Introduction to Physical Modeling with Modelica

THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE

INTRODUCTION TO PHYSICAL MODELING WITH MODELICA

MICHAEL TILLER, PH.D.
Technical Specialist, Ford Motor Company
Member, Modelica Association

Springer Science+Business Media, LLC

Library of Congress Cataloging-in-Publication Data

Tiller, Michael, Ph.D.
Introduction to physical modeling with Modelica / Michael Tiller.
p. cm. – (The Kluwer international series in engineering and computer science ; SECS 615)
Includes bibliographical references and index.
Additional material to this book can be downloaded from http://extras.springer.com.

ISBN 978-1-4613-5615-8 ISBN 978-1-4615-1561-6 (eBook)
DOI 10.1007/978-1-4615-1561-6
1. Computer simulation. 2. Object-oriented methods (Computer science) I. Title. Series.

QA76.9.C65 T55 2001
005'.35133—dc21

2001029416

Originally published by Kluwer Academic Publishers in 2001
Softcover reprint of the hardcover 1st edition 2001
Second Printing 2004.

Printed on acid-free paper.

Contents

List of Figures

List of Tables

List of Examples

Preface

In writing this book, my goal is to demonstrate how easy, useful and fun, the modeling of physical systems can be. For me, there is nothing that a computer can be used for that is more interesting than simulating the behavior of physical systems. The term "physical systems" refers to the behavior of physics-based models found across many disciplines (*e.g.,* electrical engineering, mechanical engineering, chemistry, physics). Such systems can be identified by their use of conservation principles (*e.g.,* first law of thermodynamics and conservation of mass).

In this book I will describe how the Modelica modeling language can be used to describe the behavior of physical systems. Modelica can be used for a wide range of applications from simple systems with only a few degrees of freedom all the way up to complex systems made of large networks of reusable components.

The first part of the book is focused on introducing the reader to the Modelica modeling language. The target audience would be somebody with an understanding of basic physics and calculus, an interest in modeling and no knowledge of Modelica. The intent is to cover all the basics of the language using simple examples and enable the reader to begin writing models in Modelica.

Each chapter in the first part of the book starts with an overview of the important concepts the chapter introduces. Whenever a new term is introduced it will appear *italicized* and a definition for it will be included in the glossary. The overview is then followed by a series of examples meant to gradually introduce Modelica functionality. I feel that examples are an important part of the learning process. I have tried to avoid using contrived examples. In fact, many of the examples come from real world problems I have encountered. The difficulty with examples is that they do not introduce material in a structured way, but rather in a "flowing" way. For this reason, many chapters include a "Language Fundamentals" section which attempts to formalize all of the

concepts introduced by the examples. Readers may feel free to skip over the material in the fundamentals section if they feel comfortable with the features presented in that chapter. An important note about the structure of this book is that **each example introduces new concepts**. In other words, do not assume that because you understood the first example in a chapter all the remaining examples are not worth studying.

The second part of the book demonstrates how to most effectively use the powerful features of the Modelica language. This part is intended for people who are already familiar with the basics of the Modelica language, including existing users of Modelica and beginners who have completed the first part.

This book covers nearly all of the features of the Modelica language. However, much of this material is only required in advanced applications. The "core" material required to begin doing meaningful modeling can be found in Chapters 1, 2, 3 and 7. Readers may wish to focus their attention on those chapters first and then consult the other chapters as they become more proficient.

Realize that it is not possible to introduce every nuance of the Modelica language through examples. Once you have covered the material in this book, you will require a definitive reference. The ultimate source of information about Modelica is the language specification itself. For this reason, the Modelica language specification is included on the companion CD-ROM. While not appropriate for learning the language, it is appropriate as a reference on the semantics of the language.

In summary, this book includes material that will have broad appeal and will serve both beginners and experienced users trying to get the most out of physical system modeling.

MICHAEL TILLER

Acknowledgements

I would like to start by thanking my parents who have always encouraged my curiosity. This curiosity has motivated me throughout my life to learn about and understand math, engineering and modeling. In addition, I would also like to thank my wife, Deepa Ramaswamy, for her support during this project.

The material in this book has benefited greatly from the proofreading and technical insights of Hilding Elmqvist, Sven Erik Mattsson, Hans Olsson, Deepa Ramaswamy, Michael R. Tiller, Martin Otter, Dag Brück and Paul Bowles.

This book is built on the foundations of the Modelica language itself. As such, the members of the Modelica Association (designers of the Modelica language, see Appendix A) deserve the credit for formulating such an elegant and powerful modeling language. I would also like to thank Hilding Elmqvist, Sven Erik Mattsson, Hans Olsson and Dag Brück for their work on the Dymola software used during the writing of this book and for contributing an evaluation copy of Dymola for inclusion with this book.

Learning is not possible without people willing to teach. I would like to thank the people I have worked with and learned from over the years for all I have learned from them. I hope that the material in this book inspires the reader's curiosity in the same way that the following people have inspired mine: Michael R. Tiller, Eunice Tiller, Raimond Winslow, Robert F. Miller, Anthony Lee Kimball, Anthony Varghese, Peter Steinmetz, Kim Stelson, Nicholas Zabaras, Jon Dantzig, Daniel Tortorelli, Jamshid Ghaboussi, Thomas Kerkhoven, Charles L. Tucker III, Ralph E. Johnson, Robert McDavid, Chuck Newman, George Davis, William Tobler, Hilding Elmqvist and Martin Otter

I have had the opportunity, over the last few years, to work with several very bright and energetic individuals on Modelica related projects. Cleon Davis helped to develop the initial Modelica models used at Ford Motor Company. In addition, I would like to thank Hubertus Tummescheit for the many long discussions we have had on approaches to thermodynamic modeling in Model-

ica. Finally, Paul Bowles was my co-author on several papers that were among the first papers to demonstrate the scalability of the Modelica approach. His contribution to internal projects at Ford and subsequent publications on that work have been essential to their success.

I would like to close by pointing out all of the open source tools I have used in the preparation of this book. I would like to show my appreciation to the authors of Grace, xfig, transfig, XEmacs, CVS, TkCVS, WinCVS, Kdvi, Ghostview, Ghostscript, TEX, LATEX, AucTeX, MikTeX, Linux, KDE and Gnome. It should be pointed out that all the source code listings of Modelica models in this book were done with the LATEX `listings` package by Carsten Heinz.

I
THE MODELICA LANGUAGE

Chapter 1

INTRODUCTION

1.1 WHAT IS MODELICA?

Before diving into the details of modeling using Modelica, let me provide a brief description of what Modelica is, why it was developed and what it is used for.

Since the invention of the computer, modeling and simulation have been an important part of computing. Initially, modelers were burdened with converting their models into systems of ordinary differential equations (ODEs) and then writing code to integrate those differential equations in order to run simulations. Eventually, a wide range of integrators were developed as independent software units and modelers were able to focus on the formulation of differential equations and use "off-the-shelf" integrators for simulation. This trend of allowing modelers to focus more on the behavioral description of their problems and less on the solution methods has continued ever since.

In the last three decades, numerous tools have been developed to assist modelers in performing simulations. Some of these were general purpose simulation tools such as ACSL[1], Easy5[2], SystemBuild[3] and Simulink.[4] Other tools were developed for simulations in specific engineering domains such as electrical circuits (*e.g.,* Spice[5]), multi-body systems (*e.g.,* ADAMS[6]) or chemical processes (*e.g.,* ASPEN Plus[7]). Each type of tool has its advantages.

[1] ACSL is a trademark of The AEgis Technologies Group, Inc.
[2] Easy5 is a trademark of The Boeing Company.
[3] SystemBuild is a trademark of Wind River Systems, Inc.
[4] Simulink is a trademark of The MathWorks, Inc.
[5] Spice is a trademark of the University of California at Berkeley.
[6] ADAMS is a trademark of Mechanical Dynamics, Inc.
[7] ASPEN Plus is a trademark of Aspen Technologies, Inc.

For example, general purpose tools do not restrict modelers to a particular domain but they may require the modeler to spend some time formulating their models for that particular tool. Likewise, tools developed for a specific engineering domain have numerical methods and graphical user interfaces which are optimal for that particular domain but they restrict the ability of the modeler to create mixed-domain models.

In 1978, Hilding Elmqvist pioneered, as part of his Ph.D. thesis, a new approach to modeling physical systems by designing and implementing the Dymola modeling language (Elmqvist, 1978). The basic idea behind the Dymola modeling language was to use general equations, objects and connections to allow model developers to look at modeling from a physical perspective instead of a mathematical one.[8] For the Dymola implementation, graph theoretical and symbolic algorithms were introduced to transform the model to an appropriate form for numerical solvers. An important milestone in the development of this approach came in 1988 with the development of the Pantelides algorithm for DAE index reduction (Pantelides, 1988). Following Dymola, numerous other tools (*e.g.*, Omola, see Mattsson et al., 1993) were developed to further explore this new approach to modeling.

A major problem with all simulation tools has been that models developed using one tool could not be used by another. In 1996, Hilding Elmqvist initiated an effort to unify the splintered landscape of modeling languages by initiating the development of the Modelica modeling language. Similar initiatives have been undertaken by various other groups (see Heinkel et al., 2000 and Fitzpatrick and Miller, 1995) but these efforts have been focused primarily on the electrical domain, while Modelica strives to be completely domain neutral.

The basic idea behind Modelica was to create a modeling language that could express the behavior of models from a wide range of engineering domains without limiting those models to a particular commercial tool. In other words, Modelica is both a modeling language and a model exchange specification. To accomplish this goal, the developers of previous object-oriented modeling languages like Allan, Dymola, NMF, ObjectMath, Omola, SIDOPS+ and Smile were brought together with experts from many engineering domains to create the specification for the Modelica language based on their wide range of experiences.[9]

Modelica can be used to solve a variety of problems that can be expressed in terms of differential-algebraic equations (DAEs) describing the behavior of continuous variables. The ability to formulate problems as DAEs rather than ODEs reduces the burden on the model developer because less effort is

[8] The physical and mathematical approaches are contrasted in Chapter 11.

[9] A detailed history of how the Modelica modeling language was developed is contained in Appendix A.

involved in formulating equations. In addition to handling continuous variables, Modelica includes features for describing the behavior of discrete variables (*e.g.*, digital signals). Often, it is convenient or even necessary to simulate both continuous and discrete behavior at the same time. Modelica allows both forms of behavior to be described within the same system model or even the same *component* model.

Modelica is a non-proprietary modeling language and the name is a trademark of the Modelica Association which is responsible for publication of the Modelica language specification. At present, Modelica is not an ISO, ANSI or IEEE standard. This means that Modelica is presently a "moving target" in much the same way as *C++* was for about a decade. In the case of *C++*, avoiding the rush to standardize did not prevent people from making use of the language and ultimately led to a much better language. Hopefully, Modelica will follow a similar path. If a need can be demonstrated for functionality not already present in the Modelica language, users can work with the Modelica Association to fill functionality gaps. The current Modelica specification can be found at the Modelica Association web site: `http://www.modelica.org`. Version 1.4 of the Modelica specification is included on the companion CD-ROM.

If you have ever been involved in large scale modeling projects you probably recognize that model development is in many ways similar to large scale software development. Just like a programming language, the purpose of a modeling language is to describe the behavior of small pieces of a larger system. A modeling language should encourage reuse of previous work and help manage the complexity of systems as they become larger. It should be possible, once a reusable set of components has been created, to work at an increasingly higher level (*i.e.*, getting away from writing equations at the component level and working more on the assembly of a complex system). Ultimately, this leads to the ability to build systems using a "top-down" approach rather than a "bottom-up" approach.

All simulation results presented in this book were generated using Dymola (Dynamic Modeling Laboratory).[10] An evaluation copy of Dymola is provided by Dynasim (Elmqvist et al., 2001) on the companion CD-ROM so that readers may gain hands-on experience with using the Modelica language. To understand how to simulate Modelica models using Dymola, please read the documentation titled "Getting Started with Dymola" which is included with the Dymola software. Dymola can also be used to generate models that can be imported into Simulink.

[10]Dymola is a trademark of Dynasim AB.

1.2 WHAT CAN MODELICA BE USED FOR?

Modelica can be used for many things, including simulation of electrical circuits (Clauss et al., 2000), automotive powertrains (Otter et al., 2000), power system stability (Larsson, 2000), vehicle dynamics (Tiller et al., 2000) and hydraulic systems (Beater, 2000). However, the best way to understand what Modelica can be used for is through an example. While most of the chapters in the book use relatively simple examples to highlight specific language features, we will start by giving a glimpse of "the big picture".

In this section we will show bits and pieces of a substantial library of Modelica models for simulating automobile performance. The library was developed for this book to demonstrate how reasonably complex systems can be modeled. While the library contains a large number of models, most of the models are quite simple. Because these models are relatively simple, they will give us only a rough estimate of how particular automobile designs will perform. The Modelica models from this section are provided on the companion CD-ROM and discussed in greater detail in Chapter 10.

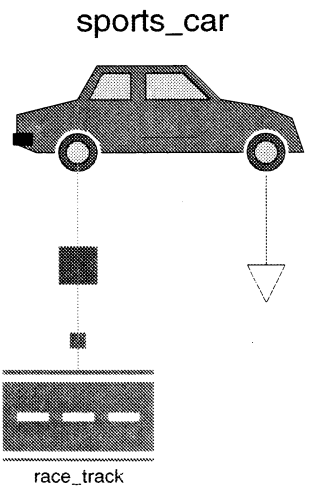

Figure 1.1. A 0-100 kilometer per hour test.

Imagine we wish to predict the acceleration performance for a particular sports car design. In order to judge the performance, we will measure the time it takes the vehicle to accelerate from zero to one hundred kilometers per hour. Figure 1.1 shows our performance test which includes a sports car and a race track.

Do not be fooled into thinking the model we are simulating is not detailed just because the picture looks simple. This is just the top-level view of the problem.

Figure 1.2 shows what we find if we look inside our sports car model. Behind the scenes, the sports car model includes models of the chassis, transmission and engine as well as a shifting strategy that decides when to change gears. Behind all of these images are behavioral models (*i.e.*, the images themselves are just used to help identify what the models represent). As we shall see, even this view of the sports car gives a deceptively simple impression.

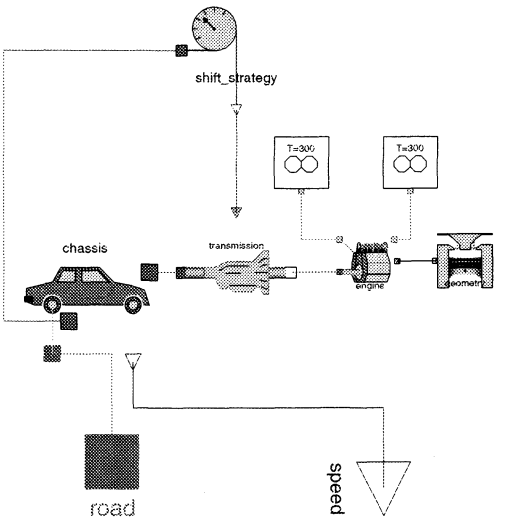

Figure 1.2. Taking a look at what is under the hood.

The engine model for our sports car is one of many components in Figure 1.2. If we open up the engine model we can see each of the four individual cylinders (shown in Figure 1.3). Again, the images of engine cylinders are graphics added to the models so they can be easily identified as engine cylinders. Behind each of these pictures is a detailed schematic of the components used to model an individual engine cylinder.

If we open up one of these cylinders, we find the numerous low-level component models shown in Figure 1.4. By zooming in to each of the various models shown so far, we have gone from the complete vehicle level (shown in Figure 1.1) all the way down to models of individual components such as engine valves (shown in Figure 1.4). The ability to construct such hierarchies is a central feature of Modelica. In addition, the ability to include graphical representations for the models, as we have seen in these figures, is also a feature provided by Modelica.

Each of the graphics shown in Figure 1.4 represents a component involved in the function of an individual engine cylinder. We cannot "zoom" into these

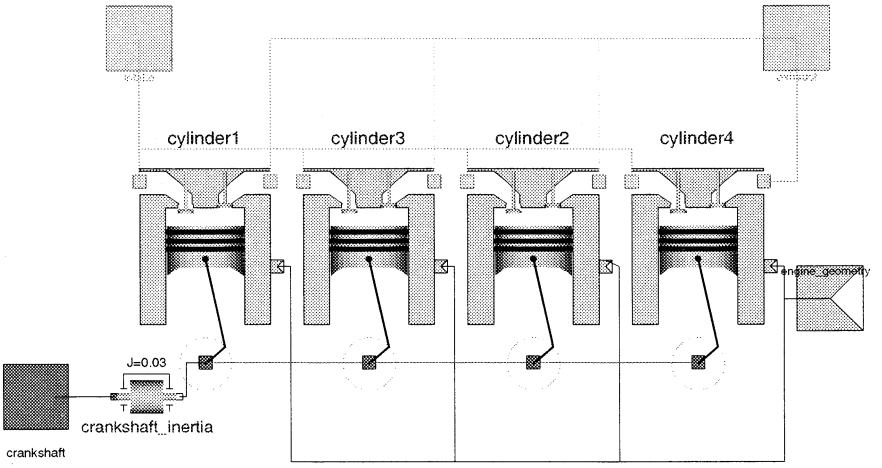

Figure 1.3. Looking inside the engine.

models because they represent the smallest pieces in the system. In a sense, they are the "atoms" of our system. It is important to understand that **these pieces are not magical primitives** that just happen to come with the software package we used to build this model. In fact, it is at this component level that we turn our attention away from all the graphics toward the real subject of this book: the Modelica modeling language. Previously, we have seen how the Modelica modeling language can be used to describe hierarchies of components. At the "atomic" level, it can also be used to describe the behavior of each of these components. The remainder of the book will provide all the necessary information to build such components and an enormous variety of other components in other engineering domains.

Building models is fun, but ultimately we want to see results from such models. When we run our simulation, we find that the sports car model presented in this section can go from zero to 100 kilometers per hour in 6.88 seconds. Figure 1.5 shows several different pieces of information recorded during the test. Notice how the transmission gear changes at different vehicle speeds. We can also see how the engine speed increases up until the transmission shifts and then it drops again. These are just a handful of signals we can extract from our simulation. Other useful pieces of information available include manifold pressure, trapped mass in the cylinder, traction force on the tires, transmission clutch pressures, *etc.* Studying such information can provide important insights during the design process.

Once we have a model that gives us good results, the next logical step is to ask ourselves "what if?". The sports car in our race model includes numerous

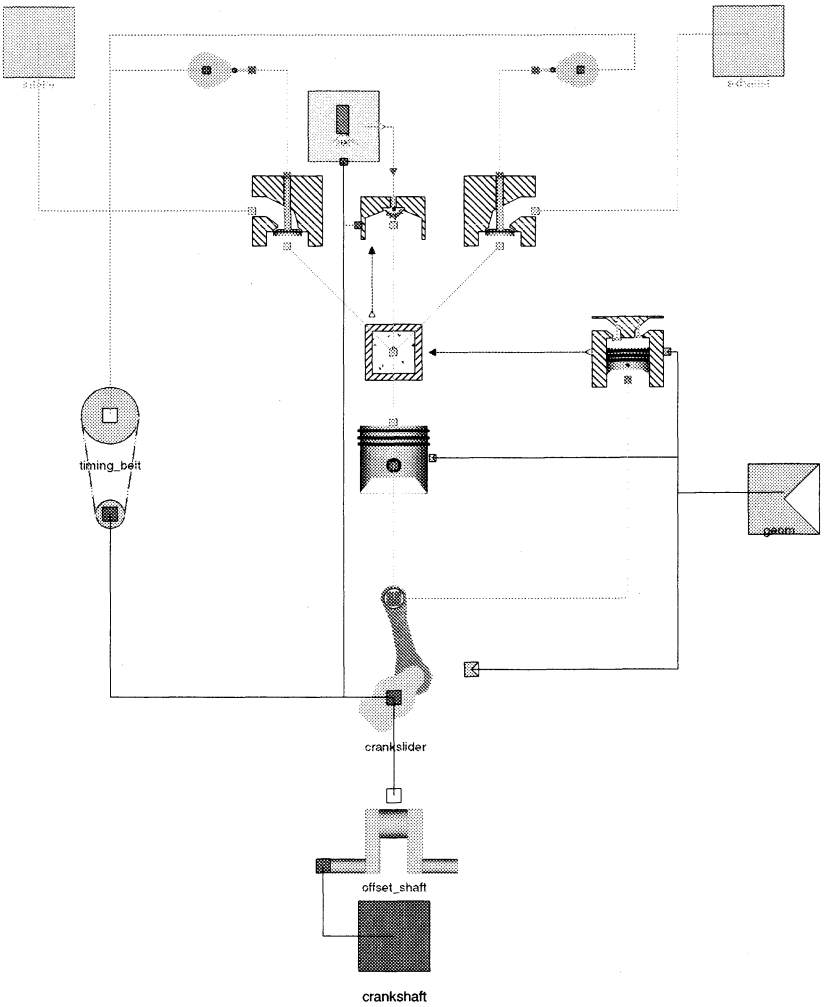

Figure 1.4. Looking inside an individual engine cylinder.

design details. For example, we can easily specify the engine geometry, valve timing, shift schedule, vehicle weight, tire radius, and so on. By changing these values, we can determine the impact each of these parameters has on overall system performance.

Remember, Modelica is a domain-neutral modeling language useful for creating models from nearly any engineering domain. The remainder of the book shows how models from many other engineering domains can be created using the Modelica modeling language.

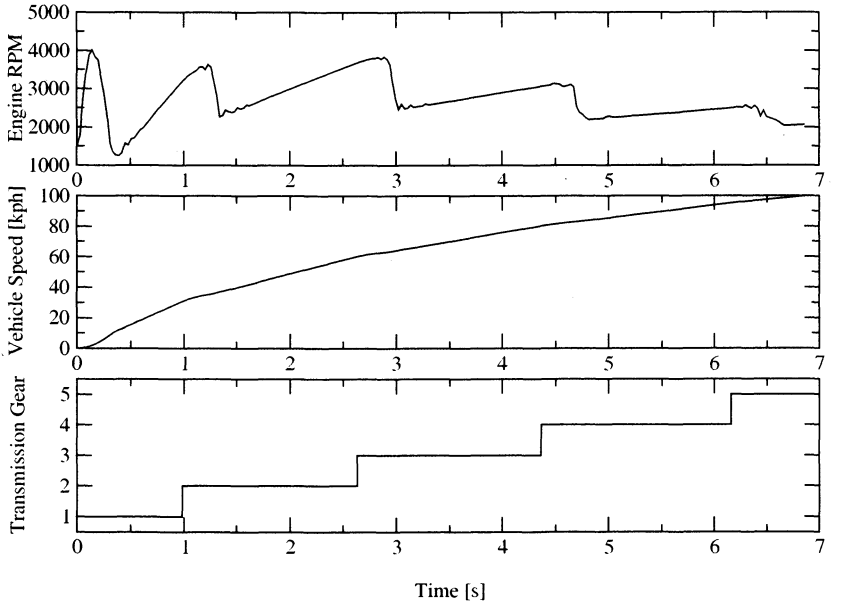

Figure 1.5. Simulation results from a sample race.

1.3 MODELING FORMALISMS

Before we start discussing how to use Modelica to develop models, let us take a moment to talk about modeling in general. There are many formalisms used for modeling continuous systems. An excellent overview of different formalisms is presented in Åström et al., 1998. Modelica supports two of the common approaches to modeling in engineering.[11] The first is called *block diagram* modeling and the other is called *acausal* modeling.[12] In this section we will discuss block diagrams and acausal formulations to better understand the differences between them.

1.3.1 Block diagrams

Using block diagrams, a system is described in terms of quantities that are known and quantities that are unknown. A block diagram consists of components, called blocks, which use the known quantities to compute the unknown quantities. A block diagram of a PI (proportional-integral) controller is shown in Figure 1.6.

[11] In addition, other formalisms like bond graphs (see Cellier, 1991) and petri nets can also be described in Modelica.

[12] Acausal modeling is sometimes referred to as *first principles* modeling.

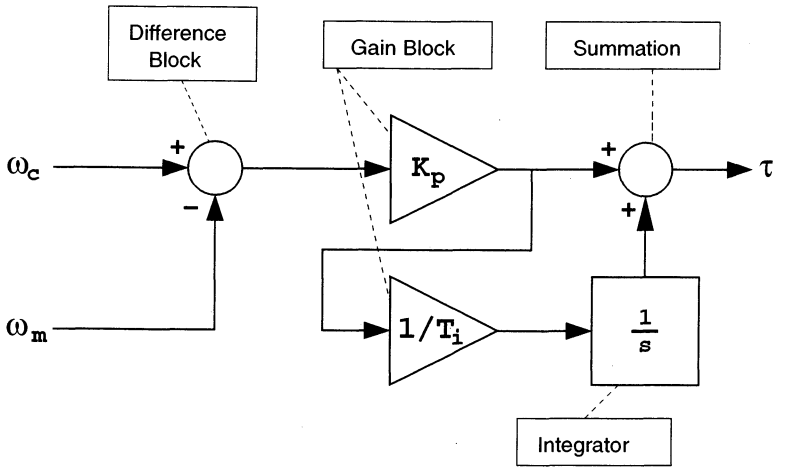

Figure 1.6. PI Controller.

On the left side of Figure 1.6 are the known quantities ω_c (the desired speed), and ω_m (the actual motor speed as read by the speed sensor). On the right side of Figure 1.6, the torque used to control the system is computed. In between are the *blocks* which describe the computations being performed. In this example, the difference block takes the desired and sensed speed as an input and computes as an output the difference (*i.e.,* the error). One gain block then multiplies the speed difference by the gain, K_p. The scaled speed difference is passed through another gain block, scaled by $\frac{1}{T_i}$ and integrated. We compute the control torque by summing the outputs from the gain blocks.

This approach to modeling is often used when designing control systems. For example, tools such as Simulink and SystemBuild use this approach. A block diagram is a natural way of expressing a control system design. However, such diagrams have their limitations as we shall demonstrate in Chapter 11.

1.3.2 Acausal modeling

Describing system or component behavior in terms of conservation laws is referred to as acausal modeling. With acausal formulations, there is no explicit specification of system inputs and outputs. Instead, the *constitutive equations* of components (*e.g.,* Ohm's law for a resistor) are combined with *conservation equations* to determine the overall system of equations to be solved. For example, when modeling electrical systems, like the circuit shown in Figure 1.7, one can use *Kirchhoff's current law* (a conservation law), which states that the sum of the currents into a particular node (in this case, a, b or c) must be zero. The application of conservation laws results, in general, in

systems of differential-algebraic equations (DAEs). Dymola and Saber[13] are two examples of tools that allow acausal formulations.

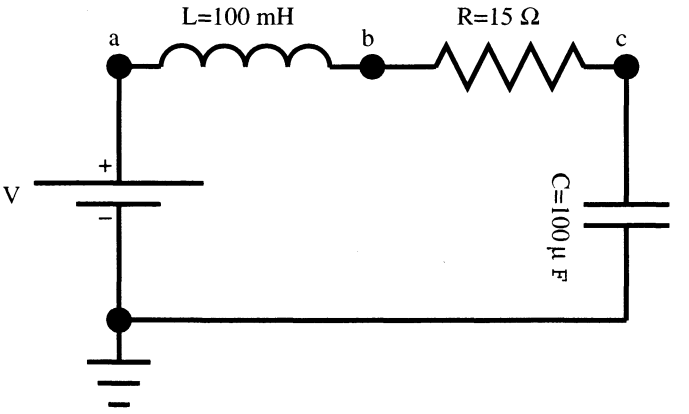

Figure 1.7. RLC circuit schematic.

In order to formulate acausal models, it is useful to identify the *through variables* and the *across variables* for the component being modeled. In general, the across variable represents the driving force in the system and the through variable represents the flow of some conserved quantity. For an electrical system, the voltage is the across variable and the current is the through variable. Note that the product of the through variable and the across variable typically has the units of power (*i.e.*, Watts in SI units). Table 1.1 includes several examples of through and across variables for different engineering domains.

Domain	*Through*	*Across*
Electrical	Current (A)	Voltage (V)
Mechanical (translational)	Force (N)	Velocity (m/s)
Mechanical (rotational)	Torque (Nm)	Angular Velocity (rad/s)
Hydraulic	Flow Rate (m^3/s)	Pressure (N/m^2)

Table 1.1. Through and across variables from various domains.

1.3.3 Further remarks on formalisms

As we shall demonstrate in Chapter 11, block diagrams are convenient for control system modeling and acausal formulations are convenient for physical

[13] Saber is a trademark of Avant! Corporation.

system modeling (*i.e.*, plant modeling). Not only does Modelica support both of these important types of modeling, but it allows both of them to be used together.

1.4 MODELICA STANDARD LIBRARY

In addition to defining the specification for the Modelica language, the Modelica Association also publishes a standard library of Modelica models. This library, called the Modelica Standard Library (or MSL), is available free of charge.[14]

The MSL was developed so that users of the Modelica language would not have to create their own basic models for the common modeling domains. Throughout this book, we start off by developing Modelica models from scratch to demonstrate the fundamentals of the language. Then, we point out similar models which already exist within the MSL. In this way, we can cover language fundamentals and models available in the MSL.

Keep in mind that the MSL is not a collection of *black box* models which are hard-wired into a tool. Instead, the Modelica representation of all the models can be viewed to help understand exactly what behavior is modeled. These models are no different than any other Modelica models. It should be noted that while the models contained in the MSL are useful, you are not required to use them.

While reading this book, be on the lookout for uses of the MSL. These can be easily recognized by looking for names that begin with "`Modelica.`". All such entities belong to the MSL. For example, the physical type `Modelica.SIunits.Voltage` is defined in the MSL. You should interpret this name to mean "`Voltage` is a type defined in the `SIunits` package nested inside the `Modelica` package". The `package` structure of Modelica libraries (including the MSL) is hierarchical and may contain numerous nested packages. Do not be surprised to see much longer names like:

```
Modelica.Electrical.Analog.Basic.Resistor
```

1.5 BASIC VOCABULARY

The Modelica language specification uses a precise vocabulary for describing the elements of the Modelica language. While being rigorous is necessary in a formal specification, it is not always good in learning material. For this reason, this book uses a simplified vocabulary. In the remaining chapters, the following terms are used:

[14] As with most things related to Modelica, the MSL can be found at `http://www.modelica.org`

model A model is a behavioral description. For example, a model of a resistor is described by Ohm's law. The model is a description of resistor behavior, not the resistor itself. In other words, it is important to separate the idea of a resistor model (*i.e.*, $V = I*R$) from the resistor instances (components with different values of resistance, R). If you are familiar with object-oriented programming, a model is analogous to a class.

component A component is an instance of a model. So, for a given model (*e.g.*, a resistor model), the actual instances (*e.g.*, the resistors) would be components.

subcomponent A subcomponent is used to refer to components which are contained within other components. For example, a resistor might be a subcomponent of another component like an electrical circuit. Furthermore, the electrical circuit could be a subcomponent of an appliance. Subcomponents are used to form hierarchical models.

system model A system model is a model which is completely self-contained. In other words, it does not have any external connections and it contains the same number of equations as unknowns.

quantity A quantity refers to those entities which have a value (*e.g.*, the resistance of a resistor). In Modelica, all values are either real, integer, string or boolean. Furthermore, a quantity might be a scalar or an array.

definition The description of all variables, parameters and equations associated with a model is called the model definition.

declaration When a component, parameter, variable or constant is instantiated (either in a system model or inside another component), that is called a declaration.

package A package refers to a collection of Modelica models, which are meant to be used together. For example, an electrical package would likely include definitions of resistor, capacitor and inductor models.

keyword A keyword is a word, such as `model`, that has a specific meaning in Modelica. As a result, keywords are reserved words and cannot be used as names in declarations (*e.g.*, of variables). In the examples, the keywords will appear in bold.

Use the explanations of these terms as a reference to help understand the more complicated explanations in this book. The glossary, which starts on page 324, includes these terms and many more used in this book.

1.6 SUMMARY

In summary, the Modelica language is a non-proprietary, domain-neutral modeling language that supports several different modeling formalisms. Modelica can be used to model both continuous and discrete behavior and an extensive multi-domain library of models known as the Modelica Standard Library is available free of charge at `http://www.modelica.org`.

Chapter 2

DIFFERENTIAL EQUATIONS

2.1 CONCEPTS

Modelica is a powerful language for describing the behavior of dynamic systems. At the heart of any model are mathematical equations. We begin our discussion of Modelica by showing how simple systems of differential equations can be expressed using Modelica. The expression of differential equations is the most basic example of Modelica's capabilities. Subsequent chapters will use increasingly complex models to demonstrate how more advanced features help model detailed physical systems, manage system complexity and promote reuse of models.

In this chapter, we will demonstrate how to write some simple models which include parameters, continuous variables and equations. These examples should provide enough information to allow readers to begin creating their own simple models. **Remember that each of the examples introduces new concepts.** The final section of this chapter provides a comprehensive review of the language features covered in this chapter.

2.2 DIFFERENTIAL EQUATIONS

2.2.1 Equations of motion

Let us consider the motion of a pendulum like the one shown in Figure 2.1. From Euler's second law we know that the sum of the torques about a fixed point must be equal to zero. There are two torques applied at the pivot point, x, in Figure 2.1:

$$\tau_g = mgL \sin(\theta) \tag{2.1}$$

$$\tau_i = mL^2\ddot{\theta} \tag{2.2}$$

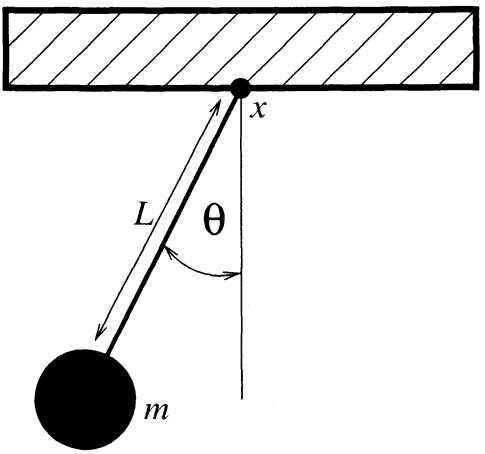

Figure 2.1. A simple pendulum

where θ is the angular position (relative to gravity), L is the length of the pendulum, m is the mass of the pendulum, g is the acceleration due to Earth's gravity, τ_g is the torque due to gravity and τ_i is the inertial torque. Using the fact that the sum of the torques about the pivot point, x, must be zero, we get:

$$\tau_g + \tau_i = mgL \sin(\theta) + mL^2\ddot{\theta} = 0 \tag{2.3}$$

which we can further reduce to

$$\ddot{\theta}(t) = -\frac{g}{L} \sin(\theta(t)) \tag{2.4}$$

Finally, one simplifying assumption we can make, for the time being, is to assume that θ is small which means we can approximate $\sin(\theta)$ as just θ. In this case, our differential equation becomes simply:

$$\ddot{\theta}(t) = -\frac{g}{L}\theta(t) \tag{2.5}$$

Let us transform Equation (2.5) into a system of first-order ordinary differential equations (ODEs):

$$\begin{pmatrix} \dot{\theta} \\ \dot{\omega} \end{pmatrix} = \begin{pmatrix} \omega \\ -\frac{g}{L}\theta \end{pmatrix} \tag{2.6}$$

where ω is the angular velocity of the pendulum. Given initial values for ω and θ, Equation (2.6) can be integrated to obtain the behavior of the pendulum as a function of time.

```
model SimplePendulum
  parameter Real L=2;
  constant Real g=9.81;
  Real theta;
  Real omega;
equation
  der(theta) = omega;
  der(omega) = -(g/L)*theta;
end SimplePendulum;
```

Example 2.1. Model of a simple pendulum.

2.2.2 Modelica model

Example 2.1 shows how we can use Modelica to represent the behavior of the pendulum in Figure 2.1. We start by using the keyword `model` followed by the name of our model, `SimplePendulum`. Next, we define the parameters and constants that characterize our model as well as the variables which appear in our equations. The parameters are quantities which remain constant during a simulation but may have different values from one simulation to another (*e.g.*, L). The variables in a problem are those quantities which are a function of time (*e.g.*, θ and ω). Lastly, constants are those quantities, like the acceleration due to gravity, which are unlikely to change. To complete the model, an `equation` section is created which includes the equations shown in Equation (2.6).

Note that the parameter quantities in Example 2.1 have the `parameter` keyword in front of them. Likewise, constants are identified by the use of the `constant` keyword. Since the declarations of `omega` and `theta` are not qualified by `parameter` or `constant`, they are assumed to be variables. All the quantities we have described are of type `Real` which means they are real numbers (as opposed to integers, for example).

Examining the `equation` more closely, we see that Modelica includes a built-in operator called `der` which is used to represent the time derivative of a variable. Example 2.1 describes a complete set of first-order ordinary differential equations with two equations and two unknowns. Figure 2.2 shows the simulated solution of Example 2.1.

Now let us reconsider the assumption that $\theta \approx \sin(\theta)$. If we anticipate seeing a wide range of motion for our pendulum, we would use the following non-linear system of differential equations:

$$\begin{pmatrix} \dot{\theta} \\ \dot{\omega} \end{pmatrix} = \begin{pmatrix} \omega \\ -\frac{g}{L}\sin(\theta) \end{pmatrix} \tag{2.7}$$

Example 2.2 shows that only a simple change is required to the Modelica model. Apart from changing the model name, the only other change is to use the

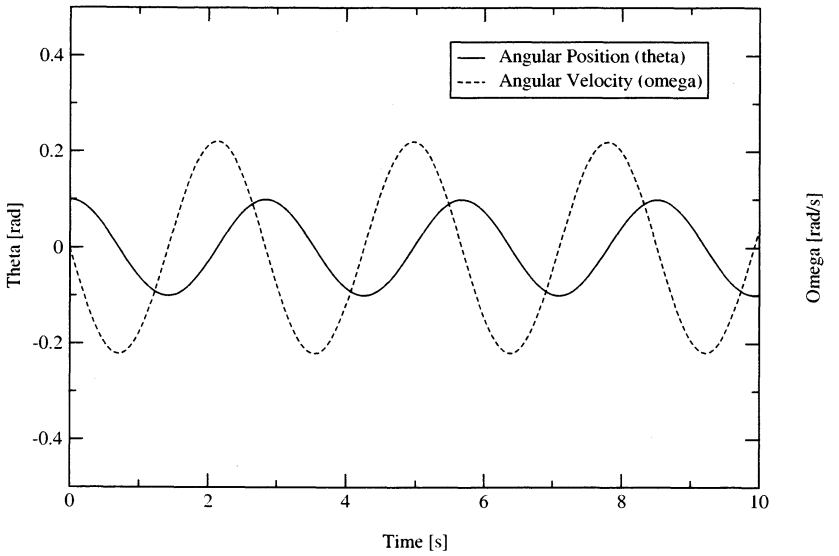

Figure 2.2. Solution for $\theta(t)$ given L=2, $\theta(0) = 0.1$ and $\omega(0) = 0$.

`Modelica.Math.sin` function. If we were to plot the linear and non-linear models for small displacements (such as shown in Figure 2.2), you would not expect to be able to see the difference. However, Figure 2.3 demonstrates that for large displacements there is a significant difference between these two models.

```
model NonlinearPendulum
  Real theta;
  Real omega;
  parameter Real L=2;
  constant Real g=9.81;
equation
  der(theta) = omega;
  der(omega) = -(g/L)*Modelica.Math.sin(theta);
end NonlinearPendulum;
```

Example 2.2. Model of a pendulum without linear assumption.

This simple example provides a good framework to demonstrate the basic features of Modelica.

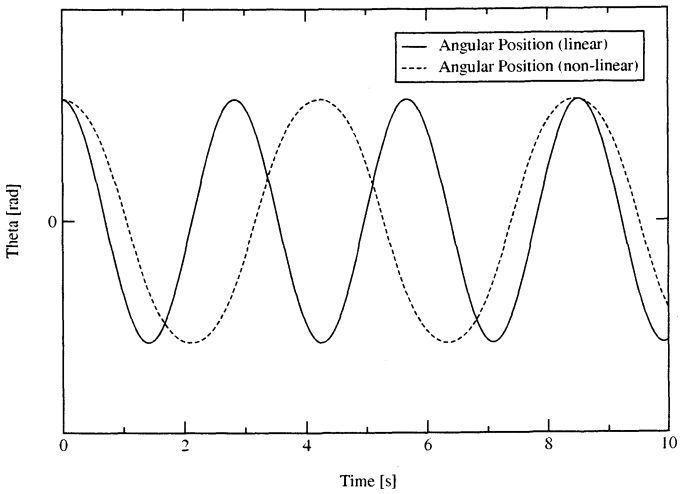

Figure 2.3. Linear and non-linear solutions for $\theta(t)$ given L=2, $\theta(0) = 2.3$ and $\omega(0) = 0$.

2.3 PHYSICAL TYPES

Physical modeling involves specifying relationships between various quantities such as voltage, pressure, mass, *etc.* Modelica includes features which allow us to specify physical types (*e.g.,* voltage, pressure, mass) and associate them with quantities in our models. To demonstrate how this is done, we will build a model of an RLC electrical circuit. An RLC circuit contains a resistor, capacitor and inductor and exhibits oscillatory behavior in response to voltage disturbances.

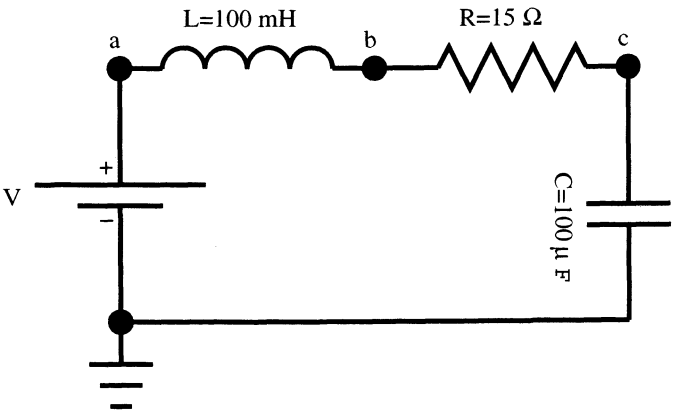

Figure 2.4. An RLC circuit.

2.3.1 Constitutive equations

Figure 2.4 shows the schematic of an RLC circuit. Before we write our Modelica model of this system, we must first write down the equations for each of the components in the system. Unlike the previous example, there is less manipulation of the fundamental equations.

First, let us assume that the voltage source, V, jumps from 0 Volts to 1 Volt after one second of simulation. We can then write an explicit expression for the voltage at node a as follows:

$$V_a = \begin{cases} 0 & : \quad 0 \leq t < 1 \\ 1 & : \quad t \geq 1 \end{cases}$$

Next, we consider the inductor model. The equation for the current through the inductor (from node a to node b) is:

$$L\frac{di_L}{dt} = (V_a - V_b)$$

Likewise, using Ohm's law, the current through the resistor (from node b to node c) can be expressed as:

$$R \cdot i_R = V_b - V_c$$

Finally, the current through the capacitor leaving node c and going to ground can be expressed as:

$$i_C = C\frac{dV_c}{dt}$$

By using Kirchhoff's current law, we know that the sum of the currents going into each node must be zero. This gives us:

$$i_V - i_L = 0 \tag{2.8}$$
$$i_L - i_R = 0 \tag{2.9}$$
$$i_R - i_C = 0 \tag{2.10}$$

Putting this all together, we have the following unknowns:

$$\{V_a, V_b, V_c, i_V, i_R, i_C, i_L\}$$

and the following equations:

$$V_a = \begin{cases} 0 & : \quad 0 \leq t < 1 \\ 1 & : \quad t \geq 1 \end{cases} \tag{2.11}$$

$$L\frac{di_L}{dt} = (V_a - V_b) \tag{2.12}$$
$$R \cdot i_R = V_b - V_c \tag{2.13}$$
$$i_C = C\frac{V_c}{dt} \tag{2.14}$$
$$i_V - i_L = 0 \tag{2.15}$$
$$i_L - i_R = 0 \tag{2.16}$$
$$i_R - i_C = 0 \tag{2.17}$$

Note that we could have simplified these equations further. For example, from Equations (2.15)-(2.17) we know that the current through all the components must be equal to i_V. This would have eliminated the need to solve for i_R, i_C and i_L altogether. For this example, we use all seven equations and all seven unknowns to demonstrate that *a priori* manipulation of the equations is not necessary. Instead, the information given in the model is sufficient for such manipulations to be performed automatically by the simulator.

2.3.2 Modelica model

```
model RLC
  parameter Modelica.SIunits.Resistance R=15;
  parameter Modelica.SIunits.Capacitance C=100e-6;
  parameter Modelica.SIunits.Inductance L=100e-3;

  Modelica.SIunits.Voltage V_a;
  Modelica.SIunits.Voltage V_b;
  Modelica.SIunits.Voltage V_c;
  Modelica.SIunits.Current i_V;
  Modelica.SIunits.Current i_R;
  Modelica.SIunits.Current i_C;
  Modelica.SIunits.Current i_L;
equation
  V_a = if time>=1 then 1.0 else 0.0;
  L*der(i_L) = (V_a - V_b);
  R*i_R = V_b - V_c;
  i_C = C*der(V_c);
  i_V - i_L = 0;
  i_L - i_R = 0;
  i_R - i_C = 0;
end RLC;
```

Example 2.3. Model for an RLC circuit.

The Modelica description of the RLC model is shown in Example 2.3 and the results of simulating this circuit can be seen in Figure 2.5. The model shown in

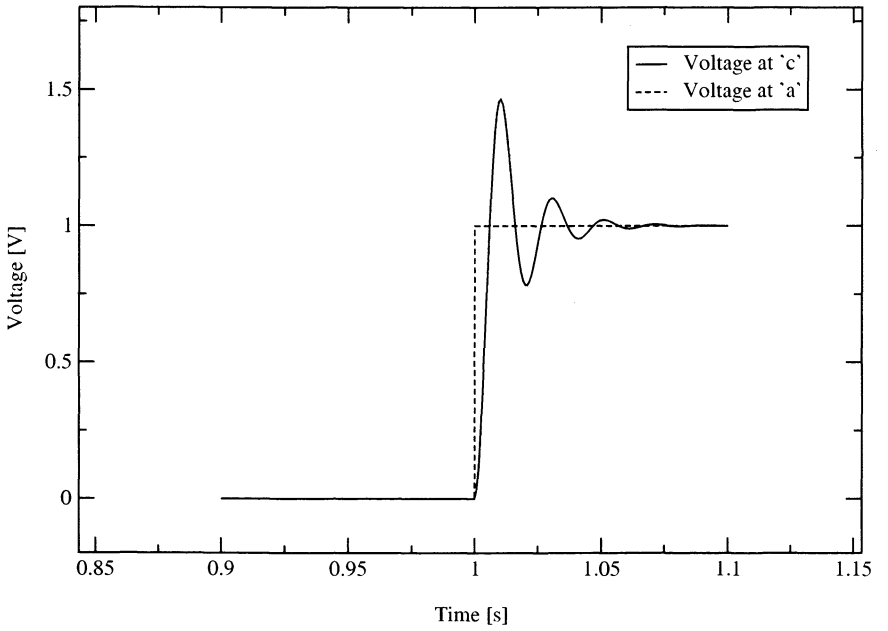

Figure 2.5. Voltage response of model `RLC`.

Example 2.3 covers several new topics not seen in the previous example. The first difference is the appearance of physical types (*i.e.*, `Voltage`, `Current`, `Resistance`, `Capacitance` and `Inductance`). As we shall see later, these physical types provide important information about the quantities they are associated with (*e.g.*, units, limits and default values). These physical types are defined in a `package` called `Modelica.SIunits`. This is why the physical types all contain `Modelica.SIunits` in their name.

In the `equation` section, we see the first use of the `if` keyword. The use of `if` in this context is called an if-expression.[1] For this example, when `time` is less than 1, $V_a = 0$ and once time is greater than 1, $V_a = 1$. The variable `time` is used to represent simulation time.

2.4 DOCUMENTING MODELS

In this section, we create a model of a hydraulic system and show how to include documentation in models. Such documentation not only helps the model developer to remember how the model functions, it also helps the

[1] An `if` expression is similar to the ternary operator in C.

developer and any new users of the model to understand exactly what each of the components and quantities represent.

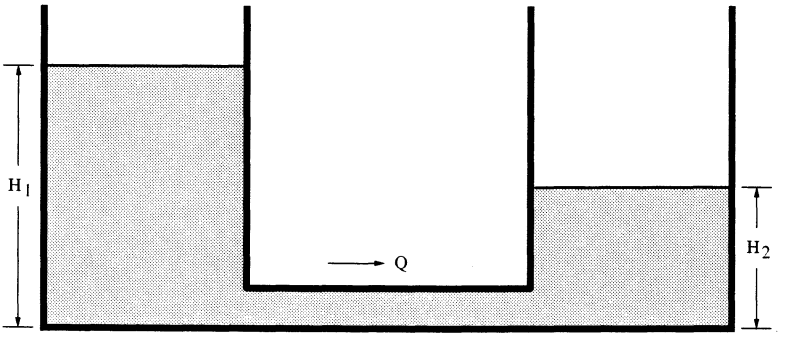

Figure 2.6. Two hydraulic tanks filled with liquid.

Figure 2.6 shows the schematic for a hydraulic system composed of two tanks connected by a cylindrical pipe. For this example, we assume that the fluid in the tanks is incompressible and each tank has a constant cross-sectional area.

2.4.1 Constitutive equations

The first step in computing the flow, Q, through the pipe is to know the pressure at the bottom of each tank. To determine the pressure we use the following equation:

$$P = \rho g H$$

where P is the pressure, H is the height of the fluid in the tank, g is the acceleration due to gravity and ρ is the density of the liquid. Using this relationship, the pressures in the two tanks are determined by the following equations:

$$P_1 = \rho g H_1 \tag{2.18}$$

$$P_2 = \rho g H_2 \tag{2.19}$$

Now that we know the pressures, we need to compute the volumetric flow rate, Q, through the pipe. For laminar flow through a cylindrical pipe, we can use the Hagen-Poiseuille relationship (see, *e.g.,* Ogata, 1978):

$$Q = (P_1 - P_2)\frac{\pi D^4}{128 \mu L} \tag{2.20}$$

where P_1 is the pressure in the tank on the left, P_2 is the pressure in the tank on the right, D is the diameter of the pipe connecting the two tanks, μ is the

dynamic viscosity and L is the length of the pipe. Note that the sign convention for Q is that a positive value indicates flow from the tank on the left to the tank on the right.

Lastly, we need an equation which relates the volumetric flow rate through the pipes with the change in fluid height in each tank. Since the fluid flowing between the tanks is incompressible, the volume of fluid flowing through the pipe must be the same as the volume of fluid exchanged with the tanks. This behavior can be expressed by the following equations:

$$A_1 \frac{dH_1}{dt} = -Q \tag{2.21}$$

$$A_2 \frac{dH_2}{dt} = Q \tag{2.22}$$

where A_1 is the cross-sectional area of the tank on the left and A_2 is the cross-sectional area of the tank on the right.

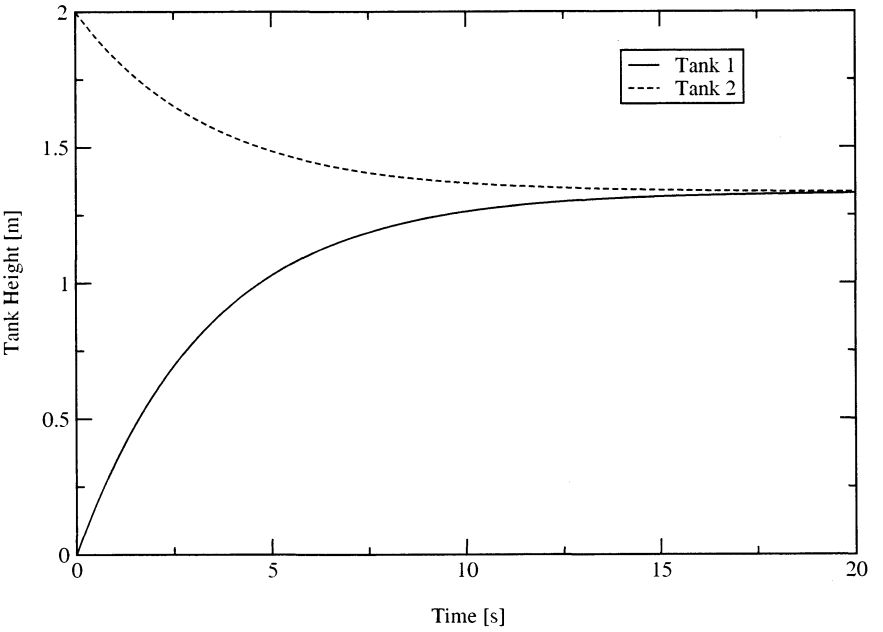

Figure 2.7. Solution with initial conditions `H1=0` and `H2=2`

2.4.2 Modelica model

Example 2.4 shows the Modelica model that corresponds to the hydraulic system shown in Figure 2.6. Figure 2.7 shows the simulation results for that

```
model TwoTanks "Hydraulic system involving two tanks"
  import Modelica.SIunits;

  // Constants
  constant Real pi=Modelica.Constants.pi;
  constant Real g=Modelica.Constants.g_n;

  // Parameters
  parameter SIunits.Length L=0.1 "Pipe length";
  parameter SIunits.Length D=0.2 "Pipe diameter";
  parameter SIunits.Density rho=0.2 "Fluid density";
  parameter SIunits.DynamicViscosity mu=2e-3;
  parameter SIunits.Area A1=1.0 "Area of left tank";
  parameter SIunits.Area A2=2.0 "Area of right tank";
  parameter SIunits.KinematicViscosity c=(pi*D^4)/(128*mu*L);

  // Variables
  SIunits.Pressure P1, P2;
  SIunits.Length H1, H2;
  SIunits.VolumeFlowRate Q;
equation
  // Pressure equations
  P1 = rho*H1*g;
  P2 = rho*H2*g;

  // Flow rate
  Q = c*(H1-H2);

  // Conservation of mass
  A1*der(H1) = -Q;
  A2*der(H2) = Q;
end TwoTanks;
```

Example 2.4. Hydraulic system of two tanks.

system assuming the first tank starts at a height of 0 meters and the second tank starts with a height of 2 meters.

Instead of typing out the `Modelica.SIunits` qualifier before each physical type as we did in Example 2.3, we instead chose to create an abbreviation, `SI`, using the **package** keyword. Using this approach we are required to type far fewer characters for each physical type name. Think of this as a way to create aliases when working with long **package** names.

In Example 2.4, we can see the use of descriptive text (contained between matching double quotation marks) associated with the model and parameters. In addition, this example includes comments which provide additional docu-

mentation. Whenever the characters "//" appear in a Modelica model, the remainder of the line is considered a comment.

The remaining difference between this example and the previous examples in this chapter is the use of the physical constants. The MSL contains a collection of physical constants which commonly appear in engineering equations. In this example, we have made use of `Modelica.Constants.g_n` (representing acceleration due to Earth's gravity) and `Modelica.Constants.pi`. Use of physical constants in the MSL serves three purposes. First, the model developer does not have to remember the value of the constants. Second, it makes sure that the constant is specified to the complete numerical precision for the computer it is used on. Third, it avoids the error prone process of typing such numbers in manually.

2.5 LANGUAGE FUNDAMENTALS

The purpose of this section is to provide a more comprehensive discussion of the language fundamentals demonstrated by the examples in this chapter. This section is included for completeness but it is not required. Readers may feel free to skip this section entirely if they are comfortable with the material presented so far.

2.5.1 Models

Models have behavior described by algebraic and/or differential equations. Recall our use of the `model` keyword in Examples 2.1, 2.3 and 2.4. The keyword `model` in Modelica is used to indicate the start of a model *definition.* As we have seen in our examples, the **`end`** keyword (followed again by the model name) is used to indicate the end of the model.[2]

As seen in Example 2.4, the definition of a model may include descriptive text to provide additional information about the model. The textual description of a model must be contained within matching double quotation marks and must appear directly after the model name. The textual description for constants, parameters, variables or any component declarations must appear just prior to the "`;`" which is used to indicate the end of the declaration. While comments are free form text with no particular association to any part of the Modelica source, textual descriptions are directly associated with specific declarations. This link to specific declarations allows textual descriptions to be used in graphical user interfaces or automatically generated documentation.

In this chapter, we have seen models which contain constants, parameters, variables and equations. While there are other things a model may contain,

[2]The reason the model name appears twice is to help identify possibly mismatched **`end`** keywords in nested structures. As we will see later, this same technique is also used to align the beginning and end of control structures such as `if` and `while`.

these are the basic elements and should be sufficient for developing simple models.

2.5.2 Variables, parameters and constants

2.5.2.1 Declarations

As you may have noticed from the examples in this chapter, the *declaration* of every quantity (*i.e.*, variables, parameters and constants) requires a type (*e.g.*, `Real` or `Length`) followed by a name. Furthermore, each declaration may include an equation for that quantity (*e.g.*, "`=12`") and/or descriptive text associated with the quantity. The end of the declaration is indicated by a semi-colon.

2.5.2.2 Types

In our first example, we used the built-in type `Real` to represent floating point values. Modelica provides three additional built-in types: `Integer`, `Boolean` and `String`.

In addition to the built-in types, it is possible to create *derived types*. Derived types are specializations of the built-in types. For example, the derived type `Length` shown in Example 2.4 is defined in the MSL as follows:

```
type Length=Real(quantity="Length", unit="m");
```

Derived types provide more specific information about the quantity. This information is useful for documentation purposes (*e.g.*, what physical units are associated with a given parameter), unit conversion and in some cases even some semantic analysis (*e.g.*, unit checking in expressions). The most commonly used derived types in the MSL are compiled in Appendix D.

2.5.2.3 Variability

Any declared quantity in Modelica has a specific *variability*. By default, all declared quantities are assumed to change as a function of simulation time. However, there are variability qualifiers which can be used to indicate different levels of variability. In this chapter, we have introduced two such qualifiers, `constant` and `parameter`. Both of these qualifiers prevent the value of a quantity from changing during a simulation. Despite the fact that both are restricted in this way, there are two important differences between constants and parameters. First, once defined within a model a constant is not intended to be changed. For this reason, the graphical user interface for some tools may not allow adjustments to constants (or even display them). In practice, this means the only way a constant can be changed is to modify the source code of a model. The other difference between constants and parameters is that the declaration of a parameter **may** include an expression for the value of that parameter but

the declaration of a constant **must** include an expression for the value of that constant (for example `g` in Examples 2.1 and 2.4).

There are other variability qualifiers but we will discuss those in the context of subsequent examples.

2.5.3 Expressions

For the most part, expressions in Modelica look similar to expressions in other computer languages. In this section, we will cover the basic types of expressions used in our examples so far.

2.5.3.1 Basic expressions

In Example 2.1, we see our first use of an expression. We compute the derivative of `omega` as `-(g/L)*theta`. In this one expression we use the multiplication, division and subtraction operators. Modelica uses the `+`, `-`, `*` and `/` operators to represent addition, subtraction, multiplication and division, respectively. Furthermore, the `^` operator is used to represent raising an expression to a power. For example, the expression `(x+y)^z` represents the sum of `x` and `y` raised to the power of `z`. Use of the `^` operator can be seen in Example 2.4 in determining the `c` parameter. The precedence of the operators (`^`, `*`, `/`, `+`, `-`) and the implications of parentheses are the same as in algebra.

As we shall see in Chapter 6, the `+`, `-`, `*` and `/` operators can also be applied to arrays (*e.g.,* vectors and matrices). The `+` and `-` operators can be used to add or subtract two arrays of the same shape. The `*` and `/` operators can be used to multiply or divide an array by a scalar. Furthermore, the `*` operator represents the inner product operator when used between two arrays of the appropriate shape.

2.5.3.2 Conditional expressions

Conditional expressions are expressions which evaluate to either `true` or `false`. Such expressions use the relational operators "`==`", "`<>`", "`<`", "`<=`", "`>`" and "`>=`" to represent equality, inequality, less than, less than or equal to, greater than and greater than or equal to relationships, respectively (Just as with basic expressions, conditional expressions in Modelica are similar to conditional expressions in other computer languages). Note that the "`==`" and "`<>`" operators cannot be applied to `Real` variables.

In Example 2.3, we saw how the "`>=`" operator was used to determine when the simulation time had exceeded 1 second. Conditional expressions can be combined using the `or` and `and` logical operators. In addition, the `not` operator can be used to negate the value of a conditional expression. Finally, parentheses can be used to explicitly control the precedence of the operators.

2.5.3.3 Function calls

The `Modelica.Math` package in the MSL includes many useful functions (see Appendix F for a complete list). For instance, we saw how the `sin` function was invoked in Example 2.2. In the case where functions require more than one argument, the arguments must be separated by commas. Chapter 5 discusses, in detail, how to write and invoke functions.

2.5.3.4 Using if-expressions

In Example 2.3, we saw how a step voltage could be defined using an if-expression. The syntax for an if-expression is:

```
if cond_expr then true_expr else false_expr
```

where `cond_expr` is a conditional expression evaluating to either `true` or `false`. In the case where the conditional expression evaluates to `true`, the if-expression evaluates to `true_expr`. If the conditional expression evaluates to `false`, the if-expression evaluates to `false_expr`. Among other things, this is a convenient way of representing simple functions and discontinuities. Such if-expressions can be used anywhere a normal expression can be used and may even be nested one inside another. For example, a step could be expressed using if-expressions as follows:

```
v = if time<=1 then 0 else if time<=2 then 1 else 2;
```

2.5.4 Equations

Each of the models in this chapter contains an `equation`. It is important to recognize that the "=" operator in Modelica **does not represent assignment**. Instead, the "=" operator defines a relationship between several quantities and it does not necessarily have to be of the form:

```
variable = expression;
```

Instead, an equation expresses equality between two expressions and has the more general form:

```
expression1 = expression2;
```

This is important because it means the model developer is not required to manipulate equations to get them into assignment form (a task which can be surprisingly difficult once complex systems of differential-algebraic equations are involved). In fact, the equations specified in the `equation` can be any combination of algebraic and differential equations. For example, consider the following set of equations:

```
x = time;
x = 4*y;
```

where `time` is the global simulation time. If Modelica were a procedural language like C or FORTRAN, the first statement would assign a value to `x` and the second statement would overwrite the value of `x` with a new value. This is because in those languages the = operator is used to represent assignment. In Modelica, the = represents an equality relationship and the := represents the operation of assignment.[3] Assignments are not allowed in an `equation`. Instead, they must be placed inside an `algorithm` section (discussed in Chapter 5).

It is possible that the equations:

```
x = time;
x = 4.0*y;
```

might be rearranged by a simulator into the following set of **assignments**:

```
x := time;
y := x/4.0;
```

Note the use of the := operator. The rearrangement of terms in this way is called *symbolic manipulation.* When you provide equations in Modelica, a simulator is free to perform such manipulations. Remember, Modelica is a descriptive language which means that the model developer is only responsible for providing the equations, not solving them.

Note that equations can appear outside the `equation`. Specifically, an equation can also appear as part of a declaration. The following code fragment demonstrates this:

```
model CoolingGlass
  parameter Modelica.SIunits.CoefficientOfHeatTransfer h;
  parameter Modelica.SIunits.SpecificHeatCapacity cp;
  parameter Modelica.SIunits.Mass m;
  Modelica.SIunits.Temperature T;
  Modelica.SIunits.Temperature T_ambient=300+20*time;
equation
  m*cp*der(T) = -h*(T-T_ambient);
end CoolingGlass;
```

The `CoolingGlass` model contains two variables and two equations although only one of the equations appears in the `equation`. The other equation appears in the declaration of `T_ambient`. The ability to include equations in this way can be convenient but also confusing since such equations are not easily spotted when glancing at the model.

[3]The left hand side of an assignment statement must be a variable.

2.5.5 Operators

In this chapter, we have used the `der` operator to represent the derivative of a variable. In this section we will discuss the `der` operator and the `delay` operator.

2.5.5.1 The derivative operator

In the expression `der(x)`, the `der` operator is used to represent the time derivative of the variable `x`. One important restriction is that the `der` operator can only be used on variables, not on expressions. Furthermore, the `der` operator cannot be used recursively. In other words, the following is not a legal way to represent the second derivative:

```
alpha = der(der(theta)); // Illegal
```

In order to represent the second derivative of a variable, the first derivative must be assigned to a variable. For example:

```
omega = der(theta); // First derivative
alpha = der(omega); // Second derivative
```

The simple pendulum model, presented in Example 2.1, shows how this is done within a model.

2.5.5.2 The delay operator

The `delay` operator can be invoked with either two or three arguments. The first argument of the `delay` operator is always an expression. The value of the `delay` operator is the value of the expression delayed by some amount of time. The amount of time delay is the second argument of the `delay` operator.

If only two arguments are present, then the second argument must be a *parameter expression* which means it cannot be a function of time. The following is an example of using the `delay` operator with a fixed delay:

```
model FixedDelay
  parameter Modelica.SIunits.Time dt=2;
  Real x, y, z;
equation
  der(x) = ...;
  y = ...;
  z = delay(x+y,dt);
end FixedDelay;
```

The response of `z` would be equal to $x(time - dt) + y(time - dt)$.

It is possible to use the `delay` operator to express a variable delay as well. If a third argument is present it represents the maximum time delay allowed and the second argument can then be a time-varying expression. If present, the third argument must be a parameter expression and the value of the second

argument must always be greater than zero and less than the value of the third argument.

2.5.6 Attributes

Each declared quantity (*e.g.,* a `parameter` or `constant`) has a set of *attributes*. These attributes can be associated either with the type of the quantity or the specific instance of the quantity. For example:

```
type Length=Real(start=1.0,
                 quantity="Length",
                 unit="m");
Length x(start=2.0);
```

In the first statement, the `start`, `quantity` and `unit` attributes are associated with the type `Length`. Any declaration of type `Length` automatically inherits the attributes of `Length`. In the second case, the declaration of `x` overrides the value of the `start` attribute inherited from type `Length`. Any such adjustment to the attributes in a declaration is called a *modification.* More details on modifications can be found in Chapter 3. We conclude this section with a brief list of common attributes.

2.5.6.1 The "start" attribute

When declaring a variable, the `start` value is used to provide a reasonable initial guess (see the explanation of the `fixed` attribute for an important exception). This can be useful in problems which involve non-linear systems of equations. In such systems, multiple solutions are possible and the `start` attribute can be used to influence which solution is found.

Each of the built-in types has a `start` attribute. The default value for the `start` attribute is zero. Note that the value of the `start` attribute for a type is ignored when declaring a constant of that type because each constant declaration **must** provide a value. For example,

```
constant Length L=2;
```

2.5.6.2 The "fixed" attribute

The `fixed` attribute can be used, in conjunction with the `start` attribute, to specify the initial value for a variable at the start of a transient simulation. When the `fixed` attribute is `false`, which is the default value, the `start` attribute merely indicates an initial guess for variables (*e.g.,* when solving non-linear equations). However, when the `fixed` attribute is `true` the `start` attribute indicates the value the variable **must** have at the start of the simulation. A more complete discussion of how the `fixed` attribute is used can be found in Chapter 13.

2.5.6.3 The "min" and "max" attributes

The `min` and `max` attributes define the minimum and maximum values for a given numeric type. These attributes are used to identify when a quantity has an unreasonable value. For example, thermodynamic temperatures are measured relative to absolute zero, so negative values are non-physical. To indicate this in a model, the `min` attribute would be set to zero. Both `Real` and `Integer` types have the `min` and `max` attributes.

These attributes are mainly used in model development to prevent the user of a model from entering non-physical values for parameters and for letting the simulator know when it has found an unreasonable solution.[4]

2.5.6.4 The "quantity" attribute

The `quantity` attribute is a character string which describes the nature of a type. In most cases, the string contains the type name. For example:

```
type Strain = Real(quantity="Strain");
```

In other cases involving derived types, the `quantity` attribute of the base type (`Energy` in this case) is inherited, as in:

```
type Energy = Real(quantity="Energy");
type PotentialEnergy=Energy; // quantity="Energy"
type KineticEnergy=Energy; // quantity="Energy"
```

All built-in types have the `quantity` attribute.

2.5.6.5 The "unit" and "displayUnit" attribute

The `unit` attribute serves mainly as documentation for a type. Assigning a string to the `unit` attribute sets the units for that type. If units are provided for a particular type, it is important that all values given for quantities of that type be in those units because all equations in the model are written with the assumption that values are provided in the specified units. Both `Real` and `Integer` types have the `unit` attribute.

The MSL provides a large collection of types with the proper units defined (see Appendix D for a list of the most commonly used types). The Modelica specification, which can be found on the companion CD-ROM, contains details about the format for strings that represent physical units (*e.g.*, "`m/s`" for meters per second or "V" for volts).[5]

When entering data or displaying results, the values for a given type are normally provided in the physical units assigned with the `unit` attribute. However, it is possible to use different units when entering data or displaying

[4]The details of what happens if these limits are violated vary from program to program.
[5]The Modelica specification defines a syntax for representing units.

results by setting the `displayUnit` attribute to the desired physical unit. Only the `Real` type has the `displayUnit` attribute.

If a tool has unit conversion capability, it can do such unit conversions behind the scenes. For example, angular position is defined in the MSL as follows:

```
type Angle = Real(quantity="Angle",
                  unit="rad",
                  displayUnit="deg");
```

This is because few people think of angles in terms of radians. For entering data or displaying results, most people would prefer to see degrees instead of radians (or revolutions per minute instead of radians per second). The `displayUnit` attribute indicates the preferred unit for pre-processing and post-processing. The `displayUnit` only has an effect if the tool you are using supports such unit conversion. In any case, the `unit` attribute will always indicate what units are used inside the models.

2.5.7 Physical types

Section 2.5.2.2 described how new, more specialized types can be created from the built-in types. Throughout this chapter we have used a variety of physical types (*e.g.*, `Voltage`, `Velocity` and `AngularAcceleration`) derived in this way.

The main advantage of these specialized types is that they are better at describing the nature of a given quantity than the generic type `Real`. In complex models it may be difficult to figure out the physical type of some quantities and what the intended units should be. As we have seen in these examples, a parameter named `L` might indicate a quantity of length or inductance. By using predefined physical types, the intended usage of a parameter or variable is much clearer. Furthermore, a clever analysis tool may check dimensional consistency of the units involved in expressions. The use of physical types in this way can improve the readability of the code as well.

2.6 PROBLEMS

PROBLEM 2.1 *Rewrite the model shown in Example 2.1 to use physical types and provide descriptive text for the model, variables and parameters.*

PROBLEM 2.2 *Write physical type definitions for frequency, absolute temperature and mass fraction. What are the units for these? Do they have upper and/or lower bounds?*

PROBLEM 2.3 *There are many conservation principles we may employ. Is the fact that all torques around a point must sum to zero a conservation law? If so, explain what is being conserved.*

PROBLEM 2.4 *Run the models shown in Examples 2.1 and 2.2 for a variety of initial positions and velocities. In each case, plot* `omega` *as a function of* `theta` *and compare the linear and non-linear trajectories. What are the interesting characteristics of these trajectories?*

PROBLEM 2.5 *Write a model for a "predator-prey" system using the Lotka-Volterra system of equations:*

$$\begin{aligned} \dot{x} &= \alpha xy - \beta x \\ \dot{y} &= \gamma y - \delta xy \end{aligned}$$

where x represents the predator population and y represents the prey population. Suggested values for model coefficients are $\alpha = 0.1$, $\beta = 2$, $\gamma = 4$ and $\delta = 0.4$. Build a Modelica *model for this system and experiment with different initial population levels. Visualize the solution by plotting prey population versus predator population.*

PROBLEM 2.6 *Create a model, similar to the one shown in Example 2.4, containing three (or more) tanks connected by pipes.*

PROBLEM 2.7 *Write a* Modelica *expression for* `V_a` *in Example 2.3 such that:*

$$V_a = \begin{cases} 0 & : \quad 0 \leq t < 1 \\ 1 & : \quad 1 \leq t < 2 \\ 0 & : \quad 2 \leq t < \infty \end{cases} \tag{2.23}$$

PROBLEM 2.8 *The longitudinal dynamics of an aircraft can be approximated by the following equations (found in Brogan, 1991):*

$$\dot{\theta} = q \tag{2.24}$$

$$\dot{q} = -\omega^2(\alpha - \delta S) \tag{2.25}$$

$$\dot{\alpha} = -\frac{\alpha}{\tau} + q \tag{2.26}$$

where θ, q and α are variables representing pitch, pitch velocity and angle of attack while τ, ω and S are flight dynamics parameters and δ is an input representing the elevator angle. Using these equations, create models which predict the behavior of the aircraft during different maneuvers (i.e., time-varying elevator positions). Sample values for τ, ω and S are 0.25, 2.5 *and* 1.6, *respectively.*

Chapter 3

BUILDING AND CONNECTING COMPONENTS

3.1 CONCEPTS

While equations are an essential part of model development, it quickly becomes tedious to write out all the equations for the components in a system. In this chapter, we show how to reuse constitutive equations like Ohm's law and automatically generate conservation equations for quantities like energy and mass. In doing so, it is possible to quickly build up large models of interacting components. Once again, examples will demonstrate various language features and the section at the end of the chapter will discuss these features in detail.

3.2 CONNECTORS

The focus of this chapter will be creating reusable component models and then connecting *instances* of these models together to form complex networks. In order to discuss the connection of components, we must first discuss a new type of definition called a `connector`.

The best physical analogy of a `connector` is an electrical plug. The advantage of electrical plugs is that when you plug, for example, a television into an electrical outlet, you can be sure that each wire in the plug will connect to the appropriate wire in the wall. A `connector` in Modelica serves the same purpose by matching up the appropriate variables from connectors on different components.

A `connector` definition contains variables which describe the interaction between components. The following is a sample `connector` definition:

```
connector HydraulicPort
  Modelica.SIunits.Pressure p;
  flow Modelica.SIunits.VolumeFlowRate q;
end HydraulicPort;
```

3.3 CREATING CONNECTORS AND COMPONENTS

Let us start by considering a familiar example. Figure 3.1 shows a slight variation on the electrical circuit shown in Figure 2.4. Because a new resistor was added in Figure 3.1, the behavioral equations are slightly different than before.

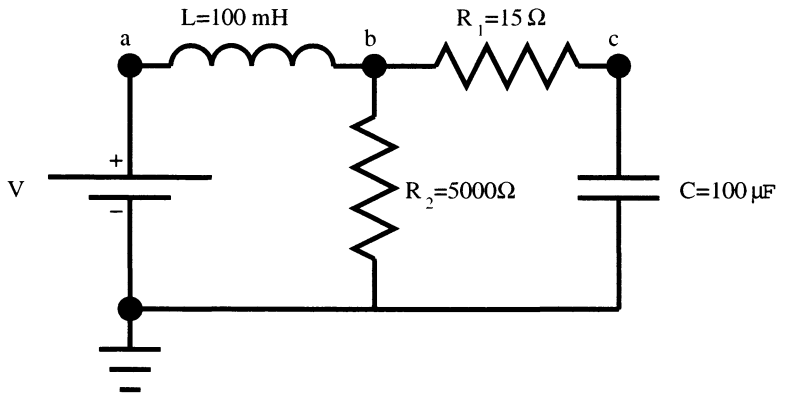

Figure 3.1. Another RLC circuit.

3.3.1 Equation based approach

If we were to write a Modelica model for the system in Figure 3.1 the same way we did for Figure 2.4, the resulting model might look something like the one shown in Example 3.1.

One problem with the Modelica model in Example 3.1 is that it is getting a little difficult to understand what it is a model of, just by looking at the source code. In other words, you would have to carefully study the description in Example 3.1 to realize it represents the circuit shown in Figure 3.1.

Another problem is that we are starting to write equations multiple times. Ohm's law, for example, appears twice in Example 3.1. While writing Ohm's law twice may not seem like much of a burden, writing it 25 times for a complex circuit is tedious and the process would be error prone. Furthermore, if a mistake is made it must be corrected in 25 places. Likewise, making sure the conservation equations are written correctly can also be a tedious and error prone task. For these reasons, the preferred way to model the system in Figure 3.1 is to use a component based approach.

```
model RLC2
  import Modelica.SIunits;

  parameter SIunits.Resistance R1=15;
  parameter SIunits.Resistance R2=5000;
  parameter SIunits.Capacitance C=100e-6;
  parameter SIunits.Inductance L=100e-3;

  SIunits.Voltage V_a, V_b, V_c;
  SIunits.Current i_V, i_R1, i_R2, i_C, i_L;
equation
  V_a = if time>=1 then 1.0 else 0.0;
  L*der(i_L) = V_a - V_b;
  R1*i_R1 = V_b - V_c;
  i_C = C*der(V_c);
  R2*i_R2 = V_b;
  i_V - i_L = 0;
  i_L - i_R1 - i_R2 = 0;
  i_R1 - i_C = 0;
end RLC2;
```

Example 3.1. Another RLC circuit.

3.3.2 Component based approach

With a component based approach, we create a single model for each of the components we require (*i.e.*, `VoltageSource`, `Resistor`, `Capacitor` and `Inductor`). Once these models exist, we can connect instances of them together in a variety combinations.

The key to modeling using a component based approach is to think about the *free body diagram* for a component. A free body diagram (borrowing a term from mechanical dynamics) is a diagram which describes all state information associated with a component and identifies all possible external influences. For example, consider the resistor shown in Figure 3.2. We can see that the resistor has two voltages associated with it, one at each connection point. In addition, there are currents flowing through each of these connection points.

Figure 3.2. A "free body diagram" of a `Resistor`.

3.3.2.1 ElectricalPin

Before we can begin writing models for the electrical components, we must first identify the appropriate `connector` for these components. Let us define the `connector` for our electrical system as:

```
connector ElectricalPin
  Modelica.SIunits.Voltage v;
  flow Modelica.SIunits.Current i;
end ElectricalPin;
```

This connector identifies the two quantities associated with a single connection point[1] in the free body diagram shown in Figure 3.2. For the `ElectricalPin` connector, `v` represents the voltage at that connection point and `i` represents the current flowing into the resistor.

An important thing to note about this connector is the `flow` qualifier in front of the current, `i`. The `flow` qualifier identifies quantities that must sum to zero, at a connection point. Identifying the appropriate `flow` variables allows a simulator to implicitly generate the conservation equations like those for Kirchhoff's current law shown in Equations (2.15)-(2.17). This is the first step in making system models easier to build and less error prone. Typically, the `flow` qualifier is applied to time derivatives of conserved quantities (*i.e.,* the current, `i`, is the time derivative of charge, which is a conserved quantity).

Now that we have defined the connector type, we can move on to the component models.

3.3.2.2 Resistor

Ohm's law describes the behavior of a resistor, *i.e.,*

$$v = iR \tag{3.1}$$

Example 3.2 shows how a Modelica model could be written for the resistor shown in Figure 3.2. The "`.`" in quantities like `p.v` is a way of accessing the internal elements of a component. Since `p` is an instance of an `ElectricalPin` it contains a variable for voltage called `v`. In this way, the quantity `p.v` represents the voltage associated with pin `p`.

It is important when developing component models to use a consistent sign convention for the `flow` quantities. The normal sign convention for Modelica components is defined such that positive flow is **into** the component. Therefore inside the `Resistor` model, the value of `p.i` refers to the current flowing **into** the resistor from pin `p` and the value of `n.i` refers to the current

[1] The black circles in Figure 3.2 are the connection points for the resistor.

```
model Resistor "An electrical resistor"
  import Modelica.SIunits;

  parameter SIunits.Resistance R=300 "Resistance";
  ElectricalPin p, n; // Naming the connection points
equation
  R*p.i = p.v - n.v;
  p.i + n.i = 0;
end Resistor;
```

Example 3.2. A model for an electrical resistor.

flowing **into** the resistor from pin `n`. From Example 3.2 we can see that a positive value for `p.i` results when `p.v` is greater than `n.v`. This is consistent with the normal sign convention. Likewise, a positive value for `n.i` results when `n.v` is greater than `p.v`.

In Example 3.2, the current `p.i` is used to represent the current in Ohm's law. This choice between using `p.i` and `n.i` is arbitrary. However, if `n.i` had been used in the equation, it would need to be written as:

```
R*n.i = n.v - p.v;
```

in order to satisfy the sign convention for flow variables (*i.e.*, a positive value represents flow into the component). Finally, note that a default value of 300Ω is given for the resistance of the resistor.

3.3.2.3 Capacitor

The constitutive equation for the behavior of an ideal capacitor is:

$$C\frac{dv}{dt} = i \tag{3.2}$$

Once we have written the model for the `Resistor`, it is easy to imagine how a model for a capacitor would be written. Example 3.3 shows just such a model. Note that we must continue to use the same sign convention. Looking at the model in Example 3.3, one might wonder if it would be possible to write the equation for `p.i` as follows:

```
p.i = C*der(p.v-n.v);
```

As we pointed out in Section 2.5.5.1, this is not legal because the `der` operator cannot be applied to an expression. In the case of the `Capacitor` model, we handle this by introducing the `v` variable which represents the voltage difference across the capacitor. The default capacitance for this capacitor model is $10^{-6}F$.

```
model Capacitor "An electrical capacitor"
  import Modelica.SIunits;

  parameter SIunits.Capacitance C=1e-6 "Capacitance";
  ElectricalPin p, n;
  SIunits.Voltage v;
equation
  v = p.v-n.v;
  p.i = C*der(v);
  p.i + n.i = 0;
end Capacitor;
```

Example 3.3. A model for an electrical capacitor.

3.3.2.4 Inductor

The constitutive equation for an inductor is:

$$v = L\frac{di}{dt} \tag{3.3}$$

```
model Inductor "An electrical inductor"
  import Modelica.SIunits;

  parameter SIunits.Inductance L=1e-3 "Inductance";
  ElectricalPin p, n;
equation
  L*der(p.i) = p.v-n.v;
  p.i + n.i = 0;
end Inductor;
```

Example 3.4. A model for an electrical inductor.

Example 3.4 shows a Modelica model for an inductor. Again, we take care to use the correct sign convention. Can you see the similarity to the `Resistor` and `Capacitor` model? We will take advantage of the similarities in later examples. The default inductance value is $10^{-3}H$.

3.3.2.5 Step voltage source

We now need a model for the voltage source shown in Figure 3.1. Example 3.5 shows how such a model could be written. The `VoltageSource` model consists of two algebraic equations. The first dictates what the voltage drop is across the `VoltageSource` from connection point `p` to `n` The other equation dictates that the current coming in one side of the component must exactly

```
model VoltageSource "A voltage source"
  import Modelica.SIunits;

  parameter SIunits.Voltage v1=0, v2=1;
  parameter SIunits.Time jump_time=1.0;

  ElectricalPin p, n;
equation
  p.v-n.v = if time>=jump_time then v2 else v1;
  p.i + n.i = 0;
end VoltageSource;
```

Example 3.5. A model for a step voltage.

balance the current going out the other side. Unlike our previous examples, the `VoltageSource` model does not contain an explicit equation for the current flowing through the device as a function of the voltage drop across it. Instead, an explicit equation is provided for the voltage across the device and the simulator will be responsible for determining what amount of current is necessary to satisfy the voltage equations.

Note that this is an idealized voltage source model and the discontinuous voltage drop might cause trouble in some circuits. For example, if this voltage source were connected in parallel to a capacitor the instantaneous jump in voltage (due to the step) should trigger an infinite current spike through the capacitor. For more realistic systems you might need to create a less idealized voltage source.

By default, the voltage source has an initial voltage drop of zero Volts and jumps to a voltage drop of 1 Volt after 1 second.

3.3.2.6 Ground

```
model Ground "Ground"
  ElectricalPin ground;
equation
  ground.v = 0;
end Ground;
```

Example 3.6. A model for electrical ground.

The last component model we require to simulate the circuit shown in Figure 3.1 is `Ground`. The `Ground` model is different from all the other electrical models presented so far because it only has a single `ElectricalPin` connector. In addition, there are no parameters associated with a `Ground` model.

The only equation required for the Ground model is to set the voltage at the connection point to zero. The Ground component model can be seen in Example 3.6. The Ground and VoltageSource are quite similar. The difference is that the Ground model constrains the absolute voltage at a connection point whereas the VoltageSource model constrains the relative voltage between two connection points. Both models will require the solver to implicitly solve for the current through the device.

3.3.2.7 Circuit model

```
model RLC3 "Yet another RLC circuit"
  Resistor R1(R=15);
  Resistor R2(R=5000);
  Capacitor C(C=100e-6);
  Inductor L(L=100e-3);
  VoltageSource vs;
  Ground g;
equation
  connect(vs.n,g.ground);
  connect(vs.p,L.p);
  connect(L.n,R1.p);
  connect(L.n,R2.p);
  connect(R1.n,C.p);
  connect(C.n,g.ground);
  connect(R2.n,g.ground);
end RLC3;
```

Example 3.7. Another model for our RLC circuit in Figure 3.1.

Now that we have written our component models, we can bring them all together to build a circuit like the one shown in Figure 3.1. Example 3.7 shows what the Modelica code for our circuit model looks like. Note in Example 3.7 that resistors R1 and R2 have different values for their Resistance parameter. We can see this because the declarations of these resistors contain a *modification* (*i.e.*, the assignments contained within parentheses) to change the value of the R parameter of each component. Because the declaration of the voltage source component, vs, does not specify values for its parameters (*i.e.*, there are no modifications), the default values defined inside the component models will be used.

In order to fully understand Example 3.7, some explanation must be provided for the connect command. Because the details of the connect command are covered in Section 3.6.1, we only present a cursory explanation here.

The connect command generates equations based on the contents of the connectors being connected. Equations are generated by considering the com-

ponents of each connector with matching names. Normally, the `connect` command generates an equation which sets the matching components equal to each other. However, if a component has the `flow` qualifier (*i.e.*, it is a through variable), then an equation is generated which sums the matching components to zero. So for our `ElectricalPin connector` definition, the voltage, `v`, is set equal to all other voltages at the connection point and the sum of all current contributions at the connection point is set equal to zero.

Using these rules, we find that the `connect` statements in Example 3.7 would generate the following equations:

Equality equations generated for across variables.

$$\left\{\begin{array}{rcl} vs.n.v &=& g.ground.v \\ vs.p.v &=& L.p.v \\ L.n.v &=& R1.p.v \\ L.n.v &=& R2.p.v \\ R1.n.v &=& C.p.v \\ C.n.v &=& g.ground.v \\ R2.n.v &=& g.ground.v \end{array}\right. \tag{3.4}$$

Conservation equations generated for the flow variables.

$$\left\{\begin{array}{rcl} 0 &=& vs.p.i + L.p.i \\ 0 &=& L.n.i + R1.p.i + R2.p.i \\ 0 &=& R1.n.i + C.p.i \\ 0 &=& vs.n.i + R2.n.i + C.n.i + g.ground.i \end{array}\right. \tag{3.5}$$

The order of the arguments to the `connect` command is not important. As a result, the statement:

```
connect(a,b);
```

is equivalent to:

```
connect(b,a);
```

3.3.3 Standard electrical components

We went to a great deal of trouble to create definitions for the connectors and electrical components (*e.g.*, `Resistor`, `Capacitor`, *etc.*) in this section. Hopefully this was instructive in showing how to build up such components. However, it turns out that **all of this has already been done** for us. This is because the MSL has a library of electrical components like the components we created in this section (and many more).

To demonstrate the usefulness of the MSL, we will reimplement Example 3.7 using components from the MSL. As Example 3.8 shows, everything we need is available within the MSL. The Modelica language features the ability

to include graphical information about components (for more details about graphical information, see Section 9.2). Because the MSL provides such graphical information about each component, schematics like the one shown in Figure 3.3 can be easily created directly from the Modelica source code.[2]

```
model RLC4 "An RLC circuit using standard components"
  import Modelica.Electrical.Analog;

  Analog.Basic.Resistor R1(R=15);
  Analog.Basic.Resistor R2(R=5000);
  Analog.Basic.Capacitor C(C=100e-6);
  Analog.Basic.Inductor L(L=100e-3);
  Analog.Sources.StepVoltage vs(startTime=1);
  Analog.Basic.Ground g;
equation
  connect(vs.n,g.p);
  connect(vs.p,L.p);
  connect(L.n,R1.p);
  connect(L.n,R2.p);
  connect(R1.n,C.p);
  connect(C.n,g.p);
  connect(R2.n,g.p);
end RLC4;
```

Example 3.8. RLC circuit using MSL.

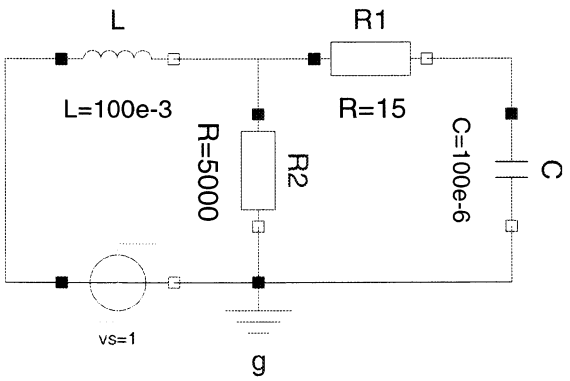

Figure 3.3. Schematic for `RLC4` model in Example 3.8.

[2]The ability to create such schematics requires a tool, such as Dymola, which is capable of parsing and rendering the graphical information contained in the models.

Note how little work was required to build the RLC circuit this time because we did not need to write any component models. The MSL contains a large number of predefined electrical components.

3.4 DEFINING A BLOCK

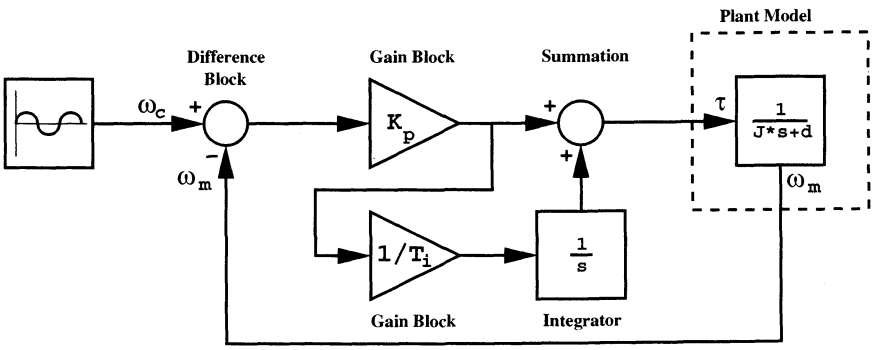

Figure 3.4. PI controller with plant model.

Recall our discussion of the block diagram approach described in Section 1.3.1. Let us consider how such a system could be constructed in Modelica. The block diagram we are interested in modeling is shown in Figure 3.4. This system represents the control of a rotational shaft (the plant) by a PI controller. The PI controller uses a speed sensor to determine the angular velocity of the system and an electric motor as an actuator. The signal ω_c represents the desired angular velocity which, for this example, happens to be a sine wave. The signal ω_m represents the speed sensor reading from the shaft. The combination of the gain blocks and the integrator yields a torque signal, τ which is applied to the plant model. The response of the plant model, given by a transfer function, then provides the feedback signal ω_m.

3.4.1 Equation based approach

If we create a model of the control system shown in Figure 3.4 by simply listing all the equations, the resulting Modelica model would look something like Example 3.9. With this approach, we will find ourselves rewriting equations over and over again. We can avoid this problem by using a component based approach.

3.4.2 Component based approach

Once again, our goal is to create reusable components. This time we will develop models which can be used to build the system shown in Figure 3.4

```
model ControlSystem1 "A PI Controller with Plant Model"
  import Modelica.SIunits;

  parameter Real Kp=0.4, Ti=1.0;
  parameter SIunits.AngularVelocity w0=1.0;
  parameter SIunits.AngularVelocity A=0.2;
  parameter SIunits.Frequency F=0.3;
  parameter SIunits.MomentOfInertia J=0.8;
  parameter SIunits.DampingCoefficient d=0.1;

  SIunits.AngularVelocity w_c, w_m, delta_w;
  SIunits.Torque tau, i, p;
equation
  w_c = A*sin(2.0*Modelica.Constants.pi*F*time)+w0;
  delta_w = w_c-w_m;
  der(i) = Kp*delta_w/Ti;
  p = Kp*delta_w;
  tau = i+p;
  J*der(w_m)+d*w_m = tau;
end ControlSystem1;
```

Example 3.9. A simple control system.

as well as any number of other possible configurations. For this example, we must introduce the concept of a `block`. A `block` is a special kind of `model` where each connector (or its contents) is explicitly marked as either an `input` or `output`. It is expected that each component will provide equations for its outputs written in terms of its inputs.

3.4.2.1 Connector definitions

```
connector Signal
  Real val;
end Signal;
```

Example 3.10. Connector used for a scalar signal.

In order to create our component models, we must know what information is available from the connectors. In this case, the information shared between these models is a single floating point value. For this reason, we define our `connector` as shown in Example 3.10.

3.4.2.2 Creating a sinusoidal signal generator

The natural place to start is with the desired angular velocity signal, $\omega_c = A\sin(2\pi Ft)$. The model for this driving signal is shown in Example 3.11.

```
block SinusoidalSignal
  output Signal out_sig;

  parameter Real A=1.0 "Amplitude";
  parameter Real offset=1.0 "Offset";
  parameter Modelica.SIunits.Frequency F=1.0 "Frequency";
equation
  out_sig.val = offset+
       A*sin(2.0*Modelica.Constants.pi*F*time);
end SinusoidalSignal;
```

Example 3.11. A sinusoidal signal generator.

3.4.2.3 Summation block

The model for a `Summation` block is shown in Example 3.12. Note that this model includes two scale factors which allow the inputs to be scaled independently (*i.e.*, $output = scale_1 * input_1 + scale_2 * input_2$) which increases the reusability of the model. For example, we can use this to turn our `Summation` block into a difference block by setting one of the scale factors to -1.

```
block Summation
  input Signal in_sig1, in_sig2;
  output Signal out_sig;
  parameter Real scale1=1.0 "Scale factor 1";
  parameter Real scale2=1.0 "Scale factor 2";
equation
  out_sig.val = scale1*in_sig1.val+scale2*in_sig2.val;
end Summation;
```

Example 3.12. A block which sums two signals.

3.4.2.4 Integrator block

Our `Integrator` model can be seen in Example 3.13. There is no built-in integration operator. Instead, integrals are represented in terms of derivatives using the `der` operator. In this case, the integration is expressed as:

$$\frac{d}{dt}output = input \tag{3.6}$$

```
block Integrator
  parameter Real init_val=0;
  input Signal in_sig;
  output Signal out_sig(val(start=init_val));
equation
  der(out_sig.val) = in_sig.val;
end Integrator;
```

Example 3.13. An integrator block.

The initial output value for the integrator is given by the `init_val` parameter which is used as the `start` value (from Section 2.5.6.1) for the output signal.

3.4.2.5 Transfer function

Example 3.14 shows a representation of a first order transfer function model. The transfer function is characterized by the following mathematical equation:

$$y(s) = \frac{1}{c_1 s + c_2} u(s)$$

where $u(s)$ represents the input signal and $y(s)$ represents the output signal. Note that the equation contains two characteristic parameters, `c1` and `c2`.

```
block TransferFunction
  input Signal in_sig;
  output Signal out_sig;
  parameter Real c1=0.8;
  parameter Real c2=0.1;
equation
  c1*der(out_sig.val)+c2*out_sig.val = in_sig.val;
end TransferFunction;
```

Example 3.14. A first order transfer function.

3.4.2.6 Gain block

The behavior of a gain block is represented by the following equation:

$$output = k * input \tag{3.7}$$

The `Gain` block model appears in Example 3.15. Can you spot the similarity the `Gain` block has with the models in Examples 3.13 and 3.14? It is a good idea to get in the habit of spotting such similarities for reasons we will explain in the next chapter.

```
block Gain
  input Signal in_sig;
  output Signal out_sig;
  parameter Real K=1.0 "Gain factor";
equation
  out_sig.val = K*in_sig.val;
end Gain;
```

Example 3.15. A multiplier block.

3.4.2.7 Complete control system

At this point, we are able to put all these models together into a complete system. Example 3.16 shows the model, written in terms of component models, for the system shown in Figure 3.4. While Example 3.16 looks more complicated than Example 3.9, component based models are easier to build because they can be constructed graphically.

```
model ControlSystem2 "A PI Controller with Plant Model"
  SinusoidalSignal sinsig(A=0.2,F=0.3,offset=1.0);
  Summation diff(scale2=-1.0), sum;
  Gain KP(K=0.4), KI(K=1.0);
  Integrator integrator;
  TransferFunction motor(c1=0.8,c2=0.1);
equation
  connect(sinsig.out_sig,diff.in_sig1);
  connect(diff.out_sig,KP.in_sig);
  connect(KP.out_sig,KI.in_sig);
  connect(KP.out_sig,sum.in_sig1);
  connect(KI.out_sig,integrator.in_sig);
  connect(integrator.out_sig,sum.in_sig2);
  connect(sum.out_sig,motor.in_sig);
  connect(motor.out_sig,diff.in_sig2);
end ControlSystem2;
```

Example 3.16. A component based control system model for the system shown in Figure 3.4.

3.4.3 Standard block diagram components

Once again, we have gone to a great deal of trouble to implement a collection of models that already exist in the MSL. Remember, the purpose of this chapter is to show you how to write models when they are not available.

3.4.3.1 Connectors

Our `Signal connector` is actually defined using two different `connector` definitions in `Modelica.Blocks`. This is because there is one `connector` for input and another for output (each with the ability to handle an array of signals). The connectors are defined as follows:

```
connector InPort "Connector with Real Inputs"
  parameter Integer n=1 "Signal Array Dimension";
  input Real signal[n] "Real Input Signals";
end InPort;

connector OutPort "Connector with Real Outputs"
  parameter Integer n=1 "Signal Array Dimension";
  output Real signal[n] "Real Output Signals";
end OutPort;
```

These connector definitions can be found in the `Modelica.Blocks.Interfaces` package which contains all the connectors for the `Modelica.Blocks` package. Instead of using:

```
input Signal in_sig;
output Signal out_sig;
```

we can now use the following MSL components, respectively:

```
Modelica.Blocks.Interfaces.InPort in_sig;
Modelica.Blocks.Interfaces.OutPort out_sig;
```

Using the MSL components seems like it would require a great deal more typing. However, a graphical tool would typically be used to construct such systems and so typing would be replaced by simpler "drag and drop" operations. Furthermore, we have already seen examples where the `import` keyword has been used to minimize the amount of typing necessary.

There are some important differences between the connectors defined in the MSL and the one we defined earlier in Example 3.10. The first is that the `input` qualifier is included inside the definition of the `InPort` connector rather than as a qualifier on each `Signal` connector. The presence of the `input` qualifier dictates that the `InPort` connector can only be connected to a "mating" connector.[3] As a result of having the `input` qualifier inside, we eliminate the possibility of accidentally forgetting to include the qualifier while building models.

The other difference is that the `signal` carried by the MSL connectors is an array whereas our `val` was a scalar. By default, the size of the `signal` array is 1 (indicated by the `n` parameter). The fact that `signal` is an array

[3]Mating connectors are ones with complementary **input** and **output** qualifiers

is useful because it allows several signals to be "multiplexed" onto the same connection. Think of it like an electronic cable with multiple wires inside it (*e.g.*, a ribbon cable). More information on using arrays in Modelica can be found in Chapter 6.

3.4.3.2 Other necessary blocks

Now that we have covered the connectors, let us turn our attention to the other blocks required to build our controller. The commanded rotational velocity was previously represented using the `SinusoidalSignal` block. The analogous block in the MSL is `Modelica.Blocks.Sources.Sine` which has parameters for amplitude, frequency and signal offset. The `Modelica.Blocks.Sources` package contains a number of other useful signal generators as well.

The `Gain` block can be replaced by the `Modelica.Blocks.Math.Gain` model from the MSL. The `Modelica.Blocks.Math` package contains the blocks which are used for the algebraic manipulation of signals. In fact, there are two models which could serve as replacements for `Summation`. The first, `Modelica.Blocks.Math.Feedback`, is used in feedback loops (like the one we have in our example). The other model, `Modelica.Blocks.Math.Add` is more like our original `Summation` model because it allows arbitrary gains to be associated with each of the input signals.

Finally, we require blocks which express relationships involving the time derivatives of signals. These can be found in the `Modelica.Blocks.Continuous` package. One useful block from this package is `Modelica.Blocks.Continuous.Integrator` which can serve as a replacement for our previous model (also called `Integrator`[4]). Another useful block from the same nested package is the `Modelica.Blocks.Continuous.TransferFunction` block which is expressed as:

$$y(s) = \frac{b(s)}{a(s)} u(s) \tag{3.8}$$

3.4.3.3 Complete diagram

Now, we have all the pieces we need to build our controller system using components defined in `Modelica.Blocks`. The result is shown in Figure 3.5. The complete Modelica code for the diagram shown in Figure 3.5 is shown in Example 3.17.

[4]There is no potential name conflict here because our previous `Integrator` was defined globally (*i.e.*, not part of a specific package) whereas the `Integrator` model we have just introduced exists within the `Modelica.Blocks.Continuous` package. The reason there is no name conflict is that the `Integrator` model in `Modelica.Blocks.Continuous` must be referenced by its fully qualified name (*i.e.*, `Modelica.Blocks.Continuous.Integrator`).

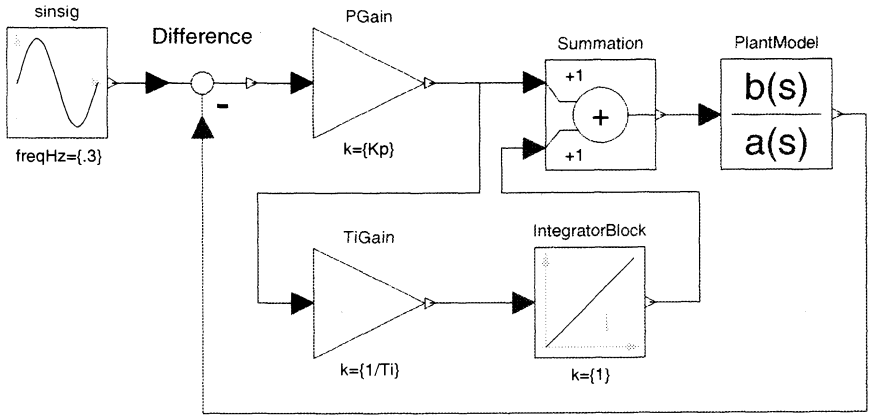

Figure 3.5. Control system model using components from `Modelica.Blocks`.

```
model ControlSystem3
  parameter Real Kp=.4;
  parameter Real Ti=1;
  import Blocks=Modelica.Blocks;
  Blocks.Math.Feedback Difference;
  Blocks.Math.Gain PGain(k={Kp});
  Blocks.Math.Gain TiGain(k={1/Ti});
  Blocks.Math.Add Summation;
  Blocks.Continuous.Integrator IntegratorBlock;
  Blocks.Sources.Sine sinsig(
    amplitude={.2}, freqHz={.3}, offset={1});
  Blocks.Continuous.TransferFunction PlantModel(
    a={.8,.1},b={1});
equation
  connect(sinsig.outPort, Difference.inPort1);
  connect(Difference.outPort, PGain.inPort);
  connect(PGain.outPort, Summation.inPort1);
  connect(PGain.outPort, TiGain.inPort);
  connect(TiGain.outPort, IntegratorBlock.inPort);
  connect(IntegratorBlock.outPort, Summation.inPort2);
  connect(Summation.outPort, PlantModel.inPort);
  connect(PlantModel.outPort, Difference.inPort2);
end ControlSystem3;
```

Example 3.17. Controller and mechanism.

3.5 EXISTING ROTATIONAL COMPONENTS

Let us revisit our simple pendulum model in Section 2.2 and use it to introduce the `Modelica.Mechanics.Rotational` package in the MSL.

3.5.1 Connectors

The `connector` used in the `Modelica.Mechanics.Rotational` library is called a "flange". There are actually two different flange connectors in the MSL. These two connectors are called `Flange_a` and `Flange_b`. With the exception of their graphical representations, these two connectors are identical. The Modelica definition for `Flange_a` is shown in Example 3.18.

```
connector Flange_a "1D Rotational Connector"
  Modelica.SIunits.Angle phi
       "Absolute rotation angle of flange";
  flow Modelica.SIunits.Torque tau
       "Torque applied to the flange";
end Flange_a;
```

Example 3.18. One-dimensional rotational connector.

3.5.2 Special models

Our original model of a pendulum was described by a combination of gravity and inertia. We will represent these behaviors by the model `RotationalPendulum` shown in Example 3.19. The `FrictionlessJoint` model, shown in Example 3.20, will be used to connect the pendulum to a frame of reference.

3.5.2.1 A rotational pendulum model

Recall from Example 2.2 that the torque on the pendulum due to gravity and inertia is:

$$\begin{aligned} \tau &= \tau_g + \tau_i \\ &= mgL\ \sin(\theta) + mL^2\ddot{\theta} \end{aligned} \tag{3.9}$$

Example 3.19 shows the Modelica code for our `RotationalPendulum` model. We use the `connector` definition from Example 3.18 along with the behavioral equations shown in Equation (3.9).

Unlike our previous pendulum example, for this model we need to consider the "free body diagram" for our pendulum. In other words, we need to consider the possibility that some external torque (*e.g.*, due to friction or elasticity) might also contribute to the motion of the pendulum. So, we must assume that the sum of the torques about the pivot point is equal to $\tau_g + \tau_i + \tau_{ext}$ where τ_{ext} is the sum of all external torques.

There are several differences between the `RotationalPendulum` model and the simpler model we created in Chapter 2. The first difference is that the

```
model RotationalPendulum
  import Modelica.SIunits;

  Modelica.Mechanics.Rotational.Interfaces.Flange_a p;
  parameter SIunits.Length L=2.0;
  parameter SIunits.Mass m=1.0;
protected
  SIunits.AngularVelocity omega;
  SIunits.AngularAcceleration alpha;
  parameter SIunits.MomentOfInertia J=m*L^2;
  constant Real g=Modelica.Constants.g_n;
equation
  omega = der(p.phi);
  alpha = der(omega);
  m*g*L*Modelica.Math.sin(p.phi)+J*alpha = p.tau;
end RotationalPendulum;
```

Example 3.19. A rotational pendulum model.

`Flange_a connector` defines the angular position of the pendulum as `phi` but previously we had used `theta` to represent the same thing.

The next difference is that in the previous example the pendulum mass, `m`, was not significant because we could cancel it out of each term in the system of equations. That was possible because we had the complete set of behavioral equations. This time though, we cannot be sure what other terms may be involved (*via* the τ_{ext} contribution) so we cannot cancel the mass out.

Finally, recall that the Modelica convention is that the `flow` quantities on a connector are assumed to be positive when they flow into the component they belong to. This means that the external torque `p.tau` would increase the momentum. So in the absence of gravity, a `p.tau` greater than zero should imply an `alpha` greater than zero. We can use this case to verify that we have used the correct sign for each term.

3.5.3 A frictionless pin model

Previously, we had not considered what relationship our pendulum had with its surroundings. The implicit assumption was that the pendulum was connected to some fixed point by a frictionless bearing. Example 3.20 shows a model which captures the behavior of such a joint. In essence, the joint transmits no torque regardless of the relative angular velocity or position between the pendulum and its surroundings.

```
model FrictionlessJoint
  Modelica.Mechanics.Rotational.Interfaces.Flange_a a;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b b;
equation
  a.tau = 0;
  b.tau = 0;
end FrictionlessJoint;
```

Example 3.20. A frictionless bearing.

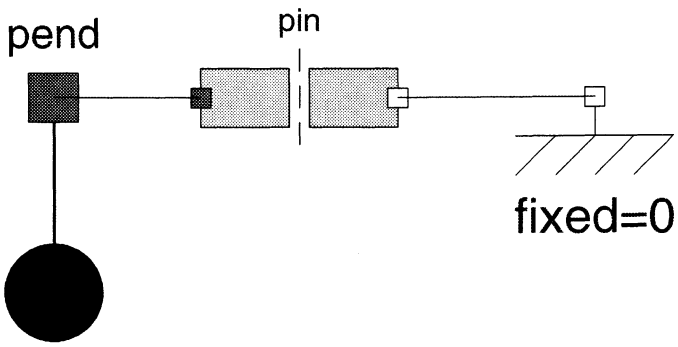

Figure 3.6. A single pendulum system.

3.5.4 A simple rotational system

The Modelica code in Example 3.21 recreates our example from Chapter 2. In addition, Figure 3.6 shows the schematic for this simple system.

```
model PendulumSystem1 "Simple Pendulum"
  RotationalPendulum pend;
  FrictionlessJoint joint;
  Modelica.Mechanics.Rotational.Fixed fixed;
equation
  connect(pend.p,joint.a);
  connect(joint.b,fixed.flange_b);
end PendulumSystem1;
```

Example 3.21. A simple pendulum system.

The `Modelica.Mechanics.Rotational.Fixed` model is analogous to the `Ground` model in Example 3.6 (*i.e.*, it provides a fixed reference for other components).

3.5.5 Building more complex systems

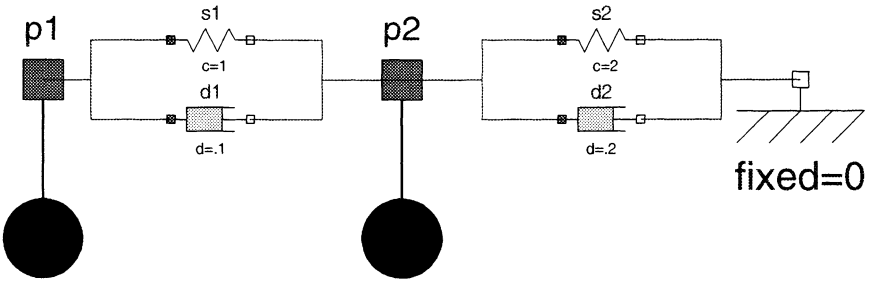

Figure 3.7. A system with multiple pendulums.

One of the great things about having libraries of components is the vast number of combinations that are possible. So far, we have only been concerned with a single pendulum, but what if we wanted to construct a system with multiple springs, dampers and pendulums? Figure 3.7 shows just such a system. The spring, damper and fixed point models all come from the MSL. The code for this system is shown in Example 3.22.

```
model PendulumSystem2 "Simple Pendulum"
  RotationalPendulum p1(m=1,p(phi(start=1,fixed=true)));
  Modelica.Mechanics.Rotational.Spring s1(c=1);
  Modelica.Mechanics.Rotational.Damper d1(d=.1);
  RotationalPendulum p2(L=7,m=.7);
  Modelica.Mechanics.Rotational.Spring s2(c=2);
  Modelica.Mechanics.Rotational.Damper d2(d=.2);
  Modelica.Mechanics.Rotational.Fixed fixed;
equation
  connect(p1.p,s1.flange_a);
  connect(p1.p,d1.flange_a);
  connect(s1.flange_b,p2.p);
  connect(d1.flange_b,p2.p);
  connect(p2.p,s2.flange_a);
  connect(p2.p,d2.flange_a);
  connect(s2.flange_b,fixed.flange_b);
  connect(d2.flange_b,fixed.flange_b);
end PendulumSystem2;
```

Example 3.22. A system with multiple pendulums.

Note that the modifications to pendulum `p1` in Example 3.22 are nested. In other words, modifications to `p1` may include modifications to components

inside `p1` (*e.g.*, `p`). Such recursive modifications allow modifications to be applied to components within a hierarchy.

3.6 LANGUAGE FUNDAMENTALS

This chapter has introduced many new ideas. Let us explore all of these new constructs in a little more depth.

3.6.1 Connections

A `connector` defines information which is to be shared between components. There is no limit on the number of variables which can be declared inside a `connector`. It is even possible to create a `connector` which contains arrays of arbitrary but fixed size like the `InPort` and `OutPort` connectors in `Modelica.Blocks`.

The `connect` keyword is used within an `equation` section to "link" connector instances. The `connect` command always appears in the `equation` section because each connection results in a set of equations being generated. Each connector must have exactly the same number of components with exactly the same names. Furthermore, components with the same name must be type compatible (see Section 2.5.2.2) and `flow` quantities can only be connected to other `flow` quantities.

A `connect` statement includes two arguments. In other words, all `connect` statements should be of the general form:

```
connect(a,b);
```

where `a` and `b` are connectors (subject to the constraints mentioned earlier). The order of the connectors is not significant so there is no difference between the previous connect statement and:

```
connect(b,a);
```

Normally, the effect of using `connect` is that equations are generated which equate each quantity in one connector with its counterpart (with the same name) in the other connector. The exception is when quantities have the `flow` qualifier applied to them. Since `flow` quantities are generally the time derivatives of conserved quantities, connecting them together generates an equation which sums all the `flow` quantities to zero (*e.g.*, Kirchhoff's current law). Quite often a connection involves multiple components (*e.g.*, connection point b in Figure 3.1). In these cases, the sum of the `flow` quantities from all connections is set equal to zero. Finally, if a `connector` is never connected to anything, then any `flow` quantities in that `connector` are **implicitly set to zero.**[5]

[5] Intuitively, this makes sense since nothing can flow through a unconnected `connector` since it would have no place to go.

As mentioned previously, connectors can have parameters associated with them.[6] Such parameters are usually used to indicate the size of one or more array quantities within the connector. Special consideration is given for parameters inside connectors. If a `parameter` appears inside a `connector`, the `connect` statement verifies that the corresponding parameters in the two connectors have the same value but no equations are generated. In other words, the following code would generate an error:

```
connector ArrayCon
  parameter Integer n=2;
  Real x[n];
end ArrayCon;

model Mismatch
  ArrayCon a(n=3);
  ArrayCon b(n=4);
equation
  connect(a,b); // Error, a.n not equal to b.n
end Mismatch;
```

In summary, the `connect` keyword allows us to quickly, safely (*i.e.*, without the possibility of human error) and automatically generate equations appropriate for a given `connector`.

3.6.2 Qualifiers

The `input` qualifier identifies quantities which are intended to be inputs to a model. The `input` qualifier is useful for two different reasons. First, it explains the intent to anybody viewing the source of the model (*i.e.*, it explicitly states that the `input` quantity should originate from an `output` somewhere else in the system). More importantly, connecting two inputs together, at the same level of the hierarchy, is not allowed. In this way, the `input` qualifier essentially **forces** the quantity to be computed by an external model. Note that Figure 3.4 seems to violate this notion that two inputs cannot be connected together because it appears that the inputs of the two gain blocks are connected. However, note that the implementations in Examples 3.16 and 3.17 show that in fact each of the gain block inputs is connected to the output of another block and not to another input.

There is one additional wrinkle in the semantics of the `input` qualifier. It is possible to connect two `input` connectors together if they are not in the same level of the hierarchy. This is necessary in order to allow input signals to be propagated into hierarchies (see Section 4.3 for an example of such usage).

[6] Recall the discussion in Section 3.4.3 of the `InPort` connector in the `Modelica.Blocks` package.

The `output` qualifier is the counterpart to the `input` qualifier. The `output` qualifier indicates that a given quantity is computed by the model in which it is contained. Again, the purpose of this qualifier is to express the intent of the model developer and to prevent certain kinds of connections (*i.e.*, it is not legal to connect together two quantities which both have the `output` qualifier).

The `flow` qualifier is useful in systems where conservation principles are applied (*e.g.*, electrical or mechanical systems). Any quantity identified as a `flow` is summed at each connection point. The `flow` qualifier is generally applied to components with a physical type (*e.g.*, `Current`, `Power`) that is the time derivative of conserved quantities (*e.g.*, charge, energy). By convention, a positive value for a `flow` in a connector represents a flow **into** the model which contains the connector. Note that we follow this convention in all examples.

As a general rule, qualifiers applied to a connector are effectively propagated into every member of the connector. When developing a connector you should consider whether you want the qualifier to be applied by the definition:

```
connector ConnectorA
  input Real x;
  input Real y;
end ConnectorA;

model ModelB
  ConnectorA a;
end ModelB;
```

or at the time of declaration:

```
connector ConnectorA
  Real x;
  Real y;
end ConnectorA;

model ModelB
  input ConnectorA a;
end ModelB;
```

In both cases, the components `a.x` and `a.y` are considered inputs. The former case is useful when a connector is meant to be used as either an `input` or `output` connector. The latter case is useful to force the connector to have a specified directionality, either `input` or `output`.[7] Finally, another reason for placing the qualifiers inside the connector would be to create "mating" connectors with complementary `input` and `output` qualifiers, *e.g.*,

[7]Note that the connectors for the `Modelica.Blocks` package (discussed briefly in Section 3.4.3) place the **`input`** and **`output`** qualifiers inside the connector definition to force them to play either an **`input`** or **`output`** role.

```
connector Plug
  input Real x;
  output Real y;
end Plug;

connector Socket
  output Real x;
  input Real y;
end Socket;
```

3.6.3 Modifications

When a model is written, default values for parameters are often provided by the model itself. As we have shown in Example 3.7, we can override these internal defaults when the component is declared. Changes to attributes, like the `start`, `min` and `unit` attributes discussed in Section 2.5.6, are also considered modifications.

Modifications can be applied to components throughout a component hierarchy. For example, consider the following example:

```
model Circuit
  Resistor R1, R2
  Capacitor C1, C2, C3(C=1e-6);
  ElectricalPin p, v;
equation
  ...
end Circuit;

model Appliance
  Circuit c(R1(R=12),C2(C=1e-3));
  ...
end Appliance;
```

In this case, from the `Appliance` level, we have applied a modification to a resistor and capacitor inside the `Circuit` component. As can be seen by this example, modifications on nested components are made within nested pairs of parentheses.

If we had attempted to modify capacitor `C3` from the `Appliance` level, we would have overridden the modification included in the `Circuit` model. As a general rule, modifications in a declaration always override modifications lower down in the hierarchy. However, there are many cases not covered by this rule. The precise rules about the precedence of modifications can be found in the Modelica language specification included on the companion CD-ROM.

Another example of hierarchical modification can be seen in Example 3.22. Instead of parameters, attributes of the `phi` variable associated with the `p` pin on the `RotationalPendulum` were modified.

3.6.4 Defining a block

As mentioned in our control system example, a `block` is a special case of a `model` where all connectors (or the contents of the connectors) are marked as either `input` or `output`. Designating something as a `block` has the potential to simplify model processing, increase simulation speed and improve the quality of diagnostic messages.

3.6.5 Finding and using component models

So far, when we have required physical units (*e.g.,* `Voltage`), we have included the line "`import Modelica.SIunits;`" at the start of our models. This allows us to use `SIunits` as an alias for the full library path where the physical unit information resides. We have to include this path because Modelica tools will not automatically search through the hierarchies of models, types, *etc.* to find something they are not familiar with. Here are three things you can do to make sure a tool can locate the definition you wish to use:

- **Use the full name**: You can use the full name (*e.g.,* `Modelica.Constants.pi` or `Modelica.SIunits.Pressure`) to refer to an entity you require within a model. Do not worry for the moment where these models are stored (to be discussed later in Chapter 9).
- **Define an alias**: For example, if we wish to access the types contained in `Modelica.SIunits` package, we can include the line:

  ```
  import Modelica.SIunits;
  ```

  within our models which creates the `SIunits` alias.
- **Place models in the same directory**: One way to make sure your model definitions are found is to store them in the directory you are working in. If you place all models in the same directory then there should be no difficulty in finding them. While we can do this for our simple examples, there are better ways of storing models which will be discussed later in Chapter 12.

This is a simplified version of the lookup rules. For a complete understanding of how such lookups are done, consult Chapter 9 or the language specification included on the companion CD-ROM.

For all of our examples, we assume that the other component models we have written are in the same directory as the models which use them. In this way, we can use component models like `Resistor` or `TransferFunction` without having to provide a qualified hierarchical name (*e.g.,* `Modelica.SIunits.Pressure`).

3.7 SUMMARY

Component based approaches have the following advantages:

1. The constitutive equations for a component need only be written once (*i.e.*, within the component model).

2. Hierarchies of components (possibly many levels deep) can be created and such hierarchies are much easier to understand compared to a "flattened" representation where all parameters, variables and equations are present in a single model.

3. By using the `connect` keyword, we can automatically generate multiple equations for a single connector. In addition, this is not as error prone as writing the equations by hand.

4. Restrictions can be imposed against connections which do not make sense (*e.g.*, connecting two `input` quantities or mixing `flow` variables with non-`flow` variables).

In summary, these are the reasons why component based approaches to model development are superior to equation based approaches for large problems.

3.8 PROBLEMS

PROBLEM 3.1 *Use the electrical components developed in this chapter as a guide to develop analogous components in other domains. For example, a translational system has the following equations:*

$$F = k\Delta x \tag{3.10}$$

$$F = c\frac{d}{dt}(\Delta x) \tag{3.11}$$

$$F = m\frac{d^2x}{dt^2} \tag{3.12}$$

to represent the behavior of springs, dampers and inertias where F represents force transmitted by an element, k is the spring stiffness, c is the damping coefficient and m is the mass.

For a given analogous domain, what are the variables associated with the connectors in that domain and which components in that domain correspond to the resistor, capacitor and inductor components in the electrical domain?

PROBLEM 3.2 *Develop a model for the circuit shown in Example 3.7 using the block diagram component models from the Examples in Section 3.4. Assume that the input to the system is the voltage at pin a in Figure 3.1 and the output we are interested in is the voltage at pin c. You might start by writing down all the constitutive and conservation equations and trying to formulate them in such a way that each equation can be represented by a block in the block diagram.*

PROBLEM 3.3 *Browse the* MSL *and look at the* `connector` *definitions for different domains. Are there any common themes?*

PROBLEM 3.4 *Implement a non-linear rotational spring with the following constitutive equation:*

$$\tau = c_1(\phi_a - \phi_b) + c_2(\phi_a - \phi_b)^3 \qquad (3.13)$$

where τ represents torque, ϕ represents angular position and the subscripts, a and b, represent connectors. Once implemented, use the the spring in a system that contains components from the MSL.

PROBLEM 3.5 *The power output of a resistor can be found by taking the product of the current through the resistor, i, and the voltage across the resistor, v. Create a model of a resistor whose resistance is a function of temperature T as follows:*

$$R = R_0 + S(T - T_r)$$

where T_r is the reference temperatures, R_0 is the nominal resistance at the reference temperature, T is the temperature of the resistor and S is the (linearized) sensitivity of the resistance with respect to temperature. Compute the temperature of the resistor using the following energy balance:

$$mc_p\dot{T} = iv - h(T - T_{amb})$$

where m is the mass of the resistor, c_p is the specific heat capacity of the resistor, h is the convective heat transfer coefficient and T_{amb} is the ambient temperature. Examine how this resistor performs in some of our RLC example circuits with different parameter values(e.g., does the natural frequency of the oscillations change?). Here are some sample values to try:

$$\begin{aligned} R_0 &= R \text{ (the normal resistance)} \\ T_{amb} &= 400 \\ T_r &= 300 \\ m &= 0.01 \\ c_p &= 384 \\ h &= 0.1 \\ S &= 0.01 \end{aligned}$$

Chapter 4

ENABLING REUSE

4.1 CONCEPTS

Knowing how to build components is only the first step in an efficient model development process. In order to maximize the usefulness of these components it is necessary to understand how to make them reusable. The Modelica Standard Library is a good example of a reusable collection of components.

Many of the features in Modelica exist to promote reuse. The object-oriented nature of Modelica was specifically introduced to parallel the reuse capabilities of languages like Ada and C++. While not necessary, being familiar with such languages and the techniques used to promote reuse in those languages will help in understanding similar features in Modelica.

There are several aspects to making models reusable in Modelica. For example, creating a set of component models that work together requires the use of common `connector` definitions in order for them to share information with each other. In Chapter 3, we showed that `connector` definitions for several domains have already been defined in the MSL. By using the MSL `connector` definitions, we can create new models that are compatible with the rich collection of existing models.

Reusability is also achieved by extending existing models. As we will show in this chapter, this technique allows common sets of equations, parameters, algorithms, *etc.* to be shared between models. While this chapter introduces functionality in the Modelica language to promote reuse, the usefulness of this kind of reusability may only become clear after you have attempted to create a non-trivial collection of models. As usual, we begin with several examples and include a summary of the features which promote reuse in the last section of this chapter.

An important aspect of physical system modeling is to exploit the reusability gained by using acausal modeling formulations. In general, acausal models (rather than block diagrams) are easier to reuse because each component model can be formulated independently without knowledge of the equations or causality assumptions used in other parts of the system. This is an issue because the *causality* for physical component models changes depending on the context in which the model is used. While we do not discuss the details in this chapter, this aspect of reusability is discussed in greater detail in Chapter 11.

One final aspect of reusability worth mentioning is that reusability is affected by the quality of model documentation. The better the documentation, the easier (and therefore more likely) it will be for others to reuse your models.

4.2 EXPLOITING COMMONALITY

In general, having the same code fragment appear multiple times in different locations usually leads to problems. This is true in languages such as C++ and FORTRAN as well as in Modelica. This redundancy is bad because it makes maintenance difficult. For example, if a bug is found in one copy of such a code fragment it is difficult to track down all places where that same code may have been repeated. In Modelica, such redundancies might include repeated equations, repeated parameter or repeated connector definitions. In this section we will describe how to avoid such redundancy.

4.2.1 Identifying commonality

Let us revisit the models presented in Section 3.3. Looking at Examples 3.2-3.5, can you see the similarities between these models? As it turns out, these models have many things in common. Note that the electrical models presented use the voltage drop across them and the current flow through them in their constitutive equations. Furthermore, they all have two pins with the same names, `p` and `n`. In other words, there is a good deal of repetition between those models.

In order to help avoid this repetition, the `Modelica.Electrical.Analog` **package** defines (in its nested `Interfaces` package) a **partial** model called `OnePort`. A `OnePort` component is one which has exactly two electrical pins associated with it and therefore only one current path through it. Example 4.1 shows the `OnePort` model which represents the common subset between the `Resistor`, `Capacitor`, `Inductor` and `VoltageSource` models. An important thing to note about the `OnePort` model is that its definition is qualified with the **partial** keyword which implies that this model is not complete but merely a base on which to build other models.

```
partial model OnePort "Two pinned electrical component"
  Modelica.SIunits.Voltage v "Voltage from pin p to pin n";
  Modelica.SIunits.Current i "Current entering at pin p";
  Modelica.Electrical.Analog.Interfaces.Pin p "Positive";
  Modelica.Electrical.Analog.Interfaces.Pin n "Negative";
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;
```

Example 4.1. Defining a common base model for one port electrical components.

4.2.2 Extending from a common definition

The advantage of defining a `partial` definition like the one shown in Example 4.1 is that we can **reuse** the different pieces (*e.g.*, connections, types, variables or equations) of the `OnePort` model when writing models for a `Resistor` or `VoltageSource`. Example 4.2 shows how compact the definition of the `Resistor` model becomes by using the `OnePort` definition.

```
model Resistor "An electrical resistor"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  parameter Modelica.SIunits.Resistance R=300 "Resistance";
equation
  i*R = v;
end Resistor;
```

Example 4.2. Model for `Resistor` using `OnePort`.

The definition shown in Example 4.2 succinctly describes what a `Resistor` is. At a glance we can easily surmise that a `Resistor` is something which shares the same characteristics as a `OnePort` with an additional parameter `R` of type `Resistance` and a constitutive relationship between `i` and `v` described by the equation `i*R=v`.

4.3 REUSABLE BUILDING BLOCKS

So far, we have shown how to create new models by extending existing ones. In this section we describe how to create new models by combining several existing models (*e.g.*, to create `package` definitions containing commonly used configurations of existing models).

4.3.1 Building a controller model

In Section 3.4, we showed a complete system containing a controller and plant model. In practice, we would not want to create a new PI controller from scratch for each system that needed a PI controller. Instead, we should build a PI controller model that we can include in any model that needs one. Figure 4.1 shows a PI controller model built using the `Modelica.Blocks` package. Example 4.3 shows the Modelica source code for the controller. Note the similarity of Example 4.3 to Example 3.17.

Models which are built from other models can be visualized in two ways. The first way is to look at them as schematic diagrams like the one shown in Figure 4.1.

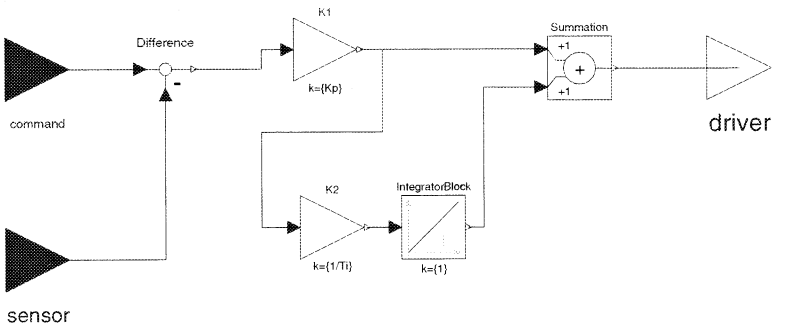

Figure 4.1. The diagram view of `PIController`.

The second way is to look at our model from the "outside". This is called the *icon view* and Figure 4.2 shows the `PIController` from this perspective. The icon view hides the internal details of the model and presents only a "black-box" representation. The input connectors, `command` and `sensor`, and the output connector, `driver`, shown in Figure 4.1 represent the external connections of the `PIController` model. These correspond to the connectors visible in Figure 4.2.

Figure 4.1 shows how a component model (*e.g.*, `PIController`) can be built from other models (*e.g.*, `Summation` and `Gain`). Think of the subcomponent models as building blocks. Ultimately, this approach makes the building and enhancement of complex systems easier.

Example 4.4 shows how we can use the `PIController` to create a system equivalent to the one presented in Example 3.17. Example 4.4 demonstrates how much simpler models become when we encapsulate the details of particular components.

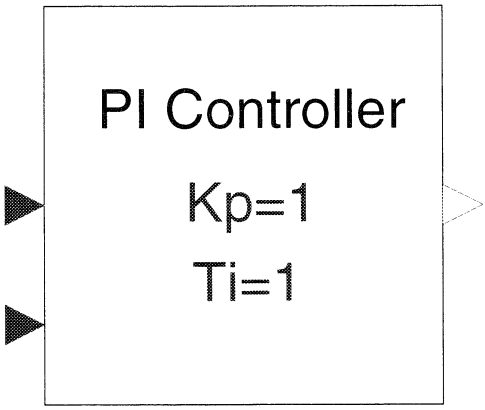

Figure 4.2. `PIController` model icon.

```
block PIController "A PI Controller"
  parameter Real Kp=1 "Proportional Gain";
  parameter Real Ti=1 "Integral Time Constant";
  import Modelica.Blocks;

  Blocks.Interfaces.InPort command "Command signal";
  Blocks.Interfaces.InPort sensor "Sensor signal";
  Blocks.Interfaces.OutPort driver "Driver signal";
  Blocks.Math.Feedback Difference;
  Blocks.Math.Gain K1(k={Kp});
  Blocks.Math.Gain K2(k={1/Ti});
  Blocks.Math.Add Summation;
  Blocks.Continuous.Integrator IntegratorBlock;
equation
  connect(command, Difference.inPort1);
  connect(sensor, Difference.inPort2);
  connect(Difference.outPort, K1.inPort);
  connect(K1.outPort, Summation.inPort1);
  connect(K1.outPort, K2.inPort);
  connect(K2.outPort, IntegratorBlock.inPort);
  connect(IntegratorBlock.outPort, Summation.inPort2);
  connect(Summation.outPort, driver);
end PIController;
```

Example 4.3. Source code for the PI controller model in Figure 4.1.

4.3.2 Propagating information

When building a new model using a collection of subcomponents it is necessary to propagate connections and parameters down through the hierarchy of subcomponents.

```
model PIControllerAndMotor
  import Modelica.Blocks;

  Blocks.Sources.Sine sinsig(
       amplitude={0.2}, freqHz={0.3}, offset={1.0});
  PIController pic(Kp=0.4);
  Blocks.Continuous.TransferFunction motor(a={0.8,0.1});
equation
  connect(sinsig.outPort,pic.command);
  connect(pic.driver,motor.inPort);
  connect(motor.outPort,pic.sensor);
end PIControllerAndMotor;
```

Example 4.4. A PI controller controlling a motor.

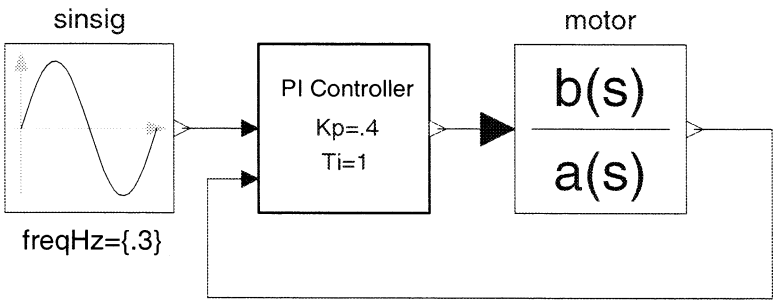

Figure 4.3. `PIControllerAndMotor` model.

4.3.2.1 Parameters

The `PIController` shown in Example 4.3 is characterized by the two parameters at the top of the model. These parameters are the proportional gain, `Kp`, and the integral time constant, `Ti`. We need to propagate these parameters to the gain block subcomponents. We do this when we declare the gain blocks. Note how, in Example 4.3, `Kp` and `Ti` are used to set the gain parameter, `k`, for subcomponents `K1` and `K2`, respectively.

4.3.2.2 Connections

Propagating connections is quite straightforward. To establish a connection between the external connection of a model and one of its subcomponents, the `connect` command is used to connect two connectors just like it would be for any other connection. Previously in Section 3.6.1 we mentioned that two `input` connectors could not be connected using `connect` **unless** they were at different hierarchical levels. Example 4.3 is an example of why this exception exists.

Note the connection between `command` and `Difference.inPort1`. Both of these connectors are input connectors from the `Modelica.Blocks` package but because they are at different hierarchical levels[1], this connection is allowed. This allows external input signals to be passed down through the hierarchy of subcomponents.

4.3.3 Summary

When building components from other "building block" components, the first step is to determine what the external connections and parameters should be. In other words, what should this model look like from the "outside". Next, we declare and connect the internal components. Finally, we propagate the external parameters and connectors to the subcomponents.

4.4 ALLOWING REPLACEABLE COMPONENTS

Example 4.4 in the previous section shows a motor being controlled by one specific controller. In this section, our goal is to demonstrate how we can easily "plug and play" different kinds of controllers and compare their performance. In order for one model to replace another, the two models must be compatible. In other words, the new model must have the same connectors and parameters as the old model. In the technical vocabulary of Modelica, the new model must be a "subtype" of the model it is replacing (see Section 4.8.3 for a complete description of the subtype concept).

4.4.1 The generic controller interface

One way to ensure that the controller blocks are compatible is to extend them from a common `partial` block. To do this, we need to establish a common interface for our controllers. We accomplish this the same way we did in Section 4.2. Specifically, we identify the common attributes of a controller and create a `partial block` that all of our controllers can be extended from.

The inputs to our generic models are a command signal and a sensor signal. The model should also have a single output for the driver signal. All these signals were present in our `PIController` model. Now, we consider the parameters for our generic controller. We cannot assume that all controllers have the same parameters, `Kp` and `Ti`, that our `PIController` has. In fact, we cannot make any assumptions about what parameters a controller might have. After considering all this, Example 4.5 shows what our generic controller model would look like.

[1] These connectors are at different hierarchical levels because one connector belongs to the `PIController` model directly (*i.e.*, it is declared within the `PIController` model) while the other connector belongs to the `Difference` component which is nested inside the `PIController`.

```
partial block Controller "A generic controller interface"
  import Modelica.Blocks;

  Blocks.Interfaces.InPort command "Command signal";
  Blocks.Interfaces.InPort sensor "Sensor signal";
  Blocks.Interfaces.OutPort driver "Driver output signal";
end Controller;
```

Example 4.5. A generic controller interface.

4.4.2 Specific controller models

Let us use the `Controller` definition in Example 4.5 to create several different types of controllers so we can do some comparisons.

4.4.2.1 Proportional controller

Example 4.6 shows a proportional gain controller that is defined by extending from the `Controller` model shown in Example 4.5. The equation for a proportional gain controller is:

$$driver = K_p * e \tag{4.1}$$

where K_p is the gain of the controller (represented by the `Kp` parameter in the model) and e is the difference between the commanded and measured response. In Example 4.6, we have used two simple equations to express the mathematical relationship between the inputs and outputs. This is in contrast to Example 4.3 where we used the mathematical blocks, such as the `Gain` block, found in the `Modelica.Blocks` package.

```
block PController "A proporational gain controller"
  extends Controller;
  parameter Real Kp=1 "Proportional gain";
protected
  Real e "reference error";
equation
  e = command.signal[1] - sensor.signal[1];
  driver.signal[1] = Kp*e;
end PController;
```

Example 4.6. A proportional gain controller.

4.4.2.2 Proportional-differential controller

Example 4.7 shows how an ideal proportional-differential controller can be created by extending the `Controller` interface. Again, we have chosen to

write the mathematical relationship, *e.g.*,

$$driver = K_p * e + K_d * \frac{de}{dt} \tag{4.2}$$

```
block PDController "An ideal PD controller"
  extends Controller;
  parameter Real Kp=1 "Proportional gain";
  parameter Real Kd=1 "Differential gain";
protected
  Real e "reference error";
equation
  e = command.signal[1] - sensor.signal[1];
  driver.signal[1] = Kp*e + Kd*der(e);
end PDController;
```

Example 4.7. An ideal proportional-differential gain controller.

4.4.2.3 Proportional-integral controller

Each of the other controllers described in this section `extends` from the `Controller` model. In this way, they inherit all the components they need. Another benefit of extending from the `Controller` model is that each of the other controllers is automatically a subtype of the `Controller` model.

In the case of the `PIController` model, we do not have to rewrite the model to extend from the partial model `Controller`. We can continue to use the model as it appears in Example 4.3. This is because our existing `PIController` can be considered a subtype (*i.e.*, specialization) of the `Controller` model because it contains all of the components of the `Controller` model. While using `extends`, as we did in Examples 4.6 and 4.7, is convenient because it automatically includes everything that is needed, it is not necessary in order for a model to be considered a subtype.

4.4.3 Using replaceable components

Let us create a new version of the model shown in Example 4.4 that allows us to replace the controller model. The new model for this system is shown in Example 4.8.

The only difference between Example 4.4 and Example 4.8 is the change of the controller declaration from:

```
PIController pic(Kp=0.4);
```

to:

```
replaceable PIController con(Kp=0.4) extends Controller;
```

Think of the new declaration as saying: "Declare a `PIController` called `con` that can be replaced by any component which is a subtype of `Controller`". In essence, the **`extends`** qualifier at the end of the declaration represents a constraint on the model being declared and any models it may be replaced with.

```
model ControllerAndMotor
  import Modelica.Blocks;

  Blocks.Sources.Sine sinsig(
    amplitude={0.2}, freqHz={0.3}, offset={1.0});
  replaceable PIController con(Kp=0.4) extends Controller;
  Blocks.Continuous.TransferFunction motor(a={0.8,0.1});
equation
  connect(sinsig.outPort,con.command);
  connect(con.driver,motor.inPort);
  connect(motor.outPort,con.sensor);
end ControllerAndMotor;
```

Example 4.8. A system containing a controller and motor.

Now, let us create a model which tests all of these controllers side by side. Example 4.9 shows a system composed of three instances of our `ControllerAndMotor` model. Note that the first instance does not make any modification and therefore uses the default `PIController`. The next instance redeclares the controller component `con`, inside the `ControllerAndMotor` model, to be of type `PController`. The last instance also redeclares the controller, but this time it specifies the controller type to be `PDController`. Figure 4.4 shows the results of a side by side comparison between the 3 controller models.

```
model CompareControllers "Comparing various controllers"
  ControllerAndMotor pic;
  ControllerAndMotor pc(
    redeclare PController con(Kp=1.1));
  ControllerAndMotor pdc(
    redeclare PDController con(Kp=1.1,Kd=.2));
end CompareControllers;
```

Example 4.9. A comparison of controllers using **`redeclare`**.

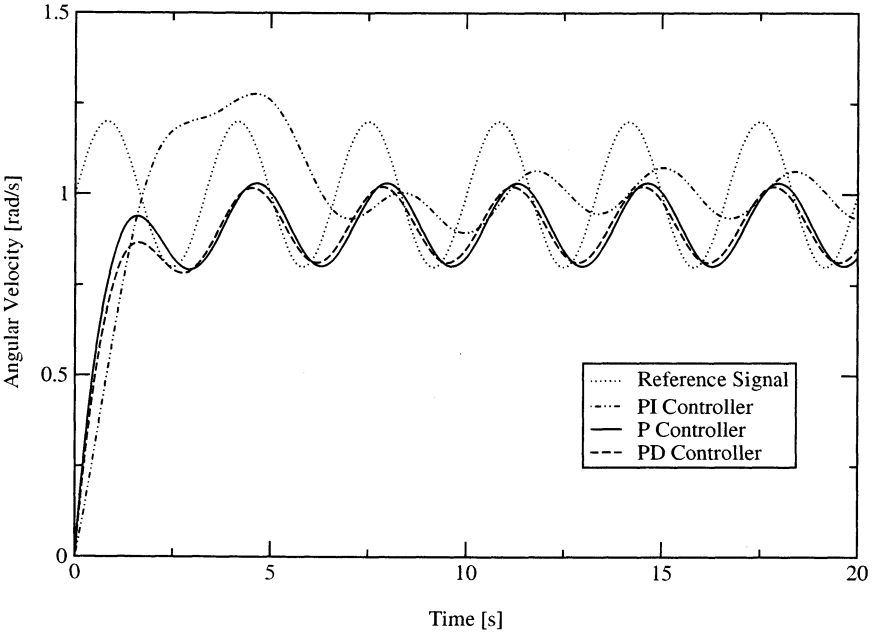

Figure 4.4. Side by side comparison of controllers.

4.4.4 Summary

In this section, we have shown how to design components or systems that allow us to easily substitute one model for another. For our example, we used this capability to do a side by side comparison of different controller types.

Another use for this functionality is to substitute models with different levels of detail. When designing controllers, the plant models are often linearized and expressed in terms of transfer functions (like our motor). However, in practice it is useful to replace these simple linear models with detailed models which consider non-linear effects (*e.g.*, backlash) and then re-simulate the system to compare how different controllers perform in the presence of such non-linearities.

4.5 OTHER REPLACEABLE ENTITIES

In Section 4.4, we showed how components could be made `replaceable`. Besides making components replaceable, the `replaceable` keyword can also be used to make the types of components replaceable. Instead of just redeclaring one component, this feature allows several components to be redeclared simultaneously.

Before going into the details of how this is done, we must first cover a few other features of the Modelica language. Let us revisit Example 3.8. Note that the two resistor components in that example are declared as follows:

```
...
Analog.Basic.Resistor R1(R=15);
Analog.Basic.Resistor R2(R=5000);
...
```

Imagine we wanted to save ourselves from having to type out the full name of the resistor model. To do this, we could define a local `model` and then declare `R1` and `R2` using our local `model` as follows:

```
...
model ResModel
  extends Analog.Basic.Resistor;
end ResModel;
ResModel R1(R=15);
ResModel R2(R=5000);
...
```

We do not appear to have saved ourselves too much typing. However, we can accomplish exactly the same thing by using an abbreviated way of extending a model called a short definition. Our previous code fragment can be rewritten as:

```
...
model ResModel=Analog.Basic.Resistor;
ResModel R1(R=15);
ResModel R2(R=5000);
...
```

In other words,

```
model ResModel=Analog.Basic.Resistor;
```

is equivalent to

```
model ResModel
  extends Analog.Basic.Resistor;
end ResModel;
```

Now, we can use a short definition in conjunction with the `replaceable` qualifier to allow the type of resistor used to be easily redeclared. Example 4.10, a variation on Example 3.8, shows how a `replaceable` object type can be implemented. The difference is that the `RLC5` model defines a local `model` called `ResModel`. This local `model` is then used when declaring components `R1` and `R2`. Figure 4.5 shows which components in the schematic use the `ResModel` definition. Note that the definition of the `model Resistor` is just a short definition with the `replaceable` qualifier in front of it.

```
model RLC5 "An RLC circuit using standard components"
  import Modelica.Electrical.Analog;

  replaceable model ResModel=Analog.Basic.Resistor;
  ResModel R1(R=15);
  ResModel R2(R=5000);
  Analog.Basic.Capacitor C(C=100e-6);
  Analog.Basic.Inductor L(L=100e-3);
  Analog.Sources.StepVoltage vs(startTime=1);
  Analog.Basic.Ground g;
equation
  connect(vs.n,g.p);
  connect(vs.p,L.p);
  connect(L.n,R1.p);
  connect(L.n,R2.p);
  connect(R1.n,C.p);
  connect(C.n,g.p);
  connect(R2.n,g.p);
end RLC5;
```

Example 4.10. An example of how to redeclare several components.

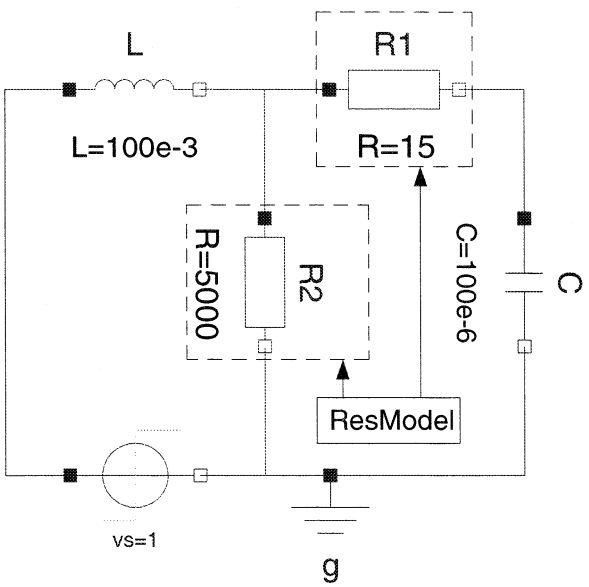

Figure 4.5. Schematic for Example 4.10.

The advantage of defining a local `model` as shown in Example 4.10 is that all instances of that model can be redeclared simultaneously. For example, if we wanted to create a new version of the circuit shown in Example 4.10 using a different resistor model, we would do something like the following:

```
model MyRLC=RLC5(redeclare model ResModel=MyResistor);
```

This code fragment shows another example of how a short definition can be used. In this case, the `MyRLC` model extends from `RLC5` but it redeclares the `ResModel` model to be an extension of some other resistor model called `MyResistor`. The result is that components `R1` and `R2` within the `MyRLC` model will be declared as instances of the `MyResistor` model. Note that the syntax of a redeclaration is the same as a declaration except that it is preceded by the keyword `redeclare`.

4.6 LIMITING FLEXIBILITY

The focus of this chapter is how to make components more reusable because this translates into greater flexibility and less duplication of effort. For that reason, it may seem strange to introduce a section to discuss limiting the reusability of a definition. However, there is good reason for this because, in addition to being reusable, we also want our models to be robust.

To understand why flexibility must sometimes be limited, let us consider one problem with Example 4.3. Note that the parameter `Ti` is used to give a value to the gain parameter, `k`, of gain block `K2`. First, let us consider a typical use of the `PIController` model:

```
model ControllerApplication1
  ...
  PIController con(Kp=0.4,Ti=0.2);
equation
  ...
end ControllerApplication1;
```

Note that the `Ti` parameter was set to 0.2. This should result in the `k` parameter of gain block `K2` being set to 5. New model developers may not immediately notice the details of Example 4.3 so they may not realize that the `Ti` parameter gets propagated to the `k` parameter of one of the gain blocks. As a result, they might try to directly set the `k` parameter themselves as follows:

```
model ControllerApplication2
  ...
  PIController con(Kp=0.4,K2(k=5));
equation
  ...
end ControllerApplication2;
```

The intention of the `PIController` model developer was that only parameters `Kp` and `Ti` should be set by users of the model and that parameters inside the `PIController` hierarchy should not be changed. To prevent such changes, the declaration of the `K2` block within the `PIController` model should be changed to:

```
Blocks.Math.Gain K2(final k={1/Ti});
```

The use of the `final` keyword in this way indicates that no further modifications to the parameter `k` are allowed. Any attempt to do so, like we showed previously in `ControlApplication2`, will trigger an error.

This kind of limitation can be important when building up configurations of existing components. This is because while a new model may be built using instances of existing models, it may be desirable for the new model to behave like a black box (*i.e.*, not allowing others to get inside and change some of the underlying assumptions). The `final` keyword can be used to disallow such changes.

```
model Gear
  import Modelica.Mechanics.Rotational;

  Rotational.Interfaces.Flange_a flange_a;
  Rotational.Interfaces.Flange_b flange_b;
  parameter Real gear_ratio=1
     "Gear ratio (flange_a.phi/flange_b.phi)";
equation
  flange_a.phi = ratio*flange_b.phi;
  0 = ratio*flange_a.tau + flange_b.tau;
end Gear;
```

Example 4.11. A simple gear model.

Another example of where `final` might be used is in the development of a timing belt model. In an internal combustion engine, the timing belt connects the crankshaft to the camshaft. The timing belt is designed so that two revolutions of the crankshaft result in one revolution of the camshaft. So it appears a timing belt is just a special case of a gear with a gear ratio of two. Example 4.11 shows one way of writing a gear model using the `Modelica.Mechanics.Rotational` library.[2]

We can reuse this `Gear` model to create our `TimingBelt` model as follows:

[2]This is nearly identical to the ideal gear model found in the MSL with the name `Modelica.Mechanics.Rotational.IdealGear`

```
model TimingBelt=Gear(final gear_ratio=2);
```

The `final` keyword prevents any subsequent change in the gear ratio by instantiation or specialization. As a result, the following examples would be prohibited:

```
model Engine
  TimingBelt belt(gear_ratio=3); // Error
  model OddTimingBelt=
        TimingBelt(gear_ratio=2.25); //Error
end Engine;
```

This is because the `gear_ratio` component of `TimingBelt` was declared `final`.

4.7 OTHER CONSIDERATIONS

4.7.1 Parameters

It is usually undesirable for a model to have numbers appearing directly in equations. For example, the `SimplePendulum` model in Example 2.1 could have been written as:

```
model SimplePendulum
  Real theta, omega;
equation
  der(theta) = omega;
  der(omega) = -4.905*theta;
end SimplePendulum;
```

However, this not only hides the fact that 4.905 is really g/L but it also prevents the length from being easily changed. Making L a `parameter` and g a `constant` makes the model easier to understand and easier to reuse (*i.e.*, allowing different values of L).

4.7.2 Generality

The most reusable models are the ones that make the fewest assumptions. Unfortunately, the more general a model, the more abstract and complicated it tends to be.

For example, we could have solved the pendulum dynamics described in Equation (2.6) by first developing a general model that described the motion of a mass in three-dimensional space. Then, we could have created a simple pendulum model that extended the general model and included additional equations to constrain the motion of the mass to be that of a simple planar pendulum. In the case of our `SimplePendulum` model, it would probably be simpler just to create a model directly from Equation (2.6) than to reuse a more general existing model by adding constraints.

When developing a reusable model try to understand how it will be reused. The model should be general enough that it can be used as the basis for several other models. At the same time, it must be simple enough that using it as the basis for developing other models leads to models that are simple and easy to understand. The `OnePort` model, shown in Example 4.1, is a good example because it satisfies both of these criteria.

4.7.3 Documentation

An important part of reusability is documentation. There are several different types of documentation that should be part of any reusable model. First, descriptive text can be used when the model and any of its components are declared. Refer to Example 2.4 for a demonstration of how this is done. In addition, comments (starting with "//") can be used within the model text to explain the origin of equations or logic of algorithms. Finally, we shall see in Section 9.2.2 that it is also possible to embed HTML code that describes the details behind the model.

The ability to document models in this way is a deliberate attempt, by the designers of the Modelica language, to give model developers every opportunity to explain the details of their models. By documenting models in this way, the **documentation moves with the model** which makes it easier to update the documentation as the model changes.

4.8 LANGUAGE FUNDAMENTALS

4.8.1 Extending a model

The basic idea behind the `extends` keyword is that, roughly speaking, it allows you to "copy and paste" the contents of one model into another. This is useful when many models have the same variables, connectors, equations, *etc.*

Through the examples in this chapter, we have shown that once a common subset is identified, other models can be specialized from the common subset using the `extends` keyword. Example 4.2 shows how the `Resistor` model was made simpler and easier to understand by deriving it from a base model. It is even possible to extend from two or more base models.

There are several important restrictions to keep in mind when using `extends`. The first restriction is that you cannot replace any of the equations in a model you are extending. For example, if we define a model `A` as follows:

```
model A
  Real x;
equation
  x = 5;
end A;
```

and then we extend `A` to create a new model `B1` which looks like this:

```
model B1
  extends A;
equation
  x = 3;
end B1;
```

The definition of `B1` above is equivalent to the following definition of `B2` which does not utilize `extends`:

```
model B2
  Real x;
equation
  x = 5;
  x = 3;
end B2;
```

As a result, in both `B1` and `B2` we end up creating a model with two equations which is probably not our intention because this results in a system of two equations with only one unknown (*i.e.*, it is over-determined).

Care should be taken when developing `partial` models to give derived models sufficient flexibility for their equations by including the minimum number of equations in the `partial` model.

4.8.2 Short definitions

In Example 4.10, we introduced the idea of a short definition. It is often the case that a new `model` is so similar to something that already exists that its definition looks something like:

```
model MyRLC
  extends RLC5(R1(R=12));
end MyRLC;
```

Note that this model involves a slight modification of an existing model (*i.e.*, it does not introduce any new subcomponents). In such cases, it is possible to use the slightly less complicated short definition approach. The short definition would be:

```
model MyRLC=RLC5(R1(R=12));
```

This second form allows you to essentially replace the `extends` keyword with an equal sign and the `end` statement with a simple semi-colon;

We have shown how to create a short definition for a `model`. A similar approach can be taken when defining a `connector`, `block` or `record` (described in Chapter 5). Note that a `type` is always defined using a short definition, *e.g.*,

```
type Pressure=Real(quantity="Pressure", unit="N/m^2");
```

4.8.3 Concept of subtype

In Section 4.4, we talked briefly about the notion of a subtype relationship. Since subtypes are important for several features presented in this chapter, a more rigorous discussion of the subtype concept is included.

4.8.3.1 Theory

Imagine two models, `A` and `B`. Roughly speaking, model `B` is a subtype of model `A` if `B` contains all of the same components (with the same names) as `A`. To illustrate this, consider the following model:

```
model BaseModel
  Real x, y;
end BaseModel;
```

Now assume we have another model like this one:

```
model DerivedModel
  extends BaseModel;
  Real z;
end DerivedModel;
```

As was described in Section 4.8.1, this definition of `DerivedModel` is equivalent to:

```
model DerivedModel
  Real x, y, z;
end DerivedModel;
```

In either case, the `DerivedModel` is a subtype of the `BaseModel` because it has all of the components of `BaseModel` (*i.e.*, `x` and `y`). It does not matter that `DerivedModel` has more components (*i.e.*, `z`) than `BaseModel`. It also does not matter whether the definition of `DerivedModel` uses the `extends` keyword or whether it contains its own declarations of `x` and `y`, with the appropriate types, as long as it has all of the components of `BaseModel`.

Subtype relationships are not limited to models. Such relationships apply for all definitions in Modelica (*e.g.*, `connector` definitions).

4.8.4 Creating partial definitions

In Example 4.1, the `model` definition was preceded by the `partial` keyword. The `partial` keyword indicates that while this model can be extended, it cannot be instantiated. In other words, the model is a foundation on which to build new models, but it is not a proper model by itself (usually because it is missing some constitutive equations). The `partial` keyword, like many features in Modelica, not only enforces certain semantics but also documents the intent of the original model developer. In other words, if you see a `partial` model,

you know immediately that this is not something you would declare but rather something to extend from.

4.8.5 Making elements of a model replaceable

4.8.5.1 Replaceable subcomponents

Example 4.8 showed how we can declare a component to be `replaceable`. There are two differences between a normal declaration and a replaceable declaration. First, the replaceable declaration is preceded by the `replaceable` keyword. In addition, a replaceable declaration can be followed by an optional `extends` clause which indicates the constraining type of the declaration. The constraining type limits what a component can be replaced with, which provides some degree of robustness. So in Example 4.8, the `extends` clause indicates that in all cases `con` must be a subtype of `Controller`. Of course, if the component is never redeclared, its type will be whatever was specified in the original declaration.

4.8.5.2 Using replaceable type definitions

Replaceable type definitions are really not significantly different than replaceable component declarations. The `replaceable` keyword is placed in front of a definition (*e.g.*, a `model` definition in Example 4.10) and an optional `extends` can be added to the end followed by the constraint type. The only difference, when compared to replaceable components, is that a replaceable type definition can be used to change the types of numerous components simultaneously rather than one at a time. Replaceable type definitions usually involve the definition of a local type (*i.e.*, a type which is only used within the context of a specific model).

4.8.6 Making components "final"

Flexibility sometimes comes at the cost of robustness. The `final` keyword is used to restrict flexibility by disallowing further changes. As was shown in Section 4.6, there are times when `final` can be used to eliminate the possibility of inappropriate modification. The `final` keyword can be placed in front of modifications to imply that those modifications cannot be subsequently changed. Modifications include the assignment of values to constants and parameters and also any redeclarations made using the `redeclare` keyword (as shown in the `MyRLC` model on page 82).

4.9 PROBLEMS

PROBLEM 4.1 *Write models for a `Capacitor` and an `Inductor` by extending the `OnePort` model shown in Example 4.1.*

PROBLEM 4.2 *The* `Integrator`, `TransferFunction` *and* `Gain` *blocks (found in Examples 3.13, 3.14 and 3.15, respectively) all have a single input and a single output. Create a base model that all of these models can be extended from (i.e., something analogous to the* `OnePort` *model in the previous problem).*

PROBLEM 4.3 *Create a version of the circuit shown in Example 3.7 (found in page 46) that uses the* MSL. *Furthermore, make the resistor, capacitor and inductor replaceable with any component that satisfies the* `OnePort` *interface from Example 4.1. What happens if you replace the capacitor with a resistor that has very small resistance? Compare the voltage across the inductor in both cases.*

PROBLEM 4.4 *Create a resistor model like the one described in Problem 3.5 that is a subtype of the* `OnePort` *model in Example 4.1. Redeclare the resistor model in the solution to Problem 4.3 as an instance of this new resistor model. Then, run a simulation using the new resistor model and compare the results to the results obtained using an ideal resistor.*

PROBLEM 4.5 *Create a version of the* `ControllerAndMotor` *model shown in Example 4.8 where the plant model is also replaceable. Use the solution from Problem 4.2 as the base model for all plant models. Then, create a plant model from the* `RotationalPendulum` *model shown in Example 3.19 and use that as the plant. You will need to connect a torque source as input and speed sensor as an output (both of which can be found in* `Modelica.Mechanics.Rotational`*).*

You will probably want to change the offset value for the signal generator, `sinsig`*, to zero. In addition, start the pendulum in a non-equilibrium state. Then, try changing the controller parameters to improve the overall performance.*

Chapter 5

FUNCTIONS

5.1 CONCEPTS

While writing equations (*i.e.,* equating two expressions) is sufficient for most modeling, there are cases where a procedural or algorithmic approach, involving explicit assignment, is necessary. To address this need, Modelica includes support for algorithmic functions. While a `function` in Modelica is like a `block` because all quantities must be explicitly labeled as either `input` or `output`, it is different from a `block` or `model` because it is not connected to other components. Instead, it is invoked during the evaluation of expressions. Another difference between a `function` and a `block` or `model` is that a `function` is not allowed to have any persistent internal state. As a result, there are several restrictions on the statements which can appear within a `function` (*e.g.,* the `der` operator cannot appear within a `function`).

In this chapter, we will describe an alternative to an `equation` section called an `algorithm` section. The `algorithm` section is used when procedural semantics are required. While a `block` or `model` definition can contain any number of `equation` or `algorithm` sections, a `function` definition must contain exactly one `algorithm` section which performs all calculations for that `function`.

In this chapter, we will show several examples of Modelica functions. In addition, we will show how to call external subroutines[1] (written in languages such as C or FORTRAN77) from within a Modelica model.

[1]To avoid confusion, the term *subroutine* will refer to C or FORTRAN77 code and *function* will refer to Modelica functions.

5.2 INTRODUCTION TO FUNCTIONS

Let us start with a simple function. Imagine we wish to search through an array of names for a particular name and find the index of that name in the array. The two arguments to the function will be an array of names and the name we are looking for. The output will be the index where the name was found or an error will occur if the name was not found. Each name is represented as a `String` in Modelica. The `input` components of the `function` correspond to the arguments of the `function` when it is invoked and the `output` component corresponds to the value of the `function` when used in an expression.

```
function FindName
  input String names[:];
  input String name_to_find;
  output Integer index;
protected
  Integer i, len=size(names,1);
algorithm
  index := -1;
  i := 1;
  while index==-1 and i<=len loop
    if names[i]==name_to_find then
      index := i;
    end if;
    i := i+1;
  end while;
  assert(index<>-1, "FindName: failed");
end FindName;
```

Example 5.1. A function to find a name in an array of names.

5.2.1 Arrays

This example is the first to contain the declaration of an array. We do not describe arrays in detail until Chapter 6. However, we need to cover a few basics in order to explain this example. First, the presence of the "`[`" and "`]`" characters in the declaration of `names` indicates that `names` is an array. In the declaration, the size for each dimension of the array appears as a comma separated list between the "`[`" and "`]`" characters. The use of "`:`" in such a declaration indicates that any size for that dimension is allowed. In this case, we can determine from the declaration that `names` is a one-dimensional array of an unspecified size. We can also see, from the `if` statement, that the "`[`" and "`]`" characters are also used in expressions to reference individual elements of the array.

5.2.2 Robustness

This `function` demonstrates several features that promote the development of robust functions. For example, note that the declaration of the `names` argument does not include a size for the array. This indicates that this `function` is defined so that it can handle any one-dimensional array of names. However, within the `function` we must know how many names are in the array. We could add an additional argument to the `function` to allow the size of the array to be passed into the function. However, the risk still remains that an incorrect size could be passed in. A more robust way to determine the size of the `names` array is to use the `size()` function. The `size()` function will always return the correct size of an array. More details on `size()` and other array related functions can be found in Chapter 6.

Another thing to notice about the `FindName` function is the use of the `assert()` function. We can use the `assert()` function to verify certain conditions. For example, if the `index` variable is unchanged after the `while` loop, then we failed to find the name we were looking for. In other words, at the end of this `function` we would like to make sure that the `index` variable has not kept its initial value of -1. We can do this by using the `assert()` function to verify that `index` is not equal to -1 at this point. If the assertion is false (*i.e.*, `index` is equal to -1), then the message contained in the second argument to the `assert()` function will be displayed to the user and the simulation will stop.

5.2.3 Function contents

Example 5.1 may be simple, but it introduces many new ideas. First, it contains a `protected` section. Within a `function`, the `protected`[2] section contains declarations for all *local variables*. The local variables are only visible within the `function`. They are created during each invocation of the function and destroyed when the function invocation is completed. Such variables are typically used as temporary variables in calculations performed internally to the function.

Note that this function uses the `:=` operator. Recall, from Chapter 2 that the `:=` operator indicates assignment rather than equality. The variable being assigned to must appear on the left hand side of the `:=` operator. We can see that the `:=` operator is used extensively in this function. Finally, we see the use of a `while` loop. In this example, we use the `while` statement to repeatedly check whether we have found the string we are looking for.

[2]The use of the `protected` keyword within models will be discussed in Chapter 6.

5.2.4 Invoking a function

```
model TestFindName
  parameter String names[:] = {"H2O","CO2","N2"};
  parameter Integer CO2=FindName(names,"CO2");
end TestFindName;
```

Example 5.2. Invoking the `FindName` function.

Example 5.2 shows an example of a model which invokes the `FindName` function. Note that the arguments to the function must be passed according to the order they were declared in the function definition (*i.e.*, `names` followed by `name_to_find`). Even though the `FindName` function is used in Example 5.2 to initialize a parameter, we have seen previously (*e.g.*, Example 2.2) that functions can be used in equations as well.

5.3 AN INTERPOLATION FUNCTION

Let us move on to a more complex example. Since it is common to require interpolation when modeling, the next example is a function which can perform linear interpolation for us. To keep things simple, we restrict this example, shown in Example 5.3, to the case of one dimensional linear interpolation. Furthermore, an assertion will fail if the value of the independent variable, `x`, is outside the range of values found in `x_grid`.

```
function Piecewise "A piecewise linear interpolation"
  input Real x "Independent variable";
  input Real x_grid[:] "Independent variable data points";
  input Real y_grid[:] "Dependent variable data points";
  output Real y "Interpolated result";
protected
  Integer n;
algorithm
  n := size(x_grid,1);
  assert(size(x_grid,1)==size(y_grid,1), "Size mismatch");
  assert(x>=x_grid[1] and x<=x_grid[n], "Out of range");
  for i in 1:n-1 loop
       if x>=x_grid[i] and x<=x_grid[i+1] then
         y := y_grid[i]+(y_grid[i+1]-y_grid[i])*
              ((x-x_grid[i])/(x_grid[i+1]-x_grid[i]));
       end if;
  end for;
end Piecewise;
```

Example 5.3. A piece-wise linear function.

5.3.1 Explanation

Once again, we see the `size()` function being used to determine the size of the `input` arrays. In addition, we see several uses of the `assert()` function. These assertions make sure that the `input` arrays, `x_grid` and `y_grid`, are the same size and that the independent variable, `x`, is within the range given by the `x_grid` argument.

Linear interpolation involves finding the location, in the `x_grid` array, of the data points immediately above and below the independent variable, `x`. Once these have been located, the output of the function, `y`, is computed by linearly interpolating between the values in the `y_grid` which correspond to the adjacent data points.

5.3.2 Using for loops

Another interesting thing about this example is that it contains a `for` loop. The expression `1:n-1` is actually short hand for a complete array which includes elements `(1,...,n-1)`. As an argument to `for`, it provides the range of values for the variable `i`. It should be pointed out that `i` is a variable which is local to the `for` loop. This means that it is created at the start of the `for` loop and disappears after the "`end for;`" statement is reached. For this reason, there is no need to declare `i` as a variable.

5.3.3 Named arguments

In Modelica, there are two ways to invoke functions. The first is to pass the arguments in the order they are declared in the `function` definition. This is what we did in Example 5.2. Let us test the `Piecewise function` using the alternative way of invoking a function:

```
model TestPiecewise
  package SI=Modelica.SIunits;
  parameter SI.Time x_vals[6] = {0, 2, 4, 6, 8, 10};
  parameter Real y_vals[6] = {0, 0, 4, 16, 36, 64};
  Real y;
equation
  y = Piecewise(x=time,x_grid=x_vals,y_grid=y_vals);
end TestPiecewise;
```

In this case, we have provided an explicit equation for each `input` argument when invoking the `function`. Invoking functions in this way is legal in Modelica so long as an equation is provided for each argument. Figure 5.1 shows the values for y generated by this code fragment.

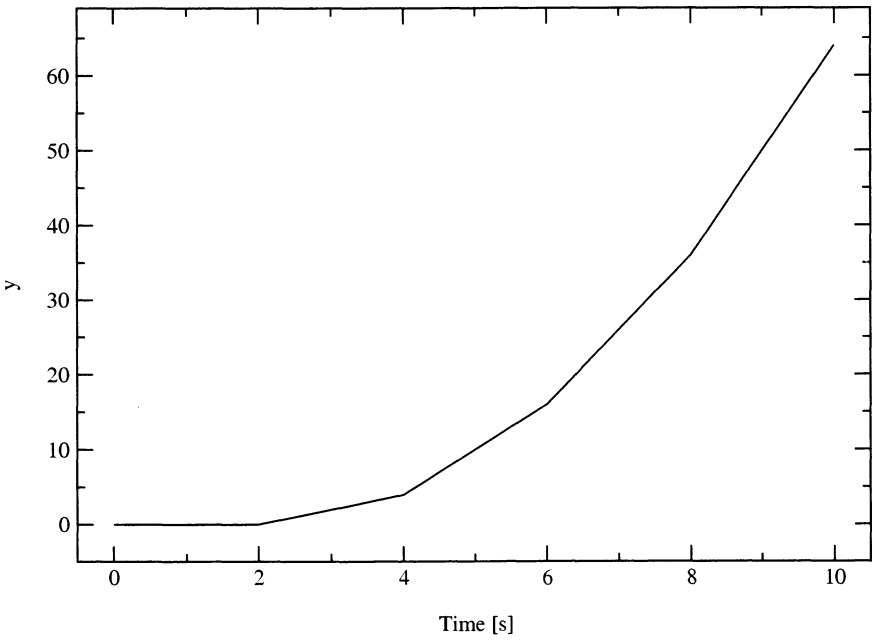

Figure 5.1. Output after simulating `TestPiecewise` for 10 seconds.

5.4 MULTIPLE RETURN VALUES

For our next example, we consider a function which has more than one return value. Imagine we need to evaluate a polynomial in `x` and the derivative of that polynomial with respect to `x`. Given an array containing the coefficients of the polynomial, Example 5.4 shows a function which uses a single loop to compute both the value of the polynomial and its derivative.

Once again, we use the `size()` function to determine the size of the arrays being passed in. Using this size information, a `for` loop can be constructed to evaluate the polynomial and its derivative. The following model can be used to validate the calculation of the derivative:

```
model TestPolyEval
  parameter Real coefs[3] = {2.0, 1.0, 2.0};
  Real y;
  Real fdy;
  Real dy;
equation
  (y,fdy) = PolyEval(time,coefs);
  dy = der(y);
end TestPolyEval;
```

We can validate the `PolyEval` function by comparing `fdy` and `dy`.

```
function PolyEval "Evaluate polynomial and derivative"
  input Real x "Independent variable";
  input Real coef[:] "Coefficients (low to high order)";
  output Real y "Result of polynomial evaluation";
  output Real dydx "Derivative of polynomial";
protected
  Integer n;
algorithm
  n := size(coef,1);
  y := coef[n];
  dydx := 0.0;
  for i in n-1:-1:1 loop
    y := y*x + coef[i];
    dydx := dydx*x + i*coef[i+1];
  end for;
end PolyEval;
```

Example 5.4. Evaluation of a polynomial and its derivative.

5.5 PASSING RECORDS AS ARGUMENTS

For complex functions, passing in large numbers of arguments can become cumbersome. In these cases, it is useful to define a `record` type which can be used to group several logically related arguments together.[3] Imagine we wish to create a function which evaluates the sum of several sine waves, *i.e.*,

$$\sum_i A_i \sin\left(2\pi x f_i + \phi_i\right) \tag{5.1}$$

where each wave has its own amplitude, A_i, frequency, f_i and phase shift, ϕ_i. Example 5.5 shows how we might write such a function.

Note that Example 5.5 includes a local `record` definition. Local definitions are useful because they are clearly associated with a specific model or function (*e.g.*, `ComplexWave`). There is no chance that this `Data record` could be confused with another `record` also named `Data` because the definition is nested within the `ComplexWave` function and therefore a qualified name (*i.e.*, `ComplexWave.Data`), must be used in any declarations of the `Data record` outside the `ComplexWave` function.

5.5.1 Building a record

The following code fragment gives an idea how the `ComplexWave` function in Example 5.5 could be used:

[3] A **record** in Modelica is similar to a `struct` in C.

```
function ComplexWave
  record Data
    constant Integer num "Number of waves";
    Real a[num] "Wave amplitudes";
    Modelica.SIunits.Frequency f[num] "Wave frequencies";
    Modelica.SIunits.Angle phase[num] "Wave phase offset";
  end Data;

  input Real x "Independent variable";
  input Data d "Wave data";
  output Real y "Sum of sine waves";
protected
  Integer n;
  Real s;
algorithm
  n := d.num;
  y := 0;
  for i in 1:n loop
       s := Modelica.Math.sin(
            2*Modelica.Constants.pi*d.f[i]*x+d.phase[i]);
    y := y + d.a[i]*s;
  end for;
end ComplexWave;
```

Example 5.5. Calculating the sum of a series of sine waves.

```
model TestComplexWave
  parameter ComplexWave.Data wdata(num=3,
            a={1.3, 2.2, 5.8},
            f={2.0, 3.0, 7.0},
            phase={0, Modelica.Constants.pi, 0});
  Real signal;
equation
  signal = ComplexWave(time,wdata);
end TestComplexWave;
```

Remember, because `Data` is defined within the **function** `ComplexWave`, we reference that **record** definition using the qualified name `ComplexWave.Data`. We then provide the necessary data for each of the waves and invoke the function. In this way, we have reduced the number of arguments to `ComplexWave` from four to two. The data provided in this case should cause the `ComplexWave` function to evaluate the following expression:

$$y(x) = 1.3\sin(4\pi x) + 2.2\sin(6\pi x + \pi) + 5.8\sin(14\pi x) \qquad (5.2)$$

Figure 5.2 shows the results of simulating the `TestComplexWave` model.

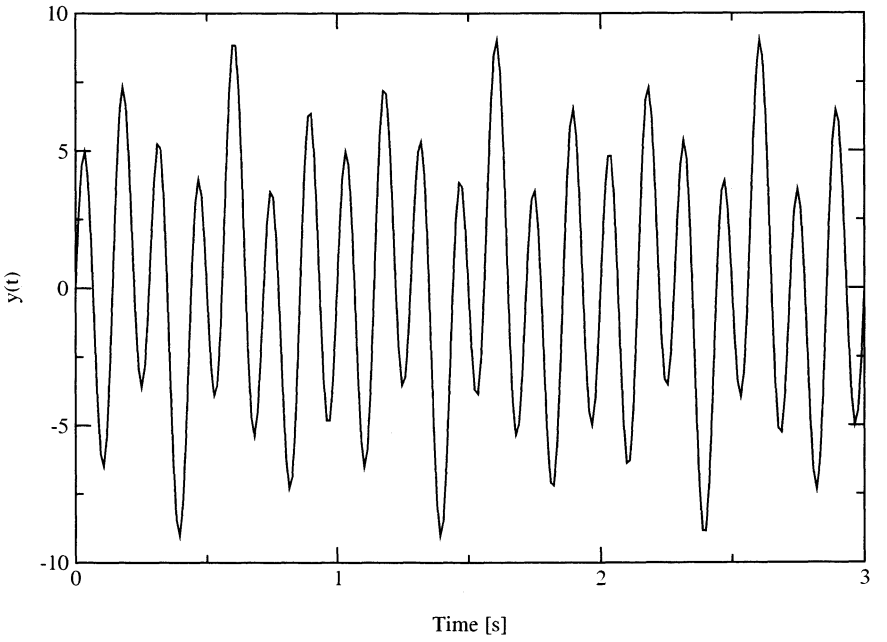

Figure 5.2. Simulation results for `TestComplexWave`.

5.5.2 Variables within a record

A `record` does not have to be a `parameter` as it is in the `TestComplexWave` model. In some cases, it may be useful to declare a `record` which contains time varying quantities. The following code fragment shows how this can be accomplished.

```
model TestComplexWave2
  ComplexWave.Data wdata(num=3); // Not a parameter
  Real signal;
equation
  wdata.a = {1.3, 2.2, 5.8*Modelica.Math.exp(-.54*time)};
  wdata.f = {2.0, 3.0, 7.0};
  wdata.phase = {0, Modelica.Constants.pi, 0};
  signal = ComplexWave(time,wdata);
end TestComplexWave2;
```

For this example, we are using essentially the same data as we did in the `TestComplexWave` model except we allow the last term to diminish with time. Mathematically, this should lead to the evaluation of the following expression:

$$y(x) = 1.3\sin(4\pi x) + 2.2\sin(6\pi x + \pi) + 5.8e^{-.54x}\sin(14\pi x) \qquad (5.3)$$

The results of simulating the `TestComplexWave2` model are shown in Figure 5.3. The results also include evaluations of the following two equations:

$$A(x) = 1.3\sin(4\pi x) + 2.2\sin(6\pi x + \pi) + 5.8\sin(14\pi x) \quad (5.4)$$

$$B(x) = 1.3\sin(4\pi x) + 2.2\sin(6\pi x + \pi) \quad (5.5)$$

As we would expect, the results show that the value of the `signal` variable initially follows the function $A(x)$ but gradually moves closer to $B(x)$ as the contribution of the last term diminishes.

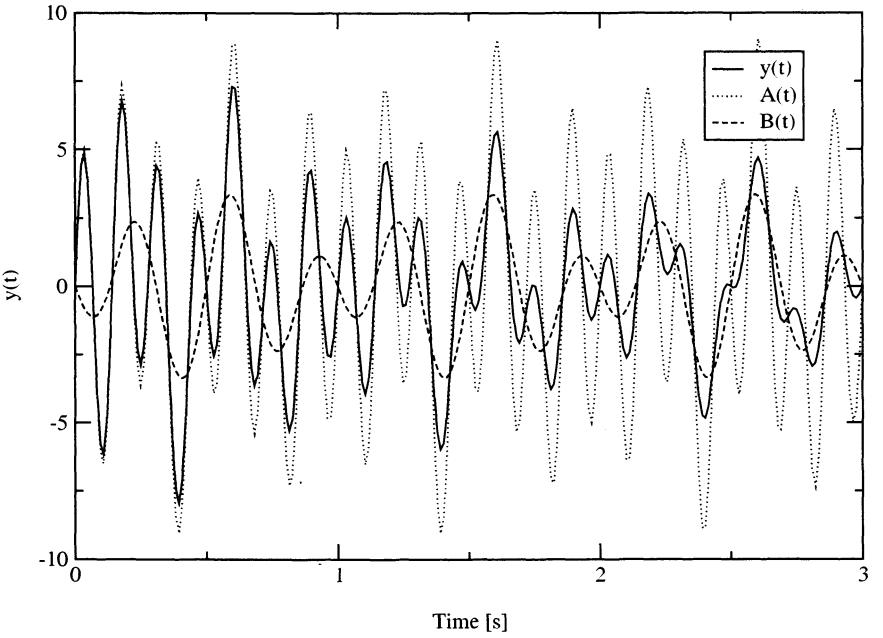

Figure 5.3. Simulation results for `TestComplexWave2`.

5.6 USING EXTERNAL SUBROUTINES

We have demonstrated how a variety of functions can be implemented in Modelica. While it is easy to write Modelica functions, it is sometimes convenient to call a subroutine written in C or FORTRAN77. In this section we will show an example of how this is done.

5.6.1 External subroutines

A common requirement for thermodynamic models is to compute properties (*e.g.*, enthalpy) of a working fluid (*e.g.*, water) for a given pressure

and temperature. Imagine we have an existing external C subroutine named `compute_enthalpy` that takes pressure and temperature as an input and computes the enthalpy of our working fluid as an output. Rather than rewrite such a subroutine as a Modelica `function`, let us look at how we can call the external subroutine directly from a Modelica model.

In order to use an existing external subroutine, we must first write a "wrapper" `function` in Modelica before we can call the external subroutine. Example 5.6 shows how we would write a wrapper function for the `compute_enthalpy` subroutine.

```
function Enthalpy
  input Modelica.SIunits.Pressure P;
  input Modelica.SIunits.Temp_K T;
  output Modelica.SIunits.Enthalpy h;

  external "C" compute_enthalpy(P,T,h);
end Enthalpy;
```

Example 5.6. A Modelica wrapper function for a C subroutine.

The subroutine `compute_enthalpy` used in Example 5.6 should have a C function prototype that looks something like:

```
void compute_enthalpy(double P, double T, double *h);
```

The detail of how the C function prototypes are defined will be covered shortly in Section 5.7.8.

5.6.2 Language specification

Enthalpy is not the only property we would typically need. In fact, many similar properties are often required. Properties of working fluids are often tabulated in what are called "steam tables". Example 5.7 shows how we could write a `function` which calls an `external` subroutine that returns several properties at once. In this example, we have assumed the subroutine is written in FORTRAN77. [4] The FORTRAN77 code for the `calcprops` subroutine from Example 5.7 would be similar to:

```
SUBROUTINE CALCPROPS(PRES, TEMP, H, U, CP, RHO)
DOUBLE PRECISION PRES, TEMP, H, U, CP, RHO
...
END
```

[4]The specification of any language besides C is required because subroutines differ in many ways from one language to another. Issues such as argument ordering, pass-by-value vs. pass-by-reference and name mangling are a few aspects of subroutines that depend on what language the subroutine was written in.

```
function SteamTable
  input Modelica.SIunits.Pressure P;
  input Modelica.SIunits.Temp_K T;
  output Modelica.SIunits.SpecificEnthalpy h;
  output Modelica.SIunits.SpecificEnergy u;
  output Modelica.SIunits.SpecificHeatCapacity cp;
  output Modelica.SIunits.Density rho;

  external "FORTRAN 77" calcprops(P,T,h,u,cp,rho);
end SteamTable;
```

Example 5.7. A Modelica wrapper function for a FORTRAN77 subroutine.

5.6.3 Invoking external subroutines

External subroutines are invoked just like any other Modelica `function`. In order to invoke the function, the simulation tool requires access to the subroutine. Typically, only a compiled version of the subroutine would be required and not the source code. How the subroutine is accessed (*e.g.*, compiled or linked) is a tool specific issue not covered by the language specification.

5.7 LANGUAGE FUNDAMENTALS

5.7.1 Arguments

As we have seen in this chapter, the arguments to a function are defined by the `input` components in the `public` section of the `function` definition. Any component preceded by the `input` qualifier represents a quantity being passed into the `function`. Likewise, any component preceded by the `output` qualifier represents a quantity being returned by the `function`. All components in the `public` section must be labeled as `input` or `output`.

As we saw in Example 5.5, when the number of arguments starts getting large it is useful to pass information into a `function` as a `record`. This reduces the number of arguments (and confusion about argument order).

5.7.2 Local variables

Any quantities which are calculated from the `input` variables, but are not `output` variables, are called local variables. As we saw in Example 5.1, such variables must be declared in the `protected` section of the `function` definition. It is important to keep in mind that the values of these local variables are **not** stored between `function` invocations. In other words, if you assign a value to a variable during one invocation of the `function`, you cannot expect it to still have that value at the next invocation.

5.7.3 Algorithmic semantics

The main purpose of a `function` is to perform algorithmic calculations. These calculations often involve looping and conditional statements and appear within an `algorithm` section.

The most important thing to remember about an `algorithm` section is that it is possible to assign to the same variable multiple times. In each case, the new assignment will replace the value from any previous assignments. To understand the significance of this, consider the following code fragment:

```
algorithm
  x := y;
  x := z;
```

In this case, only the last assignment, `x := z`, is important. This is in contrast to an `equation` section where multiple equality relationships represent multiple equations, *e.g.*,

```
equation
  x = y;
  x = z;
```

Both of these equations are significant and lead to the implication $x = y = z$.

There are two ways to tell the difference between assignments and equations. First, assignment involves the `:=` operator while equations use the `=` operator. Second, an assignment must appear within an `algorithm` section and an equation must appear within an `equation` section. This helps to avoid any confusion about whether a statement is an assignment or an equation.

Essentially, what this all means is that a `function` in Modelica behaves almost exactly like a subroutine in C or FORTRAN77 where variables can be assigned and reassigned values.

5.7.4 Branching

Examples 5.1 and 5.3 both contain `if` statements. An `if` statement can also include an `else` clause as well as several `elseif` clauses. For example:

```
if x>=0 then
  y := x;
elseif x<=-3 then
  y := -6;
else
  y := -2*x;
end if;
```

5.7.5 Looping in algorithms

Looping is used to implement algorithms that require iteration (*e.g.*, the `y` and `dydx` variables used by the `PolyEval` function in Example 5.4). There are two kinds of loops. A **`while`** loop is one where operations are performed repeatedly while some condition remains satisfied. Generally, a **`while`** loop is preceded by some initialization statements. The `FindName` model in Example 5.1 shows how a **`while`** loop can be used. A **`while`** statement has the general form:

```
// initialization (if required)
while (someCondition) loop
  // do something
end while;
```

After any initialization statements, the statements inside the **`while`** loop are evaluated repeatedly while the boolean expression `someCondition` remains **`true`**.

The **`for`** statement in Modelica is convenient for looping over the contents of vectors [5] and is similar to the "foreach" construct in languages such as Perl and Tcl. The general form of the **`for`** statement is:

```
for someVar in someVector loop
  // do something (presumably involving someVar)
end for;
```

This statement can be interpreted as: "Evaluate the statements inside this loop with `someVar` successively set to each value contained within the vector `someVector`". An important point here is that it is only possible to loop over vectors (*i.e.*, one-dimensional arrays).

In Example 5.5, the expression `1:n` evaluates to a vector of all integers between `1` and `n`. In Example 5.4, the expression `n-1:-1:1` evaluates to a vector starting with `n-1` and counting down to `1` by intervals of `-1`. This kind of vector shorthand is discussed in detail in Section 6.5.2.1.

5.7.6 Invoking a function

If a **`function`** is invoked by providing each argument in the form of an equation (as we saw in `TestPiecewise`), the arguments may appear in any order. If an equation is provided for one argument, then an equation must be present for all arguments. On the other hand, if the function invocation does not include equations for the arguments but simply a collection of values then the order of the components in the **`function`** definition determines the required order of the arguments in the invocation.

[5] Note that the use of **`for`** in Modelica is different from C and C++.

Now let us examine how to use the return value of the `function`. If a `function` has a single return value (*i.e.*, a single `output` variable in its definition as in Example 5.3), then it may be used in expressions such as:

```
y = x*Piecewise(x=time,x_grid=x_vals,y_grid=y_vals)+
    z*Piecewise(x=time,x_grid=z_vals,y_grid=y_vals);
```

On the other hand, if there are multiple return values (as in Example 5.7), the `function` invocation can only be used in an equation or assignment and it must form the complete right hand side. Furthermore, the left hand side should be a comma separated list of variables enclosed in parentheses. The following is a legal example of invoking a `function` with multiple return values:

```
(h, u, cp, rho) = SteamTable(P,T);
```

On the other hand, this is not a legal invocation:

```
(h, U/m, cp, m/V) = SteamTable(P,T);
```

because only variables (*i.e.*, no expressions) may appear on the left hand side.

5.7.7 Built-in functions

Modelica provides a collection of built-in functions. In this section, we will discuss some of the built-in functions and the remainder will be discussed in Chapter 6 because they involve array operations.[6]

5.7.7.1 Analysis type

The `analysisType()` function is used to give the model a chance to customize its behavior to different types of analyses. The `analysisType()` returns a string to indicate the type of analysis currently being performed. The possible return values may include, but are not limited to, the ones shown in Table 5.1.

Type	Meaning
`"dynamic"`	Evaluating transient response.
`"static"`	Determining steady state response.
`"linear"`	Linear analysis (*e.g.*, analyzing frequency response).

Table 5.1. Example analysis types.

[6]Chapter 7 reviews many of the built-in functions and explains their effects in the context of hybrid behavior.

5.7.7.2 Absolute value

The `abs()` function takes a single argument, x, and computes the absolute value of x. The argument type can be either `Real` or `Integer` and the return type is the same as the argument type. Mathematically, the function is defined as:

$$abs(x) = \begin{cases} -x & : \quad x < 0 \\ x & : \quad x \geq 0 \end{cases} \tag{5.6}$$

5.7.7.3 Sign

The `sign()` function takes a single argument, x, and returns an indication whether x is negative or positive. The type of x can be either `Real` or `Integer` but the return value is always an `Integer`. The `sign()` function is defined mathematically as:

$$sign(x) = \begin{cases} -1 & : \quad x < 0 \\ 0 & : \quad x = 0 \\ 1 & : \quad x > 0 \end{cases} \tag{5.7}$$

5.7.7.4 Square root

The `sqrt()` function takes a single argument, x, and returns the square root of x. The type of x can be either `Real` or `Integer` but the return value of `sqrt()` is always a `Real`. The value of x must be greater than or equal to zero or an error will occur.

5.7.7.5 Ceiling and floor function

The `ceil()` function takes a single argument, x, and returns the smallest integer not less than x. Likewise, the `floor()` function takes a single argument, x, and returns the largest integer not greater than x. An important thing to realize about these functions is that while the return value is an integer in the mathematical sense, it is not an `Integer` in the Modelica sense. Instead, the argument and return type for both `ceil()` and `floor()` is `Real`. Examples of using these functions include:

$$\begin{array}{lcl} \text{ceil}(3.2) & \rightarrow & 4.0 \\ \text{floor}(3.2) & \rightarrow & 3.0 \\ \text{ceil}(-3.2) & \rightarrow & -3.0 \\ \text{floor}(-3.2) & \rightarrow & -4.0 \end{array} \tag{5.8}$$

5.7.7.6 Truncation

The `integer()` function, just like the `floor()` function, takes a single argument, x and returns the largest integer not greater than x. The difference

is that while `floor()` returns a `Real`, `integer()` returns an `Integer`. Examples of using the `integer()` function include:

$$\begin{array}{lcl} \text{integer}(3.2) & \rightarrow & 3 \\ \text{integer}(-3.2) & \rightarrow & -4 \end{array} \tag{5.9}$$

5.7.7.7 Division

The `div()` function takes two arguments, x and y, and returns the algebraic quotient of x/y with any fractional part discarded (*i.e.*, truncation toward zero). The arguments may be of type `Real` or `Integer`. If either of the arguments is a `Real`, the result is a `Real` otherwise the result is an `Integer`. Examples of using the `div()` function include:

$$\begin{array}{lcl} \text{div}(3.2, 1.2) & \rightarrow & 2.0 \\ \text{div}(-3.2, 1.2) & \rightarrow & -2.0 \\ \text{div}(3.2, -1.2) & \rightarrow & -2.0 \\ \text{div}(-3.2, -1.2) & \rightarrow & 2.0 \\ \text{div}(7, 2) & \rightarrow & 3 \\ \text{div}(-7, 2) & \rightarrow & -3 \\ \text{div}(7, -2) & \rightarrow & -3 \\ \text{div}(-7, -2) & \rightarrow & 3 \end{array} \tag{5.10}$$

5.7.7.8 Remainder

The `rem()` function takes two arguments, x and y, and returns the remainder discarded by the `div()` function. This can be expressed mathematically as:

$$rem(x, y) = x - div(x, y) * y \tag{5.11}$$

The arguments may be of type `Real` or `Integer`. If either of the arguments is a `Real`, the result is a `Real` otherwise the result is an `Integer`. Examples of using the `rem()` function include:

$$\begin{array}{lcl} \text{rem}(3.2, 1.2) & \rightarrow & 0.8 \\ \text{rem}(-3.2, 1.2) & \rightarrow & -0.8 \\ \text{rem}(3.2, -1.2) & \rightarrow & 0.8 \\ \text{rem}(-3.2, -1.2) & \rightarrow & -0.8 \end{array} \tag{5.12}$$

5.7.7.9 Modulo

The `mod()` function takes two arguments, x and y, and returns the modulus of x and y, *i.e.*,

$$mod(x, y) = x - floor(x/y) * y \tag{5.13}$$

The arguments may be of type `Real` or `Integer`. If either of the arguments is a `Real`, the result is a `Real` otherwise the result is an `Integer`. Examples

of using the `mod()` function include:

$$\begin{array}{lcl} \mathrm{mod}(3.2, 1.2) & \rightarrow & 0.8 \\ \mathrm{mod}(-3.2, 1.2) & \rightarrow & 0.4 \\ \mathrm{mod}(3.2, -1.2) & \rightarrow & -0.4 \\ \mathrm{mod}(-3.2, -1.2) & \rightarrow & -0.8 \end{array} \tag{5.14}$$

5.7.8 External subroutines

As discussed in Section 5.6, it is often desirable to use existing subroutines written in C or FORTRAN77. Let us review the details that were not covered by the examples.

5.7.8.1 Type matching

Table 5.2 shows the Modelica built-in types and their corresponding C type.[7] This is why the C function prototype for Example 5.6 was:

```
void compute_enthalpy(double P, double T, double *h);
```

If Example 5.7 were written in C, its function prototype would be:

```
void calcprops(double P, double T, double *h, double *u,
               double *cp, double *rho);
```

Modelica type	C type (inputs)	C type (outputs)
`Real`	`double`	`double *`
`Integer`	`int`	`int *`
`Boolean`	`int`	`int *`
`String`	`const char *`	N/A (input only)
`Real`$[dim_1, \cdots, dim_n]$	`double *, size_t` $\mathbf{dim_1}$`, ..., size_t` $\mathbf{dim_n}$	
`Integer`$[dim_1, \cdots, dim_n]$	`int *, size_t` $\mathbf{dim_1}$`, ..., size_t` $\mathbf{dim_n}$	
`Boolean`$[dim_1, \cdots, dim_n]$	`int *, size_t` $\mathbf{dim_1}$`, ..., size_t` $\mathbf{dim_n}$	

Table 5.2. Modelica types ↔ C types.

When invoking a C language subroutine, it is possible to pass a `record` to the external subroutine. When passing a record, it is important to keep several things in mind. First, when a Modelica `record` is passed into a C language subroutine it appears in the C subroutine as a pointer to a structure. The structure definition should include the same components as the Modelica `record` in the same order and using the type mapping shown in Table 5.2. So, the following `record` definition:

[7] The C type `size_t` used in Table 5.2 is defined in header file `stddef.h`.

```
record RecDef
  Real a[5];
  Integer b[10];
  Real c;
end RecDef;
```

would correspond to the following C structure definition:

```
struct RecDef {
  double a[5];
  int b[10];
  double c;
};
```

Note that Modelica does not support the passing of records containing variable sized arrays. If that is an issue, it is better to pass the record as individual arguments.

In addition to the C language, the Modelica language specification also provides for the possibility that the external subroutine is written in FORTRAN77. Table 5.3 shows the mapping between Modelica built-in types and FORTRAN77 types.[8] This mapping was used to create the subroutine header shown in Section 5.6.2.

Modelica type	FORTRAN77 type
`Real`	`DOUBLE PRECISION`
`Integer`	`INTEGER`
`Boolean`	`LOGICAL`
`Real`$[dim_1, \cdots, dim_n]$	`DOUBLE PRECISION, INTEGER` $\texttt{DIM}_1$`, ..., INTEGER` $\texttt{DIM}_N$
`Integer`$[dim_1, \cdots, dim_n]$	`INTEGER, INTEGER` $\texttt{DIM}_1$`, ..., INTEGER` $\texttt{DIM}_N$
`Boolean`$[dim_1, \cdots, dim_n]$	`LOGICAL, INTEGER` $\texttt{DIM}_1$`, ..., INTEGER` $\texttt{DIM}_N$

Table 5.3. Modelica types ↔ FORTRAN77 types.

5.7.8.2 Custom subroutine invocation

In our examples, we have seen one way that external subroutines can be invoked. In all cases, the language and the order of arguments was specified explicitly. It is recommended that external subroutines be invoked in this way whenever possible to avoid any potential confusion.

However, in some circumstances it may be necessary to customize the handling of return values. This case comes about primarily when a C subroutine

[8] Note, there is no mapping to FORTRAN77 for the `String` type in Modelica.

already exists that returns its value (in the C sense) rather than assigning to a variable which was passed by reference. For example, let us imagine that the `compute_enthalpy` subroutine discussed in Section 5.6 and used in Example 5.6 had a prototype which looked like:

```
double compute_enthalpy(double P, double T);
```

and the return value of the subroutine was the enthalpy. To use this subroutine directly, we could substitute the `external` declaration in Example 5.6 with the following declaration:

```
external "C" h = compute_enthalpy(P,T);
```

5.7.8.3 Compiler options

Simply saying the external subroutines are written in C or FORTRAN77 does not always provide enough information. For example, FORTRAN77 and C compilers sometimes append or prepend a "_" character to subroutine names in the compiled object code. It is the responsibility of the simulation tool to provide a way to deal with compiler and operating system specific issues like these.

5.7.8.4 Side effects

Functions should not have *side effects* (*i.e.*, they must always return the same output for a given set of inputs). This is particularly important to keep in mind when writing external subroutines because it is easy to inadvertently introduce such side effects. These side effects may come from reading from or writing to global variables or from the use of third party libraries which themselves have side effects.

The way to avoid side effects is to make sure that a function's outputs are dependent only on the inputs. In other words, do not read from or write to any persistent data (*e.g.*, global variables or files). In some cases, it is impractical to avoid keeping persistent data but it is still possible to avoid side effects. For example it is useful, when optimizing the performance of an external subroutine, to introduce some kind of persistent cache. Such approaches are fine so long as they continue to satisfy the restriction that for a given set of input values, the output values are always the same (*i.e.*, the cache improves performance but does not affect the result).

5.8 PROBLEMS

PROBLEM 5.1 *Write a function to perform cubic interpolation. In addition to the arguments used for the* `Piecewise` *function from Example 5.3, add an additional argument that provides the slope of the function (i.e.,* $\frac{dy}{dx}$*) at each grid point. Assuming the value of* x *is defined such that* $x_k \leq x \leq x_{k+1}$*, the*

value of the function can be interpolated using the following equations:

$$\xi = \frac{x - x_k}{x_{k+1} - x_k} \tag{5.15}$$

$$a = y_{k+1} + \left(\frac{dy}{dx}\right)_{k+1} + 3\left[\left(\frac{dy}{dx}\right)_k + y_k - y_{k+1}\right] - 2\left(\frac{dy}{dx}\right)_k - y_k \tag{5.16}$$

$$b = \left(\frac{dy}{dx}\right)_k + 3(y_{k+1} - y_k - \left(\frac{dy}{dx}\right)_k) - \left(\frac{dy}{dx}\right)_{k+1} \tag{5.17}$$

$$c = \left(\frac{dy}{dx}\right)_k \tag{5.18}$$

$$d = y_k \tag{5.19}$$

$$y(x) = a\xi^3 + b\xi^2 + c\xi + d \tag{5.20}$$

where y_i is the value of the function at the i^{th} grid point and $(\frac{dy}{dx})_i$ is the slope of the function at the i^{th} grid point.

PROBLEM 5.2 *Read Section 14.5 and then create a* Modelica *function that computes the Jacobian for the interpolation function in Problem 5.1. To keep things simple, assume that the* `x_grid`*,* `y_grid` *and* `dydx_grid` *values are all constant. (Hint: You really only need to differentiate Equation (5.20) but remember that $\xi = \xi(x)$)*

PROBLEM 5.3 *Write a function that takes a vector (i.e., an array of real numbers) as an argument and returns the magnitude of the vector.*

PROBLEM 5.4 *Write a function to take the inner product of two vectors. Be sure to include assertions that ensure the vectors are the same size.*

PROBLEM 5.5 *Write a function that takes the position and masses of two free bodies and calculates the gravitational force between them. The gravitational force should be returned as a vector. The magnitude of the gravitational force is given by the equation:*

$$F = \frac{M_1 M_2 G}{r^2} \tag{5.21}$$

The force vectors are then calculated as:

$$\vec{F}_{1\to 2} = F\frac{(x_2 - x_1)}{r} \tag{5.22}$$

$$\vec{F}_{2\to 1} = F\frac{(x_1 - x_2)}{r} \tag{5.23}$$

Chapter 6

USING ARRAYS

6.1 CONCEPTS

Although we have covered enough material to build up complex systems, there is still quite a bit of important functionality left to cover. In this chapter, we focus on arrays and the control structures (*i.e.*, `if`, `for` and `while`) used to operate on them.

First, we focus on the declaration of arrays. Modelica allows us to declare arrays of scalars (*e.g.*, an array of floating point numbers) as well as arrays of components (*e.g.*, an array of `Resistor` instances or an array of `record` instances). Arrays of scalars are useful for representing mathematical entities like vectors and matrices.[1] Furthermore, Modelica includes features to support writing vector and matrix equations or assignments. Arrays of components are most useful when a large or variable number of components are needed.

As soon as you start using arrays, you quickly recognize the need for control structures like `for` and `while`. When working with arrays of scalars, control structures are useful for looping over the elements of an array. For arrays of components, these same control structures can be used to help connect components within an array to other models.

6.2 PLANETARY MOTION: ARRAYS OF COMPONENTS

In this section, we look at simulating the motion of several bodies exerting a gravitational force on each other. Figure 6.1 shows a sample configuration of bodies.

[1] Modelica supports arrays with any number of dimensions. In this chapter, we will focus on vectors (*i.e.*, one dimensional arrays).

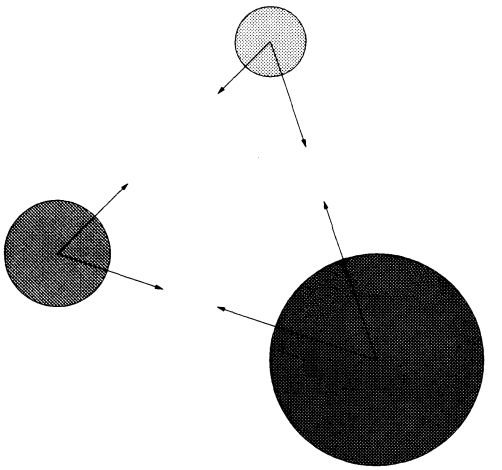

Figure 6.1. Several bodies mutually attracted by gravitational forces.

6.2.1 Connector

```
connector BodyAttachment
  import SI=Modelica.SIunits;
  SI.Position x, y, z; // Prevents using vector equations
  flow SI.Force fx, fy, fz;
  SI.Mass m;
end BodyAttachment;
```

Example 6.1. Poorly designed connector definition for use in multiple body problems.

In order to proceed, we must first decide what our `connector` definition should look like. Each of the objects in our simulation has a mass, position and external force associated with it. One possible `connector` definition for these bodies is shown in Example 6.1. However, writing equations for x, y and z is tedious since the same equation (*i.e.*, Newton's Law) applies to each of them. For that reason, we use vectors to represent position and force as shown in Example 6.2.

Because the forces on each body must sum to zero (in each dimension), the `Force` component of the connector has been declared as a `flow` variable.

```
connector BodyAttachment
  import Modelica.SIunits;

  SIunits.Position x[3];
  flow SIunits.Force f[3];
  SIunits.Mass m;
end BodyAttachment;
```

Example 6.2. Better connector definition for multiple body problems (using vectors).

```
model Body
  import Modelica.SIunits;
  parameter SIunits.Mass M=1.0;
  parameter SIunits.Velocity init_v[3] = {0,0,0};
  parameter SIunits.Position init_x[3] = {0,0,0};

  BodyAttachment b;
protected
  SIunits.Position x[3](start=init_x,fixed=true);
  SIunits.Velocity v[3](start=init_v,fixed=true);
  SIunits.Acceleration a[3];
equation
  b.x = x;
  b.m = M;
  v = der(x);
  a = der(v);
  M*a = b.f;
end Body;
```

Example 6.3. Model for a free body in three dimensional space.

6.2.2 Bodies in three dimensional space

To model the behavior of a body floating in three dimensional space subject to external forces, we use Newton's law:

$$\vec{F} = m\vec{a} \tag{6.1}$$

where $\vec{F}$ is the force exerted on the body, m is the mass of the body and $\vec{a}$ is the acceleration of the body. Using Newton's Law, creating a model for such a body is quite simple as shown in Example 6.3.

Notice how compactly the equations of motion can be written for these bodies. Equations like `v = der(b.x)` are vector equations. In other words, each component of `v` is equated to the derivative of the corresponding component of `b.x`.

The purpose of the `protected` section in models is to declare any quantities that are internal to the model. Such protected quantities, like the `Velocity` in this case, cannot be accessed by external models. These protected components can only be accessed by the model in which they appear or any model which `extends` from it.

Because the position and velocity of each body are `protected`, it is not possible for other models to directly modify the `start` attributes of these variables. Instead, their initial values are supplied by the public parameters `init_x` and `init_v` respectively. Note how these parameters are used in modifying the `start` attribute. This is one way to allow limited access (*e.g.*, access for modifying only the `start` attribute) to `protected` components.

6.2.3 Gravitational attraction

Gravitational attraction between two bodies is computed using the following equation:

$$F = \frac{M_1 M_2 G}{r^2} \tag{6.2}$$

where F is the magnitude of the attracting force, M_1 is the mass of the first body, M_2 is the mass of the second body, G is the universal gravitational constant and r is the distance between the bodies. This equation only computes the magnitude of the force. It is then necessary to multiply this force by the unit vectors representing the relative positions of each body from the other. This leads to the following two equations:

$$\vec{F}_{1\to 2} = F\frac{(x_2 - x_1)}{r} \tag{6.3}$$

$$\vec{F}_{2\to 1} = F\frac{(x_1 - x_2)}{r} \tag{6.4}$$

where $\vec{F}_{1\to 2}$ is the force exerted on the first body by the second and $\vec{F}_{2\to 1}$ is the force exerted on the second body by the first. This gravitational force is computed by the `function` shown in Example 6.4. Note that the equation for `on_body1` in Example 6.4 is a vector equation.

Just having a function that computes gravitational forces is not sufficient, we must have a gravitational attraction model which applies those forces to the bodies in our system. Once we have the `CalcForce` function in Example 6.4, we can write our gravitational attraction model as shown in Example 6.5.

Here we see again another parallel to software development. When developing a large software system, the declaration of a function with such a specific purpose and generic name would be frowned upon. The reason is that somebody else may have written a function with the same name for use with a different model. For example, imagine some day we wish to use a model developed by someone else and this model also uses a `function` called

```
function CalcForce "Calculate Force on body1 due to body2"
  import Modelica.SIunits;

  input SIunits.Position body1[3];
  input SIunits.Mass M1;
  input SIunits.Position body2[3];
  input SIunits.Mass M2;
  output SIunits.Force on_body1[3];
protected
  SIunits.Distance r;
  SIunits.Force F;
algorithm
  r := sqrt((body1[1]-body2[1])^2+(body1[2]-body2[2])^2+
          (body1[3]-body2[3])^2);
  F := M1*M2*Modelica.Constants.G/r^2;
  on_body1 := F*(body2-body1)/r;
end CalcForce;
```

Example 6.4. A function to calculate gravitational force.

```
model GravitationalAttraction
  BodyAttachment b1, b2;
equation
  b1.f = -CalcForce(b1.x, b1.m, b2.x, b2.m);
  b2.f = -b1.f;
end GravitationalAttraction;
```

Example 6.5. A gravitational attraction model.

`CalcForce`. Let us assume that their `CalcForce` is different from ours (*i.e.*, it performs a different calculation and/or uses a different number of arguments). One of two things is likely to happen. One possibility is that we will not realize that they require a **`function`** named `CalcForce` and their model will attempt to use our **`function`** called `CalcForce`. The other possibility is that we realize their function is required in which case we have two functions named `CalcForce`, which is not allowed.[2]

One way to avoid both of these situations is to declare the **`function`** within the model that uses it. In such a case, our model would look like the one seen in Example 6.6. Notice that we have included the **`function`** in a **`protected`** section so that other models may not make use of this **`function`** (except any model that **`extends`** from the `GravitationalAttraction` model). The other

[2] In Modelica, it is illegal to define two function with the same (fully qualified) name.

way to protect against such problems is to give functions very specific names so there will be no potential for confusion (*e.g.*, `CalcGravitationalForce`).

```
model GravitationalAttraction
  BodyAttachment b1, b2;
protected
  function CalcForce "Calculate Force on body1 from body2"
    input Modelica.SIunits.Position body1[3];
    input Modelica.SIunits.Mass M1;
    input Modelica.SIunits.Position body2[3];
    input Modelica.SIunits.Mass M2;
    output Modelica.SIunits.Force on_body1[3];
    ...
  end CalcForce;
equation
  b1.f = -CalcForce(b1.x, b1.mass, b2.x, b2.mass);
  b2.f = -b1.f;
end GravitationalAttraction;
```

Example 6.6. Encapsulating the gravitational force calculation

6.2.4 Simulating several bodies

```
model BinarySystem "A binary system"
  Body sun(M=1.989e+30);
  Body earth(M=5.976e+24, init_v={0, 29.29e+3, 0},
                    init_x={152.1e+9, 0, 0});
  GravitationalAttraction earth_sun;
equation
  connect(earth_sun.b1,sun.b);
  connect(earth_sun.b2,earth.b);
end BinarySystem;
```

Example 6.7. Creating a binary system.

Example 6.7 shows how we might write a model for a binary system (*i.e.*, a system containing two bodies). The model includes astronomical data for the Earth and the Sun. Example 6.8 shows how easy it is for us to extend the `BinarySystem` to include the Moon as well. Figure 6.2 shows the path of the Earth during a simulation of $31.5581 \cdot 10^6$ seconds (approximately 1 year). As expected, this results in one orbit of the Earth around the sun. In addition, it also shows the Moon's path, exaggerated in the figure by a factor of 20, as it orbits the earth during the same period.

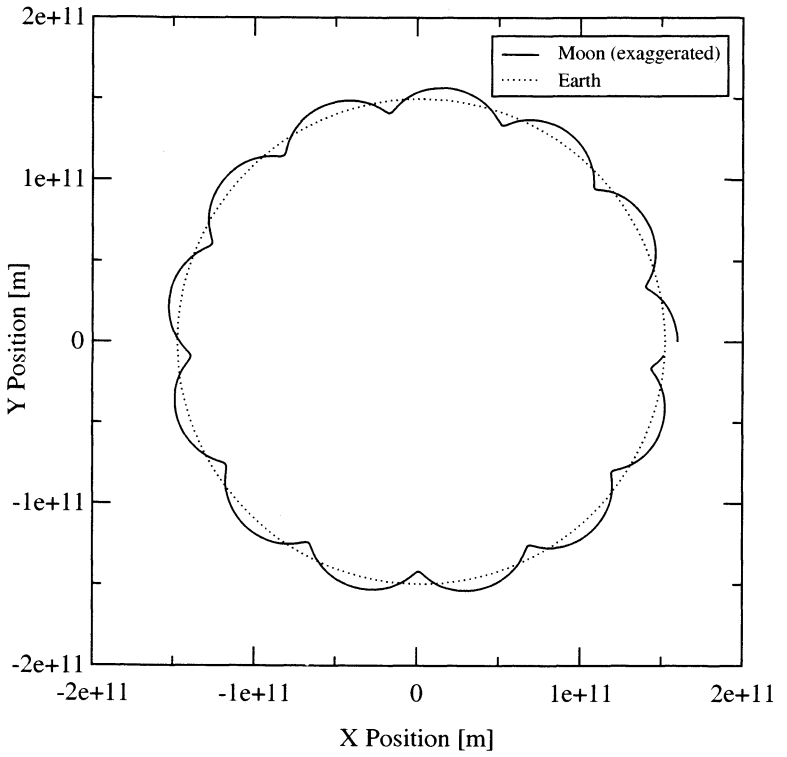

Figure 6.2. Simulating the motion of the Earth and the Moon for approximately 1 year.

```
model TernarySystem "Earth, Moon & Sun"
  extends BinarySystem;
  Body moon(M=7.349e+22,
                  init_v={0,29290+1020,0},
                  init_x={152484e+6, 0, 0});
  GravitationalAttraction moon_earth;
  GravitationalAttraction moon_sun;
equation
  connect(moon_earth.b1,moon.b);
  connect(moon_earth.b2,earth.b);
  connect(moon_sun.b1,moon.b);
  connect(moon_sun.b2,sun.b);
end TernarySystem;
```

Example 6.8. A system including the Earth, Sun and Moon.

6.3 SIMPLE 1D HEAT TRANSFER: ARRAYS OF VARIABLES

When performing simulations, a common need is to solve for variables which are not only a function of time, but also of location. Now we will use a one-dimensional heat transfer problem to demonstrate how arrays can be used to simulate such systems.

6.3.1 Governing equations

We start by listing the equations needed to solve this problem. These equations will then be transformed into Modelica models.

6.3.1.1 Conservation of energy

For example, consider the following *partial differential equation* for heat conduction (see, *e.g.*, Fowler, 1997):

$$\frac{d}{dt}\int_V \rho c_p T \, dV = -\int_S \vec{J} \cdot \hat{n} \, dS \tag{6.5}$$

where V is the volume of the domain being considered, S is the boundary surface of V, ρ is the density of the material, c_p is the specific heat capacity of the material, T is the temperature at any given point in the domain, $\vec{J}$ is the heat flux at a given point on the boundary and $\hat{n}$ is the vector normal to the surface S at a given point on the surface.

Let us assume we are solving this equation in a rod with a uniform cross-sectional area, A (see Figure 6.3). Integrating the left hand side of Equation (6.5) over a section of length L gives us:

$$\frac{d}{dt}\int_V \rho c_p T \, dV \Rightarrow AL\rho c_p \frac{\partial T_V}{\partial t} \tag{6.6}$$

which represents the thermal capacitance of that section assuming the section has an effective uniform temperature of T_V. The right hand side of Equation (6.5) could represent a variety of different heat transfer mechanisms (*i.e.*, conduction, convection or radiation) over all surfaces of the rod. For the moment, let us consider only the case of conduction over the surfaces S_l and S_r, in Figure 6.3. Using Fourier's law ($\vec{q} = -k\frac{\partial T}{\partial \vec{x}}$) the contribution of these surfaces to the right hand side of Equation (6.5) would be:

$$-\int_S \vec{J} \cdot \hat{n} \Rightarrow Ak \left.\frac{\partial T}{\partial x}\right|_S \tag{6.7}$$

where k is the thermal conductivity of the material and $\left.\frac{\partial T}{\partial x}\right|_S$ is the temperature gradient normal to the surface S. Assuming these specific surfaces and modes

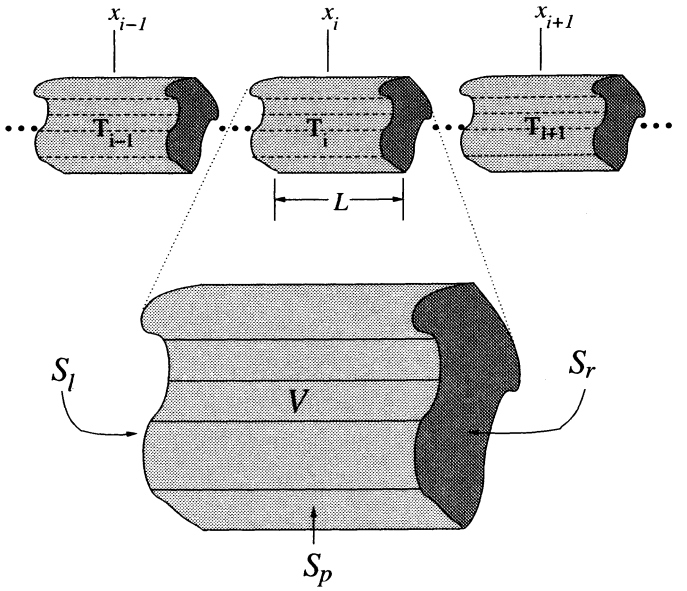

Figure 6.3. Heat transfer in a one-dimensional rod.

of heat transfer, we can rewrite Equation (6.5) as:

$$AL\rho c_p \frac{\partial T_V}{\partial t} = Ak \left.\frac{\partial T}{\partial x}\right|_{S_r} - Ak \left.\frac{\partial T}{\partial x}\right|_{S_l} \tag{6.8}$$

6.3.1.2 Spatial discretization

The next step in deriving the system of equations is to eliminate the spatial derivatives (*i.e.*, $\frac{\partial T}{\partial x}$). To do this, we must make some *a priori* assumptions about how the temperature, T, is distributed along the rod in the x-direction. Let us assume that from one discrete section of the rod to another the temperature varies linearly. If we make this assumption, Equation (6.8) can be rewritten as:

$$AL\rho c_p \frac{\partial T_i}{\partial t} = Ak \frac{T_{i+1} - T_i}{x_{i+1} - x_i} - Ak \frac{T_i - T_{i-1}}{x_i - x_{i-1}} \tag{6.9}$$

Now we have an equation for the time derivative of the temperature, T, for the i^{th} section of the rod written in terms of geometric quantities (*i.e.*, A and L), material properties (*i.e.*, k, ρ and c_p) and the temperatures of neighboring sections (*i.e.*, T_{i-1} and T_{i+1}).

6.3.1.3 Simplifications

At this point, we may be tempted to divide Equation (6.9) by A in order to simplify it into an equation like:

$$L\rho c_p \frac{\partial T_i}{\partial t} = k\left(\frac{T_{i+1} - T_i}{x_{i+1} - x_i} - \frac{T_i - T_{i-1}}{x_i - x_{i-1}}\right) \tag{6.10}$$

Furthermore, we might wish to assume that the discretization of the rod is uniform which would further reduce the equation to:

$$\rho c_p \frac{\partial T_i}{\partial t} = \frac{k\left(T_{i+1} - 2T_i + T_{i-1}\right)}{\Delta x^2} \tag{6.11}$$

In fact, this is a common form of the heat transfer equation. Notice that the terms in Equation (6.8) have units of heat flow rate (*i.e.*, the time derivative of a conserved quantity). However, in Equation (6.11) the terms have units of heat flow rate per unit volume. The problem with Equation (6.11) is that it is no longer a conservation equation. To understand this, let us revisit Equation (6.8). Let us annotate the equation with information about each term:

$$\underbrace{AL\rho c_p \frac{\partial T}{\partial t}}_{\text{thermal capacitance}} = \underbrace{Ak\left.\frac{\partial T}{\partial x}\right|_r}_{\text{conduction at } S_r} - \underbrace{Ak\left.\frac{\partial T}{\partial x}\right|_l}_{\text{conduction at } S_l} \tag{6.12}$$

Imagine we wish to add a convective heat transfer term to represent heat loss over the surface S_p (see Figure 6.3). In that case, we would amend Equation (6.12) to include an additional term giving us:

$$AL\rho c_p \frac{\partial T}{\partial t} = Ak\left.\frac{\partial T}{\partial x}\right|_r - Ak\left.\frac{\partial T}{\partial x}\right|_l - \underbrace{A_p h\left(T_i - T_\infty\right)}_{\text{convection at } S_p} \tag{6.13}$$

where T_∞ represents the ambient temperature and A_p is the area of surface S_p in Figure 6.3. Note that the simplifying assumptions are no longer possible with this form of the equation (*e.g.*, we cannot eliminate A from each term). Because this equation remains in units of heat flow rate, adding a new mode of heat transfer is as simple as adding another term. The same cannot be said of the simplified form shown in Equation (6.11). In the next sections we will consider the advantages and disadvantages of creating models based on Equation (6.9) and Equation (6.11).

6.3.2 Equation based approach

Example 6.9 shows a model which uses Equation (6.11) and also includes several boundary conditions. The initial temperature of every point is $300K$. We assume that the temperature of the first node jumps from $300K$ to $1000K$

after 1 second and the temperature of the last node is fixed at $300K$ (these conditions are enforced by the last two equations in Example 6.9). For this example, the solution reaches steady state after approximately 25 seconds of simulation time (as we will see later in Figure 6.8). Example 6.9 uses the `fill()` function (which is described in greater detail in Table 6.1) to create the array of initial temperatures.

Example 6.9 shows an equation based approach to solving partial differential equations. In this example, the temperature variables are represented by an array and the equations are generated using `for` loops. Note that we can choose how fine the discretization is by changing the value of `n` independent of the geometry of the problem (*i.e.*, total length). As mentioned previously, a system written in this way lacks the flexibility to add additional modes of heat transfer without having to reformulate the fundamental equation (*i.e.*, Equation (6.11)).

```
model HeatTransfer "One Dimensional Heat Transfer"
  import Modelica.SIunits;

  // Configuration parameters
  parameter Integer n=10 "Number of Nodes";
  parameter SIunits.Density rho=1.0 "Material Density";
  parameter SIunits.HeatCapacity c_p=1.0;
  parameter SIunits.ThermalConductivity k=1.0;
  parameter SIunits.Length L=10.0 "Domain Length";

  // Temperature Array
  SIunits.Temp_K T[n](start=fill(300,n)) "Nodal Temperatures";
protected
  // Computed parameters
  parameter SIunits.Length dx=L/n "Distance between nodes";
equation
  // Loop over interior nodes
  for i in 2:n-1 loop
    rho*c_p*der(T[i]) = k*(T[i+1]-2*T[i]+T[i-1])/dx^2;
  end for;

  // Boundary Conditions
  T[1] = if time>=1 then 1000 else 300;
  T[n] = 300;
end HeatTransfer;
```

Example 6.9. Using arrays of variables to solve Equation (6.11).

6.3.3 Component based approach

In Chapter 3, we saw how to transform a model containing a complete system of equations into a collection of reusable models. We will once again demonstrate how to perform such a transformation but this time using models which contain partial differential equations. Using Equation (6.9) as the basis for our component based approach, each term in Equation (6.9) will be represented by a different model and the terms will be summed automatically when the models are connected.

6.3.3.1 Connector definitions

```
connector ThermalNode "Thermal Connector"
  Modelica.SIunits.Temp_K T(start=300);
  flow Modelica.SIunits.HeatFlowRate q;
end ThermalNode;
```

Example 6.10. Connector for heat transfer.

As usual, we start with the connector definition. We will use a connector with temperature as the across variable and heat flow rate as the through variable. Example 6.10 shows the connector used for this example.

6.3.3.2 Thermal conduction

Now that we have our connector definition, we can begin writing the various models required. We start with the heat conduction model which represents the right hand side terms in Equation (6.9). Example 6.11 shows how we can express thermal conduction as a model independent of other modes of heat transfer.

6.3.3.3 Thermal capacitance

Next, we need to represent the contribution on the left hand side of Equation (6.9). This term represents the thermal capacitance of the rod material for a given volume. A model which describes this behavior is shown in Example 6.12.

6.3.3.4 Fixed temperature boundary condition

The last component we need, shown in Example 6.13, is one to represent a fixed temperature, or *Dirichlet*, boundary condition.

```
model ThermalConduction "1-D Conduction Heat Transfer"
  import Modelica.SIunits;
  // Physical parameters
  parameter SIunits.ThermalConductivity k=1.0;
  parameter SIunits.Length L=1.0;
  parameter SIunits.Area A=1.0;

  // Connectors
  ThermalNode a, b;
equation
  a.q = A*k*(a.T-b.T)/L;
  b.q = -a.q;
end ThermalConduction;
```

Example 6.11. Thermal conduction.

```
model ThermalCapacitance "Capacitance of a rod section"
  ThermalNode p "Midpoint connection";
  parameter Modelica.SIunits.SpecificHeatCapacity cp;
  parameter Modelica.SIunits.Density rho;
  parameter Modelica.SIunits.Length L;
  parameter Modelica.SIunits.Area A;
protected
  parameter Modelica.SIunits.Volume V=A*L;
equation
  // Conservation of energy
  V*cp*rho*der(p.T) = p.q;
end ThermalCapacitance;
```

Example 6.12. Thermal capacitance.

```
model FixedTemperature
  Modelica.Blocks.Interfaces.InPort T(final n=1);
  ThermalNode d;
equation
  d.T = T.signal[1];
end FixedTemperature;
```

Example 6.13. Fixed temperature boundary condition.

6.3.3.5 Conducting rod

Now before bringing all the components together, let us look at an example of how the spatial aspect of the problem can be bundled up within a single component model. We do this by creating a network of *lumped* components

as we have in previous sections.[3] Figure 6.4 gives a graphical representation of such a one dimensional conducting rod and Example 6.14 contains the Modelica source code. Note that the connectors `a` and `b` in `ConductingRod` represent the external connection points for the rod.

Also note that the `ConductingRod` model enforces the uniform discretization of the rod (see Figure 6.4). For example, the model makes sure the length in the conduction models (distance from center of one segment to center of another) is consistent with the length of the thermal capacitance models (distance from left surface to right surface).

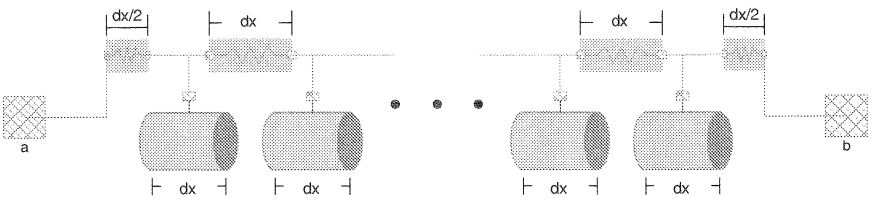

Figure 6.4. Schematic for `ConductingRod` model in Example 6.14.

6.3.3.6 Sample heat transfer problem

Now we have all the components we need to represent the same system as the one shown in Example 6.9. This time, we have created our system, `HTProblem1`, from reusable components rather than writing the complete conservation equation inside a single model as we did in the `HeatTransfer` model from Example 6.9.

The simulation results from `HTProblem1` can be seen in Figure 6.5. The temperatures at the end of the simulation are plotted in Figure 6.8 as a function of longitudinal distance along the rod. Figure 6.8 demonstrates that the steady state temperature profile develops into a linear solution which is exactly the solution expected for this problem.

The results from Example 6.9 (*i.e.*, `HeatTransfer`) and Example 6.15 (*i.e.*, `HTProblem1`) are identical. While the `HTProblem1` model is more compact and readable, some people prefer the approach taken in the `HeatTransfer` model because the partial differential equation is shown explicitly.

[3]The term lumped refers to models where the spatial aspect of the problem is not considered. The term *distributed* is used when the spatial aspect is explicitly described. In this sense, Example 6.14 encapsulates a distributed model inside a lumped model.

```
model ConductingRod
  import Modelica.SIunits;

  parameter SIunits.Length L=1.0 "Total length";
  parameter SIunits.Area A=1.0 "Cross-sectional area";
  parameter SIunits.SpecificHeatCapacity cp=1.0;
  parameter SIunits.Density rho=1.0;
  parameter SIunits.ThermalConductivity k=1.0;
  parameter Integer n=10 "Number of sections";

  ThermalNode a, b; // External connections
protected
  parameter SIunits.Length dx=L/n;
  ThermalCapacitance cap[n](L=dx,A=A,rho=rho,cp=cp);
  ThermalConduction c_cond[n-1](L=dx,A=A,k=k);
  ThermalConduction l_cond(L=dx/2,A=A,k=k);
  ThermalConduction r_cond(L=dx/2,A=A,k=k);
equation
  for i in 1:n-1 loop
    connect(c_cond[i].a,cap[i].p);
    connect(c_cond[i].b,cap[i+1].p);
  end for;
  connect(a,l_cond.a);
  connect(l_cond.b,cap[1].p);
  connect(b,r_cond.b);
  connect(r_cond.a,cap[n].p);
end ConductingRod;
```

Example 6.14. A rod which conducts heat.

```
model HTProblem1 "Conducting rod with boundary conditions"
  Modelica.Blocks.Sources.Constant Tl(k={300.0});
  Modelica.Blocks.Sources.Step Tr(height={700.0},
          offset={300.0}, startTime={10.0});
  FixedTemperature left, right;
  ConductingRod rod(n=10,L=10.0,k=1.0,cp=1.0,rho=1.0);
equation
  connect(Tl.outPort,left.T);
  connect(Tr.outPort,right.T);
  connect(left.d, rod.a);
  connect(right.d, rod.b);
end HTProblem1;
```

Example 6.15. Heat transfer in a conducting rod with boundary conditions.

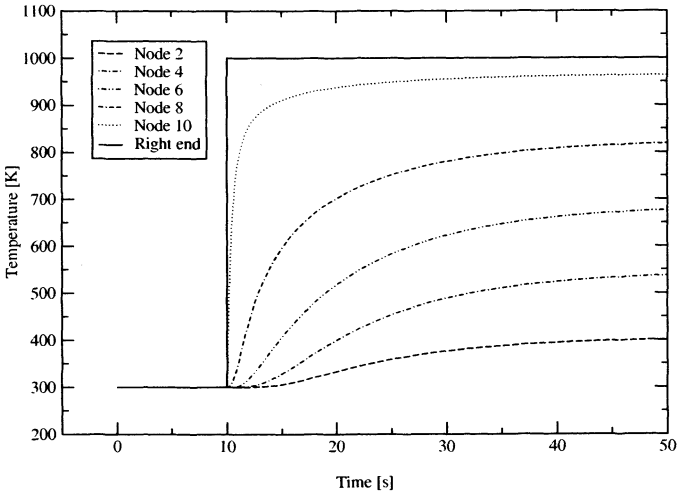

Figure 6.5. Solution for `HTProblem1` model in Example 6.15.

6.3.3.7 Conducting rod with convection

Now, let us model the system described by Equation (6.13). In other words, we wish to add a thermal convection term. Example 6.16 shows a model for representing thermal convection.

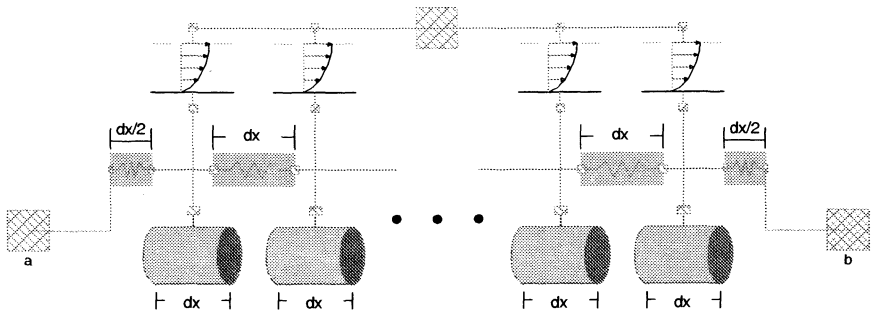

Figure 6.6. Schematic for `ConductingRodWithConvection` shown in Example 6.17.

Next, Example 6.17 shows how we can extend the `ConductingRod` model in Example 6.14 to include thermal convection by adding a few components and connections. Note the difference between the diagram for `ConductingRodWithConvection` shown in Figure 6.6 and the original diagram for `ConductingRod` shown previously in Figure 6.4.

```
model ThermalConvection "1-D Convective Heat Transfer"
  // Physical parameters
  import Modelica.SIunits;
  parameter SIunits.CoefficientOfHeatTransfer h=1.0;
  parameter SIunits.Area A=1.0;

  // Connectors
  ThermalNode a, b;
equation
  a.q = A*h*(a.T-b.T);
  b.q = -A*h*(a.T-b.T);
end ThermalConvection;
```

Example 6.16. A model of thermal convection.

```
model ConductingRodWithConvection
  import Modelica.SIunits;
  extends ConductingRod;

  parameter SIunits.Length perimeter=1.0;
  parameter SIunits.CoefficientOfHeatTransfer h=1.0;
  ThermalNode ambient;
protected
  parameter SIunits.Area As=perimeter*dx;
  ThermalConvection conv[n](h=h,A=As);
equation
  for i in 1:n loop
    connect(cap[i].p,conv[i].a);
    connect(ambient,conv[i].b);
  end for;
end ConductingRodWithConvection;
```

Example 6.17. Addition of the convection effect.

6.3.3.8 Another sample heat transfer problem

By adding a convective heat transfer contribution to Example 6.15 we arrive at the model shown in Example 6.18. Figure 6.7 shows simulation results for Example 6.18. If we compare the results in Figures 6.5 and 6.7, we can see two distinct features resulting from the convection. The first is that the temperatures shown in Figure 6.7 start rising immediately because of the convective heat transfer. Another effect, due to convection, is that the steady state temperatures are not evenly spaced as they were in Figure 6.5.

Another interesting comparison between Examples 6.15 and 6.18 is shown in Figure 6.8. The figure contains a comparison between the steady state

```
model HTProblem2 "Variation on HTProblem1"
  Modelica.Blocks.Sources.Constant Tl(k={300.0});
  Modelica.Blocks.Sources.Constant Tinf(k={600.0});
  Modelica.Blocks.Sources.Step Tr(height={700.0},
       offset={300.0}, startTime={10.0});
  FixedTemperature left, right, wall;
  ConductingRodWithConvection rod(n=10,L=10.0,
       k=1.0,cp=1.0,rho=1.0,h=0.3);
equation
  connect(Tl.outPort,left.T);
  connect(Tr.outPort,right.T);
  connect(Tinf.outPort,wall.T);
  connect(left.d, rod.a);
  connect(right.d, rod.b);
  connect(wall.d, rod.ambient);
end HTProblem2;
```

Example 6.18. Heat transfer problem involving conduction and convection.

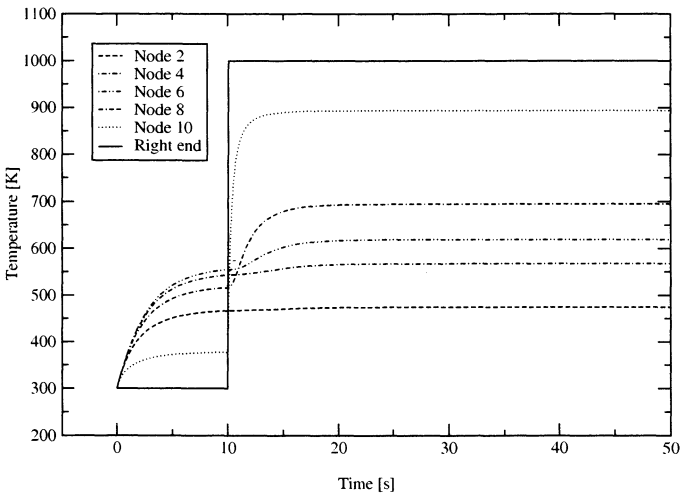

Figure 6.7. Simulation results for `HTProblem2` model shown in Example 6.18.

temperature distributions of these two examples. The solution involving only conduction develops into a linear profile which is also the analytical solution. The solution with conduction and convection is clearly influenced by the ambient temperature of $600K$.

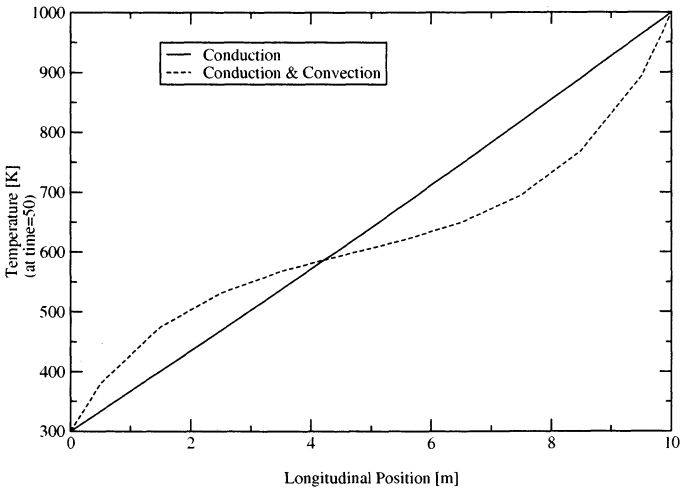

Figure 6.8. Comparison of steady-state solutions to `HTProblem1`.

6.3.4 Standard heat transfer components

While the MSL does not currently contain definitions to support heat transfer modeling, a library of heat transfer component definitions, called `Thermal`, is included on the companion CD-ROM. Furthermore, a thermal library will eventually be incorporated into the MSL and it is likely such a library will have the same basic components and `connector` definitions as those found in the `Thermal` library.

Example 6.19 shows how the `ConductingRod` model from Example 6.14 would look if we had used the components in the `Thermal` package. As we shall see in Section 10.3, the `Thermal` package contains additional models, beyond the ones shown in Example 6.19, which provide connections between the thermal domain and other domains like the electrical and mechanical domains.

6.3.5 Summary

We started by showing how we can quickly express a particular partial differential equation in Modelica. Then, we saw how, with a little more work, we could create a collection of reusable component models. With these reusable component models we can pose and solve a wide variety of heat transfer problems with different heat transfer pathways, modes and boundary conditions.

```
model ConductingRod_Thermal
  import Thermal.Basic1D;
  import Modelica.SIunits;

  parameter SIunits.Length L=1.0 "Total length";
  parameter SIunits.Area A=1.0 "Cross-sectional area";
  parameter SIunits.SpecificHeatCapacity cp=1.0;
  parameter SIunits.Density rho=1.0;
  parameter SIunits.ThermalConductivity k=1.0;
  parameter Integer n=10 "Number of sections";

  Thermal.Interfaces.Node a, b;
protected
  parameter SIunits.Length dx=L/n;
  Basic1D.Capacitance cap[n](V=dx*A,rho=rho,cp=cp);
  Basic1D.Conduction c_cond[n-1](L=dx,A=A,k=k);
  Basic1D.Conduction l_cond(L=dx/2,A=A,k=k);
  Basic1D.Conduction r_cond(L=dx/2,A=A,k=k);
equation
  for i in 1:n-1 loop
       connect(c_cond[i].a,cap[i].n);
       connect(c_cond[i].b,cap[i+1].n);
  end for;
  connect(a,l_cond.a);
  connect(l_cond.b,cap[1].n);
  connect(b,r_cond.b);
  connect(r_cond.a,cap[n].n);
end ConductingRod_Thermal;
```

Example 6.19. A conducting rod using the `Thermal` library.

6.4 USING ARRAYS WITH CHEMICAL SYSTEMS

6.4.1 Background

Simulations of chemical systems are usually concerned with the reaction and transport of chemical constituents. These constituents do not move through the system on their own. Instead, they are generally found in mixtures with other chemicals.

In this section, we first discuss how general chemical systems can be represented in Modelica. To do this, we will build a basic collection of general chemical models. Then, we will create a specific chemical system to test these basic models.

Before we start introducing models it is necessary to cover some basic notation. When a constituent is surrounded by square brackets (*e.g.*, [A]) that quantity is the concentration of that constituent measured in moles per cubic

meter. Furthermore, this chapter contains several chemical equations, *e.g.*, :

$$\mathrm{A} + \mathrm{Y} \xrightarrow[k_{ay}]{} \mathrm{X} + \mathrm{P} \tag{6.14}$$

The constituents on the left hand side of the equation are called the reactants and the constituents on the right hand side are called the products. The reaction coefficient, in this case k_{ay}, appears below the arrow and is used to compute the rate of the reaction (as we shall see shortly).

The model we have chosen to use as our example is called the "Oregonator".[4] The Oregonator is a simplified model of the Field-Körös-Noyes (FKN) mechanism (see Earley, 1998) which is a chemical model of the Belousov-Zhabotinskii reaction (described in detail in Fowler, 1997).

The Oregonator model is represented by the following reactions:

$$\mathrm{A} + \mathrm{Y} \xrightarrow[k_{ay}]{} \mathrm{X} + \mathrm{P} \tag{6.15}$$

$$\mathrm{X} + \mathrm{Y} \xrightarrow[k_{xy}]{} 2\mathrm{P} \tag{6.16}$$

$$\mathrm{A} + \mathrm{X} \xrightarrow[k_{ax}]{} 2\mathrm{X} + 2\mathrm{Z} \tag{6.17}$$

$$2\mathrm{X} \xrightarrow[k_{2x}]{} \mathrm{A} + \mathrm{P} \tag{6.18}$$

$$\mathrm{B} + \mathrm{Z} \xrightarrow[k_{bz}]{} (1/2)f\mathrm{Y} \tag{6.19}$$

where A, P, X, Y and Z represent BrO_3^-, HOBr, $HBrO_2$, Br^- and Ce^{4+} respectively, B represents oxidizable organic species and f represents the extent to which organic species participate which, in turn, regulates the regeneration of Y. Figure 6.9 shows how the Oregonator system could be visualized.

6.4.2 Chemical reactions

The Oregonator model is very simple and we could write out the differential equations in just a few lines of Modelica. On the other hand, a better investment of our time would be to build a collection of reusable Modelica models to represent chemical systems in general. Once such a collection exists, with a minimal amount of Modelica code we can create models for an enormous variety of chemical systems rather than just one.

[4]The model is called the Oregonator because it was developed at the University of Oregon.

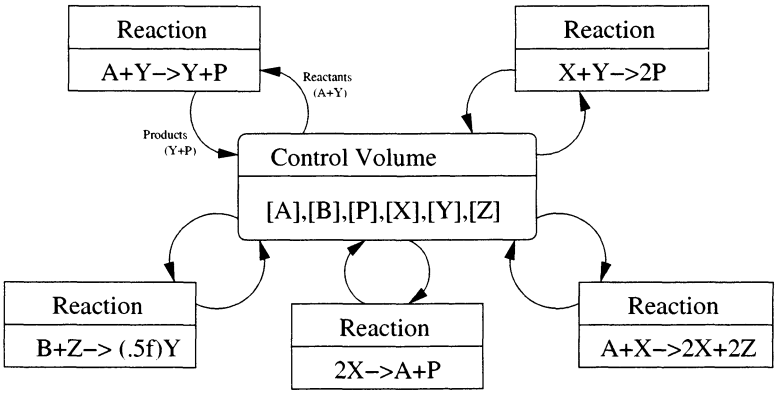

Figure 6.9. Visualization of the Oregonator reaction.

6.4.3 Mathematical form

Many chemical system models are represented as ordinary differential equations of the form:

$$[\dot{C}] = g([C]) \tag{6.20}$$

where $[C]$ is a vector that represents the concentration of the different constituents. As an example, after a tedious set of transformations the Oregonator system of equations can be written as:

$$[\dot{\mathrm{X}}] = -k_{xy}[\mathrm{X}][\mathrm{Y}] - 2k_{2x}[\mathrm{X}][\mathrm{Y}] + k_{ay}[\mathrm{A}][\mathrm{Y}] \tag{6.21}$$

$$[\dot{\mathrm{Y}}] = -k_{xy}[\mathrm{X}][\mathrm{Y}] - k_{ay}[\mathrm{A}][\mathrm{Y}] + (f/2)k_{bz}[\mathrm{B}][\mathrm{Z}] \tag{6.22}$$

$$[\dot{\mathrm{Z}}] = 2[\mathrm{A}][\mathrm{X}] - k_{bz}[\mathrm{B}][\mathrm{Z}] \tag{6.23}$$

assuming [A] and [B] are fixed. The equation for [P] is usually neglected since it is a product but not a reactant and therefore does not influence the trajectories of [X], [Y] and [Z].

The difficulty with this form is that all the reactions are combined on the right hand side of the equation. It is not possible to pick out details of specific reactions or to understand some of the fundamental assumptions that went into the formulation of the differential equations. As a consequence, in order to make adjustments (*e.g.*, adding another reaction) it is necessary to work backward from these equations to Equations (6.15)-(6.19), make any adjustments and then re-derive a new set of mathematical equations.

In contrast, the models we will develop in this section map directly to Equations (6.15)-(6.19) and no further derivation will be necessary. In other words, the natural representation of chemical reactions can be used. This allows much greater flexibility in modifying the system of reactions. Furthermore, it

will allow us to isolate the effects of individual reactions and avoid the tedious task of performing the state space transformation.

6.4.4 Basic chemical models

In this section, we will develop a library of models and place them in a `package` called `Chemistry`. The following packages will be nested inside the `Chemistry` `package`:

- `Types`: Contains any definitions that are specific to the `Chemistry` package.
- `Interfaces`: Contains `connector` definitions and any `partial` model definitions.
- `Functions`: Contains `function` definitions specific to the `Chemistry` package.
- `Basic`: Contains basic models used for chemical models.

6.4.4.1 Connector definition

Normally, we would begin by creating a `connector` definition. However, in this case we must first define a molar flow rate type as follows:

```
package Chemistry
  package Types
    type MolarFlowRate=Real(quantity="MolarFlowRate",
                            unit="mol/s");
    ...
  end Types;
  ...
end Chemistry;
```

Now that we have defined `MolarFlowRate` we can define the `connector`. For the chemical systems presented in this chapter, we represent the availability of constituents using concentrations (*i.e.*, number of moles per cubic meter). The concentration of a particular constituent is an *intensive property* of the mixture. It is quite common and convenient to measure the potentials (*i.e.*, the across variables) in a system using intensive properties. For example, in thermodynamic systems pressure and temperature (both of which are intensive) are frequently used as the potentials.

On the other hand, in order for a `connect` statement to generate proper conservation equations, the `flow` variables must be the time derivative of an *extensive property* (an issue we touched on briefly in Section 6.3.1.3 as well). For example, in thermodynamic systems the `flow` variables are typically mass flow rate and heat flow rate which are the time derivatives of mass and energy, respectively. Applying this same modeling principle (of extensive flows) to our

chemical system results in the `flow` variables being measured as molar flow rates (*i.e.*, number of moles per second).[5]

Taking these considerations into account, we will use the following `connector` definition for all of our chemical models:

```
package Chemistry
  ...
  package Interfaces
    connector Mixture "A chemical mixture"
      parameter Integer nspecies;
      Modelica.SIunits.Concentration c[nspecies];
      flow Chemistry.Types.MolarFlowRate r[nspecies];
    end Mixture;
  end Interfaces;
  ...
end Chemistry;
```

This `connector` is used for interactions involving chemical mixtures. The concentrations, `c`, will have units mol/m^3 and the flow rates will be in mol/s. This means that any model which attaches to such a `connector` will have access to the concentrations of each of the constituents at that connection point and will have the ability to absorb or emit chemicals (*e.g.*, due to chemical reactions). Note that this model (and all the others in this section) will require knowledge of how many chemical species are present. It is assumed that if there are `nspecies` number of species, then each species will have a unique identifying number between `1` and `nspecies` which will be used as an index into the various arrays in the models.

One thing to note about the nested `Types` and `Interfaces` packages inside the `Chemistry` package is that they are used by other nested packages. The lookup rules in Modelica (described in detail in Chapter 9) allow other nested packages to refer to the components of the `Types` and `Interfaces` using names such as `Interfaces.Mixture`. Such usage can be seen in several of the following code fragments.

6.4.4.2 Chemical control volume

The first thing we require is a place to keep our chemicals. For this, we define the following `Volume` model:

```
package Chemistry
  ...
```

[5]While this combination of intensive potentials and extensive flows is quite common in thermodynamic systems, it is important to note that this is an unusual convention for chemical systems. The typical convention used in chemical systems is to measure the potential in moles and the reaction rates in moles per second or to measure the potential in moles per cubic meter and the reaction rates in moles per cubic meter per second (see Barton, 2000).

```
  package Basic
    model Volume "Volume containing a chemical mixture"
      import SI=Modelica.SIunits;
      parameter Integer nspecies;
      parameter SI.Volume v=.001;
      parameter SI.AmountOfSubstance i_moles[nspecies]=
        fill(1,nspecies);
      Interfaces.Mixture p(nspecies=nspecies);
    protected
      SI.AmountOfSubstance moles[nspecies](start=i_moles);
    equation
      der(moles) = p.r;
      p.c = moles/v;
    end Volume;
    ...
  end Basic;
end Chemistry;
```

The total volume is represented by the `parameter` `v`. Internally, the `Volume` models uses the variable `moles` to keep track of the total number of moles of each constituent present in the *control volume*. The change in the total number of moles of each constituent is computed from the flow through the `connector`.

In some cases, we will wish to hold the concentration of a particular constituent constant. In this case, we require a model which can add or remove the number of moles necessary to keep the concentration fixed. The following model describes the required behavior:

```
package Chemistry
  ...
  package Basic
    ...
    model Stationary "Stationary concentration"
      parameter Integer nspecies;
      parameter Integer stat_species;
      parameter Modelica.SIunits.Concentration c;
      Interfaces.Mixture p(nspecies=nspecies);
    protected
      Types.MolarFlowRate r;
    equation
      p.c[stat_species] = c;
      for i in 1:nspecies loop
        p.r[i] = if i==stat_species then r else 0.0;
      end for;
    end Stationary;
    ...
  end Basic;
end Chemistry;
```

The variable `r` in this model represents the flow necessary to keep the concentration of a particular constituent (identified by the `stat_species` index) constant at a value of `c`.

6.4.4.3 Chemical reactions

Finally, let us look at how to model reactions which are the primary source of dynamics in a chemical system. Reactions are the result of different kinds of molecules bumping into each other. When these collisions occur, the elements sometimes rearrange themselves into new molecules. The frequency of such transformations is dependent on the availability of the reactants (the initial molecules) and their kinetic energy.

Assume we have a reaction of the form:

$$aA + bB + cC \quad \overset{\rightarrow}{_k} \quad \text{products} \tag{6.24}$$

where a molecules of A react with b molecules of B and c molecules of C.

In this section, we present a simple chemical reaction model. To keep things simple, we have ignored the temperature dependency of the reaction coefficient, k, and we make the simplifying assumption that we can compute the order of the reaction based on the stoichiometry (a reasonable assumption if the concentrations of the reactants are low, see Pauling, 1988). Based on these assumptions, we can use the following simple equation to compute the rate of the reaction, r:

$$r \quad = \quad -k[A]^a[B]^b[C]^c \tag{6.25}$$

Typically, the units of k are such that r will have units of moles per cubic meter per second. The value for r calculated from Equation (6.25) **will always be negative**. As a result of our `connector` definition, it is necessary to compute the molar flow rate of each constituent. We do this by multiplying the reaction rate, r, by the volume in which the reaction is occurring and the number of molecules of the reactant participating in the reaction. As a result, the rate at which molecules of A are converted into products is expressed as:

$$\frac{dA}{dt} = arV \tag{6.26}$$

where a is the number of A molecules participating in the reaction and V is the volume in which the reaction is taking place. For products, we use the same equation but the sign of the equation is changed since products are produced by reactions (remember, r is negative).

For each reaction we can write a vector equation that relates the reaction rate to the rate of change in the number of moles of each constituent. For example, consider the following reaction:

$$A + X \quad \overset{\rightarrow}{_{k_{ax}}} \quad 2X + 2Z \tag{6.27}$$

This reaction is particularly interesting because X appears on both sides of the reaction (*i.e.*, as both a reactant and a product). The rate for this reaction would be computed as:

$$r = -k_{ax}[\mathrm{A}][\mathrm{X}] \tag{6.28}$$

Vectorizing Equation (6.26) gives us the following equation for the total change in the number of moles of each constituent:

$$\frac{d}{dt}\begin{Bmatrix} A \\ B \\ P \\ X \\ Y \\ Z \end{Bmatrix} = \vec{m}rV \tag{6.29}$$

where $\vec{m}$ is computed based on the number of molecules of each constituent participating as reactant and product. For the reaction shown in Equation (6.27), $\vec{m}$ is computed as follows:

$$\vec{m} = \begin{Bmatrix} m_A \\ m_B \\ m_P \\ m_X \\ m_Y \\ m_Z \end{Bmatrix} = \underbrace{\begin{Bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{Bmatrix}}_{\text{reactants}} - \underbrace{\begin{Bmatrix} 0 \\ 0 \\ 0 \\ 2 \\ 0 \\ 2 \end{Bmatrix}}_{\text{products}} = \begin{Bmatrix} 1 \\ 0 \\ 0 \\ -1 \\ 0 \\ -2 \end{Bmatrix} \tag{6.30}$$

Note the sign convention used. Since $\vec{m}$ represents the number of moles consumed in the reaction, the reactant contributions are positive while the product contributions are negative.

We use the `CalcMultiplier` function to compute $\vec{m}$ from Equation (6.29). As we have seen, when $\vec{m}$ is multiplied by the reaction rate, computed by the `CalcReactionRate` function, the result is the rate of change in the number of moles for each constituent within the control volume. This formulation results in the following reaction model:

```
package Chemistry
  ...
  package Basic
    ...
    partial model Reaction
      parameter Modelica.SIunits.Volume v=0.001;
      parameter Integer nspecies;
      Interfaces.Mixture p(nspecies=nspecies);
      parameter Real k "Reaction coefficient";
```

```
    parameter Integer reactants[:,2];
    parameter Integer products[:,2];
  protected
    Types.MolarFlowRate reaction_rate "Reaction rate";
    parameter Real mult[nspecies] =
      Chemistry.Functions.CalcMultiplier(nspecies,
        reactants, products);
  equation
    reaction_rate =
       Chemistry.Functions.CalcReactionRate(nspecies,
       k, p.c, reactants);
    p.r = mult*reaction_rate*v;
  end Reaction;
 end Basic;
end Chemistry;
```

The `Reaction` model contains several interesting constructions. First, the public parameters `reactants` and `products` are both two dimensional arrays with the number of rows unspecified (indicated by the ':') and the number of columns fixed at 2. Each row of the `reactants` and `products` arrays represents a constituent (either a reactant or a product, respectively) in a reaction. The first column is the number of moles of that constituent present in the reaction and the second column is the unique index for that constituent.[6]

Internally, the model declares a **`protected`** parameter named `mult` which represents the level to which each constituent participates in the reaction (*i.e.*, $\vec{m}$, the result of the `CalcMultiplier` function call). This participation is just the balance of the number of moles of a particular constituent present as a reactant minus the number of moles present as a product. An interesting thing to note about the `mult` parameter is that since the arguments to `CalcMultiplier` are parameters[7], it is sufficient to call `CalcMultiplier` only once at the start of the simulation.

Finally, the function to calculate the reaction rate, `CalcReactionRate`, is invoked continuously during the simulation. This function computes the reaction rate based on the concentrations of the reactants and the reaction coefficient, `k`.

The `CalcMultiplier` and `CalcReactionRate` are defined as follows:

```
package Chemistry
  ...
  package Functions
```

[6]The examples that follow should help make this concept of a unique index clearer.

[7]This is a requirement in this case, since a **`parameter`**, which is fixed in time, cannot be computed from quantities that are time-varying. In other words, since `mult` is a parameter (*i.e.*, it does not vary with time), it must be computed from quantities which do not vary with time.

```
    function CalcReactionRate
      input Integer nspecies "Number of species";
      input Real k "Reaction coefficient";
      input Real c[nspecies] "Species concentrations";
      input Integer reactants[:,2] "Reactant information";
      output Real rate "Reaction rate";
    algorithm
      // Compute rate=k*[A]^a*[B]^b...
      rate = k;
      for i in 1:size(reactants,1) loop
        rate := rate*c[reactants[i,2]]^reactants[i,1];
      end for;
      assert(rate>=-1e-12,
         "Error: chemical reaction moving backward");
    end CalcReactionRate;
    function CalcMultiplier
      input Integer nspecies "Number of species";
      input Integer reactants[:,2] "Reactant information";
      input Integer products[:,2] "Product information";
      output Real m[nspecies] "Multiplier";
    algorithm
      m := zeros(nspecies);
      m[reactants[:,2]] := reactants[:,1];
      m[products[:,2]] := m[products[:,2]]-products[:,1];
    end CalcMultiplier;
  end Functions;
  ...
end Chemistry;
```

6.4.5 The Oregonator model

In order to understand how the `Chemistry` package should be used, we include an example which models the reactions in Equations (6.15)-(6.19). For this we will develop a separate **package**, called `Oregonator`, that contains all the details of the Oregonator model. We start by identifying the constituents as follows:

```
package Oregonator
  constant Integer A=1 "BrO3(-)";
  constant Integer B=2 "Organic Species";
  constant Integer P=3 "HOBr";
  constant Integer X=4 "HBrO2";
  constant Integer Y=5 "Br(-)";
  constant Integer Z=6 "Ce4+";
  constant Integer nspecies=6;
  ...
end Oregonator;
```

Next, we must create models for each of the reactions. Rather than include each of these models, we will include only one. It is trivial to see how the other reactions would be defined. As an example, the reaction which transforms [A] and [Y] into [X] and [P] is defined as:

```
package Oregonator
  ...
  package Reactions
    model R_AY
      parameter Reak k_AY=1.0;
      extends Chemistry.Basic.Reaction(k=k_AY,
            reactants={{1,A},{1,Y}},
            products={{1,X},{1,P}});
    end R_AY;
    ...
  end Reactions;
end Oregonator;
```

Now that we have all the building blocks, the complete system can be constructed as follows:

```
package Oregonator
  ...
  model ChemicalSystem
    Chemistry.Basic.Volume v(nspecies=nspecies,v=1,
                      moles(start=fill(1,nspecies)));
    Reactions.R_AY r_ay(nspecies=nspecies);
    Reactions.R_XY r_xy(nspecies=nspecies);
    Reactions.R_AX r_ax(nspecies=nspecies);
    Reactions.R_XX r_xx(nspecies=nspecies);
    Reactions.R_BZ r_bz(nspecies=nspecies);
    Chemistry.Basic.Stationary c_A(stat_species=A,
            nspecies=nspecies, c=1.0);
    Chemistry.Basic.Stationary c_B(stat_species=B,
            nspecies=nspecies, c=1.0);
  equation
    connect(v.p,r_ay.p);
    connect(v.p,r_xy.p);
    connect(v.p,r_ax.p);
    connect(v.p,r_xx.p);
    connect(v.p,r_bz.p);
    connect(v.p,c_A.p);
    connect(v.p,c_B.p);
  end ChemicalSystem;
end Oregonator;
```

The results from simulating `Oregonator.ChemicalSystem` are shown in Figure 6.10. The results show several oscillations. Each oscillation is characterized by an initial rise in both [Z] and [X]. A rise in [Y] is initially prevented because of the abundance of [X] which reacts with [Y] to produce

[P]. Once the production of [X] becomes limited, [Y] rises. This production of [Y] causes a decline in [Z]. Eventually, enough [X] is produced to consume the remaining [Y] and the cycle begins again.

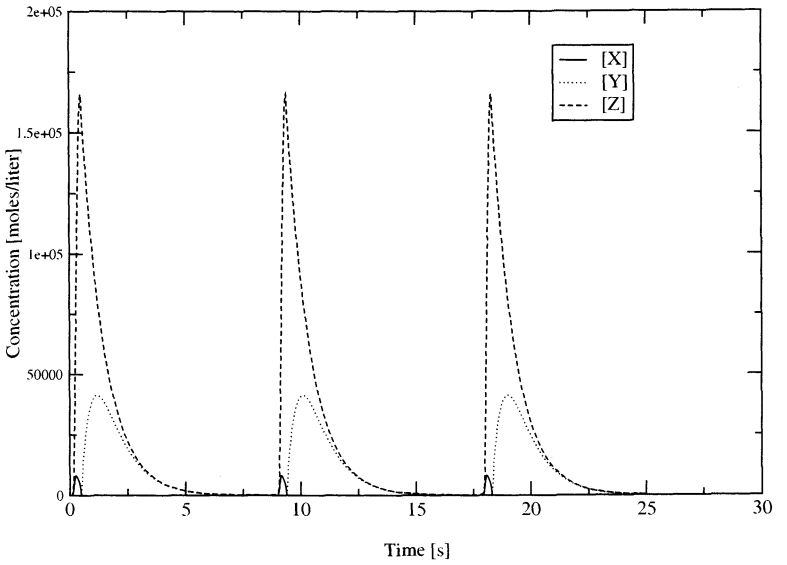

Figure 6.10. Oscillatory response from the Oregonator reaction.

6.5 LANGUAGE FUNDAMENTALS

6.5.1 Information hiding

In this chapter we have shown several uses of the `protected` keyword. If you have parameters or variables which you wish to hide from users of your model you can place them in a `protected` section. The obvious question is then, "Why would I want to hide things"?

The first reason is that internals of the model (*e.g.*, parameters and variables) contained within a `protected` section cannot be referenced externally. This allows the model developer the freedom to change some of the implementation details at some later time without fear of "breaking" any existing models that relied on the original model.

The second reason is that it is not necessary for users of the model to be aware of all of the internal details. By hiding the details of the model, the interface of the model (the publicly accessible portion) is simplified. This makes the model simpler and easier for others to use.

The drawback of making declarations `protected` is that external modifications are not possible. For example, consider a model with an internal variable

whose solution is determined by a differential equation. The `start` attribute for the internal variable cannot be modified externally (*i.e.*, modifications can only be made by the model that contains the variable or by a derived model, as we saw in the `Body` model in Example 6.3). This can make it difficult to control the initial state of the entire system since that variable cannot be modified. Of course, putting the variable in a `protected` section is still a good idea if a change in implementation (*e.g.*, one that would eliminate the variable) is likely, since it prevents users of the model from relying on the presence of that variable.

6.5.2 Arrays

As shown in this chapter, arrays in Modelica can be useful in solving many kinds of problems. In addition to creating arrays of variables (as in Section 6.3.2), it is possible to declare arrays of connectors and subcomponents as well (as shown in Section 6.3.3). In this section we will review the functionality presented in this chapter and present additional details not covered by the examples.

6.5.2.1 Arrays of scalars

Arrays of scalars are the simplest example of array usage in Modelica. For example, declaring an array, `x`, of 5 `Real` variables is done as follows:

```
Real x[5];
```

In some cases we might wish to allow a `parameter` to govern the size of the array. In that case we would do something like:

```
parameter Integer x_size=5;
Real x[x_size];
```

In yet other cases, we might wish to leave the size of the array unspecified and let an initializer determine the size. In that case the array declaration would look something like:

```
model Beam
  parameter Real x[:];
  ...
end Beam;
```

Later, when an instance of a `Beam` is declared we can initialize `x` by writing:

```
Beam b(x={0.2, 0.77, 0.92});
```

Within the `Beam` model, if we wish to know how big the `x` array is, after the initialization, we can use the `size()` function as follows:

```
model Beam
  parameter Real x[:];
  ..
equation
  for i in 1:size(x,1) loop
    ...
  end for;
end Beam;
```

The first argument to the `size()` is the array we are interested in and the second argument indicates which dimension we are interested in. In this case, `x` only has one dimension.

Arrays can be initialized in several ways, as the following code fragment shows:

```
parameter Real x[5]={0.1,0.3,0.5,0.7,0.9};
parameter Real y[:]=x;
parameter Real z[:]=0.1:0.2:0.9;
parameter Integer evens[:]=2:2:10;
```

Array `x` is initialized directly from an explicit array. The size of `y` is left unspecified in the declaration but then the initialization establishes the size as 5 because the values are copied from `x`.

In the case of `z`, the array is constructed by starting with the number .1 and incrementing by .2 until the value exceeds .9 which means that `z` will have the same values as `y` and `x`. The array construction syntax also works in the same way with integers, as can be seen in the initialization of `evens` which creates the array `{2,4,6,8,10}`. In fact, this is the most common form of such constructions and is often used in conjunction with `for` loops. If no increment value is given (*i.e.*, there are only two numbers given with a semicolon in between), it is assumed that the increment is 1 for both `Integer` and `Real` cases.

An important point to make regarding array expressions is that there is no difference between:

```
x = {1, 2, 3, 4, 5}
```

and

```
x = 1:5
```

Likewise, there is no difference between:

```
for i in {1, 2, 3, 4, 5} loop
  ...
end for;
```

and

```
for i in 1:5 loop
  ...
end for;
```

Because the loop is performed over the elements of an array, loops can be constructed over non-contiguous or non-sequential indices, for example:

```
for i in {1, 3, 2, 5, 4, 7, 9, 8, 6} loop
  ...
end for;
```

6.5.2.2 Arrays and attributes

Although we have discussed how to declare and initialize an array of scalars, there is still the issue of how to initialize array attributes. For example, we can declare an array as:

```
Real x[5];
```

But, what if we would like to set the `start` attribute for each of these five elements? Just as `x` is an array, the `start` attribute is also an array.[8] Therefore, the `start` attribute could be initialized as follows:

```
Real x[5](start={0.1,0.2,0.3,0.4,0.5});
```

6.5.2.3 Arrays of components

As we have seen in several examples, the Modelica syntax allows us to declare arrays of components. Such arrays can be useful because they provide increased flexibility for applying constitutive equations to a large number of variables. In all of the examples shown with arrays of components, each component in the array was initialized using the same parameter value. This is often the case and easily accomplished. However, there are cases where it is useful to initialize each component in the array with a unique parameter value. Unfortunately, the Modelica language specification does not completely specify how this can be accomplished.[9]

6.5.2.4 Multi-dimensional arrays

Most of the examples contain arrays with only a single dimension. Such arrays are used primarily to represent mathematical vectors. Arrays with more than a single dimension (*e.g.*, representing matrices) are also possible. The `Reaction` model in Section 6.4.4 demonstrated how to create multi-

[8] These are the semantics in version 1.4 of the Modelica language semantics. However, there are some problems with this syntax and newer versions of the semantics may be slightly different.

[9] It is currently an issue being discussed by the Modelica Association.

dimensional arrays in Modelica. When declaring multi-dimensional arrays, each dimension of the array must be separated by a comma. For example:

```
Real x[5,2,7,8,12];
```

While most models use either 1, 2, or 3 dimensional arrays, there is no limit imposed by the Modelica language on the dimensionality of arrays.

In some cases, a `type` may define the dimensionality of an array. For example:

```
type Point=Real[3];
```

In such cases, an array of that type such as

```
Point particles[12];
```

creates an array with the same shape (*i.e.*, number of rows and columns) as:

```
Real particles[12,3];
```

Another issue with multi-dimensional arrays is initialization. To initialize a multi-dimensional array from a set of literal values, an array of the appropriate shape must be constructed. For example,

```
Real x[2,3] = {{1,2,3},{4,5,6}};
```

Note that the first index represents the "outer" array (*i.e.*, the rows of a two dimensional array) and the second index represents the "inner" array (*i.e.*, the columns of a two dimensional array). In this way, a three-dimensional array could be initialized as follows:

```
Real y[2,3,4] = {{{1,2,3,4},{5,6,7,8},{9,10,11,12}},
                 {{12,11,10,9},{8,7,6,5},{4,3,2,1}}};
```

As mentioned previously, arrays can be constructed by choosing an interval and an increment value. So, the following two initializations are equivalent:

```
Real x[2,3] = {{1,2,3},{4,5,6}};
Real z[2,3] = {1:3,4:6};
```

6.5.3 Looping and equations

In this chapter, we have seen how looping can be used to generate sets of equations. We discussed looping earlier in Section 5.7.5 but the focus then was on algorithms. In this section, we will focus on the special implications of using `for` within an `equation` section as opposed to an `algorithm` section. While `for` loops can be convenient in an `equation` section, they are not always necessary. For instance, as we can see in Example 6.3, it is not necessary to write explicit loops because implicit ones are generated when working with arrays.

The important thing to remember about looping in an `equation` section is that the statements contained within the loop are **equations**, not assignments. For example, the following code fragments have quite different meanings:

```
equation
  var = 0;
  for i in 1:4 loop
    var = var*x+i;
  end loop;

algorithm
  var := 0
  for i in 1:4 loop
    var := var*x+i;
  end loop;
```

When the `for` loop appears in an `equation` section, the following 5 equations are generated:

```
equation
  var = 0;
  var = var*x+1;
  var = var*x+2;
  var = var*x+3;
  var = var*x+4;
```

Note that these equations are not linearly independent (*i.e.,* they are singular). On the other hand, when the `for` loop occurs within an `algorithm` section it generates the following assignments:

```
algorithm
  var := 0;
  var := var*x+1;
  var := var*x+2;
  var := var*x+3;
  var := var*x+4;
```

the net effect of these assignments is equivalent to:

```
algorithm
  var := 5+x*(4+x*(3+x*(2+x*1)));
```

The fact that this assignment was carried out in five separate steps is no different than if it had been carried out in one.

6.5.4 Advanced array manipulation features

6.5.4.1 MATLAB compatibility

Although the examples in this chapter have focused on basic array manipulation techniques, Modelica also includes many advanced array manipulation

features. Modelica shares many of the same features and, in general, the same syntax for array manipulation as MATLAB.[10]

6.5.4.2 Array construction and concatenation

For example, matrices can be created using the same syntax that is used in MATLAB, *i.e.*,

```
Real x[2,3] = [1,2,3;4,5,6];
```

The fact that the expressions are contained between the "[" and "]" characters indicates that this is a matrix construction. Within such matrix construction expressions, a "," indicates the construction is proceeding to the next column (*i.e.*, the second dimension) and the ";" indicates the construction is proceeding to the next row. In this way, matrices can be constructed by concatenating matrices, vectors and scalars.

6.5.4.3 Array subsets

We saw in the `CalcMultiplier` function, defined in Section 6.4.4.3, the following array shorthand:

```
m[products[:,2]] := m[products[:,2]]-products[:,1];
```

If the dimensions of each array at the location of the ":" are equal, then such equations represent relationships between subsets of matrices. For example, this equation could have been written more explicitly as:

```
for i in 1:size(products,1) loop
  m[products[i,2]] := m[products[i,2]]-products[i,1];
end for;
```

Similar types of equations can be written that specify specific elements. For example, the following is also equivalent to the previous two code fragments:

```
n = size(products,1);
m[products[1:n,2]] := m[products[1:n,2]]-products[1:n,1];
```

Remember that "`1:n`" expands to a vector containing every integer value between `1` and `n`, inclusively.

6.5.4.4 Vectorizing of functions

The semantics of Modelica are designed so that it is not necessary to create special vectorized forms of functions. Instead, the normal form of the function can still be used. For example:

[10]MATLAB is a registered trademark of The MathWorks, Inc.

```
sqrt({1, 2, 3});
```

is equivalent to:

```
{sqrt(1), sqrt(2), sqrt(3)};
```

In this way, even though `sqrt()` was defined to take a scalar argument, it can be applied element-wise to an array.

The general rule for taking advantage of this functionality is that the dimensionality of one or more of the arguments to a function can be given additional dimensions. However, all arguments that are given additional dimensions must have the same size in each additional dimension. For example, the following is legal:

```
mod({10,20,30},{4,5,6});
```

and yields:

```
{mod(10,4), mod(20,5), mod(30,6)};
```

Furthermore, this is also legal:

```
mod({10,20,30},4);
```

because only one argument was expanded and it is equivalent to:

```
{mod(10,4), mod(20,4), mod(30,4)};
```

On the other hand, this is not legal:

```
mod({10,20,30},{4,5});
```

because the additional dimensions are not the same size.

6.5.4.5 Mathematical operators

The mathematical operators such as "+" and "*" are frequently used with scalars, but can also be used with arrays. For example, the "+" and "-" operators can be used to add and subtract arrays that have the same size in each dimension. Furthermore, the "*" can be used with arrays in several ways.

The simplest example of using the "*" with arrays is the combination of multiplying a scalar by an array. Each element of the resulting array is equal to the product of the scalar and the corresponding element in the array being multiplied. Another example would be to use the "*" to take the inner product of two vectors of the same size. In other words, the following code fragment:

```
  Real u[5], v[5];
  Real s;
equation
  s = u*v;
```

is equivalent to:

```
  Real u[5], v[5];
  Real s;
algorithm
  s := 0;
  for i in 1:size(u,1) loop
    s = s + u[i]*v[i];
  end for;
```

More complex examples are also possible. For example, the "*" can be used to represent the product of any two arrays as long as the sizes are mathematically compatible. For example, the following shorthand:

```
  Real A[5,7], u[5], v[7];
equation
  u = A*v;
```

is equivalent to:

```
  Real A[5,7], u[5], v[7];
algorithm
  for i in 1:5 loop
    u[i] := 0;
    for j in 1:7 loop
      u[i] = u[i] + A[i,j]*v[j];
    end for;
  end for;
```

Another example is that the following:

```
  Real A[5,7], u[5], v[7];
equation
  v = u*A;
```

is equivalent to:

```
  Real A[5,7], u[5], v[7];
algorithm
  for j in 1:7 loop
    v[j] := 0;
    for i in 1:5 loop
      v[j] = v[j] + u[i]*A[i,j];
    end for;
  end for;
```

Taking the matrix product of two matrices is also possible, as in:

```
  Real A[5,3], B[3,7];
  Real C[5,7];
equation
  C = A*B;
```

which is equivalent to:

```
  Real A[5,3], B[3,7];
  Real C[5,7];
algorithm
  C := fill(0,5,7);
  for i in 1:size(A,1) loop
    for j in 1:size(B,2) loop
      for k in 1:size(A,2) loop
        C[i,j] = C[i,j] + A[i,k]*B[k,j];
      end for;
    end for;
  end for;
```

One final trick that can be very useful (*e.g.*, in formulating transfer functions) is to compute an array containing:

$$\left\{x, \dot{x}, \ddot{x}, ..., \frac{d^n}{dt^n}x\right\} \tag{6.31}$$

We can do this with the following code fragment:

```
  Real x;
  Real dx[5];
equation
  x = Modelica.Math.Sin(time);
  dx[1] = der(x);
  dx[2:5] = der(dx[1:4]);
```

In this way, we can construct a vector, `dx`, such that `dx[i]` represents the i^{th} derivative of `x`.

6.5.5 Built-in functions for arrays

Table 6.1 contains several of the built-in functions for manipulating arrays in Modelica. Full details of these functions (and others not described) can be found in the Modelica language specification.

6.6 PROBLEMS

PROBLEM 6.1 *Extend the `BinarySystem` model so that the total energy and momentum of the system is computed and make sure that it remains constant throughout the simulation.*

PROBLEM 6.2 *Using the material presented in Section 6.2, create a model of the solar system using the information provided in Table 6.2.*

PROBLEM 6.3 *Create a model to solve the hyperbolic PDE:*

$$\frac{d^2u}{dt^2} = c\frac{d^2u}{dx^2} \tag{6.32}$$

Function name	Purpose
`cross(x,y)`	Returns the cross product of the `x` and `y` vectors. The size of both vectors must be 3.
`diagonal(v)`	Generates a square matrix with the elements of `v` on the diagonal.
`fill(s,n1,n2,...)`	Generates an array of size `n1`×`n2`×`...` and fills it with the value `s`.
`identity(n)`	Returns an nxn identity matrix.
`linspace(x1,x2,n)`	Linearly interpolate `n` evenly spaced points along a line between `x1` and `x2`
`matrix(A)`	Similar to `vector(A)` except the size of two dimensions must be greater than 1.
`max(A)`	Returns the largest element of `A`.
`min(A)`	Returns the smallest element of `A`.
`ndims(A)`	Returns the number of dimensions `A` has.
`ones(n)`	Generates an array of length `n` and fills it with the value 1.0.
`outerProduct(v1,v2)`	Returns the outer product of `v1` and `v2`.
`product(A)`	Returns the product of all elements of `A`.
`scalar(A)`	Assuming `size(A,i)==1` for $1 \leq i \leq$ `ndims(A)`, `scalar(A)` returns the single element of A.
`size(A)`	Returns a vector containing the size for each dimension of `A`.
`size(A,i)`	Returns the size of dimension `i` in array `A`.
`skew(x)`	Returns the 3x3 skew matrix for `x` where `size(x,1)==3`.
`sum(A)`	Returns the sum of all elements of `A`.
`symmetric(A)`	Returns a matrix where the upper triangular elements of `A` are copied to the lower triangular portion.
`transpose(A)`	Permutes the first two dimensions of `A`.
`vector(A)`	If `A` is a scalar, `vector(A)` returns a vector with `A` as the only element. If `A` is an array, it must have only one dimension with a size greater than 1 and that dimension is extracted as a vector.
`zeros(n)`	Generates an array of length `n` and fills it with the value 0.0.

Table 6.1. Built-in functions for arrays in Modelica.

Object	Mass (kg)	Distance from Sun (m)	Tangential Velocity (m/s)	Radius (m)
Sun	$1.989 \cdot 10^{30}$	0	0	$1.39 \cdot 10^{9}$
Mercury	$3.303 \cdot 10^{23}$	$69.82 \cdot 10^{9}$	$38.03 \cdot 10^{3}$	$4.879 \cdot 10^{6}$
Venus	$4.869 \cdot 10^{24}$	$108.92 \cdot 10^{9}$	$34.79 \cdot 10^{3}$	$12.10 \cdot 10^{6}$
Earth	$5.976 \cdot 10^{24}$	$152.10 \cdot 10^{9}$	$29.29 \cdot 10^{3}$	$12.74 \cdot 10^{6}$
Mars	$6.421 \cdot 10^{23}$	$249.2 \cdot 10^{9}$	$21.87 \cdot 10^{3}$	$6.780 \cdot 10^{6}$
Jupiter	$1.900 \cdot 10^{27}$	$816.4 \cdot 10^{9}$	$12.42 \cdot 10^{3}$	$139.8 \cdot 10^{6}$
Saturn	$5.688 \cdot 10^{26}$	$1.510 \cdot 10^{12}$	$9.11 \cdot 10^{3}$	$116.4 \cdot 10^{6}$
Uranus	$8.686 \cdot 10^{25}$	$3.001 \cdot 10^{12}$	$6.49 \cdot 10^{3}$	$50.72 \cdot 10^{6}$
Neptune	$1.024 \cdot 10^{26}$	$4.555 \cdot 10^{12}$	$5.38 \cdot 10^{3}$	$49.24 \cdot 10^{6}$
Pluto	$1.270 \cdot 10^{22}$	$7.358 \cdot 10^{12}$	$3.58 \cdot 10^{3}$	$2.390 \cdot 10^{6}$

Table 6.2. Solar system data.

You may choose to use the following spatial approximation:

$$\frac{d^2u}{dx^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} \tag{6.33}$$

PROBLEM 6.4 *Create a model for a "collision force" model between two bodies such that when they are in contact (i.e., the total distance between the centers of the bodies is less than the sum of their radii) they generate a repelling force as follows:*

$$F = c(r - r_1 - r_2) \tag{6.34}$$

where c is a large "stiffness" coefficient, r is the distance between the centers of the bodies, r_1 is the radius of body 1 and r_2 is the radius of body 2.

Next, create a "pool table" model with several billiard balls on it. Only one ball should have an initial velocity. Position the balls so that at least two collisions take place. Using Dymola, you can declare a `Sphere` *for each body so that the collisions can be animated (see Example 9.2 for an example).*

Chapter 7

HYBRID MODELS

7.1 CONCEPTS

Up to this point, we have been discussing systems of equations involving continuous variables. In this chapter, we will discuss *hybrid system* behavior. Hybrid behavior involves not just continuous variables and equations, but also piecewise continuous variables with discontinuities and *discrete variables* which have values that are piecewise constant with respect to time (*e.g.*, an `Integer` or `Boolean`). This chapter presents Modelica functionality used to describe such hybrid behavior.

7.2 MODELING DIGITAL CIRCUITS

Hybrid models are a combination of both continuous and discrete behavior. Before mixing the two, let us introduce a few examples which highlight discrete behavior by itself. Digital circuits are an excellent example of systems which can be simulated using discrete behavior exclusively. Imagine we wish to construct a model with three inputs and two outputs which behaves according to the "truth table" shown in Table 7.1.

i1	i2	i3	o1	o2
false	false	false	true	false
true	false	false	true	false
false	true	false	true	true
true	true	false	true	false
false	false	true	true	false
true	false	true	false	false
false	true	true	true	true
true	true	true	true	false

Table 7.1. Discrete behavior truth table.

7.2.1 Connectors

For this section, we will rely on connectors from the `Modelica.Blocks.Interfaces` package. The two connectors we are interested in using are the `BooleanInPort` and the `BooleanOutPort`. The definitions for these connectors are essentially:

```
connector BooleanInPort "Boolean Input Port"
  parameter Integer n=1 "Signal vector size";
  input Boolean signal[n] "Signal values";
end BooleanInPort;

connector BooleanOutPort "Boolean Output Port"
  parameter Integer n=1 "Signal vector size";
  output Boolean signal[n] "Signal values";
end BooleanOutPort;
```

These connectors can be used to represent the logical values of the inputs and outputs in a digital circuit.

The signals carried by these connectors are `Boolean`. `Boolean` and `Integer` quantities have discrete values (*i.e.,* they cannot change continuously as a function of time). Because their values are discrete, they jump instantaneously from one value to another.

7.2.2 Components

The behavioral description in Table 7.1 can be represented using the following boolean equations:

$$o1 = \overline{(i1 \text{ AND } i3)} \text{ OR } i2 \tag{7.1}$$

$$o2 = \overline{i1} \text{ AND } i2 \tag{7.2}$$

One way to model such a system would be to write a model which directly implemented these equations, *e.g.,*

```
block LogicEquation
  Modelica.Blocks.Interfaces.BooleanInPort i1(n=1);
  Modelica.Blocks.Interfaces.BooleanInPort i2(n=1);
  Modelica.Blocks.Interfaces.BooleanInPort i3(n=1);
  Modelica.Blocks.Interfaces.BooleanOutPort o1(n=1);
  Modelica.Blocks.Interfaces.BooleanOutPort o2(n=1);
equation
  o1.signal = not (i1.signal and i3.signal) or i2.signal;
  o2.signal = not i1.signal and i2.signal;
end LogicEquation;
```

As we have seen previously, creating a model based on the specific equations for a problem results in a model without much reusability.

Just as we have done before, we want to make a library of reusable components so we can build a variety of logic circuits. For this example, we need an And model, an Or model and a Not model. These models are shown in Examples 7.1-7.3 respectively.

```
block And
  Modelica.Blocks.Interfaces.BooleanInPort inPort1(n=1);
  Modelica.Blocks.Interfaces.BooleanInPort inPort2(n=1);
  Modelica.Blocks.Interfaces.BooleanOutPort outPort(n=1);
equation
  outPort.signal = inPort1.signal and inPort2.signal;
end And;
```

Example 7.1. Model of an "and" gate.

```
block Or
  Modelica.Blocks.Interfaces.BooleanInPort inPort1(n=1);
  Modelica.Blocks.Interfaces.BooleanInPort inPort2(n=1);
  Modelica.Blocks.Interfaces.BooleanOutPort outPort(n=1);
equation
  outPort.signal = inPort1.signal or inPort2.signal;
end Or;
```

Example 7.2. Model of an "or" gate.

```
block Not
  Modelica.Blocks.Interfaces.BooleanInPort inPort(n=1);
  Modelica.Blocks.Interfaces.BooleanOutPort outPort(n=1);
equation
  outPort.signal = not inPort.signal;
end Not;
```

Example 7.3. Model of a "not" gate.

7.2.3 Simple logic circuit

Example 7.4 shows a model which should behave according to Table 7.1. The period of each input signal is such that all possible combinations of inputs are generated every 8 seconds of simulation time. Note that our input signals are generated using the boolean signal generator models from the MSL.

It might be easier to understand Example 7.4 by looking at a diagram of its components and connections as shown in Figure 7.1. The results of running this simulation can be seen in Figure 7.2.

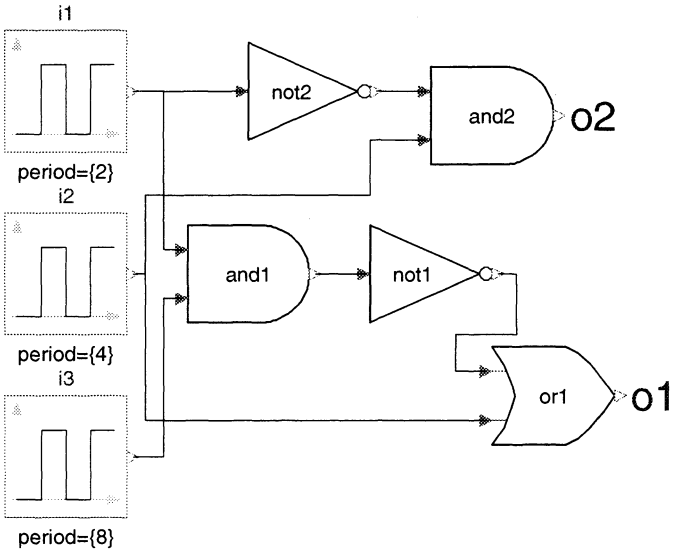

Figure 7.1. Diagram for `LogicCircuit` model in Example 7.4.

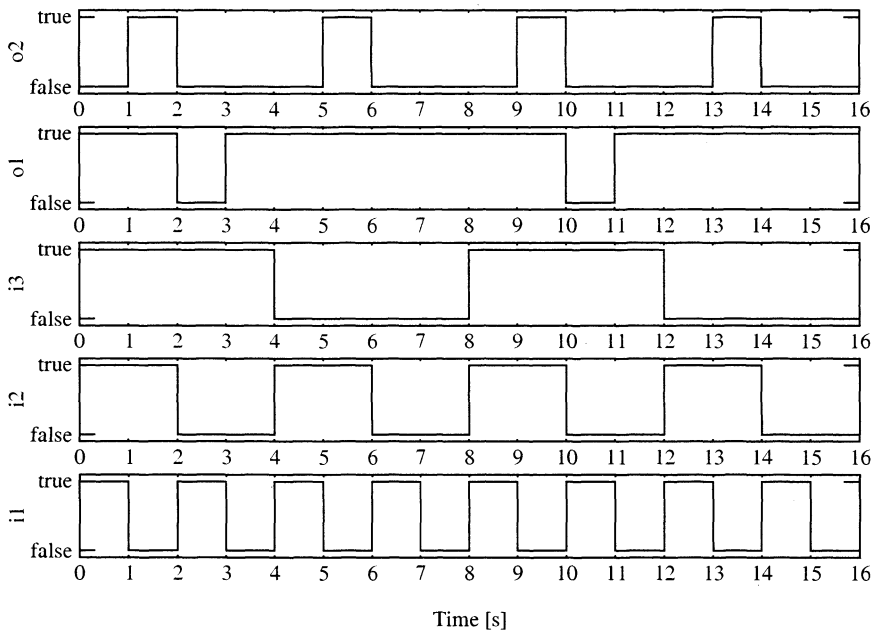

Figure 7.2. Output signals from `LogicCircuit` model shown in Example 7.4.

```
model LogicCircuit
  import BS=Modelica.Blocks.Sources;

  BS.BooleanPulse i1(width={50},period={2});
  BS.BooleanPulse i2(width={50},period={4});
  BS.BooleanPulse i3(width={50},period={8});
  And and1, and2;
  Or or1;
  Not not1, not2;
  Boolean o1, o2;
equation
  // o1
  connect(i1.outPort,and1.inPort1);
  connect(i3.outPort,and1.inPort2);
  connect(and1.outPort,not1.inPort);
  connect(not1.outPort,or1.inPort1);
  connect(i2.outPort,or1.inPort2);
  o1 = or1.outPort.signal[1];
  // o2
  connect(i1.outPort,not2.inPort);
  connect(not2.outPort,and2.inPort1);
  connect(i2.outPort,and2.inPort2);
  o2 = and2.outPort.signal[1];
end LogicCircuit;
```

Example 7.4. Model of a circuit to test `And`, `Or` and `Not`.

7.2.4 Mixing discrete and analog behavior

In our previous circuit, signals propagated through the circuit instantly. Now, let us consider a case which introduces lag (*e.g.,* due to capacitance in the circuit) in the response of the components. To model this, we use the `Lag` block shown in Example 7.5. The parameters to this model are `c`, the time constant of the response, and `threshold`, the analog threshold between `true` and `false`. The output of the `Lag` element depends on whether the continuous response of the `Lag` element is above or below the value of `threshold`.

This is an example of a hybrid model because it mixes analog and discrete behavior. Example 7.6 shows a circuit similar to the one shown in Example 7.4 except that it includes lag in the output of all of the components. A diagram of Example 7.6 is shown in Figure 7.3. Finally, the results of simulating the model with different values for `c` can be seen in Figures 7.4 and 7.5.

```
block Lag
  parameter Real c=1 "lag time constant";
  parameter Real threshold=.7 "logical threshold";
  Modelica.Blocks.Interfaces.BooleanInPort inPort(n=1);
  Modelica.Blocks.Interfaces.BooleanOutPort outPort(n=1);
protected
  Real state "Continuous state of the wire";
equation
  c*der(state) = if inPort.signal[1] then 1-state else -state;
  outPort.signal[1] = state>=threshold;
end Lag;
```

Example 7.5. Modeling lag in a digital signal.

```
model LogicCircuitWithLag
  parameter Real c=1 "lag time constant";
  model Pulse=Modelica.Blocks.Sources.BooleanPulse;
  Pulse i1(period={2});
  Pulse i2(period={4});
  Pulse i3(period={8});
  And and1, and2;
  Or or1;
  Not not1, not2;
  Boolean o1, o2;
  Lag and1_lag(c=c), and2_lag(c=c), or1_lag(c=c),
     not1_lag(c=c), not2_lag(c=c);
equation
  connect(i1.outPort,and1.inPort1);
  connect(i3.outPort,and1.inPort2);
  connect(and1.outPort,and1_lag.inPort);
  connect(and1_lag.outPort,not1.inPort);
  connect(not1.outPort,not1_lag.inPort);
  connect(not1_lag.outPort,or1.inPort1);
  connect(i2.outPort,or1.inPort2);
  connect(or1.outPort,or1_lag.inPort);
  o1 = or1_lag.outPort.signal[1];
  connect(i1.outPort,not2.inPort);
  connect(not2.outPort,not2_lag.inPort);
  connect(not2_lag.outPort,and2.inPort1);
  connect(i2.outPort,and2.inPort2);
  connect(and2.outPort,and2_lag.inPort);
  o2 = and2_lag.outPort.signal[1];
end LogicCircuitWithLag;
```

Example 7.6. Introducing lag into our logic response.

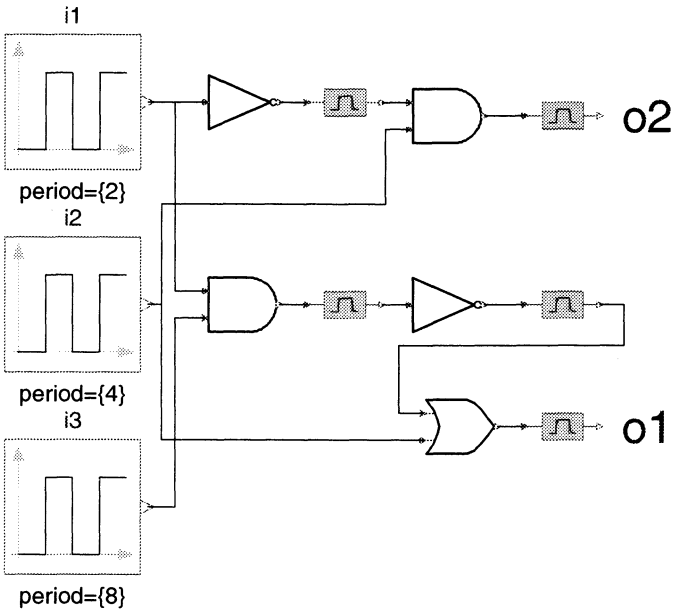

Figure 7.3. Diagram for `LogicCircuitWithLag` model shown in Example 7.6.

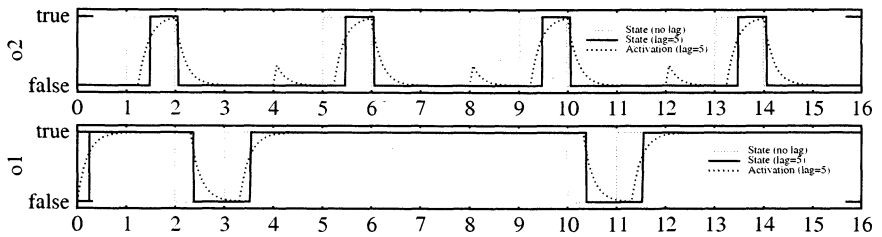

Figure 7.4. Output signals from `LogicCircuitWithLag`, $c = \frac{1}{5}$.

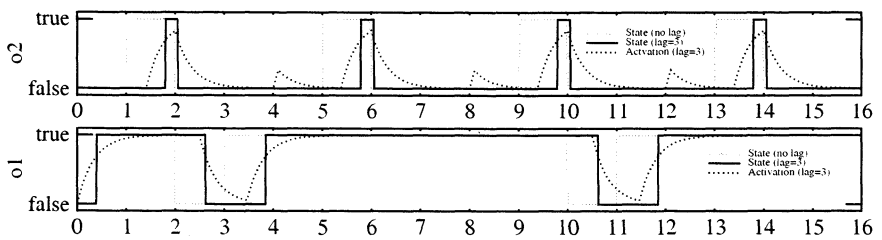

Figure 7.5. Output signals from `LogicCircuitWithLag`, $c = \frac{1}{3}$.

7.3 BOUNCING BALL

Another typical example of hybrid behavior is a bouncing ball. Using Newton's Law, we know that the equation for a falling object is:

$$ma = -mg \tag{7.3}$$

where m is the mass of the object, a is the acceleration of the object and g is the acceleration due to Earth's gravity. These equations are continuous in nature. However, what do we do when the ball actually strikes a surface? There are at least two ways to approach this problem.

The first is to treat the system as completely continuous by using a non-linear spring to model the collision. When the ball comes in contact with a surface, Equation (7.3) is changed to:

$$ma = -mg - c * (h - r) - d * v \tag{7.4}$$

where c is the compliance of the ball, h is the height of the ball's center, r is the radius of the ball, d is the damping coefficient of the ball and v is the velocity of the ball.

An example of this approach can be seen in Example 7.7 which shows a model with only continuous variables and equations. While this model is completely valid, it does have one drawback. It requires a numerical solver to resolve the collision of the ball and the surface. So, for some finite time, during the collision, the solver will be presented with a *stiff* problem.

```
model BouncingBall1
  import Modelica.SIunits;

  parameter SIunits.Mass m=1.0 "Mass of the ball";
  parameter Real c(final unit="N/m")=1e+4 "Compliance";
  parameter Real d(final unit="N/(m.s)")=20 "Damping";
  parameter SIunits.Radius r=0.02 "Radius of the ball";

  SIunits.Height h(start=5.0) "Height of the ball center";
  SIunits.Velocity v "Velocity of the ball";
  SIunits.Acceleration a "Acceleration of the ball";
  SIunits.Force f "Force on the ball";
equation
  v = der(h);
  a = der(v);
  m*a = f-m*Modelica.Constants.g_n;
  f = if h<=r then -c*(h-r)-d*v else 0.0;
end BouncingBall1;
```

Example 7.7. A "continuous" bouncing ball.

The second approach, shown in Example 7.8, is to treat the collision as an instantaneous event. Instead of providing a spring constant and damping coefficient, this model requires a *coefficient of restitution* which is defined as:

$$c_r = -\frac{v_{\text{after}}}{v_{\text{before}}} \tag{7.5}$$

where v_{after} is the velocity of the ball after the collision and v_{before} is the velocity of the ball before the collision.

```
model BouncingBall2
  import Modelica.SIunits;

  parameter SIunits.Mass m=1.0 "Mass of the ball";
  parameter Real c_r=.725 "Coef. of restitution";
  parameter SIunits.Radius r=0.02 "Radius of the ball";

  SIunits.Height h(start=5.0) "Height of the ball center";
  SIunits.Velocity v "Velocity of the ball";
  SIunits.Acceleration a "Acceleration of the ball";
equation
  v = der(h);
  a = der(v);
  m*a = -m*Modelica.Constants.g_n;
  when h<=r then
    reinit(v,-c_r*pre(v));
  end when;
end BouncingBall2;
```

Example 7.8. A "discrete" bouncing ball.

Using the second approach, the `BouncingBall2` model avoids the numerical stiffness present in Example 7.7. Instead of including equations to resolve the contact with the surface, the `BouncingBall2` model makes an instantaneous change to the velocity, based on the coefficient of restitution, at the moment of contact. This approach may allow the simulation to run faster since it will not have to solve the stiff equations associated with the collision. There is a subtle drawback to this model which is not easily demonstrated by this example but which is described along with the backlash examples found in Section 8.3. The results from simulating the `BouncingBall2` model are shown in Figure 7.6.

A practical issue for both of these models is what happens over long periods of time. Mathematically, we can show that the ball stops bouncing in some finite time but bounces an infinite number of times in that interval. Such a system is called a Zeno system. For numerical reasons, it eventually becomes impossible to resolve the motion of the ball. This is because the velocities and

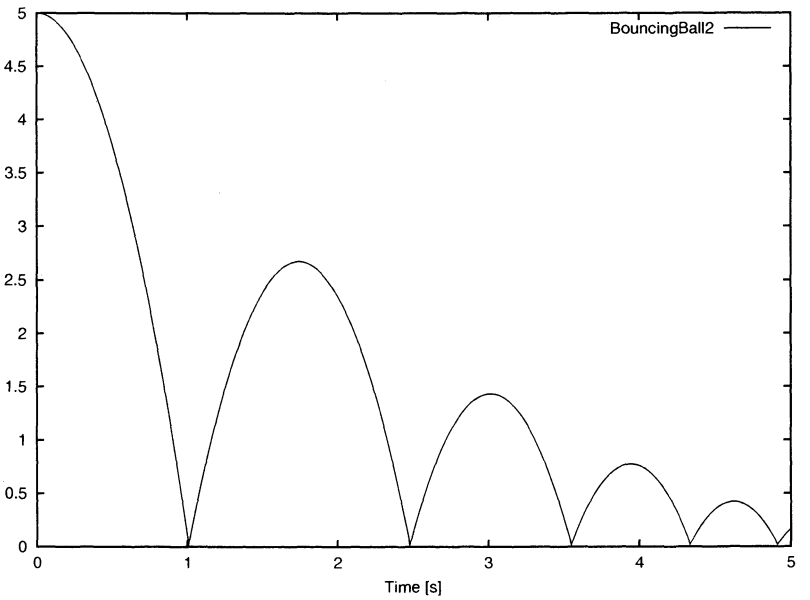

Figure 7.6. Behavior of model `BouncingBall2`.

positions become very small. At some point, we must decide that the ball is no longer bouncing. If we do not do this, large amounts of computational time will be wasted trying to resolve the tiny (and increasingly frequent) collisions. On top of that, numerical errors may result in the ball "falling" below the surface it is bouncing on because we can no longer detect the collisions.

Example 7.9 shows a modified version of the discrete model which recognizes when the ball has essentially stopped bouncing. The difficulties in such models occur because as the height of each successive bounce becomes smaller and smaller, it is possible for the condition `h<=r` to remain true (*i.e.*, the ball is still in contact with the surface) while `v` again becomes negative (*i.e.*, the ball begins settling back into the surface before contact with the surface was broken). For this reason, we wish to distinguish between the case where `impact` becomes true by itself and when the combination of `impact and v<=0` becomes true. The first case represents a real impact while the second case indicates that the ball has begun falling again before leaving the surface.

This example highlights several of the more sophisticated features in hybrid modeling. First, the conditional expression for the **`when`** clause is a vector. When a vector of conditions is provided to a **`when`** clause, the **`when`** clause is activated at the instant any of the conditions becomes true. It is important to note that:

```
model BouncingBall3
  import Modelica.SIunits;

  parameter SIunits.Mass m "Mass of the ball";
  parameter Real c_r "Coefficient of restitution";
  parameter SIunits.Radius r=1e-3 "Radius of the ball";

  SIunits.Height h(start=5.0) "Height of the ball center";
  SIunits.Velocity v "Velocity of the ball";
  SIunits.Acceleration a "Acceleration of the ball";
  Boolean bouncing(start=true)
       "Is the ball to still be bouncing?";
  Boolean impact "Indicates when impact occurs";
equation
  v = der(h);
  a = der(v);
  m*a = if bouncing then -m*Modelica.Constants.g_n else 0;
algorithm
  impact := h<=r;
  when {impact, impact and v<=0} then
    if edge(impact) then
      bouncing := pre(v)<=0;
      reinit(v,-c_r*pre(v));
    else
      reinit(v,0.0);
      bouncing := false;
    end if;
  end when;
end BouncingBall3;
```

Example 7.9. Another "discrete" bouncing ball.

```
when {impact, impact and v<=0} then
  // ...
end when;
```

is equivalent to:

```
when impact then
  // ...
end when;
when impact and v<=0 then
  // ...
end when;
```

In other words, each component in the vector of conditional expressions can be treated as if it were in a separate `when` clause. It is important to point out that the vector form of the `when` clause is **not** equivalent to:

```
when impact or (impact and v<=0) then
  // ...
end when;
```

The important thing to remember is that the vector form of a `when` clause is activated if any of the individual conditional expressions inside the vector become `true`. In the case where the `or` operator is used, the `when` clause will not be activated in response to one condition becoming `true` if the other condition is already `true` since the value of the entire expression would not change in such a case.

The other feature introduced in this example is the use of the `edge()` function inside the `when` clause to determine which of the conditions in the vector of conditional expressions triggered the activation of the `when` clause. The `edge()` function can take a `Boolean` variable as an argument and is defined as follows:

```
edge(b) = b and not pre(b);
```

In other words, "is `b` true now and was it not true before this `when` clause was activated?" In the case of the `BouncingBall3` model, we can use the `edge()` function to determine if impact just occurred or whether we are in the middle of an ongoing collision.

7.4 SENSOR MODELING

7.4.1 Introduction

While there are numerous textbooks on control theory, their emphasis is generally on the mathematical theory behind control system design (*e.g.*, Brogan, 1991) as opposed to some of the practical issues faced when deploying control systems. Oftentimes, the plant models are assumed to be linear and the sensors and actuators are assumed to be ideal. From a teaching perspective, this is desirable because it keeps the focus on the theory of control system design.

In this section, we will examine the effects of using non-ideal sensors by developing several non-ideal sensor models and comparing their performance against the benchmark system shown in Figure 7.7. As we shall see, the behavioral description of non-ideal sensors demonstrates many of the hybrid modeling features in Modelica.

The controller for the system in Figure 7.7 is a simple PI controller. In all variations of the system presented in this section, the plant model will be linear and the actuator will be ideal. The controller performs quite nicely with ideal sensor and actuator models but can easily be driven unstable by non-ideal sensor models. The non-ideal sensor models presented in this section were not implemented simply to torture the existing control system but rather to represent more realistic control system applications. The non-ideal models

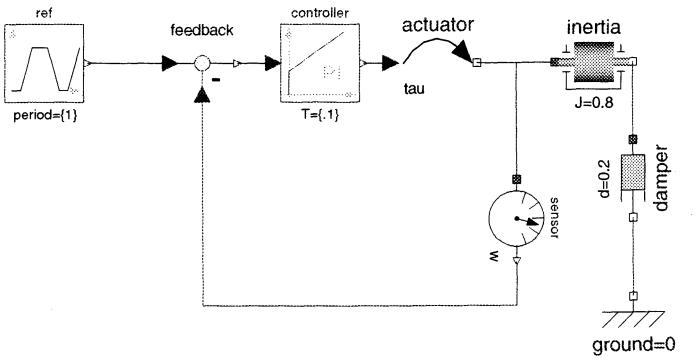

Figure 7.7. Our sensor benchmark system.

ıre quite reasonable and based on common methods for measuring system ɾesponse.

Since the emphasis of this book is on physical modeling, we have tried to present more realistic physical models. In addition to the response of the physical system, it is also important to accurately capture effects of sensors and actuators. In this section, we will examine the effects of placing non-ideal sensor models into our benchmark system using closed-loop control.

7.4.2 Ideal case

As shown in Figure 7.7, a reference signal is fed to our controller to indicate the desired speed of the system. The controller compares the desired speed to the current speed measurement from the sensor and, based on the difference, determines what torque is required from the actuator. For this case, an ideal sensor model is used to generate a baseline for comparison.

One other nice thing about this benchmark system is that it helps to reinforce some of the features discussed in Chapter 4. For example, all the sensor models discussed in this section will be derived from the MSL rotational library definition of an `AbsoluteSensor` shown below:

```
partial model AbsoluteSensor
  package Rotational=Modelica.Mechanics.Rotational;
  Rotational.Interfaces.Flange_a flange_a;
  Modelica.Blocks.Interfaces.OutPort outPort(final n=1);
end AbsoluteSensor;
```

By using this `partial model` in conjunction with the `replaceable` and `redeclare` keywords, we will see shortly that we can easily generate variations on the baseline system shown in Figure 7.7 and modeled by Example 7.10.

```
model SensorBenchmark
  import Modelica.Mechanics.Rotational;
  import Modelica.Blocks;

  Rotational.Inertia inertia(J=0.8);
  Rotational.Fixed ground;
  Rotational.Damper damper(d=0.2);
  Rotational.Torque actuator;
  replaceable Rotational.Sensors.SpeedSensor sensor extends
           Rotational.Interfaces.AbsoluteSensor;
  Blocks.Continuous.PI controller(k={100}, T={0.1});
  Blocks.Math.Feedback feedback;
  Blocks.Sources.Trapezoid ref(offset={50}, rising={0.2},
    width={0.25}, falling={0.2}, amplitude={50});
equation
  connect(ground.flange_b, damper.flange_b);
  connect(damper.flange_a, inertia.flange_b);
  connect(actuator.flange_b, inertia.flange_a);
  connect(sensor.flange_a, inertia.flange_a);
  connect(controller.outPort, actuator.inPort);
  connect(feedback.outPort, controller.inPort);
  connect(sensor.outPort, feedback.inPort2);
  connect(ref.outPort, feedback.inPort1);
end SensorBenchmark;
```

Example 7.10. Source code for our sensor benchmark system.

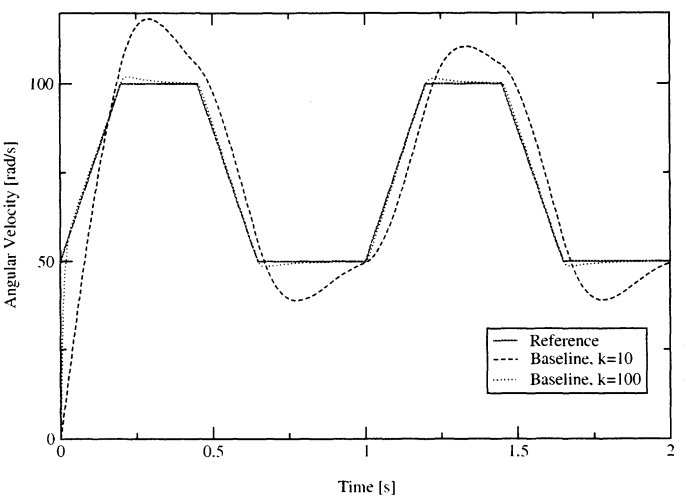

Figure 7.8. Performance of low (k=10) and high (k=100) gain controllers with ideal sensors.

Figure 7.8 shows an example of the performance of our idealized system. The figure includes three curves. The first is the reference signal for the response we are trying to achieve. The next is the response with a low gain ($k = 10$) PI controller. Finally, the performance of a high gain ($k = 100$) controller is shown. As you can see, the high gain controller has no difficulty controlling the system response to closely follow the reference signal.

7.4.3 Sample and hold sensor

7.4.3.1 Behavioral description

```
model SampleHoldSensor
  import Modelica.Mechanics.Rotational;

  extends Rotational.Interfaces.AbsoluteSensor;
  Modelica.SIunits.AngularVelocity w;
  parameter Modelica.SIunits.Time sample_interval=0.1;
equation
  w = der(flange_a.phi);
  flange_a.tau = 0;
algorithm
  when sample(0,sample_interval) then
    outPort.signal[1] := w;
  end when;
end SampleHoldSensor;
```

Example 7.11. Sensor that samples speed measurements.

The first variation on our sensor benchmark will be to utilize a sample and hold (*i.e.*, zero-order hold) sensor. The only difference between this sensor model and the ideal one is that it does not provide continuous updating of shaft speed. Instead, at regular intervals, it outputs the system speed and holds that reading until the next sampling interval. As a result, the output of the sensor is a piecewise constant signal. Example 7.11 contains the source for the `SampleHoldSensor` model.

Something to notice about Example 7.11 is the use of the `when` clause. This `when` clause is used to update the value of the output signal every time the system speed is sampled. The assignment:

```
outPort.signal[1] := w;
```

is only performed at the instant the sampling occurs. The interval between samples is given by the `sample_interval` parameter. The `sample()` function is a built-in function used to generate sampling events. The first argument to `sample()` is the time at which it should generate the first sample

event and the second argument is the time interval between subsequent sample events.

7.4.3.2 Simulation results

To create a simulation using the `SampleHoldSensor` shown in Example 7.11 all we need to do is **`redeclare`** the sensor model from our benchmark case. As described in Chapter 4, this is done as follows:

```
model SamplingCase1
  extends SensorBenchmark(redeclare
    SampleHoldSensor sensor(sample_interval=0.01));
end SamplingCase1;
```

In other words, we create a new model, `SamplingCase1`, which extends our previous model, `SensorBenchmark`, while redeclaring the `sensor` component. This saves us from having to copy and paste the model shown in Figure 7.10 with the only difference being the type of sensor model.[1]

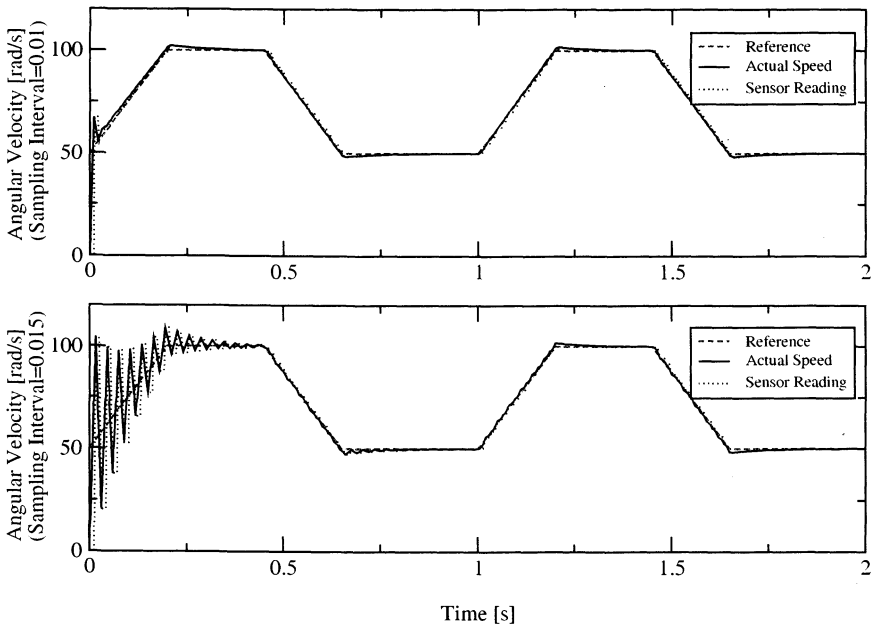

Figure 7.9. Comparison of `SampleHoldSensor` with ideal case.

[1]This kind of copying and pasting is bad because it leads to redundancy which becomes a significant maintenance problem.

Figure 7.9 shows a comparison between the `SampleHoldSensor` model and the ideal case. Two different values for the `sample_interval` parameter are shown.

7.4.3.3 Limitations

The problem with the `SampleHoldSensor` model is that we must carefully choose the `sample_interval` parameter. For Figure 7.9, the values of 0.01 and 0.015 were chosen because they represent a small fraction of the time constant used in the PI controller. However, the upper bound on the `sample_interval` parameter for this problem is approximately 0.015 (shown at the bottom of Figure 7.9). Any sampling interval significantly larger than 0.015 will drive the controller unstable during the initial transients. What we can learn from Figure 7.9 is that this type of sensor performs acceptably for small errors between the reference and actual speed but large errors will drive the system unstable.

7.4.4 Quantization

7.4.4.1 Behavioral description

In addition to sampling data at a specific frequency, data acquisition systems frequently digitize the sensor readings. In these cases, the resolution of the readings is affected by the number of bits used in the digital representation of the signal. This effect is called quantization.

Example 7.12 shows a behavioral description of the `QuantizedSensor` model. The parameters of this model are the number of bits used in the output signal, `bits`, the sampling interval, `sample_interval`, and the minimum and maximum values for the output reading. `min` and `max` respectively. Internally, the model uses an `Integer`, called `level`, with a value between 0 and 2^{bits} to indicate the digital value. The value of `level` is then scaled appropriately for output.

Finally, the actual output signal is scaled based on the number of bits and the range of measured values. If the measured signal falls outside the range of the sensor, the sensor returns either the minimum or maximum value.

7.4.4.2 Simulation results

Once again, we can easily generate a model that uses the `QuantizedSensor` by writing a few lines of Modelica code, *e.g.*,

```
model QuantizedCase1=SensorBenchmark(
  redeclare QuantizedSensor sensor(sample_interval=0.01,
                                   bits=8));
```

Simulation results using the `QuantizedSensor` are shown in Figure 7.10. Since we already know, from Figure 7.9 what the effect of different sampling

```
model QuantizedSensor
  import Modelica.SIunits;
  import Modelica.Mechanics.Rotational;
  extends Rotational.Interfaces.AbsoluteSensor;

  parameter Integer bits=4;
  parameter SIunits.Time sample_interval=0.02;
  parameter SIunits.AngularVelocity min=-150;
  parameter SIunits.AngularVelocity max=150;
  SIunits.AngularVelocity w;
protected
  parameter Real delta=(max-min)/2^bits;
  Integer level;
equation
  w = der(flange_a.phi);
  flange_a.tau = 0;
algorithm
  when sample(0,sample_interval) then
    level := integer((w-min)/delta);
  end when;
  if level<0 then
       outPort.signal[1] := min;
  elseif level>=2^bits then
    outPort.signal[1] := max;
  else
       outPort.signal[1] := level*delta+min;
  end if;
end QuantizedSensor;
```

Example 7.12. Measurement with quantization.

intervals can be, Figure 7.10 includes a comparison for different values of the `bits` parameter.

7.4.4.3 Limitations

In addition to the limits generally associated with a sample and hold sensor, as we saw in Example 7.11, the `QuantizedSensor` is also limited by the number of bits used in sampling and the allowed range of measurements. Not having enough bits results in coarse output and can drive the controller unstable by creating large swings in the sensed values. The range of measurements must be large enough to bound the actual sensed values but not so large as to contribute to coarseness of the output signal.

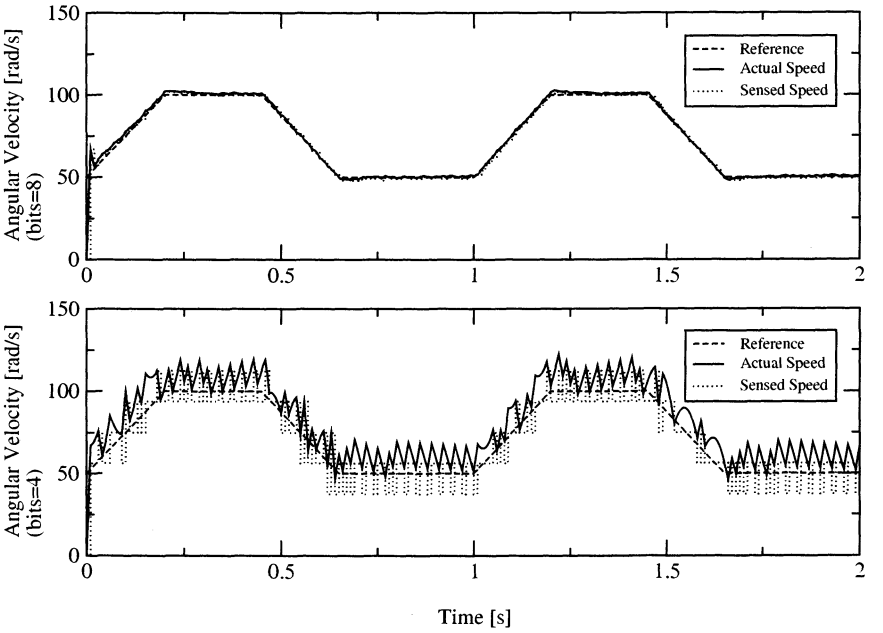

Figure 7.10. Comparison of `QuantizedSensor` with ideal case.

7.4.5 Period measurement sensor

Another way of sensing rotational velocity is to instrument a system with a sensor (*e.g.,* an optical encoder) which reports complete revolutions (or some integer fraction of a complete revolution) and then inferring the velocity from those measurements. By timing the interval between these reports, the rotational speed of the system can be estimated. Example 7.13 shows a model of such a sensor.

7.4.5.1 Behavioral description

The first thing to notice about the `PeriodSensor` model is the use of the **`discrete`** keyword. This keyword is used to indicate that the variables (*i.e.,* `upper` and `lower`) are not continuous variables. The **`discrete`** qualifier indicates that these values are piecewise constant with respect to simulation time. The **`discrete`** keyword is not required but it does have the advantage that any `Real` variable labeled as **`discrete`** must be assigned within a **`when`** clause in an **`algorithm`** section. This allows any changes that treat the variable as continuous to be detected.

The `upper` and `lower` variables represent the point, forward and backward, at which the next interval signal is triggered. Mechanically, this triggering

```
model PeriodSensor
  import Modelica.Mechanics.Rotational;

  extends Rotational.Interfaces.AbsoluteSensor;
  parameter Integer divisions=4;
protected
  parameter Modelica.SIunits.Angle trigger_interval=
    2*Modelica.Constants.pi/divisions;
  discrete Modelica.SIunits.Angle upper, lower;
  Modelica.SIunits.Time last_time;
equation
  flange_a.tau = 0;
algorithm
  when initial() or flange_a.phi>=upper
              or flange_a.phi<=lower then
    upper := flange_a.phi+trigger_interval;
    lower := flange_a.phi-trigger_interval;
    last_time := time;
    outPort.signal[1] := if initial() then 0.0
       else trigger_interval/(time-pre(last_time));
  end when;
end PeriodSensor;
```

Example 7.13. Interval encoding measurement.

usually corresponds to the location of an optical or magnetic sensor mounted on the rotating body. In our model, once that location has been reached, the `upper` and `lower` limits are changed to correspond to the next location.[2]

This example includes the use of the `initial()` function. This function returns a value of `true` only at the instant the simulation starts. As we can see in Example 7.13, the `initial()` function can be used in conditional expressions for **`when`** and **`if`** clauses. Because we have no data at the start of the simulation, we use the `initial()` function to set the initial output from the sensor to zero.

The logic in the `PeriodSensor` model is more complicated than the sensor models presented so far. This model demonstrates how the logical operators (*e.g.*, `or`) can be used in conjunction with a **`when`** clause. As a result, if any of the three conditional expressions becomes true, the body of the **`when`** clause is executed.

[2]For the sake of keeping the `PeriodSensor` model simple, it is assumed that the rotation sensed by the sensor will be in one direction for the duration of the simulation. This is a reasonable assumption for this type of sensor since it would not give accurate readings for the case where the rotational speed oscillated between forward and backward motion (unless a more complicated encoding scheme were used).

The output signal is updated depending on which of the conditions leads to the execution of the `when` clause. For initialization, the output signal is set to zero. In the case of motion, the rotational speed is estimated by dividing the sensor spacing by the time between reports.

To calculate the approximate speed, we must know the time between the last position report and the current position report. The time of the current report is represented by the global simulator variable `time`. At the time of each report, we also record the current time in a variable called `last_time`. The problem we face within the `when` clause is that we wish to use `last_time` in a calculation and then update its value. To make sure we do not accidentally use the updated value of `last_time`, we use the `pre` operator to obtain the previous value for `last_time` (*i.e.*, the value before the current `when` clause was triggered).

7.4.5.2 Simulation results

Simulation results for the `PeriodSensor` model are shown in Figure 7.11. The `divisions` parameter of the `PeriodSensor` model is used to indicate how many positions are reported over a single rotation of the sensor. Figure 7.11 compares the control system behavior using an ideal sensor as well as a `PeriodSensor` with 8 and 16 `divisions`.

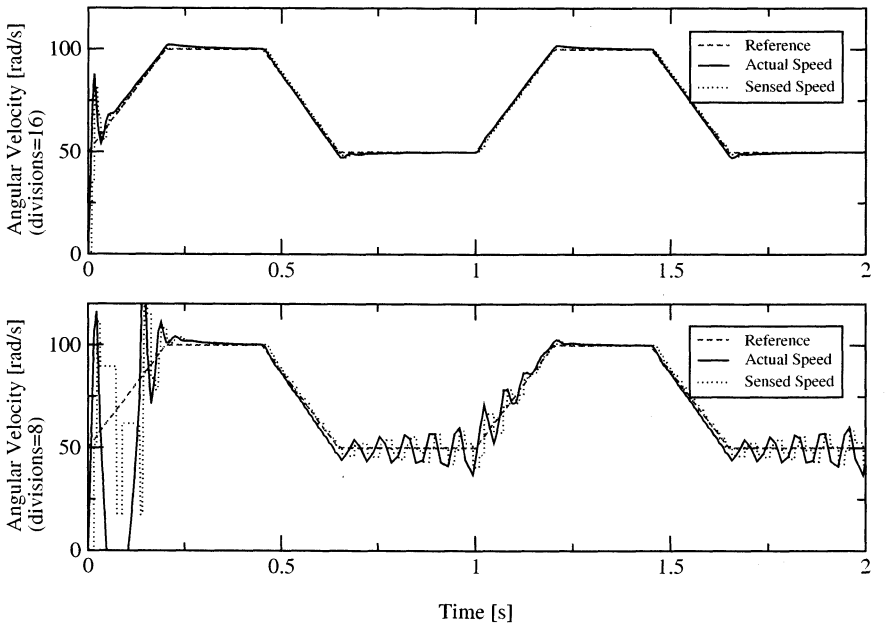

Figure 7.11. Comparison of `PeriodSensor` with ideal case.

7.4.5.3 Limitations

The `PeriodSensor` model has several limitations. First, it is incapable of distinguishing between forward and backward motion. For this reason, it would be completely inappropriate for measuring rotational speeds which oscillate around zero. It could also produce incorrect results when used to measure systems where the rotational speed came close to zero because of the possibility that the actual speed might cross zero.

Another limitation of the `PeriodSensor` is the resolution of the measurement. This is dictated largely by the number of locations (*i.e.*, `divisions` in the model) reported during a single revolution. The higher the speed, the fewer locations required. In other words, the `divisions` parameter must be chosen based on the range of speeds being measured. This effect is visible in Figure 7.11 which shows that the combination of low speeds and few divisions can lead to poor performance.

7.4.6 Counter sensor

The last model we will discuss, the `CountingSensor`, is similar to the `PeriodSensor`. It operates on the same principle by relying on position sensors to indicate when critical locations have been crossed. The difference between the `CountingSensor` and the `PeriodSensor` is that the `CountingSensor` counts the number of these intervals crossed in a given period of time to estimate speed. This can be important when large speeds are being measured because the measuring equipment may not have time to compute the speed at every report. Furthermore, the time intervals between reports become very small and this can lead to numerical scaling issues.

7.4.6.1 Behavioral description

Even though the physical principles are the same between the `PeriodSensor` and the `CountingSensor`, the approach taken in the `CountingSensor` is slightly different. As we can see in Example 7.14, the variables `upper` and `lower` used in `PeriodSensor` for reporting locations are not present. Instead, a continuous sinusoidal signal is generated by the motion of the system and when this sinusoidal signal crosses zero from below (*i.e.*, with a positive slope) a counter is incremented. The number of times such crossings will occur in a given rotation of the system is indicated by the `division` parameter.

At the sampling interval specified by the `sample_interval` parameter, the tally is cleared and the tally of reports from the previous cycle, obtained using the `pre` operator, is used to determine the speed of the sensor. Note that the `initial()` function is used in this case to initialize the `count` variable.

```
model CountingSensor
  import Modelica.Mechanics.Rotational;

  extends Rotational.Interfaces.AbsoluteSensor;
  parameter Integer divisions=4;
  parameter Modelica.SIunits.Time sample_interval=0.1;
protected
  constant Real pi=Modelica.Constants.pi;
  parameter Modelica.SIunits.Angle trigger_interval=
    2*pi/divisions;
  Integer count;
  Real s;
equation
  flange_a.tau = 0;
  s = Modelica.Math.sin(flange_a.phi*divisions);
algorithm
  when initial() then
    count := 0;
  end when;
  when s>=0 then
    count := pre(count)+1;
  end when;
  when sample(sample_interval,sample_interval) then
    count := 0;
    outPort.signal[1] :=
      (pre(count)+1)*trigger_interval/sample_interval;
  end when;
end CountingSensor;
```

Example 7.14. An interval counting approach.

Remember that the statements inside the `when` clauses are only evaluated for the instant the conditional expressions are true.

7.4.6.2 Simulation results

Figure 7.12 shows a comparison between the performance of our benchmark system using a `CountingSensor` and an ideal sensor. It should be noted that the gain on the PI controller had to be reduced in order for the control to be stable. As a result, even with an ideal sensor this system does not follow the reference signal as closely as in previous comparisons. Results are presented for different values of the `sample_interval` parameter.

7.4.6.3 Limitations

The `CountingSensor` has essentially the same limitations as the `PeriodSensor` (*i.e.*, good for one directional, high speed measurement). However, it

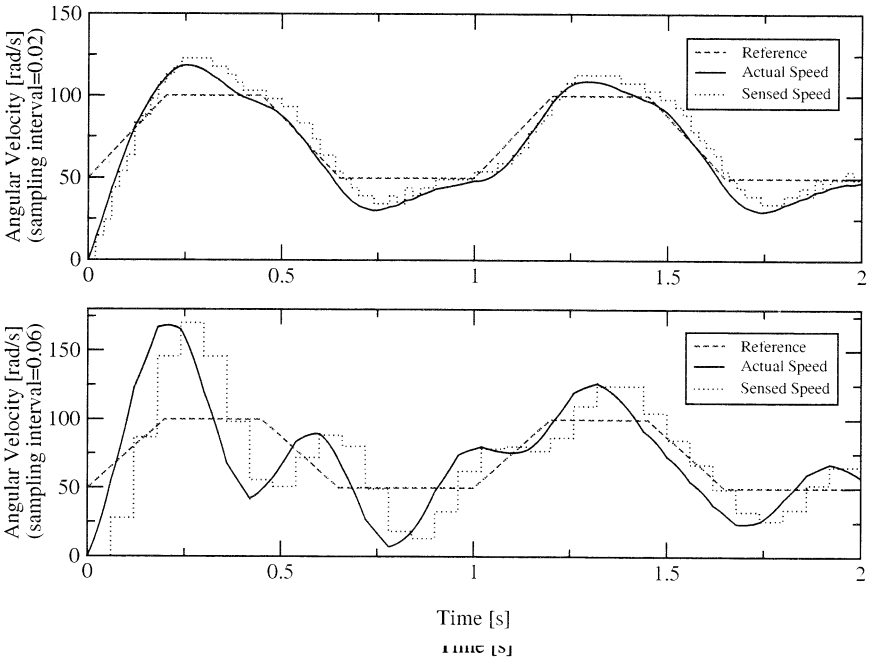

Figure 7.12. Comparison of `CountingSensor` with ideal case.

also suffers from some of the problems that the `QuantizedSensor` model has as well (*i.e.*, coarse signal output).

7.4.7 Summary of sensor modeling

This section has demonstrated several different features of the Modelica language related to hybrid modeling. In addition, we discussed sensor modeling which is an important aspect of physical system modeling. For example, such sensor modeling could be used to determine the effects of a sensor failure. Along the way, we presented several interesting examples which demonstrate how sensor characteristics must be carefully chosen to be compatible with the control system design.

7.5 LANGUAGE FUNDAMENTALS

7.5.1 Algorithms in models

In this chapter, we have seen several models that contain an `algorithm` section. It is important to understand that statements within an `algorithm` (or `equation`) section that do not appear within a `when` or `if` clause can be evaluated at any time by the simulator. On the other hand, statements inside

a **when** clause are evaluated only when the conditional expression in the **when** clause becomes true (this will be discussed in greater detail shortly).

7.5.2 Discrete variables

In Section 2.5.2.3 we mentioned that parameters, constants and variables all have a different *variability*. Recall that parameters and constants have a value which is held constant for the duration of a simulation while `Real` variables have the potential to change continuously. `Integer` and `Boolean` variables have piecewise constant solutions which means they have values that are constant most of the time but occasionally jump discontinuously to new values.

In some cases, it is useful to have `Real` variables that have piecewise constant solutions. Applying the **discrete** qualifier to the declaration of a `Real` variable ensures that the variable will be piecewise constant because the value of such a variable can only be modified by an assignment statement within a **when** clause. The **discrete** qualifier is provided to help model developers ensure that a variable is piecewise constant, but it is not required (*i.e.*, for all `Real` variables assigned within **when** clauses). The following code fragment demonstrates different uses of the **discrete** keyword:

```
model DifferentUses
  discrete Real x, y;
  Real z;
algorithm
  when initial() then
    x := 0; // ok, within when clause
  end when;
  when time>=1 then
    x := 1; // ok, within when clause
  end when;
  y := if time>=1 then 1 else 0; // error, outside of when
  z := if time>=1 then 1 else 0; // ok, not discrete
end DifferentUses;
```

Because `x` is assigned within a **when** clause it will always have a piecewise constant solution. Even though the assignment to `y` would appear to give the same solution as `x`, it will generate an error. This is because an **if** expression can contain time varying expressions (even though it does not in this case). The assignment to `z` is acceptable because it was not declared as **discrete**.

The **discrete** keyword can also be applied to a **connector** or **record**. In such a case, all `Real` variable declarations nested inside the **connector** or **record** declaration will be considered **discrete**.

7.5.3 Reacting to changing conditions

The `when` clause is generally used to either reinitialize a continuous variable using the `reinit` operator or to change the value of a discrete variable. A `when` clause may appear in either an `algorithm` section or in an `equation` section. If possible, it is preferable to place the `when` clause in an `equation` section for efficiency reasons. However, in some cases (*e.g.,* when two different `when` clauses within the same model modify the same discrete variable) it may be necessary to put the `when` clause in an `algorithm` section.

The statements inside a `when` clause are applied only at the instant in time at which the conditional expression (which follows the `when` keyword) becomes true. The `when` keyword differs from the `if` keyword because the statements inside an `if` clause continue to be applied as long as the conditional expression remains true. Furthermore, if the conditional expression in a `when` clause is a vector, then the `when` clause is activated if any of the conditions becomes true regardless of the value of the others.

When multiple `when` clauses within the same `algorithm` section of a model assign to the same discrete variables, the order of `when` clauses is important. For example, consider the case where we are developing a controller for a boat with two pumps. Assume we only have enough power to run one pump at a time. Imagine one pump is a bilge pump which keeps the boat from filling with water and the other pump is for a shower. Clearly, the bilge pump is more important. So, our controller may contain the following `when` statements:

```
model PumpController
  parameter Modelica.SIunits.Height h_crit;
  parameter Modelica.SIunits.Pressure p_crit;
  Modelica.SIunits.Height water_height;
  Modelica.SIunits.Pressure shower_pressure;
  Boolean shower_pump(start=false), bilge_pump(start=false);
algorithm
  when not bilge_pump and shower_pressure<=p_crit then
    shower_pump := true;
  end when;
  when water_height>=h_crit then
    shower_pump := false;
    bilge_pump := true;
  end when;
end PumpController;
```

The order of the `when` statements is important. Within an `algorithm` section, the last `when` clause is the last one evaluated. The ordering of the `when` clauses in this way was intentional. Consider the case where the bilge pump was not on and the shower pressure dropped below the critical level while at the same time the water height rose above the critical level. In such a case, it would be as if the statements in the `when` clause appeared in the following order:

```
shower_pump := true; // From the first when clause
shower_pump := false; // From the second when clause
bilge_pump := true;
```

In other words, the bilge pump would take precedence. Note that the shower pump came on and then was turned off immediately in this case. To prevent this, we might use an `elsewhen` clause to make sure that the actions taken were mutually exclusive, *e.g.*,

```
algorithm
  when water_height>=h_crit then
    shower_pump := false;
    bilge_pump := true;
  elsewhen not bilge_pump and shower_pressure<=p_crit then
    shower_pump := true;
  end when;
```

In this case, the shower pump can only be turned on when the water height is below the critical level. Note how the use of the `elsewhen` construct allows us to reorder the conditions so the most important condition comes first.

One final note about `when` clauses. It is important to understand that the following `when` clause:

```
when not bilge_pump and shower_pressure<=p_crit then
  shower_pump := true;
end when;
```

is not the same as:

```
when shower_pressure<=p_crit then
  if not bilge_pump then
    shower_pump := true;
  end if;
end when;
```

To understand the difference, consider the following scenario. Imagine the shower pressure drops below `p_crit` but the bilge pump is on. Then at some later time the bilge pump turns off while the shower pressure is still below `p_crit`. In the first case, the shower pump will come on as soon as the bilge pump turns off because the conditional expression will only become true at that instant. In the second case, the shower pump will not come on because the `when` clause was evaluated when the shower pressure became critical. Because the bilge pump was on, the opportunity to turn the shower pump on was lost because of the way the logic was written.

This `PumpController` example was written to be simple. However, writing such logic for controllers can be tricky. It is possible to use the basic hybrid language features in Modelica to write high level controller logic

representations (*e.g.*, petri nets). Oftentimes, such high-level representations may be a better choice than low level `when` statements.

7.5.4 Built-in functions and operators in hybrid systems

7.5.4.1 Reinitializing a variable

The `reinit` operator is used to make a discontinuous change in the value of a continuous variable. The `reinit` operator can only be applied to variables that have had the `der` operator applied to them.

The effect of the `reinit` operator is to stop simulation time, make a change to the value of one or more continuous variables and then resume simulation. It is effectively like starting a new *initial value problem*. It is important to recognize the implication a `reinit` might have on the algebraic equations as well as the differential equations in a system. A more detailed discussion of these implications can be found in Chapter 13.

7.5.4.2 Values prior to events

For the `PeriodSensor` and `CountingSensor` models (shown in Examples 7.13 and 7.14), we used the `pre` operator to access the previous value for a discrete variable at the instant it changed. Whenever a variable changes value discontinuously, the `pre` operator can be used to find the previous value at the instant of the discontinuity. The `pre` operator can be used with continuous variables, but only within a `when` clause. Mathematically, the previous value is defined as the *left limit* of the variable at the time the discontinuity occurs (*e.g.*, the time the `when` clause is activated). At the start of the simulation, the `pre` operator returns the value of its argument (*i.e.*, `pre(x)=x`).

7.5.4.3 Masking events

Conditional expressions in Modelica have an interesting property. If a conditional expression changes value during a simulation, it forces the underlying solver to stop at the point the transition occurs. This is done because discontinuities in the continuous system of equations generally occur as a result of such changes in conditional expressions. For numerical reasons, it is best to stop the integration at that point and restart so that the discontinuity does not occur in the middle of an integration step.

Most of the time, this rule makes sense. However, in some cases it is undesirable for such interruptions to occur. In general, such interruptions are not necessary when the model developer knows for certain that no discontinuity actually occurs. Such interruptions can be avoided by using the `noEvent` operator to indicate that no discontinuity occurs as a result of the conditional expression. By avoiding the interruption, some computation effort is saved both in determining exactly when the conditional expression changes value and

in computing the extra time step. Another reason to use the `noEvent` operator is to enforce interpretation of the conditional expression. To understand why this might be necessary, consider the case of a tank of liquid which empties according to the following equation:

$$\frac{dx}{dt} = \begin{cases} -\sqrt{x} & : \quad x > 0 \\ 0 & : \quad x \leq 0 \end{cases} \tag{7.6}$$

where x is the height of liquid in the tank. One way to write a model for such a tank would be:

```
model EmptyingTank1
  Real x;
equation
  der(x) = if x>0 then -(x^.5) else 0.0;
end EmptyingTank1;
```

It appears, at first glance, that there is no danger that an attempt will be made to take the square root of `x` when `x` is negative. The conditional expression $x > 0$ would seem to protect against this. However, this is not the case. To understand why, consider the following equivalent model:

```
model EmptyingTank2
  Real x;
  Boolean cond;
equation
  cond = x>0;
  der(x) = if cond then -(x^.5) else 0.0;
end EmptyingTank2;
```

If a simulation starts with a positive value for `x`, then `cond` will be `true`. As long as `cond` is true, $-\sqrt{x}$ will be evaluated. Remember that conditional expressions cause the simulator to try and identify the time at which the conditional expression changes value. As a result, the value of `cond` is not evaluated constantly. Instead, the simulator looks for the point at which `x` dips below zero. It is only once that point has been identified that the value of `cond` is changed to `false`. In the meantime, while searching for that point, negative values of `x` will be considered and used in evaluating any expressions. In summary, the simulator cannot know for sure that `x` is less than zero until it sees a negative value for `x` and then it is too late because it has used that value as the argument to a square root operation.

The remedy for this situation is to place the `noEvent` operator around the conditional expression. However, some continuous expressions can also trigger events (*e.g.*, the built-in `abs()` function), so it is best to place the entire expression within the `noEvent` as follows:

```
model EmptyingTank3
  Real x;
equation
  der(x) = noEvent(if x>0 then -(x^.5) else 0.0);
end EmptyingTank3;
```

The `noEvent` operator suppresses events from being generated by the expression it is applied to. This avoids the problem of taking the square root of a negative number because the simulator tests the condition $x > 0$ during every evaluation of the right hand side of the equation rather than waiting to identify when the condition changes. The drawback is that if a discontinuity did occur, some numerical error would be introduced.

It should be noted that there are a few restrictions on the use of the `noEvent` operator. First, the conditional expression of a **`when`** clause cannot be qualified by the `noEvent` operator. In addition, the `noEvent` operator cannot be used in `Boolean`, `Integer` or `String` equations.

As a final note, there have been proposals within the Modelica Association to revise the `noEvent` operator. As of the writing of this book, none of these proposals have been accepted. For the near term, it should continue to function as described here. However, if you run into difficulties you might want to make sure the semantics have not been revised.

7.5.4.4 Detecting changes

Two more useful functions are the `edge()` and `change()` functions. These functions are designed to indicate when variables change their values. The `edge()` function can only be used with a `Boolean` variable and the `change()` function can only be used on `Boolean`, `Integer` and `String` variables. Each function has an equivalent conditional expression. The `edge()` function is defined as:

```
edge(x) = x and not pre(x);
```

In other words, `edge()` is **`true`** at the instant that the argument, `x`, has just become **`true`** (*i.e.*, `x` is **`true`** and was not previously **`true`**). The `change()` function is defined as:

```
change(x) = x <> pre(x);
```

In other words, `change()` is **`true`** at the instant that `x` has a different value than it previously had.

7.5.4.5 Generating events at regular intervals

The `sample()` function is a built-in function that takes two arguments. The first argument is the time at which sampling begins. The second argument

is how frequently sampling occurs once it begins. Both arguments are of type `Modelica.SIunits.Time` (*i.e.*, seconds).

The `sample()` function returns the value `false` except at the instant when sampling begins (*i.e.*, the first argument) and after every sampling interval (*i.e.*, the second argument). The `true` value only occurs for an instant.

7.5.4.6 Identifying the start and end of analysis

Two more built-in functions which can be used for hybrid models are the `initial()` and `terminal()` functions. The `initial()` function becomes `true` only for an instant just as a simulation starts. It can be useful for setting initial conditions for both continuous and discrete variables (see Chapter 13 for more details). Likewise, the `terminal()` function becomes `true` for an instant at the end of a successful simulation. One example of how the `terminal()` might be used would be to call an external function which writes out final simulation results to a file.

7.5.4.7 Terminating a simulation

Finally, we come to the `terminate()` function. This function is used to indicate that it is no longer useful to continue a simulation. The `terminate()` function takes a single argument, s, of type `String`. This argument represents the message that will be displayed to explain the termination of the simulation.

The main reason to terminate a simulation is because the simulation has already evaluated what is of interest and so therefore further simulation would not yield anything useful. For example, if we wish to simulate a thermal system response up until the point where the temperature of the system stops changing, we could use terminate as follows:

```
...
  Modelica.SIunits.Temperature T;
...
algorithm
  when abs(der(T))<1e-3 then
    terminate("Temperature at steady-state");
  end when;
```

As a result, when the rate of temperature change drops below $10^{-3}\frac{K}{s}$, the simulation will be terminated.

The `terminate()` function is usually used to indicate the end of a successful simulation. If you wish to terminate a simulation because something has gone wrong with the simulation (*e.g.*, the value of a variable is outside the valid range) then the `assert()` function should be used.

7.5.4.8 Special considerations for other functions

Special consideration must be given for some of the built-in functions in Modelica. For example, the following code fragment:

```
x := if y<0 then -y else y;
```

could also be written, using the built-in `abs()`, as follows:

```
x := abs(y);
```

Since there is an implicit conditional expression within the definition of `abs()`, it has the same effect (*i.e.*, stopping the integration when y crosses zero) as the conditional expression it replaces. In order to avoid such interruptions, the `noEvent` operator can be used just as it is with other expressions, *e.g.*,

```
x := noEvent(abs(y));
```

The functions `abs()`, `ceil()`, `div()`, `floor()`, `integer()`, `mod()`, `rem()` and `sign()` may cause an interruption in the integration process due to discontinuities. These discontinuities occur because the return value of the function (or one of its derivatives) is not continuous with respect to its arguments. For example, the use of the `integer()` function in the `QuantizedSensor` model from Example 7.12 will trigger an interruption every time the `level` variable changes.

7.5.5 Well posed problems

In algebra class, we first learn that in order to solve a system of equations we must have exactly as many variables as equations. The same holds true for Modelica models. However, for Modelica we need the number of variables to be equal to the number of equations **plus** the number of assignments. There are some important caveats to this rule. First, all equations and assignments within `when` clauses are counted. Furthermore, all assignments to the same variable within a single `algorithm` section count as a single assignment.

7.6 PROBLEMS

PROBLEM 7.1 *In practice, simulations are primarily used to perform "What If?" analyses. Re-run the* `LogicCircuitWithLag` *model for different values of the lag parameter* `c`*. Look at the effect it has not only on the time delay in the circuit, but also on the accuracy of the output signals. These models can be found on the companion CD-ROM.*

PROBLEM 7.2 *The lag discussed in Section 7.2.4 is caused by capacitance in the wires, delaying the rise of the voltage. Assume that a* `true` *value corresponds to 5 Volts and a* `false` *value corresponds to 0 Volts. Create a lag*

model using two resistors and a capacitor as shown in Figure 7.13. Compare the results with those shown in Figures 7.4 and 7.5.

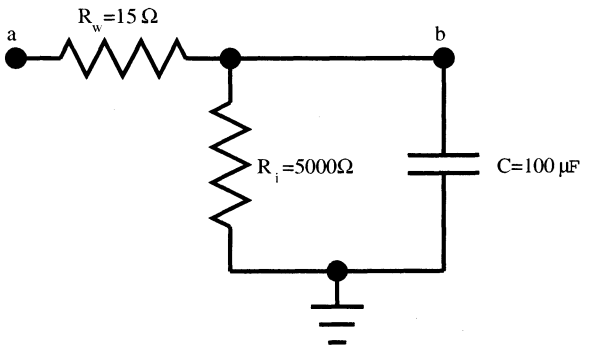

Figure 7.13. Circuit to model inertial delay.

PROBLEM 7.3 *Run the benchmarking cases from Section 7.4, but this time lower the reference speed amplitude and/or offset values. What does this do to the performance of the various controllers?*

PROBLEM 7.4 *The different kinds of sensor models developed in this section were always connected directly to the mechanical system. Create blocks for the four different types of sensor models that take the actual velocity as an input and output the velocity indicated by the sensor. Using these blocks, you can "daisy-chain" effects. For example, you could read the actual velocity using an ideal sensor and then feed that signal into a* `PeriodSensor` *block. Then, you could feed the output from the* `PeriodSensor` *block into a* `QuantizedSensor` *and look at what the overall effect is on the velocity signal.*

PROBLEM 7.5 *Create a sensor model that introduces a delay in the feedback loop. Put that in the benchmark case and study the effects for different delay values.*

PROBLEM 7.6 *As we discussed in Section 7.4, in addition to modeling the plant, modeling of sensors and actuators is important. Many of the sensor models presented in that section can cause the system to become unstable. To further demonstrate real-world situations, develop an* **actuator** *model which saturates at some predetermined level (i.e., a torque source that can only produce specified minimum and maximum values regardless of the commanded value). In which cases does this mitigate the instabilities due to non-ideal sensors and in which cases does it make things worse?*

PROBLEM 7.7 *Create a model of a block that numerically computes the derivative of its input. To do this, you must sample the input signal at regular intervals and compute a finite difference approximation for the signal, e.g., :*

$$\frac{du}{dt} = \frac{u(t) - u(t - \Delta t)}{\Delta t} \tag{7.7}$$

Chapter 8

EXPLORING NONLINEAR BEHAVIOR

8.1 CONCEPTS

The point at which modeling gets particularly interesting is when model behavior becomes increasingly nonlinear. It is no coincidence that this is the point where simulation tools start having trouble. Nonlinear behavior is hard to avoid in real world models. The examples in this chapter will introduce some of the approaches used to describe nonlinear behavior.

8.2 AN IDEAL DIODE

We begin our examples with an ideal electrical diode. This example introduces a useful parameterization technique in modeling non-linear systems.

8.2.1 Mathematical background

In order to understand the parameterization technique, we must first examine the problem that makes this technique necessary. Consider the following ideal diode equations:

$$\begin{aligned} i &= 0 \quad \text{when} \quad v \leq 0 \\ v &= 0 \quad \text{when} \quad i \geq 0 \end{aligned} \tag{8.1}$$

In order to get a better understanding of how an ideal diode behaves, consider the graphical representation of Equation (8.1) shown in Figure 8.1. The thick line represents possible states of the diode. What makes such behavior difficult to model is that the current cannot be written in terms of the voltage and the voltage cannot be written in terms of the current. In mathematical terms, the current is not a proper function of the voltage and the voltage is not a proper function of the current.

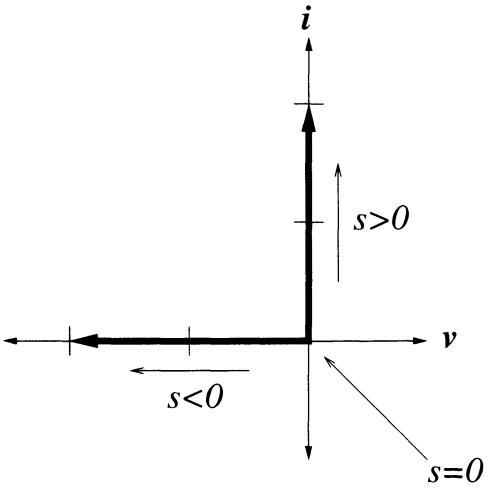

Figure 8.1. Current-voltage characteristics of an ideal diode.

This leads us to the technique which allows us to work around the issue (see Otter et al., 1999). Specifically, we must express the behavior parametrically in terms of another variable. Let us call this parametric variable s. The variable s must be chosen such that the current and the voltage can both be written explicitly in terms of s. In this way, the state of the diode becomes a continuous function of s. One way to perform this mapping is to consider s to be the distance along the curve shown in Figure 8.1, starting at the origin. Mathematically, we can then write voltage and current in terms of s as follows:

$$v = \begin{cases} s & : \quad s < 0 \\ 0 & : \quad s \geq 0 \end{cases} \tag{8.2}$$

$$i = \begin{cases} 0 & : \quad s < 0 \\ s & : \quad s \geq 0 \end{cases} \tag{8.3}$$

Once again, a graphical representation is sometimes easier to understand. Figure 8.2 shows the voltage and current plotted with respect to s.

8.2.2 Model description

Now we must translate Equations (8.2) and (8.3) into Modelica code. It turns out that by reusing the `OnePort` partial model from Example 4.1, we can write this model in a few lines as shown in Example 8.1.

Note that our `IdealDiode` model contains two internal variables, `s` and `open`. The `s` variable is our parametric variable used as the independent variable in Figure 8.2. The variable `open` represents which part of the curve

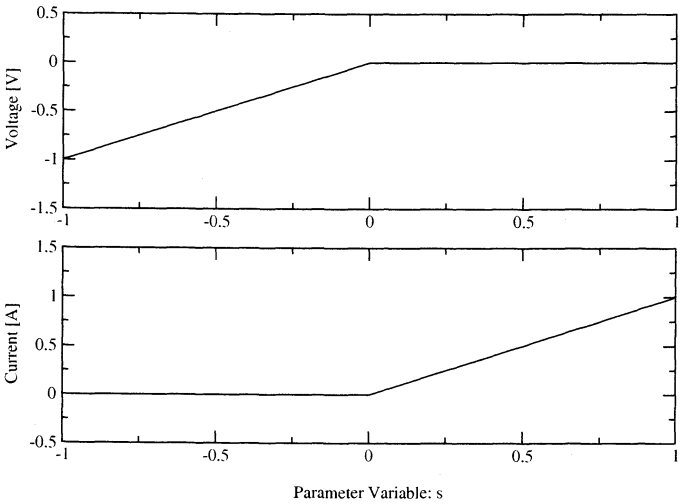

Figure 8.2. Current-voltage characteristics of an ideal diode plotted parametrically.

we are on in Figure 8.1. If `open` is `true`, then `s` represents the voltage drop across the diode (the horizontal line in Figure 8.1). On the other hand if `open` is `false`, then `s` represents the current flow through the diode (the vertical line in Figure 8.1). As a result, a change in `open` represents a fundamental behavioral change in the model (additional details about these kinds of behavioral changes can be found later in Section 8.6.2).

```
model IdealDiode "An Ideal Diode"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
protected
  Real s "Parametric independent variable";
  Boolean open;
equation
  open = s<=0;
  v = if open then s else 0;
  i = if open then 0 else s;
end IdealDiode;
```

Example 8.1. An ideal diode model.

8.2.3 Sample circuit

Figure 8.3 shows the schematic of an alternating current (AC) to direct current (DC) power supply. After the AC voltage has been stepped down using a transformer, the diode is used to rectify the resulting AC signal and the ripples

that result from this rectification are damped out by a resistor and capacitor. A switch is used to connect the load to the power supply. Figure 8.4 shows the results of simulating this system for one second. The top plot shows the supplied voltage, V_{supply}, the middle plot is the voltage across the load, V_{load}, and the bottom plot shows the state of the switch.

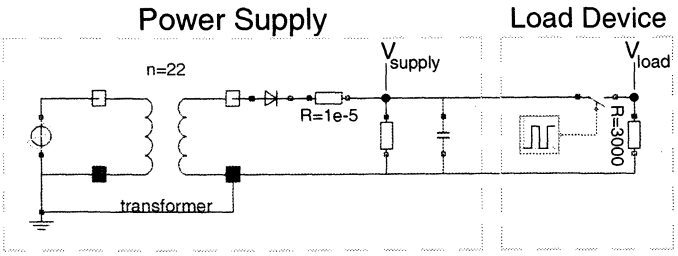

Figure 8.3. Schematic of an AC/DC power supply.

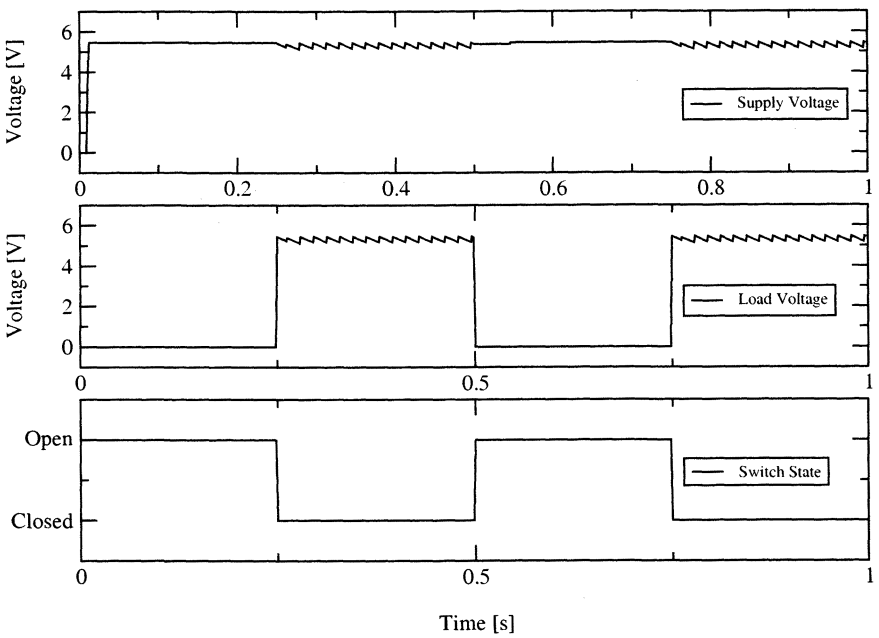

Figure 8.4. Voltage response of an AC/DC power supply.

One thing to note about the circuit in Figure 8.3 is the fact that a small resistance, R_{short} was placed in series with the diode. This was done because simulation tools often have difficulty modeling ideal components. For example,

if the resistor in Figure 8.3 is left out, the resulting system of DAEs will have a variable index (for details see Mattsson and Söderlind, 1993). For this reason, the ideal diode model in the MSL has a resistance built into the model.

8.3 BACKLASH

A common nonlinearity introduced when building mechanical models is the representation of backlash. Backlash can occur in both rotational and translational systems and can be an important effect for many types of mechanical models. We will describe two approaches to handling the backlash problem and discuss the advantages and disadvantages of both approaches.

8.3.1 Non-linear spring approach

The first approach to consider is called the "non-linear spring approach". This approach involves implementing a spring with a force-displacement curve like the one shown in Figure 8.5.

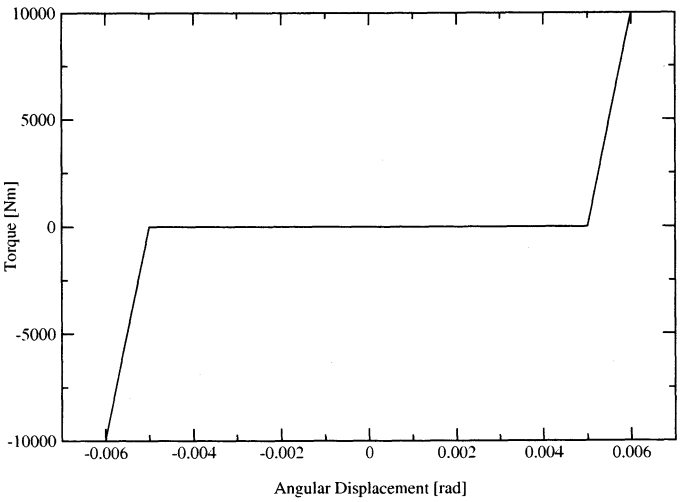

Figure 8.5. Force-displacement characteristics for a backlash.

The basic idea behind this approach is to implement a "spring" that exerts no force until sufficient angular displacement occurs such that there is no more backlash in the system. At that point, the spring becomes stiff (corresponding to the collision of two rigid bodies) and the two rigid bodies "bounce" away from each other. These collisions will rarely be modeled as perfectly elastic and therefore a damping term is usually added as well. The equation for the

force between the two colliding bodies is:

$$\tau = \begin{cases} c\left(\phi + \frac{b}{2}\right) + d\dot{\phi} & : & \phi & < -\frac{b}{2} \\ 0 & : & -\frac{b}{2} \leq & \phi & \leq \frac{b}{2} \\ c\left(\phi - \frac{b}{2}\right) + d\dot{\phi} & : & \frac{b}{2} < & \phi & \end{cases} \tag{8.4}$$

where ϕ is the angular displacement across the backlash component, b is the amount of backlash, c is the spring constant and d is the damping coefficient. Example 8.2 shows a model for such behavior.

```
model SpringBacklash
  import Modelica.Mechanics.Rotational;
  extends Rotational.Interfaces.Compliant;
  parameter Modelica.SIunits.Angle b=0.05 "Backlash amount";
  parameter Real c=1e+5 "Spring stiffness";
  parameter Real d=0 "Damping coefficient";
protected
  Modelica.SIunits.AngularVelocity w_rel;
equation
  w_rel = der(phi_rel);
  if phi_rel<=-b/2 then
       tau = c*(phi_rel+b/2)+d*w_rel;
  elseif phi_rel>=b/2 then
    tau = c*(phi_rel-b/2)+d*w_rel;
  else
    tau = 0;
  end if;
end SpringBacklash;
```

Example 8.2. Non-linear spring backlash model.

The drawback of this approach is that the system of equations becomes very "stiff" when contact is made. This is not desirable for several reasons. First, the system of equations can become difficult and time consuming to solve. Second, prolonged contact will result in a persistent, high-frequency vibrational response. Ultimately, this mode may be damped out but it could cause robustness problems with the model. Finally, it is not necessarily easy to know exactly what the compliance and damping properties of the colliding materials are. In those cases, it becomes difficult to correctly capture the dynamics.

8.3.2 Coefficient of restitution approach

The coefficient of restitution approach, modeled in Example 8.3, avoids the problems with the stiff spring. The idea behind the `BacklashCOR` model is to

```
model BacklashCOR
  import Modelica.Mechanics.Rotational;
  extends Rotational.Interfaces.Compliant;
  parameter Modelica.SIunits.Angle b=0.05;
  parameter Modelica.SIunits.Inertia I1=1, I2=1;
  parameter Real K=1 "Coefficient of Restitution";
protected
  Modelica.SIunits.AngularVelocity w1, w2;
  Modelica.SIunits.AngularAcceleration a1, a2;
equation
  w1 = der(flange_a.phi);
  w2 = der(flange_b.phi);
  a1 = der(w1);
  a2 = der(w2);
  tau = 0;
algorithm
  when phi_rel>=b/2 or phi_rel<=-b/2 then
    reinit(w1, ((I1-K*I2)*pre(w1)+I2*(1+K)*pre(w2))/(I1+I2));
    reinit(w2, ((I2-K*I1)*pre(w2)+I1*(1+K)*pre(w1))/(I1+I2));
  end when;
end BacklashCOR;
```

Example 8.3. Coefficient of restitution backlash model.

recognize the point at which the collision occurs, compute how much momentum will be lost (determined by the coefficient of restitution) and recompute the velocity of the two colliding bodies as follows:

$$\omega_1(t+\epsilon) = \frac{(I_1 - KI_2)\omega_1 + I_2(1+K)\omega_2}{I_1 + I_2} \tag{8.5}$$

$$\omega_2(t+\epsilon) = \frac{(I_2 - KI_1)\omega_2 + I_1(1+K)\omega_1}{I_1 + I_2} \tag{8.6}$$

where ω represents rotational velocity, I represents rotational inertia and K is the coefficient of restitution. This adjustment to the angular velocity is done in a single step by using the `reinit` operator (*i.e.*, the simulator does not attempt to resolve the dynamics of the collision) so the numerical issues are avoided. Prolonged contact of the two bodies can still be a problem but this can be overcome (as we previously showed with the bouncing ball model in Example 7.9).

The problem with the coefficient of restitution model is that it requires knowledge of the effective inertia of each body. This is not something a single model can know because it is possible that additional inertias are rigidly connected to the backlash model. These rigid connections change the effective inertia of the bodies and make the internal momentum calculation incorrect.

Even worse is the case where multiple inertias are connected by multiple backlashes because the effective inertia of the assembly changes depending on the current state of each backlash. This places a burden on the user of the model to make sure the correct values for the effective inertia are somehow provided to the model.

Another interesting thing to note about the `BacklashCOR` model is the fact that the `reinit` operator can only be applied to a variable which has had the `der` operator applied to it. The variables `a1` and `a2` are dummy variables introduced so that `reinit` could be applied to `w1` and `w2`.[1] Finally, the `pre` operator was used (see Section 7.5.4.2 for further details) to reference the angular velocities prior to the collision.

8.3.3 Comparison

Let us do a quick comparison of both of these approaches. Figure 8.6 shows the schematic of one test case where there are two inertias and both are directly connected to the backlash model. We will refer to this as the "two inertia" case. For the next case, an additional inertia was added. This new inertia is rigidly connected (through an ideal gear with a gear ratio of 1) to one of the previous inertias. We will call this the "three inertia" case. The values of the inertias were adjusted so that these two cases (*i.e.*, the two and three inertia cases) are physically identical. In other words, the effective inertia on both sides of the backlash is the same in both cases.

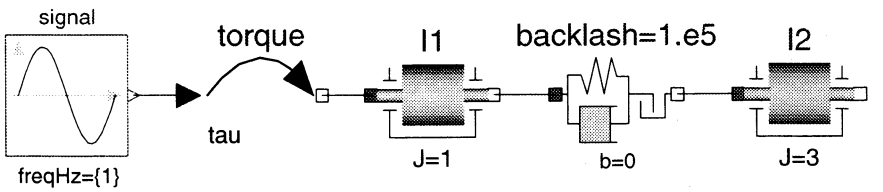

Figure 8.6. Backlash schematic with two inertias.

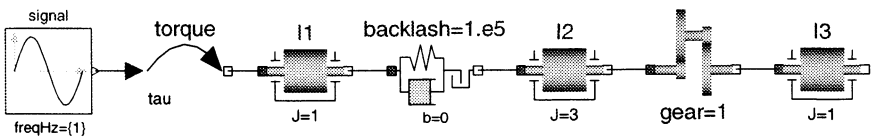

Figure 8.7. Backlash schematic with three inertias.

[1] In fact, the current semantics of re-initialization are being examined to see if there is a way to avoid the necessity of adding such dummy variables.

The problem for our `BacklashCOR` model is that it needs to know the **effective** inertia at both of its connection points, not just the value of the inertia directly connected to it. For the "3 inertia" case, we have deliberately chosen to include only the value of the inertia directly connected to the backlash and not the inertia contribution across the gear. This is an easy mistake to make because it is not necessarily obvious to somebody building such a schematic that the effective inertia is required or how to calculate it. This is because that calculation depends on the equations contained within the various components. In summary, even though we have made this error intentionally, it is the kind of error that is made easily by accident.

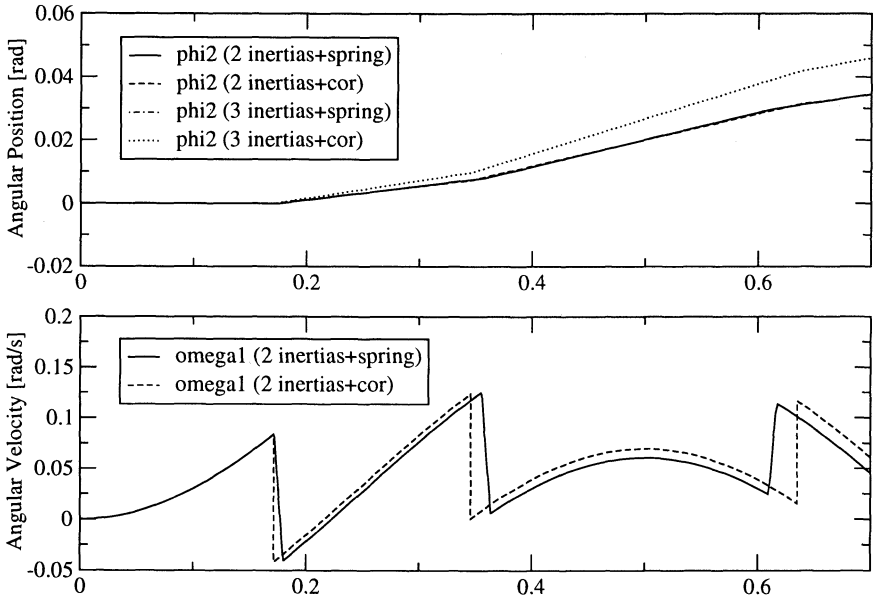

Figure 8.8. Comparison of the two backlash models for the cases shown in Figures 8.6 and 8.7.

The top plot in Figure 8.8 shows the angular position of the second inertia, `I2`, for both the "two inertia" and "three inertia" cases using both backlash models. As we can see from this plot, all the models give the correct answer except for the coefficient of restitution model using three inertias. This demonstrates that the coefficient of restitution model is less robust than the non-linear spring approach because it requires the user to carefully make sure that the data given to the backlash model (*i.e.*, the effective inertia on both sides of the backlash) is consistent with the physical characteristics of the schematic. Such situation are always troublesome and should be avoided.

The bottom plot in Figure 8.8 shows the angular velocity of the first inertia, `I1`, using both backlash models in the "two inertia" case. This plot demonstrates an interesting property of the different solution methods. The dashed line indicates the coefficient of restitution model for the "two inertia" case. Note the abrupt changes in the velocity. These jumps are the result of using the `reinit` operator.

Let us focus on the first collision. Note how both solutions at the bottom of Figure 8.8 are identical up until the first collision occurs. The coefficient of restitution model jumps immediately to a new value when the collision occurs. The non-linear spring model takes a finite amount of time to resolve the collision. The difference in the models leads to a delay for the non-linear spring model. This delay accumulates at each collision.

The amount of the delay is related to the stiffness of the spring. This means that if you choose the stiffness value arbitrarily, you will get an arbitrary delay. This highlights one of the drawbacks of the non-linear spring approach. Specifically, the method requires you to treat the colliding bodies as "very stiff". The problem is "how stiff is very stiff?". As we have seen, the value chosen for the stiffness will make some difference in the solution so it should be chosen carefully. The other drawback of the approach is that, in general, the stiffer you make the spring, the longer it will take the simulation tool to simulate the collision.

8.3.4 Summary and future directions

As pointed out in our discussion of backlash models, the non-linear spring approach has the disadvantage that it can lead to inefficient numerical simulations. In addition, it may not be easy to measure the compliance and damping factors for the materials involved. On the other hand, the `reinit` approach avoids those problems but will not work correctly if one of the inertias connected to the backlash model is rigidly connected to another inertia. If stiff connections are substituted for rigid connections, then undesirable high frequency modes will appear.

As a result, the best approach currently available for modeling backlash is to use the non-linear spring approach because it is general enough to handle all cases and it is robust. However, there is a proposal currently being formulated for an extension to the Modelica modeling language which would introduce the ability to model collisions using impulses. This would provide a general, robust and computationally efficient approach to modeling backlashes and several other phenomena. If you are interested in effects like backlash, check with the Modelica Association to find out when such impulse handling is likely to be available and whether it will be suitable for you.

8.4 THERMAL PROPERTIES

8.4.1 Background

In Section 6.3, we described how a simple heat transfer system can be modeled in Modelica. In this section, we will show how to introduce non-linear thermal properties into a heat transfer model.

By applying conservation of energy to a control volume we arrive at the following equation:

$$\frac{d}{dt}\int_V \rho u(T)\, dV = -\int_S \vec{J}\cdot\hat{n}\, dS \tag{8.7}$$

where $u(T)$ represents the specific internal energy of the material within the volume V. If we assume that the density and temperature do not vary over the volume then we can simplify Equation (8.7) to:

$$V\rho\frac{du(T)}{dt} = -\int_S \vec{J}\cdot\hat{n}\, dS \tag{8.8}$$

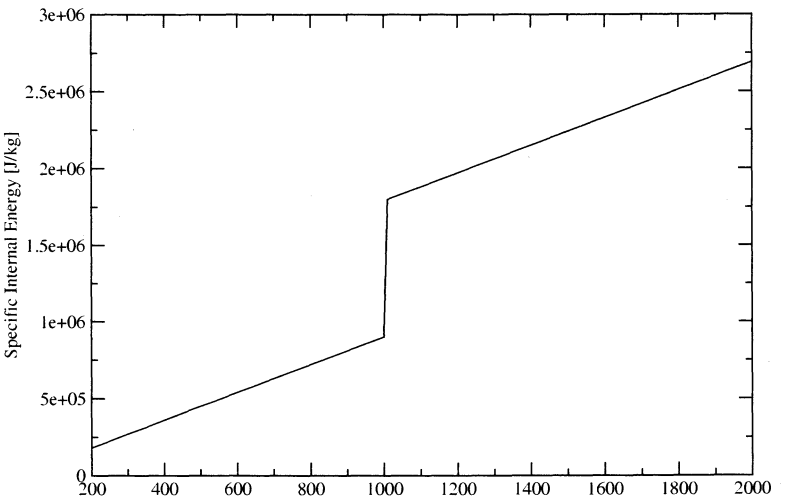

Figure 8.9. Plot of $u(T)$ from Equation (8.9).

Note that Equation (8.8) does not specify relationship between u and T. Such relationships are often non-linear and present some interesting modeling challenges. In order to explore this possibility, let us assume the following relationship between specific internal energy and temperature:

$$u(T) = \begin{cases} 900T & : \quad 200 \leq T < 1000 \\ 90000T - 89100000 & : \quad 1000 \leq T \leq 1010 \\ 900T - 891000 & : \quad 1010 < T \leq 2000 \end{cases} \tag{8.9}$$

Figure 8.9 shows a plot of u as a function of T. Since $c_p(T)$ is defined as the derivative of specific internal energy, u, with respect to temperature, we find that c_p for our non-linear material, based on Equation (8.9), would be:

$$c_p(T) = \begin{cases} 900 & : & 200 \leq & T & < 1000 \\ 9000 & : & 1000 \leq & T & \leq 1100 \\ 900 & : & 1100 < & T & \leq 2000 \end{cases} \tag{8.10}$$

One important thing to note about the material properties shown in Equations (8.9) and (8.10) is that they are limited to the temperature range $200K$ to $2000K$.

8.4.2 Creating a thermal property model

Creating a property model involves encapsulating Equation (8.9) into a Modelica model. Since different materials have different specific internal energy relationships, Example 8.4 shows how we can define a `partial` model to capture the essence of what a property model consists of.

```
partial model ThermalPropertyModel
  Modelica.SIunits.Temperature T;
  Modelica.SIunits.SpecificInternalEnergy u;
end ThermalPropertyModel;
```

Example 8.4. A general thermal property model interface.

Example 8.5 shows how we can capture the specific internal energy relationship in Equation (8.9). Note that Example 8.5 uses the `assert()` function to verify that the property model is used in a valid region. If the simulated solution ever fails to satisfy the conditional expression inside the `assert()` function invocation, an error occurs and the message provided in the `assert()` function invocation will be given as an explanation. The ability to impose such restrictions and present a meaningful message when they are violated is important in creating robust simulations.

In addition, notice that `start` values have been provided for both `u` and `T` in Example 8.5. This is always a good idea when building non-linear models. Remember, the `start` attribute is just a guess about what a reasonable value might be (*i.e.*, it provides the simulator a good initial guess).

A subtle property of Example 8.4 is that it is not represented as a `block`. In other words, we have not identified the temperature and specific internal energy as either inputs or outputs of the model. This allows the model to be used either as an explicit equation for u as a function of T *or* as an implicit equation for T as a function of u.

```
model SimplePropertyModel
  parameter Modelica.SIunits.Temperature Tl=1000;
  parameter Modelica.SIunits.Temperature Tu=1100;
  parameter Modelica.SIunits.SpecificHeatCapacity cp_s=900;
  parameter Modelica.SIunits.SpecificHeatCapacity cp_m=9000;
  parameter Modelica.SIunits.SpecificHeatCapacity cp_l=900;
  extends ThermalPropertyModel(u(start=cp_s*300));
equation
  assert(T>=200 and T<=2000, "T out of range");
  if T<=Tl then
    u = cp_s*T;
  elseif T<=Tu then
    u = cp_m*(T-Tl)+Tl*cp_s;
  else
    u = cp_l*(T-Tu)+cp_m*(Tu-Tl)+Tl*cp_s;
  end if;
end SimplePropertyModel;
```

Example 8.5. A specific thermal property model.

8.4.3 Modeling non-linear thermal capacitance

Example 8.5 defines our material property behavior but now we must use it within the context of a non-linear capacitance model. Example 8.6 shows a non-linear version of the thermal capacitance model found in the `Thermal` package.

```
model ThermalCapacitanceNL "Non-linear rod section"
  Thermal.Interfaces.Node n;
  parameter Modelica.SIunits.Density rho;
  parameter Modelica.SIunits.Volume V;
  replaceable ThermalPropertyModel props(T=n.T);
equation
  // Conservation of energy
  V*rho*der(props.u) = n.q;
end ThermalCapacitanceNL;
```

Example 8.6. A non-linear thermal capacitance model.

There are several interesting things to point out about the declaration of the property model. First, the declaration is `replaceable` which allows us to use a wide variety of property models with this capacitance model.

Secondly, note that the modification `T=n.T` has been applied to the `props` component. This modification represents an equation just as if we had added the line

```
props.T = n.T;
```

to the `equation` section of the model. It is often useful to specify equations in this way because the implications of the equation appear close to the component they modify. The disadvantage of including an equation in a component declaration is that it is easy to forget about such equations because they do not appear explicitly in an `equation` section.

8.4.4 Simulating solidification

Now we wish to combine the specific material property relationship defined in Example 8.5 with the general non-linear thermal capacitance model shown in Example 8.6. The material property relationship defined in Example 8.5 is interesting because the steep region (*i.e.*, between $1000K$ and $1100K$) represents the transition from a solid state to a liquid state.

```
model SolidifyingRod
  import Thermal.Basic1D;
  import BCs=Thermal.BoundaryConditions;
  import Modelica.SIunits;
  parameter SIunits.Length L=0.3 "Total length";
  parameter SIunits.Area A=4.0 "Cross-sectional area";
  parameter SIunits.Density rho=5.0;
  parameter SIunits.ThermalConductivity k=0.5;
  parameter SIunits.CoefficientOfHeatTransfer h=10;
  parameter Integer nsections=30 "# of sections";
  parameter SIunits.Length sec_L=L/nsections;
  // Components
  ThermalCapacitanceNL cap[nsections](V=sec_L*A,rho=rho,
     redeclare SimplePropertyModel props(Tu=1010,cp_m=90000));
  BCs.FixedTemperature Tr(T=1800);
  Basic1D.Conduction c_cond[nsections-1](L=sec_L,A=A,k=k);
  Basic1D.Convection r_conv(A=A,h=h);
equation
  for i in 1:nsections-1 loop
    connect(c_cond[i].a,cap[i].n);
    connect(c_cond[i].b,cap[i+1].n);
  end for;
  connect(Tr.n,r_conv.b);
  connect(r_conv.a,cap[nsections].n);
end SolidifyingRod;
```

Example 8.7. A rod changing from solid to liquid.

Example 8.7 is similar to the `HTProblem1` model from Example 6.15 with a few important differences. First, the thermal capacitance model shown in Example 8.6 is used for the segments of the rod. In addition, the model uses the

models contained in the `Thermal package`. Finally, the boundary conditions for this problem have changed. The left end is now adiabatic (*i.e.,* no heat transfer) and the right end has a convective boundary condition attached to it.

The results of running the simulation are shown in Figure 8.10. The plot shows a series of temperature curves as a function of longitudinal distance along the rod for various times during the solidification. The discontinuity in the slope of the temperature curve between $1000K$ and $1100K$ is an artifact of the non-linear property relationship. The classic reference for this type of problem is (Stefan, 1891).

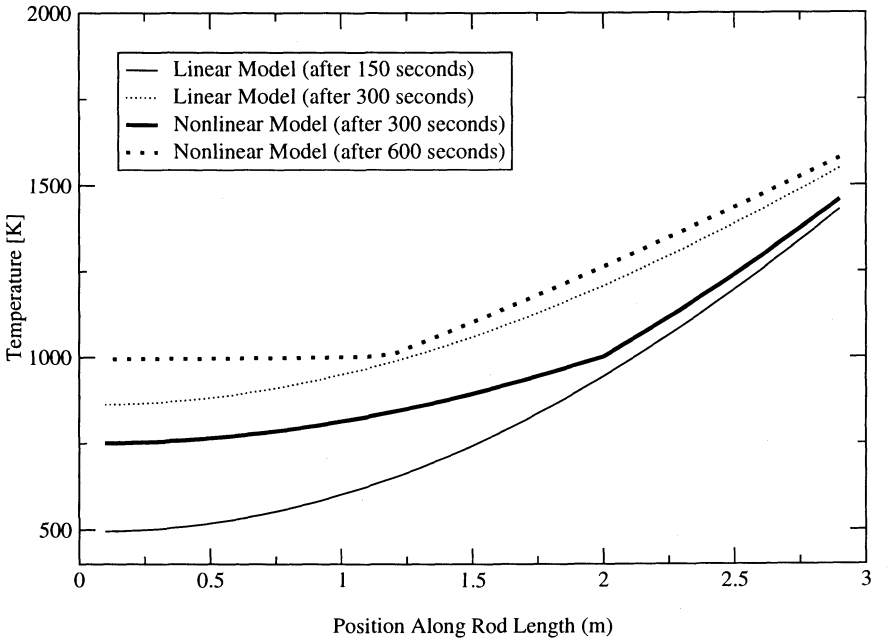

Figure 8.10. Temperature distributions in `SolidifyingRod` for linear and nonlinear property models.

8.5 HODGKIN-HUXLEY NERVE CELL MODELS

One of the most interesting non-linear models I have seen is the Hodgkin-Huxley model (see Hodgkin and Huxley, 1952). This model, used for simulating the electrical activity of nerve cells, was the basis for awarding Hodgkin and Huxley the Nobel Prize in 1963. We will only present a cursory explanation of the behavior of this model. A more detailed explanation can be found in (Bower and Beeman, 1994).

8.5.1 Background

The basic idea in this model is that molecules of sodium and potassium move across the membrane of a nerve cell. Because these molecules are ionized, they carry with them an electric charge. As a result, the motion of these molecules results in an electrical current through the membrane of the cell which then causes a change in the voltage difference across the membrane. The motion of the sodium and potassium ions is governed by ion channels in the membrane wall. The activation level of the ion channels is represented by a real number between 0 (completely closed) and 1 (completely open). Rather than having a fixed conductance, the conductance of the ion channels is determined by a differential equation which depends on the voltage drop across the membrane and the activation level of the ion channel.

8.5.2 Circuit model

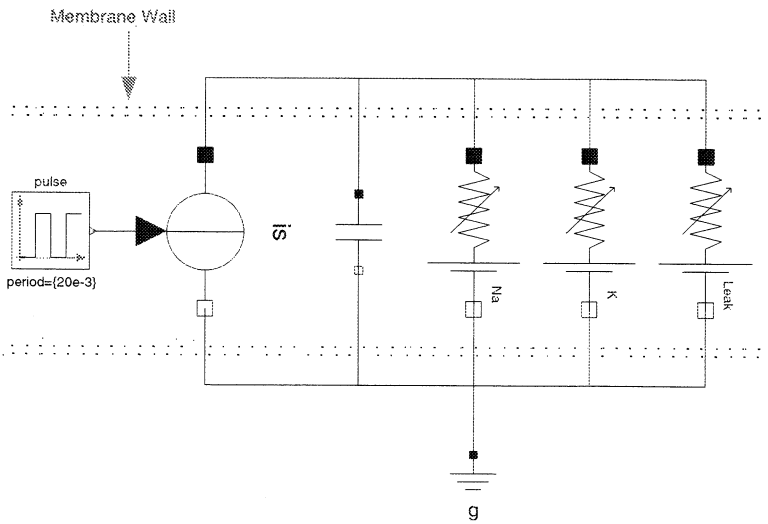

Figure 8.11. Nerve cell segment schematic.

Figure 8.11 shows an electrical schematic of a nerve cell. The purpose of the model is to predict the changes in voltage across the membrane of the nerve cell in response to the injection of a pulse of current. The non-linear behavior of the ion channels in the nerve cell along with the time scale variation in the response of each type of ion channel causes fluctuations in the voltage across the cell membrane.

Figure 8.12 shows the voltage, current and channel activation levels as a function of time for the schematic shown in Figure 8.11. Note how the voltage

spikes in response to the injected current and then recovers after the current excitation has stopped.

A complete model for the circuit shown in Figure 8.11 is included on the companion CD-ROM. However, to give a sample of the types of equations used, we will include some discussion of the sodium channel model. The conductance, G, of the sodium channel is determined by the following equations:

$$a_h = 0.07e^{-\frac{E+V_{ah}}{20}} \tag{8.11}$$

$$b_h = (1+e^{\frac{-E+V_{bh}}{10}})^{-1} \tag{8.12}$$

$$a_m = \frac{E+V_{am}}{10(1-e^{-\frac{E+V_{am}}{10}})} \tag{8.13}$$

$$b_m = 4e^{-\frac{E+V_{bm}}{18}} \tag{8.14}$$

$$\dot{h} = 1000\,(a_h(1-h) - hb_h) \tag{8.15}$$

$$\dot{m} = 1000\,(a_m(1-m) - mb_m) \tag{8.16}$$

$$G = m^3hAG_{Na} \tag{8.17}$$

where m and h are internal activation levels, A is the membrane area and G_{Na} is the maximum conductance of the sodium channel.

Using these equations, we can implement the sodium channel model in Modelica as follows:

```
model SodiumChannel "Hodgkin-Huxley Sodium Channel"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  parameter SIunits.Area membrane_area;
protected
  constant HodgkinHuxley.MilliVoltage V_am=40, V_bm=65;
  constant HodgkinHuxley.MilliVoltage V_ah=65, V_bh=35;
  Real m_prob "Probability of activation of channel";
  Real h_prob "Probability of inactivation of channel";
  SIunits.DecayConstant a_m, b_m;
  SIunits.DecayConstant a_h, b_h;
  SIunits.Conductance G;
  HodgkinHuxley.MilliVoltage E=1000*v;
  parameter SIunits.Conductance g_max=membrane_area*G_Na;
equation
  G = m_prob^3*h_prob*g_max;
  i = G*(v - E_Na);
  der(m_prob) = 1000*(a_m*(1 - m_prob) - b_m*m_prob);
  a_m = (.1*(E + V_am))/
       (1 - Modelica.Math.exp(-(E + V_am)/10));
  b_m = 4*Modelica.Math.exp(-(E + V_bm)/18);
  der(h_prob) = 1000*(a_h*(1 - h_prob) - b_h*h_prob);
  a_h = .07*Modelica.Math.exp(-(E + V_ah)/20);
  b_h = 1/(1 + Modelica.Math.exp(-(E + V_bh)/10));
end SodiumChannel;
```

To put the complexity of this model in perspective, compare it to the resistor model shown Example 3.2.

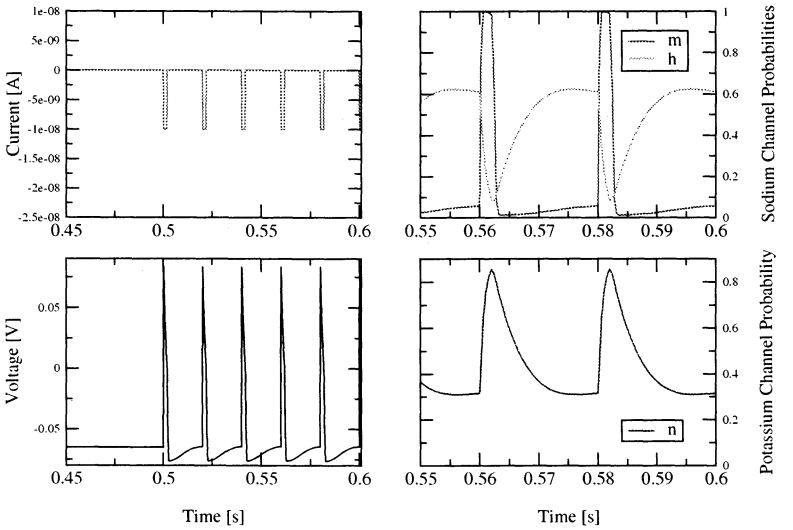

Figure 8.12. Dynamic response of the nerve cell.

8.6 LANGUAGE FUNDAMENTALS

Nonlinear models are often required for practical engineering analysis. Therefore, it is important to be able to represent such nonlinearities and have tools available which are capable of simulating highly nonlinear systems. Let us summarize some of the important aspects of the Modelica language that support the creation of nonlinear models.

8.6.1 Parametric formulation

Example 8.1 used a technique that parameterizes non-linearities in terms of an intermediate variable. This technique is quite useful when you have relationships between through and across variables that cannot be expressed using a simple functional relationship.

We saw (*e.g.*, in the `IdealDiode` model) how an `if` statement or `if` expression can be used to construct functions which compute the through and across variables for different regimes of behavior. It is important when using parametric formulations to make sure that the through and across variables can be expressed as continuous functions of the intermediate variable.

8.6.2 Behavioral changes

As we have seen (*e.g.,* in the `IdealDiode` model from Example 8.1 and the `SimplePropertyModel` model from Example 8.5), the presence of either an `if` expression or an `if` statement can introduce behavioral changes in a model.

The most obvious effect is that the model containing an `if` statement or `if` expression has two different modes of behavior. It is important that these two modes provide a continuous description of behavior. For example, the specific internal energy, `u`, shown in Example 8.5 is a continuous function of `T` even though it contains an `if` statement. Without this continuity, there would likely be numerical problems.

The more subtle effect, introduced by the presence of conditional expressions, is the generation of events as described previously in Section 7.5.4.3. Let us consider the `IdealDiode` model in Example 8.1. Note that a `Boolean` variable was introduced to represent the state of the diode. This allowed the model to be expressed with only a single conditional expression. If, instead, we had written the model as:

```
model IdealDiode "An Ideal Diode"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
protected
  Real s "Parametric independent variable";
equation
  v = if s<0 then s else 0;
  i = if s<0 then 0 else s;
end IdealDiode;
```

we might have run into some difficulty. The reason is that then we would have two conditional expressions. We can tell by looking at these expressions that they are identical. However for numerical reasons, we cannot be sure that a given tool will recognize this.

If these two conditional expressions are treated independently, we have no way to make sure that the behavioral changes triggered by the conditional expressions will happen at the same time. In other words, the simulator may determine that the conditional expression used to compute `v` becomes `true` slightly before (or after) the conditional expression used to compute `i`. This could lead to difficulties when solving a problem. The `open` variable in the `IdealDiode` model was introduced to synchronize the behavior change of both `v` and `i`.

It seems obvious that two identical conditional expressions should be treated as a single conditional expression. However, it is not always so easy to recognize equivalent conditional expressions because they may be mathematically equivalent but they are not necessarily numerically equivalent. For example,

the following conditional expressions are all mathematically equivalent but it is easy to see why a tool may not recognize that they are:

```
s < 0;
5*s < 0;
-s > 0;
not s >= 0;
4000+s < 4000;
s^3 < 0;
```

Another way to make sure that behavior changes are synchronized is to use an `if` statement. For example, the `IdealDiode` model in Example 8.1 could have been written as:

```
model IdealDiode "An Ideal Diode"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
protected
  Real s "Parametric independent variable";
equation
  if (s<0) then
    v = s;
    i = 0;
  else
    v = 0;
    i = s;
  end if;
end IdealDiode;
```

This approach also results in a single conditional expression which triggers only a single event.

It should be pointed out that both `if` statements and `if` expressions in an `equation` section **must** include an `else` clause. In addition, all possible branches of an `if` statement in an equation section must contain the same number of equations. As mentioned previously in Chapter 5, an `if` statement can also include several `elseif` clauses.

8.6.3 Discontinuities

The `reinit` operator is used to make a discontinuous change in the value of a variable providing that variable has had the `der` operator applied to it. This operator allows us to model discontinuities as we did in the `BacklashCOR` model shown in Example 8.3. The first argument to the `reinit` operator is the variable whose value should change and the second argument is the value it should change to. The `reinit` operator can only be used within a `when` clause because it is meant to represent an abrupt change. In other words, `reinit` is fundamentally different from an assignment or an equation because it represents an abrupt change in value rather than a continuous change. When

the new value is a function of the previous value, the `pre` operator can be used to refer to the previous value.

In the future, it is expected that additional features will be added to the Modelica language that will allow the description of physical collisions more directly. These will make the behavioral descriptions in such models easier to express and will address some of the robustness issues with the `BacklashCOR` model described in Section 8.3.2.

8.6.4 Implicit equations

Nonlinear models often contain implicit equations. An implicit equation is an equation where the unknowns are not solved for directly. For example, the equation

$$aT^6 + bT^5 + cT^4 + dT^3 + eT^2 + fT + g = 0 \tag{8.18}$$

cannot be solved directly for T because there is no closed form solution to such a general polynomial equation. Such an equation must be solved implicitly. As we saw in Section 8.4, we had a system with the following equations:

$$V\rho\frac{du}{dt} = q \tag{8.19}$$

$$u(T) = \begin{cases} 900T & : \quad 200 \leq T < 1000 \\ 90000T - 89100000 & : \quad 1000 \leq T \leq 1010 \\ 900T - 891000 & : \quad 1010 < T \leq 2000 \end{cases} \tag{8.20}$$

Clearly, it is not easy to rearrange this system of equations to yield an equation for T directly. Instead, Equation (8.19) would typically be used to solve for u and then T would be solved implicitly using Equation (8.20).

As we have seen in this chapter, such systems of equations are easy to pose in Modelica. While such systems are slightly more complicated to solve, the burden of solving these equations falls on the tool and not on the model developer. While other approaches (*e.g.,* block diagrams) may discourage the use of such expressions, they should not be a problem for any tool which uses Modelica because the ability of tools to solve such systems is a necessity.

8.6.5 Idealizations

One thing to keep in mind when trying to model nonlinear systems like the ones presented in this chapter, is that idealizations can sometimes make the task easier and sometimes make it harder depending on the situation. For example, we saw in Figure 8.4 that the resistance of the ideal diode had to be included to simulate that problem. In the case of the backlash models presented in Section 8.3, the idealized model (*i.e.,* the coefficient of restitution model) also had difficulties with multiple rigidly connected inertias.

Often, some understanding of the implications of such idealizations is required when building models (in Modelica or any other modeling language). With perfectly ideal models, it is often necessary to introduce some additional effect which is more physically reasonable (*e.g.*, the resistance in the diode model). This can often help mitigate numerical difficulties.

8.7 PROBLEMS

PROBLEM 8.1 *A Zener diode behaves like a regular diode (i.e., it does not allow current flow for negative voltage drops across it) except when the voltage drop supported by the diode goes below a critical negative voltage, called the breakdown voltage. When this happens, the diode again allows current flow. Figure 8.13 shows the behavior of a Zener diode in the iv-plane. Develop a parametric formulation of this behavior and create a* Modelica *model. Then, test the model by placing a sinusoidal voltage across the diode whose amplitude exceeds the breakdown voltage. When building such a model, keep in mind the issues mentioned in Section 8.6.5.*

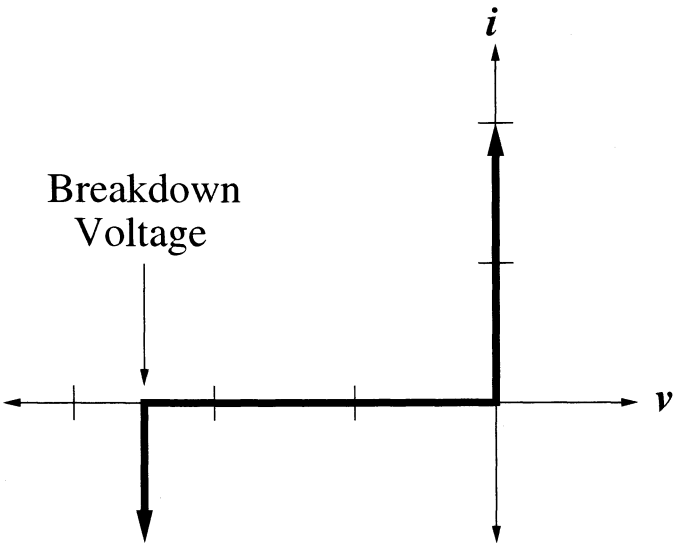

Figure 8.13. Current-voltage characteristics of an ideal Zener diode.

PROBLEM 8.2 *Create some additional property models which extend the* `ThermalPropertyModel` *shown in Example 8.4 and use them in conjunction with the* `SolidifyingRod` *model.*

PROBLEM 8.3 *Create a* Modelica *model to solve the following equations:*

$$x = t + 1 \tag{8.21}$$
$$x = y^2 \tag{8.22}$$

where t is time. Note that because the first equation already provides an equation for x in terms of quantities which are already known, the second equation becomes an implicit equation for y in terms of x.

When you simulated this problem, what solution do you get for y? How many different analytical solutions are there and which one did you find when ran a simulation? How could you control the choice of which solution is used?

PROBLEM 8.4 *Write a model to simulate the Lorentz-Lorenz equation:*[2]

$$\dot{x} = \sigma(y - x) \tag{8.23}$$
$$\dot{y} = \rho x - y - xz \tag{8.24}$$
$$\dot{z} = xy - \beta z \tag{8.25}$$

Parameters values to try are $\sigma = 10$, $\beta = 2.6667$ and $\rho = 28$. Try several initial values for x, y and z. Visualize the result by plotting any variable as a function of the other two.

If you are using Dymola, include the following declaration:

```
Sphere p(x=x, y=y, z=z, R=0.01);
```

and you can visualize the dynamics in three dimensions using the animation feature in Dymola.

[2] As an interesting aside, this same equation was derived independently by Hendrik Lorentz and Ludwig Valentin Lorenz in 1880.

Chapter 9

MISCELLANEOUS

The previous chapters focused on demonstrating language features which allow robust and reusable models to be written. In this chapter, we will cover a few "loose ends" which should be discussed for completeness but are not required in order to begin using Modelica.

9.1 LOOKUP RULES

Writing and maintaining large collections of models requires the use of packages and other organizational features of Modelica. In this section, we will explain the lookup rules used in conjunction with those features. Before we start, it is important to understand the difference between a package hierarchy and an instance hierarchy.

Package hierarchies, like the `Chemistry` package developed in Section 6.4.4, are collections of Modelica definitions. In other words, they are the definitions of records, connectors, *etc.* not actual instances. For example, the following is a package hierarchy:

```
package A
  model X
    ...
  end X;
  model Y
    record R // <- Definition of R
      Real c;
    end R;
    R r;    // <- Instantiation of R
  end Y;
end A;
```

In this case, the name `A.X` refers to the definition of the `model` named `X` nested inside the `package` named `A`. Even though this is called a package hierarchy, not everything in it has to be a `package`. For example, the name `A.Y.R` refers to the `record` definition nested inside the `model` named `Y` which is nested inside the `package` named `A`.

In addition to package hierarchies, we also have instance hierarchies which appear once a definition has been declared. For example, using the previous package hierarchy example, consider the following model definitions:

```
model M
  A.Y y;
equation
  der(y.r.c) = ...;
end M;
```

In this case, the only definition is the definition of `M`. However, within the definition, we reference a variable called `y.r.c`. This is the variable `c` contained within the record instance `r` inside the component `y`. In this case, we are using the "`.`" operator to traverse the instance hierarchy (*i.e.,* the hierarchy of instantiated components).

One important thing to note is that `constant` declarations are also considered definitions. In other words, a `constant` declaration exists in both the package hierarchy and the instance hierarchy.

When you see a collection of names separated by the "`.`" operator in Modelica (*e.g.,* `a.b.c`), it is impossible to tell whether the names represent a definition (*i.e.,* `A.Y.R` in our package hierarchy example) or an instance (*i.e.,* `y.r.c` in our instance hierarchy example). Simply being aware of the source of the confusion will help somewhat. In addition, some people choose to adopt a policy of starting definition names with a capital letter and instance names with a lowercase letter. This allows definitions and instances to be more easily distinguished.

9.1.1 Static scoping

Given an existing package hierarchy, it is important to understand how to access Modelica definitions that might be contained within the hierarchy. This problem can be broken into two distinct pieces. In this section we will discuss how to access definitions from within a given `package` hierarchy. The other issue, discussed later in Chapter 12, is how that `package` (possibly spanning multiple files) can be stored on the computer.

Every Modelica definition exists somewhere in the package hierarchy. All the different packages, taken together, are like the branches of a tree. For example, all the definitions in the MSL exist in the `Modelica` hierarchy. Any

definition that is not explicitly placed in a `package` hierarchy must be contained at the root of all the different hierarchies.

In order to understand how the lookup rules work, consider the following fragment of Modelica code:

```
package Pneumatic
  package Interfaces
    connector Port
      ...
    end Port;
  end Interfaces;
  model Valve;
    ...
  end Valve;
  model Pump
    ...
  end Pump;
end Pneumatic;

model PneumaticModel1
  ...
end PneumaticModel1;
```

This code fragment would result in the `package` hierarchy shown in Figure 9.1.

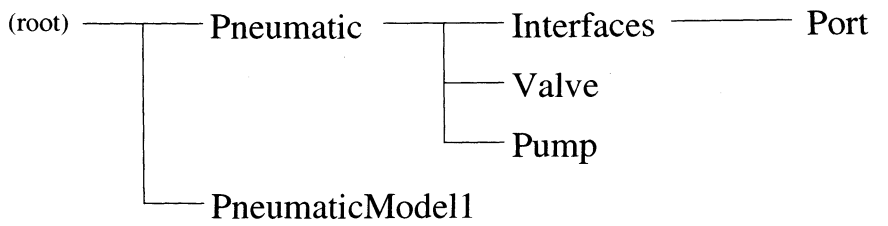

Figure 9.1. Sample package hierarchy.

9.1.1.1 Normal search behavior

Suppose, within the `Valve` and `Pump` models, we wish to use the definition for the `Port` connector found in the `Interfaces` package. To do this, we must have a way of referring to the `Port` definition. The lookup rule used in this case could be paraphrased as:

> To find a definition, first the site of the declaration is searched. If this fails, we move up the package hierarchy toward the root of the hierarchy, searching at each level as we go until the root is reached. If the definition cannot be found from any of these levels, the search fails.

Using this rule, let us examine several ways we might go about referencing the `Port` connector definition. First, we could use a name that includes each package that the `Port` definition is nested inside of, *i.e.*,

```
package Pneumatic
  ...
  model Valve
    Pneumatic.Interfaces.Port p1, p2;
    ...
  end Valve;
end Pneumatic;
```

For this case, the search consists of looking for the `Pneumatic` package and then, once that is found, looking inside it for the definition of the `Interfaces` package, and if that succeeds, finally inside that for the `Port` definition. In other words, the search is always for the outermost definition. So, we start at our current location in the hierarchy (*i.e.*, within the `Valve` model) and search for a definition for `Pneumatic`. Since it cannot be found at the current position, we must then look in the next higher level of the hierarchy (*i.e.*, the `Pneumatic` package itself). There is no definition for `Pneumatic` within the `Pneumatic` package so finally we arrive at the root. At the root, we find the definition for `Pneumatic` and it in turn contains the definitions for `Interfaces` and `Port` that we are seeking.[1] In this case, the search succeeds.

Now, let us consider another case. We could also write our model in this way:

```
package Pneumatic
  ...
  model Valve
    Interfaces.Port p1, p2;
    ...
  end Valve;
end Pneumatic;
```

Now, we are looking for the definition of something called `Interfaces` and we expect to find the definition of something called `Port` inside that. Starting at the `Valve` model, we find no definition for `Interfaces`. So, we move to the `Pneumatic` package which does contain a definition for `Interfaces` that in turn contains a definition for `Port`. So again, the search succeeds.

Once the definition of the first component in a composite name is found, if the subsequent components in the name cannot be located inside the definition of the first component, the search fails. In other words, the search will not

[1] A name that includes every package name from the root down to the definition itself is called a fully qualified name.

continue up the hierarchy to find yet another definition for the first component in the name that might contain the necessary subcomponent definitions. For example, imagine we are searching for the definition of `Interfaces.Port`. During our search we find a package called `Interfaces`. So, we check to see if a definition for `Port` can be found inside. If not, the search fails. The search does not continue looking for an alternate definition of `Interfaces`, farther along the hierarchy, that does contain a definition for `Port`.

9.1.1.2 Searching other locations

The normal search pattern is to search up the package hierarchy until the root of the package hierarchy is reached and if the definition is not found, then the search fails. There is a way to cause the search to look in other packages as well. For these cases, we use the `import` keyword. Recall from previous sections how we used `import` to shorten the names of physical types, *e.g.*,

```
model Resistor
  import Modelica.SIunits;
  parameter SIunits.Resistance R;
  ...
end Resistor;
```

An important restriction on `import` statements is that the name of the definition being imported must be a fully qualified name (*i.e.*, the name of the definition relative to the top of the package hierarchy). Furthermore, the definition being imported must be defined within a `package`.

If the search comes across an `import` statement in any of the packages it is searching, the imported package is searched as if its definition appeared at the location of the `import` statement. Sometimes, the name of the imported definition might be the same as a another definition higher up in the package hierarchy. In order to avoid confusion about which one should be searched, the `import` command allows a different name to be used for the imported package, *e.g.*,

```
model Resistor
  import SI=Modelica.SIunits;
  parameter SI.Resistance R;
  ...
end Resistor;
```

The use of `import` is not limited to packages either. An `import` statement can be used for other types of definitions. For example, this is also possible:

```
model Resistor
  import Modelica.SIunits.Resistance;
  parameter Resistance R;
  ...
end Resistor;
```

Finally, `import` is only relevant for searching, which means it does not actually change the package hierarchy in which it appears. To illustrate this, consider the following example:

```
package MyElectrical
  model Resistor=Modelica.Electrical.Analog.Basic.Resistor;
  import Modelica.Electrical.Analog.Basic.Capacitor;
  ...
end MyElectrical;

model MyCircuit
  MyElectrical.Resistor R; // Legal
  MyElectrical.Capacitor C; // Illegal, no such definition
end MyCircuit;
```

In this example, the use of the `import` statement is only useful to definitions contained within the `MyElectrical` package and does not result in new definitions being added to the `MyElectrical` package hierarchy.

9.1.1.3 Limiting searches

So far, we have described how the search proceeds up the package hierarchy. Normally, the search continues up to the root of the package hierarchy. However, in some cases we may not wish to allow definitions to be used beyond a certain point in the hierarchy. For example, we may wish to define a `package` or `model` that can be easily relocated in the package hierarchy. To understand why, consider the following example:

```
package TestProblems "A collection of tests"
  constant Real g=Modelica.Constants.g_n;
  model Pendulum
    parameter Real L=2;
    Real theta, omega;
  equation
    der(theta) = omega;
    der(omega) = -(g/L)*theta;
  end Pendulum;
  ...
end TestProblems;
```

The difficulty here is that the `Pendulum` model requires the gravitational constant, `g`, defined in its package hierarchy (remember, constants can be accessed through both a package hierarchy or an instance hierarchy). In order to be able to move this model outside of the `TestProblems` package, we would need to define `g` within the `Pendulum` model.

In this case, it is easy to see that we will have problems if we try to move the `Pendulum` model. In more complex cases, these kinds of issues are difficult to identify. For this reason, the Modelica language includes the `encapsulated`

qualifier. The effect of the `encapsulated` qualifier is to stop the search from going beyond the boundary of the `encapsulated` definition. If we rewrite our `Pendulum` example as follows:

```
package TestProblems "A collection of tests"
  constant Real g=Modelica.Constants.g_n;
  encapsulated model Pendulum
    parameter Real L=2;
    Real theta, omega;
  equation
    der(theta) = omega;
    der(omega) = -(g/L)*theta; // Error, no definition for g
  end Pendulum;
  ...
end TestProblems;
```

The search for `g` will fail because the search cannot go beyond the `encapsulated` model. By using the `encapsulated` qualifier, we can identify any current dangling references and prevent introducing any in the future.

The `encapsulated` keyword only prevents the search from proceeding up the package hierarchy. It does not limit the use of `import` statements in any way. So, to make our `Pendulum` model independent of the `TestProblems` package, we should write it as follows:

```
package TestProblems "A collection of tests"
  constant Real g=Modelica.Constants.g_n;
  encapsulated model Pendulum
    import Modelica;
    constant Real g=Modelica.Constants.g_n;
    parameter Real L=2;
    Real theta, omega;
  equation
    der(theta) = omega;
    der(omega) = -(g/L)*theta;
  end Pendulum;
  ...
end TestProblems;
```

9.1.1.4 Making a definition local

Rather than searching up through the hierarchy as we have been doing, there are several ways to make a definition local so we do not need to search through the hierarchy for it. A simple example of this would be:

```
package Pneumatic
  ...
  model Valve
    connector Port=Interfaces.Port;
    Port p1, p2;
```

```
    ...
  end Valve;
end Pneumatic;
```

What we have done is create a new definition locally (*i.e.*, within the `Valve` model itself) and used that definition.

As another example, let us consider the `Pump` model. Because the pump involves both pneumatic and rotational connections, it might be written as follows:

```
package Pneumatic
  ...
  model Pump
    Pneumatic.Interfaces.Port p1, p2;
    Modelica.Mechanics.Rotational.Interfaces.Flange_a fa;
    ...
  end Pump;
end Pneumatic;
```

Note that in this case the fully qualified names have been used. Let us try to shorten some of the names. One way would be to create local definitions as in:

```
package Pneumatic
  ...
  model Pump
    connector Port=Interfaces.Port;
    package Rotational=Modelica.Mechanics.Rotational;
    Port p1, p2;
    Rotational.Interfaces.Flange_a fa;
    ...
  end Pump;
end Pneumatic;
```

Note that in one case, we made a local definition of a `connector` and in the other case we made a local definition of a `package`. The drawback to this approach is that it creates new definitions that must be parsed and interpreted. For example, the local `package` definition recreates the entire `Modelica.Mechanics.Rotational` hierarchy inside this single model. If this is done in several models, the time required to parse and interpret such a large and complex package hierarchy becomes a factor. If these local definitions are not likely to be used by other models, modified or redeclared in the future, the definitions should be imported as follows:

```
package Pneumatic
  ...
  model Pump
    import Pneumatic.Interfaces.Port;
    import Modelica.Mechanics.Rotational;
    Port p1, p2;
```

```
    Rotational.Interfaces.Flange_a fa;
    ...
  end Pump;
end Pneumatic;
```

which avoids cluttering the package hierarchy.

9.1.1.5 Conclusion

These are just several examples of how to access definitions and components that exist outside the current model. A detailed discussion of the lookup semantics can be found in the language specification on the companion CD-ROM. The examples in this section demonstrate most of the common methods of accessing definitions.

9.1.2 Dynamic scoping

As we saw in the previous section, static scoping is used to access definitions from within the package hierarchy. In contrast, dynamic scoping involves searching the instance hierarchy rather than the package hierarchy. Dynamic scoping can be used to refer to declarations and definitions since both are contained within the instance hierarchy.

In Section 4.3.2, we described how to propagate information through a hierarchy of components. Such propagation is used to promote reuse of components. In some cases, the methods described in Section 4.3.2 are not sufficient. In this section, we will describe how to use dynamic scoping to automatically establish connections between declared components and their surroundings.

9.1.2.1 Particles and fields

To understand what dynamic scoping is and why it is useful, let us consider the case of simulating small particles in a gravitational field. In this case, each particle needs to know the gravitational acceleration at the particle's current location. The most convenient way to express such information is by using a `function`. For simulations in three dimensional space, the `function` should be a subtype of the `GravityField` `function` shown in Example 9.1.

```
partial function GravityField
  input Modelica.SIunits.Position x[3];
  output Modelica.SIunits.Acceleration g[3];
end GravityField;
```

Example 9.1. Using a `function` to describe a gravity field.

If each particle is assumed to be so small that it has no influence on other particles in the system, then the motion of the particle can be expressed as:

$$a = g(x)$$

where a is the acceleration of the particle, x is the current location of the particle and g is the gravitational acceleration.

The issue in this example is that the particle model needs information about the gravitational field it is moving in (*i.e.*, any relationship between position, x, and gravitational acceleration, g). In other words, the particle requires information about its *environment* in order to describe its behavior. This situation is not unique to gravitational examples. Environmental information can also be important in other situations such as thermal or electrical systems.

The difficulty in creating Modelica models that require such environmental information is that it is not possible to `connect` a particle declaration to a `function`. In such cases, we can use the `inner` and `outer` keywords to indicate that dynamic scoping should be used. With dynamic scoping, the particle can lookup information about its environment (*i.e.*, the component hierarchy in which the particle has been declared).

```
model Particle
  parameter Modelica.SIunits.Position x_init[3];
  parameter Modelica.SIunits.Velocity v_init[3];
protected
  outer function gravity=GravityField;
  Modelica.SIunits.Position x[3](start=x_init);
  Modelica.SIunits.Velocity v[3](start=v_init);
  Modelica.SIunits.Acceleration a[3];
  Sphere p(x=x[1],y=x[2],z=x[3],R=0.01); // Dymola specific
equation
  v = der(x);
  a = der(v);
  a = gravity(x);
end Particle;
```

Example 9.2. A particle model that uses dynamic scoping.

To see how this is done, let us look at the `Particle` model shown in Example 9.2. The variables `x`, `v` and `a` in Example 9.2 represent the position, velocity and acceleration of the particle, respectively. The declaration of the `Sphere` model is a Dymola specific enhancement that will allow a 3D animation of the particle. If you are working with another simulator that does not support animation in this way, simply comment out the `Sphere` declaration.

The gravitational field is represented by the `function` named `gravity` which is an extension of the `GravityField` function shown in Example 9.1.

Note that the `GravityField` is only a `partial` definition. In this case, the use of the `GravityField` function establishes a minimum requirement for the `gravity` function.

What is new in this example is the use of the `outer` keyword. When the `outer` keyword is placed in front of a declaration or definition it means that whatever is being declared or defined does not really exist within that model. Instead, the declaration or definition should match a similar declaration or definition in the instance hierarchy. In the case of the `Particle` model, the `gravity` function needed by the `Particle` model must be found somewhere within the enclosing models (*i.e.,* the models that contain the `Particle` declarations).

9.1.2.2 Orbiting particles

Trying to describe exactly how dynamic scoping works is difficult without looking at a concrete example. So, let us now try to construct a complete model. First, we must define the gravitational field to be used. Example 9.3 shows the gravitational field created by two masses located at $(0,0,0)$ and $(0,1,0)$, respectively.

```
function TwoBodyField
  extends GravityField;
protected
  Modelica.SIunits.Position b1[3], b2[3];
  Modelica.SIunits.Distance n1[3], n2[3];
algorithm
  b1 := {0,0,0};
  b2 := {0,1,0};
  n1 := -(x-b1)/sqrt((x-b1)*(x-b1));
  n2 := -(x-b2)/sqrt((x-b2)*(x-b2));
  g := n1/((x-b1)*(x-b1))+n2/((x-b2)*(x-b2));
end TwoBodyField;
```

Example 9.3. Gravitational acceleration generated by two bodies.

Imagine we wish to model three particles moving within the gravitational field described in Example 9.3. Setting up such a problem requires two steps. First, we must declare the three particles and their initial positions and velocities. Second, we must provide a `gravity` function that these particles can use. In other words, we must provide the `gravity` function that each of these particles requires but does not contain. It is not sufficient to simply create a function called `gravity` for this purpose. We must also qualify the declaration of the `gravity` function with the `inner` keyword so that it "matches" with the `outer` declaration inside the `Particle` definition. The

resulting model is shown in Example 9.4. As noted previously, the `Sphere` declarations are Dymola specific enhancements that can be commented out if Dymola is not being used.

```
model ParticleField
  inner function gravity=TwoBodyField;
  Sphere b1(x=0,y=0,z=0,R=0.05); // Dymola specific
  Sphere b2(x=0,y=1,z=0,R=0.05); // Dymola specific
  Particle p1(x_init={2,-2,0},v_init={0.7,0,0});
  Particle p2(x_init={0,0.5,0},v_init={-1,-1,0});
  Particle p3(x_init={0.5,2,0},v_init={-1,-0.5,0});
end ParticleField;
```

Example 9.4. Particles orbiting two bodies in interesting ways.

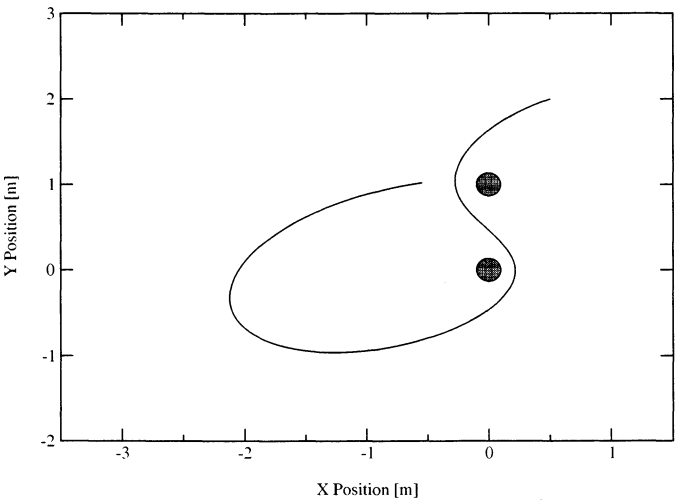

Figure 9.2. Trace of `p3` in Example 9.4.

9.1.2.3 Philosophy of dynamic scoping

The general idea behind `inner` and `outer` is to allow components to locate information contained higher up in the instance hierarchy. In our example, the `ParticleField` model contained the `gravity` function and several instances of the `Particle` model. Each `Particle` was able to use the `gravity` function defined in `ParticleField`. In other words, the sub-components (*i.e.*, the particles) were able to locate the necessary information

(*i.e.*, about the gravitational field) higher up in the instance hierarchy (*i.e.*, in the `ParticleField` model).

One way to think about the `inner` and `outer` keywords is as a statement of implicit requirements. When you see `outer` in front of a declaration think of that as a requirement (*i.e.*, the `Particle` model requires a function called gravity that is a subtype of `GravityField`). In a similar way, the `inner` qualifier indicates that this declaration *provides* something that might be required from subcomponents. The type of the `outer` declaration must be a subtype of the type used in the `inner` declaration in order for them to match. Furthermore, the `outer` declaration is not allowed to have any modifications applied to it.

9.1.2.4 Use of dynamic scoping

Dynamic scoping can be used with any declaration. For example, you might wish to create an electrical circuit that has an `outer` declaration for an electrical connector which represents the electrical ground of its environment (*i.e.*, by placing the `outer` qualifier in front of the declaration of the `Ground` component, `g`, in the `RLC3` model shown in Example 3.7). In this way, the circuit could implicitly be connected to the electrical ground of its environment (assuming there was a corresponding `inner` declaration somewhere in its instance hierarchy). Another example is having an `outer` declaration for parameters so that they are automatically available from the environment. However, both of these examples can also be written using the methods in Section 4.3.2.

The issue is whether the association is explicit, as it is using the methods in Section 4.3.2, or whether it should be implicit as it is with dynamic scoping. Implicit associations are nice because they can eliminate the task of propagating connections and parameters through complex hierarchies. The danger with such implicit associations is that it is not obvious what is being associated when looking at models for the first time. Furthermore, they may create difficulties when trying to debug and validate models. Such considerations should be kept in mind when choosing between the implicit and explicit approaches.

9.2 ANNOTATIONS

After going through the preceding chapters you might wonder where all the nice graphical schematics came from and what relation they have to the models themselves. The answer is that the graphical layout of a model and the graphical appearance of the components is achieved using the `annotation` keyword.

Annotations are used to provide additional information about the model that is **not related to the behavior of the model during analysis**. Annotations are typically used to embed documentation and graphical information inside a model. Annotations were not shown in the previous examples or discussed

earlier because they do not affect behavior and because they are generally inserted and interpreted by the analysis tool. A model developer rarely sees the textual representation of the annotations.

There are two possible locations for an `annotation`. The first location is following a declaration. In this case, the annotation will appear near the end of the declaration. The other location is within the body of a definition without association to any internal components.

Because the focus of this book is on modeling and because annotations are usually inserted and interpreted by simulation tools, we will present explanations for only a few of the different annotations. For more information on this subject, consult the Modelica language specification.

9.2.1 Graphical annotations

```
model PendulumSystem1 "Simple Pendulum"
  annotation (
    Coordsys(
       extent=[-100, -100; 100, 100],
       grid=[2, 2],
       component=[20, 20]),
    Window(x=0.13, y=0.13, width=0.6, height=0.6),
    Icon(Rectangle(extent=[-100, 100; 100, -100],
       style(color=0, fillColor=8)),
       Text(extent=[-74, 54; 76, -14],
           string="Double Pendulum", style(color=0))));
  RotationalPendulum pend
     annotation (extent=[-100, -40; 0, 60]);
  FrictionlessPin pin annotation (extent=[-20, -10; 20, 30]);
  Modelica.Mechanics.Rotational.Fixed fixed
     annotation (extent=[40, -10; 80, 30]);
equation
  connect(pend.p, pin.a) annotation (points=[-50, 10; -20, 10]);
  connect(pin.b, fixed.flange_b)
     annotation (points=[20, 10; 60, 10]);
end PendulumSystem1;
```

Example 9.5. A Modelica model with annotations.

The first `annotation` in Example 9.5 is a description of how the `PendulumSystem1` model should be represented graphically. In other words, it contains information about what drawing primitives should be used to construct the external representation for `PendulumSystem1` (*i.e.*, an "icon" view like the one shown in Figure 4.2). This information applies to the definition and therefore it is associated with all instances.

The other annotations in Example 9.5 are used to construct the diagram view for the system. These annotations describe the placement of components and connections. These annotations are then used to generate the diagram shown in Figure 9.3. Note that the `extent` annotations associated with components only provide a "bounding box" in which to draw the graphical icon of the components. The location of the icon is specific to each declaration but the actual description of what the icon is composed of is contained within the definitions of the component models themselves (as in the case of the first `annotation` in Example 9.5 described earlier).

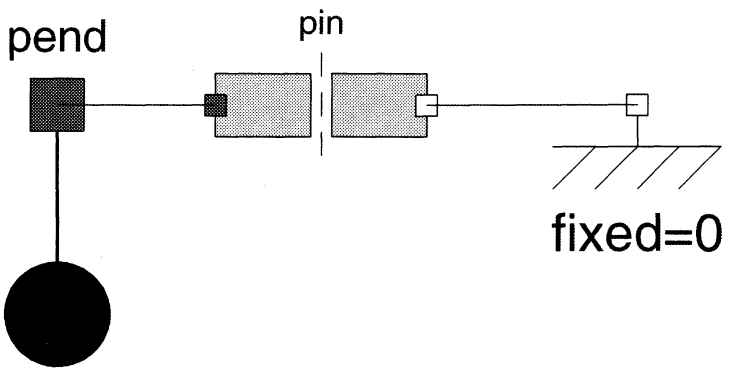

Figure 9.3. Schematic for pendulum system.

9.2.2 Documentation

Apart from descriptive text like that shown in the `TwoTanks` model from Example 2.4, there is one other method for documenting Modelica models. A specific annotation exists to allow developers to embed HTML descriptions in their models. Example 9.6 shows an alternative form of Example 2.4 where such annotations are used.[2]

Figure 9.4, generated using Dymola, shows a rendering of the HTML source code generated as documentation. Note how the descriptive text and the annotations have been merged. Also note the units and default values for parameters are also included. The point is that Modelica models contain a great deal of information (particularly if documentation annotations have been included) and this allows Modelica tools to automatically generate excellent HTML documentation for individual models or even generate documentation for complete libraries of components.

[2]The text of the model has been removed for brevity.

```
model TwoTanks "Hydraulic system involving two tanks"
  ...
  // Parameters
  parameter SI.Length L=0.1 "Pipe length";
  ...
  annotation (
    Documentation(info="<HTML>
This component represents two tanks connected by a
pipe. The constitutive relationship of the pipe is
the Hagen-Poiseuille relationship:

Q = (P1-P2)*(pi*D^4)/(128*mu*L)...</HTML>"));
equation
  ...
end TwoTanks;
```

Example 9.6. Using annotations for documentation

TwoTanks

Hydraulic system involving two tanks

Information

This component represents two tanks connected by a pipe. The constitutive relationship of the pipe is the Hagen-Poiseuille relationship:

Q = (P1-P2)*(pi*D^4)/(128*mu*L)

This component was originally developed to demonstrate how to include descriptive text in models. Subsequently, it was reused as an example of how to embed HTML documentation in models.

Parameters

Name	Default	Description
L	0.1	Pipe length [m]
D	0.2	Pipe diameter [m]
rho	0.2	Fluid density [kg/m3]
mu	2e-3	[Pa.s]
A1	1.0	Area of left tank [m2]
A2	2.0	Area of right tank [m2]
c	(pi*D^4)/(128*mu*L)	[m2/s]

Modelica definition

```
model TwoTanks "Hydraulic system involving two tanks"
  .
  .
  .
end TwoTanks;
```

HTML-documentation generated by Dymola Sun Nov 5 07:57:23 2000 .

Figure 9.4. Dymola rendering of HTML documentation for the `TwoTanks` model shown in Example 9.6.

II
EFFECTIVE MODELICA

Chapter 10

MULTI-DOMAIN MODELING

10.1 CONCEPTS

This chapter presents several multi-domain system models. Multi-domain models are characterized by the fact that they have components belonging to different engineering domains. In this chapter, we will see models from the mechanical (both rotational and translational), electrical and thermodynamic domains. In addition, many of the examples contain block diagrams for some subsystems (*e.g.,* for control systems).

Unlike the examples in previous chapters which illustrated specific language features, the purpose of this chapter is to examine what can be done when we combine Modelica language features. Because the models presented in this chapter are so large, the complete Modelica models are not presented in the text of the book. In place of the Modelica source code, schematics are used to illustrate the structure of the models. The Modelica source code for these models can be found on the companion CD-ROM.

10.2 CONVEYOR SYSTEM

The example presented in this section was inspired by a previously published example (Elmqvist et al., 1998) which nicely illustrates the Modelica modeling language. The model is composed of a control system and a plant model. The plant model contains electrical and mechanical components. Although the plant model for the conveyor system is more detailed than the `ControlSystem1`, `ControlSystem2` and `ControlSystem3` models found in Examples 3.9, 3.16 and 3.17, the architecture is very similar.

10.2.1 Mechanical load

The plant model is composed of electrical and mechanical components. We start by looking at the mechanical load. In this example, the mechanical load represents the force required to move a "product" along a conveyor belt in a factory. As shown in Figure 10.1, the schematic of the mechanical load includes a gear, an inertia, damping, the motion of the belt and the mass of the product. All the models shown in the schematic are linear (*i.e.*, no backlashes or other non-linearities).

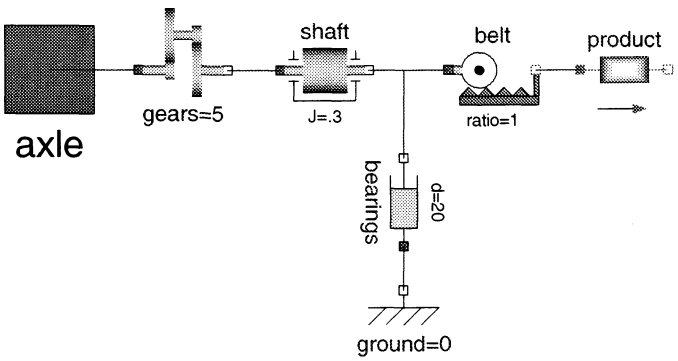

Figure 10.1. Schematic for the conveyor belt system.

10.2.2 Electric motor

The other part of the plant model consists of the electric motor used to drive the conveyor belt. The motor model, shown in Figure 10.2, consists of a resistor, inductor, electro-magnetic torque source, and inertia. Clearly, more complicated models for electric motors may be more realistic than the one shown here. Any model with the same interface as this one (*i.e.*, two electrical pins, `p` and `n`, and a rotational pin, `driver`) could be easily dropped in as a replacement for this one.

10.2.3 Control system

Figure 10.3 shows a schematic of the control system used to control the motor and conveyor belt. The desired response for this controller would move the product along the conveyor belt from station to station in the factory. The product would then pause at each station for some amount of time in order for operations to be performed on the product. This desired response is fed to the PD controller which calculates the voltage to be supplied to the motor.

Again, several idealizations have been made. For example, the voltage source which supplies power to the motor is an ideal voltage source capable of

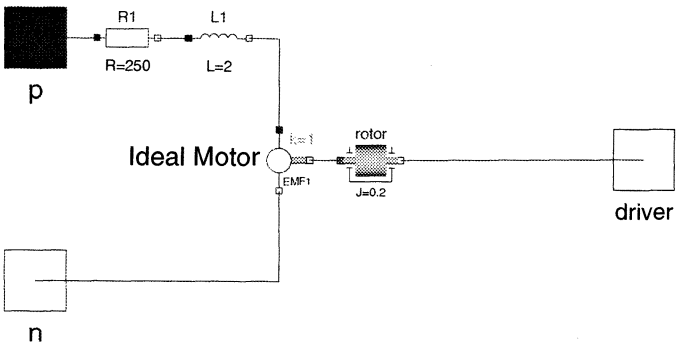

Figure 10.2. Schematic for the electric motor.

delivering as much current as required. Also, the required voltage computed by the control system does not contain any quantization error which might be present as a result of digital to analog conversions. These are just two examples of details that may have an effect on the performance of the system but could be modeled within the controller itself (*i.e.*, without having to change any of the other models).

One last point to mention about the controller model is that it has inputs for both position, `phi`, and speed, `omega`. This controller does not use the speed information. Instead, the speed is approximated internally by differentiating the position signal. Nevertheless, there are two reasons for having the speed input. First, it allows the option of changing the internal implementation of this controller without requiring any "wiring" changes of the factory level schematic (see Figure 10.4). The other reason to have a speed sensor input is that it would make it easier to substitute controllers which did actually utilize the speed input (similar to the approach discussed in Section 4.3).

10.2.4 Complete system

Finally, we bring together all the component models described so far in this section. The combination of these components (along with the position and speed sensors) can be seen in Figure 10.4. This represents the complete system including both the controller and plant models.

One interesting thing to note about the plant model in the complete system is that a mechanical connection exists between the inertia inside the motor and another inertia inside the conveyor belt. In order to simulate the response of such a system, an "effective inertia" must be formulated for the combination of the two rigidly connected inertias. When inertias are connected in this way, it is not generally sufficient to merely add the inertias together. Instead, some algebraic manipulation is necessary in order to compute the effective inertia.

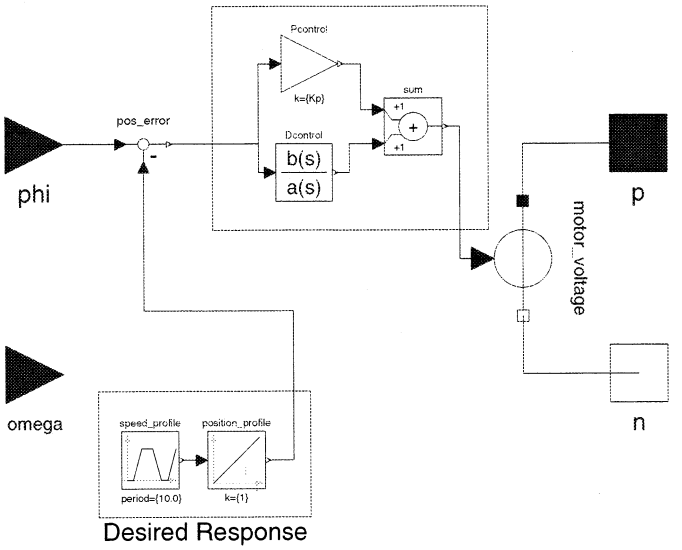

Figure 10.3. Schematic for the conveyor controller.

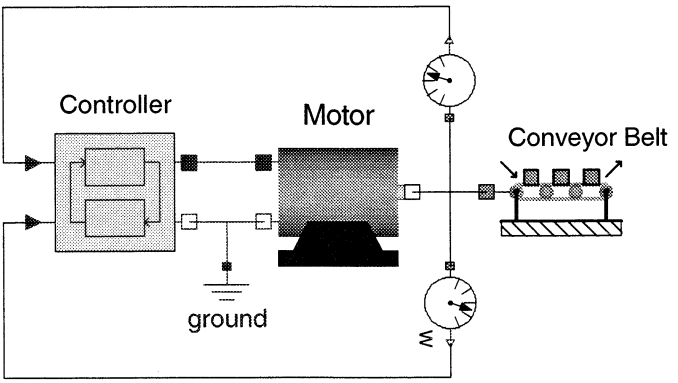

Figure 10.4. Schematic for the factory.

Fortunately, when using Modelica, this work does not have to be done by the developer of the model, but instead will be done by the tool that analyzes the system.

If we look at the operation of this factory for 100 seconds of simulation time, as shown in Figure 10.5, we see that the controller does a good job of moving the product along the conveyor belt as intended. In Figure 10.6, we can see the motor voltage required in order to achieve this level of control.

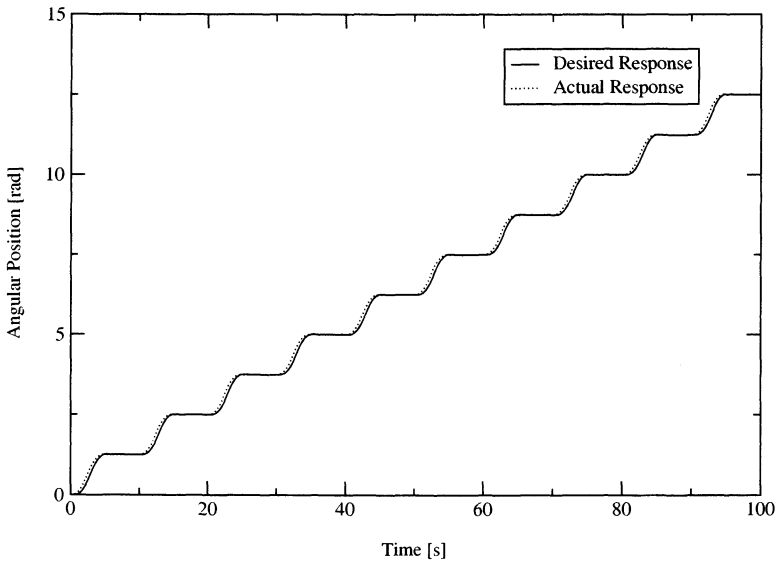

Figure 10.5. Comparison of desired vs. actual factory behavior.

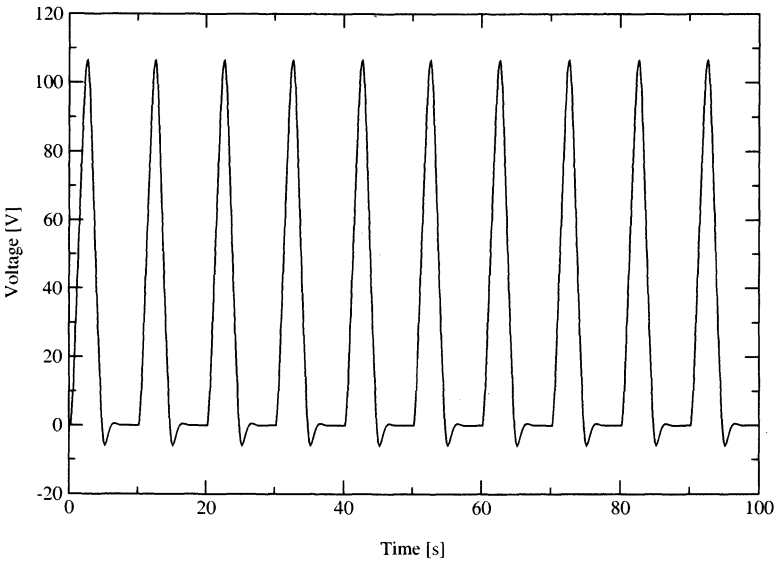

Figure 10.6. Motor voltage required.

10.3 RESIDENTIAL HEATING SYSTEM

10.3.1 Introduction

Another example of a mixed-domain problem is a residential heating system. To develop such a model, we utilize components from the rotational and electrical packages, which can be found in the `Modelica.Mechanics.Rotational` and `Modelica.Mechanics.Translational` packages in the MSL. While the majority of our models came from the MSL, a few models had to be developed from scratch for this particular application. For example, several of the examples use the `Thermal` package introduced in Chapter 6. In the remainder of this section, we will describe how to build such a system, what models we used from the MSL and what models we needed to create ourselves.

10.3.2 Indoor temperature

We start by building a model to compute the temperature inside a house. The model for indoor temperature is interesting because it makes use of all three fundamental modes of heat transfer: conduction, convection and radiation. We call this the `House` model and a schematic of if can be seen in Figure 10.7.

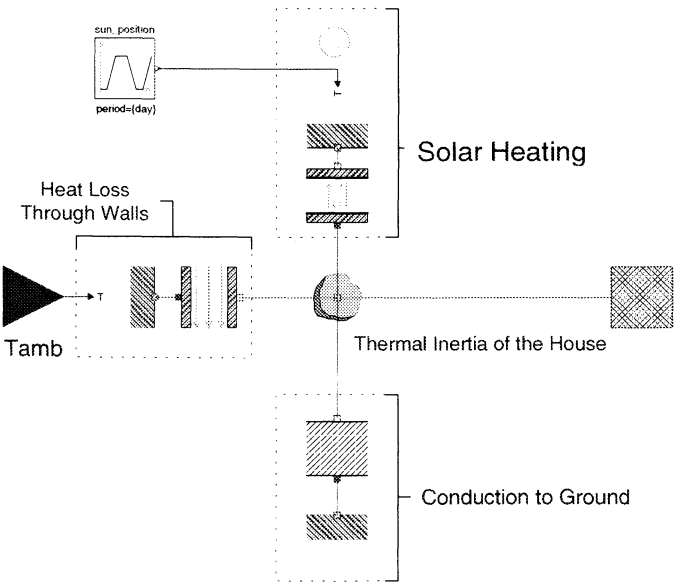

Figure 10.7. Schematic for the `House` model.

First, we must establish the geometry of the `House` model. Let us assume that our `House` has only one floor and the floor area is given by the `footprint`

parameter. We assume a default of 250 square meters. In addition, we must provide the `height` of the house. For this, we assume a default of 4 meters. Our last assumption is that the footprint of the `House` is square. We can then compute the surface area of the surrounding walls, the roof and the floor directly from the given parameters. In addition, we can compute the volume of the house as well from these same parameters.

Once we have established the geometry of the house, we next consider the thermal inertia of the house. The thermal inertia is the amount we raise the temperature of the house for every unit of heat we generate from our furnace. For this, we use the `Capacitance` model from the `Thermal` library which is similar to the one shown in the `ThermalCapacitance` model in Example 6.12. This model requires three pieces of information. The first is the volume of the house, which we already know from our geometry parameters. The second is the density of the air, ρ, inside the house, which we assume to be $1.5\ kg/m^3$. Finally, the specific heat capacity of the air is assumed to be $1000\ J/(kg \cdot K)$. By using a single capacitance to represent the entire house we are implicitly assuming that the entire house has a single uniform temperature.

Next, we turn our attention to the roof of the house. We assume that the primary mode of heat transfer for the roof is radiation.[1] We assume an effective temperature at which the house radiates to its environment. This effective temperature changes from day to night and therefore we have used a signal block to generate this temperature, which is then used to establish a thermal boundary condition for the sky. This boundary condition is then attached to the house through a black body radiation component.

We now turn our attention to the floor of the house. We assume that some heat is lost via conduction through the floor to the ground. Our default value for thermal conductivity of the ground is $0.4\ W/(m \cdot K)$. Furthermore, we assume that the ground temperature 4 meters below the surface is $280K$.

Finally, the only mode of heat transfer left is convection. We assume that this is the dominant mode of heat transfer through the walls of the house. Just like with the roof model, we need to have an ambient air temperature for the convection. We assume that the ambient air temperature is provided externally. The default value for the heat transfer coefficient of the walls was chosen to be $4.33\ W/(m^2 \cdot K)$.

In summary, this is a crude thermal model of a house but it demonstrates how a fairly complex model can be constructed using existing models.

[1] We make this assumption in the interest of keeping our example simple. It should be noted that this is probably not a good assumption. Convection would probably be quite significant across the roof as well.

10.3.3 Furnace

Notice that the schematic of the `House` model shown in Figure 10.7 includes only environmental influences. In other words, the only heat transfer is due to its surroundings; internal influences were neglected. In order to introduce the influence of an electric furnace, we must build a `Furnace` model and connect it to the house. Figure 10.8 shows a schematic of the `Furnace` used in this example.

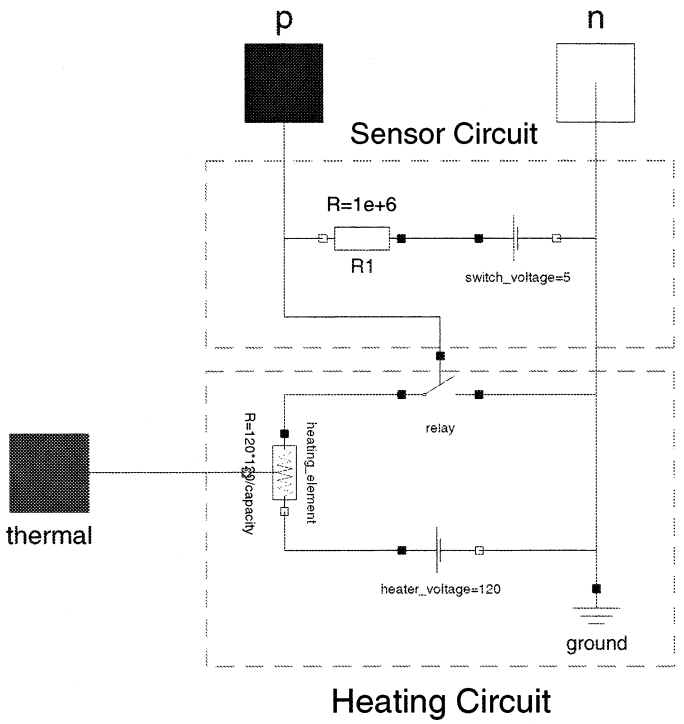

Figure 10.8. Schematic for the `Furnace` model.

The `Furnace` model contains both a thermal interface and an electrical interface. The `thermal` connector connects the furnace to the house so that the energy generated by the furnace influences the temperature of the house. The electrical interface, which is composed of the two electrical pins `p` and `n`, is used to turn the furnace on or off. The decision to turn the furnace on or off comes from the thermostat which we will describe shortly. If the thermostat closes the circuit by connecting the two electrical pins, `p` and `n`, the furnace will turn on. When the pins are disconnected, the furnace will shut off. We assume the capacity of the furnace (*i.e.*, the amount of thermal power it can generate) is provided by the `capacity` parameter. The interesting feature

of the `Furnace` model is that even though it is used for generating thermal energy, it is largely an electrical component.

The furnace is composed of two circuits. One is the sensor circuit which is connected to the thermostat to indicate whether the furnace should be on or off. The other circuit is a high-power circuit (*i.e.,* not a circuit you would want connected to other low-power, wall mounted devices like thermostats). The bridge between these circuits is a relay. Finally, the high-power circuit contains the `HeaterElement` model from the `Thermal` library to be used as both a resistor and as a source of thermal energy.

10.3.4 Thermostat

As mentioned earlier, when describing the `Furnace` model, a thermostat model is required in order to control the `Furnace`. Models for two different types of thermostats will be created. One model is the traditional mechanical thermostat and the other is a modern digital thermostat. Looking at both types is interesting because while both are mixed-domain devices, they mix different sets of domains.

10.3.4.1 Mechanical thermostat

Figure 10.9 shows a schematic of a mechanical thermostat. Traditional mechanical thermostats are controlled by turning a dial on the outside of the thermostat to indicate the desired temperature inside the house. In other words, the user's interaction is mechanical (*i.e.,* they turn something). Internally, temperature sensitive mechanical components are used and as the outside temperature changes, the internal mechanism moves. This movement is reflected in the temperature reading given by the thermostat.

In our `MechanicalThermostat` model, the point at which the furnace should be turned on is given as a temperature. Strictly speaking, it should be given as the angular position of the thermostat dial but this simplification has been made to avoid the additional complexity of modeling the control mechanism. Instead, once the desired temperature is provided via the `desired` parameter, an internal calculation is made to determine the rotational angle which corresponds to that temperature (*i.e.,* the dial setting).

The schematic in Figure 10.9 contains several noteworthy components. First, there is the temperature sensitive rotational spring. This can be identified by the fact that it has both rotational and thermal connections. The unstretched length of the spring changes as a function of temperature. Since the inertia of the mechanism is small, these slight changes in unstretched length result in a nearly immediate change in the position of the various components of the mechanism (at least compared to the time scale of the thermal system).

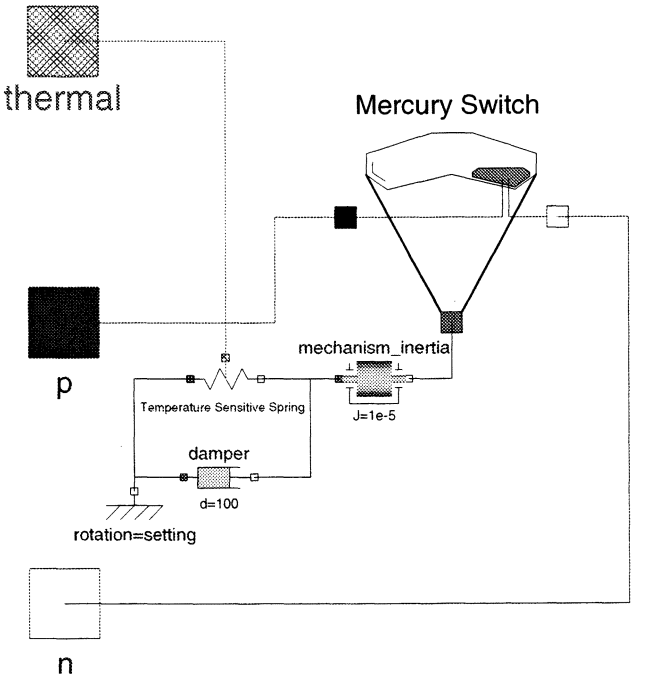

Figure 10.9. Schematic for the `MechanicalThermostat` model.

A damper is introduced because the combination of the spring and mechanism inertia introduces high frequency modes in the mechanical system. The damper acts to damp out these modes.

Finally, the internal mechanism includes a mercury switch. The switch is used to turn the furnace on or off. When the switch is rotated far enough, the mercury will move from one end of the switch to the other. An interesting thing to note about the switch (which is essential for its application in a thermostat) is that the switch has hysteresis built into it. For example, assume that the mercury switch turns on when the temperature goes below $300K$. When the switch turns on the mercury moves to one side of the switch and closes the connection between two wires. However, when the temperature rises above $300K$ the switch does not turn off. Instead, the temperature must go over some other threshold (*e.g.*, $305K$) before the switch will turn off. This is essential because if only a single temperature determined the state of the switch there would be a great deal of chatter (*e.g.*, high frequency changes in switch state).

10.3.4.2 Digital thermostat

In contrast to the mechanical thermostat, we now consider how to model a digital thermostat. Interestingly enough, even though the domains and technologies are different, a number of the same issues are present in both. Figure 10.10 shows a schematic of the `DigitalThermostat` model.

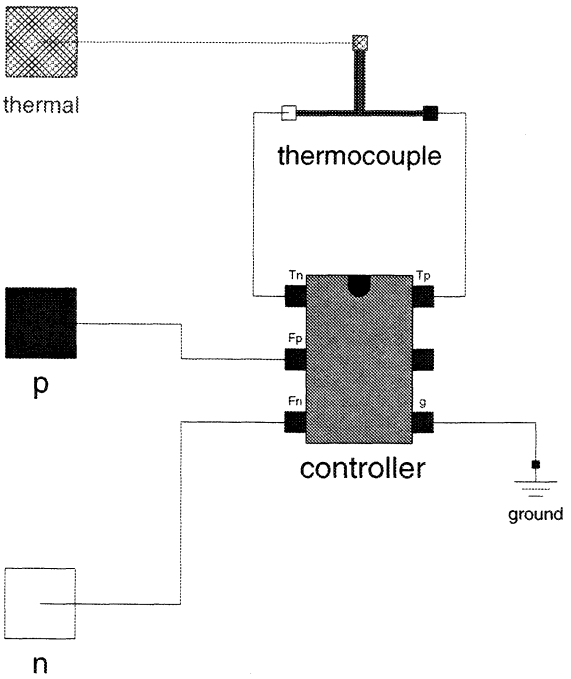

Figure 10.10. Schematic for the `DigitalThermostat` model.

The `DigitalThermostat` consists of two components. The first is a thermocouple which translates temperature differences into voltages. The thermocouple model found in the `Thermal` library assumes that the voltage difference is computed as a polynomial of the form:

$$\Delta V = \sum_{i=0}^{n} C_i T^i \tag{10.1}$$

where ΔV is the voltage drop across the thermocouple, T is the ambient temperature around the thermocouple and C is a series of coefficients which are parameters to the thermocouple model.

The other model used in the `DigitalThermostat` model is a digital controller circuit. It is assumed that the logic of the control strategy has been encapsulated within this circuit. In our case, the `DigitalCircuit`

model which represents the digital controller has a simple `algorithm` section which controls the electrical behavior between the furnace control pins, `Fp` and `Fn`, based on the voltage drop across the thermocouple pins, `Tp` and `Tn`. In order for the control system to function, it must translate the voltage drop across the thermocouple back into a temperature. To do this, it must have some understanding of the characteristics (*i.e.,* the C_i coefficients) of the thermocouple. These same coefficients are therefore used as calibration parameters for the controller.

10.3.4.3 Commonality

There are some interesting parallels between the digital and mechanical thermostats. First, they both require calibration. The gauge on the outside of the mechanical thermostat must be calibrated based on the thermal expansion characteristics of the spring inside the thermostat. Likewise, the control circuit in the digital thermostat must be programmed with the characteristics of the thermocouple's response to temperature changes.

It is quite typical in controllers to find some representation of the plant (*i.e.,* the physical system) response present inside the controller. In our cases, the models have been constructed in such a way that the controller's understanding of the physical response is "perfect" (*i.e.,* it knows exactly what is happening in the physical system). This is highly unlikely in the real world for at least two reasons. One reason is that the calibrations are never perfect (*e.g.,* the coefficients used by the controller have some error). The other reason is that the mathematical model that the controller uses to represent the physical system is not perfect either. **Studying such errors and what effects they have when a control system is actually deployed is an important aspect of modeling physical systems.**

The other common aspect of these controllers is the requirement for hysteresis. In the mechanical system, the hysteresis is generated by the geometry of the mercury switch. For the digital system, the hysteresis is introduced by the control algorithm.

10.3.5 Complete System

Figure 10.11 shows a system which exercises both the digital thermostat and the mechanical thermostat. Figure 10.12 shows the simulated temperature inside and outside the house. The devices have been setup in such a way that they behave identically so only one indoor temperature is shown. Note that the frequency with which the furnace is used is greater when the outside temperature is lower.

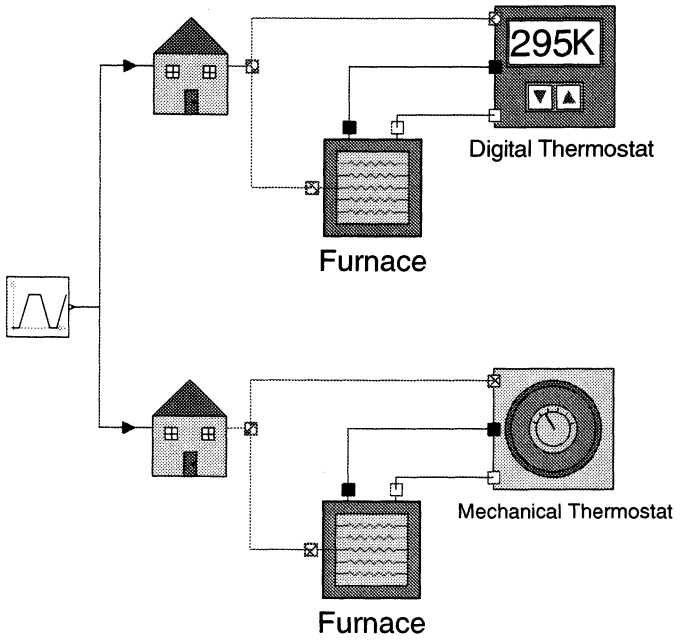

Figure 10.11. Schematic for the `ThermostatSystem` model.

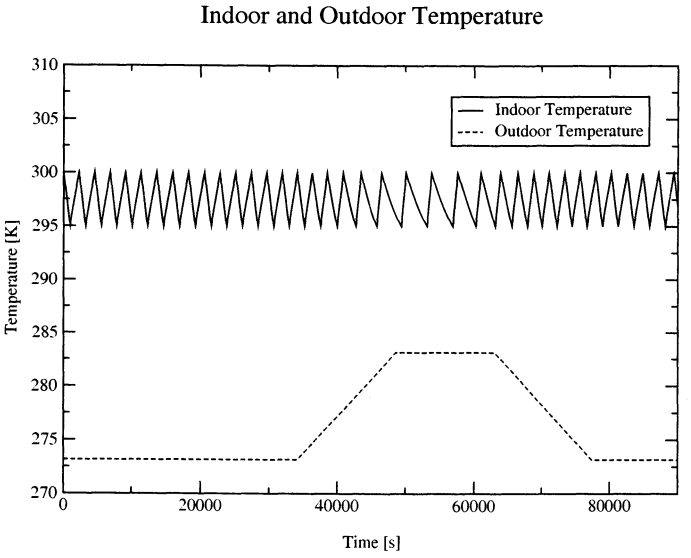

Figure 10.12. Indoor and Outdoor temperature.

10.4 AUTOMOTIVE LIBRARY

In Chapter 1, we briefly mentioned a library of models developed to evaluate vehicle performance. In this section, we will explore that library in more detail.

10.4.1 SimpleCar package

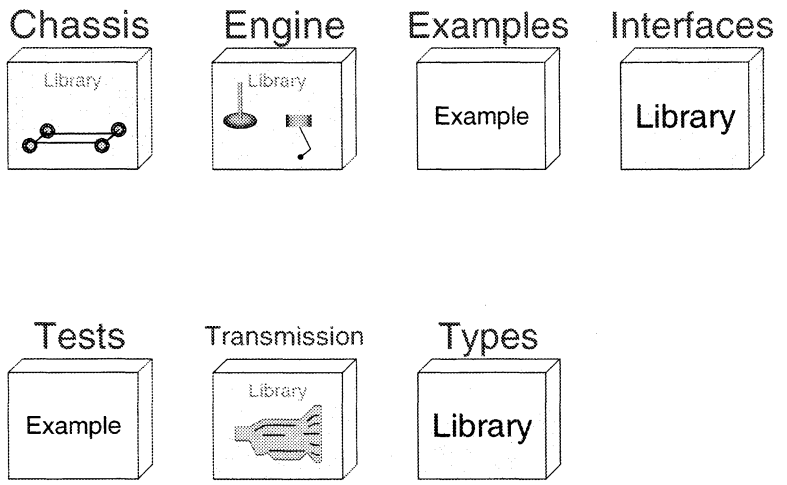

Figure 10.13. Packages nested inside the `SimpleCar` package.

As we can see in Figure 10.13, the `SimpleCar` library contains several nested packages. These will be discussed in detail shortly. The overall structure of the package follows the conventions which will be laid out in Chapter 12.[2]

The models contained in this package are relatively simple. As a result, their predictive capabilities are very limited. However, the package provides many opportunities for trying out different design ideas, as well as incorporating new component models.

10.4.2 Engine package

The `Engine` package contains models related to the function of the engine. Engine design parameters represented by these models include bore, stroke, valve diameter, valve timing, *etc.*

10.4.2.1 Interfaces

Many of the engine component models use the rotational and translational connectors found in `Modelica.Mechanics`. In addition, most components

[2]For example, the nested `Types` and `Interfaces` packages are present.

also use the Gas[3] connector found in SimpleCar.Interfaces which is defined as:

```
connector Gas "Thermodynamic connector"
  Modelica.SIunits.Pressure P "Gas pressure";
  Modelica.SIunits.Temperature T "Gas temperature";
  flow Modelica.SIunits.MassFlowRate mdot "Mass flow rate";
  flow Modelica.SIunits.HeatFlowRate q "Heat flow rate";
end Gas;
```

This connector, used to represent the state of the air-fuel mixture, is somewhat unusual because, unlike most of our previous connector definitions, it contains two across and two through variables. The across variables (*i.e.*, the potentials which drive the dynamics of the system) are pressure and temperature, P and T, while the through variables are mass flow rate, m_dot, and heat flow rate, q.

In addition to the Gas connector definition, there are several partial models representing generic interfaces for engines, transmissions, shift strategies and chassis. These interfaces are used in conjunction with vehicle models to allow easy replacement of models for these subsystems with other models satisfying the same interface requirements.

10.4.2.2 Basic components

Figure 10.14 shows what we find when we open up the SimpleCar.Engine.Components library. The following is a list of some of the components and a description of the model used:

- TimingBelt: The timing belt is a very simple device. The model for a timing belt is the same as the model for a gear with a gear ratio of two. The timing belt is used to rotate the camshaft at exactly half the speed of the crankshaft.

- ChamberVolume: During the engine cycle, it is necessary to calculate the volume inside the combustion chamber. The ChamberVolume model is responsible for this calculation. Engine geometry information is passed into this model and the chamber volume is an output. In order to compute the volume, the ChamberVolume must also be connected to the piston through a translational connector in order to determine the piston position (used in the calculation of the chamber volume).

- CrankSlider: The crank-slider mechanism in the engine is used to transform the translational force on the piston into a torque on the crankshaft.

[3] "Gas" in this context does not represent gasoline but rather the gaseous state of the mixture.

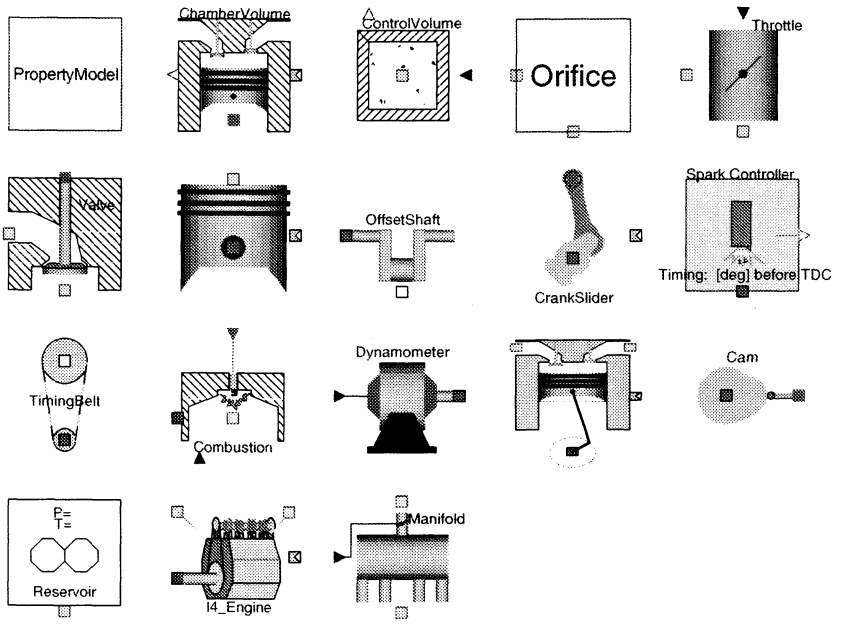

Figure 10.14. Components of the `Engine` package.

- `MasslessPiston`: The piston model is called `MasslessPiston` because we neglect the translational inertia of the piston for our analyses. The piston model itself balances the force due to pressure inside the combustion chamber with the force applied from a translational connector (presumably connected to the crank-slider mechanism).

- `ControlVolume`: At the center of the thermodynamic process of an engine is the control volume inside the combustion chamber. In addition to applying conservation of mass and energy to the contents of the control volume, the ideal gas law is also used as a constitutive equation to describe the relationship between pressure, volume and temperature. The default gas property model assumes a perfect gas (*i.e.*, $u = c_v T$ and $h = c_p T$).

- `Combustion`: The `Combustion` model used in our analyses is very simple. The combustion process is modeled as a release of energy based on the amount of mass trapped inside the combustion chamber. The instantaneous heat release, during combustion, is governed by the following equation:

$$q = Q_{\text{total}} \sin^2 \left(\pi \frac{t - t_s}{t_f - t_s} \right) \tag{10.2}$$

where t_s and t_f are the start and end times of the combustion process. These are determined at the time that the spark plug fires and are based on engine speed and burn duration in crank angle degrees. Furthermore, Q_{total} is determined using the lower heating value of the mixture, the air-fuel ratio of the mixture and the total mass trapped in the cylinder.

- `Valve`: The valve model is used for both intake and exhaust valves. The flow through the valve is governed by the standard isentropic flow relationship found in textbooks (*e.g.*, Ogata, 1978).

- `Cam`: The `Cam` model computes an idealized cam profile that is based on the maximum lift of the valve and the position of the crankshaft when the valve is intended to open and close.

- `Throttle`: The throttle model has the same underlying flow behavior as the engine valve model except that the throttle flow is controlled based on throttle angle rather than valve lift.

- `Manifold`: The manifold model is a simple "filling-and-emptying" model (*i.e.*, no wave dynamics). The equations for the manifold control volume are the same as the equations used for the combustion chamber.

- `Reservoir`: The `Reservoir` model is used to represent the ambient conditions (*i.e.*, an infinite reservoir of mass at a specified pressure and temperature).

- `Dynamometer`: The `Dynamometer` model is used to fix the speed of the engine. Dynamometers are often used in engine testing to determine the torque output of the engine. The behavior of the dynamometer is best described as: "The dynamometer generates whatever torque is necessary to keep the engine rotating at a specific speed."

10.4.2.3 Component assemblies

The `Engine` package contains more than just components. It also contains assemblies of those components. The first assembly to point out is the individual cylinder which is composed of the basic components already described. The schematic of an individual cylinder can be seen in Figure 10.15.

Just as we used the components described previously to build up the model of an individual cylinder shown in Figure 10.15, we can use the individual cylinder models to built up a complete engine. For example, a 4 cylinder engine model can be seen in Figure 10.16. Each of the cylinders you see in Figure 10.16 contains the components shown in Figure 10.15.

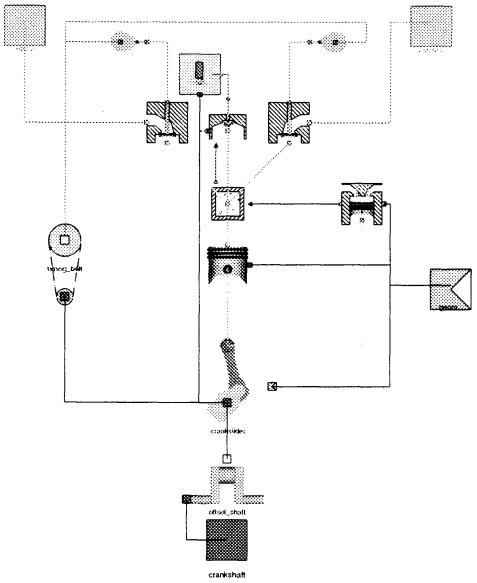

Figure 10.15. Looking inside an individual engine cylinder.

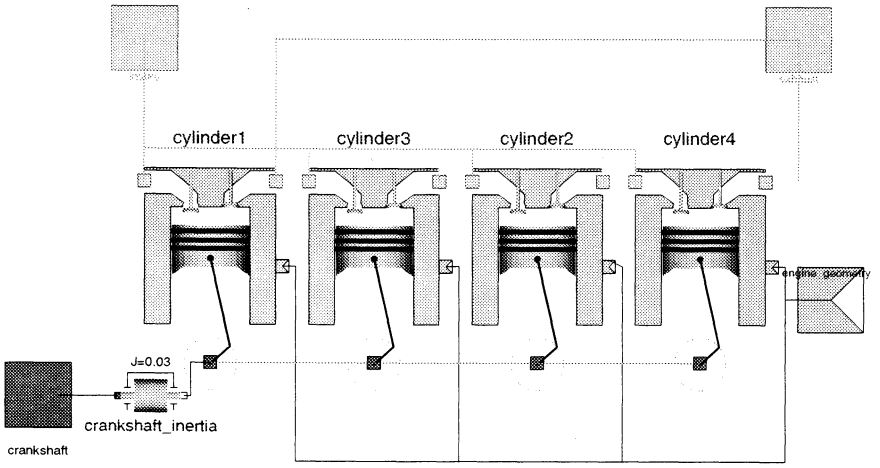

Figure 10.16. Looking inside a 4 cylinder engine.

10.4.3 Transmission package

For the engine component models, we had to develop quite a few new basic components. For the transmission, we can rely much more on the components

found in the MSL. As a result, the `Transmission` package is made up primarily of assemblies.

The five speed transmission model included in the `Transmission` package is very simple. It represents the function of a five speed automatic transmission in concept but is not assembled in the same way as a real automatic transmission (*e.g.*, the torque converter has been left out and numerous individual gears replace a smaller number of planetary gears). A schematic of the five speed transmission is shown in Figure 10.17. Using several internal equations, the model describes the simulated response of the underlying hydraulic subsystem used to actuate the clutches.

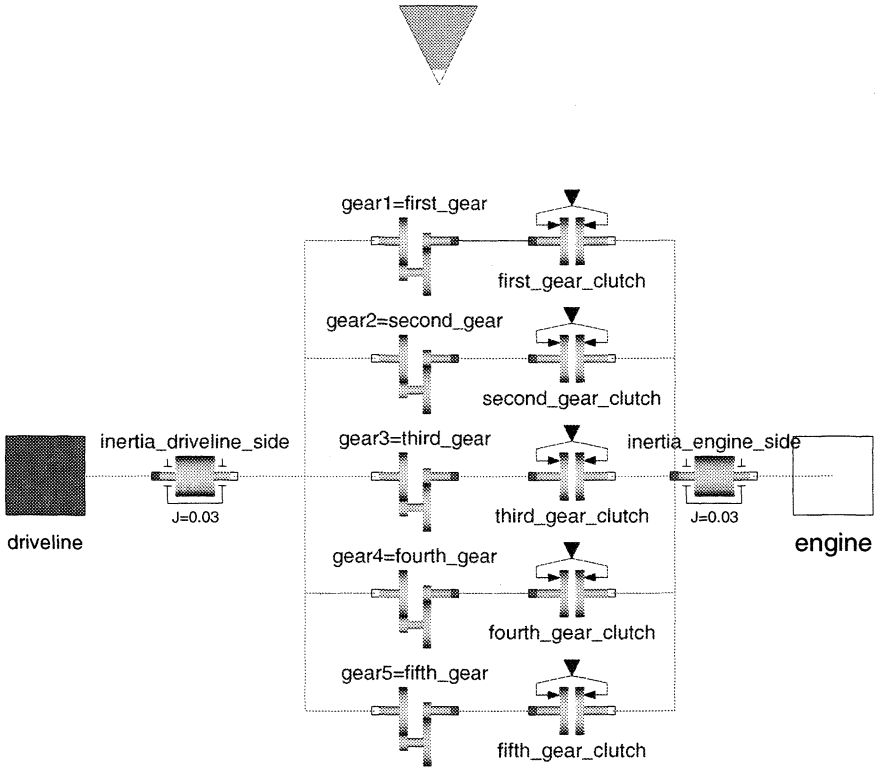

Figure 10.17. A simplistic five speed transmission.

10.4.4 Chassis package

The `Chassis` package contains components used to represent the frame and suspension of the car. A few very simple components are included (as shown in Figure 10.18). Just as with the transmission models, the chassis

components use the connector definitions and several components from the `Modelica.Mechanics` package.

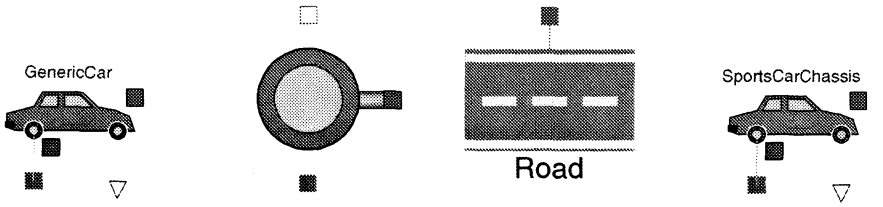

Figure 10.18. Contents of the `Chassis` package.

10.4.5 Vehicle package

In order to do vehicle level simulation (which is by no means the only use of the models presented in this section), we need to bring together assemblies from the other packages to form a complete vehicle. The particular combination used for our sports car example from Chapter 1 is shown in Figure 10.19.

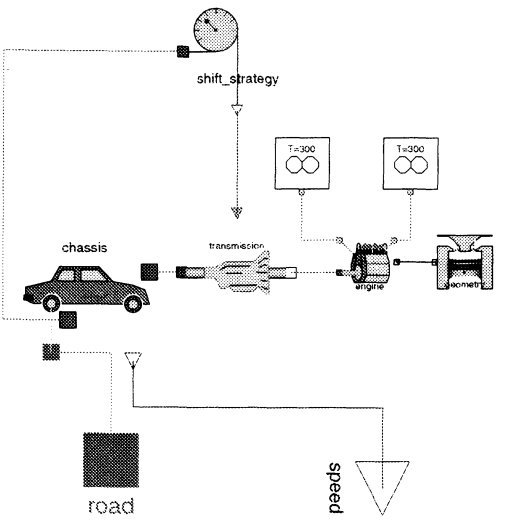

Figure 10.19. Creating a vehicle model.

10.4.6 Applications

10.4.6.1 Acceleration application

In Chapter 1, we described one possible application of the models presented in this section. Specifically, we modeled the acceleration of a sports car from 0 to 100 kilometers per hour. We used the model shown in Figure 10.19 as the vehicle model. One challenge in putting together such a model is starting with the proper initial conditions. Chapter 13 discusses how to specify the initial conditions for a simulation.

In the case of our acceleration test, the initialization was straightforward. We assume that the car starts with the transmission disengaged and the engine running at 1500 RPM.[4] At the moment the simulation starts, the transmission engages and the vehicle begins to accelerate.

We use a `when` clause to determine the point at which the vehicle has reached 100 kilometers per hour. Inside the `when` clause we use the `terminate()` function to stop the simulation.

10.4.6.2 Dynamometer testing

The vehicle acceleration test is a very nice example because everyone can relate to it. However, a more common use for models like the ones presented in this section is actually to evaluate steady-state engine performance or responses to simple transients by connecting the engine to a dynamometer. Figure 10.20 shows a schematic of such a test.

The purpose of the test is to run the engine according to some speed profile and determine the torque output of the engine under those conditions. Such tests simplify the evaluation of the engine by avoiding effects due to the other sub-systems (*e.g.*, the transmission or chassis).

10.4.7 Concluding remarks

The models in the `SimpleCar` package were developed as a "modeling playground" on which people learning the Modelica language could test their skills. There are endless possibilities for enhancement and modification of this library. The vehicle models contain numerous `replaceable` components so it is easy to make a design modifications. For example, an interested reader might try building a 6 cylinder engine to complement the 4 cylinder engine already provided. This 6 cylinder engine could then be used in place of the existing 4 cylinder engine by redeclaring the engine component in one of the vehicle models. Section 10.6 provides several modeling exercises related to the models presented in this section.

[4] If we had a torque converter in the transmission, we could have assumed that the transmission was engaged and applied the brake to keep the vehicle from moving initially.

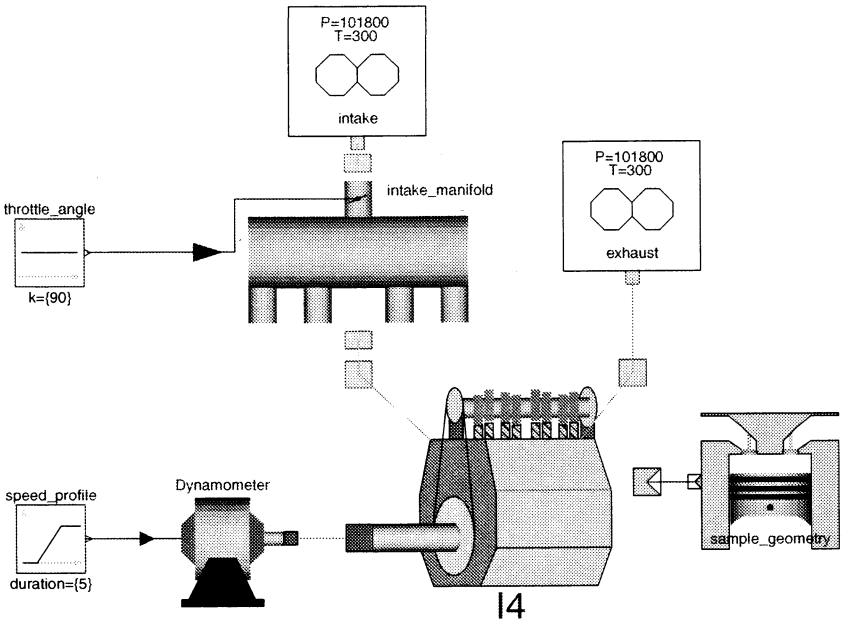

Figure 10.20. Top level model for dynamometer testing.

10.5 SUMMARY

In summary, we have presented several examples of how mixed-domain systems can be built using Modelica. In these cases, most of the models required were available as part of the MSL. The ability to build complex systems of both plant and controller models across domains is a powerful tool when used in conjunction with a "systems engineering" approach.

There are many potential applications for systems like the ones developed in this chapter. For example, simple plant models can be developed for the purpose of control system design. At a later stage, the plant models can be refined to include more detailed characteristics (*e.g.*, additional non-linearities) and used to verify the controller designs in the context of less idealized plant models. Another possibility would be to examine the effect of non-ideal sensors and actuators as we did in Section 7.4. Finally, the robustness of these systems can be tested by varying the physical characteristics of the plant or the calibration values used in the controllers to check if the system performance is extremely sensitive to slight variations in these parameters.

10.6 PROBLEMS

These problems, more so than the ones included in other chapters, demonstrate many of the practical tradeoffs made in industrial application of controller and plant modeling. Only a few of the possible complicating issues have been included in this section.

PROBLEM 10.1 *Replace the gear models in the conveyor belt used in the electric motor example with non-ideal gears which introduce backlash (e.g., due to gaps between the gear teeth). What effect does this have on the performance of the control system?*

PROBLEM 10.2 *Run the factory control example and look at the maximum voltage required by the controller using the current controller design. Then, change the model of the voltage source so that the voltage output of the controller is limited to some maximum voltage. Initially, set the maximum voltage for the controller above the maximum voltage used, and verify that there is no degradation in the performance of the system. Next, slowly lower the maximum voltage and observe the changes in the performance of the control system. Such saturation effects are quite common in modeling actuator behavior.*

PROBLEM 10.3 *Evaluate the effects of different circuit designs for the motors. For example, what is the effect of adding some capacitance to the motor or including operational amplifiers to provide gain to the input signal?*

PROBLEM 10.4 *Build a 6 cylinder engine using the 4 cylinder engine in Figure 10.16 as a guide.*

PROBLEM 10.5 *The individual cylinder model currently has only a single intake and single exhaust valve. Create an individual cylinder model with more valves to try to improve the overall torque output of the engine.*

PROBLEM 10.6 *Examine the effects of the various design parameters (e.g., valve timing and engine geometry) in the acceleration test. The full model name for the acceleration test is* `SimpleCar.Examples.Race`.

Chapter 11

BLOCK DIAGRAMS VS. ACAUSAL MODELING

11.1 OBJECTIVE

In this section, we will discuss, in detail, the differences between the block diagram and acausal approaches introduced in Section 1.3.

There are several analysis tools available which express system behavior in terms of block diagrams. Any given block in a block diagram has the following general form:

$$\dot{x} = f(t, x, u) \tag{11.1}$$

$$y = g(t, x, u) \tag{11.2}$$

where u represents the input signals, x represents the internal states and y represents the output signals. When connected together, such blocks are capable of representing and simulating large systems of differential equations (with the same general form).

Block diagrams are useful in understanding the mathematical behavior behind dynamic systems. For example, the Jacobians of the f and g functions can be used to determine the poles, zeros and overall transfer function for a linearized system. In addition, block diagrams are particularly well suited for describing control system structure.

However for describing plant model or physical system behavior, block diagram formulations take more work to create and are less reusable than acausal models. Modelica can be used for both approaches but an understanding of which approach is more appropriate will lead to more efficient model development. This chapter attempts to demonstrate some of the key differences in these two approaches.

Throughout this chapter, we will use the mechanical system shown in Figure 11.1 (Bowles et al., 2001) to demonstrate both block diagrams and acausal

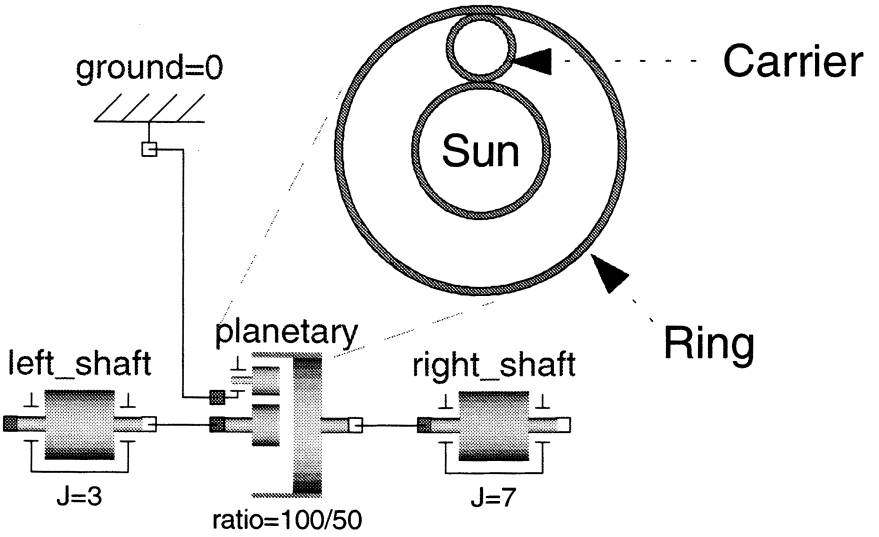

Figure 11.1. Grounded planetary gear with two inertias attached.

approaches. We assume idealized behavior for the components. For example, the behavior of the inertias in Figure 11.1 is described by the following equation:

$$J\dot{\omega} = \tau \tag{11.3}$$

where J is the moment of inertia, ω is the angular velocity of the shaft and τ is the torque applied to the shaft. Furthermore, the planetary gear behavior is expressed by the following equations:

$$\phi_s - \phi_c + R(\phi_r - \phi_c) = 0 \tag{11.4}$$

$$\tau_r = R\tau_s \tag{11.5}$$

$$\tau_c + \tau_s + \tau_r = 0 \tag{11.6}$$

where R is the ratio of ring gear teeth to sun gear teeth, ϕ_r is the angular position of the ring, ϕ_c is the angular position of the carrier, ϕ_s is the angular position of the sun, τ_r is the torque on the ring gear, τ_c is the torque on the carrier gear and τ_s is the torque on the sun gear.

Even though we have chosen simple behavioral equations, we will show that the problem is actually more complex than might first appear.

11.2 BLOCK DIAGRAMS

In Section 1.3.1, we first introduced the block diagram approach and showed a block diagram of a simple control system. The Modelica features needed to

develop block diagram components were then presented in Section 3.4. Finally, Equations (11.1) and (11.2) show the general form for the equations of both an individual block and a complete block diagram.

The block diagram approach is an elegant way of representing mathematical behavior because it is simple and easy to understand. In addition, a block diagram of a system is good for understanding the mathematical structure of the problem. Mathematical operators such as addition, multiplication and integration appear explicitly in such diagrams. These systems are more intuitive to debug because the behavior is explicitly described.

11.2.1 Problem statement

In order to understand some of the drawbacks of the block diagram approach, let us construct a model of the system shown in Figure 11.1. The first step in building a block diagram model is identifying what variables are known and what variables need to be computed.

The fact that this step must be completed first is unfortunate since we would like to keep whatever model we create for the mechanism shown in Figure 11.1 for reuse in other contexts (*i.e.,* where the set of variables involved is the same, but what is known or unknown is different). Having to make *a priori* decisions about what will be known and what will be unknown before we build the model limits the reusability of the model because applications will invariably come along which violate these *a priori* assumptions.

For the time being, let us assume we wish to apply a torque to the shaft on the left and as a result determine the position, velocity and acceleration of the shaft on the right. Figure 11.2 shows a schematic for the mechanism along with an actuator to define the system level causality.

11.2.2 Problem formulation

At this point, we have established what the behavioral equations are, what is known and what is to be computed. However, we still have a fair amount of work ahead of us. While we know the torque being applied on the left side of the shaft, it is not sufficient to just apply Equation (11.3) to compute the acceleration of the driven shaft. In order to compute the motion of the shaft, we must take into account the effective inertia of the overall mechanism.

This highlights the second disadvantage of block diagram formulations. Namely, some mathematical manipulation is necessary to formulate the problem (*i.e.,* deriving an equation to calculate ϕ_2 in terms of τ_k). The first step is to understand the effects of grounding the carrier gear. If we assume that ϕ_c is zero, the kinematic relationship for the planetary gear, Equation (11.4), can be reduced to:

$$\phi_s + R\phi_r = 0 \tag{11.7}$$

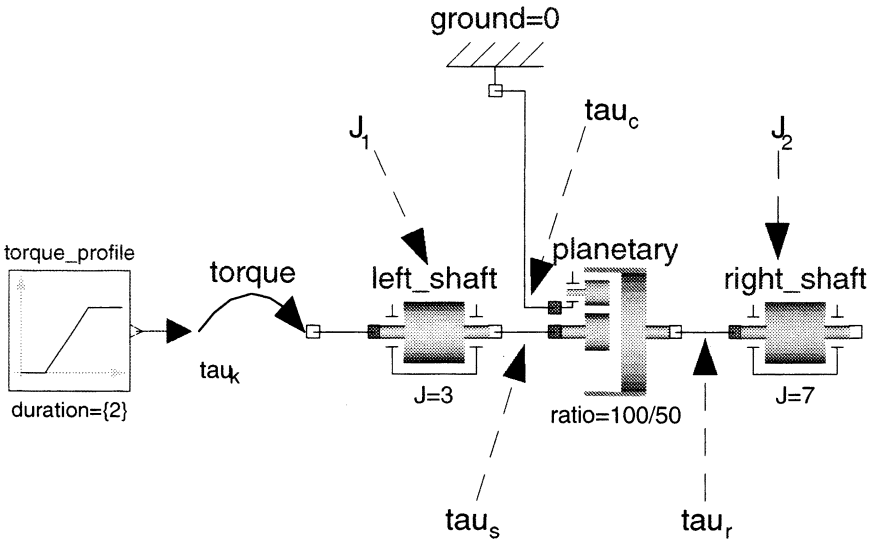

Figure 11.2. Planetary gear driven by the sun gear.

Now we must consider the equations which describe the behavior of the two shafts:

$$J_1\dot{\omega}_1 = \tau_k - \tau_s \tag{11.8}$$
$$J_2\dot{\omega}_2 = -\tau_r \tag{11.9}$$

Let us eliminate τ_r by using Equation (11.5) which leaves us with:

$$J_1\dot{\omega}_1 = \tau_k - \tau_s \tag{11.10}$$
$$J_2\dot{\omega}_2 = -R\tau_s \tag{11.11}$$

Therefore as it stands, our current system of equations is:

$$J_1\dot{\omega}_1 = \tau_k - \tau_s \tag{11.12}$$
$$J_2\dot{\omega}_2 = -R\tau_s \tag{11.13}$$
$$\phi_1 + R\phi_2 = 0 \tag{11.14}$$

Our next problem is that we have equations involving $\dot{\omega}_1$, $\dot{\omega}_2$, ϕ_1 and ϕ_2. Recall that what we really want to compute is ϕ_2. If we multiply Equation (11.12) by R and then add it to Equation (11.13) we are left with the following equations:

$$J_2\dot{\omega}_2 + RJ_1\dot{\omega}_1 = R\tau_k \tag{11.15}$$
$$\phi_1 + R\phi_2 = 0 \tag{11.16}$$

This allows us to get rid of all references to τ_s.

Now we would like to take the two remaining equations and use them to develop an equation for ϕ_2. In order to solve these equations we must write an explicit equation for $\dot{\omega}_2$ which we can then integrate twice to solve for ϕ_2. We are almost finished except that Equation (11.15) still contains a reference to $\dot{\omega}_1$. By differentiating Equation (11.16) twice we can write $\dot{\omega}_1$ in terms of $\dot{\omega}_2$ which gives us:

$$\dot{\omega}_2 = -\frac{R\tau_k}{R^2 J_1 + J_2} \tag{11.17}$$

From Equation (11.17) we see that the effective inertia, J_e, for the system (*i.e.*, the ratio of torque over acceleration) is:

$$J_e = \frac{R^2 J_1 + J_2}{R} \tag{11.18}$$

Now that we can solve for $\dot{\omega}_2$, ω_2 and ϕ_2, we can go back and use the following equations to solve for the reaction torque at ground, τ_c, the angular velocity of the driven shaft, ω_1, and the angular position of the driven shaft, ϕ_1, using the following equations:

$$\tau_c = -(1+R)\tau_s \tag{11.19}$$

$$\omega_1 = -R\omega_2 \tag{11.20}$$

$$\phi_1(t) = \phi_1(t_0) + \int_0^t \omega_1 \, dt \tag{11.21}$$

11.2.3 Block diagrams

After formulating the problem, we can construct our block diagram. The diagram can be seen in Figure 11.3. Another effect commonly observed with block diagrams is the scattering of parameter values among many components. There are three fundamental parameters to the mechanism: `R`, `J1` and `J2`. Note, in Figure 11.3, how these parameters are used in multiple places. No single object in Figure 11.3 distinctly represents, for example, the planetary gear. Instead, the effects of the planetary gears are seen in virtually every component. This scattering of parameters can cause robustness and maintenance problems because it becomes necessary to ensure that the same value is being used consistently everywhere the parameter appears.

11.2.4 Initial conditions

Figure 11.3 contains three integrator blocks. As a result, in order to integrate the system of equations we must know $\omega_2(t_0)$, $\phi_1(t_0)$ and $\phi_2(t_0)$. How do we determine what the initial conditions for these states should be?

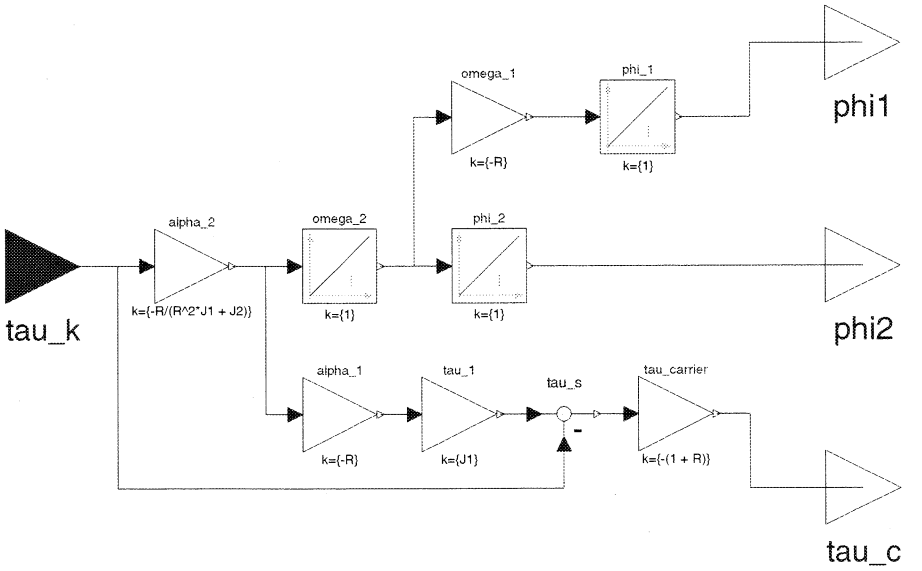

Figure 11.3. Block diagram of planetary gear system.

The simplest case is when $\omega_2(t_0)$ and $\phi_2(t_0)$ are known. In that case, by differentiating Equation (11.14), we find that:

$$\omega_1(t_0) = -\omega_2(t_0)\, R \tag{11.22}$$

Let us assume a less convenient (but just as reasonable) case where $\omega_1(t_0)$ and $\phi_1(t_0)$ are known instead. In such a case, we must formulate our initial conditions in much the same way that we formulated our behavioral equations. In other words, we must use Equation (11.14) to derive an expression for $\phi_2(t_0)$:

$$\phi_2(t_0) = -\frac{\phi_1(t_0)}{R} \tag{11.23}$$

and we must then differentiate to get an expression for $\omega_2(t_0)$:

$$\omega_2(t_0) = -\frac{\omega_1(t_0)}{R} \tag{11.24}$$

The point of these last two cases is that Equation (11.14) was required in order to find consistent initial conditions.[1] However, Equation (11.14) is not

[1] A consistent set of initial conditions is one which satisfies not just the final set of ordinary differential equations, but also any algebraic constraints (*e.g.*, Equation (11.14)) present in the original problem statement.

present in Figure 11.3. Furthermore, Equation (11.14) cannot even be derived from Figure 11.3.[2] In other words, in building our block diagram we have "lost" the information contained in Equation (11.14). When a model like the one shown in Figure 11.3 is passed along to other users it will not be clear to them that Equation (11.14) must be satisfied.

To some extent, the difficulties with initial conditions in this example are contrived. It is possible to create a block diagram formulation of this problem where only two initial conditions must be given and these two initial conditions always form a consistent set. However, coming up with such a formulation requires more work to be put into the formulation process. In addition, as problems become more complex, **difficulties related to inconsistent initial conditions cannot be avoided.**

11.2.5 Reuse

Having gone through the exercise of creating the model shown in Figure 11.3, consider for a moment the impact of changing one of our fundamental assumptions. For example, what if there were a stiff torsional spring between the carrier and ground rather than a rigid connection? Or, imagine the ring gear were connected to ground and the carrier were free. Would we be able to reuse much, if any, of the model we had created in light of such minor changes?

11.2.6 Conclusion

While block diagrams are useful, they have several drawbacks for physical systems. First, the equations used in a block diagram must be manually derived from the constitutive and conservation equations. This is not only a tedious and potentially error prone process, but it can be very difficult when the system behavior is described by DAEs (differential-algebraic equations).[3] In addition, causality assumptions must be made at the component level rather than the system level which limits the reusability of the component models. Furthermore, robustness and maintenance issues arise because of parameter scattering throughout the diagram (*i.e.*, problems can easily occur if consistent values are not used throughout the diagram). Finally, even once the models are formulated, the task of determining consistent initial conditions is quite challenging because the diagram itself may not contain sufficient information to compute them.

[2] While we can surmise from Figure 11.3 that $\omega_2 = R\,\omega_1$, you might be tempted to infer that $\phi_2 = R\,\phi_1$. However, this is not the case because an integration constant is required (*i.e.*, $\phi_2 = R\,\phi_1 + C$).

[3] A mathematical definition for differential-algebraic equations can be found in the glossary.

11.3 ACAUSAL APPROACH

Building acausal models simply involves dropping the needed components onto a schematic and connecting them. Starting from a general model, like the one shown in Figure 11.1, we can further develop the model by adding additional components (*e.g.*, dampers, sensors or actuators) until we have a complete system (*e.g.*, Figure 11.2). The important thing to note about the acausal approach is that we do not need to make any *a priori* assumptions (*i.e.*, about what is known and what is unknown) when building our model. In addition, all of the original equations are maintained (*i.e.*, no information is "lost" in the problem formulation) so that computing consistent sets of initial conditions is still possible .

Another advantage of the acausal approach is that if we decide to make a slight change in the physical configuration of our model, we do not have to derive a new problem formulation (*i.e.*, we do not have to determine the set of steps, like the ones in Section 11.2.2, that get us from what we know to what we wish to know). This makes the prospect of making configuration changes to our models less intimidating. For example, imagine we wish to introduce some torsional stiffness and damping between the carrier of the planetary gear and the mount point. Such a change would be quite typical of the changes that are made as a model evolves from an ideal system to a more detailed one. This change results in the system shown in Figure 11.4. Note that the only change required is to simply remove the rigid connection and replace it with a rotational spring and damper.

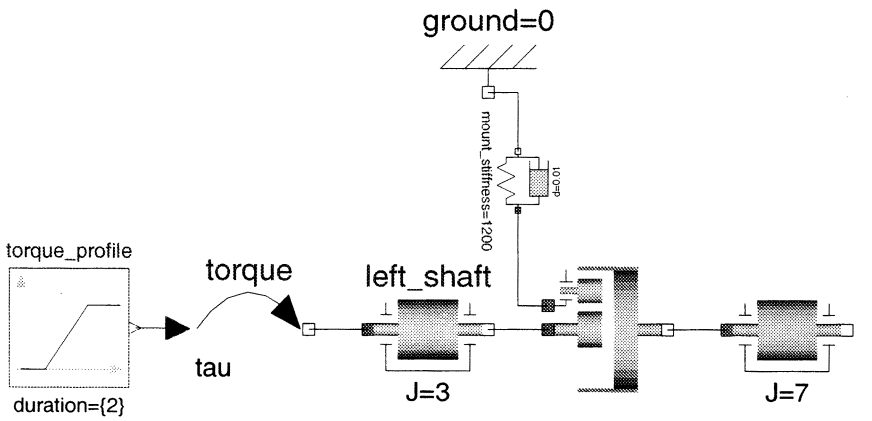

Figure 11.4. Planetary gear with torsional mount.

Finally, the fact that the models shown in Figures 11.2 and 11.4 resemble engineering schematics is an advantage in many circumstances. In other words,

the components (their icons) and the acausal nature of their connections would be intuitive to many engineers. Of course, this argument cuts both ways because the acausal formulation hides the mathematical structure of the problem which is useful when developing a control system.[4] While it is possible to automatically transform the acausal representation into a block diagram (as we did previously), it is generally impossible to carry out this process in reverse.

11.4 SUMMARY

As mentioned previously, block diagrams are useful in many situations and that is why they are supported by the Modelica language and why the MSL provides a significant collection of block diagram models. However, it is important to keep in mind some of the drawbacks of the block diagram approach when developing physical models (*i.e.,* plant models). Clearly, both block diagram and acausal formulations can be used to solve this problem. The question is not whether such block diagrams can be formulated, but whether they are the fastest and most efficient way to solve the problem. The key is to **avoid doing things manually** (*i.e.,* in a tedious and error prone way) when they can be done automatically.

While block diagrams contain useful information, there is no reason this information cannot be extracted automatically from an acausal formulation. For example, the following canonical form is typically used when analyzing system dynamics:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{11.25}$$

The A, B, C and D matrices are very useful in computing properties of the system (*e.g.,* poles and natural frequencies). Dymola is capable of automatically generating such matrices by linearizing a Modelica model around a particular solution. As a result, acausal models can generate much of the same useful information as block diagram models without the previously mentioned drawbacks.

Modelica provides acausal features for making your physical models as flexible and reusable as possible. In order to achieve maximum flexibility and reuse, it must be possible to identify when to use the acausal features. Block diagrams are preferred for conveying strictly one-way information (*e.g.,* the speed requested of a controller or the current gear of a transmission). Using acausal models in such contexts would be awkward and confusing. In cases where simultaneous equations or conservation principles are used, the acausal approach makes it easier to create and reuse models.

[4]Of course, it is still possible to extract that information automatically from an acausal formulation.

Looking closely at these two approaches, the fundamental difference is that block diagrams are inherently "low-level" formulations containing mathematical information. They represent the processing of the problem statement into a specific set of mathematical operations used to solve that particular problem. On the other hand, the acausal formulation preserves the physical details of the problem without specifying the process by which the problem should be solved.

The reason block diagrams are more prevalent is that tools which use block diagrams are easier to develop. This is because the difficult task of translating the acausal formulation into the specific mathematical operations is done by the user. From a user perspective, the ideal situation would be for simulation tools to perform this translation automatically. The emergence of tools, such as Dymola, powerful enough to automatically perform these manipulations will clearly be a benefit to the physical system model developer.

Hopefully, we have demonstrated that there are significant advantages to the acausal approach when describing the behavior of physical systems. Note that we have taken a relatively simple example for use in this chapter. Keep in mind that most of the drawbacks of block diagrams only become more pronounced as systems become larger and more complex while the acausal approach scales well with larger and more complex systems.

11.5 PROBLEMS

PROBLEM 11.1 *Create an alternative to the block diagram shown in Figure 11.3 for which it is impossible to provide an* **inconsistent** *set of initial conditions.*

PROBLEM 11.2 *Build a model for the system shown in Figure 11.4 using both block diagram and acausal components. How much similarity is there between the block diagram for 11.1 and 11.4? Compare this to the similarity between the two acausal models.*

PROBLEM 11.3 *How do you compute the initial conditions for the integrator blocks used in the Problem 11.2?*

PROBLEM 11.4 *Build both a block diagram model and an acausal model for the system shown in Figure 11.2. Then, reconfigure the acausal model of the system such that the ring gear is connected to ground instead of the carrier gear and the torque is applied to the carrier gear instead of the sun gear. Once this is complete, create a block diagram of the new configuration (either by using the previous one or creating a new one from scratch). Again, compare the two block diagram models to each other and the two acausal models to each other. What are the significant differences? How much reuse was there between the old configuration and the new configuration?*

Chapter 12

BUILDING LIBRARIES

12.1 OBJECTIVE

The `package` concept was introduced into Modelica to help organize definitions of models, connectors, *etc.* The idea was to allow for collections of related models to be bundled together. Packages which contain components (*e.g.,* connectors and models) to model a particular domain are called libraries. These libraries are usually implemented as a single package which contains several nested (or internal) packages. We will discuss the conventional structure for these nested packages.

Apart from the structural aspects, a library should also balance reusability and robustness. Ideally, a package provides users with endless possibilities for building systems by connecting up the provided components. At the same time it should be difficult for users to use these components incorrectly. When something is done incorrectly, it should either be immediately obvious to the tool (*e.g.,* making an invalid connection) or easily diagnosable (*e.g.,* providing non-physical parameter values).

In this chapter, we will revisit the `Chemistry` package presented in Section 6.4 and discuss its structuring in greater detail. You may first wish to go back and review the material in Section 6.4 before reading further.

12.2 CLASSIFICATION

Before creating a package of reusable components, it is necessary to decide what the scope of the package will be. For example, the MSL is organized by engineering discipline (*e.g.,* mechanics, and controls). Within each of these disciplines, a structure exists which makes it easy for users to locate the parts they are looking for. For example, within the `Modelica.Mechanics` pack-

age there are packages named `Translational` and `Rotational` which neatly divide the components into mutually exclusive sets.

Now, consider what would happen if the `Modelica.Mechanics` package were instead divided into categories like `Automotive`, `Appliances` and `Aerospace`. The problem with these categories is that they do not represent mutually exclusive sets. For example, all of these categories would have uses for the `IdealGear` model. Some models (*e.g.*, `OneWayClutch`) might be unique to one category but on the whole it would be difficult to locate models using such a categorization. The source of the problem is attempting to categorize based on applications which utilize many overlapping models.

To avoid this, choose a "taxonomy" (*i.e.*, a systematic classification scheme) which results in minimal overlap between any two packages in different branches of the package hierarchy. Such overlap cannot be entirely avoided (*e.g.*, where would you place the electronic thermostat model found in Figure 10.10?), but it can be minimized.

12.3 STRUCTURE

Once the decision has been made concerning what definitions should be included within a package, it is necessary to organize those definitions. While the Modelica language does not place many restrictions on the structure of a `package`, there are conventions for how packages should be structured. Ideally, all packages should have a consistent structure because, if each package had a completely unique structure, it would be quite disconcerting for the package users. For this reason, we describe several of the conventional package elements and their order of appearance in a package.

Before we describe the conventional nested packages, we should point out that the MSL includes several `partial package` definitions that provide basic icons for different types of packages. Typically, a new `package` (*e.g.*, `Chemistry`) should extend from the `Library2` definition found in `Modelica.Icons`. Any nested packages should extend from the `Library` definition also found in `Modelica.Icons`. The `Library2` graphics leave room for customization (*i.e.*, additional graphical annotations) while the graphics for `Library` are more generic, with less room for customization.

12.3.1 Types

In the case of the `Chemistry` package, we included a nested package that contained all the types that were specific to the chemistry package, *i.e.*,

```
package Chemistry
  extends Modelica.Icons.Library2;
  package Types
    extends Modelica.Icons.Library;
    type MolarFlowRate=Real(unit="mol/sec",
```

```
            quantity="MolarFlowRate");
  end Types;
  ...
end Chemistry;
```

The `MolarFlowRate` type is defined within the `Chemistry` package because it is needed but it does not exist in the `SIunits` package in the MSL. The `Types` package might also contain **`record`** definitions used within the package. It is entirely possible that a **`package`** will be written that does not require additional **`type`** or **`record`** definitions beyond those already available in other packages such as the MSL. In such cases, a nested `Types` package is not required.

12.3.2 Interfaces

Every package usually includes some interface definitions which are used throughout the package. For the most part, such definitions are either connectors or **`partial`** models (*e.g.*, `OnePort`, see Example 4.1).

Such definitions should be included in a nested **`package`** called `Interfaces`. Because these definitions are so important and so widely used within the package, the `Interfaces` package usually appears at, or near, the top of a **`package`**. For the `Chemistry` package, the `Interfaces` package is defined as:

```
package Chemistry
  ...
  package Interfaces
    extends Modelica.Icons.Library;
    connector Mixture "A chemical mixture"
      parameter Integer nspecies;
      Modelica.SIunits.Concentration c[nspecies];
      flow Types.MolarFlowRate r[nspecies];
    end Mixture;
  end Interfaces;
  ...
end Chemistry;
```

Note the use of the `MolarFlowRate` type defined previously in the `Types` package.

Some descriptive text following the declarations in the `Interfaces` package will help users to better understand any connector definitions. An important thing to remember when developing a package is to follow the Modelica sign conventions regarding **`flow`** variables in connectors (*i.e.*, positive flow is into the component).

Although the `Chemistry` package does not contain any **`partial`** definitions, such definitions are quite common (see Appendix C some **`partial`**

definitions found in the MSL). See Section 12.5.1 for a discussion on the advantages and disadvantages of `partial` definitions.

12.3.3 Functions

The `Functions` package is similar to the `Types` package. If there are package specific `function` definitions that are used throughout the package or potentially useful outside the context of the package, it is helpful to collect them in the `Functions` package. However, it is quite possible that no such functions exist for a given package, and in such a case the `Functions` package is not necessary.

For the `Chemistry` package, the `CalcRate` and `CalcMultiplier` functions are defined in the `Functions` package. These functions represent some of the fundamental constitutive relationships for chemistry. Because they could potentially be used by multiple models (in a more fully developed `Chemistry` package), they are kept in the `Functions` package rather than nested inside the models that currently use them (*e.g.*, the `Reaction` model). The structure of the `Chemistry.Functions` package is:

```
package Chemistry
  ...
  package Functions
    extends Modelica.Icons.Library;
    function CalcRate ... end CalcRate;
    function CalcMultiplier ... end CalcMultiplier;
  end Functions;
end Chemistry;
```

12.3.4 Sensors

For packages which contain definitions related to physical systems, a nested `package` of sensor models is typically provided. At a minimum, these sensors should be capable of outputting a signal that corresponds to the measured value for either a through or across variable for that system. Typically, at least three different sensors are included. One sensor type is called an "absolute sensor" and it outputs the absolute value for an across variable at a point (*e.g.*, the temperature at a point). Another type of sensor is a "relative sensor" which measures the difference between across variables at two different points (*e.g.*, the temperature difference between two points). Finally, a "flow sensor" measures the through variable between two points (*e.g.*, the heat flow through one path). The sensors for the across variables are generally connected in parallel while the "flow sensor" is generally connected in series with the flow path. To see an example of such sensors, look at the models in `Modelica.Mechanics.Rotational.Sensors`.

12.3.5 Examples

To help users understand how models should be connected, it is helpful to include a collection of runnable examples. For any package without examples, users will invariably misunderstand some aspect of the package. The result is that if a `package` is distributed to several users, there will be questions, mistakes or misunderstandings that occur among many of the users. In some cases, these may be due to bad model design but often the issues are fundamental to the package and require a certain understanding on the part of the user that is not obvious from the current structure or documentation of the package. It is precisely these issues which should be addressed by a nested `Examples` package.

The `Chemistry` package is somewhat unusual in that it contains no specific reaction models. This means that it is impossible to create an example without first identifying all participating species and reactions. Therefore, it is difficult to include simple examples. However, one possibility would be to nest the `Oregonator` package (also described in Section 6.4) inside the `Chemistry.Examples` package to serve as a demonstration of how the `Chemistry` package can be used.

All of the packages in the MSL contain `Examples` packages and examining those packages will be useful in understanding how to create an `Examples` package that will clearly demonstrate the use of a `package`. In addition, the MSL contains a special `partial model` called `Modelica.Icons.Example` that example models can be derived from. This provides them an icon that makes the model easily recognizable as an example.

It is a good idea to apply the `encapsulated` qualifier (see Section 9.1.1.3) to any of the models (or nested packages) contained within the `Examples` package. This will allow new users of the package to copy the examples out of the package hierarchy to create stand-alone models. These models can then be easily modified and studied further.

12.3.6 Tests

As a developer, it is important to maintain a suite of test cases to validate the definitions in the package. Such a suite should ideally test every component model and subsystem defined within the package. While test cases are generally ones which should work, it is sometimes useful to intentionally include examples which should not work. In this way, a test suite may be designed to validate not only the package, but also the tool with which the package is used (*e.g.*, to make sure it is capable of properly diagnosing common mistakes).

While test cases are important, they should probably not be distributed as part of the `package`. Instead, they should be kept in a separate `package` because users may be confused by the test cases. This is especially true when tests

are included which are intended to fail. While users might glean some useful information from the test cases (*i.e.*, the test cases may show more sophisticated examples of usage than the `Examples` package) test cases are typically not acceptable substitutes for illustrative examples.

12.3.7 Package specific structure

So far, we have described nested packages that commonly appear in a `package`. However, most packages will also have several nested packages which are specific to the engineering domain or organization of that particular package.

For example, the `Chemistry` package contains a nested package called `Basic` which in turn contains the following definitions:

```
package Chemistry
  ...
  package Basic
    extends Modelica.Icons.Library2;
    partial model Reaction ... end Reaction;
    model Reservoir ... end Reservoir;
    model Stationary ... end Stationary;
    model Volume ... end Volume;
  end Basic;
end Chemistry;
```

These models provide the basic models for chemical systems. While these models are not sufficient for representing all chemical systems, at least one of these models is typically present in every chemical system.

The `Modelica.Electrical.Analog` package is a good example of a package with considerable package specific content. The current version is organized as follows:

```
package Modelica
  ...
  package Electrical
    ...
    package Analog
      package Interfaces ... end Interfaces;
      package Basic ... end Basic;
      package Ideal ... end Ideal;
      package Lines ... end Lines;
      package Semiconductors ... end Semiconductors;
      package Sensors ... end Sensors;
      package Sources ... end Sources;
    end Analog;
  end Electrical;
end Modelica;
```

Note that in addition to the typical `Interfaces` and `Sensors` packages, there are several other domain specific packages (*e.g.*, `Semiconductors`).

12.3.8 Canonical form of a package

The discussions in the previous sections on conventions for nested packages can be summarized by the following canonical form for a new **package**:

```
package NewPackageName
  extends Modelica.Icons.Library2;
  package Types
    extends Modelica.Icons.Library;
    ...
  end Types;
  package Interfaces
    extends Modelica.Icons.Library;
    ...
  end Interfaces;
  package Functions
    extends Modelica.Icons.Library;
    ...
  end Functions;
  ...
  // Package specific structure
  ...
  package Examples
    extends Modelica.Icons.Library;
    ...
  end Examples;
end NewPackageName;
```

Such conventions are important primarily to the user of the package. The usability of a given **package** is greatly influenced by how organized and documented it is. The ability to quickly view the `Interfaces` package, to browse the connectors and see what kinds of information they contain or to view the `Examples` package to see how the components in the package can be combined, helps users get a feel for how the **package** should be used.

12.4 DOCUMENTATION

Documentation was discussed previously in Sections 2.4 and 9.2.2. The importance of documentation, particularly when developing a **package**, cannot be overstated. As Figure 9.4 shows, the documentation added by the developer can be nicely formatted by the tool before it is seen by the end user.

It is advisable to provide some kind of documentation for each model. Ideally, a sufficient explanation of the model should be provided in HTML as a documentation annotation (see Section 9.2.2). Also, it is useful to provide

additional documentation annotations for each `package` to give an overview of that `package` and its contents.

Once again, the MSL is an excellent example to work from. The documentation for the MSL is automatically generated from the Modelica source code and the embedded documentation annotations. The documentation is generated in such a way that hyperlinks are included to allow jumping between related definitions. For example, if you look at the definition for a component model you will find hyperlinks to the `connector` definitions. In addition, the graphical annotations can be used to generate graphical images that are also included in the generated documentation.

12.5 MAXIMIZING REUSABILITY

Now, let us turn our attention to making packages as reusable as possible. The whole idea behind making a package in the first place is to develop component models which can be used across different systems, projects and users. Included in this section are several ideas on how to maximize reusability that should be kept in mind when developing a package.

The general rule when developing a package is to try and see beyond the applications you are familiar with and consider applications that other projects or users might have.

12.5.1 Including partial definitions

Something to keep in mind when developing a package is that users may find the definitions in the package useful but incomplete for their purposes. In such cases, a user might wish to create a component which `extends` from yours. By anticipating such usage, you will make your package more reusable.

For example, the `Modelica.Mechanics.Rotational.Interfaces` package contains the following definition for a `Compliant` component[1]:

```
partial model Compliant
  Modelica.SIunits.Angle phi_rel;
  Modelica.SIunits.Torque tau;
  Interfaces.Flange_a flange_a;
  Interfaces.Flange_b flange_b;
equation
  phi_rel = flange_b.phi - flange_a.phi;
  flange_b.tau = tau;
  flange_a.tau = -tau;
end Compliant;
```

[1]The definitions for `Flange_a` and `Flange_b` are accessible since `Compliant` is also in the `Modelica.Mechanics.Rotational.Interfaces` package.

The `Compliant` definition is comparable to the `OnePort` definition shown in Example 4.1. This `partial` definition allows us to easily write models for springs and dampers in much the same way that the `OnePort` definition allows us to easily create models for resistors and capacitors (*i.e.*, by simply adding the necessary constitutive equation).

While the use of the `extends` keyword can help in minimizing redundancy across components, it can also lead to confusing models. Remember that any use of `extends` results in the complete component definition being distributed across the package hierarchy. This can make it difficult to form a complete picture of the derived component. For example, consider the following definitions:

```
partial model TwoRotationalConnections
  import Modelica.Mechanics.Rotational.Interfaces;
  Interfaces.Flange_a flange_a;
  Interfaces.Flange_a flange_b;
end TwoRotationalConnections;

partial model RotationalComponent
  extends TwoRotationalConnections;
  Modelica.SIunits.Angle phi_rel(start=0);
  Modelica.SIunits.Torque tau;
equation
  phi_rel = flange_b.phi - flange_a.phi;
  flange_b.tau = tau;
  flange_a.tau = -tau;
end RotationalComponent;

partial model GenericSpring
  extends RotationalComponent;
  Real c;
equation
  tau = c*phi_rel;
end GenericSpring;

model NonLinearSpring
  extends GenericSpring;
  parameter Real a, b;
equation
  c = a*phi_rel+b;
end NonLinearSpring;
```

While `partial` definitions are good for promoting reuse, finely fragmented `partial` models like those above can be hard to understand.

The purpose of most `partial` definitions is to define an interface. The interface of a component is usually composed of `parameter` and `connector` declarations. In many cases, it is useful to introduce some variables and equations in the `partial` definition. For example, the voltage, `v`, in the `OnePort`

model was introduced to represent the voltage drop across the component. Note that an equation for `v` was also included. In most cases, it is recommended that an equation be included for any variable declared in a `partial` definition. Anytime a variable is left "dangling" (*i.e.*, without an equation) it should be well documented what that variable represents and that it requires an equation.

The point is that all use of `extends` creates some additional complexity in a package because it fragments the complete definition. However, an understanding of how this complexity is generated will help in choosing the frequency and context of `extends` usage such that complexity is kept manageable.

12.5.2 Making components replaceable

Recall from our discussion in Chapter 4 how the `replaceable` keyword can be used when building components (*e.g.*, Example 4.8). This same approach can be employed when building packages. The most common application of the `replaceable` keyword involves `type` and `model` instantiation. However, keep in mind that the `replaceable` keyword can be used with `record`, `block`, `function` and even `package` instantiation. When a component is made reusable, it is a good idea to add a type constraint clause to the declaration (as discussed in Section 4.8.5.2). The constraining type should be a `partial` definition from the `Interfaces` package.

12.5.3 Package granularity

While building a `package`, make sure to clearly define the level of detail you expect users of the `package` to work at. For example, a library of basic hydraulic components is useful for developers of hydraulic circuits but constructing complete hydraulic systems from these basic components might be quite time consuming. For this reason, the creator of such a package may be tempted to include several complex hydraulic subsystems like pressure regulator systems or hydrostatic transmissions. By doing so, the organization of the package may suffer because the level of detail changes across the package.

The issue, in a nutshell, is deciding whether the package should be a set of primitive models or a set of composite models. If both types of models are being developed, there are at least two possibilities. First, you can try to organize the `package` to clearly delineate between the primitive models and the composite models. The second approach is to create two separate packages, one for the primitive models and one that uses the primitive models to provide the composite models.

12.6 MAXIMIZING ROBUSTNESS

When developing a `package` it is important to anticipate, as much as possible, all the different uses a model developer may have for the definitions you

are providing. Such uses can be broken down into three broad categories. First, the model developer may use the definitions as they were intended to be used. For a well tested `package`, all the definitions should function properly (*i.e.*, no obvious bugs) if they are used as intended. Second, the model developer may use the definitions incorrectly (*e.g.*, using non-physical values for parameters). These uses should be detected and result in a reasonable diagnostic message (how this is done will be discussed shortly). Finally, there may be uses that were not anticipated (not all possibilities can be anticipated). As they are uncovered, they should either be made to work correctly or provide diagnostic information as to why they are not allowed.

12.6.1 Using assertions and limits

If you look in the `Modelica.SIunits` package you will find the following definition for `Resistance`:

```
type Resistance = Real(final quantity="Resistance",
     final unit="Ohm", min=0);
```

Note the use of the `min` attribute. This is an example of setting a limit on a physical type to prevent misuse. A resistance of less than zero is not physically meaningful and using the `min` attribute in this way prevents the situation from occurring and therefore avoids getting results that may be confusing.

The `min` attribute is just one way of preventing non-physical values. In more general cases, it is not sufficient to identify a simple limit. Instead, a conditional expression is used to determine whether a value is physical. For example, it may also be necessary to identify when the amount of power dissipated by a resistor exceeds some critical value. In such a case, the following assertion could be added to the resistor model:

```
assert(i*v<1e+6, "Maximum power exceeded");
```

Including such an assertion will generate a diagnostic message when the resistor is used in an improper way (*e.g.*, in a circuit where the component would fail). Additional examples using `assert()` can be found in Section 8.4.

12.6.2 Finalizing choices

The ability to apply modifications to components and their subcomponents is an important feature of Modelica. However, such flexibility should be curtailed if it is not appropriate. For example, consider the following sensor definition found in the `Modelica.Mechanics.Rotational` package:

```
partial model AbsoluteSensor
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a;
  Modelica.Blocks.Interfaces.OutPort outPort(final n=1);
end AbsoluteSensor;
```

Since this sensor is only designed to output a single velocity signal, the size of the output port, `outPort`, has been permanently fixed to one. This is done using the `final` qualifier on a modification. Using the `final` qualifier (as described in more detail in Section 4.6) disallows further, in this case nonsensical, modifications.

12.6.3 Reducing the potential for modeling errors

The final class of robustness issues are commonly called "modeling errors". These cases are difficult to prevent because they depend on how the models are used.

For example, many simple models (*e.g.*, resistors, springs) behave the same way regardless of their orientation (*i.e.*, it does not matter how you connect them because they are non-directional). However, a diode is an example of a component which is sensitive to orientation. If the diode model is not clearly marked (either by the connector names or the graphical annotations) it is easy for a user to incorrectly place it into a schematic. Such an error cannot be detected because it requires an understanding of what the modeler intended.

Other examples might include the misuse of idealized components. For example, connecting a step voltage to a capacitor could cause simulation problems because the derivative of the voltage, used in the constitutive equation of the capacitor, would be infinite when the step occurs. This is another example of something that is difficult to prevent.

Because such situations cannot be automatically diagnosed (by either the tool or the use of assertions), proper documentation is about the only way such situations can potentially be avoided.

12.7 STORAGE OF MODELICA SOURCE CODE

When we include all of the behavioral descriptions, graphical annotations and documentation, the Modelica code for the `Chemistry` package could become quite long if contained within a single file. In order to avoid very large files, the `Chemistry` package can be split into smaller files while maintaining the same hierarchical structure. There are several benefits to using multiple files. First, if a single file is used and a component within that `package` is required, then the entire file must be read and parsed by the analysis tool. On the other hand, if the components are kept in separate files, only the file containing the required component definitions must be read and parsed. For large packages, this can be very convenient because it will speed up the process of reading and parsing definitions. A second reason for using multiple files is to make the system more manageable. With separate files it is easier to rename, move or edit the files individually without having to restructure a larger file.

To create a single `package` which spans multiple files, certain conventions must be followed. The fundamental idea behind the multiple file approach is to use a directory structure on the computer file system (*e.g.*, the hard drive) to represent the structure of the `package`. The key is that any `package`, nested or otherwise, may be represented by a single file or as a directory. If a directory is used to represent the `package`, that directory must contain a file called "*package.mo*" that contains only the `package` definition. The other definitions contained within that `package` must be placed in individual files with the same name as the entity they contain, followed by the suffix "`.mo`". Furthermore, all "`.mo`" files must contain a `within` statement on the first line indicating where, in the package hierarchy, subsequent definitions should be placed.[2]

At first these rules seem a bit confusing, so let us look at how the `Chemistry` package might be represented using multiple files. One possible directory structure is shown in Figure 12.1. The directories are shown in bold. Note that each directory represents a `package` in our original structure. However, not all packages are represented as directories. For Figure 12.1, we have arbitrarily chosen to have the `Functions` and `Basic` nested packages represented as directories, but the `Interfaces`, `Types` and `Sensors` packages represented as individual files.

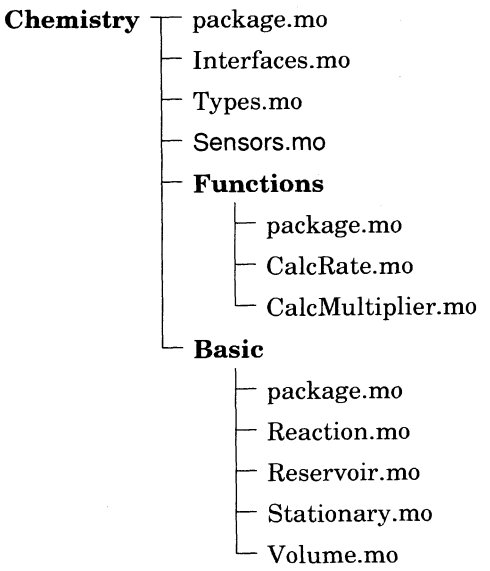

Figure 12.1. Possible file and directory structure for the `Chemistry` package.

[2]While this information is redundant, it can quickly identify misplaced files.

For packages represented as directories, note that the directory contains a "*package.mo*" file. As an example, the "*package.mo*" file in the `Functions` directory looks like:

```
within Chemistry;
package Functions
  extends Modelica.Icons.Library2;
end Functions;
```

Remember, no definitions appear in the "*package.mo*" file. Instead, all definitions in a package represented as a directory exist in individual files within that directory (see Figure 12.1).

In cases where a package is represented by a file, the file includes the package hierarchy starting from that package. As a result, the "*Types.mo*" file looks like:

```
within Chemistry;
package Types
  extends Modelica.Icons.Library2;
  type MolarFlowRate=Real(quantity="MolarFlowRate",
                          unit="mol/sec");
end Types;
```

It is not necessary that all top-level packages exist in the same directory. For example, the MSL is generally kept at one place on the hard drive, while other packages are kept somewhere else. In addition to searching the current working directory and the directory where the MSL is kept, Modelica tools also search the directories given in the `MODELICAPATH` environment variable.[3] This environment variable lists all of the directories that will be searched for Modelica definitions. Each directory is typically separated by a semicolon.

12.8 CONCLUSION

This chapter should provide you with some useful information on how to get started building reusable Modelica libraries. If you wish to see further examples you should study the MSL. In addition, there are a number of other free Modelica packages available. A list of such packages can be found at the Modelica web site (`http://www.modelica.org`).

[3]How you set this environment variable is specific to the operating system you are using.

Chapter 13

INITIAL CONDITIONS

13.1 OBJECTIVE

Before we can run a transient analysis, we must find the appropriate set of initial conditions for the variables. The most important requirement of initial conditions is that they do not contradict any of the equations in the models. Beyond that, they should make physical sense and represent an appropriate (usually quiescent) state for the system. This chapter will describe techniques for finding the initial conditions that are appropriate for a given simulation.

13.2 MATHEMATICAL FORMULATION

In order to better understand the process used to formulate initial conditions, we will examine several simple problems and examine how their variables are initialized. Simple problems are used to help illustrate the difficulties of finding initial conditions. In fact, these examples are nearly trivial and are not representative of even the most basic systems that result from connecting just a few components. Hopefully, these examples will provide some insights about how tools perform these same operations on much larger and more complex problems. [1]

Imagine we have constructed a model which results in the following system of equations:

$$x = 3y \tag{13.1}$$

$$\dot{x} = -2x \tag{13.2}$$

[1] To get an appreciation for how difficult it is to find initial conditions for complex problems, see Pantelides, 1988 or Mattsson and Söderlind, 1993.

Once a simulation has begun, solving such a system is straightforward since for a given value of x, which at any given time during transient simulation will always be known, we can determine y and $\dot{x}$. However, at the start of a simulation these three quantities are unknown. As a result, we are left with the following system of equations for our initial conditions:

$$x(t_0) = 3y(t_0) \tag{13.3}$$
$$\dot{x}(t_0) = -2x(t_0) \tag{13.4}$$

The problem of finding initial conditions for such a system essentially boils down to the problem of adding additional equations until we have as many equations as we have unknown quantities. An important caveat regarding this statement is that the resulting system of equations must lead to a non-singular system of equations.[2] For example, adding an additional equation so that we have the following three equations:

$$x(t_0) = 3y(t_0) \tag{13.5}$$
$$\dot{x}(t_0) = -2x(t_0) \tag{13.6}$$
$$x(t_0) = 2 \tag{13.7}$$

results in a non-singular system of equations that, when solved, yields the following initial conditions:

$$x(t_0) = 2 \tag{13.8}$$
$$y(t_0) = \frac{2}{3} \tag{13.9}$$
$$\dot{x}(t_0) = -4 \tag{13.10}$$

However, if instead, we add a "linearly dependent" equation (*i.e.*, one that is linearly dependent on another equation in the system) such as:

$$\dot{x}(t_0) = -6y \tag{13.11}$$

then the resulting system of equations:

$$x(t_0) = 3y(t_0) \tag{13.12}$$
$$\dot{x}(t_0) = -2x(t_0) \tag{13.13}$$
$$\dot{x}(t_0) = -6y(t_0) \tag{13.14}$$

is singular and a unique solution cannot be found. As another example, consider the following system:

$$x = 3y \tag{13.15}$$

[2] A non-singular system of equations is one for which a unique solution can be found.

$$\dot{x} = -2x \tag{13.16}$$
$$\dot{z} = -z \tag{13.17}$$

For initialization purposes, these equations are transformed into:

$$x(t_0) = 3y(t_0) \tag{13.18}$$
$$\dot{x}(t_0) = -2x(t_0) \tag{13.19}$$
$$\dot{z}(t_0) = -z(t_0) \tag{13.20}$$

Now we have three equations and five unknowns, namely:

$$\{x(t_0), y(t_0), z(t_0), \dot{x}(t_0), \dot{z}(t_0)\} \tag{13.21}$$

As a result, we must provide two additional equations. However, not all combinations will work. Let us look more carefully at the mathematical structure to understand what the restrictions are. First, because there is an algebraic constraint between x and y, we cannot provide independent initial values for both x and y. In other words, the following is a singular system of equations:

$$x(t_0) = 3y(t_0) \tag{13.22}$$
$$\dot{x}(t_0) = -2x(t_0) \tag{13.23}$$
$$\dot{z}(t_0) = -z(t_0) \tag{13.24}$$
$$x(t_0) = 2 \tag{13.25}$$
$$y(t_0) = 12 \tag{13.26}$$

Second, because there is a differential equation for z, we must provide an equation which leads to an initial value for z. As a result, we could provide additional equations for $\{x(t_0), z(t_0)\}$ or $\{y(t_0), z(t_0)\}$ but as we have shown, it is not sufficient to provide additional equations for $\{x(t_0), y(t_0)\}$.

Now, let us discuss the topic of initial values for derivatives. Let us consider our original system of equations:

$$x = 3y \tag{13.27}$$
$$\dot{x} = -2x \tag{13.28}$$

Again, this leads to the following system of equations involving the initial values:

$$x(t_0) = 3y(t_0) \tag{13.29}$$
$$\dot{x}(t_0) = -2x(t_0) \tag{13.30}$$

In addition to providing initial values for variables (*e.g.*, $x(t_0)$, $y(t_0)$), it is also useful to provide initial values for derivatives as well. For example, it is

common to set all derivative values to zero when choosing initial conditions. The idea behind such an assumption is that the transient analysis should start from a state of rest. If all the derivatives in the system are zero, that means that no change in the variables will occur until some external influence disturbs it. For this system, that would lead to the following non-singular system:

$$x(t_0) = 3y(t_0) \tag{13.31}$$
$$\dot{x}(t_0) = -2x(t_0) \tag{13.32}$$
$$\dot{x}(t_0) = 0 \tag{13.33}$$

Unfortunately, it is not currently possible to specify initial values for derivatives in the Modelica source code (*i.e.*, inside the `model` definitions). However, some tools do support this activity through their graphical user interface.

These are simple examples and it is easy to analyze their structure to understand why some combinations of equations are valid and others are not. In general though, complex problems cannot be analyzed in this way. Instead, the tool will generally present some choice of variables for which initial values can be provided (*i.e.*, entered by the user). It is possible that particular choices will not be available for the reasons discussed here.

The purpose of this chapter is to help demonstrate that a consistent set of initial conditions must be found before a transient simulation can be performed. As a result, some sets of initial conditions may not be allowed or may lead to numerical problems (*e.g.*, singular systems of equations).

13.3 USING ATTRIBUTES

One way to control the initialization of the system variables is to use the `start` attribute in conjunction with the `fixed` attribute. All `Real` variables have these attributes. If the `fixed` attribute is set to `true` for a variable, it has the effect of adding an equation to the existing set of equations used to solve for the initial conditions. That additional equation will equate the variable with the value of the `start` attribute for that variable. To understand this better, let us look at an example. Consider the following Modelica model:

```
model FirstOrderSystem
  Real x, y;
equation
  x = 3*y;
  der(x) = -2*x;
end FirstOrderSystem;
```

This results in the same system of equations discussed in Section 13.2:

$$x = 3y \tag{13.34}$$
$$\dot{x} = -2x \tag{13.35}$$

These equations are then transformed into the following system for initialization:

$$x(t_0) = 3y(t_0) \tag{13.36}$$
$$\dot{x}(t_0) = -2x(t_0) \tag{13.37}$$

So far, this is identical to the process described in Section 13.2.

By default, the `start` attribute has a value of zero and the `fixed` attribute has a value of `false`. However, if we modify our model as follows:

```
model FirstOrderSystem
  Real x(start=2,fixed=true), y;
equation
  x = 3*y;
  der(x) = -2*x;
end FirstOrderSystem;
```

we get the same system of equations for transient analysis but when these equations are transformed into the equations used to solve for the initial conditions an additional equation, $x(t_0) = 2$, will be added, resulting in the following system:

$$x(t_0) = 3y(t_0) \tag{13.38}$$
$$\dot{x}(t_0) = -2x(t_0) \tag{13.39}$$
$$x(t_0) = 2 \tag{13.40}$$

In this way, the `start` and `fixed` attributes can be used to provide additional equations for the initialization of the system. However, this approach only allows additional equations involving variables to be included (*i.e.*, this approach could not be used to add the equation for a derivative, $\dot{x}(t_0) = 0$). Even if the `fixed` attribute is set to `false`, a tool may choose to introduce extra equations, as needed, in which case the `start` attribute may still be used in the equations.

While it is easy to see how such attributes can be used to introduce additional equations into a fully assembled system, it is less obvious how they should be used within individual components. It is important not to overuse the `fixed` attribute because this can lead to over-constrained systems of equations with no solution. A good "rule of thumb" to follow is to only set the `fixed` attribute to be `true` for variables that are internal to a component model and that have had the `der` operator applied to them.

13.4 START OF SIMULATION

Another way the initial values can be set is with the `initial()` function. For example, we could initialize our `FirstOrderSystem` described previously using the following method:

```
model FirstOrderSystem
  Real x, y;
equation
  x = 3*y;
  der(x) = -2*x;
algorithm
  when initial() then
    reinit(x, 2);
  end when;
end FirstOrderSystem;
```

For most new users of Modelica, this will probably seem like the most natural way to initialize a problem because it is more procedural. However, this is not a good initialization method. The problem is that the `initial()` function only becomes `true` the instant after the simulation starts. However, before the analysis can start, it still needs to have a consistent set of initial conditions. As a result, two sets of initial conditions will be used. The first set will be solved for at $t = t_0$ using the methods described in Sections 13.2 and 13.3, while the next set will be solved for at time $t = t_0 + \epsilon$ (*i.e.*, just after the simulation starts) based on the contents of any `when` clauses triggered by the `initial()` function.

In addition to being confusing, this method of initialization (*i.e.*, using `reinit`) is limited to the variables that have had the `der` operator applied to them (see Chapter 7). In other words, in this case while `x` can be initialized in this way the variable `y` cannot. The use of the `initial()` function is best reserved for the initialization of `discrete` variables since all `discrete` variables can be initialized in this way (not just some) and their values are always assigned within `when` clauses. Several examples of such usage can be found in Chapter 7.

13.5 INITIALIZATION BASED ON ANALYSIS TYPE

The final way of controlling the initialization of a system of equations is to use the `analysisType()` function.[3] As described in Section 5.7.7.1, this function allows different equations to be used depending on the type of analysis being performed. A special analysis type, represented by the literal Modelica string `"static"`, is returned when the initial conditions are being determined. Revisiting our `FirstOrderSystem` model, we might choose to rewrite the model as follows:

```
model FirstOrderSystem
  Real x, y;
equation
```

[3]Currently, no simulation tools implement the `analysisType()` function but it should become available in time.

```
  x = 3*y;
  if analysisType()=="static" then
    x = 2;
  else
    der(x) = -2*x;
  end if;
end FirstOrderSystem;
```

By posing our model this way, different sets of equations will be generated depending on whether we are interested in transient analysis (*i.e.*, solving differential equations) or finding initial conditions. For example, by using the `analysisType()` function the following set of equations will be generated for finding initial conditions:

$$x(t_0) = 3y(t_0) \tag{13.41}$$
$$x(t_0) = 2 \tag{13.42}$$

Solving this system leads to the following initial conditions:

$$x(t_0) = 2 \tag{13.43}$$
$$y(t_0) = \frac{2}{3} \tag{13.44}$$

However, for transient analysis the usual system, *i.e.*,

$$x = 3y \tag{13.45}$$
$$\dot{x} = -2x \tag{13.46}$$

will be generated, but the initial values, $x(t_0)$ and $y(t_0)$, found in the `"static"` analysis case will be used at the start of the transient analysis.

One advantage that the `analysisType()` approach has over the `start` and `fixed` attribute approach is that a wider range of equations are possible. For example, the following is another possible way to write the `FirstOrderSystem` model using an equation involving `x`:

```
model FirstOrderSystem
  Real x, y;
equation
  x = 3*y;
  if analysisType()=="static" then
    4*x-6*y = 4;
  else
    der(x) = -2*x;
  end if;
end FirstOrderSystem;
```

Ultimately, this leads to the same initial conditions (*i.e.*, $x = 2, y = 2/3$), but the added power of being able to pose simultaneous systems of equations can be useful in some circumstances.

13.6 CONCLUSION

The following is a quick summary of how to use these various initialization techniques. The `start` and `fixed` attributes should be used to initialize internal variables within a component that have had the `der` operator applied to them, but not variables appearing in connectors. The `initial()` function should be used in conjunction with a `when` clause to initialize the values of discrete variables. Finally, the `analysisType()` function is useful in the same way that the `start` and `fixed` attributes are useful, except that a wider variety of expressions can be used instead of fixed values.

Ultimately, all of these techniques will generally lead to systems of equations that are still under-constrained which means additional equations will be required. In such cases, there are at least two possibilities. First, the tool being used will probably have some sophisticated capabilities for setting up and modifying the calculation of the initial conditions. Such facilities can be used to add the final few equations required or to try different combinations of equations. Furthermore, modifications of the `start` and `fixed` attributes can be made to add additional equations. Since initial conditions are typically specified at the system level, modifications to the `start` and `fixed` attributes for individual variables should be made from the system model using recursive modifications. If the system is still under-constrained, be aware that tools are likely to pick variables, as needed, and introduce equations setting those variables equal to the value of their `start` attribute (even if `fixed=false`).

Chapter 14

EFFICIENCY

14.1 OBJECTIVE

Once models have been developed and validated, it is natural to try and speed up the simulation of these models. In this chapter, we will describe techniques which can be used to reduce the simulation time of Modelica models. The goal will be to improve simulation time without having to sacrifice the clarity of the model description.

14.2 USE EQUATIONS

Because people are comfortable with assignment semantics, beginners often write models that look like:

```
model ModelUsingAssignment
  parameter Real b, c, d;
  Real x, y, z;
algorithm
  x := b*b+c/2-(d*b-d*c)^.5;
  y := (b+c)*x^2/(x^3+1);
  z := a*y^2+b*y+c;
end ModelUsingAssignment;
```

In some cases, an `algorithm` is used because the model was rewritten from a C or FORTRAN subroutine and the model developer wanted to preserve the spirit of the original subroutine. In other cases, an `algorithm` is used because the model developer does not trust tools to perform symbolic (algebraic) manipulation on such relations.[1]

[1] The semantics of `algorithm` sections prohibit tools from performing symbolic manipulation.

Using an `algorithm` section when an `equation` section would be sufficient is bad for several reasons. First, the use of `algorithm` sections makes symbolic differentiation difficult for analysis tools. This is because a variable can be assigned to multiple times using the `:=` operator and it still only counts as a single assignment which complicates the task of deriving symbolic derivatives. This may prevent tools from computing analytical Jacobians used in the integration process, which forces the Jacobians to be computed numerically (a considerably more expensive task).

A second reason to avoid `algorithm` sections in favor of `equation` sections is to allow an analysis tool to perform symbolic manipulation on the system of equations. Such manipulations can lead to significant performance increases and every effort should be made to allow such manipulations.

14.3 AVOID UNNECESSARY EVENTS

As mentioned in Chapter 7, any time a conditional expression changes value during a simulation, an event will be generated if the expression is not contained within the `noEvent` operator. Most of the time, these events are necessary but in some cases you can avoid them. For example, the following expression:

```
der(x) = if y<0 then 0 else y^2;
```

will generate an event (*i.e.*, stop the integrator and restart) at the point where `y` crosses zero. However, because the expression is continuous there is no need to actually have an event at that point. Using the `noEvent` operator (see Section 7.5.4.3), we can avoid such an event. The expression would then be written as:

```
der(x) = noEvent(if y<0 then 0 else y^2);
```

14.4 TIME SCALES

One factor that often results in slow simulations is when systems contain dynamics with substantially different time scales and these dynamics are coupled. This effect is called *stiffness*. For example, stiffness is quite common in chemical systems because different chemical reactions, involving the same set of reactants, usually occur at dramatically different rates. This kind of stiffness is hard to avoid.

An example of stiffness that can often be avoided is shown in Figure 14.1. At the top of Figure 14.1, you can see a collection of inertias which are rigidly connected while at the bottom we see the same inertias but with stiff springs and dampers between them. The difference between these two systems is that the bottom one has very high frequency oscillating modes because of the springs while the top one does not.

Figure 14.2 compares the two systems shown in Figure 14.1. The top plot compares the angular velocity of shaft 3, `I3`, for both cases while the bottom plot compares the simulation times. Note the high frequency oscillations in the angular velocity for the system connected by springs. The solver must work harder in order to resolve these oscillations. We can see the evidence of this when we compare the CPU time taken to solve each of the problems. Note that the system of rigidly connected inertias was solved in less than half the time of the one connected by springs.

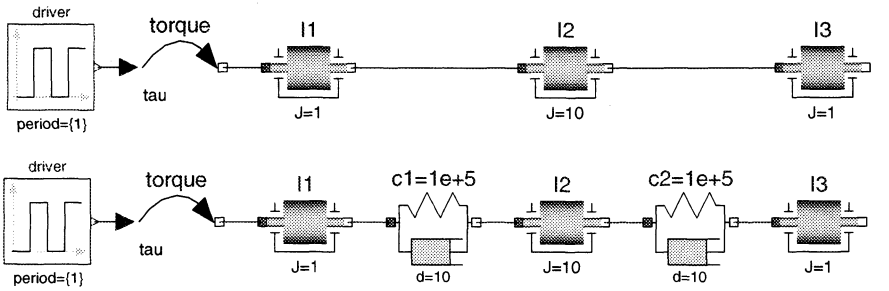

Figure 14.1. Comparison between a non-stiff (top) and stiff (bottom) system.

Another factor that increases the CPU time needed for simulation is the presence of fluctuating time scales in the model. When this happens, the simulation solver must compensate for the change in time scales, and this will result in worse performance. Examples 7.7 and 8.2 are typical of models which will exhibit this problem. This is because they have non-linearities which dramatically change the time scale of the dynamic response. From a physical perspective, this is because both examples involve a collision and the simulation solver must resolve the details of the collision. This means the time steps have to be refined (*i.e.*, made smaller) around the collision event. Ultimately, this leads to more integration steps and longer simulation time.

14.5 PROVIDING JACOBIANS FOR FUNCTIONS

In order to simulate a model described in Modelica, it is often necessary to differentiate certain expressions. For example, consider the following model:

```
model JacobianExample
  Real x, y;
equation
  der(y) = 2.0;
  y = f(x);
end JacobianExample;
```

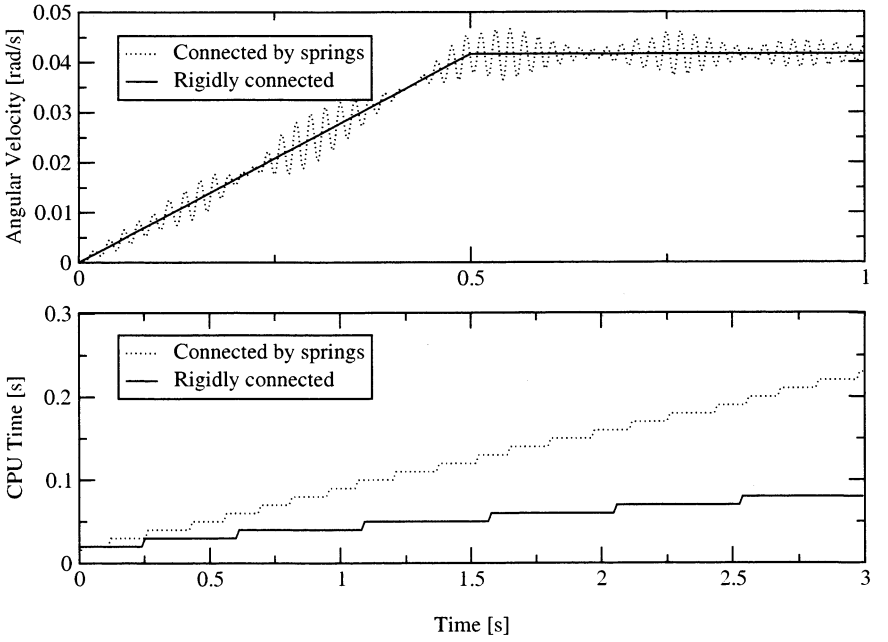

Figure 14.2. Comparison of simulation time and results for the systems in Figure 14.1.

Based on the first equation, it is obvious that the solution for $y(t)$ is

$$y(t) = y_0 + 2t \tag{14.1}$$

The question then remains, what is the solution for $x(t)$? The difficulty is that we are left with the implicit equation $y(t) = f(x(t))$. This is called an implicit equation because it does not allow us to calculate $x(t)$ explicitly. In other words, without knowing anything about the function f, we cannot write an equation of the form:

$$x(t) = ... \tag{14.2}$$

Instead, to solve for x, we must use an implicit method like Newton-Raphson iteration. Such methods need to be able to evaluate the partial derivatives of the function, f, with respect to its arguments. This matrix of partial derivatives is called a Jacobian.

Most of the time, the Jacobians are easy to compute because all the mathematical operations involved are expressed in Modelica. This allows a tool to symbolically differentiate the function with respect to its arguments. However, in some cases the tool may not be able to derive a symbolic Jacobian for an `algorithm` or the function may be a wrapper for an external subroutine written in C or FORTRAN (as described in Section 5.7.8).

When a Jacobian is required but a tool is not able to derive a symbolic Jacobian, one of two things happens. One possibility is that the tool will compute the Jacobian numerically. This is done by evaluating the function with different argument values and then approximating a derivative from the results. This method has the drawback that it is both slow and not very accurate. The other possibility is that the developer of the function also writes a companion function that is accurate and inexpensive to evaluate. To prevent a tool from computing the Jacobian numerically[2], an `annotation` can be used to indicate an analytical Jacobian is available. To demonstrate how this is done, consider the following function:

$$y(x) = 2x^5 + 4x^4 - x^3 + .5x^2 + x - 6\sqrt{x} \tag{14.3}$$

This function can be written in Modelica as follows:

```
function f
  input Real x;
  output Real y;
algorithm
  y := 2*x^5+4*x^4-x^3+.5*x^2+x-6*x^.5;
end f;
```

If we differentiate both sides of the equation, we get the following equation:

$$dy = dx\left(10x^4 + 16x^3 - 3x^2 + x + 1 - \frac{3}{\sqrt{x}}\right) \tag{14.4}$$

In other words, we can evaluate the incremental change in y, dy, that results from an incremental change in x, dx, for a particular value of the input argument x.

The rule for determining the order of the arguments to the Jacobian function is straightforward. First, all the `input` arguments to the original function are included. Then, a d-argument is included for all `Real input` arguments to the original function. Finally, additional d-arguments are included for each of the `Real output` arguments.

Using the rules above, we can construct the Jacobian function, `f_Jac`, for our original function, `f`, as follows:

```
function f_Jac
  input Real x;
  input Real dx;
  output Real dy;
algorithm
  dy := dx*(10*x^4+16*x^3-3*x^2+x+1-3*x^-.5);
end f_Jac;
```

[2]Assuming the tool was not able to compute an analytical Jacobian directly from the Modelica description.

Now that we have both of these functions, we need to have a way to indicate that the Jacobian of function `f` is computed by the function `f_Jac`. To do this, we must make a slight modification to our original function, `f`, as follows:

```
function f
  input Real x;
  output Real y;
  annotation(derivative=f_Jac); // f_Jac provides the Jacobian
algorithm
  y := 2*x^5+4*x^4-x^3+.5*x^2+x-6*x^.5;
end f;
```

It is possible to provide higher order derivatives for functions as well.[3] Details on how to do this can be found in the Modelica language specification.

14.6 CHOOSING THE PROPER INTEGRATION ROUTINE

Another important factor in simulation performance is the choice of which solver to used. Different solvers perform differently on different types of problems. For example, explicit solvers work well for systems with a narrow range of time scales. On the other hand, implicit solvers work well for problems that have mixed time scales like the ones described in Section 14.4.

A wide range of solvers should be tested for a given model to find out which one works best. With enough understanding of the underlying equations, it is possible to make a good educated guess about which solver will perform best.

14.7 TOLERANCES

Related to the issue of which solver to choose is the issue of what tolerances to use. Most integrators allow tolerances to be provided to guide them in making choices about how accurate the simulation results need to be. Tolerances can be given in different ways depending on the tool being used. For most tools, a single tolerance is used to characterize the allowable error in a simulation. The more accuracy needed by the user, the tighter (smaller) the tolerance. In some cases, it may be possible to specify or influence the tolerance used for particular variables.

Ultimately, it is up to the analyst to decide what tolerances are appropriate. For some applications, for example, it may be reasonable to sacrifice some accuracy by loosening tolerances to make sure the simulation will run quickly. For other types of analysis, where accuracy is important, it might be necessary to tighten tolerances to get a proper result.

[3] Such higher order derivatives may be required as a result of index reduction.

Normally, experimenting with tolerances will help to find an optimal tolerance value where tightening the tolerances does not significantly change the result. In other words, it is possible to find the loosest possible tolerances that yield essentially the same result as would be achieved with tighter tolerances without having to incur the performance penalty of tighter tolerances.

Keep in mind that, if the simulation results (*i.e.*, the time varying solution trajectories) are very sensitive to the choice of tolerances, then such results should be carefully scrutinized. It is always a good idea to verify that the choice of tolerance is reasonable by tightening the tolerances until no further significant change in the results is observed. If the results do change significantly with the tolerance value, the results are probably dubious at best. If successively tighter tolerances do not eventually lead to a repeatable solution, then you should bring the results to the attention of your tool vendor because this is a very undesirable situation. Together, you should be able to determine whether this is a tool issue or a modeling issue.

14.8 VARIABLE ELIMINATION

One common technique traditionally used for improving simulation time, particularly for models written in C or FORTRAN, is to eliminate as many variables as possible. For example, suppose we expand the contribution from the `OnePort` model (see Example 4.1) to get the following model:

```
model Resistor "An electrical resistor"
  import Modelica.Electrical;
  import Modelica.SIunits;
  extends Electrical.Analog.Interfaces.OnePort;
  parameter SIunits.Resistance R=300 "Resistance";
equation
  i*R = v;
end Resistor;
```

is equivalent to:

```
model Resistor
  import Modelica.SIunits;
  import Modelica.Electrical;
  SIunits.Voltage v "Voltage from pin p to n";
  SIunits.Current i "Current entering at pin p";
  Electrical.Analog.Interfaces.Pin p "Positive";
  Electrical.Analog.Interfaces.Pin n "Negative";
  parameter SIunits.Resistance R=300 "Resistance";
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
  i*R = v;
end Resistor;
```

The intermediate variables `i` and `v` were introduced by the `OnePort` model to make it easier to read the constitutive equations in models for components like resistors and capacitors. Model developers might be tempted to create their own resistor model as follows:

```
model Resistor "An electrical resistor"
  import SI=Modelica.SIunits;
  parameter SI.Resistance R=300 "Resistance";
  Modelica.Electrical.Analog.Interfaces.Pin p, n;
equation
  R*p.i = p.v-n.v;
  p.i + n.i = 0;
end Resistor;
```

The reasoning behind such a simplification is that it gets rid of a few variables (*i.e.,* `v` and `i`). Of course, by doing this they do not take advantage of the commonality with the `OnePort` model, which also means that their `Resistor` model is now slightly more difficult to read and understand than it was before.

As it turns out, such simplifications are unlikely to reduce simulation time. This is because the Modelica language has been built with the idea that analysis tools will parse Modelica models and make such simplifications as part of the analysis process. In other words, such simplifications are almost trivial for a tool to make when it has access to the complete Modelica behavioral model. Unlike simulation tools that rely on C or FORTRAN subroutines to describe model behavior, a Modelica tool is generally aware of all mathematical operations and what variables are involved, so it is able to make such simplifications.

For this reason, it is best to focus on some of the other issues raised in this chapter and only resort to variable elimination if there is strong evidence to suggest that it actually has an effect on simulation performance.

14.9 CONCLUSION

Improving the performance of simulations is an art and this chapter only provides a few hints on ways to improve performance. Despite the fact that this discussion has only covered a few of the basic ideas, these ideas should be useful in reducing simulation times. Most of the other optimizations that are possible must be done by the creators of the tools, because they are directly related to the solution methods, rather than the way the models are expressed.

Appendix A
History of Modelica

Since the definition of CSSL in 1967 (Strauss et al., 1967), most modeling languages have essentially been block oriented with inputs and outputs and the mathematical models have been defined as assignment statements for auxiliary variables and derivatives. Physical equations thus needed to be transformed to a form suitable for calculations. The only aid in transforming the equations to an algorithm for calculating derivatives was automatic sorting of the equations.

Among the recent research results in modeling and simulation, two significant concepts have been identified:

- Object oriented modeling languages demonstrate how object oriented concepts can be successfully employed to support hierarchical structuring, reuse and evolution of large and complex models independent from the application domain and specialized graphical formalisms.

- Acausal modeling demonstrates that the traditional simulation abstraction - the input/output block - can be generalized by relaxing the causality constraints (*i.e.*, by not committing ports to an 'input' or 'output' role early) and that this generalization enables both more simple models and more efficient simulation while retaining the capability to include submodels with fixed input/output roles.

The following is a list of several modeling languages that have explored these concepts in detail:

- Dymola

[1] Portions of this history are reprinted, with the permission of the Modelica Association, from the Modelica Specification and Modelica Rationale.

Dymola, as introduced already in 1978 (Elmqvist, 1978), is based on equations for acausal modeling, model types for reuse and submodel invocation for hierarchical modeling. The Dymola translator utilizes graph theoretical methods for causality assignment, for sorting and for finding minimal systems of simultaneous equations. Computer algebra is used for solving for the unknowns and to make simplifications of the equations. Constructs for hybrid modeling, including instantaneous equations, was introduced in 1993 (Elmqvist et al., 1993). Crossing functions for efficient handling of state events are automatically generated. A graphical editor is used to build icons and to make model compositions (Elmqvist et al., 2001). Major application areas include multi-body systems, drive-trains, power electronics and thermal systems.

- Omola[2]

  Omola (Andersson, 1994 and Mattsson et al., 1993) is an object-oriented and equation based modeling language. Models can be decomposed hierarchically with well-defined interfaces that describe interaction. All model components are represented as classes. Inheritance and specialization support easy modification. Omola supports behavioral descriptions in terms of differential-algebraic equations (DAE), ordinary differential equations (ODE) and difference equations. The primitives for describing discrete events allow implementation of high level descriptions as Petri nets and Grafcet. An interactive environment called OmSim supports modeling and simulation: graphical model editor, consistency analysis, symbolic analysis and manipulation to simplify the problem before numerical simulation, ODE and DAE solvers and interactive plotting. Applications of Omola and OmSim include chemical process systems, power generations and power networks.

- NMF (The Neutral Model Format)[3]

  NMF (Sahlin et al., 1996) is a language in the Dymola and Omola tradition and was first proposed as a standard to the building and energy systems simulation community in 1989. The language is formally controlled by a committee within ASHRAE (Am. Soc. for Heating, Refrigerating and Air-Conditioning Engineers). Several independently developed NMF tools and model libraries exist, and valuable lessons on language standardization and development of reusable model libraries have been learned. Salient features of NMF are: good support for model documentation, dynamical vector and parameter dimensions (*e.g.*, a model can calculate required spatial resolution

[2] http://www.control.lth.se/ cace/omsim.html
[3] http://urd.ce.kth.se/

for PDE) and full support for calls to foreign models (*e.g.*, legacy or binary Fortran or C models) including foreign model event signals.

- ObjectMath (Object Oriented Mathematical Modeling Language)[4]

  ObjectMath (Fritzson et al., 1995) is a high-level programming environment and modeling language designed as an extension to Mathematica. The language integrates object-oriented constructs such as classes, and single and multiple inheritance with computer algebra features from Mathematica. Both equations and assignment statements are included, as well as functions, control structures, and symbolic operations from standard Mathematica. Other features are parameterized classes, hierarchical composition and dynamic array dimension sizes for multi-dimensional arrays. The environment provides a class browser for the combined inheritance and composition graph and supports generation of efficient code in C++ or Fortran90. The user can influence the symbolic transformation of equations or expressions by manually specifying symbolic transformation rules, which also gives an opportunity to control the quality of generated code. The main application area so far has been in mechanical systems modeling and analysis.

- U.L.M. - Allan

  The goal of ALLAN (Pottier, 1983 and Jeandel et al., 1997) is to free engineers from computer science and numerical aspects, and to work towards capitalization and reuse of models. This means acausal and hierarchical modeling. A graphical representation of the model is associated to the textual representation and can be enhanced by a graphical editor. A graphical interface is used for hierarchical model assembly. The discrete actions at the interrupts in continuous behavior are managed by events. Automatons (synchronous or asynchronous) are available on events. FORTRAN or C code can be incorporated in the models. Two translators toward the NEPTUNIX and ADASSL (modified DASSLRT) solvers are available. Main application domains are energy systems, car electrical circuits, geology and naval design. The language U.L.M. has been designed in 1993 with the same features as the ALLAN language in a somewhat different implementation (Jeandel et al., 1996). It is a model exchange language linked to ALLAN. All aspects of modeling are covered by the textual language. There is an emphasis on the separation of the model structure and the model numerical data for reuse purposes. It also has an interesting feature on model validation capitalization.

[4] http://www.ida.liu.se/labs/pelab/omath/

- SIDOPS+[5]

  SIDOPS+ supports nonlinear multidimensional bond-graph and block-diagram models, which can contain continuous-time parts and discrete-time parts (Breunese and Broenink, 1997). The language has facilities for automated modeling support like polymorphic modeling (separation of the interface and the internal description), multiple representations (component graphs, physical concepts like bond graphs or ideal physical models and (acausal) equations or assignment statements), and support for reusability (*e.g.*, documentation fields, physical types). Currently, SIDOPS+ is mainly used in the field of mechatronics and (neural) control. It is the model description language of the package 20-SIM (Broenink, 1997).[6] SIDOPS+ is the third generation of SIDOPS which started as a model description language for single-dimensional bond-graph and block-diagram models.

- Smile[7]

  Smile is an object-oriented and equation-based modeling and simulation environment. The object-oriented and imperative features of Smile's model description language are very similar to Objective-C. Equations may either be specified symbolically or as procedures; external modules can be integrated. Smile also has a dedicated experiment description language. The system consists of translators for the above-mentioned languages, a simulation engine offering several numeric solvers, and components for interactive experimenting, visualization, and optimization. Smile's main application domain traditionally has been the simulation of solar energy equipment and power plants (Tummescheit and Pitz-Paal, 1997), but thanks to its object-oriented modeling features it is applicable to other classes of complex systems as well. An extension of Smile to support Modelica is planned (Ernst et al., 1997).

While these languages all demonstrated important new ideas, they also fragmented the the market for modeling languages. In 1996, Hilding Elmqvist initiated an effort to unify the concepts of these approaches into a single language. Having started as an action within ESPRIT project *Simulation in Europe Basic Research Working Group (SiE-WG)* and then operating as *Technical Committee 1* within Eurosim and *Technical Chapter on Modelica* within Society for Computer Simulation International, a working group made up of simulation tool builders, users from different application domains, and computer scientists has made an attempt to unify the concepts and introduce a common

[5] http://www.rt.el.utwente.nl/proj/modsim/modsim.htm
[6] http://www.rt.el.utwente.nl/20sim
[7] http://www.first.gmd.de/smile/smile0.html

modeling language. This language, called Modelica, is intended for modeling within many application domains (for example: electrical circuits, multi-body systems, drive trains, hydraulics, thermodynamical systems and chemical systems) and possibly using several formalisms (for example: ODE, DAE, bond graphs, finite state automata and Petri nets). Tools which might be general purpose or specialized to certain formalism and/or domain will store the models in the Modelica format in order to allow exchange of models between tools and between users. Much of the Modelica syntax will be hidden from the end-user because, in most cases, a graphical user interface will be used to build models by selecting icons for model components, using dialogue boxes for parameter entry and connecting components graphically.

The work started in the continuous time domain since there is a common mathematical framework in the form of differential-algebraic equations (DAE) and there are several existing modeling languages based on similar ideas. There is also significant experience of using these languages in various applications. It thus seems to be appropriate to collect all knowledge and experience and design a new unified modeling language or neutral format for model representation. The short range goal was to design a modeling language for differential-algebraic equation systems with some discrete event features to handle discontinuities and sampled systems. The design should be extendible in order that the goal can be expanded to design a multi-formalism, multi-domain, general-purpose modeling language.

The Modelica Association was formed in Feb. 5, 2000 and is now responsible for the design of the Modelica language. After 24 three-day meetings, Modelica 1.4 was released December 15, 2000.

A.1 CONTRIBUTORS TO THE MODELICA LANGUAGE

Bernhard Bachmann, Fachhochschule Bielefeld, Germany
Fabrice Boudaud, Gaz de France, France
Peter Bunus, MathCore, Linköping, Sweden
Jan Broenink, University of Twente, The Netherlands
Dag Brück, Dynasim, Lund, Sweden
Hilding Elmqvist, Dynasim, Lund, Sweden
Vadim Engelson, Linköping University, Sweden
Thilo Ernst, GMD-FIRST, Berlin, Germany
Jorge Ferreira, University of Aveiro, Portugal
Rüdiger Franke, ABB Corporate Research Center, Heidelberg, Germany
Peter Fritzson, Linköping University, Linköping, Sweden
Pavel Grozman, Equa, Stockholm, Sweden
Johan Gunnarsson, MathCore, Linköping, Sweden
Alexandre Jeandel, Gaz de France, France

Mats Jirstrand, MathCore, Linköping, Sweden
Kaj Juslin, VTT, Finland
David Kågedal, Linköping University, Sweden
Clemens Klein-Robbenhaar, Germany
Matthias Klose, Technical University of Berlin, Germany
Pontus Lidman, MathCore, Linköping, Sweden
Nathalie Loubere, Gaz de France, France
Sven Erik Mattsson, Dynasim, Lund, Sweden
Pieter Mosterman, German Aerospace Center, Oberpfaffenhofen, Germany
Henrik Nilsson, Linköping University, Sweden
Hans Olsson, Dynasim, Lund, Sweden
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
Tommy Persson, Linköping University, Sweden
Per Sahlin, Equa Simulation Technology Group, Stockholm, Sweden
Levon Saldamli, Linköping University, Sweden
André Schneider, Fraunhofer Institute, Dresden, Germany
Michael Tiller, Ford Motor Company, Detroit, United States of America
Hubertus Tummescheit, Lund Institute of Technology, Sweden
Hans Venghaluwe, University of Gent, Belgium Hans-Jürg Wiesmann, ABB Corporate Research Ltd., Baden, Switzerland

A.2 CONTRIBUTORS TO THE MODELICA STANDARD LIBRARY

Peter Beater, University of Paderborn, Germany
Christoph Clauß, Fraunhofer Institute, Dresden, Germany
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
André Schneider, Fraunhofer Institute, Dresden, Germany
Hubertus Tummescheit, Lund Institute of Technology, Sweden

Appendix B
Modelica Syntax

This chapter includes the grammar for version 1.4 of the Modelica language. This grammar depends on the following lexical definitions:

```
IDENT = NONDIGIT { DIGIT | NONDIGIT }
NONDIGIT = "_" | letters "a" to "z"
        | letters "A" to "Z"
STRING = """ { S-CHAR | S-ESCAPE } """
S-CHAR = any member of the source character set
      except double-quote """ & backslash "\"
S-ESCAPE = "\'" | "\"" | "\?" | "\\" | "\a"
        | "\b" | "\f" | "\n" | "\r" | "\t" | "\v"
DIGIT = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
UNSIGNED_INTEGER = DIGIT { DIGIT }
UNSIGNED_NUMBER = UNSIGNED_INTEGER
     [ "." [ UNSIGNED_INTEGER ] ]
     [ ( e | E ) [ "+" | "-" ] UNSIGNED_INTEGER ]
```

The grammar definition is as follows:

```
stored_definition
  : [ within [ name ] ";" ]
    { [ final ] class_definition ";" }

class_definition
  : [ encapsulated ] [ partial ]
    ( class | model | record | block | connector
      | type | package | function )
    IDENT class_specifier

class_specifier
  : string_comment composition end IDENT
  | "=" name [ array_subscripts ]
     [ class_modification ] comment
```

```
composition
  : element_list
    { public element_list | protected element_list |
      equation_clause | algorithm_clause }
    [ external [ language_specification ]
               [ external_function_call ] ";"
               [ annotation_decl ";" ] ]

language_specification
  : STRING

external_function_call
  : [ component_reference "=" ]
    IDENT "(" [ expression { "," expression } ] ")"

element_list
  : { element ";" | annotation_decl ";" }

element
  : import_clause
  | extends_clause
  | [ final ] [ inner | outer ]
    ( ( class_definition | component_clause )
      | replaceable
        ( class_definition | component_clause )
        [ constraining_clause ] )

import_clause
  : import ( IDENT "=" name | name ["." "*"] )
    comment

extends_clause
  : extends name [ class_modification ]

constraining_clause
  : extends_clause

component_clause
  : type_prefix type_specifier
    [ array_subscripts ] component_list

type_prefix
  : [ flow ] [ discrete | parameter | constant ]
    [ input | output ]

type_specifier
  : name

component_list
```

```
  : component_declaration
    { "," component_declaration }

component_declaration
  : declaration comment

declaration
  : IDENT [ array_subscripts ] [ modification ]

modification
  : class_modification [ "=" expression ]
  | "=" expression
  | ":=" expression

class_modification
  : "(" { argument_list } ")"

argument_list
  : argument { "," argument }

argument
  : element_modification
  | element_redeclaration

element_modification
  : [ final ] component_reference modification
    string_comment

element_redeclaration
  : redeclare
    ( ( class_definition | component_clause1 )
      | replaceable
        ( class_definition | component_clause1 )
        [ constraining_clause ] )

component_clause1
  : type_prefix type_specifier
    component_declaration

equation_clause
  : equation { equation ";" | annotation_decl ";" }

algorithm_clause
  : algorithm { algorithm ";" | annotation_decl ";" }

equation
  : ( simple_expression "=" expression
    | conditional_equation_e
    | for_clause_e
    | connect_clause
```

```
    | when_clause_e
    | assert_clause ) comment

algorithm
  : ( component_reference
      ( ":=" expression | function_call )
    | "(" expression_list ")" ":="
      component_reference function_call
    | conditional_equation_a
    | for_clause_a
    | while_clause
    | when_clause_a
    | assert_clause ) comment

conditional_equation_e
  : if expression then { equation ";" }
    { elseif expression then { equation ";" } }
    [ else { equation ";" } ]
    end if

conditional_equation_a
  : if expression then { algorithm ";" }
    { elseif expression then { algorithm ";" } }
    [ else { algorithm ";" } ]
    end if

for_clause_e
  : for IDENT in expression loop
    { equation ";" }
    end for

for_clause_a
  : for IDENT in expression loop
    { algorithm ";" }
    end for

while_clause
  : while expression loop
    { algorithm ";" }
    end while

when_clause_e
  : when expression then
    { equation ";" }
    end when

when_clause_a
  : when expression then
    { algorithm ";" }
    { elsewhen expression then
```

```
    { algorithm ";" } }
    end when

connect_clause
  : connect "(" connector_ref ","
    connector_ref ")"

connector_ref
  : IDENT [ array_subscripts ]
    [ "." IDENT [ array_subscripts ] ]

assert_clause
  : assert "(" expression "," STRING { "+" STRING } ")"
  | terminate "(" STRING { "+" STRING } ")"

expression
  : simple_expression
  | if expression then expression else expression

simple_expression
  : logical_expression
    [ ":" logical_expression
      [ ":" logical_expression ] ]

logical_expression
  : logical_term { or logical_term }

logical_term
  : logical_factor { and logical_factor }

logical_factor
  : [ not ] relation

relation
  : arithmetic_expression
    [ rel_op arithmetic_expression ]

rel_op
  : "<" | "<=" | ">" | ">=" | "==" | "<>"

arithmetic_expression
  : [ add_op ] term { add_op term }

add_op
  : "+" | "-"

term
  : factor { mul_op factor }

mul_op
```

```
  : "*" | "/"

factor
  : primary [ "^" primary ]

primary
  : UNSIGNED_NUMBER
  | STRING
  | false
  | true
  | component_reference [ function_call ]
  | "(" expression_list ")"
  | "[" expression_list { ";" expression_list } "]"
  | "{" expression_list "}"

name
  : IDENT [ "." name ]

component_reference
  : IDENT [ array_subscripts ]
    [ "." component_reference ]

function_call
  : "(" function_arguments ")"

function_arguments
  : expression_list
  | named_arguments

named_arguments
  : [named_argument { "," named_argument }]

named_argument
  : IDENT "=" expression

expression_list
  : expression { "," expression }

array_subscripts
  :  "[" subscript { "," subscript } "]"

subscript
  : ":"
  | expression

comment
  : string_comment [ annotation_decl ]

string_comment
  : [ STRING { "+" STRING } ]
```

```
annotation_decl
  : annotation class_modification
```

Appendix C
Modelica Standard Library: Connectors

C.1 ELECTRICAL (ANALOG)

```
within Modelica.Electrical.Analog.Interfaces;

connector PositivePin
  Modelica.SIunits.Voltage v "Potential at the pin";
  flow Modelica.SIunits.Current i
      "Current flowing into the pin";
end PositivePin;

connector NegativePin
  Modelica.SIunits.Voltage v "Potential at the pin";
  flow Modelica.SIunits.Current i
      "Current flowing into the pin";
end NegativePin;

partial model OnePort "Component with two electrical pins"
  SIunits.Voltage v "Voltage drop between the two pins";
  SIunits.Current i "Current flowing from pin p->n";
  Interfaces.PositivePin p;
  Interfaces.NegativePin n;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;

partial model TwoPort "Component with two electrical ports"
  SIunits.Voltage v1 "Voltage drop over the left port";
  SIunits.Voltage v2 "Voltage drop over the right port";
  SIunits.Current i1 "Current flowing from p1->n1";
  SIunits.Current i2 "Current flowing from p2->n2";
```

```
  Interfaces.PositivePin p1 "Positive pin of the left port";
  Interfaces.NegativePin n1 "Negative pin of the left port";
  Interfaces.PositivePin p2 "Positive pin of the right port";
  Interfaces.NegativePin n2 "Negative pin of the right port";
equation
  v1 = p1.v - n1.v;
  v2 = p2.v - n2.v;
  0 = p1.i + n1.i;
  0 = p2.i + n2.i;
  i1 = p1.i;
  i2 = p2.i;
end TwoPort;
```

C.2 BLOCK DIAGRAMS

```
within Modelica.Blocks.Interfaces;

connector InPort "Connector with Real inputs"
  parameter Integer n=1 "Dimension of signal vector";
  replaceable type SignalType = Real "type of signal";
  input SignalType signal[n] "Real input signals";
end InPort;

connector OutPort "Connector with Real outputs"
  parameter Integer n=1 "Dimension of signal vector";
  replaceable type SignalType = Real "type of signal";
  output SignalType signal[n] "Real output signals";
end OutPort;

connector BooleanInPort "Connector with Boolean inputs"
  parameter Integer n=1 "Dimension of signal vector";
  input Boolean signal[n] "Boolean input signals";
end BooleanInPort;

connector BooleanOutPort "Connector with Boolean outputs"
  parameter Integer n=1 "Dimension of signal vector";
  output Boolean signal[n] "Boolean output signals";
end BooleanOutPort;

partial block SO "Single Output continuous control block"
  OutPort outPort(final n=1) "Output signal connector";
protected
  Real y=outPort.signal[1];
end SO;

partial block MO "Multiple Output continuous control block"
  parameter Integer nout(min=1) = 1 "Number of outputs";
  OutPort outPort(final n=nout) "Output signals connector";
protected
  Real y[nout]=outPort.signal;
```

```
end MO;

partial block SISO "Single Input Single Output block"
  InPort inPort(final n=1) "Input signal connector";
  OutPort outPort(final n=1) "Output signal connector";
protected
  Real u=inPort.signal[1];
  Real y=outPort.signal[1];
end SISO;

partial block SI2SO "2 Single Input/1 Single Output block"
  InPort inPort1(final n=1) "Input signal 1 connector";
  InPort inPort2(final n=1) "Input signal 2 connector";
  OutPort outPort(final n=1) "Output signal connector";
protected
  Real u1=inPort1.signal[1] "Input signal 1";
  Real u2=inPort2.signal[1] "Input signal 2";
  Real y=outPort.signal[1] "Output signal";
end SI2SO;

partial block MISO "Multiple Input Single Output block"
  parameter Integer nin=1 "Number of inputs";
  InPort inPort(final n=nin) "Input signals connector";
  OutPort outPort(final n=1) "Output signal connector";
protected
  Real u[:]=inPort.signal "Input signals";
  Real y=outPort.signal[1] "Output signal";
end MISO;

partial block MIMO "Multiple Input Multiple Output block"
  parameter Integer nin=1 "Number of inputs";
  parameter Integer nout=1 "Number of outputs";
  InPort inPort(final n=nin) "Input signals connector";
  OutPort outPort(final n=nout) "Output signals connector";
protected
  Real u[:]=inPort.signal "Input signals";
  Real y[:]=outPort.signal "Output signals";
end MIMO;

partial block BooleanSISO "Boolean SISO block"
  BooleanInPort inPort(final n=1) "Input signal connector";
  BooleanOutPort outPort(final n=1) "Output signal connector";
protected
  Boolean u=inPort.signal[1];
  Boolean y=outPort.signal[1];
end BooleanSISO;

partial block BooleanSignalSource "Boolean source block"
  parameter Integer nout(min=1) = 1 "# of Boolean outputs";
  BooleanOutPort outPort(final n=nout) "Output connector";
```

```
end BooleanSignalSource;
```

C.3 TRANSLATIONAL MOTION

```
within Modelica.Mechanics.Translational.Interfaces;

connector Flange_a
  Modelica.SIunits.Position s "absolute flange position";
  flow Modelica.SIunits.Force f "cut force in flange";
end Flange_a;

connector Flange_b
  Modelica.SIunits.Position s "absolute flange position";
  flow Modelica.SIunits.Force f "cut force in flange";
end Flange_b;

partial model Rigid "Rigid translational component"
  SIunits.Position s "position of component center";
  parameter SIunits.Length L=0 "length of component";
  Translational.Interfaces.Flange_a flange_a;
  Translational.Interfaces.Flange_b flange_b;
equation
  flange_a.s = s - L/2;
  flange_b.s = s + L/2;
end Rigid;

partial model Compliant "Compliant translational component"
  Translational.Interfaces.Flange_a flange_a;
  Translational.Interfaces.Flange_b flange_b;
  SIunits.Distance s_rel "relative distance";
  flow SIunits.Force f "force between flanges";
equation
  s_rel = flange_b.s - flange_a.s;
  flange_b.f = f;
  flange_a.f = -f;
end Compliant;

partial model AbsoluteSensor "Absolute sensor"
  Translational.Interfaces.Flange_a flange_a;
  Modelica.Blocks.Interfaces.OutPort outPort(final n=1);
end AbsoluteSensor;

partial model RelativeSensor "Relative sensor"
  Translational.Interfaces.Flange_a flange_a;
  Translational.Interfaces.Flange_b flange_b;
  Modelica.Blocks.Interfaces.OutPort outPort(final n=1);
end RelativeSensor;
```

C.4 ROTATIONAL MOTION

```
within Modelica.Mechanics.Rotational.Interfaces;
```

```
connector Flange_a
  Modelica.SIunits.Angle phi "absolute rotation";
  flow Modelica.SIunits.Torque tau "cut torque in flange";
end Flange_a;

connector Flange_b
  Modelica.SIunits.Angle phi "absolute rotation";
  flow Modelica.SIunits.Torque tau "cut torque in flange";
end Flange_b;

partial model Rigid "Rigid rotational component"
  SIunits.Angle phi "Absolute rotation angle"
  Interfaces.Flange_a flange_a;
  Interfaces.Flange_b flange_b;
equation
  flange_a.phi = phi;
  flange_b.phi = phi;
end Rigid;

partial model Compliant "Compliant rotational component"
  SIunits.Angle phi_rel(start=0) "Relative rotation angle";
  SIunits.Torque tau "Torque between flanges";
  Interfaces.Flange_a flange_a;
  Interfaces.Flange_b flange_b;
equation
  phi_rel = flange_b.phi - flange_a.phi;
  flange_b.tau = tau;
  flange_a.tau = -tau;
end Compliant;

partial model AbsoluteSensor "Absolute sensor"
  Interfaces.Flange_a flange_a;
  Modelica.Blocks.Interfaces.OutPort outPort(final n=1);
end AbsoluteSensor;

model RelativeSensor "Relative sensor"
  Interfaces.Flange_a flange_a;
  Interfaces.Flange_b flange_b;
  Modelica.Blocks.Interfaces.OutPort outPort(final n=1);
end RelativeSensor;
```

Appendix D
Modelica Standard Library: Common Units

D.1 TIME AND SPACE

```
within Modelica;

package SIunits
  type Angle = Real(final quantity="Angle", final unit="rad",
       displayUnit="deg");
  type SolidAngle = Real(final quantity="SolidAngle",
       final unit="sr");
  type Length = Real(final quantity="Length", final unit="m");
  type Position = Length;
  type Radius = Distance;
  type Diameter = Distance;
  type Area = Real(final quantity="Area", final unit="m2");
  type Volume = Real(final quantity="Volume",
       final unit="m3");
  type Time = Real(final quantity="Time", final unit="s");
  type AngularVelocity = Real(final unit="rad/s",
       final quantity="AngularVelocity",
       displayUnit="rev/min");
  type AngularAcceleration = Real(final unit="rad/s2",
       final quantity="AngularAcceleration");
  type Velocity = Real(final quantity="Velocity",
       final unit="m/s");
  type Acceleration = Real(final quantity="Acceleration",
       final unit="m/s2");
  ...
end SIunits;
```

D.2 PERIODIC PHENOMENON

```
within Modelica;
```

```
package SIunits
  ...
  type Period = Real(final quantity="Time", final unit="s");
  type Frequency = Real(final quantity="Frequency",
       final unit="Hz");
  type AngularFrequency = Real(final unit="s-1",
       final quantity="AngularFrequency");
  type AmplitudeLevelDifference = Real(final unit="dB",
       final quantity="AmplitudeLevelDifference");
  type PowerLevelDifference = Real(final unit="dB",
       final quantity="PowerLevelDifference");
  ...
end SIunits;
```

D.3 MECHANICS

```
within Modelica;

package SIunits
  ...
  type Mass = Real(final quantity="Mass",
       final unit="kg", min=0);
  type Density = Real(final quantity="Density",
       final unit="kg/m3", displayUnit="g/cm3", min=0);
  type Momentum = Real(final quantity="Momentum",
       final unit="kg.m/s");
  type AngularMomentum = Real(final quantity="AngularMomentum",
       final unit="kg.m2/s");
  type MomentOfInertia = Real(final quantity="MomentOfInertia",
       final unit="kg.m2");
  type Inertia = MomentOfInertia;
  type Force = Real(final quantity="Force", final unit="N");
  type Torque = Real(final quantity="Torque",
       final unit="N.m");
  type Pressure = Real(final quantity="Pressure",
       final unit="Pa", displayUnit="bar");
  type AbsolutePressure = Pressure (min=0);
  type Stress = Real(final unit="Pa");
  type Strain = Real(final quantity="Strain", final unit="1");
  type ModulusOfElasticity = Stress;
  type CoefficientOfFriction = Real(final unit="1",
       final quantity="CoefficientOfFriction");
  type DynamicViscosity = Real(final unit="Pa.s",
       final quantity="DynamicViscosity", min=0);
  type KinematicViscosity = Real(final unit="m2/s",
       final quantity="KinematicViscosity", min=0);
  type Work = Real(final quantity="Work", final unit="J");
  type Energy = Real(final quantity="Energy", final unit="J");
  type PotentialEnergy = Energy;
  type KineticEnergy = Energy;
```

```
  type Power = Real(final quantity="Power", final unit="W");
  type Efficiency = Real(final quantity="Efficiency",
       final unit="1", min=0);
  type MassFlowRate = Real(final quantity="MassFlowRate",
       final unit="kg/s");
  type VolumeFlowRate = Real(final quantity="VolumeFlowRate",
       final unit="m3/s");
  ...
end SIunits;
```

D.4 THERMODYNAMICS

```
within Modelica;

package SIunits
  ...
  type ThermodynamicTemperature = Real(final unit="K",
       final quantity="ThermodynamicTemperature",
       displayUnit="degC");
  type Temperature = ThermodynamicTemperature;
  type CelsiusTemperature = Real(final unit="degC",
       final quantity="CelsiusTemperature");
  type Heat = Real(final quantity="Energy", final unit="J");
  type HeatFlowRate = Real(final quantity="Power",
       final unit="W");
  type ThermalConductivity = Real(final unit="W/(m.K)",
       final quantity="ThermalConductivity");
  type CoefficientOfHeatTransfer = Real(final unit="W/(m2.K)",
       final quantity="CoefficientOfHeatTransfer");
  type ThermalResistance = Real(final unit="K/W",
       final quantity="ThermalResistance");
  type ThermalConductance = Real(final unit="W/K",
       final quantity="ThermalConductance");
  type ThermalDiffusivity = Real(final unit="m2/s",
       final quantity="ThermalDiffusivity");
  type HeatCapacity = Real(final unit="J/K",
       final quantity="HeatCapacity");
  type SpecificHeatCapacity = Real(final unit="J/(kg.K)",
       final quantity="SpecificHeatCapacity");
  type RatioOfSpecificHeatCapacities = Real(final unit="1",
       final quantity="RatioOfSpecificHeatCapacities");
  type Entropy = Real(final quantity="Entropy",
       final unit="J/K");
  type SpecificEntropy = Real(final unit="J/(kg.K)",
       final quantity="SpecificEntropy");
  type InternalEnergy = Heat;
  type Enthalpy = Heat;
  type SpecificEnergy = Real(final unit="J/kg",
       final quantity="SpecificEnergy");
  type SpecificInternalEnergy = SpecificEnergy;
```

```
  type SpecificEnthalpy = SpecificEnergy;
  ...
end SIunits;
```

D.5 ELECTRICITY

```
within Modelica;

package SIunits
  ...
  type ElectricCurrent = Real(final unit="A",
       final quantity="ElectricCurrent");
  type Current = ElectricCurrent;
  type ElectricCharge = Real(final unit="C",
       final quantity="ElectricCharge");
  type Charge = ElectricCharge;
  type ElectricPotential = Real(final unit="V",
       final quantity="ElectricPotential");
  type Voltage = ElectricPotential;
  type PotentialDifference = ElectricPotential;
  type ElectromotiveForce = ElectricPotential;
  type Capacitance = Real(final unit="F", min=0,
       final quantity="Capacitance");
  type Inductance = Real(final unit="H", min=0,
       final quantity="Inductance");
  type Resistance = Real(final unit="Ohm", min=0,
       final quantity="Resistance");
  type Resistivity = Real(final quantity="Resistivity",
       final unit="Ohm.m");
  type Conductivity = Real(final quantity="Conductivity",
       final unit="S/m");
  type Impedance = Resistance;
  type Conductance = Real(final unit="S", min=0,
       final quantity="Conductance");
  ...
end SIunits;
```

D.6 PHYSICAL CHEMISTRY

```
within Modelica;

package SIunits
  ...
  type AmountOfSubstance = Real(final unit="mol", min=0,
       final quantity="AmountOfSubstance");
  type MolarMass = Real(final quantity="MolarMass",
       final unit="kg/mol");
  type MolarVolume = Real(final quantity="MolarVolume",
       final unit="m3/mol");
  type Concentration = Real(final quantity="Concentration",
       final unit="mol/m3");
```

```
  type MassFraction = Real(final quantity="MassFraction",
       final unit="1");
  type MoleFraction = Real(final quantity="MoleFraction",
       final unit="1");
  type ChemicalPotential = Real(final unit="J/mol",
       final quantity="ChemicalPotential");
  type PartialPressure = Real(final unit="Pa", min=0,
       displayUnit="bar", final quantity="Pressure");
  type ActivityCoefficient = Real(final unit="1",
       final quantity="ActivityCoefficient");
  type StoichiometricNumber = Real(final unit="1",
       final quantity="StoichiometricNumber");
  ...
end SIunits;
```

Appendix E
Modelica Standard Library: Constants

```
within Modelica;

package Constants
  // Mathematical constants
  constant Real e=Modelica.Math.exp(1.0);
  constant Real pi=2*Modelica.Math.asin(1.0);

  // Machine dependent constants
  constant Real eps=1.e-15
    "Biggest number such that 1.0 + EPS = 1.0";
  constant Real small=1.e-60 "Smallest Real number";
  constant Real inf=1.e+60 "Biggest Real number";
  constant Integer Integer_inf=2147483647
    "Biggest Integer number";

  // Constants of nature
  constant Modelica.SIunits.Velocity c=299792458
    "Speed of light inside a vacuum";
  constant Modelica.SIunits.Acceleration g_n=9.80665
    "Standard acceleration of gravity on earth";
  constant Real G(final unit="m3/(kg.s2)") = 6.673e-11
    "Newtonian constant of gravitation";
  constant Real h(final unit="J.s") = 6.62606876e-34
    "Planck constant";
  constant Real k(final unit="J/K") = 1.3806503e-23
    "Boltzmann constant";
  constant Real R(final unit="J/(mol.K)") = 8.314472
    "Molar gas constant";
  constant Real sigma(final unit="W/(m2.K4)") = 5.670400e-8
    "Stefan-Boltzmann constant";
  constant Real N_A(final unit="1/mol") = 6.02214199e23
    "Avogadro Constant";
```

```
  constant Real mue_0(final unit="N/A2") = 4*pi*1.e-7
    "Magnetic Constant";
  constant Real epsilon_0(final unit="F/m") = 1/(mue_0*c*c)
    "Electric Constant";
  constant Modelica.SIunits.CelsiusTemperature T_zero=-273.15
    "Absolute zero temperature";
end Constants;
```

Appendix F
Modelica Standard Library: Math Functions

F.1 GEOMETRIC FUNCTIONS

- Sine: `y = Modelica.Math.sin(u);`
- Cosine: `y = Modelica.Math.cos(u);`
- Tangent: `y = Modelica.Math.tan(u);` where $u \neq \frac{(2*n-1)*\pi}{2}$.

F.2 INVERSE GEOMETRIC FUNCTIONS

- Inverse sine: `y = Modelica.Math.asin(u);` where $-1 \leq u \leq 1$.
- Inverse cosine: `y = Modelica.Math.acos(u);` where $-1 \leq u \leq 1$.
- Inverse tangent: `y = Modelica.Math.atan(u);`
- Four quadrant inverse tangent: `y = Modelica.Math.atan2(u,v);`

F.3 HYPERBOLIC GEOMETRIC FUNCTIONS

- Hyperbolic sine: `y = Modelica.Math.sinh(u);`
- Hyperbolic cosine: `y = Modelica.Math.cosh(u);`
- Hyperbolic tangent: `y = Modelica.Math.tanh(u);`

F.4 EXPONENTIAL FUNCTIONS

- Exponential, base e: `y = Modelica.Math.exp(u);`
- Natural logarithm: `y = Modelica.Math.log(u);` where $u > 0$
- Base 10 logarithm: `y = Modelica.Math.log10(u);` where $u > 0$

GLOSSARY

acausal An approach to modeling where no assumptions are made about causality when developing component models. This leads to be reusability of the developed models because they contain fewer assumptions about the context of their use.

attributes An attribute is associated with quantities such as parameters and variables. Attributes provide additional information about that quantity such as upper and lower bounds or physical units.

across variables Variables which represent the "driving force" across a component (see Section 1.3.2 for more details).

algebraic loop An algebraic loop is a coupled, simultaneous system of equations. As a result, unlike a conventional block diagram it is not possible to solve such a system one variable at a time.

black box A model for which the implementation details are hidden.

block diagram Block diagrams are used to explicitly describe the set of mathematical operations that must be performed in order to compute a set of unknowns (outputs) from a set of knowns (inputs). Furthermore, the blocks in such a diagram may have their own internal states as well (*e.g.*, an integrator block).

blocks Components with clearly defined inputs and outputs used to create block diagrams.

causality Causality is the cause and effect relationship between components in a complete physical system. In general, it is not possible to determine the causality of individual components. For example, does the voltage drop across a resistor result from a current going through the resistor or is it the current that results from a voltage drop? It is only once a complete system of components has been constructed that the causality can be determined.

coefficient of restitution A measure of how much momentum is conserved in an inelastic collision.

component A component is an instance of a model. So, for a given model (*e.g.*, a resistor model), the actual instances (*e.g.*, the resistors) would be components. In object-oriented programming, a component is analogous to an object.

conservation equations *see* **conservation law**

conservation laws Conservation laws state that the amount of some quantity (*e.g.*, energy or mass) does not change over time. This quantity is often

called a conserved quantity. Conservation laws are used to derive conservation equations which explicitly state that the time derivative of the conserved quantity is zero.

conservative system A system in which some quantity (*e.g.*, energy or mass) is conserved.

constitutive equations Relationships between the potentials (*i.e.*, across variables) in a system and the flow of conserved quantities (*i.e.*, through variables). Examples include Hook's law, Ohm's law, Fourier's law, Newton's law, *etc.*

control systems Control systems use information from sensors to determine how actuators should be used to achieve a desired response from a dynamic system.

control volume A control volume is the thermodynamic equivalent of a free body diagram in mechanics. A control volume contains energy and mass (generally in liquid or gaseous form). Any change in mass or energy must be due to some external influence (*e.g.*, work, flow).

DAE *see* **differential-algebraic equations**

declaration When a component is instantiated (either in a system or inside another component), that is called a component declaration.

definition The description of all variables, parameters and equations associated with a model is called the model definition.

derived types A type which is created by specializing one of the intrinsic types (*i.e.*, `Real`, `Integer`, `String` and `Boolean`).

diagram view The view of a model from the "inside". This view reveals all internal subcomponents and connections.

differential-algebraic equations Systems of equations that involve both differential and algebraic equations. Such systems have the general mathematical form $f(x, \dot{x}, t) = 0$.

discrete variables Variables with values that are piecewise constant with respect to time.

domain neutral Something is domain neutral if it does not exhibit a bias toward a specific engineering domain.

encapsulation The ability to hide the details of a component. Ideally, an understanding of these details should not be necessary. In this way, the

amount of detail which must be understood in order to comprehend a complete system is reduced.

event Something which occurs instantaneously at a specific time or when a specific condition occurs.

extensive property An extensive property is a material property which is related to the amount of mass present. Energy is an extensive property because if you remove half the mass in a homogeneous mixture, you will also remove half the energy. For most extensive properties there are associated intensive properties (*e.g.,* specific internal energy). *see* **intensive property**

explicit equation An equation where all the solution variables and/or their derivatives can be solved explicitly (*i.e.,* they are the only term on the left hand side of an equation). An example of an explicit equation is $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$. *see* **implicit equation**

first principles Modeling approach based on using conservation laws (*see* **Kirchhoff's current law**).

flow variables *see* **through variables**

free body diagram A diagram which shows all state information (*e.g.,* orientation, position) for a given component and all possible external influences (*e.g.,* torques, forces).

Kirchhoff's current law The sum of all current at a point must be zero.

hybrid system A system involving both continuous and discrete behavior.

icon view This view of a model from the "outside". This view attempts to hide the internal details and present a "black-box" representation which only includes the external connections of the model.

implicit equation An equation in which the solution variable does not appear by itself on the left hand side of an equation (*e.g.,* $f(T) = 0$). *see* **explicit equation**

initial value problem A mathematical problem which is solved by starting from a set of initial conditions and then integrating a system of differential equations.

intensive property An intensive property is a material property that is normalized to the amount of material. For example, temperature and concentration are both intensive properties. If you take mixture of uniform temperature or concentration and you remove half the mixture the remaining mixture will

continue to have the same temperature and concentration. *see* **extensive property**

interface Generally, the interface of an object consists of all object details visible externally. In Modelica, this would usually include parameters and connectors. Recognizing that several model types have common interfaces leads to the development of partial models like the one shown in Example 4.1.

instance When a declaration involving a model or type is made, an instance is created. This instance has its own set of parameters and attributes separate from other instances of the same model or type. In other words, `Resistor` is a model and `R1` could be specific instance of a `Resistor` with its own value for resistance.

left limit The limit of any time varying value when approached from "the left" (*i.e.,* values of time lower than the time at which the limit is being taken).

local variable A variable which is only visible to the entity to which it belongs. For example, a local variable in a function is only visible to that function.

model A model is a behavioral description. For example, a model of a resistor is described by Ohm's law. The model is a description of resistor behavior not the resistor itself. In other words, it is important to separate the idea of a resistor model (*i.e.,* $V = I*R$) from the resistor instances (components with different values of resistance, R). If you are familiar with object-oriented programming, a model is analogous to a class.

model developer A person who is responsible for creating models. For large simulation projects the model developer, model user and end user may be different people.

modification Modifications are used to override the defaults in a declaration. Modifications typically involve overriding values for attributes of a type or instance.

network A collection of components connected together. Often, energy flows between components in networks according to the constitutive equations of the components.

node In networks, nodes are the points at which components are connected. The large black circles in Figure 3.1 are nodes in that particular network.

package A package refers to a collection of Modelica models, which are meant to be used together. For example, an electrical package would likely include definitions of resistor, capacitor and inductor models.

package hierarchy A diagram showing what is contained within a package. In object-oriented terminology, this is a diagram showing the "has-a" relationships. *see* **inheritance hierarchy**

parameter expression An expression which does not change with time.

partial differential equation An equation which contains derivatives with respect to spatial dimensions and possibly (although not always) derivatives with respect to time.

PDE *see* **partial differential equation**

plant model A plant model is the model of the physical system (and its associated dynamic response) for which a controller is designed. Sensors and actuators usually define the boundary between the controller and the plant.

physical modeling This type of modeling is characterized by a first principles approach to formulating behavioral equations. Physical modeling refers to what control system engineers call "plant modeling".

physical types Physical types give detailed information about the physical significance of quantities. For example, the voltage in a circuit is usually represented by the `Voltage` type contained in the `Modelica.SIunits` package. In this way, additional information (*e.g.,* units or limits) can be associated with that quantity. In this way, variables are treated as more than just numbers with a value.

quantity A quantity refers to those entities which have a value (*e.g.,* the resistance of a resistor). In Modelica, all values are either real, integer, string or boolean. Furthermore, a quantity might have different levels of variability (*i.e.,* variables, parameters or constants) and it might be a scalar or an array.

schematic A schematic is a graphical representation of a system containing individual components and their connections two each other.

semantics The semantics of the Modelica language define what the intent of a syntactical construct is. In essence, the semantics of the language are the "meaning" that gets associated with keywords, operators and so on.

short definition A definition which is so similar to an existing definition that it can be defined in terms of modifications on the existing definition.

side effects A function is said to have side effects if it store information to be used during subsequent invocations. Externally, such side effects cause the function to return different results even though the same inputs are passed in.

solver The software responsible for solving the system of hybrid DAEs which result from a Modelica model.

state space form An equation system is said to be in state space form when it is represented as:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

stiffness A property of systems with coupling between fast dynamics and slow dynamics. This property can have a large impact on performance and effects some solvers more than others.

subcomponent A subcomponent is used to refer to components which are contained within other components. For example, a resistor might be a subcomponent of another component like an electrical circuit. Furthermore, the electrical circuit could be a subcomponent of an appliance. Subcomponents are used to form hierarchical models. In object-oriented programming, a subcomponent would be a member object.

symbolic manipulation Using algebra to rearrange equations into a form that is easier to solve or results in more efficient simulation.

system model A system model is a model which is completely self-contained. In other words, it does not have any external connections and it represents a complete model.

through variables Variables which represent quantities flowing through a component (see Section 1.3.2 for more details).

type Modelica is a strongly typed language. Every entity in Modelica has a type. Each quantity has a type indicating whether it is a floating point, integer or boolean. Each component has a type indicating what model it is an instance of.

variability The variability of a quantity is an indication how free that quantity is to change (see Section 2.5.2.3).

variability qualifier A qualifier that restricts how a variable may change. The variability qualifiers in Modelica are `constant`, `parameter` and `discrete`.

References

Andersson, M. (1994). *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Lund Institute of Technology, Department of Automatic Control, Lund, Sweden. ISRN LUTFD2/TFRT–1043–SE.

Åström, K. J., Elmqvist, H., and Mattsson, S. E. (1998). Evolution of continuous-time modeling and simulation. In *The 12th European Simulation Multiconference*.

Barton, A. (2000). Introductory chemical kinetics notes. http://wwwscience.murdoch.edu.au/teaching/m237/m237notes01.html.

Beater, P. (2000). Modeling and digital simulation of hydraulic systems in design and engineering education using Modelica and HyLib. In *Proceedings of the 2000 Modelica Workshop.*

Bower, J. M. and Beeman, D. (1994). *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.* Springer-Verlag.

Bowles, P., Tiller, M., Elmqvist, H., Brück, D., Mattsson, S. E., Möller, A., Olsson, H., and Otter, M. (2001). Feasibility of detailed vehicle modeling. In *Proceedings of the 2001 SAE Congress and Exposition.*

Breunese, A. P. J. and Broenink, J. F. (1997). Modeling mechatronic systems using the SIDOPS+ language. In *Proceedings of ICBGM'97, 3rd International Conference on Bond Graph Modeling and Simulation*, volume 19 of *Simulation Series*, pages 301–306, Phoenix, Arizona. SCS Publishing. ISBN 1-56555-050-1.

Broenink, J. F. (1997). Modeling, simulation and analysis with 20-sim. *Journal A, (Benelux quarterly journal on automatic control)*, 38(3).

Brogan, W. L. (1991). *Modern Control Theory*. Prentice Hall, third edition edition.

Cellier, F. E. (1991). *Continuous System Modeling*. Spring Verlag.

Clauss, C., Schneider, A., Leitner, T., and Schwarz, P. (2000). Modelling of electrical circuits with Modelica. In *Proceedings of the 2000 Modelica Workshop.*

Earley, J. E. (1998). An introductory course in modeling dynamic chemical and ecological systems. http://www.georgetown.edu/earleyj/ch42.html.

Elmqvist, H. (1978). *A Structured Model Language for Large Continuous Systems.* PhD thesis, Lund Institute of Technology, Sweden, Department of Automatic Control.

Elmqvist, H., Brück, D., Mattsson, S. E., Olsson, H., and Otter, M. (2001). *Dymola - Dynamic Modeling Language - User's Manual.* Dynasim AB, Research Park Ideon, SE-223 70 Lund, Sweden. http://www.Dynasim.se.

Elmqvist, H., Cellier, F. E., and Otter, M. (1993). Object-oriented modeling of hybrid systems. In *In Proceedings of European Simulation Symposium, ESS'93.* The Society of Computer Simulation.

Elmqvist, H., Mattsson, S. E., and Otter, M. (1998). Modelica - an international effort to design an object-oriented modeling language. In *Proceedings of the 1998 Summer Computer Simulation Conference.*

Ernst, T., Jähnichen, S., and Klose, M. (1997). The architecture of the smile/m simulation environment. In *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 6, pages 653–658.

Fitzpatrick, D. and Miller, I. (1995). *Analog Behavioral Modeling With The Verilog-A Language.* Kluwer Academic Publishers.

Fowler, A. C. (1997). *Mathematical Models in the Applied Sciences.* Cambridge University Press.

Fritzson, P., Viklund, L., Fritzson, D., and Herber, J. (1995). High-level mathematical modeling and programming. *IEEE Software.*

Heinkel, U., Padeffke, M., Hass, W., Buerner, T., Glauert, W., Wahl, M., Braisz, H., Gentner, T., and Grassman, A. (2000). *The VHDL Reference: A Practical Guide to Computer-Aided Integrated Circuit Design (including VHDL-AMS).* Jon Wiley and Sons.

Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544.

Jeandel, A., Boudaud, F., and Larivière, E. (1997). *ALLAN.* Gaz de France, France, simulation release 3.1 description edition.

Jeandel, A., Boudaud, F., Ravier, P., and Buhsing, A. (1996). U.L.M: Un Langage de Modélisation, a modelling language. In *In Proceedings of the CESA'96 IMACS Multiconference.* IMACS. Lilli, France.

Larsson, M. (2000). Objectstab - a Modelica library for power system stability studies. In *Proceedings of the 2000 Modelica Workshop.*

Mattsson, S. E., Andersson, M., and Åström, K. J. (1993). Object-oriented modelling and simulation. In *CAD for Control Systems*, chapter 2, pages 31–69. Marcel Dekker, New York.

Mattsson, S. E. and Söderlind, G. (1993). Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal of Scientific Computing*, 14:677–692.

Ogata, K. (1978). *System Dynamics*. Prentice-Hall.

Otter, M., Dempsey, M., and Schlegel, C. (2000). Package PowerTrain. a Modelica library for modeling and simulation of vehicle power trains. In *Proceedings of the 2000 Modelica Workshop.*

Otter, M., Elmqvist, H., and Mattsson, S. E. (1999). Hybrid modeling in modelica based on synchronous data flow principles. In *IEEE Symposium on Computer-Aided Control System Design.*

Pantelides, C. D. (1988). The consistent initialization of differential-algebraic systems. *SIAM Journal of Scientific and Statistical Computing*, 9:213–231.

Pauling, L. (1988). *General Chemistry*. Dover, Mineola, NY.

Pottier, M. (1983). Extensions et applications envisageables des procédures complémentaires établies pour accéder au progiciel ASTEC 3 : ALLAN 6. Technical report, Gaz de France, France. Technical report M.D6 no. 4034.

Sahlin, P., Bring, A., and Sowell, E. F. (1996). The neutral model format for building simulation, version 3.02. Technical report, The Royal Institute of Technology, Stockhom, Sweden.

Stefan, J. (1891). Ueber die theorie der eisbildung, insbesondere über die eisbildung im polarmeere. *Ann. Phys. Chem.*, 42:269–286.

Strauss, J. C., Augustin, D. C., Fineberg, M. S., Johnson, B. B., Linebarger, R. N., and Sanson, F. H. (1967). The sci continuous system simulation language. *Simulation.*

Tiller, M., Bowles, P., Elmqvist, H., Brück, D., Mattsson, S. E., Möller, A., Olsson, H., and Otter, M. (2000). Detailed vehicle powertrain modeling in modeling. In *Proceedings of the 2000 Modelica Workshop.*

Tummescheit, H. and Pitz-Paal, R. (1997). Simulation of a solar thermal central receiver power plant. In *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 6, pages 671–676.

About the Author

Dr. Michael Tiller is a Technical Specialist in the Ford Research Laboratory at Ford Motor Company. He is also a member of the Modelica Association. Dr. Tiller received his Ph.D. from the Department of Mechanical and Industrial Engineering at the University of Illinois, Urbana-Champaign. His Ph.D. work focused on developing reusable simulation software for sensitivity analysis with applications to solidification processing.

Index

C

D

E

F

G