Vergleichende Betrachtung von SimScape und OpenModelica anhand von ingenieurtechnischen Standardbeispielen

Wissenschaftliche Projektarbeit

Hochschule Wismar
Fakultät für Ingenieurwissenschaften

Vorgelegt von:	Steffen Podelleck (114098)
Betreuer und Prüfer:	Prof. Dr.-Ing. Thorsten Pawletta
Vorgelegt am:	24. August 2015

Erklärung

Ich erkläre hiermit, daß ich diese Arbeit selbständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

Ich bin mir bewußt, daß eine falsche Erklärung rechtliche Folgen haben wird.

Wismar, 24. August 2015

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

In dieser Arbeit sollen die zwei Software-Tools SimScape und OpenModelica miteinander verglichen werden. Beide Werkzeuge dienen der objektorientierten Modellbildung und Simulation von Systemen. Beim Vergleich soll vor Allem auf Unterschiede in den grundlegenden Ansätzen der objektorientierten Modellbildung geachtet werden. Dazu zählen Unterschiede in den Definitionen grundlegender Begriffe wie dem des Systems und syntaktische Unterschiede wie beispielsweise die Handhabung wichtiger Variablen.

Die Unterschiede werden anhand einfacher Standardbeispiel aus den Bereichen der Mathematik und Physik verdeutlicht und erklärt.

2 Einführung in wesentliche Begriffe und zu untersuchende Software-Tools

In diesem Kapitel soll zum Verständnis auf grundlegende Begriffe der Modellbildung und Simulation eingegangen werden. Zudem werden die zu vergleichenden Software-Tools kurz vorgestellt.

2.1 Begriffsklärung

Wie bereits angekündigt werden im Folgenden einige Grundbegriffe der Modellbildung und Simulation erläutert.

System

In [Fri11] beschreibt Fritzson ein System als ein Objekt oder eine Ansammlung von Objekten, dessen oder deren Eigenschaften untersucht werden sollen. Als Beispiele für Systeme nennt er ingenieurtechnische Erzeugnisse wie Space Shuttles oder das Universum, Sekundenbruchteile nach dem Urknall. Das sind zum Einen künstliche vom Menschen erschaffene Systeme und zum Anderen die Natur oder Teile von ihr.

Unterschiedliche Systeme können aber auch miteinander verbunden werden. In dem Zusammenhang nennt Fritzson hier das in Abbildung 2.1 dargestellte Wohnhaus, das mit einer künstlichen Solarheizung Warmwasser erzeugt. Sonne und Wolken sind ebenfalls in einem derartigen System zu beachten und daher im Bild dargestellt: Ist die Sonne beispielsweise von Wolken verdeckt, steht sie nicht als Wärmequelle zur Verfügung und das Wasser muß elektrisch beheizt werden.

Signalflußmodellierung

Bei der Signalflußmodellierung geht es gemäß [Myr12] um die Modellierung des Datenaustausches zwischen den Funktionen eines Systems und dessen Umwelt. Der Signalflußmodellierung wird dabei als zentraler Technik der kausalen Analyse große Bedeutung beigemessen, wobei sie auch in anderen Methoden relevant ist.

Signalflußdiagramm

Das Signalflußdiagramm ist ein wichtiges Werkzege der signalflußorientierten Modellierung. Nach [Myr12] ist es mit Hilfe verschieden detaillierter Versionen eines derartigen Diagramms möglich, Datenflüße zwischen einem System und seiner Umgebung

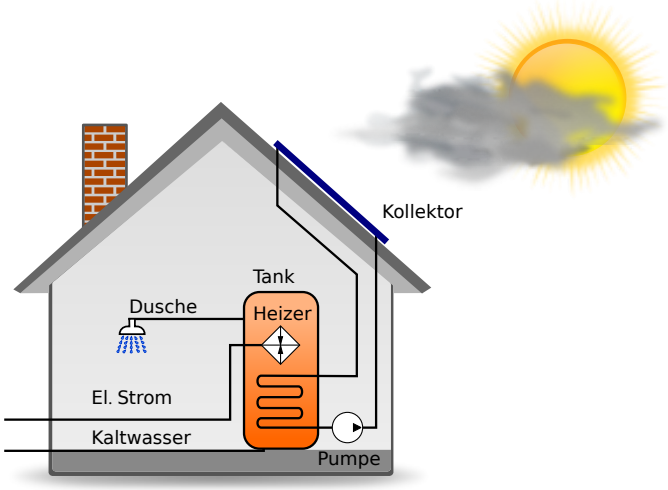


Abbildung 2.1: Ein Haus mit solarbeheiztem Warmwasser in Anlehnung an [Fri11]

oder zwischen Teilfunktionen inerhalb des Systems darzustellen und ein Modell des Systems zu erhalten. Abbildung 2.2 zeigt zwei Versionen eines Signalflußdiagramms.

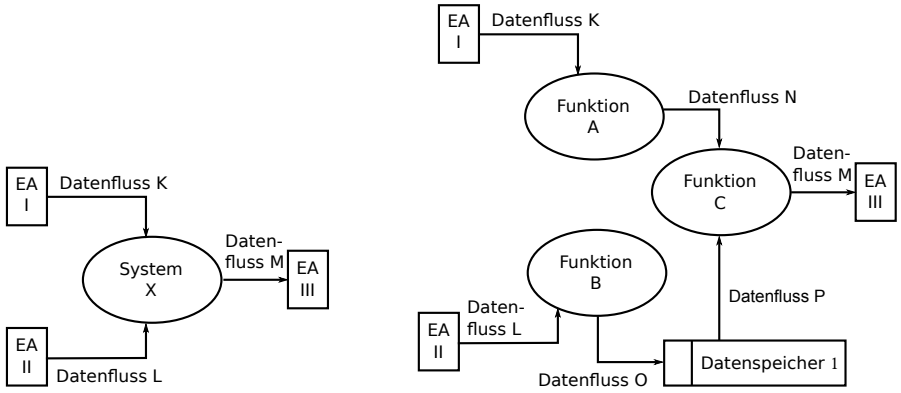
Objektorientierte Modellierung

Nach [Fet12] bildet die objektorientierte Modellierung einen Ansatz zur Analyse und Entwicklung von Systemen, der im Wesentlichen auf den Konzepten Objekt, Klasse und Vererbung beruht. Alle folgenden nicht separat gekennzeichneten Begriffsklärungen dieses Abschinitts wurden sinngemäß [Fet12] entnommen.

Ojekte

Fettke stellt Objekte als das zentrale Konzept der objektorientierten Modellierung dar. Jedes Objekt weist dabei sowohl strukturelle als auch verhaltensbezogene Eigenschaften auf. Betrachtet man zum Beispiel das Objekt Toaster, zählen sein Energieverbrauch und die Geschwindigkeit, mit der er ein Stück Brot röstet, zu seinen strukturellen Eigenschaften. Das Eintauchen der Brotscheiben in die Röstkammer und das Regulieren der Wärme an einem Drehgriff sind Dienste, die das Objekt Toaster anbietet, und damit verhaltensbezogene Eigenschaften.

Das Objekt ist ein abstraktes Konzept, das sich nicht nur auf physische Gegenstände anwenden läßt. In der Implementierungstechnik können Applikationen oder Betriebssystemprozesse und im Bereich der Wirtschaftswissenschften Konten oder Verträge ebenfalls als Objekte betrachtet werden.



(a) Signalflußdiagramm (b) Verfeinertes Signalflußdiagramm

Abbildung 2.2: Beispiel eines Signalflußdiagramms in Anlehnung an [Myr12]

Klassen

Klassen dienen dazu, eine Menge von Objekten mit ähnlichen Eigenschaften zu sammeln. Objekteigenschaften, die alle Objekte der Klasse aufweisen, müssen so nur einmal in der Klasse definiert werden und werden dann von allen Objekten der Klasse geleichermaßen übernommen.

Vererbung

Besteht eine gerichtete Beziehung zwischen zwei Klassen, bei der die untergeordnete Klasse Eigenschaften der übergeordeneten Klasse übernimmt, spricht man von Vererbung. Dabei kann die untergeordnete Klasse alle Eigenschaften der übergeordneten Klasse übernehmen, sie überschreiben oder durch weitere Eigenschaften ergänzen.

Verkapselung

Die innere Struktur eines Objekts ist weder für andere Ojekte ersichtlich, noch können sie die innere Sturktur direkt beeinflußen. Es kann nur auf die innere Struktur eines Objekts über die von dem Objekt zur Verfügung gestellten Dienste (verhaltensbezogene Eigenschaften) Einfluß genommen werden.

Nachrichten

Nachrichten dienen der Interaktion zwischen unterschiedlichen Objekten. Über eine Nachricht kann ein Objekt auf die Verhaltenseigenschaft beziehungsweise auf den Dienst eines anderen Objekts zugegriffen werden. Versteht das Zielobjekt die übermittelte Nachricht, wird der entsprechende Dienst ausgeführt.

Polymorphismus

Polymorphismus beschreibt die kontextabhängige Interpretation von Nachrichten beziehungsweise Anweisungen. So kann eine Nachricht an unterschiedliche Objekte gesendet werden, die dann von den einzelnen Objekten abhängig ihrer Klassenzugehörigkeit unterschiedlich interpretiert werden.

Potential- und Flußgrößen

In der objektorientierten Simulation technischer und pysikalischer Systeme ist es wichtig zu verstehen, daß zwischen den einzelnen Objekten eines Modells keine gerichteten Signalübertragungen stattfinden, sondern vielmehr ein Austausch von physikalischer Leistung P. Nach [Mat15] sind jedem Objekt an seinen Verbindungselementen jeweils zwei Variablen zugewiesen, die sogenannten Potential- und Flußgrößen. MATHWORKS bezeichnet diese als *Across*- und *Through*-Variablen. Das Produkt der beiden Größen ergibt immer die zwischen den Objekten übertagene Leistung P. In der Elektrotechnik setzt sich die Leistung P aus der Spannung U und dem elektrischen Strom I und in der Mechanik beispielsweise aus der Geschwindigkeit v und der Kraft F zusammen. Durch den Austausch von Nachrichten zwischen Objekten in Form einer allgemeinen Größe wie der Leistung, wird in der objektorientierten Modellbildung der zuvor beschriebene Polymorphismus erreicht.

Tabelle 2.1: Gängige Potential- und Flußgrößen gemäß [Mat15]

Disziplin	**Potentialgröße**	**Flußgröße**
Elektrotechnik	Spannung U	el. Strom I
Mechanik	Geschwindigkeit v	Kraft F
	Winkelgeschwindigkeit ω	Drehmoment M
Thermodynamik	Temperatur t	Wärmestrom $\dot{Q}$
Hydraulik	Druck p	Massestrom $\dot{m}$

2.2 Softwaretools

Nachdem die Einführung in die grundlegenden Begriffe abgeschloßen ist, folgt nun eine kurze Vorstellung der zu vergleichenden Softwaretools.

SimScape

SIMSCAPE ist nach [Mat15] eine Erweiterung für das von MATHWORKS vertriebene Entwicklungwerkzeug MATLAB/SIMULINK zur objektorientierten Modellierung und Simulation technischer Systeme. Es bietet grundlegende Blöcke zur Modellierung von beispielsweise elektrischen Motoren oder hydraulischen Ventilen und die Möglichkeit mit der auf MATLAB basierenden SIMSCAPE-Programmiersprache neue Blöcke zu entwickeln. Das Werkzeug ist eine einzelne Entwicklungsumgebung zur

Modellierung und Simulation fachübergreifender pysikalischer Systeme mit Hilfe von Modell-Blöcken, die die einzelnen Systemkomponenten repräsentieren. Die Integration in MATLAB ermöglicht eine Parametrisierung über MATLAB-Variablen.

OpenModelica

OPENMODELICA ist gemäß [OMW15] eine vom OPEN SOURCE MODELICA CONSORTIUM unterstützte freie Simulationssofware. Sie basiert auf der MODELICA-Sprache und damit auf dem Prinzip der objektorientierten akausalen Modellierung von Systemen. Das langfristige Ziel ist es ein freies Werkzeug zu Forschungs- und Bildungszwecken der Industrie und Bildungseinrichtungen zur Verfügung zustellen.

Die MODELICA-Sprache stellt sowohl numerische ODE-Solver für das Lösen von Differentialgleichungen als auch DAE-Solver für das Lösen differentiell-algebraischer Gleichungssysteme zur Verfügung.

3 Vergleich der kausalen mit der akausalen Modellierung

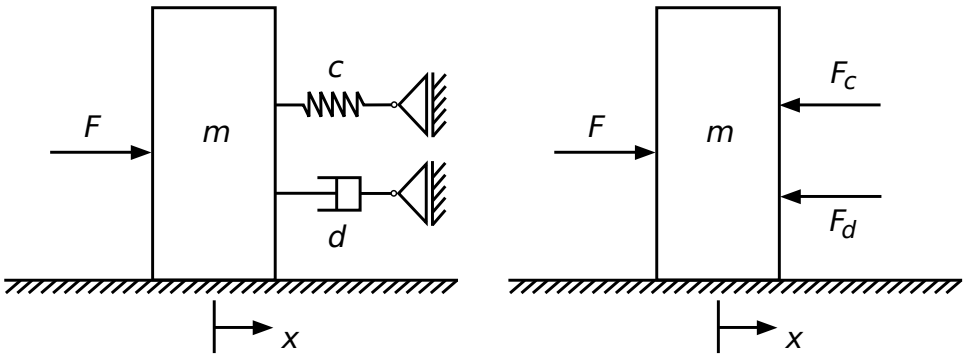
In diesem Kapitel sollen die objektorientierte und die signalflußorientierte Modellierung technischer Systeme anhand des Beipiels eines Ferder-Masse-Dämpfer Systems dargestellt werden. Dafür soll das System zunächst als Differentialgleichung und anschließend als Signalflußbild in Simulink umgesetzt werden. Danach wird das selbe Feder-Masse-Dämpfer System als akausales Modell in Simscape simuliert.

3.1 Modellbildung und Simulation mit Simulink

In diesem Abschnitt erfolgt die signalflußorientierte Modellbildung und Simulation eines Ferder-Masse-Dämpfer Systems mit Simulink.

Modellbildung

Abbildung 3.1 zeigt eine mit einer Feder und einem Dämpfer verbundene Masse m. Eine Kraft F wirkt auf die Mase m ein, was eine Auslenkung x bewirkt. Verluste durch Reibung bleiben dabei unberücksichtigt.



(a) Skizze (b) Freischnitt

Abbildung 3.1: Skizze eines Ferder-Masse-Dämpfer Systems

Für dieses einfache System soll jetzt die entsprechende Differentialgleichung aufgestellt werden. Dafür wird zunächst die Kräftebilanz gezogen.

$$\begin{aligned}\sum F_H = m\ddot{x} &= F - F_c - F_d \\ \ddot{x} &= \frac{F - F_c - F_d}{m} \\ &= \frac{F - c \cdot x - d \cdot \dot{x}}{m}\end{aligned}$$

mit

$$m = 0{,}5\,\mathrm{kg};\ F = 7{,}5\,\mathrm{N};\ c = 50\,\mathrm{N/m};\ d = 2{,}5\,\mathrm{Ns/m}$$

Umsetzung in Simulink

Setzt man die Differentialgleichung in Simulink um, erhält man den in Abbildung 3.2 dargestellten Signalflußgraphen. Es ist sehr gut zu erkennen, daß bei der kausalen Modellierung in einem Signalflußgraphen die Signale ausschließlich in eine Richtung fließen können. Außerdem sind für den Aufbau eines derartig simplen Bespiels bereits recht viele Simulink-Blöcke notwendig.

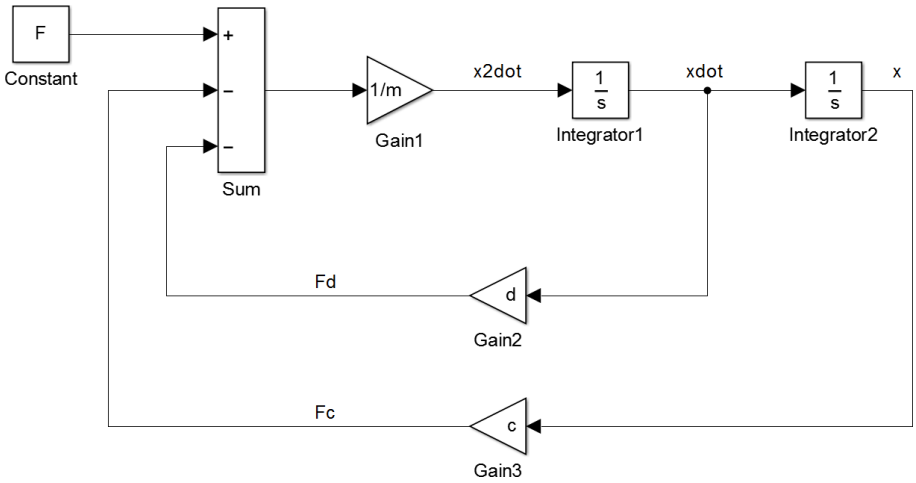


Abbildung 3.2: Signalflußgraph eines Ferder-Masse-Dämpfer Systems in Simulink

3.2 Akausale Modellierung und Simulation mit SimScape

Abbildung 3.3 zeigt das mit der SimScape Basis-Toolbox erstellte Modell des Feder-Masse-Dämpfer Systems. Der Block *Mass* beschreibt eine Masse, die eine reibungsfreie Translationsbewegung auf einer Ebene ausführt. Auf sie wirkt die von *Force* generierte Kraft. Der *Force*-Block wandelt dazu ein zeitabhängiges Signal in eine physikalishe Größe. Da SimScape in Simulink integriert ist, wäre es überflüssig, extra Signalquellen für SimScape zu programmieren. Als Signalquellen dienen die

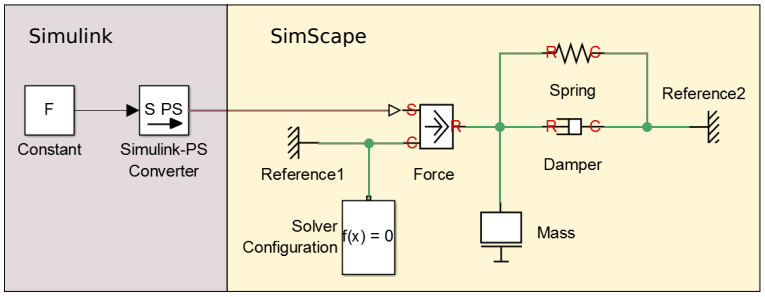


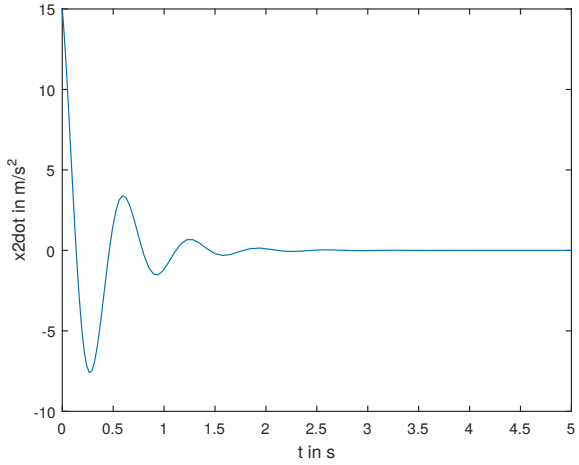
Abbildung 3.3: Modellgraph eines Ferder-Masse-Dämpfer Systems in SimScape

jeweiligen Simulink-Blöcke, deren Signale über einen *Converter*-Block zu SimScape-Signalen umgewandelt werden. In diesem Beispiel bleibt das Signal über die Zeit konstant. Die Masse ist rechts im Bild mit einer Feder (*Spring*) und einem Dämpfer (*Damper*) verbunden, die zueinander parallel geschaltet sind. Im Bild 3.3 sind zusätzlich zwei *Reference*-Blöcke zu erkennen. Gemäß [Mat15] stellen sie eine Feste Referenz im Koordinatensystm dar. Ohne Reference1 würde keine Kraft zustande kommen, die die Masse verschiebt und ohne Reference2 würden die Masse, Feder und Dämpfer nur um einen Weg x verschoben werden, da sie keine Kraft aufnehmen könnten.

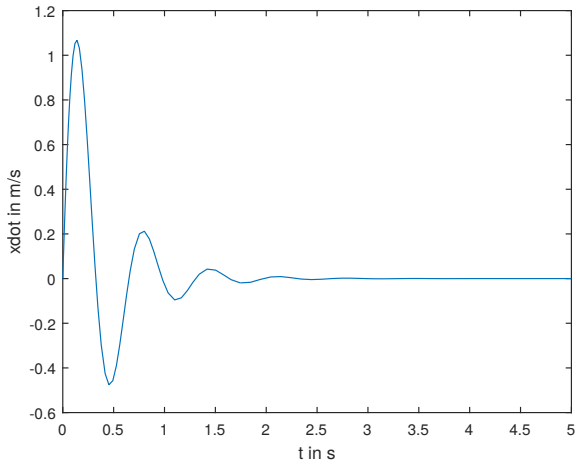
Zusätzlich findet sich im Bild ein *Solver Configuration*-Block. In ihm wird unabhängig von den Solver-Einstellungen in Simulink der SimScape-Solver eingestellt. Jedes SimScape-Modell muß mit genau einem *Solver Configuration*-Block verbunden sein. Die Position des Blocks spielt dabei keine Rolle. In diesem Beispiel können die Standard-Einstellungen beibehalten werden, weshalb Sie nicht weiter erklärt werden sollen. Nähere Information zu diesem Block sind [Mat15] zu entnehmen.

3.3 Simulationsergebnisse

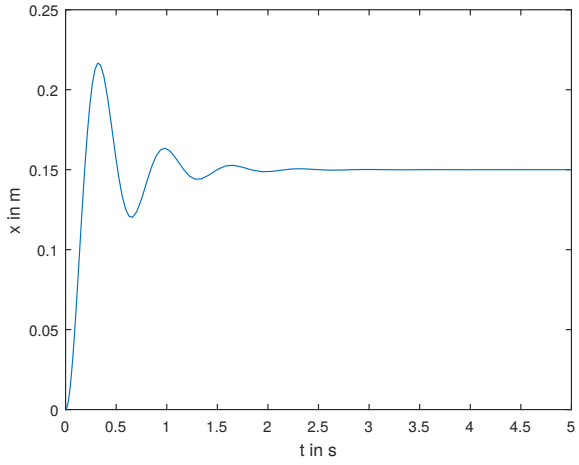
In Abbildung 3.4 sind die Ergebnisse der Simulation dargestellt. Die Simulationen mit Simulink und SimScape führten beide zu dem gleichen Ergebnis. Da es sich in diesem Beispiel um ein steifes System handelt, wurde zum Ermitteln der Lösung der *ODE23s*-Solver gewählt. Der in Simulink standardmäßig eingestellte *ODE45*-Solver hätte in diesem Fall falsche Ergebnisse geliefert.



(a) Beschleunigungsverlauf



(b) Geschwindigkeitsverlauf



(c) Auslenkung der Masse

Abbildung 3.4: Ergebnisse der Simulation

4 Vergleich von SimScape und Modelica

Im Folgenden sollen die beiden Softwaretools OpenModelica und Simscape anhand mehrerer Beispiele unterschiedlicher Komplexität miteinander verglichen werden. Begonnen wird mit einem simplen Beispiel, mit lediglich einer Differentialgleichung. Anschließend wird in beiden Sytemen ein Fadenpendel-Objekt modelliert, das durch phsikalische Einheiten ergänzt wird. Dieses Pendel wird als Teil eines physikalischen Models mit in den Standard-Bibliotheken enthaltenen Blöcken verbunden. Zum Schluß soll anhand eines auf- und abspringenden Gummiballs ein System mit Diskontinuitäten betrachtet werden.

4.1 Beispiel: Hello World

Wie in [Fri11] soll zur Einführung in die Sprachen auch hier ein Programm mit dem traditionsträchtigen Namen *Hello World* dienen. Da es hier aber keinen Sinn macht, einen simplen String auszugeben, soll stattdessn die Differentialgleichung $\dot{x} = -a \cdot x$ gelöst werden. Ziel dieses Beispiels ist es, zu untersuchen, wie die Implementierung eigener Objekte in den beiden Sprachen bzw. Werkzeugen funktioniert.

OpenModelika

Wie bereits erwähnt, wurde das folgende Beispiel so von Fritzson in [Fri11] verwendet und soll hier unverändert übernommen werden:

```
1 class HelloWorld
2   Real x(start = 1);
3   parameter Real a = 1;
4 equation
5   der(x) = -a * x;
6 end HelloWorld;
```

Im obigen Quellcode wurde ein Objekt vom Typ *class* mit dem Namen *HelloWorld* erstellt. In ihm wurden eine Variable x vom Typ *Real*, d.h. eine reelle einheitenlose Zahl, und dem standardmäßigen Startwert 1 und ein Parameter a, ebenfalls vom typ *Real* und dem Standardwert 1, erstellt. Der Unterschied zwischen den beiden Werten ist, daß x als Zustandsvariable über die Laufzeit der Simulation veränderlich ist und a als Parameter das System zwar beeinflußt, aber zeitlich konstant bleibt.

Gemäß Fritzsons Definition, kann ein einzelnes Objekt als System betrachtet und simuliert werden, ohne als Block in einem Blockschaltbild integriert worden zu sein. Abbildung 4.1 auf der nächsten Seite zeigt, den von OpenModelica ermittelten zeitlichen Kurvenverlauf von x für $0 \leq t \leq 2$s.

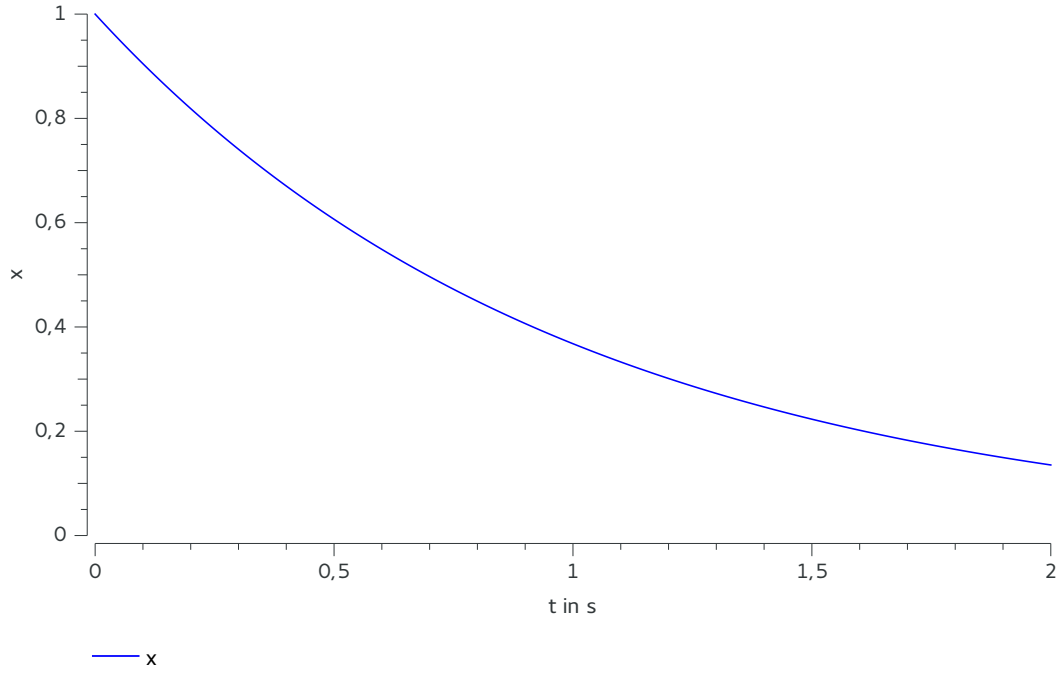


Abbildung 4.1: Ergebnis der Simulation des Beispiels *HelloWorld* in OpenModelica

SimScape

Setzt man das Beispiel in SimScape um, sieht der Quellcode wie folgt aus:

```
1  component HelloWorld
2    variables
3      x = 1;
4    end
5    parameters
6      a = 1;
7    end
8    equations
9        der(x) == -a * x * { 1, '1/s' };
10   end
11 end
```

Neben der Tatsache, daß sich die Syntax stark an Matlab orientiert, fällt der Ausdruck *{ 1, '1/s'}* auf, mit dem die rechte Seite der Gleichung multipliziert werden muß. Ursache ist, daß die *der()*-Funktion hier die ihr übergebene Größe grundsätzlich nach der Zeit ableitet und somit die physikalische Einheit der übergebenen Größe mit $^1/_s$ multipliziert. Damit die Einheiten auf beiden Seiten der Gleichung identisch sind, wird die rechte Seite mit 1 multipliziert, die die Einheit $^1/_s$ trägt.

Zudem ist das dargestellte Objekt so nicht als eigenständiges System simulierbar. Da SimScape lediglich eine Erweiterung der Software MATLAB/Simulink darstellt, können Komponenten nur in Form von SimScape-Blöcken innerhalb von

SIMULINK-Blockdiagrammen simuliert werden. Um aus dem obigen Quelltext eine neue Bibliothek, die den Block enthält, zu generieren, müssen zunächst entsprechende Ports im Quellcode definiert werden, über die der zukünftige Bock dann mit anderen Blöcken verbunden werden kann.

```
1  component HelloWorld
2    variables
3      x = 1;
4    end
5    parameters
6      a = 1;
7    end
8    outputs
9      X = 1; % X:right
10   end
11   nodes
12     na = foundation.mechanical.translational.translational; % na:left
13   end
14   equations
15     der(x) == -a * x * { 1, '1/s' };
16     X == x;
17   end
18 end
```

Es wurden die Definitionen von zwei Ports hinzugefügt, ein Signal-Output X und ein Port *na*. Durch den Output X wurde eine neue Variable hinzugefügt. Damit das unter *equations* definierte Gleichungssystem lösbar ist, muß die Anzahl an Variablen und die Anzahl Gleichungen übereinstimmen. Mit dem Hinzufügen der Gleichung $X == x$ wird diese Bedingung erfüllt. Gleichzeitig wird so sichergestellt, daß über die Ausgabevariable X die gewünschte Größe x ausgegeben wird. Damit der zukünftige Block mit anderen Blöcken und der SIMSCAPE *Solver Configuration* verbunden werden kann, muß Kontenpunkt (*node*) ertellt werden. Über diese Port verläuft in der Simulation der Austausch von Leistungen mittels Potential- und Flußgrößen. Hinter den Ports sind Komentare der Form *% label:position* angehängt. Über diese Kommentare wird der Anzeigename und die Position der definierten Ports im späteren Blocksysmbol angegenen. Für dieses Beispiel wurde ein Port der Klasse *translational* gewählt. Der String *foundation.mechanical.translational.translational beschreibt* den Namen und Speicherort der Klassendefinition. In diesem Typ Port sind die Geschwindigkeit v und die Kraft F als Potential- und Flußgrößen definiert, was momentan keine Rolle spielt, da die Variablen des Ports nicht mit den Variablen des *HelloWorld*-Blocks verbunden sind. Das soll im nachfolgenden Beispiel des Stabpendels beschrieben werden.

Über den Befehl *ssc_build* wird dann eine SimScape-Bibliothek erstellt, die den oben definierten Block enthält. Jezt kann das in Abbildung 4.2 dargestellte Blockdiagramm für die Simulation erstellt werden. Es zeigt, daß auch hier der zuvor erstellte Block *HelloWorld* nicht als eigenständiges System simuliert werden kann. Erst eine Gruppe aus mindestens zwei Blöcken mit einer festen Referenz, die an den *Solver Configuration* Block angeschloßen sind, wird von MATLAB/SIMULINK als

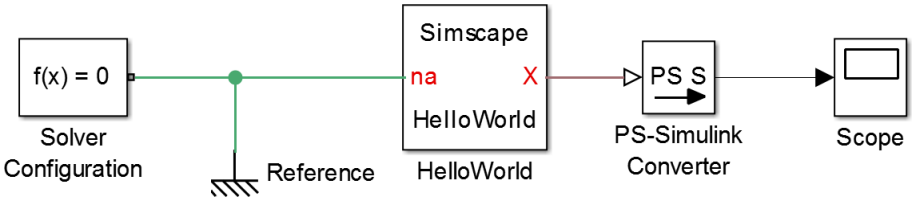


Abbildung 4.2: *HelloWorld* als SIMSCAPE-Blockdiagramm

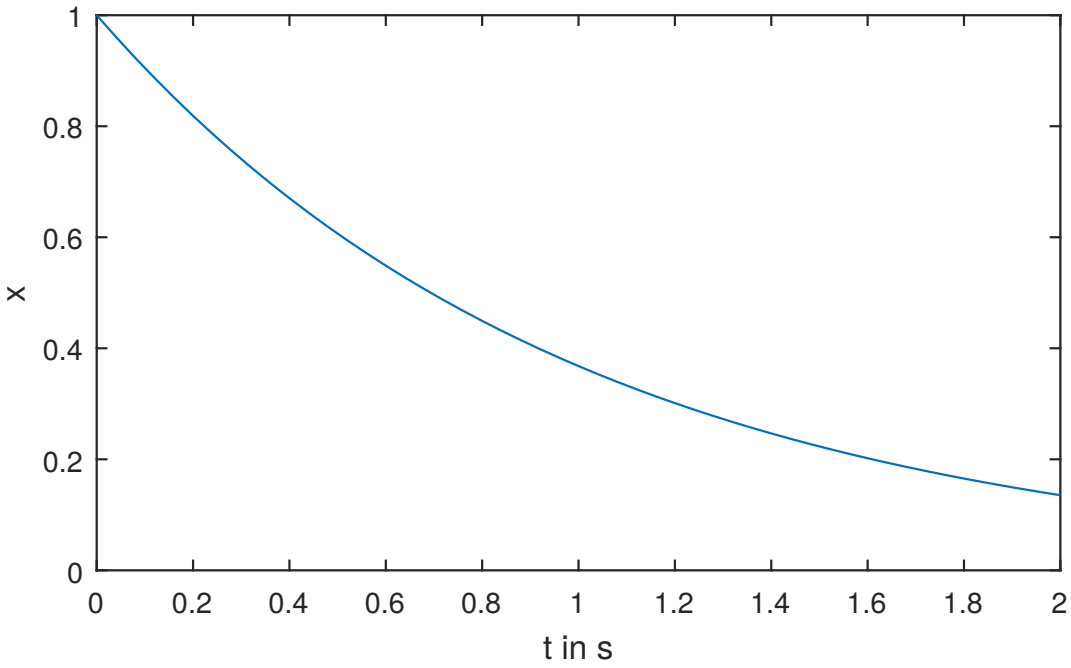


Abbildung 4.3: Ergebnis der Simulation des Beispiels *HelloWorld* in SIMSCAPE

simulierbares System anerkannt. Abbildung 4.3 zeigt die Ergebnisse der Simulation.

Vergleich der Ergebnisse

Bei gleicher Schrittweite und den Standardeinstellungen der Solver, ist die maximale Abweichung zwischen den Ergebnisse der beiden Anwendungen $\Delta x_{max} = 1{,}0380 \cdot 10^{-6}$. Ein detaillierter Vergleich der Zeit, die die Simulation in Anspruch nimmt, würde den Rahmen dieser Arbeit Sprengen und ist zu vernachläßigen.

4.2 Beispiel: Stabpendel

In diesem Kapitel soll ein einfaches Stabpendel in OPENMODELICA und SIMSCAPE umgesetzt werden. Znunächst wird ein entsprechender Block in beiden Sprachen erstellt und anschließend einzeln simuliert. Danach wird er mit Blöcken der Standard-

Bibliotheken ergänzt, die das System um Reibung und ein angreifendes Drehmoment ergänzen. Zuvor soll das Modell des Stabpendels skizziert und mathematisch beschrieben werden.

Modellbildung

Abbildung 4.4 stellt ein einfaches Stabpendel dar. Simuliert werden soll der zeitliche Verlauf der Auslenkung x in Abhängigkeit der Anfangsauslenkung φ_0, der Masse m und der Länge des Stabes L. Die Masse wird dabei als Punktmasse angenommen. Der Stab sei ideal steif und masselos.

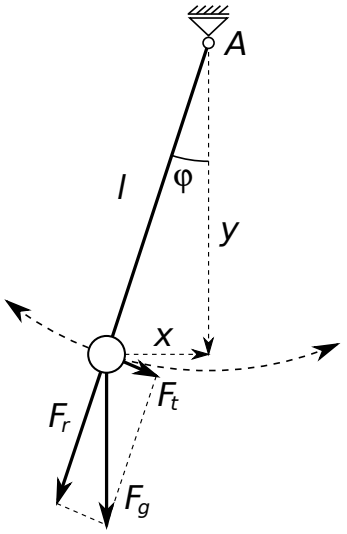


Abbildung 4.4: Modellskizze eines Stabpendels

Zur Beschreibung des Modells wird die Momentenbilanz um die Aufhängung des Pendels gezogen:

$$\begin{aligned}\sum M^{(A)} = M =& F_t \cdot l + M_J \\ =& m \cdot g \cdot l \cdot \sin(\varphi) + m \cdot l^2 \cdot \ddot{\varphi} \\ =& m \cdot g \cdot x + m \cdot l^2 \cdot \ddot{\varphi}\end{aligned}$$

Pendel ohne Reibung

Die zuvor erstellte Modellgleichung soll im Folgenden ähnlich wie im vorhergehenden Beispiel in beiden Systemen umgesetzt werden. Hinzu kommen in diesem Beispiel die Definition von physikalischen Einheiten, Ports und deren Integration ins Gleichungssystem sowie eine Dokumentation des Objekts.

OpenModelica

Es soll nun wie angekündigt das beschriebene Modell in OpenModelica umgesetzt werden. Im Folgenden ist der entsprechende Quellcode aufgelistet.

```
1  model Pendulum
2    constant Real PI = 3.141592653589793;
3    parameter Modelica.SIunits.Mass m = 1 "Mass";
4    parameter Modelica.SIunits.Acceleration g = 9.81 "Earth's
         gravitational acceleration";
5    parameter Modelica.SIunits.Length l = 0.5 "Length of the stick";
6    Modelica.SIunits.AngularVelocity omega(start = 0) "Angular velocity
         ";
7    Modelica.SIunits.Angle phi(start = PI / 2) "Angular deflexion of the
          pendulum";
8    Modelica.SIunits.Length x(start = 0) "Deflexion of the pendulum";
9    Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
         annotation(Placement(visible = true, transformation(origin =
         {-100, 0}, extent = {{-10, -10}, {10, 10}}, rotation = 0),
         iconTransformation(origin = {-100, 0}, extent = {{-10, -10},
         {10, 10}}, rotation = 0)));
10   Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
         annotation(Placement(visible = true, transformation(origin =
         {100, 0}, extent = {{-10, -10}, {10, 10}}, rotation = 0),
         iconTransformation(origin = {100, 0}, extent = {{-10, -10}, {10,
          10}}, rotation = 0)));
11 equation
12   flange_a.tau + flange_b.tau = m * g * x + m * l ^ 2 * der(omega);
13   omega = der(phi);
14   phi = flange_a.phi;
15   phi = flange_b.phi;
16   x = l * sin(phi);
17 annotation(Diagram(coordinateSystem(extent = {{-100, -100}, {100,
       100}}, preserveAspectRatio = true, initialScale = 0.1, grid = {2,
       2})), Documentation(info = "<html>
18 <p>
19   This Block block simulates the dynamic behavior of a simple stick
         pendulum.
20 </p>
21 </html>"), Icon(coordinateSystem(extent = {{-100, -100}, {100, 100}},
       preserveAspectRatio = true, initialScale = 0.1, grid = {2, 2}),
       graphics = {Text(origin = {0, 105}, lineColor = {0, 0, 255},
       fillColor = {0, 0, 255}, extent = {{-80, -25}, {80, 25}},
       textString = "%name"), Bitmap(origin = {-23, 50}, extent = {{-63,
       32}, {115, -138}}, fileName = "modelica://Pendulum/Icon.png")}));
22 end Pendulum;
```

Der Quelltext beschreibt ein Objekt vom typ *model* mit dem Namen *Pendulum*. Für den weiteren Gebrauch wurde die Konstante π definiert, gefolgt von einigen Parametern. Die Parameter sind aber nicht mehr vom Typ *Real*, sondern wurden den Klassen verschiedener physikalischer Größen zugeordnet. Über die Klassenzuordnungen werden entsprechende SI-Einheiten an die Parameter vererbt. Äquivalent wurde bei der Definition der drei Objektvariablen vorgegangen. Alle Parameter und Variablen

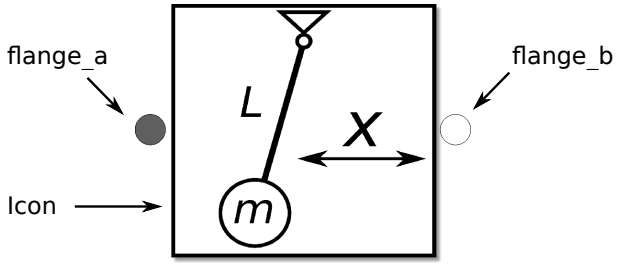


Abbildung 4.5: Pendel-Icon in OpenModelica

wurden mit Kommentaren versehen. Setzt man das Objekt in einem Blockdiagramm ein und möchte die Parameter und Anfangswerte der Variablen ändern, erscheinen die Kommentare als Beschreibungen der Formulare im Parametrierungsfenster.

In den Zeilen 6 und 7 werden zwei Ports (*Flanges*) des Plendels definiert. Die Definition eines Ports beginnt mit dem Klassenaufruf. In diesem Besispiel soll die Rotation des Pendels simuliert werden. Es ist also sinnvoll, die Ports aus der *Modelica.Mechanics.Rotational*-Bibliothek zu nutzen. Die Bibliothek enthält zwei Ports: *Interfaces.Flange_a* und *Interfaces.Flange_b*. Beide unterscheiden sich zunächst nur in ihrem Aussehen im Blockschaltbid. Auf den Klassenaufruf folgt die Definition des Namens der Instanz dieser Klasse: Der in Zeile 6 definierte Port gehört zum Beispiel zur Klasse *Modelica.Mechanics.Rotational.Interfaces.Flange_a* und trägt den Namen *flange_a*. Über den Befehl *annotation()* wird am Ende das genaue Aussehen der der Ports im Icon definiert.

Anschließend erfolgt die Definition des Gleichungssystems. Es wurden in diesem Objekt drei Variablen definiert. Zusätzlich wurden je eine Potential- und Flußgröße von den Ports an das Pendel vererbt, womit das Objekt insgesamt über sieben Variablen verfügt. Entsprechend muß das Gleichungssystem auch über sieben Gleichungen verfügen, um eindeutig lösbar zu sein. Über die Klassen der Ports wurde zusätzlich zu den Variablen je eine Gleichung an das Pendel vererbt, die sogenannten *Connection Equations*. Nähere Informationen sind in [Mod12] zu finden. Deshalb müssen im Objekt *Pendulum* fünf Gleichungen aufgestellt werden. Das Gleichungssystem enthält als nennenswerte Gleichungen zum einen die in der Modellbildung aufgestellte Momentenbilanz, die um die Flußgrößen der Ports ergänzt wurde, welche dadurch die Bilanz beeinflussen, und die Gleichungen, die die Beziehungen zwischen den Potentialgrößen des Objekts und des Ports definieren.

Zum Schluß erfolgt die Gestaltung des Block-Icons über einen *annotation()*-Befehl. Zusätzlich ist in diesem Befehl eine in HTML formatierte Dokumentation des Blocks enthalten. Alle *annotation()*-Befehle dieses Objekts wurden automatisch von OpenModelica generiert. OpenModelica verfügt dazu über einen rudimentären Zeichenmodus, über den aussagekräftige Icons erstellt werden können. Abbildung 4.5 zeigt das entstandene Icon in OpenModelica. Die Ports und die Icon-Graphik sind entsprechend gekennzeichnet.

SimScape

Nun soll der gleiche Block analog in SimScape erstellt werden. Wie bisher soll dabei auf Unterschiede zwischen SimScape und OpenModelica aufmerksam gemacht werden.

```
1  component Pendulum
2    % Pendulum
3    % This block simulates the dynamic behaviour of a
4    % simple stick pendulum.
5    variables
6      omega = { 0, 'rad/s' };  % Angular velocity
7      phi   = { pi/2, 'rad' }; % Angular deflexion of the pendulum
8      tau   = { 0, 'N*m' };    % Torque inside the bearing
9    end
10   outputs
11     x = { 0, 'm' }; % x:right
12   end
13   parameters
14     m = { 1, 'kg' };        % Mass
15     g = { 9.81, 'm/s^2' }; % Earth's gravitational acceleration
16     l = { 0.5, 'm' };       % Length of the stick
17   end
18   nodes
19     R = foundation.mechanical.rotational.rotational; % R:left
20     C = foundation.mechanical.rotational.rotational; % C:right
21   end
22   branches
23     tau : R.t -> C.t; % Flow variable from R to C
24   end
25   equations
26     tau   == m*g*x + m*l^2*der(omega);
27     omega == R.w - C.w; % Potential variable between R and C
28     omega == der(phi);
29     x     == l*sin(phi);
30   end
31 end
```

Als Erstes fällt auf, daß die Dokumentation nicht über einen *annotation()*-Befehl erfolgt, sondern, wie es in Matlab üblich ist, über Kommentarzeilen zu Beginn der Datei. Variablen und Parameter werden in separaten *Sections* definiert. Standardmäßiger Startwert und Einheit werden den Variablen und Parametern als *Cell Array* übergeben. Die Erste Zelle enthält dabei den Betrag als Zahl und die zweite Zelle die physikalische Einheit als String. Einheiten werden nicht im Zuge einer Klassenzuordnung vererbt, sondern für jede Variable separat definiert. Das ist in Modelica zwar möglich, widerspricht aber der Vererbungs-Philosophie und ist nicht üblich.

Die Definition der Ports erfolgt analog zu Modelica mit dem Unterschied, daß die Beschriftung und Positionierung über einen kurzen Kommentar erfolgt. Ein weiterer prinzipieller Unterschied ist, daß die Potential- und Flußgrößen der Ports in SimScape nicht als Variablen an das untergeordnete Objekt vererbt werden. Sie können aber als Parameter in das Gleichungssystem eingefügt werden, wodurch sie

das Verhalten der Komponente beeinflußen. Daher müssen in jedem Objekt eigene Potential- und Flußgrößen definiert werden. Das dominante Prinzip ist in SimScape demnach nicht die Vererbung, sondern die Verkapselung. In Simscpe existiert zudem keine Unterscheidung von Ports äquivalent zu den zwei Portklassen *Flange_a* und *Flange_b* in Modelica. Die positive Flußrichtung der Flußgrößen wird nicht automatisch über verschiedene Portklassen in *Connection Equations* definiert, sondern manuell in der *branches section* der Komponente festgelegt. In diesem Beispiel fließt die Flußgröße vom Port R zum Port C. Gemäß [Mat15] wird *tau* von der Energieerhaltungsgleichung in Port R subtrahiert und zu der Erhaltungsgleichung in Port C addiert.

Da keine Variablen und Gleichungen an die Komponente vererbt werden, müssen nur vier Gleichungen für die vier Komponenten-Variablen definiert werden. Zuerst wurde die Bilanzgleichung aus der Modellbildung übernommen. Die Potentialgröße ist im Gegensatz zu Modelica nicht der Winkel *phi*, sondern die Winkelgeschwindigkeit *omega* oder auch *w*. Sie wird in SimScape als Differenz der Potentialgrößen der Ports entsprechend der Flußrichtung der Flußgröße definiert. Die Komponenten-Potentialgröße stellt in SimScape den Potentialzuwachs bzw. Abfall über die Komponente zwischen den Ports dar. Sie wird über die Laufzeit der Simulation über die Ports beeinflußt und verändert dadurch die Momententbilanz des Pendels. Eine vergleichbare Verknüpfung zwischen Komponente und Ports hat im Beispiel *Hello World* nicht stattgefunden.

Abschließend wurde die Datei *Pendulum.ssc* über den Befehl *ssc_build* in einen SimScape-Block überführt. Das Icon wurde als *Pendulum.png* im gleichen Verzeichnis hinterlegt und von *ssc_build* als Block-Icon übernommen. Abbildung 4.6 zeigt den entstandenen SimScape-Block mit beiden physikalischen Ports R und C und dem Output-Port x.

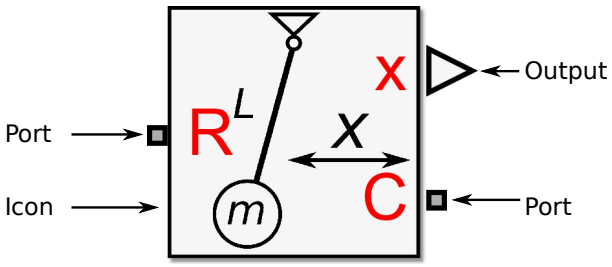


Abbildung 4.6: Pendel-Icon in SimScape

Simulationsergebnisse

In OpenModelica kann das Pendel wieder als eigenständiges System simuliert werden. In SimScape muß erneut ein Modellgraph erstellt werden. Dieser ist in Abbildung 4.7 dargestellt. Da die Variablen der Ports mit den Variablen der Komponente verknüpft wurden, benötigt der erzeugte *Pendulum*-Block eine rotatorische Quelle, um fehlerfrei simuliert werden zu können. Hier wurde der *Torque*-Block als Quelle

gewählt. An den *Torque*-Block wurde ein konstantes Signal mit dem Wert 0 angelegt, wodurch die Drehmomentquelle keinen Einfluß auf das Schwingungsverhalten des Pendels nimmt. Es wurde der *ODE*45-Solver mit einer maximalen Schrittweite von 0,01 s gewählt.

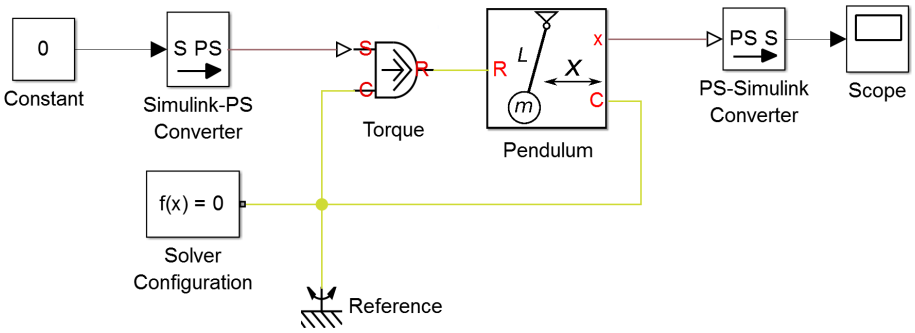


Abbildung 4.7: Modellgraph des ungedämpften Systems

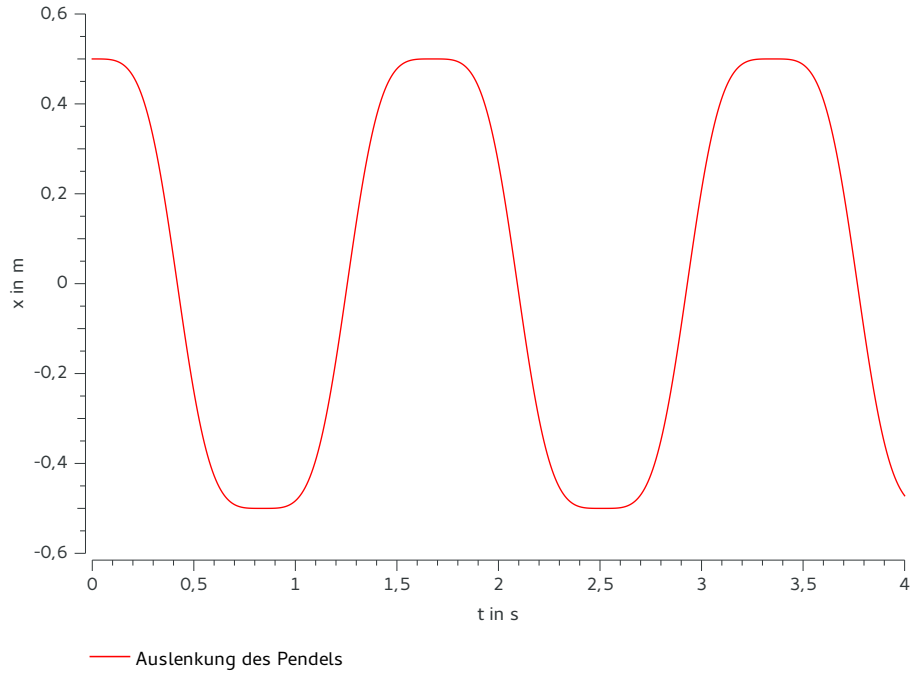
Beide Simulationen zeigen die gleiche zeitliche Änderung der Auslenkung des Pendels. Die Ergebnisse werden in Abbildung 4.8 gezeigt.

Pendel mit Reibung

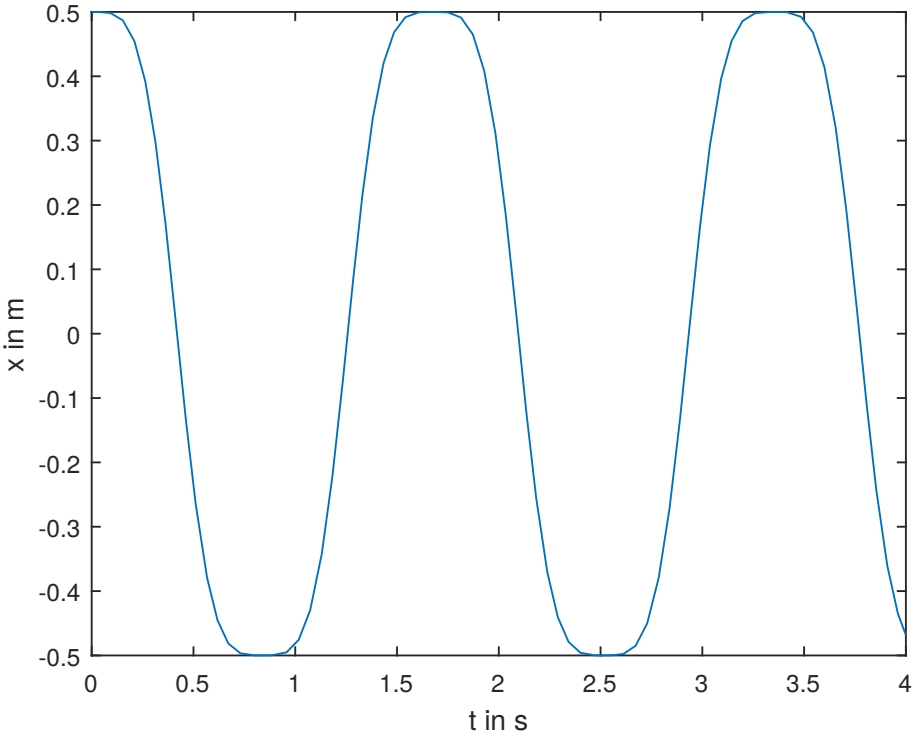
Dem Modell soll nun eine Reibung an der Aufhängung des Pendels hinzugefügt werden, die die Schwingung dämpft. Ferner soll ein Drehmoment hinzugefügt werden, das das Pendel alle zehn Sekunden wieder in Schwingung versetzt. Dadurch wird der Vorteil der objektorientierten Modellbildung sichtbar: Einmal erstellte Objekte können in unterschiedlichen Situationen wiedervervendet werden, ohne modifiziert zu werden.

OpenModelika

In OpenModelica wird dazu eine neue **.mo*-Datei erstellt in die das Pendel-Objekt eigefügt wird. Hinzu kommen die Blöcke *bearingFriction* und *torque* für die Erzeugung von Reibung und Drehmoment, wie es in Abbildung 4.9 dargestellt ist. An das Drehmoment ist eine Signalquelle angeschloßen, die alle zehn Sekunden eine kurze Signalspitze mit dem Betrag 10 ausgibt. Über die Verbinder erfolgt nun ein Austausch von Leistung zwischen den Komponenten, was zur Folge hat, daß das zuvor freischwingende Pendel gedämpft und wieder angetrieben wird.



(a) In OpenModelica



(b) In SimScape

Abbildung 4.8: Ergebnisse der Simulation des ungedämpften Systems

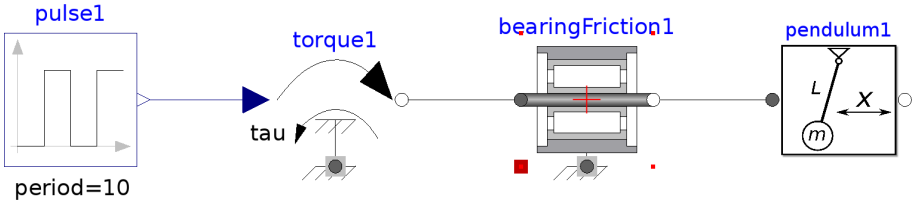


Abbildung 4.9: Modellgraph des gedämpften Systems in OpenModelica

SimScape

Der in Abbildung 4.10 gezeigte Modellgraph ähnelt sehr dem aus Abbildung 4.7. Er wurde lediglich um den Block *Rotational Friction* ergänzt. Zudem wurde die konstante Signalquelle durch einen *Pulse Generator* ersetzt, der genauso wie der in OpenModelica benutzte Block arbeitet.

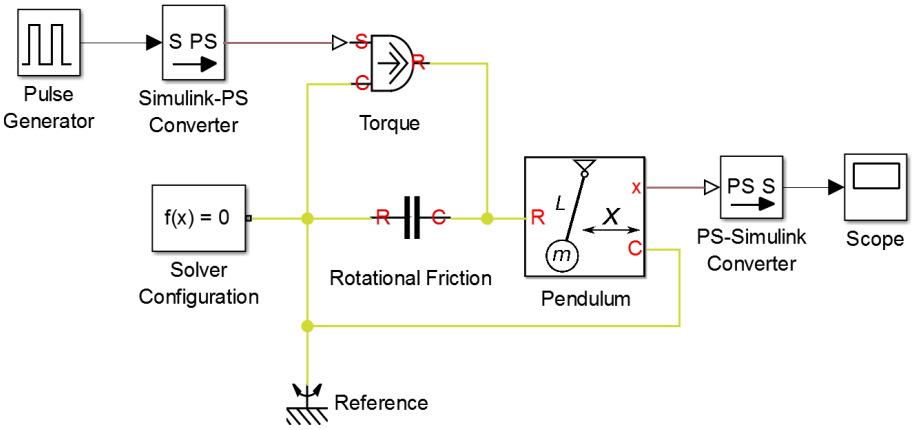
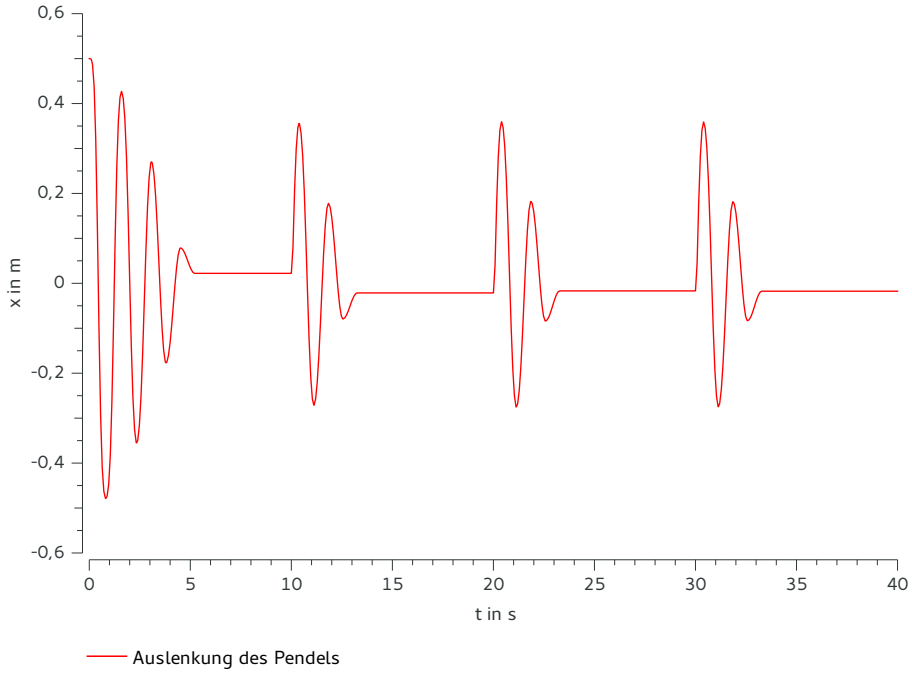


Abbildung 4.10: Modellgraph des gedämpften Systems in OpenModelica

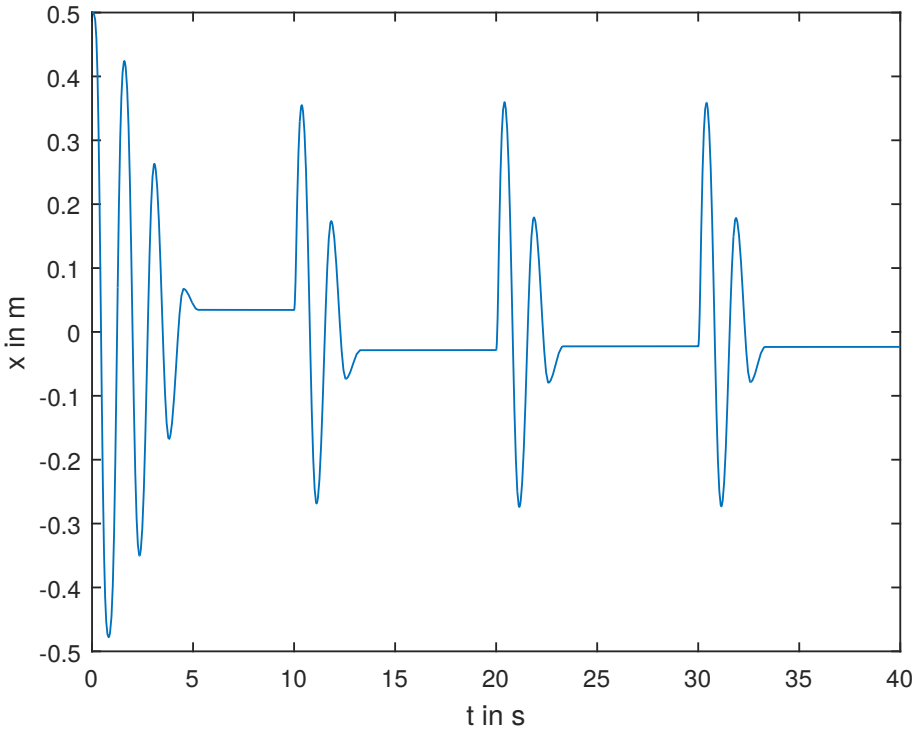
Simulationsergebnisse

Abbildung 4.11 zeigt, daß beide Systeme qualitativ gleiche Ergebnisse liefern. Ein genauer Vergleich der Simulationsergebnisse wäre nicht aussagekräftig, da die Reibungsblöcke der zwei Systeme unterschiedlich funktionieren. Dem Modelica-Block wird das Reibmoment in Abhängigkeit der Winkelgeschwindigkeit als Matrix übergeben. Zwischen den in der Matrix angegebenen Stützpunkten wird dann der für die im Lager herrschende Winkelgeschwindigkeit das entsprechende Reibmoment linear interpoliert. Werte außerhalb des definierten Bereichs werden extrapoliert. Im SimScape-Block Rotational Friction werden diverse Konstanten festgelegt, unter Anderem das Haftreibmoment und Gleitreibmoment.

Da das gedämpfte System eine hohe Dynamik in kurzer Zeit aufweist, wurde zum Ermitteln der Simulationsergebnisse der *ODE23s*-Solver gewählt.



(a) In OpenModelica



(b) In SimScape

Abbildung 4.11: Ergebnisse der Simulation des gedämpften Systems

4.3 Beispiel: Springender Ball

In diesem Kapitel soll anhand des Beispiels des springenden Balls betrachtet werden, wie in beiden Modellierungswerkzeugen kontinuierliche Systeme mit Diskontinuitäten realisiert werden. Wie in Abbildung 4.12 ersichtlich, ist die positive Richtung der Geschwindigkeit v der Fallbeschleunigung g entgegengesetzt. Bei jedem Aufprall auf den Boden verliert der Ball an Geschwindigkeit. Der Verlust wird über die Stoßzahl k realisiert. Ziel ist es, den zeitlichen Verlauf der Flughöhe h zu ermitteln. Zur Beschreibung des physikalischen Verhaltens gilt

$$\begin{aligned} \dot{h} &= v \\ \dot{v} &= \begin{cases} -g & \text{für } h > 0 \\ k \cdot g & \text{für } h \leq 0 \end{cases} \end{aligned}$$

mit $h_0 = 1\,\mathrm{m}$, $v_0 = 0\,\mathrm{m/s}$, $g = 9{,}81\,\mathrm{m/s^2}$ und $k = 0{,}7$.

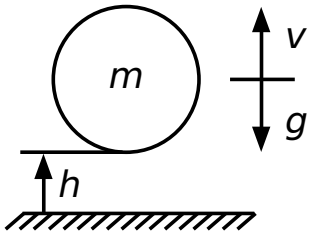


Abbildung 4.12: Modellskizze eines springenden Balls

OpenModelica

Zunächst sollte das Modell in OpenModelica umgesetzt werden. Dazu wurde das Objekt *bouncingBall* erstellt.

```
1  model bouncingBall
2    parameter Modelica.SIunits.Acceleration g = 9.81 "Gravitational acceleration";
3    parameter Real k = 0.7 Coefficient of restitution";
4    Modelica.SIunits.Length h(start = 1) "Height of the ball";
5    Modelica.SIunits.Velocity v(start = 0) "Velocity of the ball";
6  equation
7    der(h) = v;
8    der(v) = -g;
9    when h <= 0 then
10     reinit(v, -k * pre(v));
11   end when;
12 end bouncingBall;
```

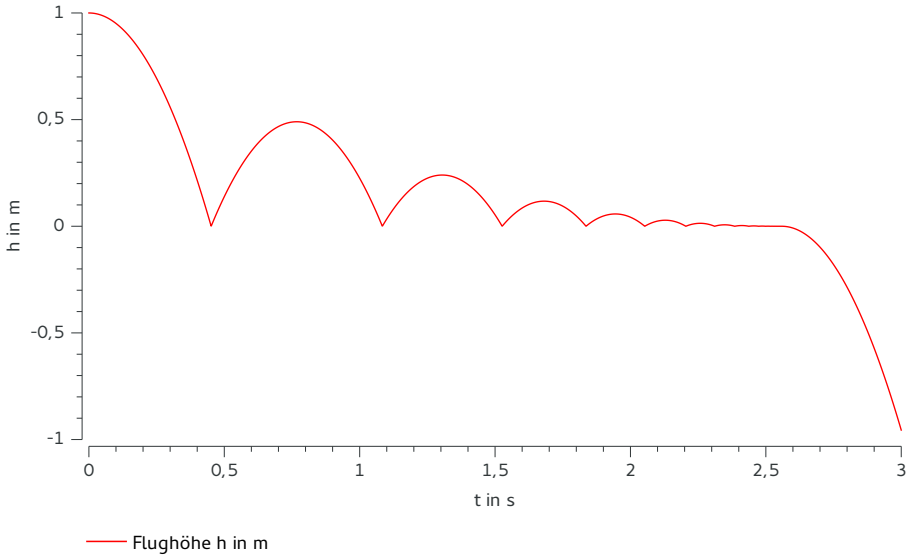
Als Erstes wurden die Parameter g und k sowie die Variablen für die Flughöhe h und die Geschwindigkeit v mit den entspechenden Klassenzuweisungen für die physikalischen Einheiten definiert. Danach erfolgte das Aufstellen des Gleichungssystems. Das kontinuierliche Verhalten wurde auf die gleiche Weise wie in den vorangegangenen Beispielen in einem Gleichungssystem beschrieben. Zusätzlich wurde

die Diskontinuität des Aufpralls definiert. In MODELICA wird ein diskretes Ereignis über die *when*-Abfrage realisiert. Der Unterschied zur *if*-Abfrage ist, daß bei *if* ein bestimmter Code ausgeführt wird, *solange* die entsprechende Bedingung erfüllt wird, und bei *when* der Code ausgeführt wird, *sobald* die Bedingung erfüllt wird. In diesem Beispiel wird zum Ereigniszeitpunkt die Geschwindigkeit einmalig über den *reinit()*-Befehl mit *-k * pre(v)* reinitialisiert. Der Ausdruck *-k * pre(v)* ist dabei der Wert der Geschwindigkeit aus dem vorangegangenen Iterationsschritt multipliziert mit der Stoßzahl k und umgekehrter Flugrichtung. Nach dem Aufprall wird die kontinuierliche Simulation mit reinitialisierter Geschwindigkeit fortgesetzt. Nähere Informationen zu den Befehlen *when*, *reinit()* und *pre()* sind [Mod12] zu entnehmen. Die Abbildung 4.13a zeigt das Ergebnis der Simulation.

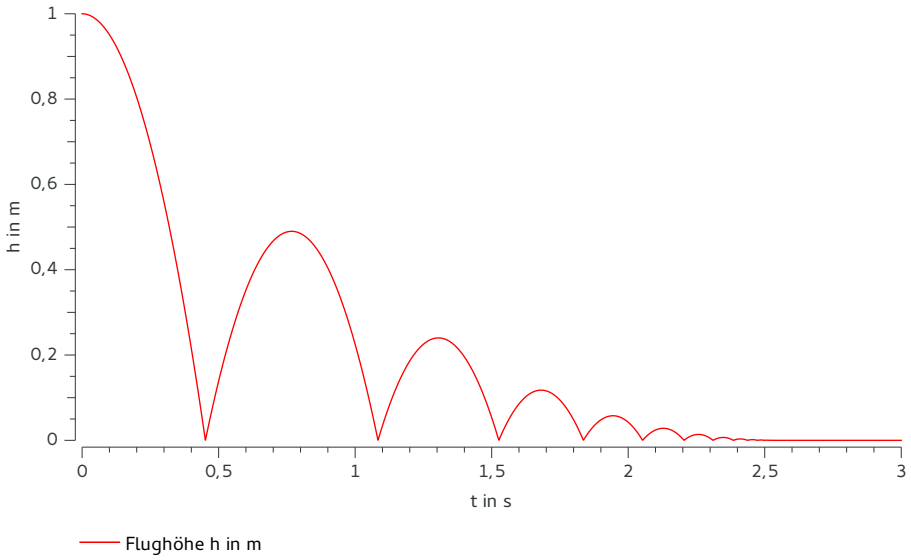
Außerdem ist in diesem Bild ein Phänomen zu erkennen, das gemäß [Fri11] als *Zeno-Effekt* bekannt ist. Mit jedem Aufprall verliert der Ball an Geschwindigeit, wodurch die Flughöhen und zeitlichen Aufprallabstände mit jeder Flugphase kleiner werden und das Aufprallereignis ab einem gewissen Punkt aufgrund von numerischen Ungenauigkeiten nicht mehr erkannt wird. Der Ball fällt dann durch den Boden ins Leere. Diesem Problem wurde im Objekt *zenoBall* begegnet.

```
1  model zenoBall
2    parameter Modelica.SIunits.Acceleration g = 9.81 "Gravitational acceleration";
3    parameter Real k = 0.7 Coefficient of restitution";
4    Modelica.SIunits.Length h(start = 1) "Height of the ball";
5    Modelica.SIunits.Velocity v(start = 0) "Velocity of the ball";
6    Boolean contact "Boolean flag to detect impact";
7  equation
8    contact = h <= 0 and v <= 0;
9    der(h) = v;
10   der(v) = if contact then 0 else -g;
11   when contact then
12     reinit(v, if edge(contact) then -k * pre(v) else 0);
13   end when;
14 end zenoBall;
```

In Zeile 6 wurde zunächst die boolsche Variable *contact* hinzugefügt. In der Gleichung in Zeile 8 wird die Bedingung für diese Variable definiert. Nach dieser Definition nimmt *contact* den Wert *true* an, solange der Ball den Boden berührt. Die Höhe h und Geschwindigkeit v sind in diesem Fall kleiner oder gleich 0. Zudem wurde der Gleichung in Zeile 10 eine Fallunterscheidung hinzugefügt: Ist der Ball im Kontakt mit dem Boden, soll die Änderung der Geschwindigkeit $\dot{v}$ den Wert 0 erhalten. Befindet sich der Ball im Flug, beträgt die Änderung $-g$. Auch die Reinitialisierung der Geschwindigkeit wurde modifiziert. Die Variable v wird nur dann mit *-k * pre(v)* intialisiert, wenn die Bedingung *edge(contact)* erfüllt ist. Der Ausdruck *edge(contact)* ist gemäß *[Mod12]* eine Kurzform von *contact and not pre(contact)*. Wenn der Ball also den Boden berührt, sich aber im vorhergehenden Iterationsschritt im Flug befand, wird v wie bisher mit *-k * pre(v)* intialisiert, sonst mit 0. Diese Maßnahmen verhindern, daß es während der Simuation zu Fehlern kommt, was in Abbildung 4.13b zu erkennen ist.



(a) Mit Zeno-Effekt



(b) Ohne Zeno-Effekt

Abbildung 4.13: Simulationsergebnis des Beispiels *bouncingBall in* OpenModelica

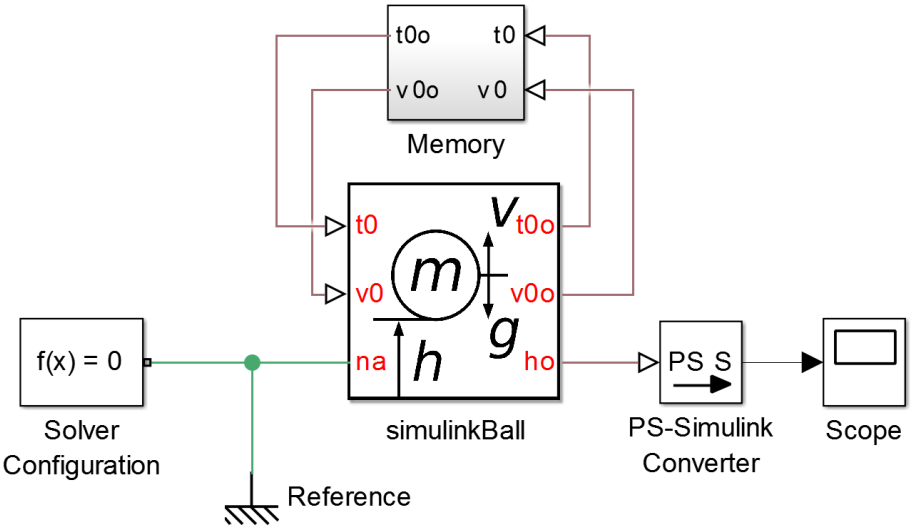
SimScape

Möchte man das gleiche Objekt in SimScape erstellen, stößt man schnell auf das Problem, daß in in der SimScape-Sprache kein *reinit()*-Befehl existiert, weshalb die Reinitialisierung der Geschwindigkeit anders erfolgen muß. Dazu sollen hier zwei unterschiedliche Lösungen vorgestellt werden, in denen das SimScape-Modell zum Einen mit Simulink-Blöcken kombiniert und zum Anderen über ein Matlab-Script gesteuert wird.

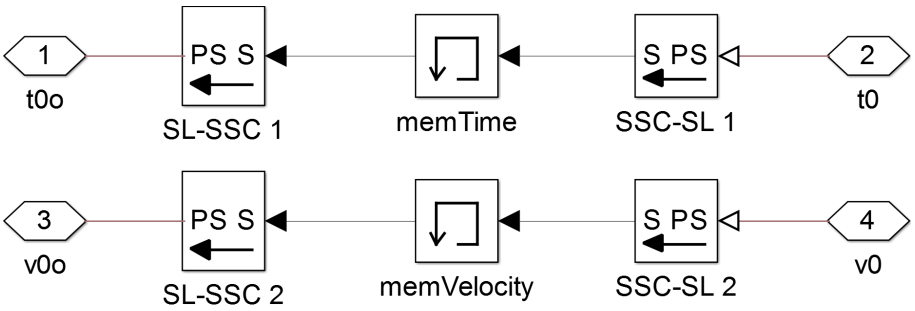
Kombination mit Simulink

Für die Simulation des Springenden Balls wurde der folgende SimScape-Block erstellt und in den in Abbildung 4.14 dargestellten Modellgraphen integriert.

```
1  component simulinkBall
2    parameters
3      g = { 9.81, 'm/s^2' }; % Gravitational acceleration
4      k = 0.7;               % Coefficient of restitution
5    end
6    variables
7      v = { 0, 'm/s' }; % Velocity of the ball
8      h = { 1, 'm' };   % Height of the ball
9    end
10   inputs
11     t0 = {0, 's'};   %t0:left
12     v0 = {0, 'm/s'}; %v0:left
13   end
14   outputs
15     t0o = {0, 's'};  %t0o:right
16     v0o = {0 'm/s'}; %v0o:right
17     ho  = {1 'm'};   %ho:right
18   end
19   nodes
20     na = foundation.mechanical.translational.translational; % na:left
21   end
22   equations
23     der(h) == v;
24     v      == -k*v0 - g*(time - t0);
25     ho     == h;
26     % write memory
27     if (h <= 0 && v < 0)
28       t0o == time;
29       v0o == v;
30     else
31       t0o == t0;
32       v0o == v0;
33     end
34     end
35 end
```



(a) Hauptsystem



(b) *Memory*-Subsystem

Abbildung 4.14: Modellgraph des springenden Balls in Kombination mit Simulink

Entscheidend sind die im SimScape-Block erstellten Ein- und Ausgänge und die in der *equations section* hinterlegte Fallunterscheidung. Die Ein- und Ausgänge des Blocks sind mit dem *Memory*-Subsystem verbunden, das zwei *Memory*-Blöcke aus der Simulink-Bibliothek enthält. Diese Speicherblöcke erhalten die Werte der Flughöhe und Geschwindigkeit als Eingangssignale, um sie mit einem Iterationsschritt Verzögerung an den SimScape-Block zurückzugeben. In der Fallunterscheidung ist dann definiert, welche Werte in die Signalausgänge geschrieben werden sollen. Beim Aufprall werden die Geschwindigkeit und der Aufprallzeitpunkt in die Ausgänge geschrieben und stehen ab der nächsten Iteration zur weiteren Berechnung der Flugbahn zu Verfügung. Solange sich der Ball im Flug befindet, rotieren die Werte des letzten Aufpralls zwischen den Speicherblöcken und dem Modellblock und dienen als reinitialisierte Ausganswerte zur Berechnung.

Auch in SimScape kommt es zum Zeno-Effekt. Abbildung 4.16a zeigt das Simulationsergebnis in SimScape ohne Berücksichtigung des Effekts bei der Programmierung des Blocks. Im folgenden Quellcode ist die notwendige Anpassung hervorgeho-

ben, die nötig ist, um den Zeno-Effekt zu berücksichtigen. Es handelt sich dabei um eine Fallunterscheidung, die bei Unterschreitung eines Grenzwertes die Berechnung abbricht. Das entsprechende Simulationsergebnis ist in Abbildung 4.16b dargestellt.

```
1   equations
2     der(h) == v;
3           == if abs(v) < { 1e-5, 'm/s' } && h < { 1e-5, 'm' }, ...
4              { 0, 'm/s' } else -k*v0 - g*(time - t0) end;
5     ho     == h;
6     % write memory
7     if (h <= 0 && v < 0)
8       t0o == time;
9       v0o == v;
10    else
11      t0o == t0;
12      v0o == v;
13    end
14  end
```

Als Solver wurde *ODE15s* verwendet. Alternativ zur Anpassung des Quellcodes können in der Solverkonfiguration von Simulink die Grenzen der Schrittweite angepasst werden (z. B. max. 10^{-2} s und min. 10^{-4} s). Die Simulation dauert in diesem Fall aber merklich länger.

Kombination mit Matlab

Das Prinzip bei der Reinitialisierung der Geschwindigkeit über ein MATLAB-Script ist, daß das Simulationsmodell aus diesem Script heraus gestartet und während der Laufzeit unterbrochen wird. Im Script wird dann die Richtung der Geschwindigkeit umgekehrt und die Simulation danach fortgeführt. Der folgende Code zeigt das verwendete SIMSCAPE-Modell. In Abbildung 4.15 ist das entsprechende Blockschaltbild gezeigt.

```
1  component matlabBall
2    parameters
3  %   k  = 0.7;               % Coefficient of restitution
4      g  = { 9.81, 'm/s^2' }; % Gravitational acceleration
5      h0 = {1, 'm'};          % Initial height of the ball
6      v0 = {0, 'm/s'};        % Initial velocity of the ball
7    end
8    variables
9      h = { 1, 'm' };   % Height of the ball
10     v = { 0, 'm/s' }; % Velocity of the ball
11   end
12   outputs
13     hout    = { 1, 'm' };   % h:right
14     vout    = { 0, 'm/s' }; % v:right
15     contact = 0;            % c:right
16   end
17   nodes
18     na = foundation.mechanical.translational.translational; % na:left
19   end
```

```
20    function setup
21      h = h0; % initializes h with h0
22      v = v0; % initializes v with v0
23    end
24    equations
25      contact == if h<=0 && v<0, 1 else 0 end;
26      der(h)  == v;
27      der(v)  == -g;
28      hout    == h;
29      vout    == v;
30    end
31  end
```

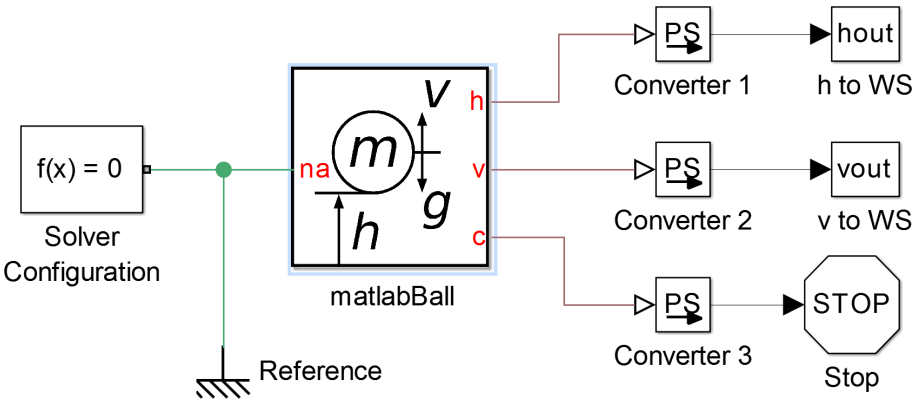


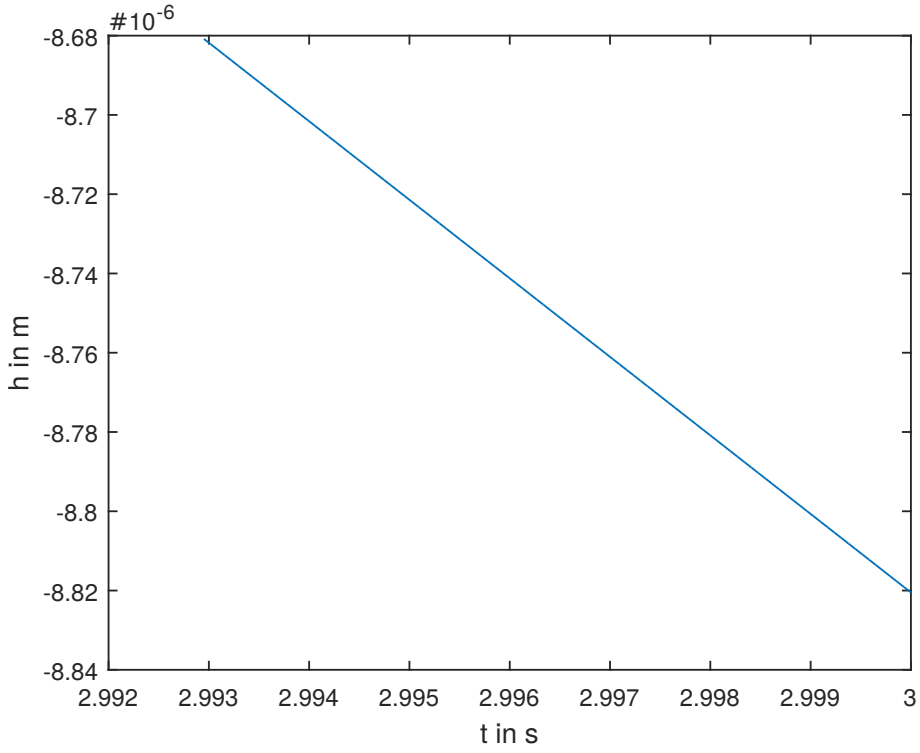
Abbildung 4.15: Modellgraph des springenden Balls in Kombination mit MATLAB

Beudeutsam ist in diesem Block die *setup section,* in der die Höhe und Geschwindigkeit bei jedem Simulationsstart mit den Parametern *h0* und *v0* initialisiert werden. Über die Ausgabevariable *contact* und den *Stop*-Block aus der SIMULINK-Bibliothek wird die Simulation bei Aufprall angehalten. Die bisherigen Werte für Höhe und Geschwindigkeit werden über entsprechende SIMULINK-Blöcke and den MATLAB-*Workspace* übergeben und im folgenden Script weiterverarbeitet.

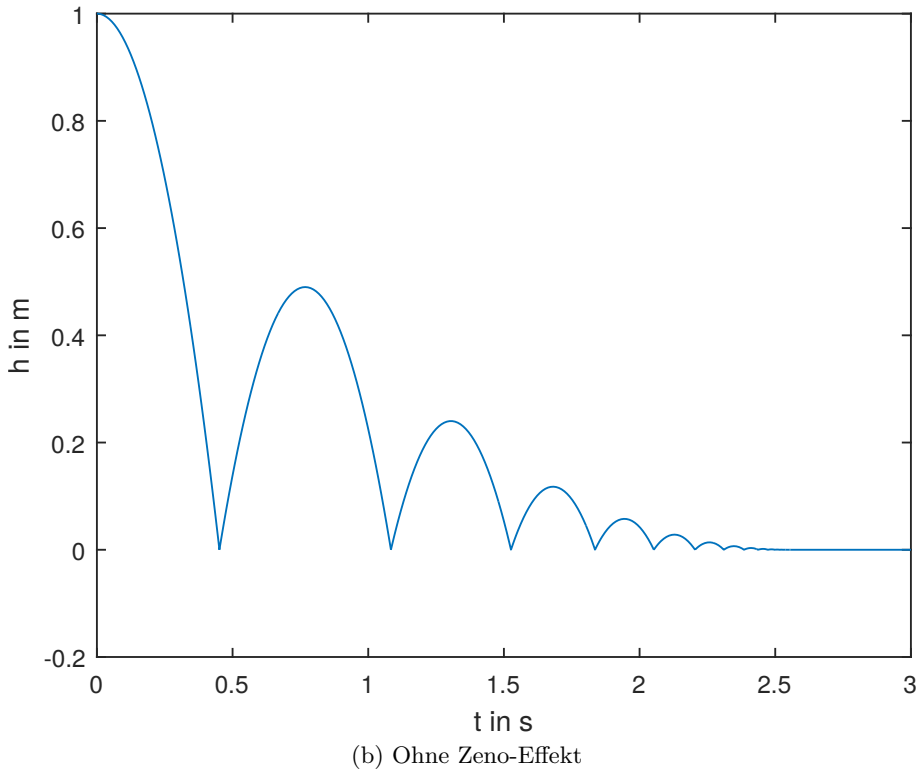
```
1  % Simulation of a bouncing ball using a simscape model
2
3  % Simulation parameters
4  k     = 0.7;
5  g     = 9.81;     % m/s^2
6  h0    = 1;        % m
7  v0    = 0;        % m/s
8  tend  = 3;        % s
9  epsilon = 1e-5; % tolerance
10
11 % Initialization
12 t   = [];
13 h   = [];
14 v   = [];
15 t0 = 0; % s
```

```
16
17 % Simulation loop
18 while t0 < tend
19   %simulate until impact
20   simout=sim('matlabBallModel.slx','StartTime','t0','StopTime','tend')
         ;
21
22   %get results
23   tout = simout.get('tout');
24   hout = simout.get('hout');
25   vout = simout.get('vout');
26   t = [t; tout];
27   h = [h; hout];
28   v = [v; vout];
29   fprintf('simulation stopped at t=%f\n',tout(end));
30
31   %set new initial values
32   t0 = tout(end);
33   h0 = hout(end);
34   v0 = -k*vout(end);
35
36   % check if new iteration is necessray
37   if abs(v(end)) <= epsilon,
38     t = [t; tend];
39     h = [h; 0];
40     v = [v; 0];
41     break;
42   end
43 end
```

Dieses Script besteht hauptsächlich aus einer *while*-Schleife, die das Simulationsmodell solange aufruft, bis die maximale Laufzeit der Simulation erreicht ist. Auch hier kommt es zum Zeno-Effekt und dabei wird eine Schwäche dieser Methode sichtbar: Der wiederholte Neustart der Simulation benötigt sehr viel Zeit. Nach Auftreten des Zeno-Effekts arbeitet der Solver *ODE15s*, sofern nicht anders eingestellt, mit sehr kleiner Schrittweite. Abbilung 4.16a zeigt, daß nach jeder Iteration die Abbruchbedingung des Simulationsmodells erfüllt ist und für jede weitere Iteration neugestartet werden muß, was unverhältnismäßig viel Zeit in Anspruch nimmt. Um dem entgegen zu wirken, wurde der Schleife eine weitere Abbruchbedingung hinzugefügt: Unterschreitet die Geschwindigkeit den Grenzwert *epsilon*, wird die Simulation beendet.



(a) Mit Zeno-Effekt



(b) Ohne Zeno-Effekt

Abbildung 4.16: Simulationsergebnis des Beispiels *bouncingBall in* SimScape

5 Zusammenfassung und Ausblick

Alle in dieser Arbeit untersuchten Beispiele ließen sich sowohl in OpenModelica als auch in SimScape umsetzen. Es wird aber schnell sichtbar, daß es sich bei Modelica um eine eigenständige Sprache und bei SimScape nur um eine Erweiterung zu einem bestehenden System handelt. Entsprechend ist Modelica oftmals einfacher und flexibler einsetzbar und die Verknüpfung von SimScape mit Matlab und Simulink, was von Mathworks als großer Vorteil beworben wird, wird selbst bei einfachen Anwebdungen schnell zur zwingenden Notwendigkeit. Was an der derzeitigen Version von OpenModelica von Nachteil ist, ist der sehr strenge Parser, der überall eine genaue Anzahl von Leerzeichen zwischen allen Symbolen vorschreibt. Das in Verbinung mit den Formatierungsbefehlen, die in Modelica sehr lang sein können, kann den Quellcode schnell sehr unübersichtlich machen. Zudem können die vielschichtigen Vererbungsbeziehungen zwischen den einzelnen Objekten in Modelica stellenweise irritieren. Die strenge Verkapselung in SimScape ist da besser beherrschbar.

Ein hinreichender Vergleich bezüglich der Möglichkeiten zwischen OpenModelica und SimScape ist mit dieser Arbeit noch nicht erreicht. Zukünfitig können die Unterstützung von Mehrkörpersystemen, multiphysikalischen Systemen und die Kombination von kontinuierlichen Systemen mit Zustandsautomaten untersucht werden.

Literaturverzeichnis

[ebo13] *Modellbildung mit SIMULINK und SimScape.* eBook. http://www.ebookaktiv.de/. Version: 2013

[Fet12] Fettke, Peter: *Objektorientierte Modellierung.* Internetlexikon. http://www.enzyklopaedie-der-wirtschaftsinformatik.de/. Version: August 2012

[Fri11] Fritzson, Peter: *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica.* Wiley-IEEE Press, 2011

[Mat15] Mathworks (Hrsg.): *Matlab Documentation.* Mathworks, Juli 2015. http://de.mathworks.com/help/physmod/simscape/getting-started-with-simscape.html

[Mod12] Modelica Association (Hrsg.): *Language Specification Version 3.3.* Modelica Association, Mai 2012. https://www.modelica.org/documents/ModelicaSpec33.pdf

[Myr12] Myrach, Thomas: *Datenflussmodellierung.* Internetlexikon. http://www.enzyklopaedie-der-wirtschaftsinformatik.de/. Version: Oktober 2012

[OMW15] *Open Modelica Webseite.* Internetseite. https://www.openmodelica.org/. Version: Juli 2015