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

Kluwer Academic Publishers
Boston/Dordrecht/London

Distributors for North, Central and South America:
Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061 USA
Telephone (781) 871-6600
Fax (781) 681-9045
E-Mail <kluwer@wkap.com>

Distributors for all other countries:
Kluwer Academic Publishers Group
Distribution Centre
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS
Telephone 31 78 6576 000
Fax 31 78 6576 454
E-Mail <services@wkap.nl>


Electronic Services <http://www.wkap.nl>

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://extra.springer.com.
ISBN 0-7923-7367-7 (alk. paper)
1. Computer simulation. 2. Object-oriented methods (Computer science) I. Title. Series.

QA76.9.C65 T55 2001
005'.35133—dc21

2001029416

Printed on acid-free paper.

Printed in the United States of America.

Contents

List of Figures

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.

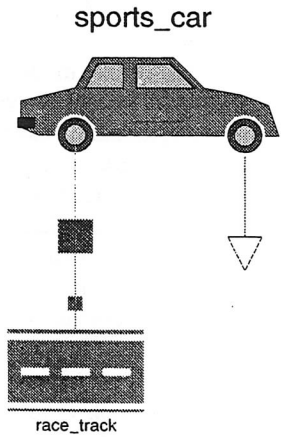

Figure 1.1. A 0-100 kilometer per hour test.

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.

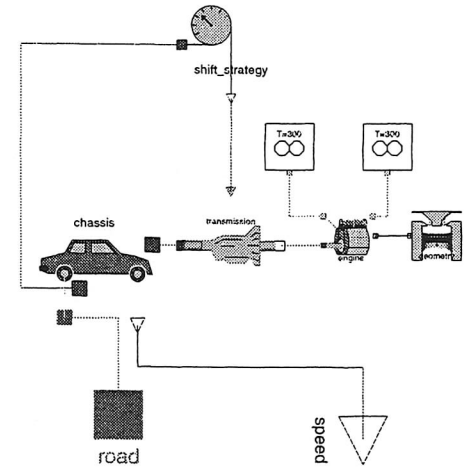

Figure 1.2. Taking a look at what is under the hood.

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

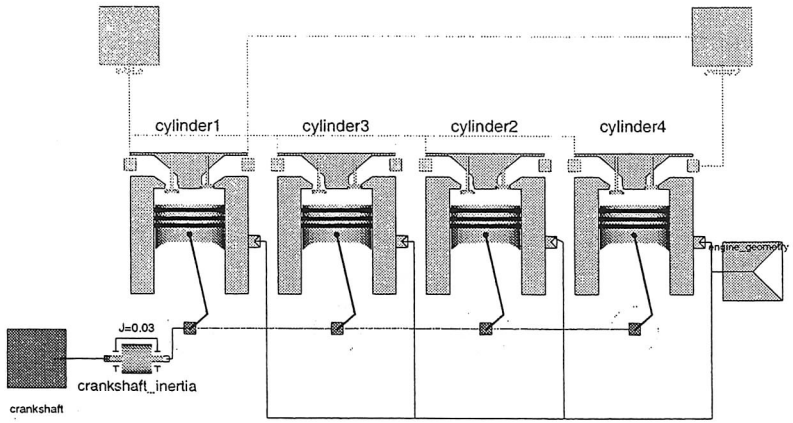

Figure 1.3. Looking inside the engine.

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

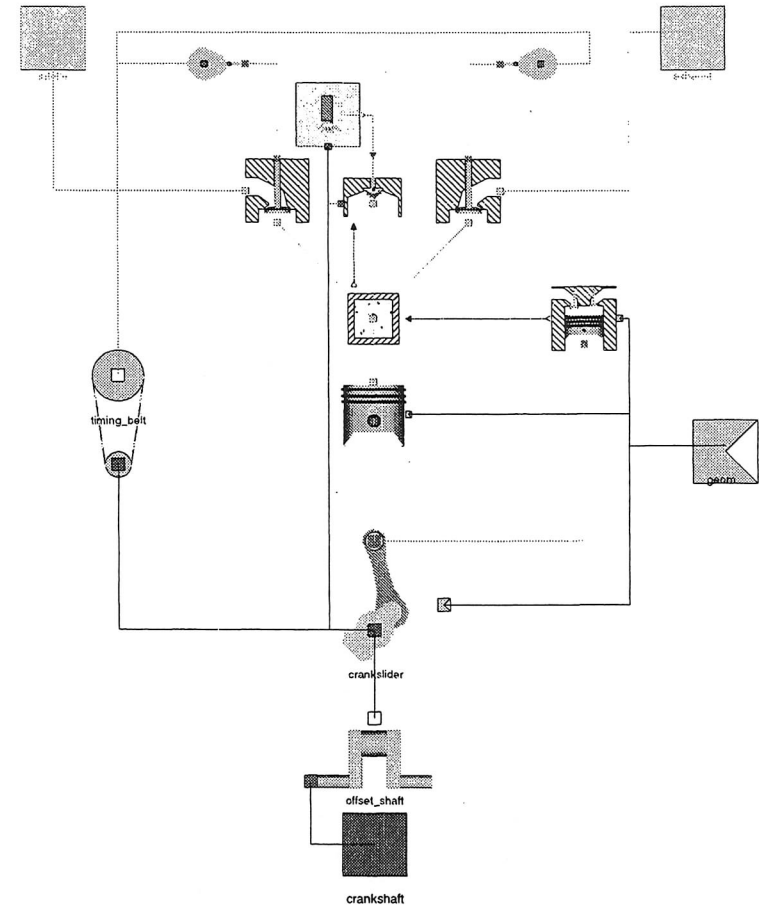

Figure 1.4. Looking inside an individual engine cylinder.

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.

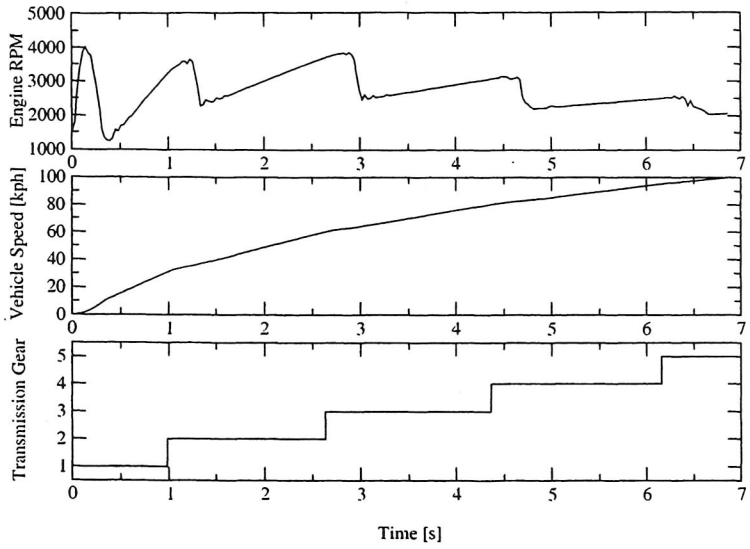

Figure 1.5. Simulation results from a sample race.

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.

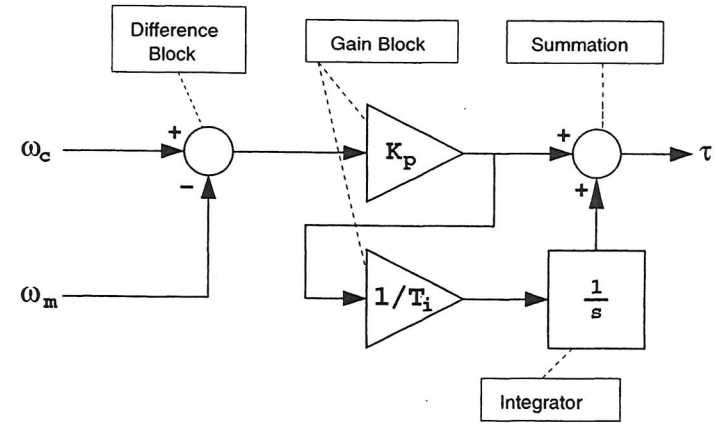

Figure 1.6. PI Controller.

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.

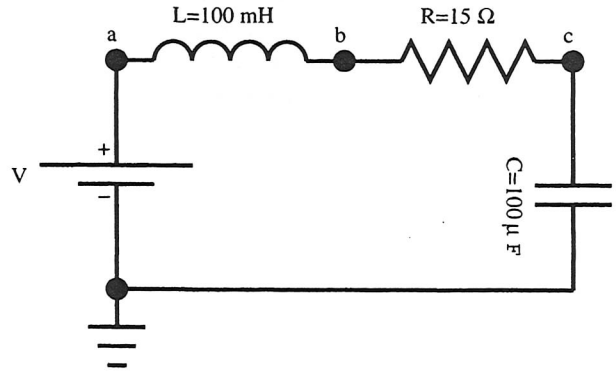

Figure 1.7. RLC circuit schematic.

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:

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

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

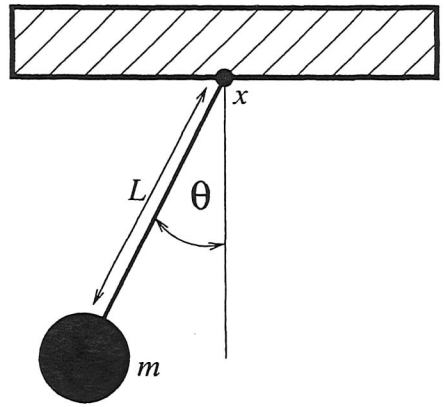

Figure 2.1. A simple pendulum

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

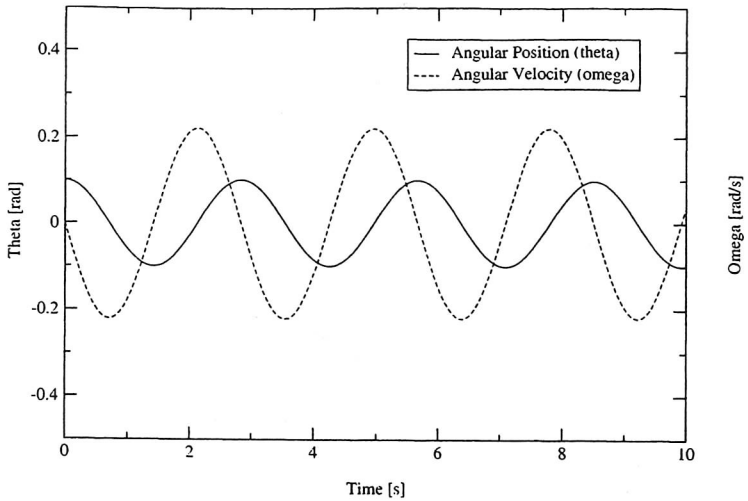

Figure 2.2. Solution for $\theta(t)$ given L=2, $\theta(0) = 0.1$ and $\omega(0) = 0$.

`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.

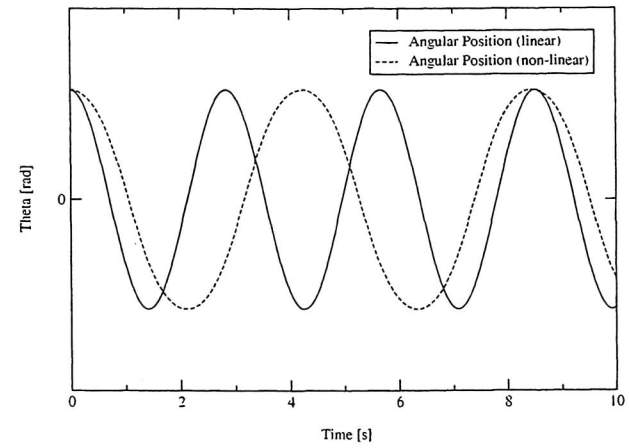

Figure 2.3. Linear and non-linear solutions for $\theta(t)$ given L=2, $\theta(0) = 2.3$ and $\omega(0) = 0$.

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.

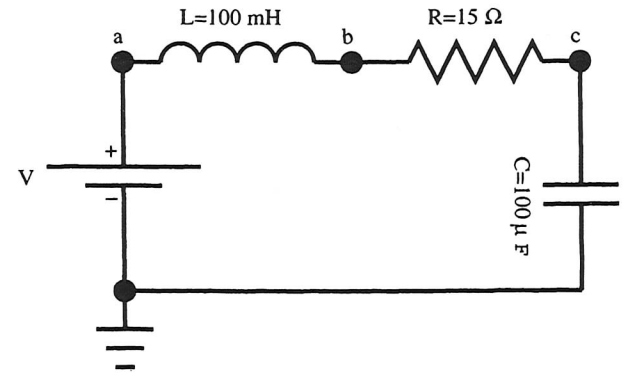

Figure 2.4. An RLC circuit.

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:

$$\begin{aligned} i_V - i_L &= 0 && (2.8) \\ i_L - i_R &= 0 && (2.9) \\ i_R - i_C &= 0 && (2.10) \end{aligned}$$

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:

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

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

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

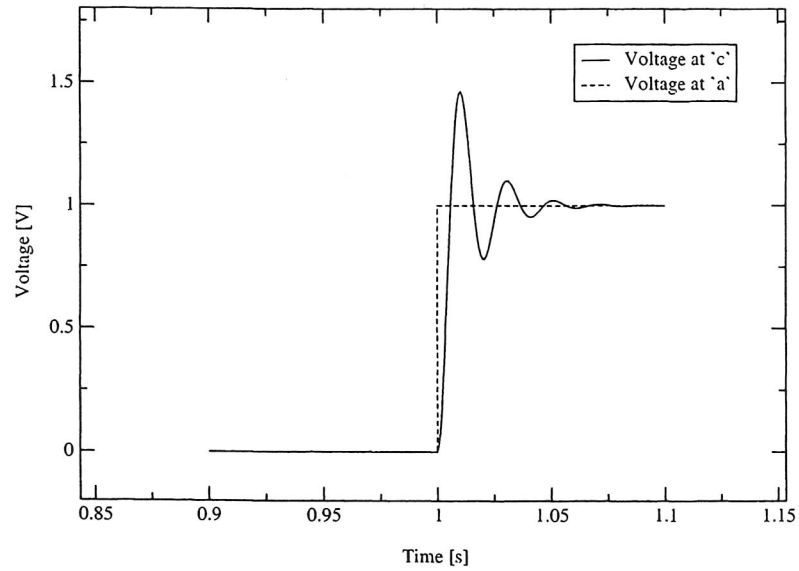

Figure 2.5. Voltage response of model `RLC`.

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.

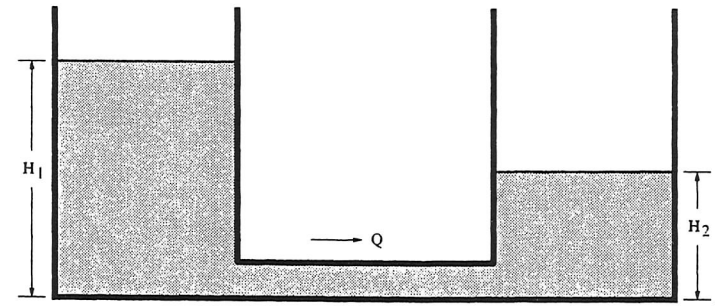

Figure 2.6. Two hydraulic tanks filled with liquid.

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.

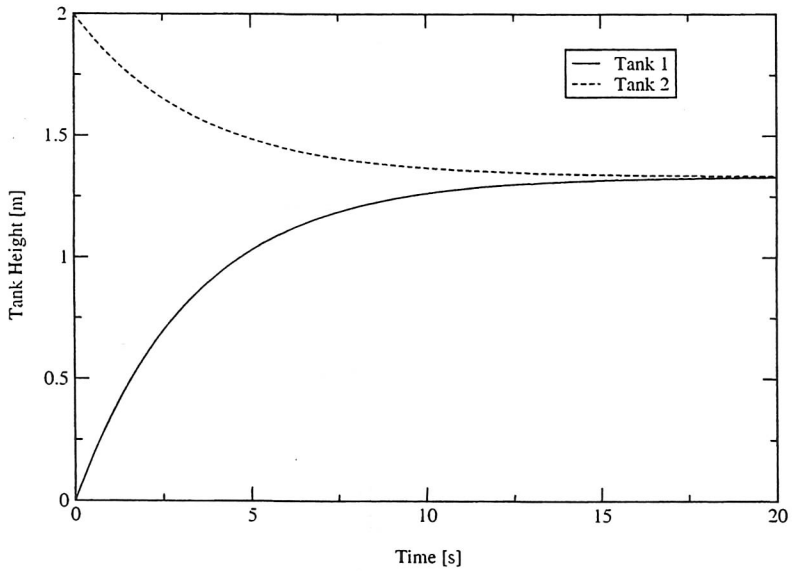

Figure 2.7. Solution with initial conditions `H1=0` and `H2=2`

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.