[5] kann als momentan umfassendster Modellcompiler für Modelica-Modelle gesehen werden. Eine Lizenz ist jedoch kostenpflichtig. Außerdem ist die Laufzeitbibliothek nicht im Code verfügbar und die Struktur im Modellcode durch Verwendung von Makroexpansionen verschleiert. Der Code lässt sich deshalb nicht geeignet modifizieren und auch nicht in SMP2 einbetten. Mit *JModelica*[6] wird ein Modellcompiler mit Schwerpunkt auf Modelloptimierung entwickelt. Die Software Scilab[7] enthält den Modellcompiler *Modelicac*. Die beiden letztgenannten Compiler unterstützen bisher aber nur einen kleinen Teil der Modelica-Spezifikation.

OpenModelica erzeugt standardmäßig Simulationscode der Sprache C. So wird eine OpenModelica-Simulation erzeugt. Zur Simulation ruft OpenModelica den generierten Code selbst auf. Die Simulationsergebnisse sind deshalb mit denen der normalen Simulation identisch. Sie werden in einer Ergebnisdatei abgelegt.

Vor der Generierung des Simulationscodes lassen sich Simulationsparameter wie die Dauer oder die gewünschte Fehlertoleranz einstellen. Diese Parameter und die im Modelica-Modell definierten Parameter lassen sich vor Ausführung des Simulationscodes in einer eigens von OpenModelica angelegten Parameterdatei ändern. Die Datei wird

[3] http://www.ida.liu.se/~pelab/modelica/OpenModelica.html (am 13.11.2009 zuletzt eingesehen)
[4] http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/Documents/LICENSE.txt (am 13.11.2009 zuletzt eingesehen)
[5] http://www.3ds.com/products/catia/portfolio/dymola (am 13.11.2009 zuletzt eingesehen)
[6] http://www.jmodelica.org/ (am 13.11.2009 zuletzt eingesehen)
[7] http://www.scilab.org/ (am 13.11.2009 zuletzt eingesehen)

vor Simulationsbeginn eingelesen, so dass sich die Änderungen auf die Berechnungen niederschlagen.

In OpenModelica stehen zwei Lösungsverfahren für differential-algebraische Gleichungen (DAE) zur Verfügung: Ein einfaches Euler-Verfahren und DASSL (Differential Algebraic System Solver) [BCP96]. Für die Einbettung soll der Löser DASSL verwendet werden, da er die genaueren Ergebnisse errechnet. Er rechnet intern mit variablen Schrittweiten, um das System feiner berechnen zu können. Mit der Konfiguration der Simulation lässt sich aber auch eine Schrittweite angeben, so dass DASSL zusätzlich zu den variabel ermittelten Zeitpunkten auch zu wiederkehrenden festen Zeitpunkten Werte berechnet.

5.3. SMP2-Einbettung

Um ein Modelica-Modell in ein SMP2-Modell einzubetten, muss einerseits die Ausführung der einzelnen Simulationsschritte vom Simulationscode auf SMP2-Entrypoints übertragen werden. Andererseits müssen Variablen des Simulationscodes im SMP2-Modell bekannt gemacht werden. Wie diese beiden Schritte für OpenModelica-Simulationscode angewendet werden, erklären die folgenden Abschnitte.

5.3.1. Übertragung der Ausführung

Die Ausführungssteuerung im Quelltext der OpenModelica-Laufzeitbibliothek ist ein großer Anweisungsblock (Datei `solver_dasrt.cpp`). Er besteht aus Anweisungen zur Initialisierung, zur Berechnung und zum Aufräumen. Die Berechnung findet in einer Schleife statt, die für jeden Simulationsschritt wiederholt wird und zum vorgegebenen Simulationsende abbricht.

Um die Ausführungsteuerung auf ein SMP2-Modell zu übertragen, wird wie in [Rö08] verfahren. Der Quelltext der OpenModelica-Laufzeitbibliothek wird in drei Teile aufgetrennt. Je eine Methode kapselt die Initialisierung, die Berechnung eines Simulationsschrittes und das Aufräumen nach der Berechnung. Die Methoden heißen entsprechend `InitialiseHandler`, `StepHandler` und `FinaliseHandler`.

Im SMP2-Modell werden diese drei Methoden aufgerufen. `InitialiseHandler` und `FinaliseHandler` werden einmalig aufgerufen. Die Methode `StepHandler` wird aus einem Entrypoint heraus aufgerufen. Das SMP2-Modell definiert einen Entrypoint namens `Modelica_battery_update` (als Beispiel für ein Modell namens *battery*). Dieser Update-Entrypoint führt die zur Berechnung des nächsten Zeitschrittes notwendigen Anweisungen aus. Unter anderem wird dafür die Methode `StepHandler` aufgerufen.

Damit bleibt auch das bekannte Phänomen bestehen, dass der Simulationscode Teilschritte der konfigurierten Schrittweite berechnet. Intern berechnet DASSL fünf Teilschritte. Durch die direkte Extraktion, berechnet jeder Aufruf von `StepHandler` nur einen Teilschritt. `StepHandler` wird also fünffach so häufig wie eigentlich geplant aufgerufen. Es resultiert eine Abweichung der Zeitskala im Simulationscode und der Skala im SMP2-Simulator von einem Fünftel. Dies ist ein Problem, wenn so erstellte SMP2-Modelle mit

SMP2-Modellen gekoppelt werden, die mit normaler Schrittweite laufen, z.B. ein eingebettetes Simulink-Modell. Um die Abweichung auszugleichen, wird im SMP2-Schedule einfach ein zweites Ereignis definiert, das mit fünffacher Geschwindigkeit ausgelöst wird, also alle 0,001 s wenn die Schrittweite eigentlich 0,005 s beträgt. Werden eingebettete Simulink-Modelle mit der Schrittweite 0,005 s und eingebettete Modelica-Modelle mit der Schrittweite 0,001 s ausgeführt, dann laufen alle Modelle zeitsynchron.

Andere Lösungen der Abweichung der Zeitskalen sind in Betracht zu ziehen, da für Zeitschritte, auf die beide Ereignisse fallen (z.B. bei 0,01 s), ihre Ausführungsreihenfolge nicht definiert ist. Eine andere Lösung ist die Verwendung eines Modelica-Mechanismus und wird im Ausblick beschrieben.

Einige Abschnitte im Quelltext der Laufzeitbibliothek wurden weggelassen. So die Behandlung von Kommandozeilenparametern, die ja nicht mehr nötig ist. Ebenso wird keine Ergebnisdatei mehr erzeugt.

5.3.2. Übertragung der Variablen

Ein- und Ausgangsvariablen des Modelica-Modells sollen im SMP2-Modell repräsentiert sein, damit sie während der Simulation eingesehen, bzw. vor der Simulation noch angepasst werden können. Nach der Codegenerierung sind alle diese Variablen in global deklarierten Arrays im OpenModelica-Modellcode abgelegt. Ein Array speichert alle Namen von Variablen, ein anderes ihre Werte. Um auf den Wert einer Variablen zugreifen zu können, muss anhand ihres Namens der Index in den Arrays ermittelt werden.

Für jede Ein- und Ausgangsvariable wird eine äquivalente SMP2-Field-Variable angelegt. Ähnlich dem bei der Einbettung von Simulink-Modellen verwendeten Ansatz werden diese Variablen mit den Variablen im Simulationscode abgeglichen. Dazu wird im Update-Entrypoint vor der Berechnung des nächsten Simulationsschrittes für jede Eingangsvariable der Wert ihrer SMP2-Field-Variable in den Simulationscode übertragen. Nach der Berechnung werden die Werte aller Ausgangsvariablen im Simulationscode in die entsprechenden SMP2-Field-Variablen übertragen. Auf SMP2-Ebene werden dann bei der Simulation die Werte zwischen den Field-Variablen verschiedener Modelle mit SMP2-Mechanismen geeignet synchronisiert.

5.3.3. Logging

Wie bei der Einbettung von Simulink-Modellen werden auch für eingebettete Modelica-Modelle die Werte der Ausgangsvariablen protokolliert. Dies kann ebenfalls auf alle anderen internen Variablen und Parameter erweitert werden.

5.4. Umsetzung

Der folgende Abschnitt beschreibt, wie die eben beschriebenen Einbettungsschritte umgesetzt wurden. Es werden die externen Funktionen zur Aktualsierung der Eingangsva-

riablen beschrieben. Dann wird beleuchtet, wie die Variablen im OpenModelica-Simulationscode abgelegt sind und wie Zugriff darauf erlangt werden kann. Anschließend wird erklärt, wie der Code des SMP2-Modells erzeugt wird. Danach wird beschrieben, wie der SMP2-Modellcode so angepasst wird, dass Ein- und Ausgangsvariablen im SMP2-Modell und im Simulationscode synchron bleiben. Zum Schluss wird darauf eingegangen, wie die Folge der Einbettungsschritte ausgelöst wird.

5.4.1. Erweiterung des Modelica-Modells

Bei der Erstellung des Erweiterungsmodells und bei der Übertragung der Variablen im SMP2-Modell müssen die Ein- und Ausgangsvariablen des Modelica-Modells bekannt sein. Sie werden direkt aus dem Modelica-Quelltext mit regulären Ausdrücken gelesen. Alle Variablendeklarationen der Modelica-Typen RealInput und RealOutput werden gesucht und die Namen gespeichert. Andere Verknüpfungstypen wie IntegerInput oder BooleanOutput werden vorerst nicht unterstützt, können aber leicht hinzugefügt werden. Im Simulationscode sind sogar keine Änderungen zu erwarten, da OpenModelica alle Modelica-Typen in den C++-Typ double konvertiert und deshalb im Code nur mit double umgegangen werden muss.

Das Erweiterungs-Modell wird mit einem Template erzeugt. Für jede ermittelte Eingangsvariable wird im Erweiterungsmodell eine externe Funktion referenziert. Die externen Funktionen werden in einer C++-Datei implementiert. Dort wird für jeden Eingang eine Funktion und eine globale Variable definiert. Die externe Funktion gibt einfach den Wert der globalen Variablen zurück. Die globale Variable kann dann sehr einfach aus dem SMP2-Modell heraus geändert werden. Ein Beispiel für den einzigen Eingang `u` eines Modells *battery.*:

```
#ifdef __cplusplus
extern "C" {
#endif
    double g_battery_u = 0;
    double battery_u(double in)
    {
        return g_battery_u;
    }
#ifdef __cplusplus
}
#endif
```

Das Funktionsargument “in” wird wieder wegen genanntem OpenModelica-Fehler bei leeren Argumentlisten benötigt. Der Name der globalen Variablen und der Name der externen Funktion leiten sich beide nach einem festen Schema vom Namen des entsprechenden Eingangs ab. Für das Modell *battery* und den Eingang `u` heißt die globale Variable `g_battery_u` und die externe Funktion `battery_u`. Mit der durchgehenden Verwendung

dieses Namensschemas ist im Code des SMP2-Modells jederzeit klar, welche globale Variable geändert werden muss, um eine bestimmte Eingangsvariable zu setzen.

5.4.2. Zugriff auf Variablen im Simulationscode

Nach Anwendung von OpenModelica ist der Simulationscode des Modells vorhanden. Im Modelica-Modell verwendete Bezeichner für Variablen und Parameter finden sich im Code unter gleichem Namen wieder. Wird zum Beispiel das in Kapitel 6.3.2 verwendete Batteriemodell mit dem in Kapitel 5.1 vorgestellten Modell erweitert, finden sich im Simulationscode die Namen der Variablen und Parameter in folgenden Arrays:

```
char* algvars_names[12]={"sourcemodel.y", "sourcemodel.integrator.u",
                         "sourcemodel.max1.u1", "sourcemodel.gain.u",
                         "sourcemodel.u", "sourcemodel.max.u1",
                         "sourcemodel.max.u2", "sourcemodel.max.y",
                         "sourcemodel.max1.u2", "sourcemodel.max1.y",
                         "sourcemodel.gain.y", "sourcemodel.const.y"};
char* param_names[4]={"sourcemodel.integrator.k",
                      "sourcemodel.integrator.y_start",
                      "sourcemodel.gain.k", "sourcemodel.const.k"};
```

Die Werte der jeweiligen Variablen sind in ähnlichen Arrays gespeichert, die in einer Instanz der zur Aufnahme aller Variablen verwendeten Datenstruktur `DATA` abgelegt sind:

```
static DATA* localData = 0;

typedef struct sim_DATA {
  double* states; //x STATES
  double* statesDerivatives; //xd DERIVATIVES
  double* algebraics; //y ALGVARS
  double* parameters; //p; PARAMETERS
  [...]
  double timeValue; //the time for the simulation
  //used in some generated function
  // this is not changed by initializeDataStruc
  double lastEmittedTime; // The last time value that has been emitted.
  int forceEmit; // when != 0 force emit,
                 // set e.g. by newTime for equidistant output signal.
} DATA;
```

Name und Wert einer Variablen sind also in verschiedenen Arrays gespeichert. Um auf eine bestimmte Variable zugreifen zu können, muss ihr Index bekannt sein. Der Index wird ermittelt, indem die Definition des Namens-Arrays im Quelltext gesucht wird, und

die Liste der Namen durchgegangen wird. Die Namen der relevanten Ein- und Ausgangsvariablen sind aus dem Modelica-Modell bekannt und werden ihren Indizes zugeordnet.

5.4.3. SMP2-Modell erzeugen

Ein SMP2-Modell wird erzeugt, das je eine Field-Variable für jeden Ein- und Ausgang hat und einen Entrypoint, den Update-Entrypoint. Diese Spezifikation wird zuerst als SMP2-Catalogue erzeugt. Aus einem Catalogue-Template wird durch Binden des Modellnamens, des Entrypoint-Namens und der Namen der Ein- und Ausgänge ein richtiges Catalogue-Dokument generiert.

Anschließend wird der erzeugte Catalogue validiert. Dazu wird eine SIMSAT-Funktion benutzt, die auf der Kommandozeile die Überprüfung mit der Catalogue-XSD-Spezifikation durchführt. Ebenfalls auf der Kommandozeile stellt SIMSAT ein C++-Mapping zur Verfügung [ESO05a]. Mit dem C++-Mapping wird für die SMP2-Modell-Beschreibung des Catalogues C++-Code generiert. Ein Beispiel dafür findet sich im Anhang A.2. Der C++-Code des SMP2-Modells beinhaltet zunächst nur die für die Kommunikation mit dem SMP2-Simulator nötigen Funktionen und Anweisungen. Kompiliert ließe er sich schon in SIMSAT laden, aber es fehlt noch der Aufruf der Berechnungen, die Variablensynchronisation und die Aufrufe für Initialisierung und Aufräumen der Laufzeitbibliothek.

5.4.4. SMP2-Modell vervollständigen

Um das SMP2-Modell funktional zu vervollständigen, werden wieder mit regulären Ausdrücken an verschiedenen Stellen Ergänzungen gemacht. Die extrahierte Laufzeitbibliotheksfunktionen `InitialiseHandler`, `StepHandler` und `FinaliseHandler` werden deklariert. `InitialiseHandler` und `FinaliseHandler` werden dann in der Configure-Methode bzw. im Modell-Destruktor aufgerufen, `StepHandler` in der Entrypoint-Funktion. Durch den Aufruf der Funktion `InitialiseHandler` werden die Werte aus der OpenModelica-Parameterdatei geladen. Dazu muss die Parameterdatei im aktuellen Pfad liegen.

Die Variablen und der Entrypoint sind bereits bei der Codegenerierung mit dem C++-Mapping deklariert worden, da sie im Catalogue angegeben sind. In der Implementierung der Entrypoint-Funktion werden nun vor dem Aufruf von `StepHandler` die Eingangsvariablen aktualisiert und nach dem Aufruf die Ausgangsvariablen. Ein Beispiel für ein Modell *battery* mit einem Eingang und einem Ausgang:

```
// Handler for Entry Point: Modelica_battery_update
void battery::_Modelica_battery_update()
{
    // Update input signals
    g_battery_u = u;
```

```
    // Call OpenModelica runtime step code
    StepHandler();

    // Update field variable by explicitly copying its value
    y = (Smp::Float64) globalData->algebraics[0];
    [...]
}
```

Die angelegten SMP2-Field-Variablen und der Update-Entrypoint müssen dem Simulator bekannt gemacht werden. Das wird in der Publish-Phase des SMP2-Modells gemacht:

```
void battery::Publish( ::Smp::IPublication* receiver ) throw ( [...] )
{
    ::Smp::Mdk::Management::ManagedModel::Publish( receiver );
    receiver->PublishField( "u", "", &u, true, true, false, true );
    receiver->PublishField( "y", "", &y, true, true, true, false );
    receiver->PublishOperation( "Modelica_battery_update",
                                "Timestep update of all state variables",
                                ::Smp::Publication::Uuid_Void );
}
```

5.4.5. Automatisierung und Buildprozess

Die in Abbildung 10 ersichtlichen Teilschritte *Erweiterung des Modells*, *Modellcompiler OpenModelica* und *SMP2-Einbettung* werden im Ant-Skript wie folgt abgearbeitet.

Die Erweiterung des Modells wie sie in Abschnitt 5.4.1 beschrieben wird, übernimmt ein Groovy-Skript. Über ein OpenModelica-Skript werden dann die Modelica-Bibliothek, das Ursprungsmodell und das Erweiterungsmodell geladen. Anschließend wird der Quelltext mit der C-Implementierung der externen Funktionen kompiliert. Zum Schluss wird schließlich der Modellcompiler angewiesen, den Modellcode zu erzeugen. Als Simulationsparameter wird eine Simulationsdauer von 40 s und eine Intervallanzahl von 8000 gesetzt. Das entspricht einer Schrittweite von 0,005 s. Die Simulationsparameter können später in der Parameterdatei angepasst werden. Das Lösungsverfahren ist mit der Modifikation der Laufzeitbibliothek auf DASSL festgelegt worden. Das OpenModelica-Skript wird aus einem Template erzeugt:

```
loadModel(Modelica);
loadFile("battery.mo");
loadFile("battery_wrapper.mo");
cd("code/battery");
system("gcc -c -o libbattery_ext_input.o battery_ext_input.c");
buildModel(battery_wrapper, startTime=0, stopTime=40,
           numberOfIntervals=8000);
```

Für die Erzeugung des SMP2-Modellcodes wird entweder eine vollständige SIMSAT-Installation benutzt, oder eine auf das C++-Mapping reduzierte Version. Für das Projekt *Virtueller Satellit* wurde eine SIMSAT-Installation auf das C++-Mapping reduziert. Sie ist dadurch mit 50 MB wesentlich kleiner als eine volle Installation mit 300 MB. Das C++-Mapping wird über die Kommandozeile aufgerufen.

Damit das SMP2-Modell überhaupt erst vom SMP2-Simulator geladen werden kann, ist noch die Definition einer Klasse nötig, welche die Instanz des Modells an den Simulator übergibt. Diese im SMP2-Kontext Package-Code genannten Dateien können einfach als Templates angelegt werden, in denen der Modellname eingesetzt wird. Ursprünglich wurden sie ebenfalls mit dem C++-Mapping in SIMSAT erzeugt. Bis auf den Modellnamen sind die Dateien jedoch für jedes Einzelmodell identisch. Das gleiche gilt für die Makefiles, um den gesamten Code zu kompilieren. Sie entstammen ebenfalls SIMSAT. In den Makefiles mussten lediglich noch Eintragungen zum Einbinden der OpenModelica-Laufzeitbibliothek gemacht werden. Dann erzeugen sie auf einem Linux-System ein ausführbares SMP2-Modell.

Die Dateien der OpenModelica-Laufzeitbibliothek sind in den Buildprozess integriert. Die Laufzeitbibliothek wird per Umgebungsvariable `OMRL_INSTALL` eingebunden und bei Bedarf kompiliert. Alle benötigten Abhängigkeiten beim Kompilieren sind so erfüllt.

5.5. Modellbenutzung

Das fertig kompilierte SMP2-Modell lässt sich nun in einer SMP2-Simulation verwenden. Da die in Kapitel 3 vorgestellte Schnittstelle eingehalten wird, lässt sich das Modell parallel zu anderen über einen Entrypoint regelmäßig ausführen. Ein- und Ausgänge von Modellen lassen sich über Field-Links synchronisieren. Beim Einsatz eines eingebetteten Modelica-Modells ist weiterhin noch wichtig, wie unmittelbar vor dem Einsatz die Parameter des Modells geändert werden können, wie das Modell im Schedule zur Ausführung zu bringen ist und inwiefern Vektoren als Signale verwendet werden können.

5.5.1. Parametrisierung

Modellparameter und Simulationsparameter können momentan nur über die OpenModelica-Parameterdatei geändert werden. Für das Projekt *Virtueller Satellit* ist es wünschenswert, dass auch über das zur Simulation verwendete Assembly-Dokument Parameter gesetzt werden können. Momentan sind nur die Ein- und Ausgangsvariablen auf SMP2-Ebene definiert, aber keine Parameter. Deshalb können auch keine Parameter im Assembly parametrisiert werden. Die Werte der Parameterdatei werden in der SMP2-Modellphase *Configure* geladen, die Werte des Assemblys nach der Modellphase *Configure* und vor der Phase *Connect*. Die Assembly-Werte würden also gleiche Einträge aus der Parameterdatei überschreiben, was schon das gewünschte Verhalten darstellt, wenn die Parameter im SMP2-Modell bekannt gemacht würden.

5.5.2. Ausführung im Schedule

Für die korrekte Ausführung ist es erforderlich, dass der Entrypoint eines eingebetteten Modelica-Modells mit einem Fünftel der für die Simulation vorgesehenen Schrittweite ausgeführt wird. Im Schedule könnten dafür zwei Simulationszeitereignisse definiert werden. Eines, das Ereignisse mit der beabsichtigten Schrittweite beschreibt. Nach diesem Ereignis würden dann z.B. eingebettete Simulink-Modelle ausgeführt. Ein anderes Ereignis wird mit einem Fünftel dieser Schrittweite definiert und führt z.B. eingebettete Modelica-Modelle aus.

Für die Ausführung ist ferner nur der Aufruf genau eines Entrypoints nötig. Bei eingebetteten Simulink-Modellen waren es genau drei Entrypoints, falls das Modell Ein- und Ausgänge benutzt. Dies ist beim Erstellen eines Schedules für die gekoppelte Simulation mehrerer, unterschiedlicher Modelle zu beachten. Die Umsetzung stellt kein Problem dar, dennoch führt dies einen Unterschied in der Benutzung von eingebetteten Modelica- und Simulink-Modellen auf SMP2-Ebene ein. In Zukunft sollten die drei Simulink-Entrypoints zu einem zusammengefasst werden.

5.5.3. Verwendung von Vektorsignalen

Modelica definiert die Datenstruktur Array, mit der Vektoren und Matrizen implementiert werden können [Fri04]. Aber nicht jede Funktion ist darauf vorbereitet, sowohl Skalar-, Vektor- oder Matrixgrößen in ihren Variablen zu berücksichtigen. Beispielsweise sind wichtige mathematische Funktionen wie Sinus, Cosinus, etc. nur in skalarem Kontext definiert. Auch kennt Modelica nur eingeschränkte Vektor- oder Matrixkonzepte in Blockdiagrammen. Es lassen sich Blöcke mit Vektor- und Matrix-Ein- und -Ausgängen definieren, es lassen sich Matrixsignale auftrennen oder zusammenfassen. Aber mit Ausnahme des State-Space-Blockes können Vektor- oder Matrixsignale nicht an Standard-Modelica-Blöcke angelegt werden.

Es gibt mit Simelica [Dem03] allerdings eine Erweiterung der Modelica-Bibliothek, die die meisten der in Simulink verfügbaren Blöcke in Modelica reimplementiert. Die Blöcke dieser s.g. AdvancedBlocks library können mit beliebig dimensionierten Vektor- oder Matrixsignalen angesteuert werden. So ließen sich auch Modelica-Komponenten mit Vektorgrößen als Ein- und Ausgänge erstellen. Simelica ist lizenzpflichtig.

6. Simulation gekoppelter SMP2-Modelle

Der entwickelte Einbettungsprozess wird nun auf konkrete Modelle angewendet. Beispielmodelle werden in SMP2 simuliert und die Ergebnisse vorgestellt, interpretiert und Beschränkungen der Nutzung abgeleitet.

Zunächst werden zwei Modelle vorgestellt, mit denen die Simulation untersucht wird. Ihre geschlossene Simulation in Simulink bzw. OpenModelica liefert die für weitere Vergleiche benötigten Referenzergebnisse.

Zuerst wird das eine Beispielmodell vollständig in eine SMP2-Komponente eingebettet. Die Komponente berechnet dann eine geschlossene Simulation, so wie sie im jeweiligen Simulationswerkzeug selbst berechnet wird. Der Vergleich mit den Referenzergebnissen stellt sicher, dass Komponenten isoliert betrachtet jeweils korrekte Ergebnisse berechnen.

Dann wird dieses Modell aufgeteilt und die Teilmodelle einzeln in SMP2-Komponenten eingebettet. Die Komponenten werden in einem SMP2-Simulator gekoppelt, ausgeführt und die Ergebnisse mit der Referenzsimulation verglichen. So können Phänomene beobachtet werden, die bei einer einfachen Kopplung auftreten.

Schließlich wird das zweite, etwas komplexere Modell aufgeteilt, die Teilmodelle einzeln in SMP2-Komponenten eingebettet, gekoppelt ausgeführt und mit der Referenzsimulation verglichen. So werden Phänomene beobachtet, die bei gekoppelter Simulation dynamischer Systeme auftreten.

6.1. Beispielmodelle

Zwei Modelle werden zur Demonstration der verschiedenen Kopplungsszenarien verwendet. Das erste ist ein sehr einfaches Batterie-Modell. Es kann in zwei in Reihe geschaltete Teilmodelle getrennt werden und dient deshalb zur Untersuchung einer einfachen Kopplung.

Das zweite Modell ist ein Räuber-Beute-Modell, das durch ein nichtlineares Differentialgleichungs-System repräsentiert ist. Es kann in drei Teilmodelle getrennt werden, die voneinander abhängig sind und untereinander rückgekoppelt sind. Mit diesem Modell wird die Simulation eines rückgekoppelten Systems untersucht.

6.1.1. Batterieladestand

Das Batteriemodell berechnet den Ladestand E (elektrische Energie in Ws) einer Batterie. Die Berechnung ist von einem zeitabhängigen Verbrauch P abhängig (abgegriffene elektrische Leistung in W). Die grundlegende Formel lautet:

$E'(t) = -P(t)$

mit $E(0) = E_0$.

Es werden noch zwei Beschränkungen eingeführt, so dass der Verbrauch sowie der

Ladestand nicht negativ werden:

$P(t) \geq 0$ und $E(t) \geq 0$.

Der Verbrauch P wird durch einen einfachen Sinus der Periode 2π mit Wertebereich $-1 \leq sin(t) \leq 1$ repräsentiert:

$P(t) = sin(t)$.

Mit der Anfangsladung $E_0 = 10Ws$ ist die Batterie nach 28,27 s entleert. Abbildung 11 zeigt den Verlauf von $E(t)$ für 40 Sekunden.

Zur Modellierung im Blockdiagramm wird die Formel mit ihren Beschränkungen umformuliert in:

$E(t) = max(E_0 + \int -max(P(t), 0)dt, 0)$.

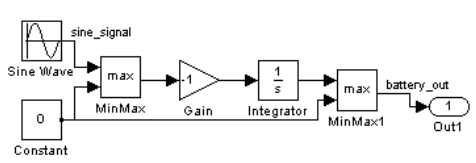
Abbildung 11a zeigt das in Simulink modellierte Blockdiagramm des Modells, Abbildung 11b das Modelica-Äquivalent in Dymola (Dymola wird hier zur Modellierung eingesetzt, OpenModelica zum Codeexport und zur Simulation). Der Quelltext des Modelica-Modells spiegelt direkt den Aufbau des Blockdiagramms wider:

```
model battery
  Modelica.Blocks.Sources.Sine sine(freqHz=0.1592);
  Modelica.Blocks.Interfaces.RealOutput y;
  Modelica.Blocks.Continuous.Integrator integrator(y_start=10);
  Modelica.Blocks.Math.Max max;
  Modelica.Blocks.Math.Max max1;
  Modelica.Blocks.Math.Gain gain(k=-1);
  Modelica.Blocks.Sources.Constant const(k=0);
equation
  connect(sine.y, max1.u1);
  connect(const.y, max1.u2);
  connect(max1.y, gain.u);
  connect(gain.y, integrator.u);
  connect(integrator.y, max.u1);
  connect(max.y, y);
  connect(const.y, max.u2);
end battery;
```

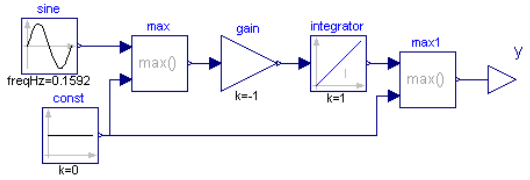
Das Modell kann jeweils hinter dem Sinus-Block leicht in zwei Teilmodelle zerlegt werden. Die beiden Teilmodelle sind dann einfach gekoppelt. Das Batteriemodell ist [Rö08] entnommen.

6.1.2. Fischfang

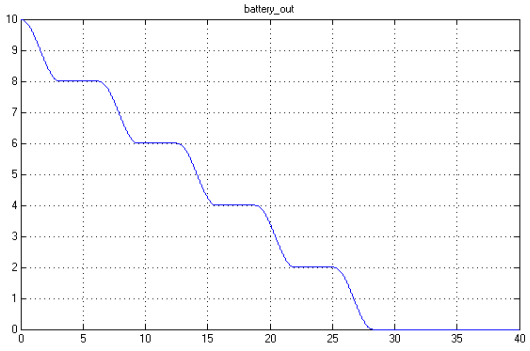
Das Fischfangmodell ist ein Räuber-Beute-System, das den Zusammenhang zwischen den zwei Zustandsvariablen Größe der Fischpopulation z_1 sowie Anzahl der Fischerboote z_2



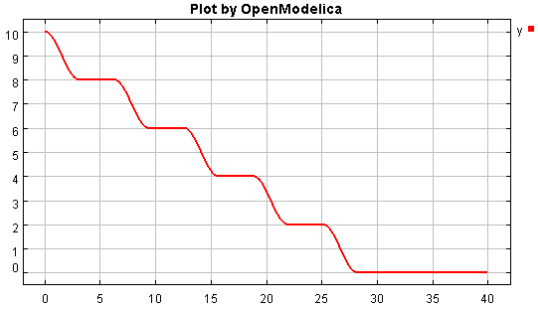
(a) Simulink-Modell



(b) Modelica-Modell als Blockdiagramm in Dymola



(c) Ergebnis in Simulink



(d) Ergebnis in OpenModelica

Abbildung 11: Blockdiagramm und berechneter Ladestand des Batterie-Modells. Y-Achse: Ladestand in Ws. X-Achse: Zeit in s. Simulationsdauer 40 s. Leerung bei 28,27s.

beschreibt [Bos04]. Es berücksichtigt dabei Zusammenhänge wie den Verfall und Neukauf von Booten, Unterhaltskosten, Verdienst durch Fischverkauf, u.v.m. Es hat mannigfaltige Parameter, wie z.B. die Fisch-Zuwachsrate a, die jährliche Fangmenge eines Bootes f, oder die Fischkapazität k des untersuchten Gewässers.

Das Modell wird durch ein nichtlineares Differentialgleichungs-System erster Ordnung beschrieben. Die Gleichungen lauten:

$z_1' = a * z_1 * (1 - z_1/k) - m$

$z_2' = c * m - e * z_2,$

wobei c und e Zusammenfassungen von Parametern sind:

$c = i * p/q$

$e = i * o/q + d$

Die kontinuierliche Funktion m gibt die Menge gefangener Fische an. Für ihre Berechnung gibt es zwei Varianten. Bei der ersten Variante hängt die Fangmenge von der Fischdichte z_1/k ab: $m = f * z_2 * z_1/k$. Berechnet sich die Fangmenge aus der Fischdichte, kann immer nur ein Teil der vorhandenen Fische gefangen werden und die Fischpopulation in der Simulation nie zusammenbrechen. Es gibt immer ein Gleichgewicht. Berechnet man die Fangmenge stattdessen über eine Fangchance der Fischer, ergibt sich:

$m = f * z_2 * chance.$

Diese Variante lässt den Einbruch der Fischpopulation zu. Zusätzlich wird eine Obergrenze für den Neuerwerb von Booten festgelegt. Wenn die Bootsanzahl über der Grenze liegt, können keine neuen Boote erworben werden. Außerdem darf der Fischbestand nicht negativ werden.

Mit einer Neuerwerbsgrenze von 25 Booten ergibt sich das Phasendiagramm in Abbildung 12. Darin sind zwei Gebiete erkennbar. Im linken Gebiet führt die Startwertekonfiguration immer zum Einbruch der Fischpopulation. Im rechten Gebiet führen alle Zustandspfade auf den Gleichgewichtspunkt von 7236 t Fisch und 25 Booten.

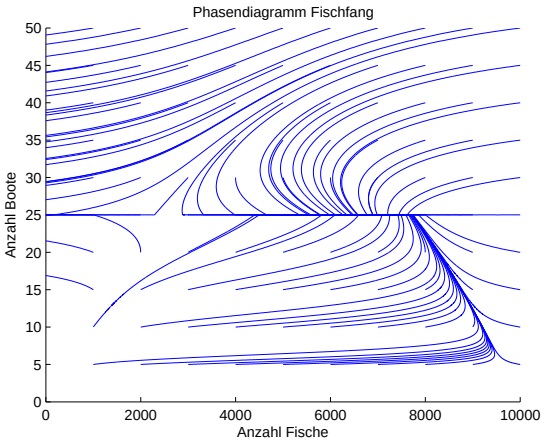
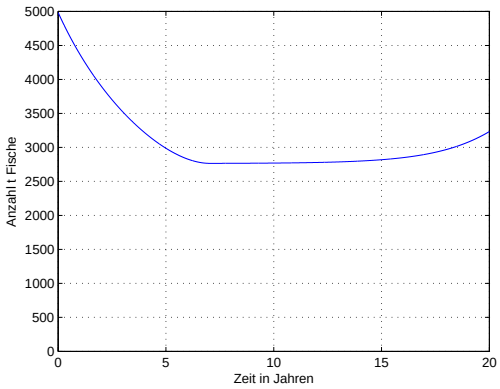
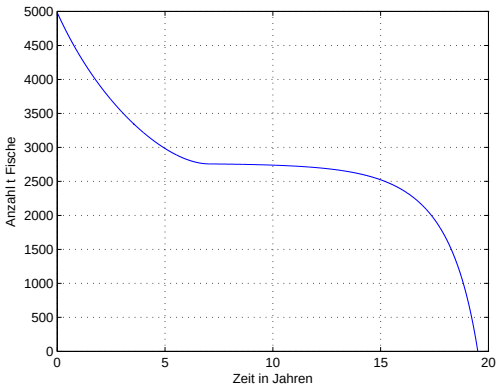


Abbildung 12: Phasendiagramm des Fischfangmodells für Simulationsläufe mit unterschiedlichen Startwerten.



(a) 4985 t Fisch



(b) 4984 t Fisch

Abbildung 13: Simulation der Fischpopulation mit Startwerten 4985 bzw. 4984 t Fisch und 40 Boote. Schrittweite 0,001.

Befindet sich eine Startwertekonfiguration in unmittelbarer Nähe der Grenze beider Gebiete, können geringe Änderungen zu völlig anderem Verhalten führen. Z.B. führt eine Konfiguration von 4985 t Fisch und 40 Booten zur Stabilisierung im Gleichgewichtspunkt (Abbildung 13a). Eine Änderung auf 4984 t Fisch führt jedoch gleich zum Einbruch der Fischpopulation (Abbildung 13b). Beide Läufe wurden mit dem Simulink-Löser *ode5* (ein Runge-Kutta-Verfahren der Ordnung 5) und einer Schrittweite von 0,001 Jahren durchgeführt.

6.2. Ungekoppelte Simulation

Die Batterie wird vollständig sowohl in Simulink als auch in Modelica modelliert. Zuerst wird das Simulink-Modell in Simulink und das Modelica-Modell in OpenModelica simuliert (Abbildungen 11c und 11d). Das sind die Referenzergebnisse. Beide Modelle werden mindestens 30 s lang simuliert, so dass der Ladestand seinen Nullpunkt erreicht. Unter Simulink wird der bereits vorgestellte Runge-Kutta-Löser *ode5* mit fester Schrittweite 0,005 s genutzt. Unter OpenModelica wird der DASSL-Löser genutzt und ebenfalls auf 0,005 s konfiguriert.

Dann werden das Simulink- und das Modelica-Modell jeweils vollständig in eine SMP2-Komponente eingebettet und geschlossen in SIMSAT simuliert. Der so berechnete Ladestand wird mit dem Referenzladestand verglichen. Dazu werden beide Signale voneinander abgezogen und das Ergebnis negiert, falls es kleiner als Null ist. Es wird also die absolute Differenz gebildet. Es sollten keine wesentlichen Abweichungen existieren.

6.2.1. Batterie in Simulink

Abbildung 14 zeigt die absolute Differenz des in Simulink berechneten Referenzladestandes und des in SIMSAT berechneten Ladestandes.

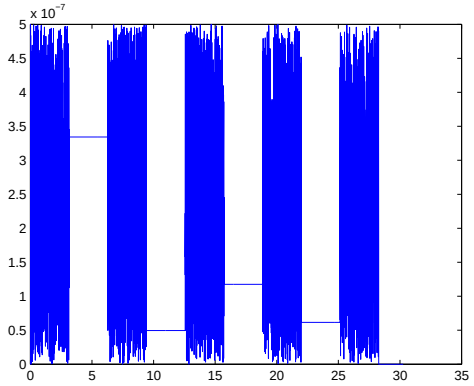


Abbildung 14: Absolute Differenz der Simulink-SMP2-Komponente zum Referenzergebnis.

Darin sind konstante Abschnitte zu erkennen, die sich mit fluktuierenden Abschnitten abwechseln. In den Zeitspannen in denen das Batteriesignal stagniert, ist auch die Differenz beider Ladestände konstant. In den Zeitspannen in denen das Batteriesignal abfällt, fallen die Differenzen unterschiedlich groß aus, etwa gleichverteilt von nahe Null bis $5 * 10^{-7}$. Dieser Fehler tritt bei der Ergebnisausgabe in SIMSAT auf. Momentan werden die Ergebnisse bei Ausgabe in SIMSAT auf die sechste Nachkommastelle gerundet. Die größte Abweichung tritt dann auf, wenn das Ergebnis z.B. 0,0000005 ist und auf 0,000001 gerundet wird. Die Genauigkeit der Ausgabe kann im Einbettungsprozess erhöht werden.

6.2.2. Batterie in Modelica

Abbildung 15 zeigt die absolute Differenz des OpenModelica-Referenzladestandes und des Ladestandes der SMP2-Komponente. Die Abweichungen gleichen denen der Simulink-Batterie. Die Ungenauigkeiten der Größe $5*10^{-7}$ sind wieder auf die Ausgabe in SIMSAT zurückzuführen. Allerdings ist zusätzlich ein leichter Anstieg sowohl bei den konstanten als auch bei den fluktuierenden Abschnitten zu erkennen. Da das Batteriesignal im weiteren Verlauf konstant Null bleibt, sind Untersuchungen mit anderen Parametern oder Modellen nötig, um festzustellen, ob ein systematischer Fehler oder Zufall vorliegt. Diese Untersuchung konnte in dieser Arbeit nicht mehr vorgenommen werden.

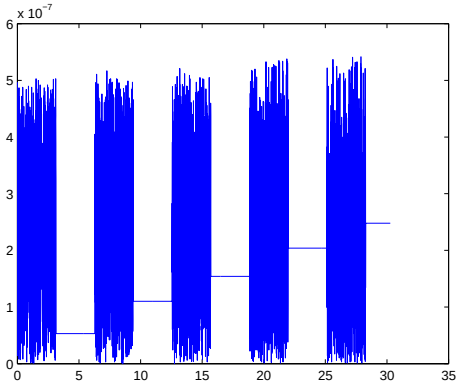


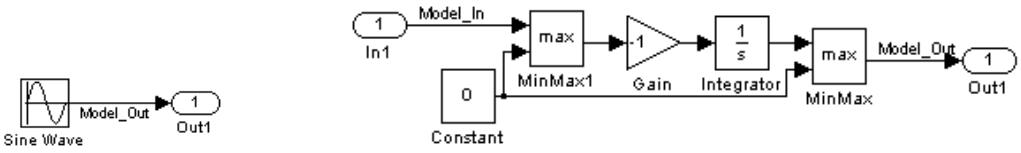
Abbildung 15: Absolute Differenz der OpenModelica-SMP2-Komponente zum Referenzergebnis.

6.3. Einfach gekoppelte Simulation

Das Batteriemodell wird in zwei Modelle geteilt und jedes Teilmodell als eine SMP2-Komponente eingebettet. Beide Komponenten werden auf SMP2-Ebene gekoppelt und ausgeführt. Die Ergebnisse werden wieder mit den Referenzergebnissen verglichen. Zuerst werden beide Komponenten aus Simulink erzeugt, dann aus OpenModelica, und schließlich untereinander ausgetauscht, um zu zeigen, dass auch aus unterschiedlichen Werkzeugen stammende Komponenten gekoppelt werden können.

6.3.1. Kopplung von Simulink-Komponenten

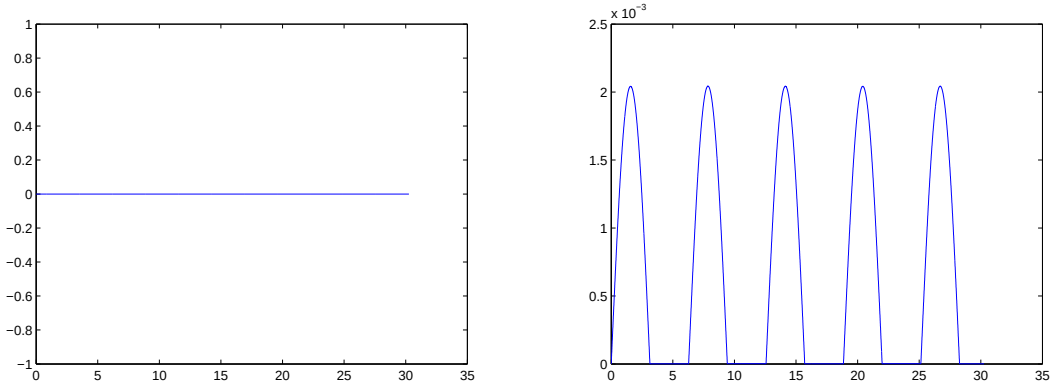
Abbildung 16 zeigt, wie das Batterie-Modell unter Simulink aufgeteilt ist. Jedes Teilmodell wird einzeln in eine SMP2-Komponente eingebettet. Unter SIMSAT werden sie gekoppelt ausgeführt. Der Verlauf des Sinus-Signals sowie der Verlauf des Batterie-Ladestandes werden mit den Referenzergebnissen verglichen.



(a) Sinus-Eingang (b) Batterie

Abbildung 16: Getrenntes Batterie-Modell in Simulink

Das Sinus-Signal ist identisch zum Sinus-Signal im Referenzmodell (Abbildung 17a). Das Batterie-Signal zeigt eine periodische Abweichung (Abbildung 17b). Sinkt der Ladestand, gibt es eine sinusförmige Abweichung. Stagniert der Ladestand, gibt es eine wesentlich kleinere Abweichung (der Größenordnung $5 * 10^{-7}$, siehe Abschnitt 6.2).



(a) Differenz des Sinus-Signals (b) Differenz des Batterieladestandes

Abbildung 17: Absolute Differenz des Sinus-Signals bzw. des Batterie-Ladestandes zur Referenzsimulation.

Es wird geschlussfolgert, dass bei Kopplung der Lösungsalgorithmus anders zum Ergebnis beiträgt, als bei einer geschlossenen Simulation. Das verwendete Verfahren *ode5* berechnet in jedem Simulationsschritt insgesamt 5 Teilschritte, aus denen der eigentliche nächste Wert berechnet wird. In einer geschlossenen Simulation kann der Löser für

die Batterie in jedem Teilschritt einen neuen Eingangswert beziehen. In der gekoppelten Simulation bleiben Eingangswerte für alle Teilschritte konstant. Deshalb rechnen alle Teilschritte mit demselben Wert. Ganz so, als ob das Eingangssignal stückweise konstant wäre. Das äußert sich für dieses Modell in einer kleinen Signalverschiebung, die nur beim Kurvenabfall zur Geltung kommt und deren größte Differenz auf der Y-Achse 0,0021 beträgt.

Um diesen Effekt auch unter Simulink nachzuvollziehen, wurde das Modell erweitert. Das Simulink-Modell in Abbildung 18 simuliert diese Beschränkung. Zwischen Sinus-Signal und Batterie-Berechnung wurde ein Umsetzer für Schrittweiten eingefügt. Dieser stellt sicher, dass nur zu Hauptschritten neue Werte kommuniziert werden. Die absolute Differenz des berechneten Ladestandes zum Referenz-Ladestand ist in Abbildung 19 abgetragen. Die Form der Abweichung gleicht der in Abbildung 17b; lediglich die maximale Abweichung ist mit 0,0025 etwas höher als mit 0,0021.

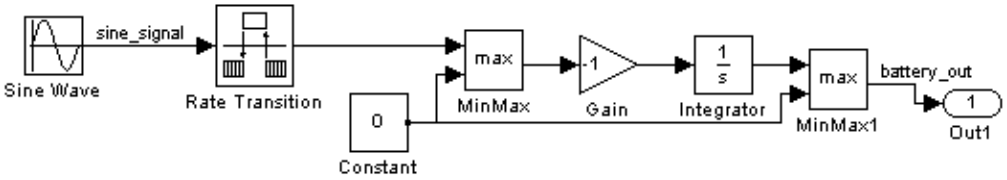


Abbildung 18: Simulink-Modell zur Simulation der Kommunikationsbeschränkung.

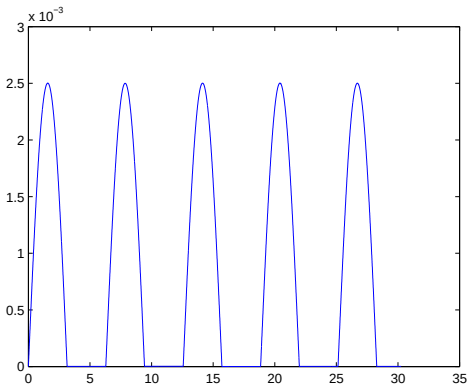


Abbildung 19: Absolute Differenz des beschränkten Batterie-Ladestandes zur Referenzsimulation mit simulierter Kommunikationsbeschränkung.

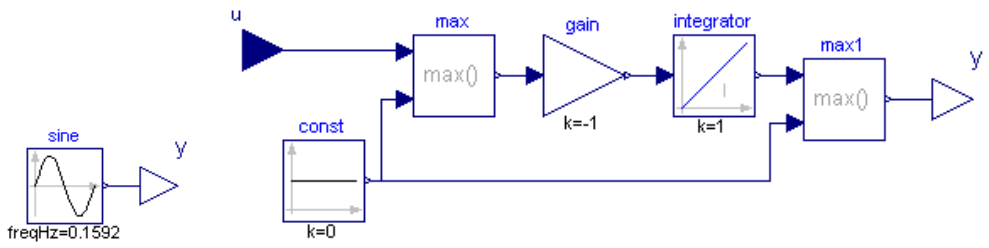
Die Abweichung lässt sich nicht durch Änderung der Genauigkeitsschranken des Lösungsalgorithmus verbessern. Lediglich eine Verkleinerung der Schrittweite führt zu einer Verkleinerung der Abweichung.

6.3.2. Kopplung von OpenModelica-Komponenten

Abbildung 20 zeigt, wie das Batterie-Modell unter Modelica aufgeteilt ist. Es wird genauso getrennt und gekoppelt ausgeführt wie das Simulink-Batteriemodell zuvor. Die entsprechenden Modelica-Quelltexte lauten:

```
model sine_source
  Modelica.Blocks.Interfaces.RealOutput y;
  Modelica.Blocks.Sources.Sine sine(freqHz=0.1592);
equation
  connect(sine.y, y);
end sine_source;

model battery
  Modelica.Blocks.Interfaces.RealInput u;
  Modelica.Blocks.Interfaces.RealOutput y;
  Modelica.Blocks.Continuous.Integrator integrator(y_start=10);
  Modelica.Blocks.Math.Max max;
  Modelica.Blocks.Math.Max max1;
  Modelica.Blocks.Math.Gain gain(k=-1);
  Modelica.Blocks.Sources.Constant const(k=0);
equation
  connect(u, max1.u1);
  connect(const.y, max1.u2);
  connect(max1.y, gain.u);
  connect(gain.y, integrator.u);
  connect(integrator.y, max.u1);
  connect(max.y, y);
  connect(const.y, max.u2);
end battery;
```



(a) Sinus-Eingang (b) Batterie

Abbildung 20: Getrenntes Modelica-Batterie-Modell in Dymola visualisiert.

Die gekoppelte Simulation der Modelica-Modelle zeigt ganz ähnliche Abweichungen von der geschlossenen OpenModelica-Referenzsimulation. Das Sinus-Signal ist ebenfalls identisch. Abbildung 21 zeigt die Abweichungen des Batterie-Ladestandes. Die Abweichungen sind im Vergleich zu den Simulink-Komponenten mit höchstens 0,00045 etwas kleiner. Das liegt daran, dass OpenModelica-Komponenten in der jetzigen Implementierung mit einem Fünftel der eigentlichen Schrittweite arbeiten. Eine Vergleichssimulation unter Simulink mit der Schrittweite 0,001 statt 0,005 ergab eine ähnliche Maximalabweichung von 0,0005.

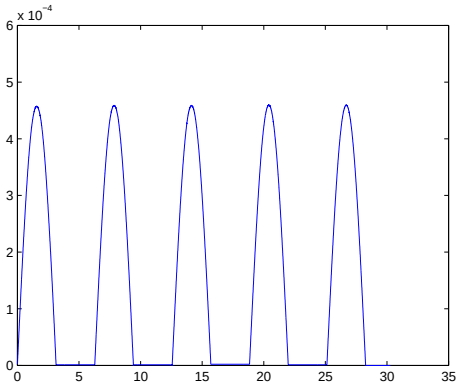


Abbildung 21: Absolute Differenz des Batterie-Ladestandes zur OpenModelica-Referenzsimulation.

6.3.3. Kopplung Gemischter Komponenten

Die bisher vorgestellten Komponenten werden nun untereinander kombiniert. Zuerst wird die aus dem Simulink-Modell erzeugte Sinus-Komponente mit der mit OpenModelica erzeugten Batterie-Komponente gekoppelt. Die Abweichung der Sinus-Komponente ist deshalb identisch zum vorher gezeigten in Abbildung 17a gleich null.

Die Differenz des Batterie-Signals zu den Simulink-Referenzergebnissen in Abbildung 22 ähnelt den vorigen Ergebnissen. Die maximale Abweichung liegt noch unter 0,0016. Die OpenModelica-Batterie arbeitet also mit der Simulink-Sinus-Komponente gut zusammen und berechnet einen Ladestand der vergleichbar ist mit dem der Simulink-Batterie-Komponente. Die Differenz der OpenModelica-Komponente zum Simulink-Referenzergebnis ist sogar kleiner als die Differenz der aus Simulink erzeugten Batterie-Komponente.

Jetzt wird die mit OpenModelica erzeugte Sinus-Komponente mit der Batterie-Komponente des Simulink-Modells gekoppelt und wieder mit den Simulink-Referenzergebnissen verglichen. Die Differenz des Sinus-Signals in Abbildung 23a zeigt einen sich aufschwingenden Fehler. Dieser geht auf unterschiedliche Sinus-Frequenzen in den Modellen zurück. Die Standarddefinition des Sinus unterscheidet sich zwischen Simulink und Modelica [Rö08]. Bei der OpenModelica-Implementierung des Sinus wurde mit einer Frequenz von

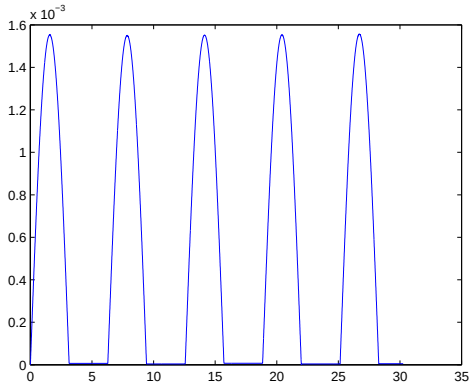
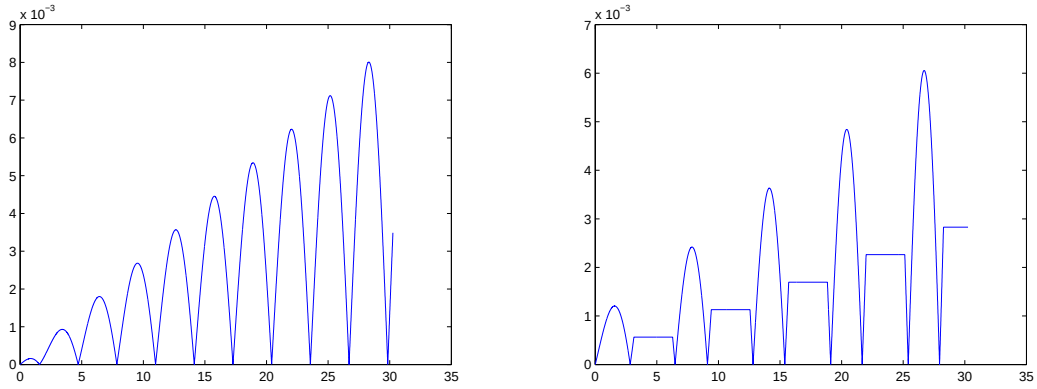


Abbildung 22: Absolute Differenz des Batterie-Ladestandes zur Simulink-Referenzsimulation.

$(2\pi)^{-1} \approx 0{,}1592$ versucht, den Simulink-Sinus zu approximieren. Das gelingt aber erwartungsgemäß nur auf die Zehntausendtel-Stelle genau. Der Unterschied in den Frequenzen schwingt sich auf und schlägt sich schnell in einer Größenordnung von 10^{-3} nieder. Das Modell kann leicht auf einen genaueren Wert geändert und der Fehler so behoben werden.



(a) Sinus-Eingang

(b) Batterie

Abbildung 23: Absolute Differenz des Sinus-Signals bzw. des Batterie-Ladestandes zur Simulink-Referenzsimulation.

Dementsprechend weist auch der Ladestand in Abbildung 23b eine sich aufschwingende Abweichung auf. Der Unterschied ist wieder auf die unterschiedliche Frequenz im Sinus-Signal zurückzuführen. Der Frequenzunterschied führt auch zu einer zeitlich versetzten Stagnation des Ladestandes. Deshalb zeigen sich auch während der Stagnationsphasen Abweichungen, die stetig größer werden.

6.4. Simulation von Systemen mit Rückkopplung

Das Fischfang-Modell wird in die drei Teilmodelle z1, z2 und Fangmenge aufgeteilt (Abbildung 24). Die Teilmodelle z1 und z2 berechnen jeweils die Zustandsgröße Anzahl der Fische bzw. Anzahl der Boote. Das Teilmodell Fangmenge berechnet aus den Anzahlen Fische und Boote die Fangmenge. Die Fangmenge geht als Differentialwert wieder in die Berechnung von z1 und z2 ein. Alle Teilmodelle sind in Simulink implementiert, um Nebeneffekte gemischter Simulationen (im vorigen Abschnitt betrachtet) auszuschließen. Die Implementierung ist in Anhang A.4 ersichtlich.

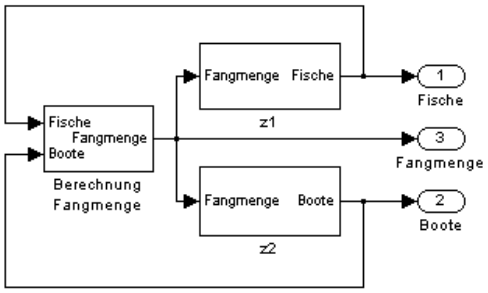


Abbildung 24: Simulink-Modell der gekoppelten Teilmodelle z_1 (Fischpopulation), z_2 (Bootsanzahl) und Fangmenge.

Die drei Teilmodelle werden jeweils in eine SMP2-Komponente eingebettet und auf SMP2-Ebene gekoppelt und ausgeführt. Über die drei Komponenten wird so ein dynamisches System mit Rückkopplungen aufgespannt. Die simulierte Zeitdauer beträgt 20 Jahre.

6.4.1. Instabiles Fischfang-Modell

Werden die Startparameter des Systems so gewählt, dass der Anfangszustand möglichst nah an einer Grenze von Zustandsregionen eines Systems liegt, können kleine Änderungen der Parameter zu unterschiedlichen Stabilitätsregionen und damit zu einem völlig anderen Verhalten führen. Ein Beispiel dafür wurde bei der Modellvorstellung genannt. Für das Fischfang-Beispiel liegen die Startparameter 4985 t Fisch und 40 Boote sehr nah an dieser Grenze. Simuliert man das System mit diesen Parametern unter Simulink, verringert sich die Fischmenge im Verlauf bis auf 2600 t Fisch und erholt sich langsam wieder (Abbildung 13a).

Die Simulation der gekoppelten SMP2-Komponenten bei einer Schrittweite von 0,001 Jahren gibt diesen Verlauf wieder (Abbildung 25). Es sind aber schon leichte Abweichungen festzustellen. Während sich die Fischpopulation nach 20 Jahren eigentlich schon auf über 3000 t erholt haben müsste, bleibt sie in dieser Simulation noch knapp darunter.

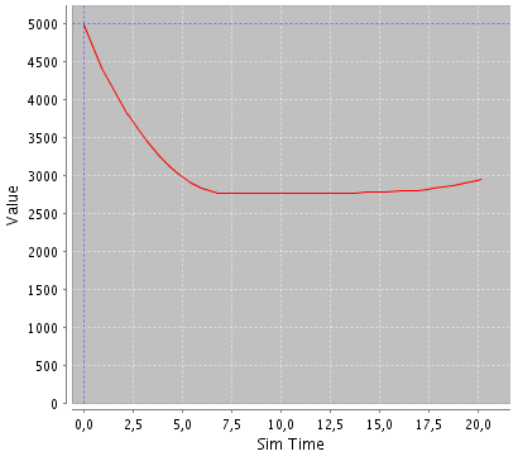
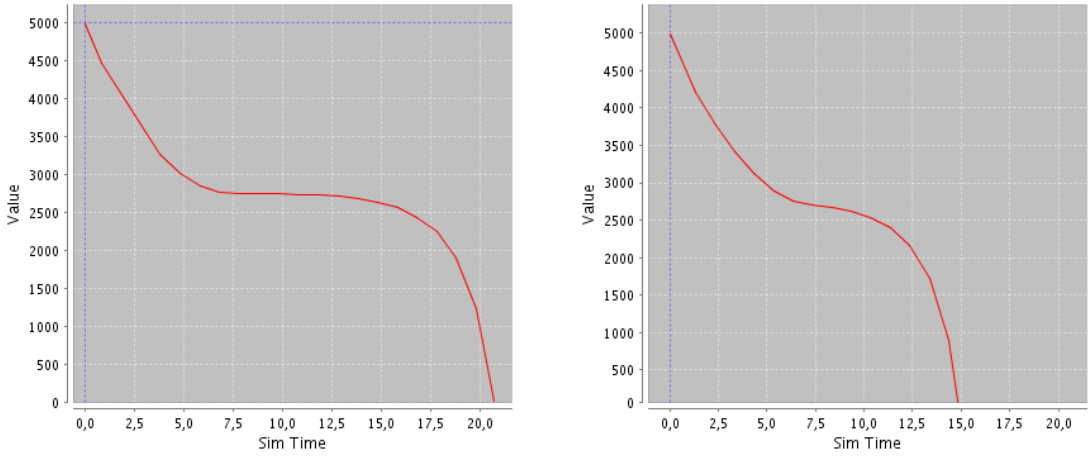


Abbildung 25: Simulation der Fischpopulation in SMP2 mit Startwerten 4985 t Fisch, 40 Boote, Schrittweite 0,001.

Die Kurve der Fischpopulation verläuft über die simulierte Zeit von 20 Jahren relativ träge. Es könnte angenommen werden, dass es bei der Schrittweite nicht auf Tausendstel Jahre ankommt. Eine Schrittweite von 0,1 sollte demnach ähnliche Ergebnisse liefern wie eine Schrittweite von 0,001. Ändert man die Schrittweite jedoch nur wenig auf 0,005, bricht die Fischpopulation plötzlich ein (Abbildung 26a).



(a) Sinus-Eingang

(b) Batterie

Abbildung 26: Simulation der Fischpopulation in SMP2 mit Startwerten 4985 t Fisch, 40 Boote, sowie Schrittweiten von 0,005 bzw. 0,05.

Für noch größere Schrittweiten bricht der Verlauf noch schneller ein. Z.B. in Abbildung 26b mit 0,05 Jahren Schrittweite.

Anscheinend ist das Systemverhalten nicht nur von kleinen Parameteränderungen, sondern bei SMP2-Kopplung auch empfindlich von der gewählten Schrittweite abhängig. Das wird im Folgenden erklärt.

6.4.2. Probleme SMP2-gekoppelter dynamischer Systeme

Das beobachtete plötzliche Zusammenbrechen der Fischpopulation hängt allgemein zunächst mit der Nähe der Startparameter zur Stabilitätsgrenze zusammen (Abbildung 12). Für zu große Schrittweiten ist dann nicht vorhersagbar, in welche Zustandsregion eine Berechnung laufen wird. Bei SMP2-Kopplung von dynamischen Systemen hängt der Zusammenbruch speziell mit der Größe der Schrittweite, mit der Beschränkung des Lösungsverfahrens, dem Abtasten von Differentialsignalen und der Auftrennung einer Rückkopplung zusammen.

Größe der Schrittweite

Da die Berechnungsschritte diskret sind, wird mit jedem Schritt der genaue Zustandspfad um eine gewisse Genauigkeit verfehlt und in einen anderen Pfad gewechselt. Je feiner die Schrittweiten dabei sind, desto genauer wird der durch die Startparameter gewählte Zustandspfad verfolgt. Je gröber die Schrittweiten sind, desto größer sind die Sprünge zwischen den Pfaden.

Das Problemverhalten kann auch direkt in Simulink nachvollzogen werden. Im Folgenden wird das Fischfangmodell geschlossen unter Simulink simuliert. Die Startparameter sind wieder 4985 t Fisch und 40 Boote, der Lösungsalgorithmus ebenfalls wieder *ode5*. Experimente ergaben, dass das Einbruchverhalten des SMP2-gekoppelten Systems mit 0,005 Jahren Schrittweite nachvollzogen werden kann, wenn in der geschlossenen Simulink-Simulation mit 0,1 Jahren Schrittweite gerechnet wird (Abbildung 27).

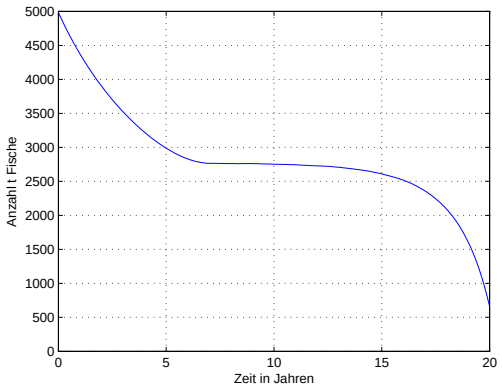


Abbildung 27: Simulation der Fischpopulation in Simulink mit Startwerten 4985 t Fisch, 40 Boote, Schrittweite 0,1.

Es wird deutlich, dass die geschlossene Simulink-Simulation wesentlich genauer rech-

net als die SMP2-gekoppelte Simulation. Ein vergleichbarer Einbruch der Fischpopulation tritt bei der Simulink-Simulation erst bei einer Schrittweite von 0,1 auf, bei der SMP2-gekoppelten aber schon bei 0,005. Es sind also jenseits der Schrittweite noch weitere Faktoren vorhanden, die die Genauigkeit der SMP2-Simulation beeinflussen. Diese werden im Nachfolgenden beleuchtet.

Beschränkung des Lösungsverfahrens

Die bereits beobachtete Beschränkung des Lösungsverfahrens einer SMP2-Komponente durch die getaktete Aktualisierung ihrer Eingangsvariablen lässt sich auch hier nachvollziehen. Dazu wird das Simulink-Modell wieder so modifiziert, dass die verbindenden Signale nur zu Hauptsimulationsschritten Werte aktualisieren können (Abbildung 28).

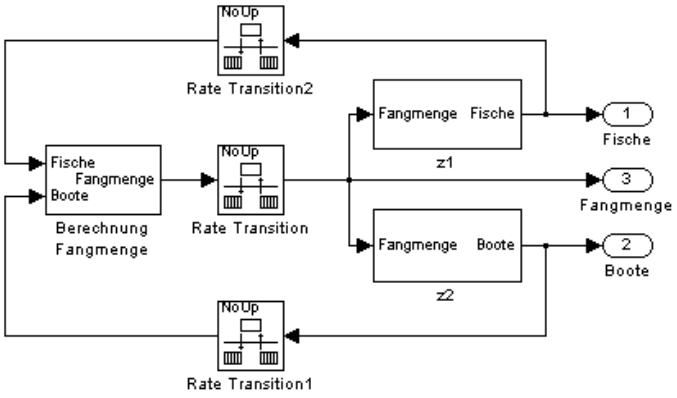


Abbildung 28: Simulink-Fischfang-Modell zur Simulation der Kommunikationsbeschränkung.

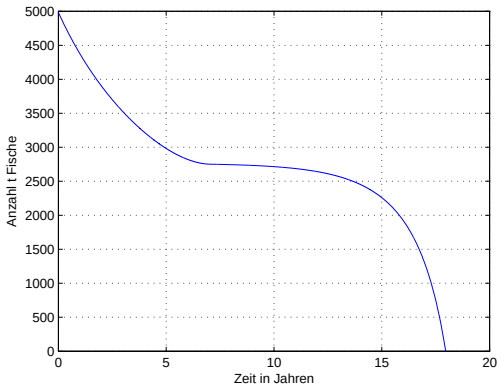


Abbildung 29: Simulation der Kommunikationsbeschränkung beim Fischfangmodell in Simulink mit Startwerten 4985 t Fisch, 40 Boote, Schrittweite 0,005.

Das in Abbildung 29 gezeigte Verhalten tritt diesmal schon bei der Schrittweite von 0,005 auf. Die Funktionsweise ist also mit der der SMP2-Simulation vergleichbar. Die Beschränkung bewirkt hier eine Abschwächung der Schrittweiteneffektivität um den Faktor 20. Mit anderen Worten: Ergebnisse, die mit geschlossener Simulation der Schrittweite 0,1 berechnet werden, können in der gekoppelten SMP2-Simulation erst mit einer Schrittweite von 0,005 reproduziert werden.

Warum die Fischpopulation im Simulink-Modell mit der simulierten Beschränkung noch schneller einbricht als bei der gekoppelten SMP2-Simulation konnte in dieser Arbeit nicht weiter verfolgt werden.

Abgetastete Differentialsignale Bei der Modellierung von Differentialgleichungen liegt häufig ein Differentialsignal als Eingang einer Komponente an und wird in der Komponente integriert. Wenn dieses Signal mit einer zu großen Schrittweite abgetastet wird, können unvorhersagbare Kurvenverläufe entstehen. Bei periodischen Signalen ist eine Schrittweite ausreichend klein, wenn ihr Inverses größer als das Doppelte der höchsten, im Signal vorkommenden Frequenz ist (Nyquist-Abtasttheorem).

Während bei der Unterabtastung von einfachen Signalen zwar neue Kurven entstehen können, ist doch zu den Abtastzeiten der abgetastete Wert mit dem Funktionswert identisch. Bei Abtastung von Differentialsignalen trifft dies nach der Integration des Signals nicht mehr zu. Abbildung 30 zeigt ein Beispiel mit der Differentialgleichung:

$f'(t) = cos(t) - 1/4$.

Das Integral lässt sich leicht bestimmen:

$f(t) = sin(t) - t/4 + f(0)$.

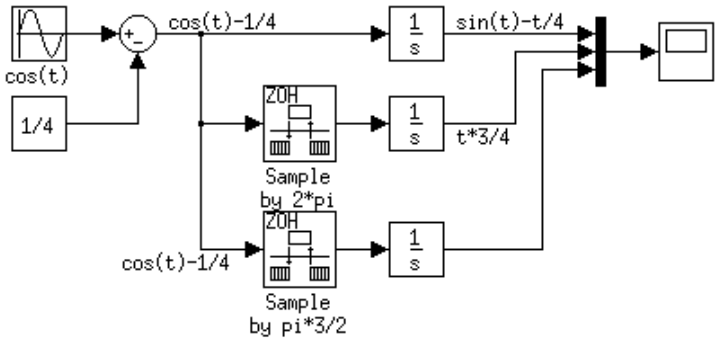


Abbildung 30: Simulink-Modell zur Demonstration abgetasteter Differentialsignale.

Das Beispiel lässt den Lösungsalgorithmus von Simulink die Differentialgleichung lösen. Dazu wird das Signal, das den Differentialterm $cos(t) - 1/4$ trägt, integriert. Zusätzlich wird das Signal mit dem Differentialterm mit zwei verschiedenen Takten abgetastet. Die resultierenden Signale werden ebenfalls integriert. Abbildung 31 zeigt die Ergebnisse.

Die durchgezogene Kurve ist das nicht abgetastete, integrierte Signal. Bei der Strich-

punktlinie wurde alle $2 * \pi$ Sekunden abgetastet, bei der gepunkteten Kurve alle $3\pi/2$ Sekunden. Während die durchgezogene Kurve wie erwartet verläuft, ist die mit $2 * \pi$ abgetastete Kurve eine lineare Funktion und die mit $3\pi/2$ abgetastete Kurve ein abfallender Sinus mit niedrigerer Frequenz.

Die lineare Funktion ergibt sich, weil ein Cosinus an allen Stellen $2\pi k$ den Wert 1 hat. Als Differentialsignal ergibt sich nach Abtastung alle 2π Sekunden deshalb eine konstante Funktion. Deren Ableitung ist die dargestellte lineare Funktion $t * 3/4$. Die gepunktete Kurve ergibt sich, weil durch die Abtastung alle $3/2\pi$ ein Cosinus größerer Frequenz entsteht.

Wenn keine Vorannahmen über die Dynamik im simulierten System getroffen werden können, kann das Abtasttheorem nicht eingehalten werden. Es muss dann davon ausgegangen werden, dass Ergebnisse auch durch Unterabtastung verfälscht werden.

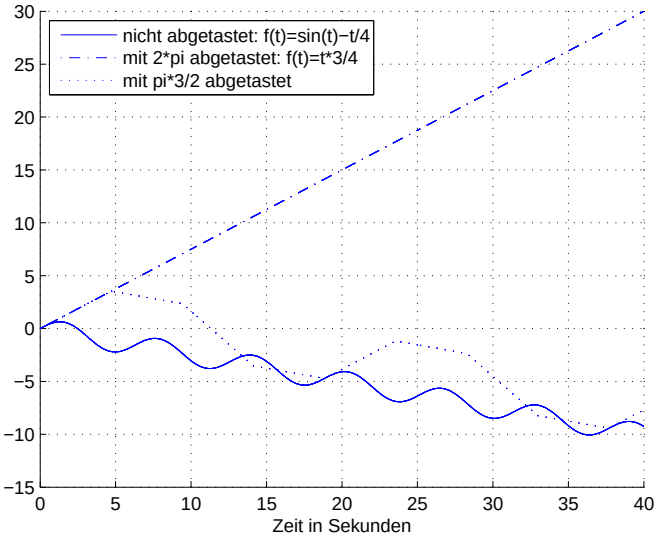


Abbildung 31: Ein korrekt berechnetes Signal (durchgezogen) und durch Unterabtastung des Differentialsignals entstandene Signale (gepunktete Linie, Strichpunktlinie).

Rückkopplungen

Wenn Schleifen über mehrere Komponenten aufgespannt werden, wie z.B. im Fischfangmodell, können sich weitere Ungenauigkeiten ergeben.

Bei der Abarbeitung von Blockdiagrammen werden Blöcke aktualisiert und ihre Ausgangswerte an Eingänge anderer Blöcke geleitet. Dabei spielt die Reihenfolge eine bedeutende Rolle. Wird in einem Zeitschritt eine Komponente aktualisiert, bei der noch keine aktuellen Werte der anderen Komponenten anliegen, berechnet sie den Wert des letzten Zeitschrittes. In Blockdiagrammen ohne Schleifen lassen sich alle Blöcke in eine geeignete Reihenfolge bringen, so dass bei der Aktualisierung jedes Blockes schon aktuelle Werte

verfügbar sind. Kommen Schleifen ins Spiel, funktioniert das nicht mehr.

Simulink kennt zwei Ansätze, mit Schleifen umzugehen. Hat keiner der Blöcke, über welche die Schleife führt, einen internen Zustand (wie z.B. der Stand eines Integrators), versucht Simulink durch mehrmaliges Aktualisieren der Schleifenblöcke, in jedem Simulationsschritt eine Konvergenzlösung herbeizuführen. Gelingt dies, wird mit dieser Lösung der Zeitschritt weiter berechnet. Gelingt dies nicht, bricht die Berechnung ab. Diese Variante führt zu einem stark erhöhten Berechnungsaufwand.

Hat einer der Schleifenblöcke einen internen Zustand, wird die Schleife an dieser Stelle aufgebrochen. Der zustandsbehaftete Block berechnet als erster seine Ausgangswerte aus dem inneren Zustand. Darauf aufbauend werden dann alle folgenden Blöcke aktualisiert. Der zustandsbehaftete Block wird zum Schluss aktualisiert. So verschleppt sich ein kleiner Fehler von Schritt zu Schritt.

Bei der Kopplung von SMP2-Komponenten muss ebenfalls mit Schleifen umgegangen werden. Da auf SMP2-Ebene in jedem Zeitschritt Komponenten nur einmal aktualisiert werden, kann keine Konvergenzlösung berechnet werden. Stattdessen müssen Schleifen an geeigneten Stellen aufgetrennt werden. Bei der Simulation des Fischfangmodells wurden die zwei Schleifen in Anlehnung an Simulink bei den zustandsbehafteten Komponenten z_1 und z_2 aufgetrennt (Abbildung 28), um die Modelle unter SMP2 gekoppelt ausführen zu können.

7. Zusammenfassung

Simulink-Modelle lassen sich nun automatisch in SMP2-Modelle einbetten. Die bisherige Lösung mit den Programmen Real-Time Workshop und MOSAIC wird verwendet. Der Quelltext eines SMP2-Modells wird anschließend noch modifiziert, damit das SMP2-Modell kommunizieren kann. Eingebettete Simulink-Modelle können nun einfach über Field-Links mit anderen Modellen verknüpft werden. Außerdem waren Modifikationen nötig, um den von MOSAIC generierten Quelltext überhaupt SMP2-konform werden zu lassen. Es konnten bestehende Probleme, wie die fehlende Schritt–0-Initialisierung gelöst werden.

Die Einbettung ist auf Simulink-Modelle bestimmter Eigenschaften beschränkt. Die Beschränkungen werden an verschiedenen Stellen des Einbettungsprozesses eingeführt. Sie sind im Anhang A.1 vollständig aufgezählt.

Modelica-Modelle lassen sich ebenfalls automatisch in SMP2-Modelle einbetten. Dazu wird der Modellcompiler OpenModelica verwendet. Seine Laufzeitbibliothek wird modifiziert und der erzeugte Modellcode in ein SMP2-Modell eingebettet. Eingebettete Modelica-Modelle können ebenfalls über Field-Links mit anderen Modellen verknüpft werden. Unter anderem wurde das Problem der Aktualisierung der Eingangsvariablen des Modells gelöst. Die Lösung benutzt den Modelica-Mechanismus der externen Funktionen statt die Variablen im Code zu überschreiben und könnte so auch mit allen anderen Modelica-Compilern genutzt werden.

Verwendbare Modelica-Modelle unterliegen nur wenigen Einschränkungen. Diese sind im Anhang A.1 zu finden.

Es hat sich gezeigt, dass mit einfachem SMP2 keine komplexen Steuerungsstrukturen abbildbar sind. Der Lösungsalgorithmus eines Modells stellt mitunter fest, dass zusätzliche Berechnungen als Teilschritte nötig werden. Dann müssten auch andere Modelle zu diesen Teilschritten Werte berechnen, um selbst aktuell zu bleiben und andere Modelle mit aktuellen Ergebnissen zu versorgen. Um die Berechnungen aller Modelle zu koordinieren sind Wiederholungen und Rücksprünge in der Zeit notwendig. Diese sind in SMP2 nicht umsetzbar.

Durch SMP2 wird die Koordination der Berechnungsausführung daher beschränkt. Direkt mit SMP2-Mechanismen lassen sich Modelle mit festen Schrittweiten ausführen, nicht mit variablen. Auch die Einbettung von Simulink-Modellen unterstützt nur feste Schrittweiten. Daher wurde weiter untersucht, welche Einschränkungen für die Ergebnisse gekoppelter Simulationen gelten.

7.1. Simulationsergebnisse

Bei der Einbettung von Einzelmodellen ohne Kopplung zu anderen Komponenten werden die gleichen Ergebnisse wie bei einer geschlossenen Simulation in Simulink bzw. OpenModelica berechnet. Die Werteprotokollierung in SIMSAT rundet Ergebnisse momentan auf

sechs Stellen, so dass Abweichungen der Größenordnung 10^{-7} auftreten. Die Genauigkeit kann bei Bedarf einfach erhöht werden.

Bei der einfachen Kopplung zweier SMP2-Komponenten wurden Abweichungen von Referenzergebnissen festgestellt. Der Vergleich der Sinus-Signale von Simulink und OpenModelica ergab eine Frequenzverschiebung im Modell. Dieser Unterschied ist leicht durch die Anpassung des Testmodells zu beheben, zeigt aber ansatzweise, dass bei Kopplung von Komponenten unterschiedlicher Herkunft Probleme auftreten können.

Es wurden jedoch auch Abweichungen festgestellt, die sich systematisch aus der Kopplungsmethode ergeben. Die Komponenten einer SMP2-Simulation werden nur zu den Hauptschritten der festen Schrittweite mit neuen Eingangswerten der anderen Komponenten versorgt. Die internen Lösungsalgorithmen benötigen aber gegebenenfalls auch Werte von Zwischenschritten. Konkret kam ein Runge-Kutta-Verfahren der Ordnung 5 zum Einsatz, das je Simulationsschritt fünf Stützwerte innerhalb des Intervalls benötigt. Für alle Zwischenschritte sind aber nur die gleichen, bei der letzten Aktualisierung übertragenen Eingangswerte verfügbar. Für das verwendete Batteriebeispiel bestimmen die Eingangswerte vollständig die Differentialgleichung des Modells. Es kann gezeigt werden, dass alle Runge-Kutta-Verfahren mit dieser Beschränkung effektiv wie ein einfaches Euler-Verfahren rechnen (siehe Anhang A.5).

Diese Beschränkung muss mindestens bei allen Lösungsverfahren angenommen werden, die in einem Simulationsschritt Werte von Teilschritten benötigen. Die Beschränkung ist immer dann in Kraft, wenn der Lösungsalgorithmus einer Komponente Eingangswerte zu Zwischenschritten benötigt, denn die Eingangswerte können nur zu Hauptschritten aktualisiert werden. Im Beispielmodell führte das schon zu Abweichungen der Größenordnung $2 * 10^{-3}$.

Wenn die Kopplung von SMP2-Komponenten ein Differentialgleichungssystem ergibt, führt die beschriebene Beschränkung der Lösungsalgorithmen zu wesentlich stärkeren Ergebnisabweichungen. Ein Beispielmodell musste mit 20fachem Aufwand berechnet werden, um die entstandenen Abweichungen auszugleichen. Zum einen treten bei solchen Systemen potentiell alle identifizierten Probleme auf: ungeeignete Schrittweiten, Beschränkung der Genauigkeit verwendeter Lösungsverfahren, Unterabtastung von Differentialsignalen und Informationsverzögerung durch Schleifen. Zum anderen können durch die Ungenauigkeiten andere Stabilitätsregionen des dynamischen Systems angesteuert werden. Im Beispiel brach die Simulation einer florierenden Fischpopulation durch eine Schrittweitenänderung um 0,004 plötzlich ein.

7.2. Ausblick

Ein kurzer Ausblick soll die Erweiterungsmöglichkeiten der Einbettungs selbst, aber auch im Umgang mit gekoppelten Simulationen aufzeigen.

7.2.1. Implementierung

Die Implementierung der Modelica-Einbettung zeigt, wie die SMP2-Einbettung anderer Modelliersprachen funktionieren kann. Erforderlich ist ein Modellcompiler, der aus einer Modellbeschreibung die Berechnung des Modells als Simulationscode generieren kann. Um die Ausführung auf SMP2-Mechanismen zu übertragen, wird der Code der Laufzeitbibliothek des Modellcompilers neu organisiert. Die Anweisungen zur Berechnung eines Simulationsschrittes werden extrahiert und zu einer Funktion zusammengefasst. Diese Funktion kann dann über SMP2 getaktet werden und berechnet jeweils einen Simulationsschritt. Der Zugriff auf Variablen kann über den Modellcode geschehen. Wissen über Namen und Struktur der gespeicherten Variablen ist notwendig.

Bei der Simulink-Einbettung und der Modelica-Einbettung werden Variablen des Simulationscodes auf SMP2-Ebene dupliziert. Statt der Duplizierung könnten die zusätzlich angelegten Variablen einfach die Speicherstelle im Simulationscode referenzieren. So macht es das Programm MOSAIC. Dadurch entfallen die Kopieroperationen zur Laufzeit und die Konsistenz der Variablen wird gewahrt. Speziell bei der Modelica-Einbettung könnte man in Analogie zu MOSAIC auch die vollständigen Arrays, in denen die Variablenwerte gespeichert sind, bekannt machen. Dann wäre allerdings zu erwarten, dass dies die Benutzung des Field-Link-Mechanismus wieder einschränkt. Mit solchen Erweiterungen sollte also gewartet werden, bis der Fehler in SIMSAT behoben ist.

Eine weitere Verbesserung der Modelica-Einbettung ist eindeutig die Verfügbarkeit von Parametern auf SMP2-Ebene. Dies war für die erstmalige Einbettung in dieser Arbeit noch nicht wichtig, spielt aber beim Einsatz mehrerer Modelle eine große Rolle. Insbesondere Simulationsparameter wie Schrittweite und Simulationsdauer sollten über das SMP2-Assembly festgelegt werden können.

Weiterhin kann die Übertragung der Ausführung bei Modelica-Modellen statt über die Laufzeitbibliothek besser mit einem Modelica-Mechanismus erfolgen. Dazu wird im Erweiterungsmodell eine zeitliche Taktung definiert und über einen *when*-Anweisungsblock wieder eine externe Funktion aufgerufen. Diese externe Funktion gibt die Ausführung an das SMP2-Modell ab und blockiert ab diesem Zeitpunkt. Wenn der nächste Simulationsschritt berechnet werden soll, kehrt die externe Funktion erst zurück und lässt den Simulationscode weiterrechnen. Auf diese Weise müsste die Laufzeitbibliothek gar nicht modifiziert werden. Die Einbettung wäre dann unabhängig von Änderungen des Quelltextes bei neuen Versionen von OpenModelica. Auf diese Weise ließe sich sogar Dymola-Code nutzen.

Vektorsignale sollten als Simulink-spezifische Elemente angesehen werden. Datentypen und Funktionen für Vektoren und Matrizen sind die Spezialität von MATLAB (MATrix LABoratory). Es kann nicht vorausgesetzt werden, dass andere Modelliersprachen ähnliche Konzepte bereitstellen, bzw. ihre Modellcompiler ebenfalls Vektoren als Datentyp implementiert haben und in ihren Modellcode exportieren. Modelica bietet z.B. so gut wie keine Unterstützung für Vektorsignale bei Blockdiagrammen. Für Modelica lässt sich diese Einschränkung mit dem Programm Simelica umgehen. Bei Verwendung anderer

Modelliersprachen sind jedoch wieder ähnliche Probleme zu erwarten.

7.2.2. Gekoppelte Simulation

Die Einbettung von Modellen und ihre gekoppelte Ausführung ist dann sinnvoll, wenn die Modelle geeignet gekapselt werden. Komplexe Berechnungen wie das Lösen nichtlinearer Differentialgleichungssysteme sollten am Besten innerhalb einer Komponente durchgeführt werden. So kann der jeweils gewählte Löser seine Stärken ausspielen und ist an keine Beschränkungen gebunden. Über Komponentengrenzen hinweg wird die Wirksamkeit des Lösungsalgorithmus hingegen eingeschränkt. Im Beispiel rechnete ein Runge-Kutta-Verfahren der Ordnung 5 nur so genau wie ein einfaches Euler-Verfahren. Die Beschränkung dürfte typisch bei zu verwendenden Modellen sein, wenn die Modelle zeitabhängige Eingänge besitzen.

Um diese Beschränkung aufzuheben, müsste der Lösungsalgorithmus einer Komponente in jedem Teilschritt mit aktuellen Werten versorgt werden. Dafür müssen andere Komponenten kurzfristig aktuelle Werte berechnen. Es muss also weitere Steuerungsmöglichkeiten für Komponenten geben. Diese Steuerungsmöglichkeiten müssen dann innerhalb der SMP2-Ebene umgesetzt werden. Da jede Komponente jederzeit einen Teilschritt anfordern könnte, gestaltet sich die Ablaufsteuerung viel komplizierter als bisher. Komponenten müssten zeitlich zurückgesetzt werden können, in einem Schritt mehrmals aufgerufen werden und eventuell Berechnungen sogar abgebrochen werden können. Mit einfachen SMP2-Mechanismen ist das nicht umsetzbar.

Das Fischfang-Beispiel hat Probleme bei der Kopplung dynamischer Systeme gezeigt. Es ist davon auszugehen, dass Modelle im Einsatz ein noch komplexeres Stabilitätsverhalten aufweisen. Die Ergebnisse lassen sich dann viel weniger zuverlässig zuordnen.

Eine umfangreiche (Stabilitäts-)Analyse des zu betrachtenden Systems ist erforderlich um erhaltene Simulationsergebnisse zuverlässig interpretieren zu können. Aber der Ansatz im *Virtuellen Satelliten*, bei dem bereits implementierte Komponenten zu Simulationen gekoppelt werden, versteckt die Implementierung der Komponenten. Das vereinfacht die Simulationserstellung für Ingenieure zwar, macht aber eine Analyse des Systems umso schwieriger. Auch die benötigte Schrittweite kann nicht zuverlässig bestimmt werden. Ergebnisse müssten bei jeder Berechnung neu in Frage gestellt werden.

Literatur

[BCP96] Brenan, K.E., S.L. Campbell und L.R. Petzold: *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. siam - Society for Industrial and Applied Mathematics, Philadelphia, republication of 1989 edition from North-Holland, New York Auflage, 1996.

[Bos04] Bossel, Hartmut: *Systeme, Dynamik, Simulation. Modellbildung, Analyse und Simulation komplexer Systeme*. Books on Demand GmbH, Norderstedt, 1 Auflage, 2004.

[Dem03] Dempsey, Mike: *Automatic translation of Simulink models into Modelica using Simelica and the AdvancedBlocks library*. In: *Proceedings of the 3rd International Modelica Conference*, Linköping, 2003.

[Dro04] Dronka, Sven: *Die Simulation gekoppelter Mehrkörper- und Hydraulik-Modelle mit Erweiterung für Echtzeitsimulation*. Shaker Verlag Aachen, 2004.

[ESO05a] ESOC: *Simulation Model Portability 2.0 C++ Mapping EGOS-SIM-GEN-TN-0102 Issue 1 Revision 2*. ESA/ESOC Darmstadt, 2005.

[ESO05b] ESOC: *Simulation Model Portability 2.0 Component Model EGOS-SIM-GEN-TN-0101 Issue 1 Revision 2*. ESA/ESOC Darmstadt, 2005.

[ESO05c] ESOC: *Simulation Model Portability 2.0 Handbook EGOS-SIM-GEN-TN-0099 Issue 1 Revision 2*. ESA/ESOC Darmstadt, 2005.

[ESO05d] ESOC: *Simulation Model Portability 2.0 Metamodel EGOS-SIM-GEN-TN-0100 Issue 1 Revision 2*. ESA/ESOC Darmstadt, 2005.

[FAP+06] Fritzson, P., P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman und A. Sandholm: *OpenModelica - A Free Open-Source Environment for System Modeling, Simulation, and Teaching*. In: *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design, Munich, Germany, October 4-6, 2006*, 2006.

[Fau99] Fausett, Laurene V.: *Applied Numerical Analysis Using MATLAB*. Prentice Hall, 1999.

[Fri04] Fritzson, Peter: *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Computer Society Press, 2004.

[GKL06] Geimer, Marcus, Thomas Krüger und Peter Linsel: *Co-Simulation, gekoppelte Simulation oder Simulatorkopplung? - Ein Versuch der Begriffsvereinheitlichung*. Ölhydraulik und Pneumatik, Heft 11-12:572–576, 2006.

[LF05] Lundvall, Håkan und Peter Fritzson: *Event Handling in the OpenModelica Compiler and Run-time System.* In: *Proceedings of the 46th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS2005)*, 2005.

[LM06] Lammen, W.F. und J. Moelands: *MOSAIC 7.1 User Manual - Automated model transfer from MATLAB7.1/Simulink to ESA's Simulation Model Portability SMP2 standard and the real-time simulation engine EuroSim Mk4.* included electronically in MOSAIC 7.1 distribution, 2006.

[LP08] Lundvall, Håkan und Adrian Pop: *OpenModelica Simulation Runtime.* Powerpoint Presentation online. URL: http://openmodelica.ida.liu.se:8080/cb/displayDocument/OpenModelica-SimulationRuntime.ppt (zuletzt eingesehen am 26.10.2009), 2008.

[Rö08] Röhnsch, Alexander: *Verhaltensintegration in SMP2-Modelle.* Studienarbeit, Humboldt-Universität zu Berlin, 2008.

[RTW05] *Real-Time Workshop User's Manual Version 6.3.* distributed electronically alongside MATLAB Release 14 ServicePack 3, 2005.

[SBMR08] Schumann, Holger, Axel Berres, Olaf Maibaum und Alexander Röhnsch: *DLR's Virtual Satellite Approach.* In: *10th International Workshop on Simulation on European Space Programmes SESP*, 2008.

[SIM05] *Simulink User's Manual Version 6.3.* distributed electronically alongside MATLAB Release 14 ServicePack 3, 2005.

[VGvdS+99] Veitl, A., T. Gordon, A. van de Sand, M. Howell, M. Valasek, O. Vaculin und P. Steinbauer: *Methodologies for Coupling Simulation Models and Codes in Mechatronic System Analysis and Design.* In: Fröhling, Robert (Herausgeber): *The Dynamics of Vehicles on Roads and on Tracks. Proceedings of the 16th IAVSD Symposium held in Pretoria, South Africa*, Band 33, Seiten 231–243. Swets & Zeitlinger, 1999.

A. Anhang

A.1. Modellierungsrichtlinien

Im Folgenden werden die Richtlinien zusammengefasst, nach denen sich ein Ingenieur bei der Erstellung von Simulink- oder Modelica-Modellen richten muss, damit die Modelle in SMP2 eingebettet werden können. Die Richtlinien werden als Liste angegeben. In Klammern ist die Instanz vermerkt, welche die Einschränkung bewirkt.

A.1.1. Simulink-Modelle

- Keine algebraischen Schleifen verwendbar (RTW). Ausnahme: Schleifen über "triggered subsystems" und Schleifen zum Reset-Port eines Integrators
- Keine Blöcke "MATLAB function" und "S-function" verwendbar (RTW). Ausnahme: Die Blöcke rufen keine M-Quelltexte auf.
- Keine ToWorkspace-Blöcke verwendbar (RTW).
- Keine Scopes verwendbar (MOSAIC).
- Alle Ein-/Ausgänge müssen mit Ports verknüpft sein (MOSAIC).
- Als "Target Language Compiler" muss "Generic Real-Time Target with Dynamic Memory" eingestellt werden (MOSAIC).
- Es muss ein Lösungsverfahren mit fester Schrittweite aus ode1 bis ode5 gewählt werden (MOSAIC).
- Der Modellname darf nicht länger als 21 Zeichen sein (MOSAIC).
- Modellnamen dürfen nicht wie Standard-C++-Bezeichner lauten (MOSAIC).
- Ein-/Ausgangsports dürfen nur die Signaltypen *double*, *uint8* oder Vektorsignale fester Länge besitzen (Einbettungsprozess).

A.1.2. Modelica-Modelle

- Ein-/Ausgangsvariablen dürfen nur den Modelica-Datentyp RealInput oder RealOutput besitzen (Einbettungsprozess). Sonst werden sie nicht nach SMP2 übertragen.
- Es muss das DASSL-Lösungsverfahren verwendet werden (Einbettungsprozess).
- Es müssen im Modell alle Parameter gesetzt werden, weil sie später nicht geändert werden können (Einbettungsprozess).

A.2. SMP2-Dokumente für Beispielmodell in Simulation

Für das Modelica-Modell des Batterieladestandes aus Abbildung 20 ist hier das Catalogue-Dokument der SMP2-Einbettung abgebildet. Es sind mehrere Teile ausgelassen worden, um die Übersicht zu wahren. Erkennbar ist der Hauptknoten, der das Catalogue-Dokument definiert. Darin eingebettet ist ein Namespace-Knoten, darin wiederum einbettet ein Type-Knoten, der das SMP2-Modell der Batterie definiert. Das SMP2-Modell beinhaltet die Angabe einer UUID, eines Basis-Interfaces und die Definition eines Entrypoints `Modelica_battery_update` und zweier Field-Variablen `u` sowie `y`.

```
<?xml version="1.0" encoding="UTF-8"?>
<Catalogue:Catalogue
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:Catalogue="http://www.esa.int/2005/10/Smdl/Catalogue"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    Name="battery">
  <Namespace Id="battery_Namespace" Name="battery">
    <Type xsi:type="Catalogue:Model" Id="battery" Name="battery">
      <Uuid>20c1d187-91a2-3600-9f5d-f7867d4e6bf5</Uuid>
      <Interface xlink:title="Interface IModel"
                 xlink:href="http://www.esa.int/2005/10/Smp#Smp.IModel"/>
      <EntryPoint Id="Modelica_battery_update"
                  Name="Modelica_battery_update"/>
      <Field Id="Modelica_battery_u" Name="u" Output="true">
        <Type xlink:title="PrimitiveType Float64"
              xlink:href="http://www.esa.int/2005/10/Smp#Float64"/>
      </Field>
      <Field Id="Modelica_battery_y" Name="y" Input="true">
        <Type xlink:title="PrimitiveType Float64"
              xlink:href="http://www.esa.int/2005/10/Smp#Float64"/>
      </Field>
    </Type>
  </Namespace>
</Catalogue:Catalogue>
```

Als Beispiel für generierten Code ist im Folgenden der Inhalt der aus dem Catalogue generierten C++-Header-Datei abgebildet, in der die Elemente des voranstehenden Catalogue-Dokuments auf Quelltext-Ebene definiert werden. So finden sich die Definition des Entrypoints `Modelica_battery_update` und der Field-Variablen `u` und `y` wieder. Es ist ersichtlich, dass die UUID, mit der die Implementierung später referenziert wird, im Catalogue wie im Quelltext identisch ist. Ebenso werden die Modell-Simulator-Interaktions-Funktionen `Publish`, `Configure` und `Connect` definiert, die der C++-Klasse `IModel` de-

klariert sind und überschrieben werden müssen. Auch im Quelltext wurden der Übersichtlichkeit wegen Teile ausgelassen.

```
namespace battery
{
    static const ::Smp::Mdk::Uuid Uuid_battery
                 ("20c1d187-91a2-3600-9f5d-f7867d4e6bf5");
    class battery:
          virtual public ::Smp::Mdk::Management::ManagedModel,
          virtual public ::Smp::Mdk::Management::EntryPointPublisher,
          virtual public ::Smp::IModel
    {
        // Constructors/Destructor
        battery(void);
        battery( ::Smp::String8 name,
                 ::Smp::String8 description,
                 ::Smp::IComposite* parent )
               throw (::Smp::InvalidObjectName);
        virtual ~battery(void);

        // Fields
        ::Smp::Float64 u;
        ::Smp::Float64 y;

        // Entry Points
        ::Smp::IEntryPoint *Modelica_battery_update;

        // Entry Point Handlers
        void _Modelica_battery_update();

        // IModel
        void Publish( ::Smp::IPublication *receiver )
                    throw ( ::Smp::IModel::InvalidModelState );
        void Configure( ::Smp::Services::ILogger* logger )
                      throw ( ::Smp::IModel::InvalidModelState );
        void Connect( ::Smp::ISimulator *simulator )
                    throw ( ::Smp::IModel::InvalidModelState );
    };
}
```

Das Assembly-Dokument beschreibt eine Instanzierung von SMP2-Modellen, die zur Simulation genutzt werden. Es werden ihre Parameter gesetzt und gegebenenfalls Field-Links definiert. Im gezeigten Assembly wird eine Modellinstanz mit Namen *battery* erzeugt. Sie enthält die Angabe von Startwerten für ihre zwei Field-Variablen `u` und `y`.

Anschließend wird ein Field-Link definiert, mit der Eingangsvariablen u der Batterie als Ziel und der Ausgangsvariablen y einer anderen Instanz namens “sine_source” als Quelle. Die Definition der Instanz “sine_source” wird angedeutet, ist aber ebenso wie andere Teile des Assemblys ausgelassen worden.

```
<?xml version="1.0" encoding="UTF-8"?>
<Assembly:Assembly
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:Assembly="http://www.esa.int/2005/10/Smdl/Assembly"
    xmlns:Types="http://www.esa.int/2005/10/Core/Types"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Model xlink:title="System" xlink:role="Model"
      xlink:href="Container.cat#ID_f0597721-4398-455e-8e7d-8a32cf93c7e4"/>
  <Implementation>f0597721-4398-455e-8e7d-8a32cf93c7e4</Implementation>

  <ModelInstance Id="ID_ceb44b00-1c4d-455e-8cae-7373b33917f6"
                 Name="battery">
    <Model xlink:title="battery" xlink:role="Model"
           xlink:href="battery.cat#battery"/>
    <Implementation>20c1d187-91a2-3600-9f5d-f7867d4e6bf5</Implementation>

    <FieldValue>
      <Field xlink:title="u" xlink:href="battery.cat#Modelica_battery_u"/>
      <Value xsi:type="Types:SimpleValue">
        <Value xsi:type="xsd:double">0.0</Value>
      </Value>
    </FieldValue>

    <FieldValue>
      <Field xlink:title="y" xlink:href="battery.cat#Modelica_battery_y"/>
      <Value xsi:type="Types:SimpleValue">
        <Value xsi:type="xsd:double">0.0</Value>
      </Value>
    </FieldValue>

    <Link xsi:type="Assembly:FieldLink"
          Id="ID_44879b9d-7212-47ea-99bb-b3d33a8b60cc"
          Name="get_source">
      <Description></Description>
      <Input xlink:title="u" xlink:role="Input"
             xlink:href="battery.cat#Modelica_battery_u"/>
      <Source xlink:title="ModelInstance sine_source"
              xlink:href="#ID_598e1543-cf9e-4f6b-9502-6df2bb8cf1a8"/>
```

```
      <Output xlink:title="y" xlink:role="Output"
              xlink:href="sine_source.cat#Modelica_sine_source_y"/>
    </Link>

    <Container xlink:title="SystemContainer" xlink:role="Container"
      xlink:href="Container.cat#ID_dab55c83-29f7-4b0e-b966-23f8cadac453"/>
  </ModelInstance>

  <ModelInstance Id="ID_598e1543-cf9e-4f6b-9502-6df2bb8cf1a8"
                 Name="sine_source" ...>
    [...]
  </ModelInstance>
</Assembly:Assembly>
```

Im Folgenden ein zur Simulation verwendetes Schedule-Dokument. Darin werden zunächst drei Aktivitäten definiert. Zuerst wird der Entrypoint des Sinus-Modells aufgerufen und das Modell damit für einen Zeitschritt aktualisiert. Dann wird der Field-Link ausgeführt, d.h. der Werte der Sinus-Ausgangsvariablen auf die Batterie-Eingangsvariable übertragen. Anschließend wird das Batterie-Modell über seinen Entrypoint aktualisiert. Es schließt sich die Definition eines Events an. Darin werden alle 0,001 s die drei Aktivitäten, gruppiert über ihren SMP2-Task, ausgeführt.

```
<?xml version="1.0" encoding="UTF-8"?>
<Schedule:Schedule
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:Schedule="http://www.esa.int/2005/10/Smdl/Schedule"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <Task Id="ID_939be25b-e5a4-457d-8b27-7883d0de34f7" Name="PerformStep">
    <Activity xsi:type="Schedule:Trigger"
              Id="ID_c2ee243d-6362-4a88-9160-baf4c8b8b223"
              Name="update_sine">
      <Provider xlink:title="sine_source" xlink:role="Provider"
                xlink:href=
        "om_battery_system.asb#ID_598e1543-cf9e-4f6b-9502-6df2bb8cf1a8"/>
      <EntryPoint xlink:title="Modelica_sine_source_update"
              xlink:role="Entry_Point"
              xlink:href="sine_source.cat#Modelica_sine_source_update"/>
    </Activity>

    <Activity xsi:type="Schedule:Transfer" Name="transfer_input"
              Id="ID_b28e02f2-27fc-4f39-98df-8d3f879a31b4">
      <FieldLink xlink:title="get_source" xlink:role="Field_Link"
             xlink:href=
        "om_battery_system.asb#ID_44879b9d-7212-47ea-99bb-b3d33a8b60cc"/>
```

```
    </Activity>

    <Activity xsi:type="Schedule:Trigger"
              Id="ID_9136b0de-efb0-463a-9c7c-e75aa299bef6"
              Name="update_battery">
      <Provider xlink:title="battery" xlink:role="Provider"
                xlink:href=
        "om_battery_system.asb#ID_ceb44b00-1c4d-455e-8cae-7373b33917f6"/>
      <EntryPoint xlink:title="Modelica_battery_update"
                  xlink:role="Entry_Point"
                  xlink:href="battery.cat#Modelica_battery_update"/>
    </Activity>
  </Task>

  <Event xsi:type="Schedule:SimulationEvent"
         Id="ID_98236433-2073-408c-b826-e4870e430191"
         Name="SimulationStep" SimulationTime="PT0.001S">
    <CycleTime>PT0.001S</CycleTime>
    <Task xlink:title="PerformStep" xlink:role="Task"
          xlink:href="#ID_939be25b-e5a4-457d-8b27-7883d0de34f7"/>
  </Event>
</Schedule:Schedule>
```

A.3. Variable Schrittweitensteuerung

In diesem Abschnitt werden Möglichkeiten aufgezeigt, SMP2-Modelle auszuführen, deren Lösungsalgorithmen mit variabler Schrittweite arbeiten. Die Möglichkeiten können für diesen Zweck jedoch nicht auf SMP2-Mechanismen aufbauen, sondern verlagern die Last der Ausführung auf den SMP2-Programmierer.

A.3.1. Simulator und Schedule

Ereignisse können mit fester Schrittweite im Schedule vereinbart werden. Der Simulator, der allein die Simulationszeit beeinflussen kann, liest den Schedule und führt nach dem darin vorgegebenen Schema Entrypoints aus.

Rechnet eine Komponente mit variablen Schritten, ist vor Ausführung eines Simulationsschrittes die Weite desselben unbekannt. Deshalb müsste im Nachhinein die Simulationszeit des Simulators angepasst werden können. SMP2 bietet aber keine Möglichkeiten dazu [ESO05b]. Aber es ist möglich, zur Laufzeit neue Simulationszeitereignisse zu definieren. Nach Berechnung des Simulationsschritts einer Komponente und vor der anschließenden Rückkehr an den Simulator könnte ein einmaliges Ereignis definiert werden, das so weit in der simulierten Zukunft liegt, wie der Zeitschritt, der ja nun bekannt ist. So könnte man sich von Schritt zu Schritt mit dynamisch definierten Ereignissen hangeln. Die Simulationszeit des Simulators hinkt dabei immer einen Schritt zurück, führt aber keine Ergebnisverfälschung ein.

A.3.2. Komponenten

Rechnen zwei Komponenten mit variabler Schrittweite, wird ohne Einschränkung der Allgemeinheit eine der beiden den nächsten Simulationsschritt früher als die andere ansetzen. Wenn die Komponenten zudem untereinander Werte austauschen, muss auch die andere Komponente für den früheren Simulationsschritt aktualisiert werden. Daher muss es für Komponenten eine Möglichkeit geben, die Zeit ihres nächsten zu berechnenden Schrittes von außerhalb dieser Komponente festzulegen. Dies ist nicht mit MOSAIC-Komponenten möglich, da sie RTW-Code fester Schrittweite benötigen. Es wäre vielleicht möglich, über neue Entrypoints den noch enthaltenen RTW-Code anzusteuern und dort die Simulationszeit zu ändern. Bei OpenModelica-Komponenten könnnte man ähnlich vorgehen. In beiden Fällen sind jedoch tiefgreifende Kenntnisse über die Berechnung und Handhabung im jeweiligen Simulationscode nötig.

A.4. Simulink-Implementierung des Fischfang-Modells

Im Folgenden wird die Implementierung des Simulink-Modells aus Abbildung 24 beleuchtet. Dazu wird die Implementierung der einzelnen Teilmodelle z_1, z_2 und *Fangmenge* gezeigt. Das Teilmodell Fangmenge besteht wiederum aus zwei Teilmodellen, die ebenfalls gezeigt werden. Im Anschluss wird die Konfiguration von MATLAB-Variablen gezeigt, die für die Ausführung der Simulink-Modelle erforderlich ist.

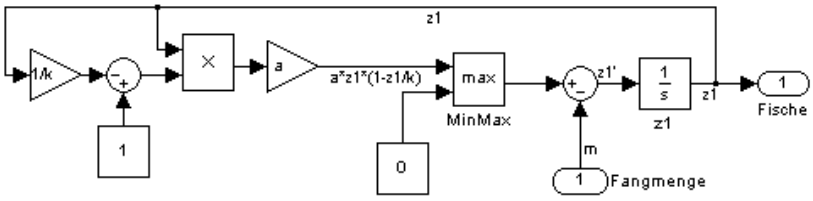


Abbildung 32: Simulink-Implementierung der Fischpopulation

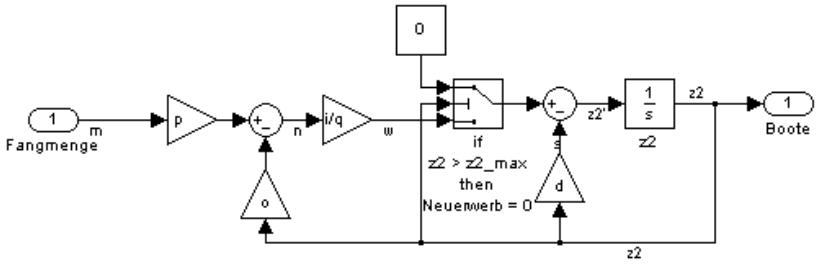


Abbildung 33: Simulink-Implementierung der Bootsanzahl

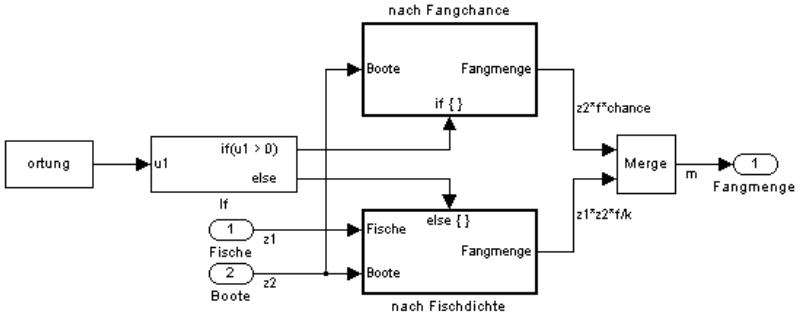
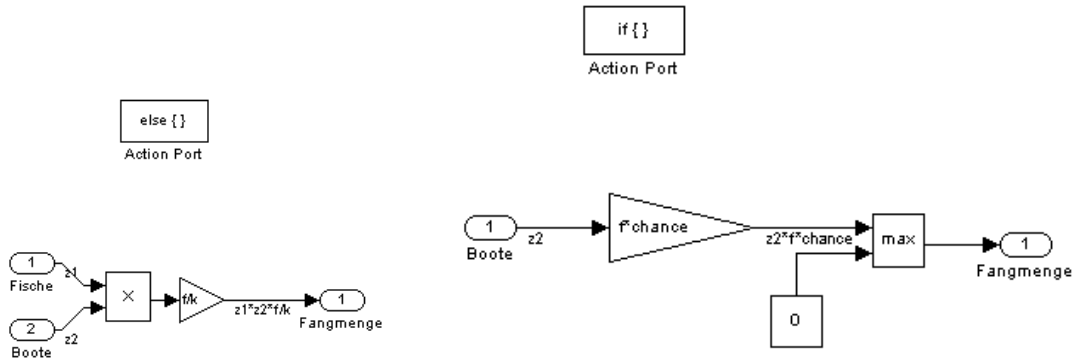


Abbildung 34: Simulink-Implementierung der Fangmenge



(a) Fangmenge nach Fischdichte

(b) Fangmenge nach Fangchance

Abbildung 35: Tiefergehende Simulink-Implementierung der Fangmenge

Konfiguration:

```
Ar = 100;     % Fanggebiet [km^2]
Ks = 100;     % spez. Fischkapazität [t Fisch/km^2]

a = 1;        % max. spez. Fischzuwachsrate [1/Jahr]
di = 15;      % Bootslebensdauer [Jahre]
f = 100;      % max. spez. Fangmenge je Boot und Jahr [t Fisch/(Boot*Jahr)]
i = 0.5;      % Anteil der Investitionsmittel für Bootsbeschaffung []
o = 50000;    % spez. Bootsunterhaltskosten [EUR/(Boot*Jahr)]
p = 1000;     % Fischpreis [EUR/t Fisch]
q = 100000;   % Bootsneukosten [EUR/Boot]

z1_0 = 5000; % Startpopulation Fische [t Fisch]
z2_0 = 100;  % Startanzahl Boote [Boot]


k = Ar * Ks; % max. Fischkapazität [t Fisch]
d = 1/di;    % spez. Stilllegungsrate der Boote [1/Jahr]

ortung = 0;    % Wird Ortungstechnik eingesetzt? [{0, 1}]
chance = 0.8;  % Fangchance der Ortungstechnik []
z2_max = 1000; % maximale Bootszahl [Boot]
```

A.5. Beschränkung von Runge-Kutta-Verfahren bei SMP2-Kopplung

Ohne Beschränkung der Allgemeinheit wird die Herleitung an einem Runge-Kutta-Verfahren der Ordnung 4 durchgeführt. Die Formel nach [Fau99] lautet:

$k_1 = hf(x_n, y_n)$

$k_2 = hf(x_n + c_2 h, y_n + a_{21} k_1)$

$k_3 = hf(x_n + c_3 h, y_n + a_{31} k_1 + a_{32} k_2)$

$k_4 = hf(x_n + c_4 h, y_n + a_{41} k_1 + a_{42} k_2 + a_{43} k_3)$

$y_{n+1} = y_n + w_1 k_1 + w_2 k_2 + w_3 k_3 + w_4 k_4$

Die Differentialgleichung $y' = f(x, y)$ kann x und y als Variablen enthalten. Es sei angenommen, dass sie zusätzlich noch eine Variable u enthält. u sei der Vektor aller Eingangsvariablen des Modells. Die Repräsentation $f(x, y)$ enthält nur die Argumente x und y, da sie bei der Berechnung des Runge-Kutta-Verfahrens eine Rolle spielen. u ist während der Berechnung eines Schrittes immer konstant.

Es werden drei Fälle unterschieden:

- f(x, y) ist nur von der Eingangsfunktion u abhängig, nicht aber von x oder y. Z.B. $f(x, y) = 3u - 5$.
- f(x, y) ist von x, y oder beiden, aber nicht von u abhängig. Z.B. $f(x, y) = y - x$.
- f(x, y) ist von x, y oder beiden, und von u abhängig. Z.B. $f(x, y) = \pi x u$.

Für den ersten Fall bestimmt sich die Ableitungsfunktion f einzig aus dem Modelleingang u. Dieser wird nur zu Hauptschritten aktualisiert und ist deshalb während aller Teilschritte des Runge-Kutta-Verfahrens konstant. Damit ist auch die Ableitungsfunktion f für alle Argumente konstant. Daher gilt für alle vorkommenden Argumente d, e von f:

$f(d, e) = f(x_n, y_n)$.

Somit gilt:

$k_1 = k_2 = k_3 = k_4$, und

$y_{n+1} = y_n + hf(x_n, y_n) * \sum_{i=1}^{j} w_i$.

Die Summe der Wichtungsparameter w_i ist immer eins:

$\sum_{i=1}^{j} w_i = 1$.

Daher ergibt sich:

$y_{n+1} = y_n + hf(x_n, y_n)$.

Dies ist die Berechnungsformel für ein einfaches Euler-Verfahren. Für den ersten Fall rechnen alle Runge-Kutta-Verfahren also so genau wie ein Euler-Verfahren. Das Batterie-Modell dieser Arbeit ist ein Beispiel für Fall 1. Am Eingang des Integrators liegt ein Signal, das ausschließlich aus dem Modelleingang (ein Sinus-Signal) berechnet wird. Weder

wird der Integratorwert selbst zur Berechnung einbezogen (was y entspräche), oder die Simulationszeit (was x in obiger Formel entspräche).

Für den zweiten Fall bestimmt sich die Ableitungsfunktion einzig aus x und y und aus keinen Modelleingängen. Die Teilschritte verlaufen wie geplant durch Variation von x und y. Daher kann das Runge-Kutta-Verfahren hier effizient arbeiten. Modelle, die Fall 2 erfüllen, kapseln die Berechnung so gut wie möglich. Es kann erwartet werden, dass die Ergebnisse zu denen einer Simulink- oder OpenModelica-Simulation identisch sind.

Für den dritten Fall lassen sich keine genauen Angaben über die Genauigkeit eingesetzter Runge-Kutta-Verfahren treffen. Einerseits hängt die Ableitungsfunktion von x und/oder y ab. Deshalb kann das Verfahren wie beabsichtigt Werte variieren, um Teillösungen zu berechnen und diese schließlich zu wichten. Andererseits hängt die Ableitungsfunktion auch von den Eingängen u ab.

Das Problem hierbei ist, dass über Verknüpfungen außerhalb des Modells, welches das Runge-Kutta-Verfahren einsetzt, der berechnete Integratorwert oder die Simulationszeit wieder in die Berechnung des Modelleingangs einfließen könnten. Mit anderen Worten könnte u durch externe Verknüpfungen selbst wieder von x und y abhängig sein. Nun ergibt sich das Problem, dass innerhalb des Modells aktuelle Werte von x und y zur Teilschrittberechnung gebildet werden können. Die Modelleingänge bleiben jedoch konstant, und damit auch der Anteil von x und y, der aus externen Verknüpfungen zugeführt wird. Teilweise rechnet das Runge-Kutta-Verfahren also mit variablen x und y, teilweise mit konstanten x und y während der Teilschritte.

Die Ergebnisse können daher nicht genau eingeschätzt werden. Die Ergebnisse sind bestenfalls so genau wie das verwendete Runge-Kutta-Verfahren. Sie können aber auch nur so genau wie ein Euler-Verfahren sein, oder auf einer Skala zwischen der Genauigkeit dieser beider Verfahren. Es kann in Folgearbeiten untersucht werden, ob noch schlechtere Ergebnisse als mit einem Euler-Verfahren möglich sind.

Das Fischfang-Modell dieser Arbeit ist ein Beispiel für Fall 3. Die Berechnung der Differentialfunktionen z_1', welche die Änderung der Fischpopulation beschreibt hängt innerhalb des Modells von $y = z_1$, also dem Integratorwert selbst ab. Außerhalb des Modells fließt der Integratorwert z_1 jedoch auch wieder über die Berechnung der Fangmenge in das Modell ein.

A.6. MATLAB-Skript-Template zur Codeerzeugung mit Real-Time Workshop

```
% MATLAB script that configures Real-Time Workshop and builds the model code
%  for the model $model.

% Expects these variables to be inserted into the template by markers
% prefixed with the Dollar symbol ('\$'):
%
% model .. the model name
% mpath .. the path where the model lies
% lpath .. the libraries root path containing the setPaths.m script file
% wpath .. the path where the RTW conversion should take place
% config .. the config MATLAB script function; to be found in the model path


% if lpath template variable has been resolved and substituted
if (~strcmp('$lpath',strcat('\$','lpath'))) && (~strcmp('$lpath','null'))
    % go to library root
    cd '$lpath'

    % set library references through function setPaths
    'Setting library paths up from root $lpath'
    setPaths('$lpath')
else
    'No library path will be set.'
end

% change to model directory
cd '$mpath'

% if config template variable has been resolved and substituted
if (~strcmp('$config',strcat('\$','config'))) &&
   (~strcmp('$config','null') &&
   (length('$config')~=0))
    % call config script
    % config script may call clear; thus, instanciate
    % all local variables only after this call!
    'Config file $config will be used.'
    $config
else
    'No config file will be used.'
end
```

```
% model name
% local variables must be instanciated after the config script call
model = '$model'

% open simulink model
open_system(model)

% Solver settings
set_param(model, 'Solver', 'ode5')
set_param(model, 'StopTime', '40.0')
set_param(model, 'FixedStep', '0.005')

% Optimization settings
set_param(model, 'BlockReduction', 'off')

% Real-Time Workshop settings
set_param(model, 'SystemTargetFile', 'grt_malloc.tlc')
set_param(model, 'GenCodeOnly', 'On')
set_param(model, 'TargetLang', 'C')

% change to working directory
cd '$wpath'

% trigger build process
rtwbuild(model)

% close simulink model and discard changes
close_system(model, 0)

% quit matlab
exit
```

Selbständigkeitserklärung

Ich habe die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt.

Berlin, den

Unterschrift

Einverständniserklärung

Die vorliegende Arbeit darf in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin ausgestellt werden.

Berlin, den

Unterschrift