
Modelica by Example

Release v0.6.0

Michael M. Tiller

Jun 03, 2019

CONTENTS

Front Matter	I
Preface	I
Acknowledgments	I
Technical	I
Personal	II
Contributors	II
Tools	IV
Introduction	V
What is Modelica?	V
Why Modelica?	V
What Modelica allows me to do	V
Dedication	V
1 Describing Behavior	1
1.1 Basic Equations	1
1.1.1 Examples	1
1.1.2 Review	27
1.2 Discrete Behavior	42
1.2.1 Examples	42
1.2.2 Review	82
1.3 Vectors and Arrays	87
1.3.1 Examples	87
1.3.2 Review	101
1.4 Functions	116
1.4.1 Examples	116
1.4.2 Review	137
2 Object-Oriented Modeling	151
2.1 Packages	151
2.1.1 Examples	151
2.1.2 Review	157
2.2 Connectors	169
2.2.1 Introduction	169
2.2.2 Examples	170
2.2.3 Review	177
2.3 Components	183
2.3.1 Examples	183
2.3.2 Review	252
2.4 Subsystems	264
2.4.1 Examples	264
2.4.2 Review	297
2.5 Architectures	304
2.5.1 Examples	304
2.5.2 Review	345

3 Indices and tables	355
Bibliography	357
Index	359

FRONT MATTER

Preface

This book is a bit unusual in the way it is structured. This is because it is expected that most readers will be reading an HTML version of the book. As a result, the book makes heavy use of hyperlinking to allow users to explore the contents of the book as best suits their needs. For the most part, this linking should only enhance eBook versions of the book, but it may not always translate well into a print medium format. But every attempt has been made to provide a quality result in all supported formats.

One aspect of the book worth remarking on at this point is the fact that there are a couple of different “flows” supported in the book. Overall, the book attempts to present material in a logical order in the progression of chapters shown in the table of contents. This means that the first few chapters focus on expressing different types of mathematical behavior and deferring discussions about building more structured models (e.g. packages, component models, subsystems, *etc.*) until later. However, when reading about a particular example, the provided links will make it possible to sidestep this normal flow of topics and simply continue with further expansion on that particular example in subsequent chapters (which introduce additional language features). Hopefully, this approach enhances the reading experience without disorienting the user.

Most chapters are organized into three parts. The first part introduces the topics to be discussed in the chapter. Next, an extensive set of examples are used to demonstrate the language features relevant to the topics introduced in part one. Note that each example typically introduces a new topic, so it is best to review all the examples to provide the most complete coverage of these topics. Finally, most chapters include a review of the topics and a summary of any details not found in the examples to provide thorough coverage of the topics.

Acknowledgments

This book is the culmination of so many lectures, books, discussions, conferences, *etc.* that it is impossible to provide a full accounting to all the people who’ve had an impact on the book. Undoubtedly, there are people I’ve failed to mention here and I apologize in advance to anyone who feels they’ve been overlooked.

Technical

On the subject of Modelica, the primary acknowledgment has to be to Hilding Elmquist for not only having the technical vision to recognize the potential impact that symbolic manipulation techniques could have on solving engineering problems but having the leadership to push this vision forward as an open standard. Hilding is the undisputed “father of Modelica” and he deserves considerable credit for all that Modelica has become.

Second only to Hilding is Martin Otter, a man who has worked tirelessly for the advancement of Modelica. I can honestly say that Martin works harder than anybody else I know. He has not only made countless technical contributions to Modelica, but he’s taken on the unenviable and generally thankless task of managing the Modelica Association. It is important to remember that having great technical ideas is by no means a recipe for success. Somebody has to be there to push and push and push those ideas along. I’d like to thank Martin for all his hard work in making the Modelica Association what it is today.

Of course, Martin isn’t alone in running the Modelica Association. The Modelica Association Board and the Members are also extremely important for raising awareness about Modelica. In the end, the Modelica Association is an organization committed to open standards that support engineering and I’d like to thank all the members for their hard work in this regard.

There wasn't always a Modelica Association, but there was always a Modelica Design Group. These are the people who come together several times a year and work on continuously improving the Modelica language. It is astounding how much time and energy people have volunteered toward this effort. I'd like to thank all those people who've participated in the development of the Modelica Language and the Modelica Standard Library.

Personal

On a personal note, I am forever indebted to my mother, my father, my wife, my kids and my in-laws for their endless support of my passion for science, engineering and math. They are responsible for cultivating and sustaining my interests in these topics so I owe all the energy and time I'm able to apply to projects like this one to them.

The very idea of a book that is available for free might seem radical to some. The basic premise of this project was largely inspired by listening to and reading the works of Cory Doctorow and Lawrence Lessig. I'd like to thank them for opening my eyes to the idea of Creative Commons licensing. I'd also like to thank Dietmar Winkler for the many discussions we had on alternative publishing models. We would frequently discuss the ideas of Doctorow and Lessig and how their ideas could be applied to the creation of more accessible content for the Modelica world.

Looking back, I feel very fortunate to have worked for several companies that supported my involvement in Modelica. I first got involved with Modelica while I was working at Ford Motor Company and I was fortunate that they were willing to sponsor my participation in many different Modelica related events. After Ford, I went to work at Emmeskay (eventually acquired by LMS). I benefited enormously from the interactions I had with my Emmeskay family. In particular, I'd like to thank my partners, Swami Gopalswamy and Shiva Shivashankar for giving me the opportunity to be part of Emmeskay and for being great friends. While at Emmeskay, it was a privilege to work with Michael Sasena and John Batteh on several Modelica related projects. Emmeskay was an incredible company and this was entirely a reflection on the great group of people who worked there. Finally, I'd like to thank Dassault Systèmes for giving me the opportunity to work with all the excellent people there as well. In particular, I'd like to thank Hilding Elmquist and Marc Frouin for encouraging me to come work there. I'd also like to thank Martin Malmheden, Dag Brück and Sandrine Loembe for all the good times during my excellent year in Paris.

Contributors

This project was really an experiment to see if the Kickstarter approach to publishing could be applied to a niche technical field like Modelica. I was pleasantly surprised to see that it could and that this project had enough support to be funded. For that reason, I'd like to thank the backers of the Kickstarter project. In particular, I'd like to thank the following people for their exceptionally generous contributions:

- Hilding Elmquist
- Robert Norris
- Matthias Thorade
- Henning Francke
- Yang Ji
- Christoph Höger
- Philipp Mossmann
- John Batteh
- Dirk Zimmer
- Jan Brugård
- Swami Gopalswamy

- Peter Aronsson
- Michael P. Case
- Markus Groetsch
- Vicente Ramírez Perea
- Tisha Villanueva
- Adrian Pop
- Nimalendiran Kailasanathan
- Kevin Davies
- Peter Harman
- Dietmar Winkler
- Johan Rhodin

I'd also like to thank the corporate sponsors:

- Gold Sponsors
 - CyDesign
 - Wolfram Research
 - Modelon
 - Maplesoft
 - Dassault Systèmes
- Silver Sponsors
 - Ricardo Software
 - ITI
 - Modelica Association
 - Global Crown Technology
 - Siemens
- Bronze
 - Suzhou Tongyuan
 - Open Source Modelica Consortium
 - DOFWare
 - Bausch-Gall GmbH
 - Technische Universität Hamburg/Harburg
 - Schlegel Simulation GmbH

This project shows the power of community to achieve the mutual goal of creating more quality educational material around Modelica. Literally, this project could not have happened without them.

The Kickstarter funding allowed me to commit time to this project, but I also had several people helping me on this project. First and foremost, I'd like to once again thank my father who helped proof-read the initial draft of this book. Proof-reading is a necessary but rather boring job so I think he deserves extract credit for making that sacrifice. Similarly, I'd like to thank Dietmar Winkler and Michael O'Keefe for providing additional feedback on the book content. Dietmar has also helped me test publishing issues related to supporting ePub and PDF formats.

I'd like to thank Jeff Waters for being the “voice of the sponsor”. I had several very productive discussions with Jeff during the course of writing this book to make sure that the layout and graphical design lived up to sponsors' expectations.

Tools

Building a book like this requires a lot of different tools. My productivity was amplified enormously by the use of these tools.

This book was written using Sphinx, a documentation generation tool that supports multiple outputs. Sphinx allows me to focus on the content of the book and takes care of generating the book in multiple formats.

In creating this book, I needed a way to test the models that appear in the book, generate simulation results for plots and generated JavaScript code that allows the browser integrated simulation capabilities in the HTML book. OpenModelica supported all of these use cases. But beyond that, I owe a big “Thank You” to Martin Sjölund and the OpenModelica team for quickly responding to various issues I had during the creation of the book. Many times I would see Martin on Skype late at night (Sweden time) and he was gracious enough to help me out.

The initial version of the book featured browser integrated simulation capabilities in the HTML version. These capabilities were only possible because of a tool called Emscripten which allows ordinary code in languages like C and C++ to be cross-compiled (via LLVM) into JavaScript. Although I knew this was possible, I didn’t really think this avenue was viable until I saw [the work of Tom Short¹](#) integrating OpenModelica and Emscripten. The browser integrated simulation capabilities were greatly enabled by his work in this area. Ultimately, I removed this functionality in part to enhance the use of the book on mobile devices (where memory constraints were an issue). But I am optimistic that at some point we can re-instate those capabilities. Nevertheless, I am greatful to those who contributed to the development of those capabilities.

This book was written using Git as the version control system and [GitHub²](#) for hosting. Most people think of the version control system as some arcane backup system. But version control systems are at the heart of collaboration and I’d like to see them used more widely in engineering. For this book, the “pull request” system from GitHub was very useful in incorporating feedback from reviewers. I’d like to again thank Dietmar Winkler for enlightening me about many different features in Git. In addition to GitHub, newer versions of the book leverage the Continuous Integration (CI) capabilities of Gitlab. I also leverage Docker to greatly improve the portability and repeatability of the build process.

I used the Emacs editor for this book. Despite the proliferation of really excellent editors that support a wide range of languages and platforms, Emacs remains my editor of choice for most writing work. It seems to support just about every type of file I need to edit out of the box. However, for the code developed for this book (*e.g.*, the interactive UI elements, the static page generation templates in TypeScript), I used Visual Studio Code which is a truly excellent open source editor that is gaining mindshare by leaps and bounds.

While the documentation processing was done with Sphinx, the actual HTML generation for v0.6.0+ is done using the server side rendering framework Next.js. This framework leverages the excellent React UI framework (which I use extensively) to create server rendered HTML (for good SEO) that can include dynamic components when rendered client side. This is truly a best of both worlds solution because it makes the book fast to load but provides instant interactivity in the browser. All my React based projects are written in the excellent TypeScript language and leverage the TSX functionality in order to create reusable and strongly typed UI components.

During the production of this book several tool vendors gave me access to their proprietary tools. I didn’t utilize these very much, but I wanted to acknowledge their generosity in providing me with temporary licenses. Specifically, I’d like to thank Dassault Systèmes, Maplesoft, Wolfram Research and ITI for giving me access to Dymola, MapleSim, SystemModeler and SimulationX, respectively.

Much of this book was written on a MacBook Air. My very first computer was an Apple //e. But since that time, I’ve worked mainly with PCs and Unix workstations. Most recently, I’ve done a great deal of development on Linux machines. I always dismissed using Macs because I was convinced they couldn’t support the kind of command-line oriented development work I typically do. I could not have been more wrong. The eco-systems for MacOS is almost identical to the one I was used to in the Linux world. I am able to seamlessly transition between MacOS and Linux environments without any

¹ <https://github.com/tshort/openmodelica-javascript>

² <http://github.com>

significant adjustments. The power and portability of the MacBook Air gave my entire work process a big boost.

Developing this book involved a lot of testing and debugging of HTML layout, styling and embedded JavaScript. Most of this work was done using Chrome but I've also used Firefox from time to time as well. I'd like to thank both the Mozilla Foundation and Google for creating such wonderful, standards compliant browsers.

Prior to v0.6.0, the style of the book owes a fair amount to the Semantic UI³ CSS framework. After v0.6.0, the styling is mostly due to BlueprintJS.

Introduction

If, for some reason, you are coming upon this book without any previous knowledge of Modelica, there are probably a couple of questions you have. Let me attempt to address these questions in the hope that they will intrigue you and cause you to dig deeper.

What is Modelica?

Modelica is a high-level declarative language for describing mathematical behavior. It is typically applied to engineering systems and can be used to easily describe the behavior of different types of engineering components (*e.g.*, springs, resistors, clutches, *etc.*). These components can then be combined into subsystems, systems or even architectures.

Why Modelica?

Modelica is compelling for several reasons. First and foremost, it is technically very capable. By using complex algorithms behind the scenes, Modelica compilers allow engineers to focus on high-level mathematical descriptions of component behavior and get high performance simulation capability in return without having to be deeply knowledgeable about complex topics like differential-algebraic equations, symbolic manipulation, numeric solvers, code generation, post-processing, *etc.*.

The key to Modelica's technical success is its support for a wide range of modeling formalisms that allow the description of both continuous and discrete behavior framed in the context of hybrid differential-algebraic equations. The language supports both causal (often used for control system design) and acausal (often used in creating schematic oriented physical designs) approaches *within the same model*.

Finally, another compelling aspect of Modelica is the fact that it was designed from the start as an open language. The specification is freely available and tool vendors are encouraged to support the import and export of Modelica (without being compelled to pay any royalty of any kind).

What Modelica allows me to do

Modelica is really an ideal language for modeling the behavior of engineering systems in nearly any engineering domain. It seamlessly supports both physical design and control design in a single language. It is also multi-domain so it doesn't impose any artificial boundaries that restrict its use to select engineering domains or systems. The result is that it provides a complete set of capabilities for building lumped system models of nearly any engineering system.

Dedication

³ <http://www.semantic-ui.com>

This book started as a Kickstarter project. The idea of writing a book and making it freely available, under a Creative Commons license, was largely inspired by the writings of Cory Doctorow and Lawrence Lessig. Their message of sharing resonated very strongly with me. I'm not interested in capitalizing on my knowledge of Modelica, I want to share it with others so they can experience the same joy I do in exploring this subject. Happily, this Kickstarter project was successfully funded on December 4th, 2012 and my odyssey to create an entire Creative Commons licensed book began. If circumstances had been different, this book would probably be dedicated to Cory Doctorow and Lawrence Lessig for inspiring me to take on this project.

But this book isn't dedicated to them. Instead, it is dedicated to **Aaron Swartz**. Aaron was without a doubt a wunderkind. At the age of 14, Aaron participated in the development of the RSS standard. He went on to start his own company, Infogami, that was eventually acquired by Reddit. Along the way, he worked on the Creative Commons License and fought against Internet censorship. He was both brilliant and successful yet he was also altruistic and cared deeply about technology and the future of mankind. He had countless qualities that I admire and aspire to. But on January 11th, 2013, Aaron Swartz took his own life.

Barely a month after the successful funding of my Kickstarter project, Aaron Swartz was dead at the age of 26. It is astounding what Aaron managed to accomplish in his young life and it makes his death all the more tragic to think of all the good he could have done with the rest of his life. Aaron, despite being almost 20 years younger than me, had helped blaze the trail for me, this book project and, ultimately, those reading this book through his work with RSS, Creative Commons, the Internet, and so on. It was heartbreaking that just as this project started, his life ended. I resolved then to dedicate this book to him.

Because his life has been chronicled⁴ by better writers who actually knew him⁵, I won't go into detail about Aaron's life except to say that as a business owner, a researcher and a U. S. citizen, I truly admire his work and I am thoroughly outraged by the way he was bullied by federal prosecutors (and Carmen Ortiz⁶ in particular) who were determined to make an example of him.

This book is dedicated to Aaron Swartz in an attempt to keep his memory and, more importantly, the things he stood for alive. Over a year has past since he died but I haven't forgotten Aaron Swartz and I hope this dedication reminds people what he stood for. We need to learn from this tragedy and make sure our governments understand that people like Aaron Swartz, while perhaps passionate and overzealous, don't deserve to be prosecuted and treated like criminals. They deserve guidance and coaching on ways to channel their abundant energies and talents to help society.

“Aaron dead. World wanderers, we have lost a wise elder. Hackers for right, we are one down. Parents all, we have lost a child. Let us weep.”

—Tim Berners-Lee⁷



⁴ <http://boingboing.net/2013/01/12/rip-aaron-swartz.html>

⁵ http://www.huffingtonpost.com/lawrence-lessig/aaron-swartz-suicide_b_2467079.html

⁶ <https://petitions.whitehouse.gov/petition/remove-united-states-district-attorney-carmen-ortiz-office-overreach-case-aaron-swartz/RQNrG1Ck>

⁷ https://twitter.com/timberners_lee/status/290140454211698689

DESCRIBING BEHAVIOR

1.1 Basic Equations

As mentioned in the *Preface* (page I), our exploration of Modelica starts with understanding how to represent basic behavior. In this chapter, our focus will be on demonstrating how to write basic equations.

1.1.1 Examples

Simple First Order System

Let us consider an extremely simple differential equation:

$$\dot{x} = (1 - x)$$

Looking at this equation, we see there is only one variable, x . This equation can be represented in Modelica as follows:

```
model FirstOrder
  Real x;
equation
  der(x) = 1-x;
end FirstOrder;
```

This code starts with the keyword `model` which is used to indicate the start of the model definition. The `model` keyword is followed by the model name, `FirstOrder`. This, in turn, is followed by a declaration of all the variables we are interested in.

Since the variable x in our equation is clearly meant to be a continuous real valued variable, its declaration in Modelica takes the form `Real x;`. The `Real` type is just one of the types we can use (a more complete description of the various possibilities will be discussed in the upcoming section on *Variables* (page 28)).

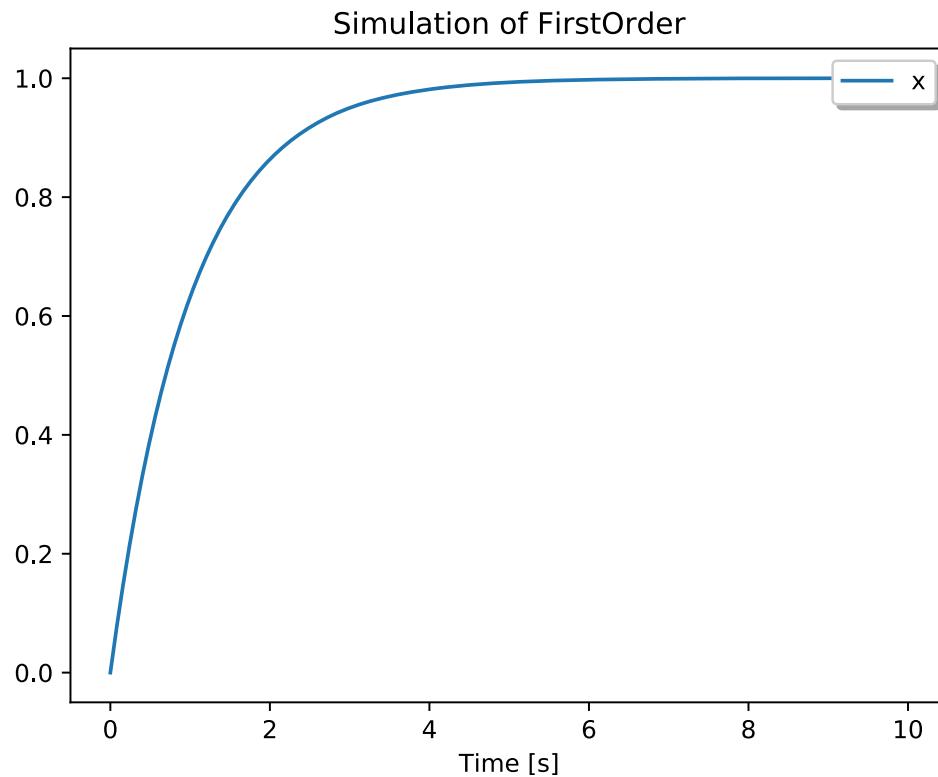
Once all the variables have been declared, we can begin including the equations that describe the behavior of our model. In this case, we can use the `der` operator to represent the time derivative of x . Thus,

```
der(x) = (1-x)
```

is equivalent to:

$$\dot{x} = (1 - x)$$

Unlike most programming languages, we don't approach code like this as a "program" that can be interpreted as a set of instructions to be executed one after the other. Instead, we use a Modelica compiler to transform this model into something that we can simulate. This simulation step essentially amounts to solving (usually numerically) the equation and providing a solution trajectory like this:



This gives you the first initial hint at one of the compelling aspects about using a modeling language to describe mathematical behavior. We didn't need to describe how to solve this differential equation. The focus is entirely on behavior. As we work our way through more complex examples, we will see that **much of the tedious work involving the solution process is handled automatically by the Modelica compiler.**

Adding Some Documentation

Now that we've solved this simple mathematical equation, let's turn our attention briefly to how we can make the model a bit more readable. Consider the following model:

```
model FirstOrderDocumented "A simple first order differential equation"
  Real x "State variable";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
end FirstOrderDocumented;
```

Note the quoted text in this model.

It is important to understand that the quoted text blocks, called “strings” in computer science, are **not** comments. They are “descriptive strings” and, unlike comments, they cannot be added in arbitrary places. Instead, they can only be inserted in specific places to provide additional documentation about the elements of the model they are associated with.

For example, the first string “A simple first order differential equation” is used to describe what this is a model of. Note how it follows the name of the model. If you wish to include such documentation about the model, it must appear immediately after the model name as shown.

As we will see later, this kind of documentation can be used by tools in many ways. For example,

when searching for models, a tool may use these descriptive strings when identifying matches. This text may also be associated with a graphical representation of the models. And, of course, this kind of documentation is very useful for anybody reading the model.

As this example demonstrates, there are other places that the descriptive text can appear. For example, it may be included after the declaration of a variable or equation to document them.

Initialization

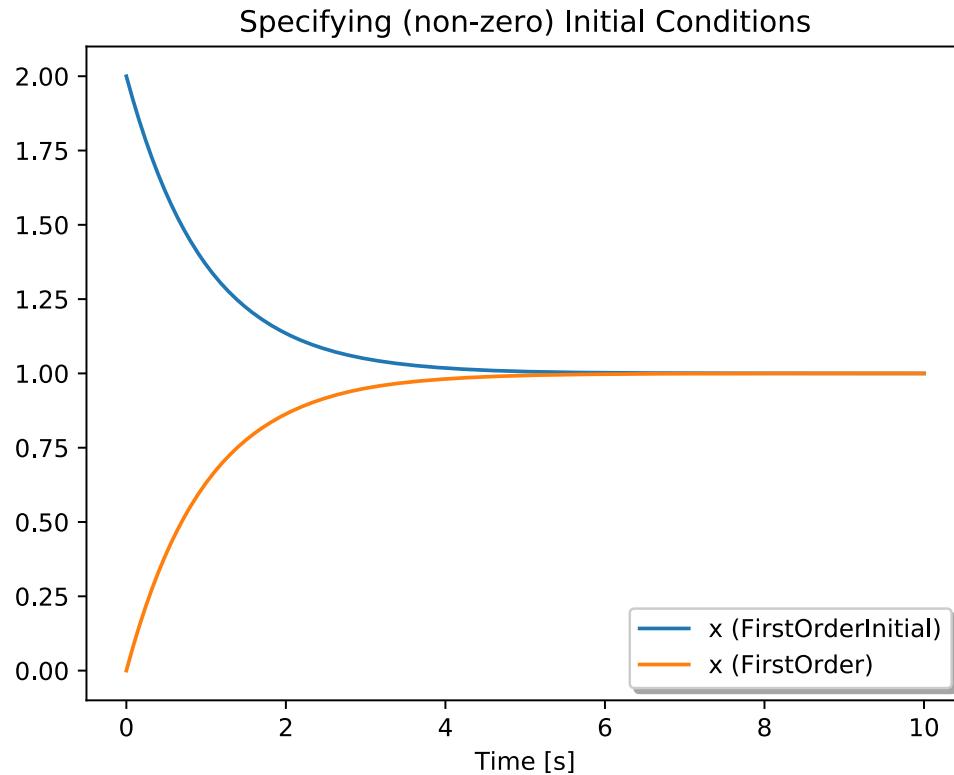
As we have seen already, Modelica allows us to describe model behavior in terms of differential equations. But the initial conditions we choose are just as important as the equations.

For this reason, Modelica also provides constructs for describing the initialization of our system of equations. For example, if we wanted the initial value of x in our model to be 2, we could add an `initial equation` section to our model as follows:

```
model FirstOrderInitial "First order equation with initial value"
  Real x "State variable";
initial equation
  x = 2 "Used before simulation to compute initial values";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
end FirstOrderInitial;
```

Note that the only difference between this model and the previous one, presented in the section on *Adding Some Documentation* (page 2), is the addition of the `initial equation` section which contains the equation $x = 2$. In the previous example, the initial value of x at the start of the simulation was unspecified. Generally speaking, this means that the initial value for x will be the value of its `start` attribute (which is zero by default). However, because each tool uses their own specific algorithms to formulate the final system of equations, it is always best to state initial conditions explicitly, as we have done here. By adding this equation to the `initial equation` section, we are explicitly specifying the initial condition for x .

As a result, the solution trajectory is quite different as we can see in the following figure:



The model `FirstOrderInitial` shows a typical way of initializing a system by providing explicit initial values for the states of the system. In fact, a system of differential equations is incomplete without a specification for how the initial conditions are determined. Our `FirstOrderInitial` model would be represented mathematically as:

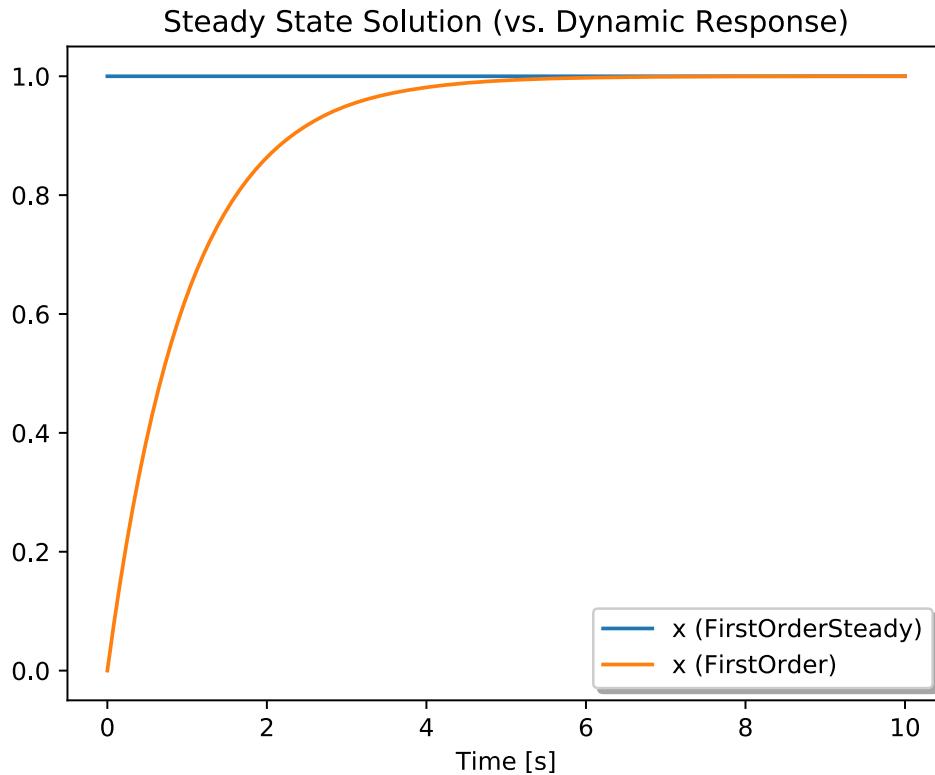
$$\dot{x} = (1 - x); \quad x(0) = 2$$

However, there are many cases where you want a more sophisticated type of initialization. An `initial equation` section can contain more than just explicit equations for the initial values of the state variables.

For example, we might want our initial conditions to be such that the derivative of x was zero at the start of the simulation. In this case, only a bit of trivial algebra is required to realize that this could be accomplished by specifying an initial condition of $x(0) = 1$. However, for more complex systems, it is far from trivial to determine the initial state values that would satisfy such a requirement. In those cases, it is possible to express the constraint that $\dot{x}(0) = 0$ directly in Modelica as follows:

```
model FirstOrderSteady
  "First order equation with steady state initial condition"
  Real x "State variable";
initial equation
  der(x) = 0 "Initialize the system in steady state";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
end FirstOrderSteady;
```

Simulating this system gives the following solution:



As we see from these results, the initial derivative of x is zero at the start of the simulation and remains zero because there are no external influences acting on this system that would disrupt this equilibrium.

This provides a glimpse of the initialization capabilities in Modelica. More complete coverage of the initialization topic can be found in the [Initialization](#) (page 35) section later in this chapter.

Experimental Conditions

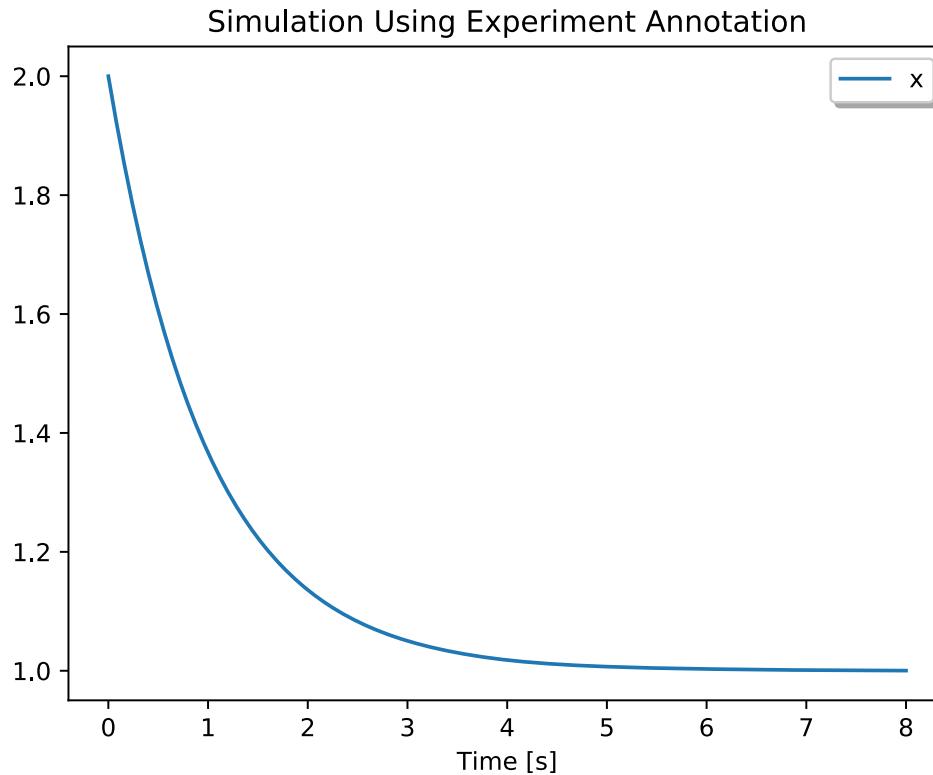
When building a model, the model developer might wish to associate specific experimental conditions with the model. This can be done using something called an **annotation**. An annotation includes information that is not directly related to the behavior of the model.

For example, experimental conditions describe information like the start time of the simulation, the stop time, solution tolerance and so on. This is not information about the behavior of the model itself, but rather information about how to approach simulating that behavior. Experimental conditions are stored in a model using a specific annotation called the **experiment** annotation.

There are four pieces of information that can be specified in an experiment annotation. All of them are optional. The following is a model of our first order system that includes an experiment annotation:

```
model FirstOrderExperiment "Defining experimental conditions"
  Real x "State variable";
initial equation
  x = 2 "Used before simulation to compute initial values";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
  annotation(experiment(StartTime=0,StopTime=8));
end FirstOrderExperiment;
```

The following trajectory was simulated using these experimental conditions:



The trajectory terminates at 8 seconds because the simulator used the `experiment` annotation to determine how long to run the simulation.

Annotation Support

The `experiment` annotation is widely supported. But it is important to keep in mind that, in general, a tool is free to ignore any or all annotations.

Getting Physical

Although the previous section got us started with representing mathematical behavior, it doesn't convey any connection to *physical* behavior. In this section, we'll explore how to build models that represent the modeling of physical behavior. Along the way, we will highlight some of the language features we can leverage that will not only tie these models to physical and engineering domains, but, as we shall see, they can even help us avoid mistakes.

Let's start with the following example:

```
model NewtonCooling "An example of Newton's law of cooling"
  parameter Real T_inf "Ambient temperature";
  parameter Real T0 "Initial temperature";
  parameter Real h "Convective cooling coefficient";
  parameter Real A "Surface area";
  parameter Real m "Mass of thermal capacitance";
  parameter Real c_p "Specific heat";
  Real T "Temperature";
initial equation
```

```

T = T0 "Specify initial value for T";
equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCooling;

```

As we saw in the examples in our discussion of [Simple First Order System](#) (page 1), the previous example consists of a `model` definition that includes variables and equations.

However, this time we see the word `parameter` for the first time. Generally speaking, the `parameter` keyword is used to indicate variables whose value is known *a priori* (*i.e.*, prior to the simulation). More precisely, `parameter` is a keyword that specifies the *variability* of a variable. This will be discussed more thoroughly in the section on [Variability](#) (page 28). But for now, we can think of a `parameter` as a variable whose value we must provide.

Looking at our `NewtonCooling` example, we see there are five parameters: `T_inf`, `T0`, `h`, `A`, `m` and `c_p`. We don't need to bother explaining what these variables are because the model itself includes a descriptive string for each one. At the moment, there are no values for these parameters, but we will return to that topic shortly. As with all the variables we have seen so far, these are all of type `Real`.

Let's examine the rest of this model. The next variable is `T` (also a `Real`). Since this variable doesn't have the `parameter` qualifier, its value is determined by the equations in the model.

Next we see the two `equation` sections. The first is an `initial equation` section which specifies how the variable `T` should be initialized. It should be pretty clear that the initial value for `T` is going to be whatever value was given (by us) for the parameter `T0`.

The other equation is the differential equation that governs the behavior of `T`. Mathematically, we could express this equation as:

$$mc_p\dot{T} = hA(T_{\infty} - T)$$

but in Modelica, we write it as:

```
m*c_p*der(T) = h*A*(T_inf-T)
```

Note that this is really no different from the equation we saw in our `FirstOrder` model from the [Simple First Order System](#) (page 1) example.

One thing worth noting is that the equation in our `NewtonCooling` example contains an `expression` on the left hand side. In Modelica, it is **not** necessary for each equation to be an explicit equation for a single variable. An equation can contain arbitrary expressions on either side of the equals sign. It is the compiler's job to determine how to use these equations to solve for the variables contained in the equations.

Another thing that distinguishes our `NewtonCooling` example from the `FirstOrder` model is that we can independently adjust the different parameter values. Furthermore, these parameter values are tied to physical, measurable properties of the materials or environmental conditions. In other words, this version is slightly more physical than the simple mathematical relationship used in the `FirstOrder` model because it is related to physical properties.

Now, we can't really run the `NewtonCooling` model as is because it lacks *values* for the six parameters. In order to create a model that is ready to be simulated, we need to provide those values, *e.g.*,

```

model NewtonCoolingWithDefaults "Cooling example with default parameter values"
  parameter Real T_inf=25 "Ambient temperature";
  parameter Real T0=90 "Initial temperature";
  parameter Real h=0.7 "Convective cooling coefficient";
  parameter Real A=1.0 "Surface area";
  parameter Real m=0.1 "Mass of thermal capacitance";
  parameter Real c_p=1.2 "Specific heat";
  Real T "Temperature";
initial equation
  T = T0 "Specify initial value for T";

```

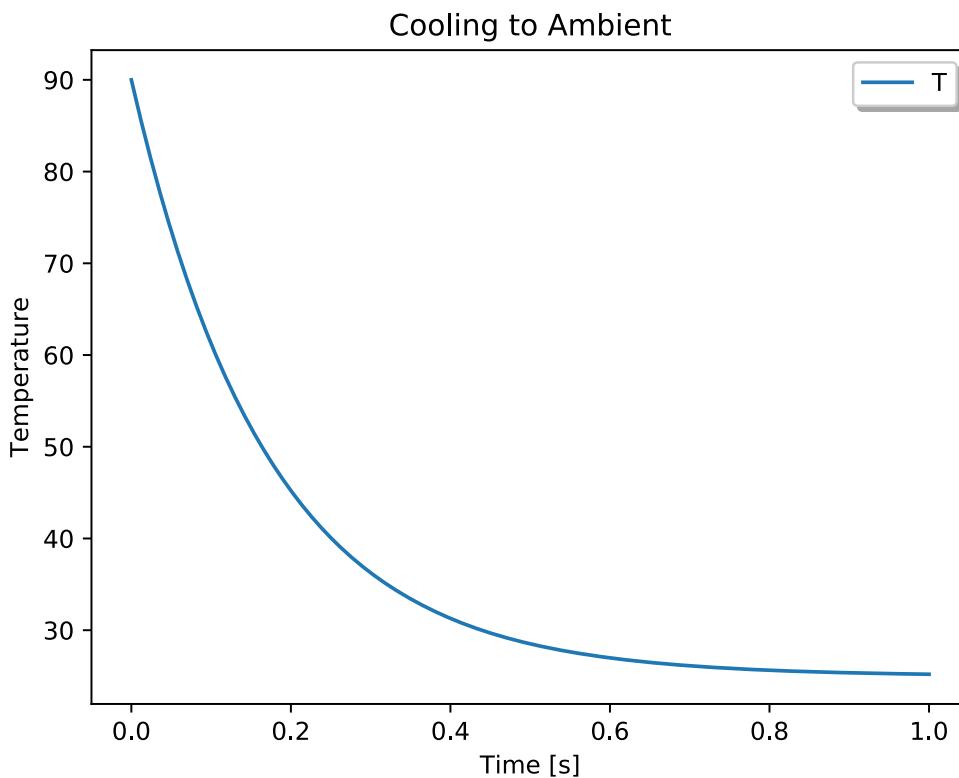
```

equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingWithDefaults;

```

The only real difference here is that each of the `parameter` variables now has a value specified. One way to think about the `NewtonCooling` model is that we could not simulate it because it had 7 variables (total) and only one equation (see the section on [Initialization](#) (page 35) for an explanation of why the `initial` equation doesn't really count). However, the `NewtonCoolingWithDefaults` model has, conceptually speaking, 7 equations (6 of them coming from specifying the values of the `parameter` variables + one in the equation section) and 7 unknowns.

If we simulate the `NewtonCoolingWithDefaults` model, we get the following solution for `T`.



Physical Units

As mentioned already in this section, these examples are a bit more physical because they include individual physical parameters that correspond to individual properties of our real world system. However, we are still missing something. Although these variables represent physical quantities like temperature, mass, *etc.*, we haven't explicitly given them any physical types.

As you may have already guessed, the variable `T` is a temperature. This is made clear in the descriptive text associated with the variable. Furthermore, it doesn't take a very deep analysis of our previous model to determine that `T0` and `T_inf` must also be temperatures.

But what about the other variables like `h` or `A`? What do they represent? Even more important, are the equations **physically consistent**? By physically consistent, we mean that both sides of the equations have the same physical units (*e.g.*, temperature, mass, power).

We could convey the physical units of the different variables more rigorously by actually including them in the variable declarations, like so:

```
model NewtonCoolingWithUnits "Cooling example with physical units"
  parameter Real T_inf(unit="K")=298.15 "Ambient temperature";
  parameter Real T0(unit="K")=363.15 "Initial temperature";
  parameter Real h(unit="W/(m2.K)")=0.7 "Convective cooling coefficient";
  parameter Real A(unit="m2")=1.0 "Surface area";
  parameter Real m(unit="kg")=0.1 "Mass of thermal capacitance";
  parameter Real c_p(unit="J/(K.kg)")=1.2 "Specific heat";

  Real T(unit="K") "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingWithUnits;
```

Note that each of the variable declarations now includes the text (`unit="..."`) to associate a physical unit with the variable. What this additional text does is specify a value for the `unit` attribute associated with the variable. Attributes are special properties that each variable has. The set of attributes a variable can have depends on the type of the variable (this is discussed in more detail in the upcoming section on *Variables* (page 28)).

At first glance, it may not seem obvious why specifying the `unit` attribute (*e.g.*, `(unit="K")`) is any better than simply adding "Temperature" to the descriptive string following the variable. In fact, one might even argue it is worse because "Temperature" is more descriptive than just a single letter like "K".

However, setting the `unit` attribute is actually a more formal approach for two reasons. The first reason is that the Modelica specification defines relationships for all the standard SI unit attributes (*e.g.*, K, kg, m). This includes complex unit types that can be composed of other base units (*e.g.*, N).

The other reason is that the Modelica specification also defines rules for how to compute the units of complex mathematical expressions. In this way, the Modelica specification defines everything that is necessary to **unit check** Modelica models for errors or physical inconsistencies. This is a big win for model developers because adding units not only makes the models clearer, it provides better diagnostics in the case of errors.

Physical Types

But truth be told, there is one drawback of the code for our `NewtonCoolingWithUnits` example and that is that we have to repeat the `unit` attribute specification for every variable. Furthermore, as mentioned previously, K isn't nearly as descriptive as "Temperature".

Fortunately, we have a simple solution to both problems because Modelica allows us to define *derived types*. So far, all the variables we have declared have been of type `Real`. The problem with `Real` is that it could be anything (*e.g.*, a voltage, a current, a temperature). What we'd like to do is narrow things down a bit. This is where derived types come in. To see how to define derived types and then use them in declarations, consider the following example:

```
model NewtonCoolingWithTypes "Cooling example with physical types"
  // Types
  type Temperature=Real(unit="K", min=0);
  type ConvectionCoefficient=Real(unit="W/(m2.K)", min=0);
  type Area=Real(unit="m2", min=0);
  type Mass=Real(unit="kg", min=0);
  type SpecificHeat=Real(unit="J/(K.kg)", min=0);

  // Parameters
  parameter Temperature T_inf=298.15 "Ambient temperature";
  parameter Temperature T0=363.15 "Initial temperature";
```

```

parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
parameter Area A=1.0 "Surface area";
parameter Mass m=0.1 "Mass of thermal capacitance";
parameter SpecificHeat c_p=1.2 "Specific heat";

// Variables
Temperature T "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingWithTypes;

```

You can read the definition type `Temperature=Real(unit="K", min=0)`; as “Let us define a new type, `Temperature`, that is a specialization of the built-in type `Real` with physical units of Kelvin (K) and a minimum possible value of 0”.

From this example, we can see that once we define a physical type like `Temperature`, we can use it to declare multiple variables (*e.g.*, `T`, `T_inf` and `T0`) without having to specify the `unit` or `min` attribute for each variable. Also, we get to use the familiar name `Temperature` instead of the SI unit, K. You might be wondering what other attributes are available when creating derived types. For further discussion, see the section on *Built-In Types* (page 29).

At this point, you might find the idea of defining `Temperature`, `ConvectionCoefficient`, `SpecificHeat` and `Mass` in every model extremely tedious. It would be, if it were truly necessary. But don’t worry, there is an easy solution to this as you will see in a later section where we discuss *Importing Physical Types* (page 154).

An Electrical Example

Let us return now to an engineering context. For readers who are more familiar with electrical systems, consider the following circuit:

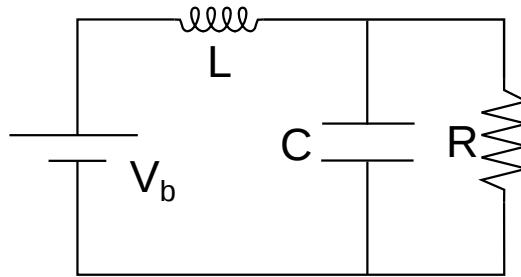


Fig. 1.1: Low-Pass RLC Filter

Suppose we want to solve for: V , i_L , i_R and i_C . To solve for each of the currents i_L , i_R and i_C , we can use the equations associated with inductors, resistors and capacitors, respectively:

$$V = i_R R$$

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

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

where V_b is the battery voltage.

Since we have only 3 equations, but 4 variables, we need one additional equation. That additional equation is going to be Kirchoff's current law:

$$i_L = i_R + i_C$$

Now that we have determined the equations and variables for this problem, we will create a basic model (including physical types) by translating the equations directly into Modelica. But in a later section on [Electrical Components](#) (page 192) we will return to this same circuit and demonstrate how to create models by dragging, dropping and connecting models that really look like the circuit components in our [Low-Pass RLC Filter](#) (page 10).

But for now, we will build a model composed simply of variables and equations. Such a model could be written as follows:

```
model RLC1 "A resistor-inductor-capacitor circuit model"
  type Voltage=Real(unit="V");
  type Current=Real(unit="A");
  type Resistance=Real(unit="Ohm");
  type Capacitance=Real(unit="F");
  type Inductance=Real(unit="H");
  parameter Voltage Vb=24 "Battery voltage";
  parameter Inductance L = 1;
  parameter Resistance R = 100;
  parameter Capacitance C = 1e-3;
  Voltage V;
  Current i_L;
  Current i_R;
  Current i_C;
equation
  V = i_R*R;
  C*der(V) = i_C;
  L*der(i_L) = (Vb-V);
  i_L=i_R+i_C;
end RLC1;
```

Let's go through this example bit by bit and reinforce the meaning of the various statements. Let's start at the top:

```
model RLC1 "A resistor-inductor-capacitor circuit model"
```

Here we see that the name of the model is RLC1. Furthermore, a description of this model has been included, namely "A resistor-inductor-capacitor circuit model". Next, we introduce a few physical types that we will need:

```
type Voltage=Real(unit="V");
type Current=Real(unit="A");
type Resistance=Real(unit="Ohm");
type Capacitance=Real(unit="F");
type Inductance=Real(unit="H");
```

Each of these lines introduces a physical type that specializes the built-in `Real` type by associating it with a particular physical unit. Then, we declare all of the `parameter` variables in our problem:

```
parameter Voltage Vb=24 "Battery voltage";
parameter Inductance L = 1;
parameter Resistance R = 100;
parameter Capacitance C = 1e-3;
```

These `parameter` variables represent various physical characteristics (in this case, voltage, inductance, resistance and capacitance, respectively). The last variables we need to define are the ones we wish to solve for, *i.e.*,

```
Voltage V;
Current i_L;
Current i_R;
Current i_C;
```

Now that all the variables have been declared, we add an `equation` section to the model that specifies the equations to use when generating solutions for this model:

```
equation
  V = i_R*R;
  C*der(V) = i_C;
  L*der(i_L) = (Vb-V);
  i_L=i_R+i_C;
```

Finally, we close the model by creating an `end` statement that includes the `model` name (*i.e.*, RLC1 in this case):

```
end RLC1;
```

One thing that distinguishes this example from the previous examples is the fact that it contains more equations. As with the `NewtonCooling` example, we have some equations with expressions on both the left and right hand sides. We also have a mix of differential equation (ones that include the derivative of a variable) and others that are simply algebraic equations.

This further emphasizes the point that in Modelica it is not necessary to put the system of equations into the so-called “explicit state-space form” required in some modeling environments. We could, of course, rearrange the equations into a more explicit form like this:

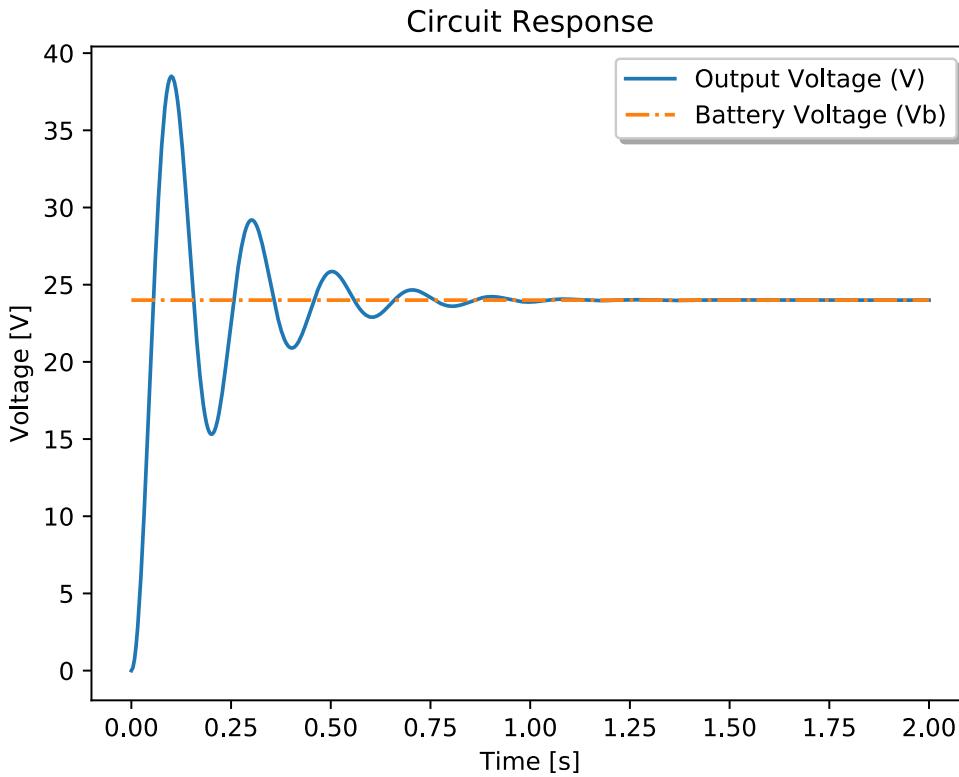
```
der(V) = i_C/C;
der(i_L) = (Vb-V)/L;
i_R = i_L-i_C;
V = i_R*R;
```

But the important point is that with Modelica, we do not need to perform such manipulations. Instead, we are free to write the equations in whatever form we chose.

Ultimately, these equations will probably need to be manipulated into a form like explicit state-space form. But if such manipulations are necessary, it will be the responsibility of the Modelica compiler, not the model developer, to perform these manipulations. This eliminates the need for the model developer to deal with this tedious, time consuming and error prone task.

The ability to keep equations in their “textbook form” is important because, as we will show in later sections, we eventually want to get to the point where these equations are “captured” in individual components models. In those cases, we won’t know (when we create the component model) exactly what variable each equation will be used to solve for. Making such manipulations the responsibility of the Modelica compiler not only makes the model development faster and easier, but it dramatically improves the **reusability** of the models.

The following figure shows the dynamic response of the RLC1 model:

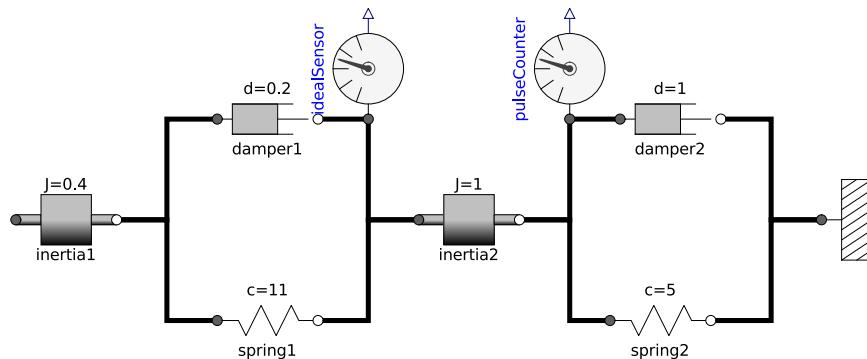


Expanding on these electrical examples

As mentioned in the *Preface* (page I), the structure of this book allows us to explore a more hypermedia based approach in which readers are encouraged to process the material that is most aligned with their goals and interests. The next chapter will present a model whose equations are derived from a mechanical system. If you would prefer instead to see this electrical example extended to include more complex behavior, you may want to skip ahead to the *Switched RLC Circuit* (page 61) example.

A Mechanical Example

If you are more familiar with mechanical systems, this example might help reinforce some of the concepts we've covered so far. The system we wish to model is the one shown in the following figure:



It is worth pointing out how much easier it is to convey the intention of a model by presenting it in schematic form. Assuming appropriate graphical representations are used, experts can very quickly understand the composition of the system and develop an intuition about how it is likely to behave. While we are currently focusing on equations and variables, we will eventually work our way up to an approach (in the upcoming section of the book on *Components* (page 183)) where **models will be built in this schematic form from the beginning**.

For now, however, we will focus on how to express the equations associated with this simple mechanical system. Each inertia has a rotational position, φ , and a rotational speed, ω where $\omega = \dot{\varphi}$. For each inertia, the balance of angular momentum for the inertia can be expressed as:

$$J\dot{\omega} = \sum_i \tau_i$$

In other words, the sum of the torques, τ , applied to the inertia should be equal to the product of the moment of inertia, J , and the angular acceleration, $\dot{\omega}$.

At this point, all we are missing are the torque values, τ_i . From the previous figure, we can see that there are two springs and two dampers. For the springs, we can use Hooke's law to express the relationship between torque and angular displacement as follows:

$$\tau = c\Delta\varphi$$

For each damper, we express the relationship between torque and relative angular velocity as:

$$\tau = d\Delta\dot{\varphi}$$

If we pull together all of these relations, we get the following system of equations:

$$\begin{aligned}\omega_1 &= \dot{\varphi}_1 \\ J_1\dot{\omega}_1 &= c_1(\varphi_2 - \varphi_1) + d_1 \frac{d(\varphi_2 - \varphi_1)}{dt} \\ \omega_2 &= \dot{\varphi}_2 \\ J_2\dot{\omega}_2 &= c_1(\varphi_1 - \varphi_2) + d_1 \frac{d(\varphi_1 - \varphi_2)}{dt} - c_2\varphi_2 - d_2\dot{\varphi}_2\end{aligned}$$

Let's assume our system has the following initial conditions as well:

$$\begin{aligned}\varphi_1 &= 0 \\ \omega_1 &= 0 \\ \varphi_2 &= 1 \\ \omega_2 &= 0\end{aligned}$$

These initial conditions essentially mean that the system starts in a state where neither inertia is actually moving (*i.e.*, $\omega = 0$), but there is a non-zero deflection across both springs.

Pulling all of these variables and equations together, we can express this problem in Modelica as follows:

```
model SecondOrderSystem "A second order rotational system"
  type Angle=Real(unit="rad");
  type AngularVelocity=Real(unit="rad/s");
  type Inertia=Real(unit="kg.m2");
  type Stiffness=Real(unit="N.m/rad");
  type Damping=Real(unit="N.m.s/rad");
  parameter Inertia J1=0.4 "Moment of inertia for inertia 1";
  parameter Inertia J2=1.0 "Moment of inertia for inertia 2";
  parameter Stiffness c1=11 "Spring constant for spring 1";
  parameter Stiffness c2=5 "Spring constant for spring 2";
  parameter Damping d1=0.2 "Damping for damper 1";
  parameter Damping d2=1.0 "Damping for damper 2";
  Angle phi1 "Angle for inertia 1";
```

```

Angle phi2 "Angle for inertia 2";
AngularVelocity omega1 "Velocity of inertia 1";
AngularVelocity omega2 "Velocity of inertia 2";
initial equation
  phi1 = 0;
  phi2 = 1;
  omega1 = 0;
  omega2 = 0;
equation
  // Equations for inertia 1
  omega1 = der(phi1);
  J1*der(omega1) = c1*(phi2-phi1)+d1*der(phi2-phi1);
  // Equations for inertia 2
  omega2 = der(phi2);
  J2*der(omega2) = c2*(phi1-phi2)+d2*der(phi1-phi2);
end SecondOrderSystem;

```

As we did with the low-pass filter example, RLC1, let's walk through this line by line.

As usual, we start with the name of the model:

```
model SecondOrderSystem "A second order rotational system"
```

Next, we introduce physical types for a rotational mechanical system, namely:

```

type Angle=Real(unit="rad");
type AngularVelocity=Real(unit="rad/s");
type Inertia=Real(unit="kg.m2");
type Stiffness=Real(unit="N.m/rad");
type Damping=Real(unit="N.m.s/rad");

```

Then we define the various parameters used to represent the different physical characteristics of our system:

```

parameter Inertia J1=0.4 "Moment of inertia for inertia 1";
parameter Inertia J2=1.0 "Moment of inertia for inertia 2";
parameter Stiffness c1=11 "Spring constant for spring 1";
parameter Stiffness c2=5 "Spring constant for spring 2";
parameter Damping d1=0.2 "Damping for damper 1";
parameter Damping d2=1.0 "Damping for damper 2";

```

For this system, there are four non-parameter variables. These are defined as follows:

```

Angle phi1 "Angle for inertia 1";
Angle phi2 "Angle for inertia 2";
AngularVelocity omega1 "Velocity of inertia 1";
AngularVelocity omega2 "Velocity of inertia 2";

```

The initial conditions (which we will revisit shortly) are then defined with:

```

initial equation
  phi1 = 0;
  phi2 = 1;
  omega1 = 0;
  omega2 = 0;

```

Then come the equations describing the dynamic response of our system:

```

equation
  // Equations for inertia 1
  omega1 = der(phi1);
  J1*der(omega1) = c1*(phi2-phi1)+d1*der(phi2-phi1);
  // Equations for inertia 2

```

```
omega2 = der(phi2);
J2*der(omega2) = c1*(phi1-phi2)+d1*der(phi1-phi2)-c2*phi2-d2*der(phi2);
```

And finally, we have the closing of our model definition.

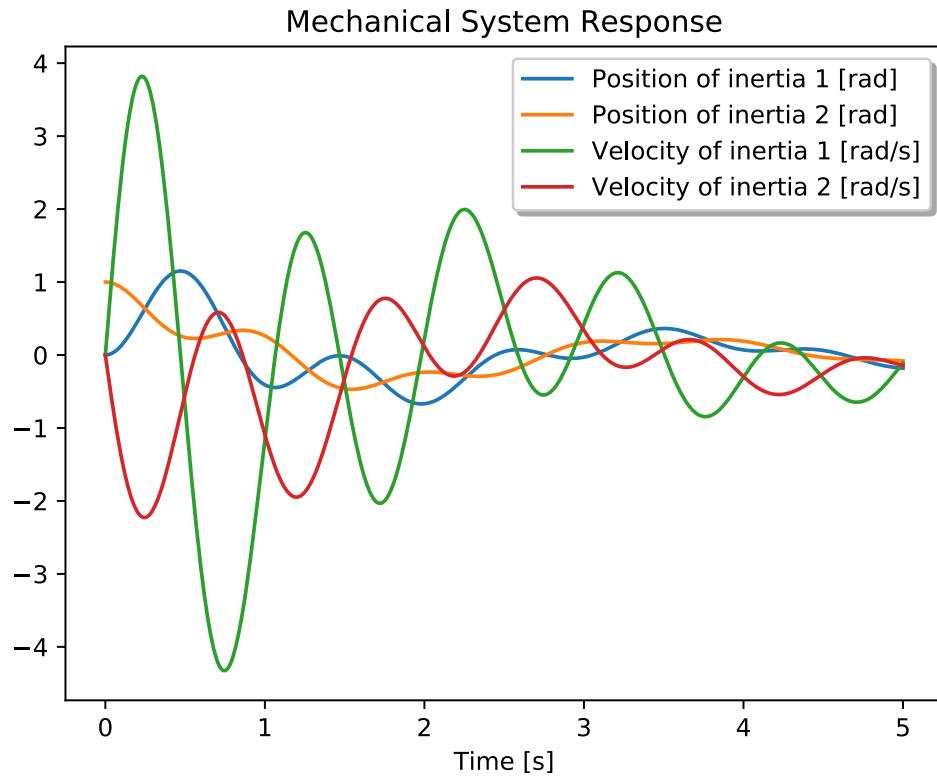
```
end SecondOrderSystem;
```

The only drawback of this model is that all of our initial conditions have been “hard-coded” into the model. This means that we will be unable to specify any alternative set of initial conditions for this model. We can overcome this issue, as we did with our *Newton cooling examples* (page 6), by defining **parameter** variables to represent the initial conditions as follows:

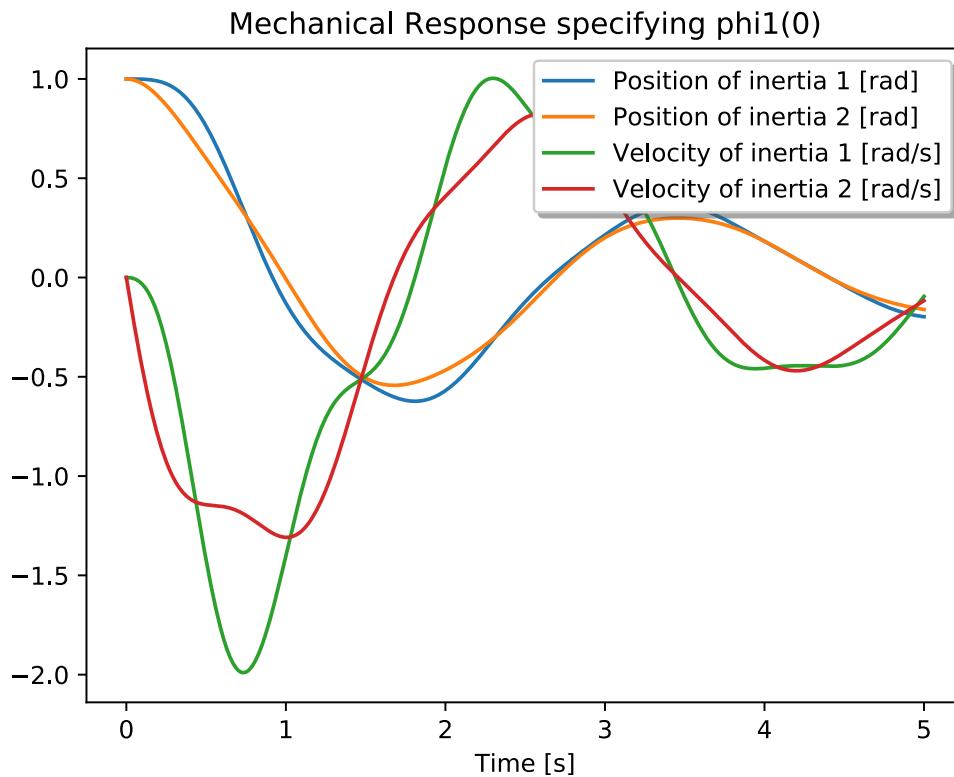
```
model SecondOrderSystemInitParams
  "A second order rotational system with initialization parameters"
  type Angle=Real(unit="rad");
  type AngularVelocity=Real(unit="rad/s");
  type Inertia=Real(unit="kg.m2");
  type Stiffness=Real(unit="N.m/rad");
  type Damping=Real(unit="N.m.s/rad");
  parameter Angle phi1_init = 0;
  parameter Angle phi2_init = 1;
  parameter AngularVelocity omega1_init = 0;
  parameter AngularVelocity omega2_init = 0;
  parameter Inertia J1=0.4 "Moment of inertia for inertia 1";
  parameter Inertia J2=1.0 "Moment of inertia for inertia 2";
  parameter Stiffness c1=11 "Spring constant for spring 1";
  parameter Stiffness c2=5 "Spring constant for spring 2";
  parameter Damping d1=0.2 "Damping for damper 1";
  parameter Damping d2=1.0 "Damping for damper 2";
  Angle phi1 "Angle for inertia 1";
  Angle phi2 "Angle for inertia 2";
  AngularVelocity omega1 "Velocity of inertia 1";
  AngularVelocity omega2 "Velocity of inertia 2";
initial equation
  phi1 = phi1_init;
  phi2 = phi2_init;
  omega1 = omega1_init;
  omega2 = omega2_init;
equation
  omega1 = der(phi1);
  omega2 = der(phi2);
  J1*der(omega1) = c1*(phi2-phi1)+d1*der(phi2-phi1);
  J2*der(omega2) = c1*(phi1-phi2)+d1*der(phi1-phi2)-c2*phi2-d2*der(phi2);
end SecondOrderSystemInitParams;
```

In this way, the parameter values can be changed either in the simulation environment (where parameters are typically editable by the user) or, as we will see shortly, via so-called “modifications”.

You will see in this latest version of the model that the values for the newly introduced parameters are the same as the hard-coded values used earlier. As a result, the default initial conditions will be exactly the same as they were before. But now, we have the freedom to explore other initial conditions as well. For example, if we simulate the **SecondOrderSystemInitParams** model as is, we get the following solution for the angular positions and velocities:



However, if we modify the `phi1_init` parameter to be `1` at the start of our simulation, we get this solution instead:



Expanding on this mechanical example

If you would like to see this example further developed, you may wish to jump to the set of examples involving rotational systems found in the section on [Speed Measurement](#) (page 63).

Otherwise, you can continue to the next set of examples which involve population dynamics.

Lotka-Volterra Systems

So far, we've seen thermal, electrical and mechanical examples. In effect, these have all been engineering examples. However, Modelica is not limited strictly to engineering disciplines. To reinforce this point, this section will present a common ecological system dynamics model based on the relationship between predator and prey species. The equations we will be using are the [\[Lotka\]](#) (page 357)-[\[Volterra\]](#) (page 357) equations.

Classic Lotka-Volterra

The classic Lotka-Volterra model⁸ involves two species. One species is the “prey” species. In this section, the population of the prey species will be represented by x . The other species is the “predator” species whose population will be represented by y .

There are three important effects in a Lotka-Volterra system. The first is reproduction of the “prey” species. It is assumed that reproduction is proportional to the population. If you are familiar with chemical reactions, this is conceptually the same as the [Law of Mass Action](#)⁹ [\[Guldberg\]](#) (page 357). If

⁸ http://en.wikipedia.org/wiki/Lotka-Volterra_equation

⁹ http://en.wikipedia.org/wiki/Law_of_mass_action

you aren't familiar with the Law of Mass Action, just consider that the more potential mates are present in the environment, the more likely reproduction is to occur. We can represent this mathematically as:

$$\dot{x}_r = \alpha x$$

where x is the prey population, \dot{x}_r is the change in prey population *due to reproduction* and α is the proportionality constant capturing the likelihood of successful reproduction.

The next effect to consider is starvation of the predator species. If there aren't enough "prey" around to eat, the predator species will die off. When modeling starvation, it is important to consider the effect of competition. We again have a proportionality relationship, but this time it works in reverse because, unlike with prey reproduction, the more predators around the more likely starvation is. This is expressed mathematically in much the same way as reproduction:

$$\dot{y}_s = -\gamma y$$

where y is the predator population, \dot{y}_s is the change in predator population *due to starvation* and γ is the proportionality constant capturing the likelihood of starvation.

Finally, the last effect we need to consider is "predation", *i.e.*, the consumption of the prey species by the predator species. Without predators, the prey species would (at least mathematically) grow exponentially. So predation is the effect that keeps the prey species population in check. Similarly, without any prey, the predator species would simply die off. So predation is what balances out this effect and keeps the predator population from going to zero. Again, we have a proportionality relationship. But this time, it is actually a bilinear relationship that is, again, conceptually similar to the Law of Mass Action. This relationship is simply capturing, mathematically, the fact that the chance that a predator will find and consume some prey is proportional to both the population of the prey and the predators. Since this particular effect requires both species to be involved, this mathematical relationship has a slightly different structure than reproduction and starvation, *i.e.*,

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

where \dot{x}_p is the decline in the prey population *due to predation*, \dot{y}_p is the increase in the predator population *due to predation*, β is the proportionality constant representing the likelihood of prey consumption and δ is the proportionality constant representing the likelihood that the predator will have sufficient extra nutrition to support reproduction.

Taking the various effects into account, the overall change in each population can be represented by the following two equations:

$$\begin{aligned}\dot{x} &= \dot{x}_r + \dot{x}_p \\ \dot{y} &= \dot{y}_p + \dot{y}_s\end{aligned}$$

Using the previous relationships, we can expand each of the right hand side terms in these two equations into:

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

Using what we've learned in this chapter so far, translating these equations into Modelica should be pretty straightforward:

```
model ClassicModel "This is the typical equation-oriented model"
  parameter Real alpha=0.1 "Reproduction rate of prey";
  parameter Real beta=0.02 "Mortality rate of predator per prey";
  parameter Real gamma=0.4 "Mortality rate of predator";
  parameter Real delta=0.02 "Reproduction rate of predator per prey";
  parameter Real x0=10 "Start value of prey population";
  parameter Real y0=10 "Start value of predator population";
  Real x(start=x0) "Prey population";
  Real y(start=y0) "Predator population";
```

```

equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end ClassicModel;

```

At this point, there is only one thing we haven't discussed yet and that is the presence of the `start` attribute on `x` and `y`. As we saw in the `NewtonCoolingWithUnits` example in the previous section titled *Getting Physical* (page 6), variables have various attributes that we can specify (for a detailed discussion of available attributes, see the upcoming section on *Built-In Types* (page 29)). We previously discussed the `unit` attribute, but this is the first time we are seeing the `start` attribute.

The observant reader may have noticed the presence of the `x0` and `y0` parameter variables and the fact that they represent the initial populations. Based on previous examples, one might have expected these initial conditions to be captured in the model as follows:

```

model ClassicModelInitialEquations "This is the typical equation-oriented model"
  parameter Real alpha=0.1 "Reproduction rate of prey";
  parameter Real beta=0.02 "Mortality rate of predator per prey";
  parameter Real gamma=0.4 "Mortality rate of predator";
  parameter Real delta=0.02 "Reproduction rate of predator per prey";
  parameter Real x0=10 "Initial prey population";
  parameter Real y0=10 "Initial predator population";
  Real x(start=x0) "Prey population";
  Real y(start=y0) "Predator population";
initial equation
  x = x0;
  y = y0;
equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end ClassicModelInitialEquations;

```

However, for the `ClassicModel` example we took a small shortcut. As will be discussed shortly in the section on *Initialization* (page 35), we can specify initial conditions by specifying the value of the `start` attribute directly on the variable.

It is worth noting that this approach has both advantages and disadvantages. The advantage is one of flexibility. The `start` attribute is actually more of a hint than a binding relationship. If the Modelica compiler identifies a particular variable as a state (*i.e.*, a variable that requires an initial condition) **and** there are insufficient initial conditions already explicitly specified in the model via `initial equation` sections then it can substitute the `start` attribute as an initial condition for the variable it is associated with. In other words, you can think of the `start` attribute as a “fallback initial condition” if an initial condition is needed.

There are a couple of disadvantages to the `start` attribute that you need to watch out for. First, it is only a hint and tools may completely ignore it. Next, whether it will be ignored is also hard to predict since different tools may make different choices about which variables to treat as states.

One way to avoid both of these disadvantages is to use the `fixed` attribute (also discussed in the section on *Built-In Types* (page 29)). The `fixed` attribute can be used to tell the compiler that the `start` attribute **must** be used as an initial condition. In other words, an `initial equation` like this:

```

Real x;
initial equation
  x = 5;

```

is equivalent to the following declaration utilizing the `start` and `fixed` attributes:

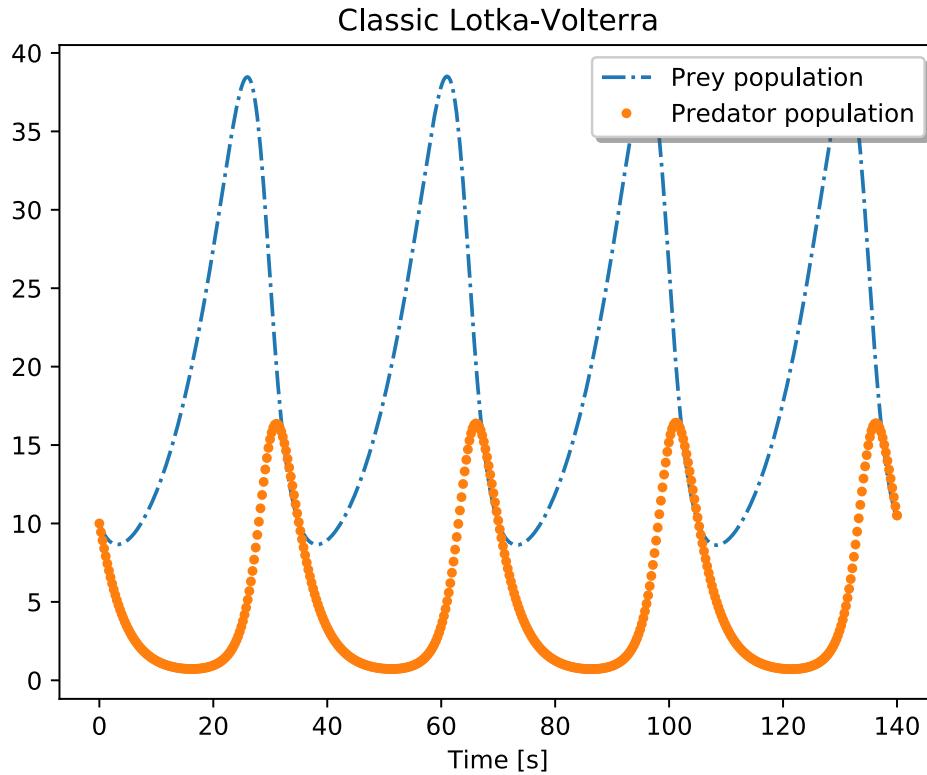
```
Real x(start=5, fixed=true);
```

Finally, one additional complication is that the `start` attribute is also “overloaded”. This means that it is actually used for two different things. If the variable in question is not a state, but is instead an

“iteration variable” (*i.e.*, a variable whose solution depends on a non-linear system of equations), then the `start` attribute may be used by a Modelica compiler as an initial guess (*i.e.*, the value used for the variable during the initial iteration of the non-linear solver).

Whether to specify a `start` attribute or not depends on how strictly you want a given initial condition to be enforced. Knowing that is something that takes experience working with the language and is beyond the scope of this chapter. However, it is worth at least pointing out that there are different options along with a basic explanation of the trade-offs.

Using either initialization method, the results for these models will be the same. The typical behavior for the Lotka-Volterra system can be seen in this plot:



Note the cyclical behavior of each population. Initially, there are more predators than can be supported by the existing food supply. Those predators that are present consume whatever prey they can find. Nevertheless, some starvation occurs and the predator population declines. The rate at which predators consume the prey species is so high during this period that the rate at which the prey species reproduces is not sufficient to make up for those lost to predation so the prey population declines as well.

At some point, the predator population gets so low that the rate of reproduction in the prey species is larger than the rate of prey consumption by the predators and the prey species begins to rebound. Because the predator species population takes longer to rebound, the prey species experiences growth that is, for the moment, virtually unchecked by predation. Eventually, the predator population begins to rebound due to the abundance of prey until the system returns to the original predator and prey populations **and the entire cycle then repeats itself *ad infinitum***.

The fact that the system returns again and again to the same initial conditions (ignoring numerical error, of course) is one of the most interesting things about the system. This is even more remarkable given the fact that the Lotka-Volterra system of equations is actually non-linear.

Steady State Initialization

Let's imagine that these extreme swings in species population had some undesirable ecological consequences. In such a case, it would be useful to understand what might reduce or even eliminate these fluctuations. A simple approach would be to keep the populations in a state of equilibrium. But how can we use these models to help us determine such a “quiescent” state?

The answer lies in the initial conditions. Instead of specifying an initial population for both the predator and prey species, we might instead choose to initialize the system with some other equations that somehow capture the fact that the system is in equilibrium (you may remember this trick from the `FirstOrderSteady` model [discussed previously](#) (page 3)). Fortunately, Modelica's approach to initialization is rich enough to allow us to specify this (and many other) useful types of initial conditions.

To ensure that our system starts in equilibrium, we simply need to define what equilibrium is. Mathematically speaking, the system is in equilibrium if the following two conditions are met:

$$\begin{aligned}\dot{x} &= 0 \\ \dot{y} &= 0\end{aligned}$$

To capture this in our Modelica model, all we need to do is use these equations in our `initial equation` section, like this:

```
model QuiescentModel "Find steady state solutions to LotkaVolterra equations"
  parameter Real alpha=0.1 "Reproduction rate of prey";
  parameter Real beta=0.02 "Mortality rate of predator per prey";
  parameter Real gamma=0.4 "Mortality rate of predator";
  parameter Real delta=0.02 "Reproduction rate of predator per prey";
  Real x "Prey population";
  Real y "Predator population";
initial equation
  der(x) = 0;
  der(y) = 0;
equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end QuiescentModel;
```

The main difference between this and our previous model is the presence of the highlighted initial equations. Looking at this model, you might wonder exactly what those initial equations mean. After all, what we need to solve for are `x` and `y`. But those variables don't even appear in our initial equations. So how are they solved for?

The answer lies in understanding that the functions $x(t)$ and $y(t)$ are solved for by integrating the differential equations starting from some initial equations. During the simulation, we see that x and \dot{x} are “coupled” by the following equations:

$$x(t) = \int_{t_0}^{t_f} \dot{x} \, dx + x(t_0)$$

(and, of course, a similar relationship exists between y and \dot{y})

However, during initialization of the system (*i.e.*, when solving for the initial conditions) this relationship doesn't hold. So there is no “coupling” between x and \dot{x} in that case (nor for y : and \dot{y}). In other words, knowing x or y doesn't give you any clue as to how to compute \dot{x} or \dot{y} . The net result is that for the initialization problem we can think of x , y , \dot{x} and \dot{y} as four independent variables.

Said another way, while simulating, we solve for x by integrating \dot{x} . So that integral equation is the equation used to solve for x . But during initialization, we cannot use that equation so we need an additional equation (for each integration that we would otherwise perform during simulation).

In any case, the bottom line is that during initialization we require four different equations to arrive at

a unique solution. In the case of our `QuiescentModel`, those four equations are:

$$\begin{aligned}\dot{x} &= 0 \\ \dot{x} &= x(\alpha - \beta y) \\ \dot{y} &= 0 \\ \dot{y} &= y(\delta x - \gamma)\end{aligned}$$

It is very important to understand that these equations **do not contradict each other**. Especially if you come from a programming background you might look at the first two equations and think “Well what is \dot{x} ? Is it zero or is it $x(\alpha - \beta y)$?” The answer is **both**. There is no reason that both equations cannot be true!

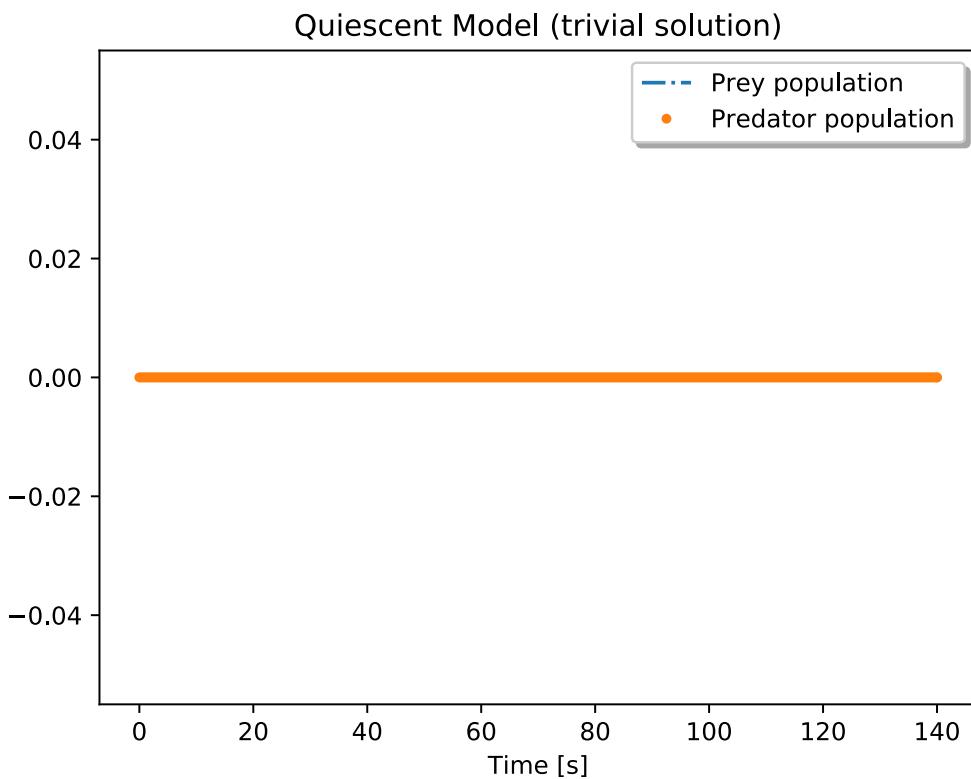
The essential thing to remember here is that these are **equations not assignment statements**. The following system of equations is mathematically identical and demonstrates more clearly how x and y could be solved:

$$\begin{aligned}\dot{x} &= 0 \\ \dot{y} &= 0 \\ x(\alpha - \beta y) &= \dot{x} \\ y(\delta x - \gamma) &= \dot{y}\end{aligned}$$

In this form, it is a bit easier to recognize how we could arrive at values of x and y . The first thing to note is that we cannot solve explicitly for x and y . In other words, we cannot rearrange these equations into the form $x = \dots$ without having x also appear on the right hand side. So we have to deal with the fact that **this is a simultaneous system of equations** involving both x and y .

But the situation is further complicated by the fact that this system is non-linear (which is precisely why we cannot use linear algebra to arrive at a set of explicit equations). In fact, if we study these equations carefully we can spot the fact that there exist two potential solutions. One solution is trivial ($x = 0; y = 0$) and the other is not.

So what happens if we try to simulate our `QuiescentModel`? The answer is pretty obvious in the plot below:



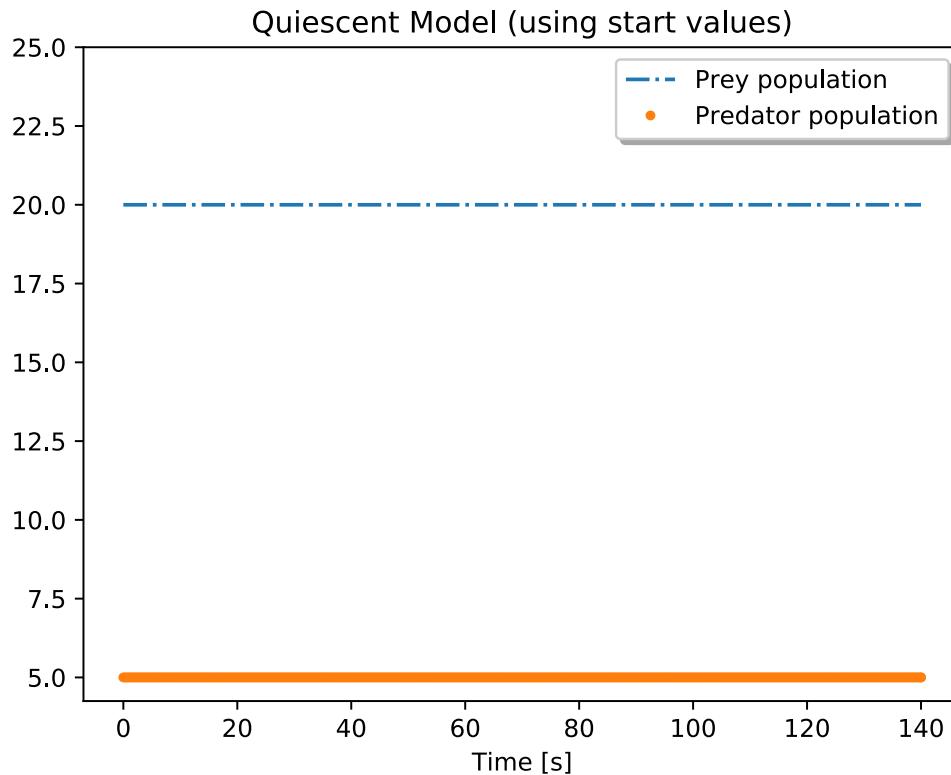
We ended up with the trivial solution where the prey and predator populations are zero. In this case, we have no reproduction, predation or starvation because all these effects are proportional to the populations (*i.e.*, zero) so nothing changes. But this isn't a very interesting solution.

There are two solutions to this system of equations because it is non-linear. How can we steer the non-linear solver away from this trivial solution? If you were paying attention during the discussion of the [Classic Lotka-Volterra](#) (page 18) model, then you've already been given a hint about the answer.

Recall that the `start` attribute is overloaded. During our discussion of the [Classic Lotka-Volterra](#) (page 18) model, it was pointed out that one of the purposes of the `start` attribute was to provide an initial guess if the variable with the `start` attribute was chosen as an iteration variable. Well, our `QuiescentModel` happens to be a case where `x` and `y` are, in fact, iteration variables because they must be solved using a system of non-linear equations. This means that if we want to avoid the trivial solution, we need to specify values for the `start` attribute on both `x` and `y` that are “far away” from the trivial solution we are trying to avoid (or at least closer to the non-trivial solution we seek). For example:

```
model QuiescentModelUsingStart "Find steady state solutions to LotkaVolterra equations"
  parameter Real alpha=0.1 "Reproduction rate of prey";
  parameter Real beta=0.02 "Mortality rate of predator per prey";
  parameter Real gamma=0.4 "Mortality rate of predator";
  parameter Real delta=0.02 "Reproduction rate of predator per prey";
  Real x(start=10) "Prey population";
  Real y(start=10) "Predator population";
initial equation
  der(x) = 0;
  der(y) = 0;
equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end QuiescentModelUsingStart;
```

This model leads us to a set of initial conditions that is more inline with what we were originally looking for (*i.e.*, one with non-zero populations for both the predator and prey species).



It is worth pointing out (as we will do shortly in the section on [Built-In Types](#) (page 29)), that **the default value of the “start” attribute is zero**. This is why when we simulated our original `QuiescentModel` we happened to land exactly on the trivial solution...because it was our initial guess and it happened to be an exact solution so no other solution or iterating was required.

Avoiding Repetition

We've already seen several different models (`ClassicModel`, `QuiescentModel` and `QuiescentModelUsingStart`) based on the Lotka-Volterra equations. Have you noticed something they all have in common? If you look closely, you will see that they have almost **everything** in common and that there are actually hardly any **differences** between them!

In software engineering, there is a saying that “Redundancy is the root of all evil”. Well the situation is no different here (in no small part because Modelica code really is software). The code we have written so far would be very annoying to maintain. This is because any bugs we found would have to be fixed in each model. Furthermore, any improvements we made would also have to be applied to each model. So far, we are only dealing with a relatively small number of models. But this kind of “copy and paste” approach to model development will result in a significant proliferation of models with only slight differences between them.

So what can be done about this? In object-oriented programming languages there are basically two mechanisms that exist to reduce the amount of redundant code. They are *composition* (which we will address in the future chapter on [Components](#) (page 183)) and *inheritance* which we will briefly introduce here.

If we look closely at the `QuiescentModelUsingStart` example, we see that there are almost no differences between it and our original `ClassicModel` version. In fact, the only real differences are shown here:

```
model ClassicModel "This is the typical equation-oriented model"
  parameter Real alpha=0.1 "Reproduction rate of prey";
  parameter Real beta=0.02 "Mortality rate of predator per prey";
  parameter Real gamma=0.4 "Mortality rate of predator";
  parameter Real delta=0.02 "Reproduction rate of predator per prey";
  parameter Real x0=10 "Start value of prey population";
  parameter Real y0=10 "Start value of predator population";
  Real x(start=x0) "Prey population";
  Real y(start=y0) "Predator population";
equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end ClassicModel;
```

```
model QuiescentModelUsingStart "Find steady state solutions to LotkaVolterra equations"
  parameter Real alpha=0.1 "Reproduction rate of prey";
  parameter Real beta=0.02 "Mortality rate of predator per prey";
  parameter Real gamma=0.4 "Mortality rate of predator";
  parameter Real delta=0.02 "Reproduction rate of predator per prey";
  Real x(start=10) "Prey population";
  Real y(start=10) "Predator population";
initial equation
  der(x) = 0;
  der(y) = 0;
equation
  der(x) = x*(alpha-beta*y);
  der(y) = y*(delta*x-gamma);
end QuiescentModelUsingStart;
```

In other words, the only real difference is the addition of the `initial equation` section (the original `ClassicModel` already contained non-zero `start` values for our two variables, `x` and `y`). Ideally, we could avoid having any redundant code by simply defining a model in terms of the differences between it and another model. As it turns out, this is exactly what the `extends` keyword allows us to do. Consider the following alternative to the `QuiescentModelUsingStart` model:

```
model QuiescentModelWithInheritance "Steady state model with inheritance"
  extends ClassicModel;
initial equation
  der(x) = 0;
  der(y) = 0;
end QuiescentModelWithInheritance;
```

Note the presence of the `extends` keyword. Conceptually, this “extends clause” simply asks the compiler to insert the contents of another model (`ClassicModel` in this case) into the model being defined. In this way, we copy (or “inherit”) everything from `ClassicModel` without having to repeat its contents. As a result, the `QuiescentModelWithInheritance` is the same as the `ClassicModel` with an additional set of initial equations inserted.

But what about a case where we don’t want **exactly** what is in the model we are inheriting from? For example, what if we wanted to change the values of the `gamma` and `delta` parameters?

Modelica handles this by allowing us to include a set of “modifications” when we use `extends`. These modifications come after the name of the model being inherited from as shown below:

```
model QuiescentModelWithModifications "Steady state model with modifications"
  extends QuiescentModelWithInheritance(gamma=0.3, delta=0.01);
end QuiescentModelWithModifications;
```

Also note that we could have inherited from `ClassicModel`, but then we would have had to repeat the initial equations in order to have quiescent initial conditions. But by instead inheriting from

`QuiescentModelWithModifications`, we reuse the content from **two** different models and avoid repeating ourselves even once.

More population dynamics

This concludes the set of examples for this chapter. If you'd like to explore the Lotka-Volterra equations in greater depth, an upcoming section titled *Lotka-Volterra Equations Revisited* (page 217) demonstrates how to build complex models of population dynamics using graphical components that are dropped onto a schematic and connected together.

1.1.2 Review

The first part of this chapter introduced languages features through examples. This part will review the various features and concepts and provide a more complete discussion of each topic.

Model Definition

A `model` definition is the most generic type of definition in Modelica. Later in the book (and even in this chapter), we'll be introducing other types of definitions (*e.g.*, `record` definitions) that share the same syntax as a `model` definition, but include some restrictions on what the definition is allowed to contain.

Syntax of a Model Definition

As we saw throughout this chapter, a model definition starts with the `model` keyword and is followed by a model name (and optionally a model description). The name of the model must start with a letter and can be followed by any collection of letters, numbers or underscores (_).

Naming conventions

Although not strictly required by the language. It is a convention that **model names start with an upper case letter**. Most model developers use the so-called “camel case” convention where the first letter of each word in the model name is upper case.

The model definition can contain variables and equations (to be discussed shortly). The end of the model is indicated by the presence of the `end` keyword followed by a repetition of the model name. Any text appearing after the sequence `//` and until the end of the line or between the delimiters `/*` and `*/` is considered a comment.

In summary, a model definition has the following general form:

```
model SomemodelName "An optional description"
    // By convention, variables are listed at the start
equation
    /* And equations are listed at the end */
end SomemodelName;
```

Inheritance

As we saw in the section on *Avoiding Repetition* (page 25), we can reuse code from other models by adding an `extends` clause to the model. It is worth noting that a model definition can include multiple `extends` clauses.

Each `extends` clause must include the name of the model being extended from and can be optionally followed by modifications that are applied to the contents of the model being extended from. In the case of a model definition that inherits from other model definitions, you can think of the general syntax as looking something like this:

```
model Specialized modelName "An optional description"
    extends Model1; // No modifications
    extends Model2(n=5); // Including modification
    // By convention, variables are listed at the start
    equation
        /* And equations are listed at the end */
end Specialized modelName;
```

By convention, `extends` clauses are normally listed at the very top of the model definition, before any variables.

In later chapters, we will show how this same syntax can be used to define other entities besides models. But for now, we will focus primarily on models.

Variables

As we saw in the previous section, a model definition typically contains variable declarations. The basic syntax for a variable declaration is simply the “type” of the variable (which will be discussed shortly in the section on [Built-In Types](#) (page 29)) followed by the name of the variable, *e.g.*,

```
Real x;
```

Variables sharing the same type can be grouped together using the following syntax:

```
Real x, y;
```

A declaration can also be followed by a description, *e.g.*:

```
Real alpha "angular acceleration";
```

Variability

Parameters

By default, variables declared inside a model are assumed to be continuous variables (variables whose solution is generally smooth, but which may also include discontinuities). However, as we first saw in the section titled [Getting Physical](#) (page 6), it is also possible to add the `parameter` qualifier in front of a variable declaration and to indicate that the variable is known *a priori*. You can think of a parameter as “input data” to the model that is constant with respect to time.

Constants

Closely related to the `parameter` qualifier is the `constant` qualifier. When placed in front of a variable declaration, the `constant` qualifier also implies that the value of the variable is known *a priori* and is constant with respect to time. The distinction between the two lies in the fact that a `parameter` value can be changed from one simulation to the next whereas the value of a `constant` cannot be changed once the model is compiled. The use of `constant` by a model developer ensures that end users are not given the option to make changes to the `constant`. A `constant` is frequently used to represent physical quantities like π or the Earth’s gravitational acceleration, which can be assumed constant for most engineering simulations.

Discrete Variables

Another qualifier that can be placed in front of a variable declaration is the `discrete` qualifier. We have not yet shown any example where the `discrete` qualifier would be relevant. However, it is included now for completeness since it is the last remaining variability qualifier.

Built-In Types

Many of the examples so far referenced the `Real` type when declaring variables. As the name suggests, `Real` is used to represent real valued variables (which will generally be translated into floating point representations by a Modelica compiler). However, `Real` is just one of the four built-in types in Modelica.

Another of the built-in types is the `Integer` type. This type is used to represent integer values. `Integer` variables have many uses including representing the size of arrays (this use case will be discussed shortly in an upcoming section on [Vectors and Arrays](#) (page 87)).

The remaining built-in types are `Boolean` (used to represent values that can be either `true` or `false`) and `String` (used for representing character strings).

Each of the built-in types restricts the possible values that a variable can have. Obviously, an `Integer` variable cannot have the value 2.5, a `Boolean` or `String` cannot be 7 and a `Real` variable cannot have the value "Hello".

Derived Types

As we saw in the previous examples that introduced [Physical Types](#) (page 9), it is possible to “specialize” the built-in types. This feature is used mainly to modify the values associated with [Attributes](#) (page 30) like `unit`. The general syntax for creating derived types is:

```
type NewTypeName = BaseTypeName(* attributes to be modified *);
```

Frequently, the `BaseTypeName` will be one of the built-in types (*e.g.*, `Real`). But it can also be another derived type. This means that multiple levels of specialization can be supported, *e.g.*,

```
type Temperature = Real(unit="K"); // Could be a temperature difference
type AbsoluteTemperature = Temperature(min=0); // Must be positive
```

Enumerations

An `enumeration` type is very similar to the `Integer` type. An `enumeration` is typically used to define a type that can take on only a limited set of specific values. In fact, enumerations are not strictly necessary in the language. Their values can always be represented by integers. However, the `enumeration` type is safer and more readable than an `Integer`.

There are two built-in enumeration types. The first of these is `AssertionLevel` and it is defined as follows:

```
type AssertionLevel = enumeration(warning, error);
```

The significance of these values will be discussed in a forthcoming section on [assert](#) (page 258).

The other built-in enumeration is `StateSelect` and it is defined as follows:

```
type StateSelect = enumeration(never, avoid, default, prefer, always);
```

Attributes

So far in this chapter we have mentioned attributes (*e.g.*, `unit`), but we haven't discussed them in detail. For example, *which* attributes are present on a given variable? This depends on the type of the variable (and which built-in and derived types it is based on). The following table introduces all the possible attributes indicating their types (*i.e.*, what type of value can be given for that attribute), which types they can be associated with and finally a brief description of the attribute:

Attributes of Real

quantity A textual description of what the variable represents

Default: ""

Type: String

start The `start` attribute has many uses. The main purpose of the `start` attribute (as discussed extensively in the section on [Initialization](#) (page 35)) is to provide "fallback" initial conditions for state variables (see `fixed` attribute for more details).

The `start` attribute may also be used as an initial guess if the variable has been chosen as an iteration variable.

Finally, if a `parameter` doesn't have an explicit value specified, the value of the `start` attribute will be used as the default value for the `parameter`.

Default: 0.0

Type: Real

fixed The `fixed` attribute changes the way the `start` attribute is used when the `start` attribute is used as an initial condition. Normally, the `start` attribute is considered a "fallback" initial condition and only used if there are insufficient initial conditions explicitly specified in the `initial equation` sections. However, if the `fixed` attribute is set to `true`, then the `start` attribute is treated as if it was used as an explicit `initial equation` (*i.e.*, it is no longer used as a fallback, but instead treated as a strict initial condition).

Another, more obscure, use of the `fixed` attribute is for "computed parameters". In rare cases where a `parameter` cannot be initialized explicitly, it is possible to provide a general equation for the parameter in an `initial equation` section. But in cases where the `parameter` is initialized in this way, the `fixed` attribute for the parameter variable must be set to `false`.

Default: false (except for `parameter` variables, where it is true by default)

Type: Boolean

min The `min` attribute is used to specify the minimum allowed value for a variable. This attribute can be used by editors and compilers in various ways to inform users or developers about potentially invalid input data or solutions.

Default: -DBL_MAX where DBL_MAX is the largest floating point value that can be represented for the given platform.

Type: Real

max The `max` attribute is used to specify the maximum allowed value for a variable. This attribute can be used by editors and compilers in various ways to inform users or developers about potentially invalid input data or solutions.

Default: DBL_MAX where DBL_MAX is the largest floating point value that can be represented for the given platform.

Type: Real

unit As discussed extensively in this chapter, variables can have physical units associated with them. There are rules about how these units are expressed, but the net result is that by using the **unit** attribute it is possible check models to make sure that equations are physically consistent. A value of "1" indicates the value has no physical units. On the other hand, a value of "" (the default value if no value is given) indicates that the physical units are simply unspecified. The difference between "1" and "" is that the former is an explicit statement that the quantity is dimensionless (has no units) while the latter indicates that the quantity may have physical units but they are left unspecified.

Default: "" (*i.e.*, no physical units specified)

Type: String

displayUnit While the **unit** attribute describes what physical units should be associated with the value of a variable, the **displayUnit** expresses a preference for what units should be used when displaying the value of a variable. For example, the SI unit for pressure is *Pascals*. However, standard atmospheric pressure is 101,325 *Pascals*. When entering, displaying or plotting pressures it may be more convenient to use *bars*.

The **displayUnit** attribute doesn't affect the value of a variable or the equations used to simulate a model. It only affects the *rendering* of those values by potentially transforming them into more convenient units for display.

Default: ""

Type: String

nominal The **nominal** attribute is used to specify a nominal value for a variable. This nominal value is generally used in numerical calculations to perform various types of scaling used to avoid round-off or truncation error.

Default:

Type: Real

stateSelect The **stateSelect** attribute is used as a hint to Modelica compilers about whether a given variable should be chosen as a state (in cases where there is a choice to be made). As discussed previously in the section on *Enumerations* (page 29), the possible values for this attribute are **never**, **avoid**, **default**, **prefer** and **always**.

Default: default

Type: StateSelect (enumeration, see *Enumerations* (page 29))

Attributes of Integer

quantity A textual description of what the variable represents

Default: ""

Type: String

start It is worth noting that an **Integer** variable can be chosen as a state variable or as an iteration variable. Under these circumstances, the **start** attribute may be used by a compiler in the same was as it is for **Real** variables (*see previous discussion of Attributes of Real* (page 30))

In the case of a **parameter**, the **start** attribute will (as usual) be used as the default value for the **parameter**.

Default: 0.0

Type: Integer

fixed *see previous discussion of Attributes of Real* (page 30)

Default: false (except for **parameter** variables, where it is true by default)

Type: Boolean

min The `min` attribute is used to specify the minimum allowed value for a variable. This attribute can be used by editors and compilers in various ways to inform users or developers about potentially invalid input data or solutions.

Default: $-\infty$

Type: Integer

max The `max` attribute is used to specify the maximum allowed value for a variable. This attribute can be used by editors and compilers in various ways to inform users or developers about potentially invalid input data or solutions.

Default: ∞

Type: Integer

Attributes of Boolean

quantity A textual description of what the variable represents

Default: ""

Type: String

start It is worth noting that an `Boolean` variable can be chosen as a state variable or as an iteration variable. Under these circumstances, the `start` attribute may be used by a compiler in the same was as it is for `Real` variables (*see previous discussion of Attributes of Real* (page 30))

In the case of a `parameter`, the `start` attribute will (as usual) be used as the default value for the `parameter`.

Default: 0.0

Type: Boolean

fixed *see previous discussion of Attributes of Real* (page 30)

Default: false (except for `parameter` variables, where it is true by default)

Type: Boolean

Attributes of String

quantity A textual description of what the variable represents

Default: ""

Type: String

start Technically, a `String` could be chosen as a state variable (or even an iteration variable), but in practice this never happens. So for a `String` variable the only practical use of the `start` attribute is to define the value of a `parameter` (that happens to have the type of `String`) if no explicit value for the parameter is given.

Default: ""

Type: String

It is worth noting that *Derived Types* (page 29) retain the attributes of the built-in type that they are ultimately derived from. Also, although the type of, for example, the `min` attribute on a `Real` variable is listed having the type `Real` it should be pointed out explicitly that attributes cannot themselves have attributes. In other words, the `start` attribute doesn't have a `start` attribute.

Modifications

So far, we've seen two types of modifications. The first is when we change the value of an attribute, *e.g.*,

```
Real x(start=10);
```

In this case, we are creating a variable `x` of type `Real`. But rather than leaving it "as is", we then apply a modification to `x`. Specifically, we "reach inside" of `x` and change the `start` attribute value. In this example, we are only going one level into `x` to make our modification. But as we will see in our next example, it is possible to make modifications at arbitrary depths.

The other case where we have seen modifications was in the section on [Avoiding Repetition](#) (page 25). There we saw modification used in conjunction with `extends` clauses, *e.g.*,

```
extends QuiescentModelWithInheritance(gamma=0.3, delta=0.01);
```

Here, the modification is applied to elements that were inherited from the `QuiescentModelWithInheritance` model. As with modifications to attributes, the element being modified (a model in this case) is followed by parentheses and inside those parentheses we specify the modifications we wish to make.

It is worth noting that modifications can be nested arbitrarily deep. For example, imagine we wanted to modify the `start` attribute for the variable `x` inherited from the `QuiescentModelWithInheritance` model. In Modelica, such a modification would be made as follows:

```
extends QuiescentModelWithInheritance(x(start=5));
```

Here we first "reach inside" the `QuiescentModelWithInheritance` model to modify the contents that we "inherit" from it (`x` in this case) and then we "reach inside" `x` to modify the value of the `start` attribute.

One of the central themes of Modelica is support for reuse and avoiding the need to "copy and paste" code. Modifications are one of the essential features in Modelica that support reuse. We'll learn about others in future sections.

Equations

Although equations are probably the single most important mathematical aspect of Modelica, they are also the simplest to explain.

Basic Equations

There are really no complicated semantics to explain about equations. All equations are composed of a left hand expression and a right hand expression separated by an equals sign, *i.e.*,

```
<left-hand expression> = <right-hand expression>;
```

Through the examples presented in this chapter, the reader has been exposed to this pattern over and over again in each example. The only real deviation from the syntax shown above is the case where a description of the equation is included as well, *e.g.*,

```
V = i*R "Ohm's law";
m*der(v) = F "Newton's law";
```

As was pointed out previously, the left hand and right hand sides of an equation in Modelica are expressions, not assignments. In other words (and in contrast to most programming languages), the left hand side does **not** have to be a variable (as we can see in the case of Newton's law above).

Initial Equations

As we saw in many of the examples in this chapter, it is possible to specify equations within a model to be used to solve for initial conditions. This entire topic of initialization will be discussed in detail in the next section, titled [Initialization](#) (page 35). For now, all we will say on this topic is that if an equation is to be applied *only* to solve for initial conditions, the `equation` section must be qualified by the `initial` keyword as follows:

```
initial equation
  x = 0; // Only used to solve for initial conditions
```

Conditional Equations

In the next chapter, we'll discuss how to use `if` statements to represent conditional behavior. It is worth getting ahead of ourselves a little bit to point out that equations can be conditional. There are really two forms of conditional equations. The first is the balanced form, *e.g.*,

```
if a>b then
  x = 5*time;
else
  x = 3*time;
end if;
```

In the balanced case, the number of equations is always the same (1 in the code above), but *which* equation can change. This is important because to simulate a model in Modelica, the number of variables must equal the number of equations and the number of equations must be fixed during the simulation.

The other type of conditional equations are ones where the number of equations is unbalanced. This means that the number of equations on the `if` side may not be equal to the number of equations on the `else` side (like it was in the balanced case, previously).

But remember, the number of equations cannot change during a simulation. So how is it then that the number of equations can be different from the `if` side to the `else` side? It can only happen if **the value of the conditional expression cannot change during the simulation**. In order to be able to ensure that the conditional expression can never change, it is necessary that all variables in the conditional expression have so-called *parametric variability*.

Remember in our discussion of [Variability](#) (page 28) the fact that variables with the `parameter` qualifier cannot change during a simulation? If a variable with the `parameter` qualifier cannot change during a simulation and all the variables in an expression have this parametric variability then the entire expression must also have parametric variability (*i.e.*, the value of the expression cannot change during a simulation).

At this point, you might be asking yourself why this unbalanced case would be useful? Again, we are getting ahead of ourselves here, but one use case would be the conditional application of initial equations, *e.g.*,

```
...
parameter Boolean steady_state;
initial equation
  if steady_state then
    der(x) = 0;
    der(y) = 0;
  ...

```

In other words, if the Boolean parameter `steady_state` is true, then the initial equations are enforced. But if the parameter is false, they are not. The conditional expression here clearly has parametric variability because the expression contains only a variable and that variable is a parameter.

That's all we'll say on this topic for now, since discrete and conditional behavior will be discussed in detail in the [next chapter](#) (page 42).

Initialization

Overview

As we already touched on during our previous discussion on *Steady State Initialization* (page 22), behavior is represented by both the equations contained in a model as well as the initial conditions given to the state variables in the model. In Modelica, the initial conditions are computed by combining the normal equations (present in *equation* sections) with any initial equations (present in *initial equation* sections).

One of the first sources of confusion for new users is understanding how many initial conditions are required. The answer to this question is simple. In order to have a well-posed initialization problem (one where we don't have too many or too few initial equations), we need to have the same number of equations in the *initial equation* sections as we have states in our system. **Note**, we can get away with having too few, because tools can augment the initial equations we provide with additional ones until the problem is well-posed, but we may not be able to solve a problem where we have too many initial equations (since this depends on the tools ability to recognize and eliminate redundant equations and different tools provide different levels of support for this).

Of course, saying the number of initial equations has to be equal to the number of states answers one question, but quickly creates another, *i.e.*, *how do we determine how many states there are?* For the models we've seen in this chapter, the answer is quite simple. The states in each of our examples so far are the variables that appear inside the `der(...)` operator. In other words, every variable that we differentiated in those examples is a state.

Ordinary Differential Equations

It is important to note that **it will not always be the case** that every variable that we differentiate will be a state. In this chapter, all the models we have seen so far are ordinary differential equations (ODEs). When dealing with ODEs, every differentiated variable is a state, which, in turn, means that you need an initial equation for each of these differentiated variables. But in subsequent chapters we will eventually run across examples that are so-called differential-algebraic equations (DAEs). In those cases, only *some* of the differentiated variables can be considered states.

As it turns out, understanding initialization doesn't really require us to get into a detailed discussion about DAEs. In practice, all Modelica tools perform something called "index reduction". While the index reduction algorithms themselves are fairly complicated (so we won't get into those now), the effect is quite simple. Index reduction transforms the DAEs into ODEs. In other words, Modelica compilers will transform whatever DAE problem contained in our Modelica code into this relatively easy to explain ODE form.

So let's side-step the discussion about DAEs and index reduction and just pick up our discussion of initialization assuming our problem has already been reduced to an ODE. In this case, the only thing we really need to understand is that initialization is required for all states in the model and that our model will have the following general ODE form:

$$\begin{aligned}\dot{\vec{x}}(t) &= \vec{f}(\vec{x}(t), \vec{u}(t), t) \\ \vec{y}(t) &= \vec{g}(\vec{x}(t), \vec{u}(t), t)\end{aligned}$$

where t is the current simulation time, $\vec{x}(t)$ are the values of the states in our system at time t , $\vec{u}(t)$ are the values of any external inputs to our system at time t .

Note that the arrow over a variable simply indicates that it is a vector, not a scalar. Also note that the only variable that appears differentiated in this problem is \vec{x} . This is how we know that \vec{x} represents the states in the system. One final thing to note about this system is that neither function, \vec{f} nor \vec{g} , depends on \vec{y} .

If you think about it, both t and $\vec{u}(t)$ are external to our system. We don't compute them or control them. The reason that we call \vec{x} the state of our system is that it is the only information (from within our system) needed to compute $\dot{\vec{x}}(t)$ and $\vec{y}(t)$ (which, in turn, are the only things we need to compute in order to arrive at a solution).

Getting back to the topic of initialization, during a normal time step we will solve for $\vec{x}(t)$ by integrating $\dot{\vec{x}}(t)$ to compute $\vec{x}(t)$. In other words:

$$\vec{x}(T) = \int_{t_i}^T \dot{\vec{x}}(t) dt + \vec{x}(t_i)$$

This all works as long as there **was** a previous time step. When there wasn't a previous time step, then the value of \vec{x} that we plug into our equations has to be the very first value of \vec{x} in our simulation. In other words, our initial conditions.

One might imagine that we would specify our initial conditions by adding an equation like this:

$$\vec{x}(t_0) = \vec{x}_0$$

where t_0 is the start time of our simulation and \vec{x}_0 is an explicit specification of the initial values. Providing explicit values for states is a very common case when specifying initial conditions. So we definitely need to be able to handle this case. But this approach won't work for the cases we showed in [Steady State Initialization](#) (page 22). There we didn't provide explicit initial values for states. Instead, we provided initial values for $\dot{\vec{x}}(t_0)$. So how can we capture both of these cases?

Initial Equations

The answer is to assume that at the start of our simulation we need to solve a problem that looks like this:

$$\begin{aligned}\dot{\vec{x}}(t_0) &= \vec{f}(\vec{x}(t_0), \vec{u}(t_0), t_0) \\ \vec{y}(t_0) &= \vec{g}(\vec{x}(t_0), \vec{u}(t_0), t_0) \\ \vec{0} &= \vec{h}(\vec{x}(t_0), \dot{\vec{x}}(t_0), \vec{u}(t_0), t_0)\end{aligned}$$

Note the introduction of a new function, \vec{h} . This new function represents any equations we have placed in *initial equation* sections. The fact that \vec{h} takes both \vec{x} **and** $\dot{\vec{x}}$ as arguments allows us to express a wide range of initial conditions. To define explicit initial values for states, we could define \vec{h} as:

$$\vec{h}(\vec{x}(t_0), \dot{\vec{x}}(t_0), \vec{u}(t_0), t_0) = \vec{x}(t_0) - \vec{x}_0$$

But we could also express our desire to start with a steady state solution by defining \vec{h} as:

$$\vec{h}(\vec{x}(t_0), \dot{\vec{x}}(t_0), \vec{u}(t_0), t_0) = \dot{\vec{x}}(t_0)$$

And, of course, we could mix these different forms or use a wide range of other forms on a per state basis to describe our initial conditions. So when writing initial equations, all you need to keep in mind is that they need to be of the general form shown above and that you cannot have more of them than you have states in your system.

Conclusion

As we've demonstrated in this chapter, the *initial equation* construct in Modelica allows us to express many ways to initialize our system. In the end, all of them will compute the initial values for the states in our system. But we are given tremendous latitude in describing exactly how those values will be computed.

This is an area where Modelica excels. Initialization is given first class treatment in Modelica and this flexibility pays off in many real world applications.

Record Definitions

Earlier, we introduced the idea of a `model` definition. Although we haven't seen any yet, Modelica also includes a `record` type. A `record` can have variables, just like a `model`, but it is not allowed to include equations. Records are primarily used to group data together. But as we will see shortly, they are also very useful in describing the data associated with [Annotations](#) (page 37).

Syntax

The **record** definition looks essentially like a **model** definition, but without any equations:

```
record RecordName "Description of the record"
  // Declarations for record variables
end RecordName;
```

As with a **model**, the definition starts and ends with the name of the record being defined. An explanation of the **record** can be included as a string after the name. All the variables associated with the record are declared within the **record** definition.

The following are all examples of **record** definitions:

```
record Vector "A vector in 3D space"
  Real x;
  Real y;
  Real z;
end Vector;

record Complex "Representation of a complex number"
  Real re "Real component";
  Real im "Imaginary component";
end Complex;
```

Record Constructors

Now that we know how to define a **record**, we need to know how to create one. If we are declaring a variable that happens to be a **record**, the declaration itself will create an instance of the **record** and we can specify the values of variables inside the record using modifications, *e.g.*,

```
parameter Vector v(x=1.0, y=2.0, z=0.0);
```

But there are some cases where we might want to create an instance of a **record** that isn't a variable (*e.g.*, to use in an expression, pass as an argument to a function or use in a modification). For each **record** definition, a function is automatically generated with the **same name** as the **record**. This function is called the “record constructor”. The record constructor has input arguments that match the variables inside the **record** definition and returns an instance of that **record**. So in the case of the **Vector** definition above, we could also initialize a **parameter** using the record constructor as follows:

```
parameter Vector v = Vector(x=1.0, y=2.0, z=0.0);
```

In this case, the value for **v** comes from the **expression** `Vector(x=1.0, y=2.0, z=0.0)` which is a call to the record constructor.

Annotations

Recall in the discussion on *Experimental Conditions* (page 5) we included information about the simulation start and stop time using an **annotation**. An **annotation** is a way to include information that is not related to the behavior of the model. In the case of experimental conditions, we injected information about how a particular model should be simulated. But annotations are used extensively in Modelica to provide all kinds of additional information about models. For example, as we'll see *later in the book* (page 178), annotations are used to describe the graphical appearance of components and connectors. For now, the important thing is to understand that annotations are additional data, above and beyond behavior, that can be “attached” to different elements in Modelica.

In this section, we will first cover where an **annotation** can appear in a Modelica model. Next, we'll explain how we can use *Record Definitions* (page 36) to describe the contents of an annotation. Finally,

we'll describe a few of the many "standard" annotations that are included as part of the Modelica specification.

Annotation Locations

Annotations can appear in many different places in Modelica. We will discuss each potential location and demonstrate the syntax for each case.

Declaration Annotations

A declaration annotation comes at the end of a declaration, right before the ;. Here is a simple declaration that includes an annotation:

```
parameter Real length "Rod length" annotation(...);
```

Note that the `annotation` comes after the descriptive string and before the ;. Also, the ... is simply a place holder for the *Annotation Data* (page 39), which will be discussed shortly.

Statement and Equation Annotations

It is also possible to associate annotations with equations, for example:

```
T = T0 "Specify initial value for T" annotation(...);
```

In declarations and equations, the `annotation` is always at the very end and comes immediately before the ;.

Inheritance Annotations

We briefly discussed the `extends` keyword when we talked about *Modifications* (page 33) and *Avoiding Repetition* (page 25). It is possible to associate an `annotation` with an `extends` clause as follows:

```
extends QuiescentModelWithInheritance(gamma=0.3, delta=0.01) annotation(...);
```

As we've observed in each previous case, the `annotation` immediately precedes the ;.

Model Annotations

A model annotation associates annotation data directly with the model definition itself. This is exactly the kind of annotation we saw when describing *Experimental Conditions* (page 5), e.g.,

```
model FirstOrderExperiment "Defining experimental conditions"
  Real x "State variable";
initial equation
  x = 2 "Used before simulation to compute initial values";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
  annotation(experiment(StartTime=0,StopTime=8));
end FirstOrderExperiment;
```

Note how, unlike all the previous annotation locations we've described, this annotation isn't really "attached" to anything. This indicates that it is annotating the model itself.

Annotation Data

General Syntax

The syntax of an annotation is the same syntax used for *Modifications* (page 33). This means the annotation will include either an assignment to a variable in the annotation, *e.g.*,

```
annotation(Evaluate=true);
```

or it will include a modification to something **inside** a variable in the annotation, *e.g.*,

```
annotation(experiment(StartTime=0,StopTime=8));
```

User Annotations

Annotations were designed to allow model developers to attach **arbitrary data** to their models. For example, if a user wanted to associate a part number with a given model definition, they might introduce a model annotation like this:

```
annotation(PartNumber="FF78-E4B879");
```

A general principle of annotation data is that if a tool reads in a model, **it must preserve the annotation information** when it writes it back out. The tool does not (and, in general, will not) have to understand the data. But the data must be preserved.

Multiple Annotations

Imagine a user wanted to specify **both** a part number and an experiment annotation. Then they might end up with an annotation like this one:

```
annotation(PartNumber="FF78-E4B879",
           experiment(StartTime=0,StopTime=8));
```

Note how these two pieces of information can exist side by side. One way to think about annotations is to visualize them as a tree like this:

- PartNumber="FF78-E4B879"
- experiment
 - StartTime=0
 - StopTime=8

Namespaces

This introduces another principle of annotations which is that it should be possible to have more than one **as long as the names are different**. For this reason, choosing names is very important and they should be chosen to avoid potential conflicts with other names. For example, a better approach for including the part number would be to enclose it in a variable that is more likely to be unique to your company or application, *e.g.*:

```
annotation(XogenyIndustries(PartNumber="FF78-E4B879"),
           experiment(StartTime=0,StopTime=8));
```

In this case, the variable **XogenyIndustries** can be used to carve out a “namespace” for a specific organization or purpose. If another organization came along and wanted to associate a different part

number with the same model, they could do that by establishing their own separate hierarchy in the annotation, *e.g.*:

```
annotation(XogenyIndustries(PartNumber="FF78-E4B879"),
           AcmeEquipment(PartNumber="A23335-992"),
           experiment(StartTime=0,StopTime=8));
```

Occasionally, Modelica tool vendors include their own special annotations (*e.g.*, in the Modelica Standard Library). By convention, tool vendors use names that are prefixed by two underscores, *e.g.*,

```
annotation(XogenyIndustries(PartNumber="FF78-E4B879"),
           __ModelicateTechnologies(enableCoolFeature10=true),
           AcmeEquipment(PartNumber="A23335-992"),
           experiment(StartTime=0,StopTime=8));
```

Interpretation

Remember that annotation data is arbitrary. This allows arbitrary data to be associated with the model. The **meaning** of that data is, in general, not defined in the Modelica specification. As we will see shortly, there are a few “standard” annotations (they will be described throughout this book) and they are documented in the specification. But when users add annotations beyond the standard annotations it is assumed that they have some way (using some Modelica tool, compiler or other Modelica aware technology) of extracting and interpreting their annotation data.

The bottom line is that while you can inject (non-standard) annotation data into the model, tools are only required to preserve it and not to interpret it.

Documentation

It is very common to document Modelica annotations **as if** they had *Record Definitions* (page 36) associated with them. We’ll see several examples of this technique in our next topic. Using this approach to document expected annotation data are strongly encouraged. In fact, this technique is so popular and useful that there are proposals to actually make it part of the language itself in the future.

Introductory Annotations

This section introduces just a few of the “standard annotations” in Modelica. As discussed previously, annotations are generally allowed to include arbitrary data that is preserved by tools and, presumably, interpreted at some point. The syntax and meaning of the standard annotations are described in the Modelica specification so they can be interpreted consistently and universally by Modelica tools.

We will follow a convention (whenever possible) of describing standard annotations in terms of **record** definitions. These **record** definitions don’t formally exist, they are simply a concise way of expressing the data contained in the annotation.

Documentation

Type: Model Annotation

The **Documentation** annotation in Modelica allows raw text or HTML to be associated with a model as documentation. This documentation is composed of two components. The first is information about the model and the second is revision history information. The structure of the **Documentation** annotation is described by the following record definition:

```
record Documentation
  String info "Documentation in text or HTML format";
  String revision "Revision information in text or HTML format";
end Documentation;
```

When embedding HTML inside an annotation, the HTML code must be surrounded by `<html>` tags, e.g.,

```
model MyWidget
  // ... declarations
  annotation(
    Documentation(
      info="<html><h1 class=\\"heading\\>Introduction</h1><p>...</p></html>");
    // ... equations
  end MyWidget;
```

Here the model `MyWidget` contains HTML documentation. The documentation is wrapped by `<html>` tags and all quotes used to define attributes are escaped by `\` to avoid accidentally terminating the `info` string.

`experiment`

Type: Model Annotation

The `experiment` annotation is used to specify information about how a given model should be simulated. The annotation data can be represented in `record` form as:

```
record experiment
  Real StartTime "Time at which the simulation should start";
  Real StopTime "Time at which the simulation should stop";
  Real Interval(min=0) "Time interval between results";
  Real Tolerance(min=0) "Solver tolerance to use";
end experiment;
```

`Evaluate`

Type: Declaration Annotation (applies to parameters)

The `Evaluate` annotation indicates to a Modelica compiler that the value of a given `parameter` can be transformed into a `constant` at compile time. In other words, it indicates that the user does not require the ability to change the value of the `parameter` from one simulation to the next.

The motivation behind having such an annotation is that it allows the Modelica compiler to assume many things about the `parameter` during model compilation that it otherwise couldn't. These assumptions might restrict the system of equations in such a way that the underlying systems of equations are easier to solve than in the general case where the parameter could take on a range of values.

The `Evaluate` annotation is simply a Boolean variable (`true` indicating that the `parameter` value can be transformed into a `constant`). It is used in an annotation as follows:

```
parameter Real x annotation(Evaluate=true);
```

`HideResult`

Type: Declaration Annotation

The `HideResult` annotation is used to indicate that the solution for a given variable is not of interest to the analyst. By setting the value of `HideResult` to `true`, the model developer is indicating to

the Modelica compiler that it need not store the annotated variable in any simulation results that are produced. This can save both simulation time and disk space because it avoids writing out data that will never be viewed.

The `HideResult` annotation would be used as follows:

```
Real z "Uninteresting variable" annotation(HideResult=true);
```

1.2 Discrete Behavior

So far, all the examples we've seen have been of a purely continuous nature. This means that there have been no abrupt disturbances in the system. In this chapter, we'll focus on how to express what we call "discrete behavior". There are a wide variety of different engineering use cases for describing such behavior and we'll explore these through the various examples presented in this chapter.

Normally, when we talk about discrete behavior we often refer to "events". An event is something that occurs in our system that triggers some kind of discontinuity. Differential equations normally result in continuous solutions. But when events occur, they can introduce various kinds of discontinuities.

The simplest types of events are ones that happen at a particular time. These are, not surprisingly, called "time events". Because these events are tied to time, we know what time they will occur even before they happen. Examples of time events would be things like changes triggered by some kind of digital clock that is activated at some specified frequency.

The other type of event we will encounter are so-called "state events". These kinds of events are much trickier to handle. The reason is that we do not know *a priori* when these events will occur. Unlike time events, we have to actually wait for some signal in our system to cross some specified threshold. Generally speaking, we don't know when that crossing will occur. Furthermore, determining the precise moment when the event occurs is somewhat expensive.

In this chapter, we'll look at examples of both of these kinds of events and the various Modelica language features that can be used to describe when these events occur and how we describe responses to them.

1.2.1 Examples

Cooling Revisited

Changing Ambient Conditions

We will start with a simple example that demonstrates time events. We will revisit the thermal model introduced previously in the section on [Physical Types](#) (page 9). However, this time we will introduce a disturbance to that system. Specifically, we will trigger an abrupt decrease in the ambient temperature after half a second of simulation. This revised model is written as follows:

```
model NewtonCoolingDynamic
  "Cooling example with fluctuating ambient conditions"
  // Types
  type Temperature=Real(unit="K", min=0);
  type ConvectionCoefficient=Real(unit="W/(m2.K)", min=0);
  type Area=Real(unit="m2", min=0);
  type Mass=Real(unit="kg", min=0);
  type SpecificHeat=Real(unit="J/(K.kg)", min=0);

  // Parameters
  parameter Temperature T0=363.15 "Initial temperature";
  parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
  parameter Area A=1.0 "Surface area";
  parameter Mass m=0.1 "Mass of thermal capacitance";
  parameter SpecificHeat c_p=1.2 "Specific heat";
```

```

// Variables
Temperature T_inf "Ambient temperature";
Temperature T "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  if time<=0.5 then
    T_inf = 298.15 "Constant temperature when time<=0.5";
  else
    T_inf = 298.15-20*(time-0.5) "Otherwise, decreasing";
  end if;
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingDynamic;

```

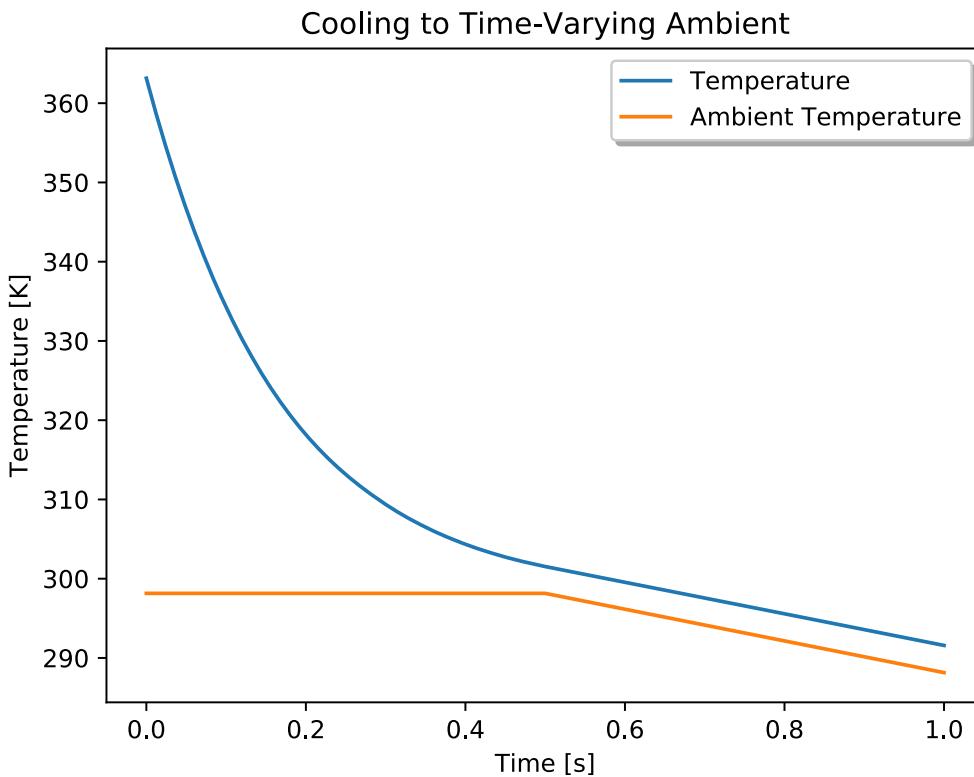
The highlight lines show an `if` statement. This particular `if` statement provides two different equations for computing `T_inf`.

Time

You will note in this model the variable `time` is not declared within our model. This is because `time` is a built-in variable in all Modelica models.

The decision about which of the two equations will actually be used depends on the conditional expression `time<=0.5`. It is because this expression only depends on `time` and not any other variables in our model that we can characterize the transition between these two equations as a “time event”. The key point is that when integrating these equations, we can tell the solver that integrates our system of equations to stop precisely at 0.5 seconds and then resume again using a different equation. We’ll see examples of other state events where this would not be possible, in the next section when we present the classic *Bouncing Ball* (page 47) example.

But for now, let us continue with our cooling example. If we simulate this model for one second, we get the following temperature trajectory:



As you can see in these results, the ambient temperature does indeed start to decrease after half a second. In studying the dynamic response of the temperature itself, we see two distinct phases. The first phase is the initial transient response toward equilibrium (to match the ambient temperature). The second phase is the tracking of the ambient temperature as it decreases.

Initial Transients

It is worth noting that this is a very common issue in modeling. Frequently, you wish to model the systems response to some disturbance (like the ambient temperature decrease in this case). However, if you don't start your system in some kind of equilibrium state, the system response will also include some kind of initial transient (like the one shown here). In order to distinguish these two responses clearly, we want to avoid any overlap between them. **The simplest way to do that is to start the simulation in an equilibrium condition** (as discussed previous in our discussion of *Steady State Initialization* (page 22)). This avoids the initial transient altogether and allows us to focus entirely on the disturbance that we are interested in.

As we learned during our discussion of *Initialization* (page 35), we can solve this problem of initial transients by simply including an initial equation that will determine a value for T such that our system starts in an equilibrium state, *i.e.*,

```
model NewtonCoolingSteadyThenDynamic
  "Dynamic cooling example with steady state conditions"
  type Temperature=Real(unit="K", min=0);
  type ConvectionCoefficient=Real(unit="W/(m2.K)", min=0);
  type Area=Real(unit="m2", min=0);
  type Mass=Real(unit="kg", min=0);
  type SpecificHeat=Real(unit="J/(K.kg)", min=0);
```

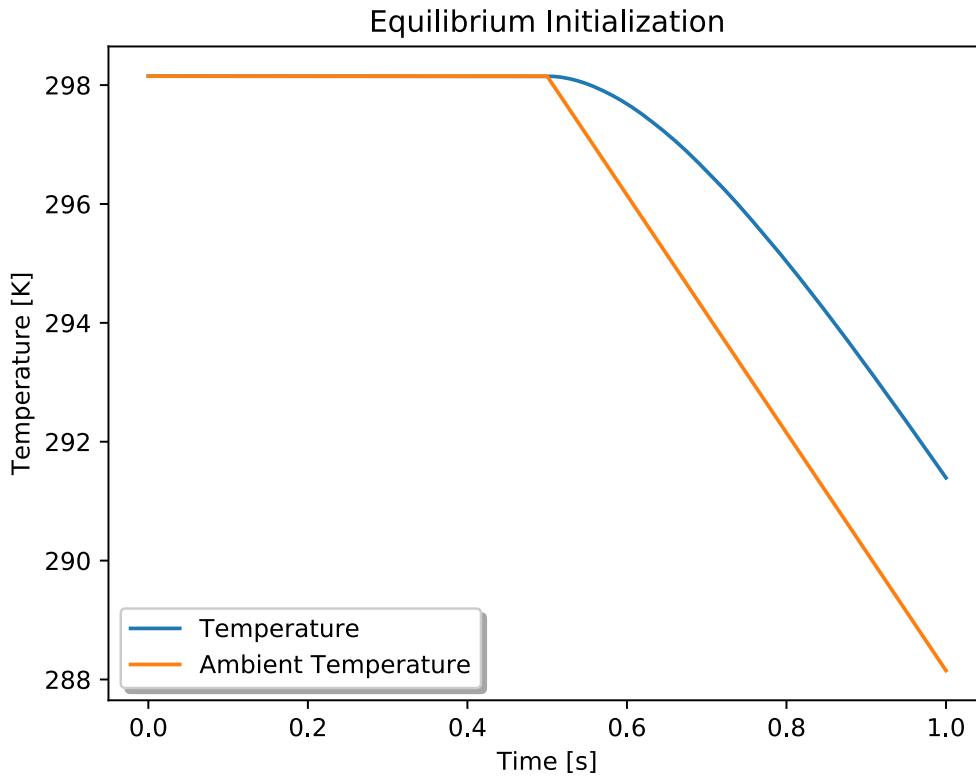
```

parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
parameter Area A=1.0 "Surface area";
parameter Mass m=0.1 "Mass of thermal capacitance";
parameter SpecificHeat c_p=1.2 "Specific heat";

Temperature T_inf "Ambient temperature";
Temperature T "Temperature";
initial equation
  der(T) = 0 "Steady state initial conditions";
equation
  if time<=0.5 then
    T_inf = 298.15 "Constant temperature when time<=0.5";
  else
    T_inf = 298.15-20*(time-0.5) "Otherwise, decreasing";
  end if;
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingSteadyThenDynamic;

```

The only thing we've changed is the initial equation. Instead of starting our system at some fixed temperature, we start it at a temperature such that the change in temperature (at least initially, prior to our disturbance) is zero. Now the temperature response no longer includes any initial transient and we can focus only on the response to the disturbance:



Compactness

One issue with `if` statements is that they can make relatively simple changes in behavior appear quite complicated. There are a couple of alternative constructs we can use to get the same behavior with fewer lines of code.

The first approach is to use an **if expression**. Whereas an **if** statement includes “branches” containing equations, an **if** expression has branches that contain only expressions. Furthermore, the syntax for an **if** expression is also less verbose. If we had chosen to use an **if** expression our **equation** could have been simplified to:

```
equation
  T_inf = 298.15 - (if time<0.5 then 0 else 20*(time-0.5));
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
```

Alternatively, we could use one of the many built-in Modelica functions, like **max**, to represent the change in the ambient temperature, *e.g.*,

```
equation
  T_inf = 298.15 - max(0, 20*(time-0.5));
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
```

Events

We’ve seen several ways to express the fact that there is an abrupt change in the behavior of our system. But it’s important to point out that we are not just describing a change in the ambient temperature, we are also specifying **when** it changes. This a subtle, but very important, point.

Consider the last example, where our system began in an equilibrium state. At the start of the simulation, there are no significant dynamics. Since nothing is really changing in the system, the integrator is unlikely to accumulate significant integration error. So, in order to minimize the amount of time required to complete the simulation, variable time step integrators will, in such circumstances, increase their step size.

There is, however, a risk in doing this. The risk is that the integrator may get “blind-sided” by a sudden disturbance in the system. If such a disturbance were to occur, the integrator’s assumptions that a large step will not lead to significant integration error would not be true.

The question then becomes, how can the integrator *know* when it can take a large time step and when it cannot. Typically, these integration schemes use a kind of trial and error approach. They try to take large step and then estimate the amount of error introduced by that step. If it is less than some threshold, then they accept the state (or perhaps try a larger step). If, on the other hand, the step introduces too much error, then they try a smaller step. But they cannot know how small a step will be required to get under the error threshold, which means they will continue to blindly try smaller and smaller steps.

But Modelica is about much more than integrating the underlying system. Modelica compilers study the **structure** of the problem. In all of our examples, the compiler can see that there is a distinct change in behavior. Not only that, it can see that this change in behavior is a time event, *i.e.*, an event whose time is known *a priori* without any knowledge of the solution trajectory.

So, what a Modelica compiler will do is inform the underlying integrator that there will be an abrupt change in behavior at 0.5 seconds and it will instruct the integrator to simply integrate exactly up to that point and no further. As a result, the abrupt change never occurs **within a time step**. Instead, the integrator will simply restart on the other side of the event. This completely avoids the blind searching for the cutoff time that minimizes the error in the step. Instead, the integrator will integrate right up to that point automatically and then restart after that point.

This is one of many examples of features in Modelica that optimize the way a simulation is carried out. A more detailed discussion of this kind of handling can be found in the upcoming section on [Events](#) (page 82). In the coming sections, we’ll also see more complex examples of events that depend on the solution variables.

Bouncing Ball

Modeling a Bouncing Ball

In the [previous example](#) (page 42), we saw how some events are related to time. These so-called “time events” are just one type of event. In this section, we’ll examine the other type of event, the state event. A state event is an event that depends on the solution trajectory.

State events are much more complicated to handle. Unlike time events, where the time of the event is known *a priori*, a state event depends on the solution trajectory. So we cannot entirely avoid the “searching” for the point at which the event occurs.

To see a state event in action, let us consider the behavior of a bouncing ball bouncing on a flat horizontal surface. When the ball is above the surface, it accelerates due to gravitational forces. When the ball eventually comes in contact with the surface, it bounces off the surface according to the following relationship:

$$v_{\text{final}} = -ev_{\text{initial}}$$

where v_{final} is the (vertical) velocity of the ball immediately after contact with the surface, v_{initial} is the velocity prior to contact and e is the coefficient of restitution, which is a measure of the fraction of momentum retained by the ball after the collision.

Bringing all this together in Modelica might look something like this:

```
model BouncingBall "The 'classic' bouncing ball model"
  type Height=Real(unit="m");
  type Velocity=Real(unit="m/s");
  parameter Real e=0.8 "Coefficient of restitution";
  parameter Height h0=1.0 "Initial height";
  Height h "Height";
  Velocity v(start=0.0, fixed=true) "Velocity";
initial equation
  h = h0;
equation
  v = der(h);
  der(v) = -9.81;
  when h<0 then
    reinit(v, -e*pre(v));
  end when;
end BouncingBall;
```

In this example, we use the parameter `h0` to specify the initial height of the ball off the surface and the parameter `e` to specify the coefficient of restitution. The variables `h` and `v` represent the height and vertical velocity, respectively.

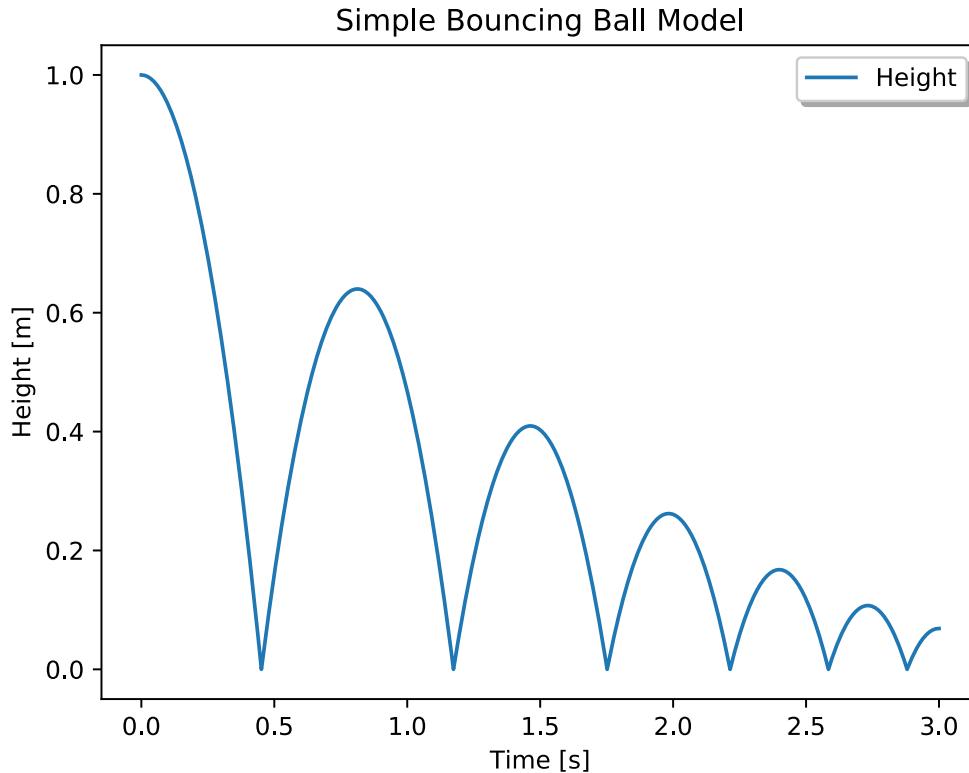
What makes this example interesting are the equations. Specifically, the existence of a `when` statement:

```
equation
  v = der(h);
  der(v) = -9.81;
  when h<0 then
    reinit(v, -e*pre(v));
  end when;
```

A `when` statement is composed of two parts. The first part is a conditional expression that indicates the moment the event takes place. In this case, the event will take place “when” the height, `h`, first drops below 0. The second part of the `when` statement is what happens when the event occurs. In this case, the value of `v` is re-initialized via the `reinit` operator. The `reinit` operator allows us to specify a new initial condition for a state. Conceptually, you can think of `reinit` as being like an `initial equation` inserted in the middle of a simulation. But it only changes one variable and it always sets it explicitly (*i.e.*, it isn’t as flexible as an `initial equation`). In this case, the `reinit` statement will reinitialize the

value of v to be in the opposite direction of the value of v before the collision, represented by `pre(v)`, and scaled by the factor e .

Assuming that h_0 has a positive value, the relentless pull of gravity ensures that the ball will eventually hit the surface. Running the simulation for the case where h_0 is 1.0, we see the following behavior from this model:

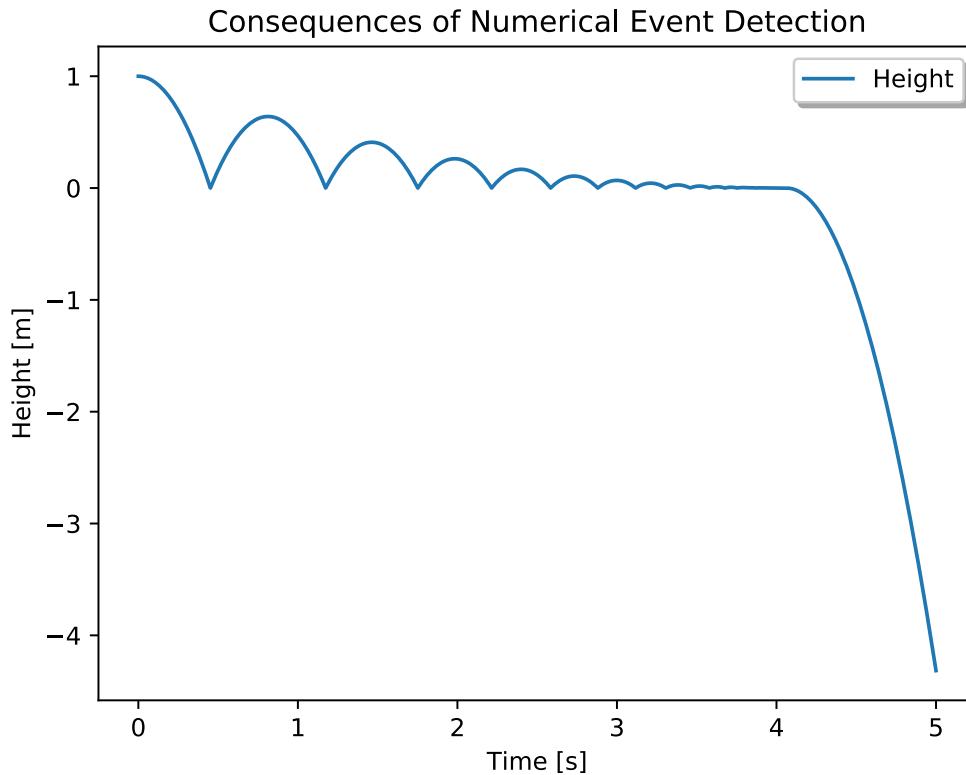


In this plot, we see that at around 0.48 seconds, the first impact with the surface occurs. This occurs because the condition $h < 0$ first becomes true at that moment. Note that what makes this a state event (unlike our example in [previous cooling examples](#) (page 42)) is the fact that this conditional expression references variables other than `time`.

As such, the simulation proceeds assuming the ball is in free fall until it identifies a solution trajectory where the value of the conditional $h < 0$ changes during a time step. When such a step occurs, the solver must determine the precise time when the value of the conditional expression becomes true. Once that time has been identified, it computes the state of the system at that time, processes the statements within the `when` statement (*e.g.* any `reinit` statements) that affect the state of the system and then `restarts` the integration starting from these computed states. In the case of the bouncing ball, the `reinit` statement is used to compute a new post-collision value for v that sends the ball (initially) upward again.

But it is important to keep in mind that, in general, the solutions for most Modelica models are derived using numerical methods. As we shall see shortly, this has some profound implications when we consider discrete behavior. This is because at the heart of all events (time or state events) are conditional expressions, like $h < 0$ from our current example.

The implications become clear if we simulate our bouncing ball a bit longer. In that case, most Modelica tools will provide a solution like this:



It should be immediately obvious when looking at this trajectory that something has gone wrong. But what?

Numerical Precision

The answer, as we hinted at before, lies in the numerical handling of the when condition $h < 0$. More specifically, what do we do if we start a state extremely close to an event? Because of numerical imprecision, we do not know whether we are starting our step right after an event has just occurred or whether we are starting a step where an event is just about to occur.

To address this problem, we must introduce a certain amount of hysteresis (dead-banding). What this means in this case is that once the condition $h < 0$ has become true, we have to get “far enough” away from the condition before we allow the event to happen again. In other words, the event happens whenever h is less than zero. But before we can trigger the event again we require that h must first become greater than some ϵ . In other words, it is not simply enough that h becomes greater than zero, h must become greater than ϵ (where ϵ is determined by the solver by examining various scaling factors).

The problem in the previous simulations is that each time the ball bounces, the peak value of h goes down a little bit. By peak value, we mean the value of h when the ball first begins to fall again. Eventually, the peak value of h isn’t enough to exceed the critical value of ϵ . This, in turn, means that the when statement never fires and the reinit statement will never again reset v . As a result, the ball continues, indefinitely, in free fall.

So this raises the obvious question of how to achieve the behavior we truly intended (which is that the ball never drops below the surface). For that, we have to make a few minor changes to our model as follows:

```
model StableBouncingBall
  "The 'classic' bouncing ball model with numerical tolerances"
```

```

type Height=Real(unit="m");
type Velocity=Real(unit="m/s");
parameter Real e=0.8 "Coefficient of restitution";
parameter Height h0=1.0 "Initial height";
constant Height eps=1e-3 "Small height";
Boolean done "Flag when to turn off gravity";
Height h "Height";
Velocity v(start=0.0, fixed=true) "Velocity";
initial equation
  h = h0;
  done = false;
equation
  v = der(h);
  der(v) = if done then 0 else -9.81;
when {h<0,h<-eps} then
  done = h<-eps;
  reinit(v, -e*(if h<-eps then 0 else pre(v)));
end when;
end StableBouncingBall;

```

It should be noted that there are many ways to solve this problem. The solution presented here is only one of them. In this approach, we have effectively created two surfaces. One at a height of 0 and the other at a height of $-eps$ (just below 0). When the ball is bouncing “normally” it will only trigger the first condition in our `when` statement. If, however, the ball does not rebound high enough after contact and “falls through” the first surface, we detect that (and the fact that it has fallen through) and set the `done` flag. The effect of the `done` flag is to effectively turn off gravity.

Note the syntax of the `when` statement in this case:

```

when {h<0,h<-eps} then
  done = h<-eps;
  reinit(v, -e*(if h<-eps then 0 else pre(v)));
end when;

```

In particular, note that it doesn’t have just one conditional expression, **but two**. More specifically, it actually has a vector of conditional expressions. We’ll introduce *Vectors and Arrays* (page 87) later in the book, but for now it is just important to point out that in this chapter we have shown that a `when` can include either a scalar conditional expression or a vector of conditional expressions.

If a `when` statement includes a vector of conditionals, then the statements of the `when` statement will be triggered when **any** conditional expression in the vector **becomes true**. Note the grammar of this explanation carefully. It is very common for people to read Modelica code like this:

```

when {a>0, b>0} then
  ...
end when;

```

as “when a is greater than zero **or** b is greater than zero”. But it is **very important** not to make the very common mistake of misinterpreting this to mean that the following two `when` statements are equivalent:

```

when {a>0, b>0} then
  ...
end when;

when a>0 or b>0 then
  ...
end when;

```

These are not equivalent. To understand the difference, let’s change the conditional expressions as follows:

```
when {time>1, time>2} then
  ...
end when;

when time>1 or time>2 then
  ...
end when;
```

Remember our original statement that the vector notation for `when` statements means that the statements in the `when` statement are triggered when **any** condition becomes true. Assuming we run a simulation that starts at `time=0` and runs until `time=3`, then the `when` statement:

```
when {time>1, time>2} then
  ...
end when;
```

will be triggered **twice**. Once when `time>1` becomes true and the other when `time>2` becomes true. In contrast, in this case:

```
when time>1 or time>2 then
  ...
end when;
```

there is only a **single** conditional expression and it becomes true **only once** (when `time>1` becomes true...and stays true). The `or` operator essentially masks the second conditional, `time>2`, such that it may as well not even be present in this particular case. In other words, this conditional only **becomes true** once. As a result, the statements inside the `when` statement are only triggered once.

The key thing to remember is that for `when` statements, a vector of conditionals means **any, not or**. Furthermore, the statements are only active at the instant when the conditional **becomes true**. The implications of this last statement will be discussed in greater details later in this chapter when we talk about the important differences between *if vs. when* (page 86).

State Event Handling

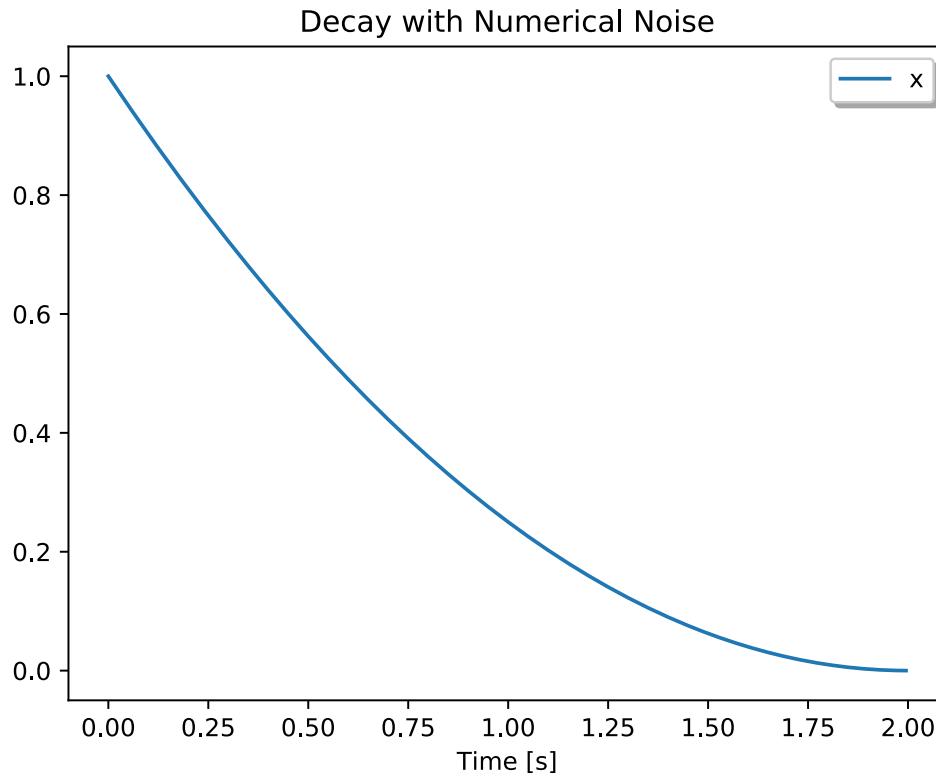
Now that we have already introduced both *time events* (page 42) and *state events* (page 47), let's examine some important complications associated with state events. Surprisingly, these complications can be introduced by even simple models.

Basic Decay Model

Consider the following almost trivial model:

```
model Decay1
  Real x;
initial equation
  x = 1;
equation
  der(x) = -sqrt(x);
end Decay1;
```

If we attempt to simulate this model for 5 seconds, we find that the simulation terminates after about 2 seconds with the following trajectory:



Again, numerical issues creep in. Even though mathematically it should not be possible for the value of x to drop below zero, using numerical integration techniques it is possible for small amounts of error to creep in and drive x below zero. When that happens, the `sqrt(x)` expression generates a floating point exception and the simulation terminates.

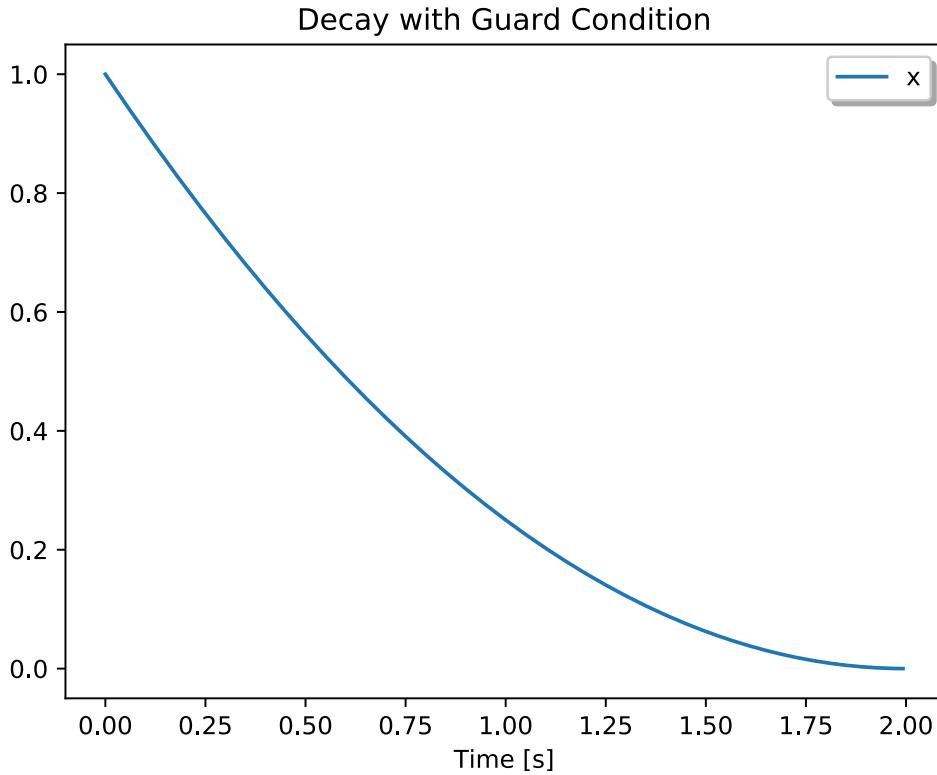
Guard Expressions

To prevent this, we might introduce an `if` expression to guard against evaluating the square root of a negative number, like this:

```
model Decay2
  Real x;
initial equation
  x = 1;
equation
  der(x) = if x>=0 then -sqrt(x) else 0;
end Decay2;
```

Simulating this model we get the following trajectory¹⁰:

¹⁰ This model will not always fail. The failure depends on how much integration error is introduced and this, in turn, depends on the numerical tolerances used.



Again, the simulation fails. But why? It fails for the same reason, a numerical exception that results from taking the square root of a negative number.

Most people are quite puzzled when they see an error message about a floating point exception like this (or, for example division by zero) after they have introduced a guard expression as we have done. They naturally assume that there is no way that `sqrt(x)` can be evaluated if `x` is less than zero. **But this assumption is incorrect.**

Events and Conditional Expressions

The Role of Events in Behavior

Given the `if` expression:

```
der(x) = if x>=0 then sqrt(x) else 0;
```

it is entirely possible that `sqrt` will be called with a negative argument. The reason is related to the fact that this is a state event. Remember, the time at which a *time event* will occur is known in advance. But this is not the case for a state event. In order to determine when the event will occur, we have to search the solution trajectory to see when the condition (*e.g.*, $x \geq 0$ becomes false).

The important thing to understand is that **until the event occurs, the behavior doesn't change**. In other words, the two sides of this `if` expression represent two types of behavior, `der(x)=sqrt(x)` and `der(x)=0`. Since `x` is initially greater than zero, the initial behavior is `der(x)=sqrt(x)`. The solver will continue using this equation until it has determined the time of the event represented by $x \geq 0$. In order to determine the time of that event, **it must go past the point where the value of the conditional expression changes**. This means that while attempting to determine precisely when

the condition `x>=0` changes from true to false, it will continue to use the equation `der(x)=sqrt(x)` even though `x` is negative.

Most users initially assume that each time `der(x)` is evaluated, the `if` expression is evaluated (specifically the conditional expression in the `if` expression). Hopefully the previous paragraph has made it clear that this is not the case.

This time spent trying and retrying integration steps can be saved thanks to the fact that Modelica can extract a so-called “zero crossing” function from the `if` expression. This function is called a zero crossing function because it is normally constructed to have a root at the point where the event will occur. For example, if we had the following `if` expression:

```
y = if a>b then 1 else 0;
```

The zero crossing function would be $a - b$. This function is chosen because it changes from positive to negative precisely at the point where `a>b`.

Recall our previous equation:

```
der(x) = if x>=0 then sqrt(x) else 0;
```

In this case, the zero crossing function is simply x since the event occurs when x itself crosses zero.

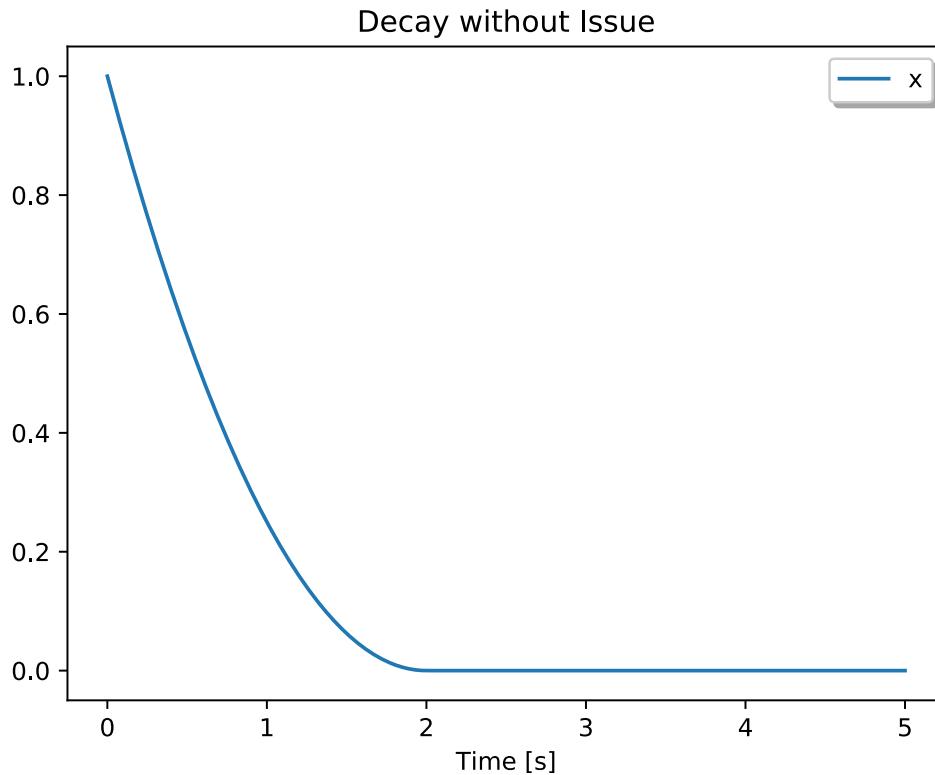
The Modelica compiler collects all the zero crossing functions in the model for the integrator to use. During integration, the integrator checks to see if any of the zero crossing functions have changed sign. If they have, it uses the solution it computed during that step to interpolate the zero crossing function to find the location, in time, of the root of the zero crossing function and this is the point in time where the event occurs. This process is much more efficient because the root finding algorithms have more information to help them identify to location of the root (information like the derivative of the zero crossing function) and evaluation is very cheap because it doesn't involve taking additional integration steps, only evaluating the interpolation functions from the triggering integration step.

Event Suppression

But after all this, it still isn't clear how to avoid the problems we saw in the `Decay1` and `Decay2` models. The answer is a special operator called `noEvent`. The `noEvent` operator suppresses this special event handling. Instead, it does what most users expected would happen in the first place, which is to evaluate the conditional expression for every value of `x`. We can see the `noEvent` operator in action in the following model:

```
model Decay3
  Real x;
initial equation
  x = 1;
equation
  der(x) = if noEvent(x>=0) then -sqrt(x) else 0;
end Decay3;
```

and the results can be seen here:



Now the simulation completes without any problem. This is because the use of `noEvent` ensures that `sqrt(x)` is never called with a negative value of `x`.

It might seem strange that we have to explicitly include the `noEvent` operator in order to get what we consider the most intuitive behavior. Why not make the default behavior the most intuitive one? The answer is performance. Using conditional expressions to generate events improves the performance of the simulations by giving the solver clues about when to expect abrupt changes in behavior. Most of the time, this approach doesn't cause any problem. The examples we have presented in this chapter were designed to highlight this issue, but they are not representative of most cases. For this reason, `noEvent` is not the default, but must be used explicitly. It should be noted that the `noEvent` operator should only be used when there is a smooth transition in behavior, otherwise it can create performance issues.

Chattering

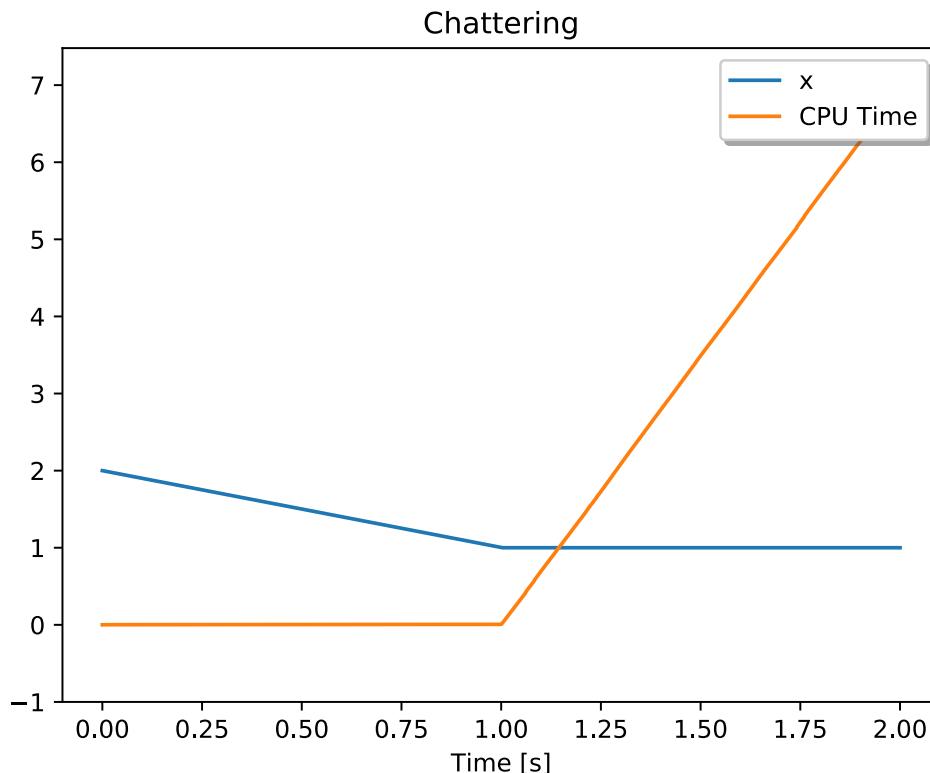
There is a common effect known as “chattering” that you will run into sooner or later with Modelica. Consider the following model:

```
model WithChatter
  Real x;
initial equation
  x = 2;
equation
  der(x) = if x>=1 then -1 else 1;
end WithChatter;
```

Effectively, the behavior of this model is that for any initial value of `x`, it will progress linearly toward 1. Mathematically speaking, once the value of `x` gets to 1, it should just stay there. This is because any deviation away from 1, either greater than or less than, will immediately cause it to go back to 1.

But we will not be solving these equations in a strictly mathematical way. We'll be using floating point representations and using numerical integrators. As such, we have limited precision and integration error to contend with. The net effect will be that the trajectory of x will not remain exactly 1 but will deviate slightly above and below. Each time this happens, it will generate an event.

Simulating this model gives us the following results:

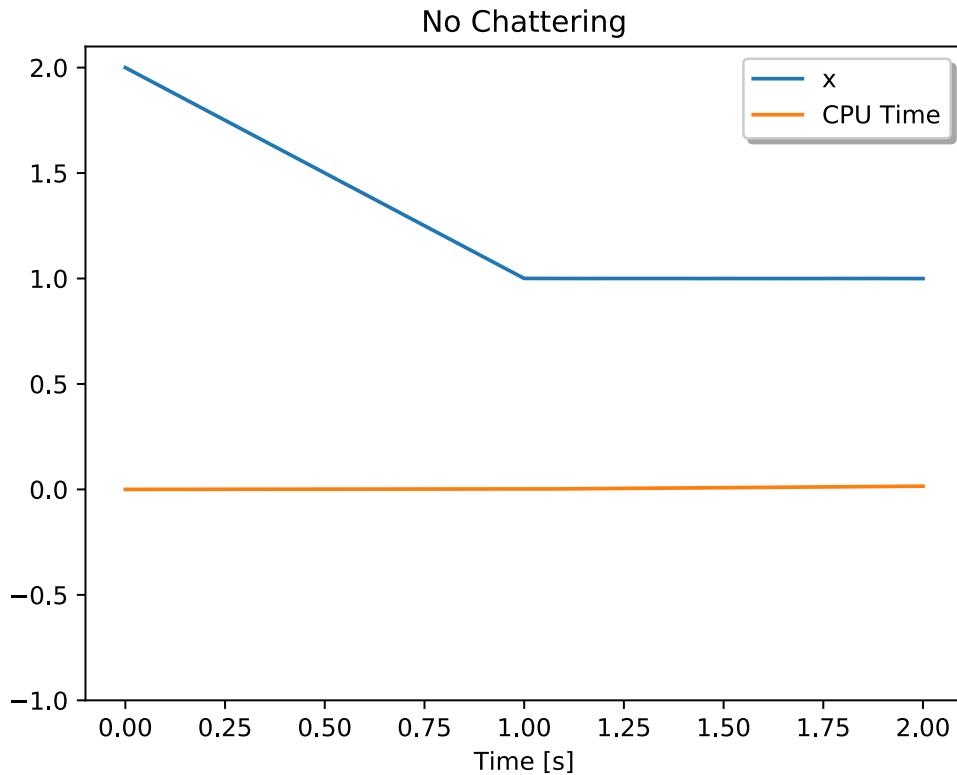


This kind of model can introduce an effect known as “chattering”. Chattering is simply the degradation in simulation performance due to a large number of events occurring that artificially shorten the time steps taken by the solver. The impact on simulation performance is clear if we look at the CPU time taken during the simulation. It starts to rise dramatically once x is close to 1. This is because behind the scenes the events are causing lots of very small time steps which dramatically increases the number of computations being performed. The important thing about the `WithChatter` example is that it has a seemingly obvious mathematical solution but still suffers from degraded simulation performance because of the high frequency of events.

This is another case where the `noEvent` operator can help us out. We can suppress the events being generated by the conditional expression by using the `noEvent` operator as follows:

```
model WithoutChatter
  Real x;
initial equation
  x = 2;
equation
  der(x) = if noEvent(x>=1) then -1 else 1;
end WithoutChatter;
```

In doing so, we will get approximately the same solution, but with better simulation performance:



Note how there is no perceptible change in the slope of the CPU usage before and after x arrives at 1. Contrast this with the significant difference seen in the case of `WithChatter`.

In reality, equations like this are uncommon. In this case, we've used an extreme case in an attempt to clearly show the impact of chattering. The behavior being described here is not particularly realistic or physical. In this case, we've exaggerated the effect to clearly demonstrate the impact on simulation performance.

More typical examples of chattering in the real world will feature a conditional expression that sits at some stable point (`Decay2` is a good example). In such cases, chattering occurs because the system tends to naturally settle at or near the point where the conditional expression occurs. But because of precision and numerical considerations, the event associated with the conditional expression is frequently triggered. The effect is exacerbated in cases where there are many components in the system all sitting at or near such equilibrium points.

Speed vs. Accuracy

Hopefully the discussion so far has made it clear why it is necessary to suppress events in some cases. But one might reasonably ask, why not skip events and just evaluate conditional expressions all the time? So let's take some time to explore this question and explain why, on the whole, associating events with conditional expressions is very good idea¹¹.

Without event detection, the integrator will simply step right over events. When this happens, the integrator will miss important changes in behavior and this will have a significant impact on the accuracy of the simulation. This is because the accuracy of most integration routines is based on assumptions

¹¹ A special thanks to Lionel Belmon for challenging my original discussion and identifying several unsubstantiated assumptions on my part. As a result, this explanation is much better and includes results to support the conclusions drawn.

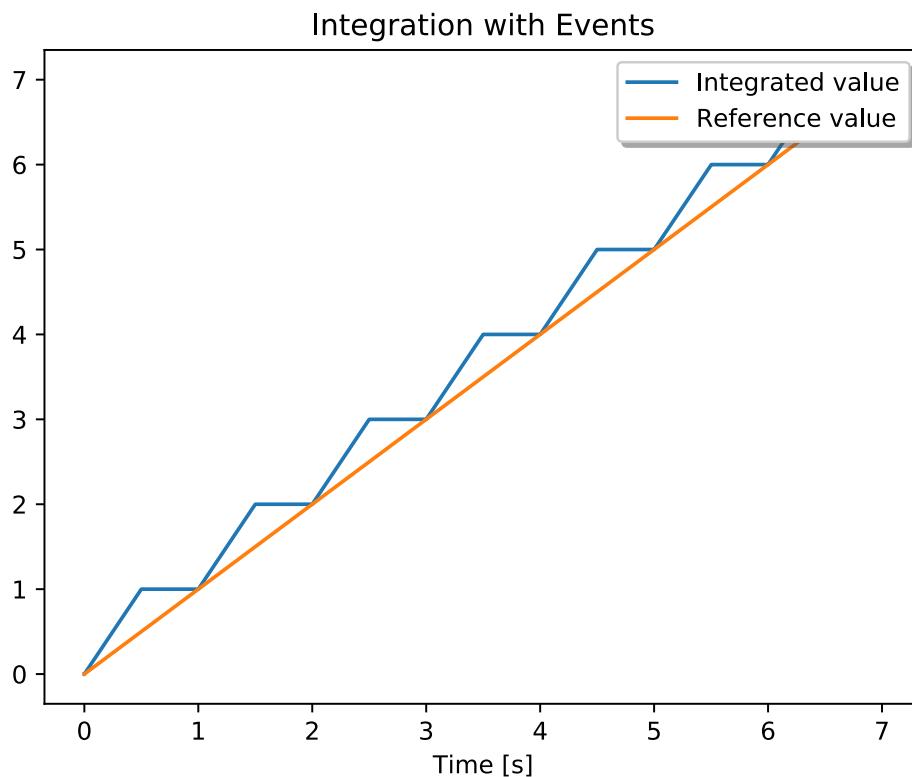
about the continuity of the underlying function and its derivatives. If those assumptions are violated, we need to let the integration routines know so they can account these changes in behavior.

This is where events come in. They force the integration to stop at the point where a behavior change occurs and then restart again after the behavior change has occurred. The result is greater accuracy but at the cost of slower simulations. Let's look at a concrete example. Consider the following simple Modelica model:

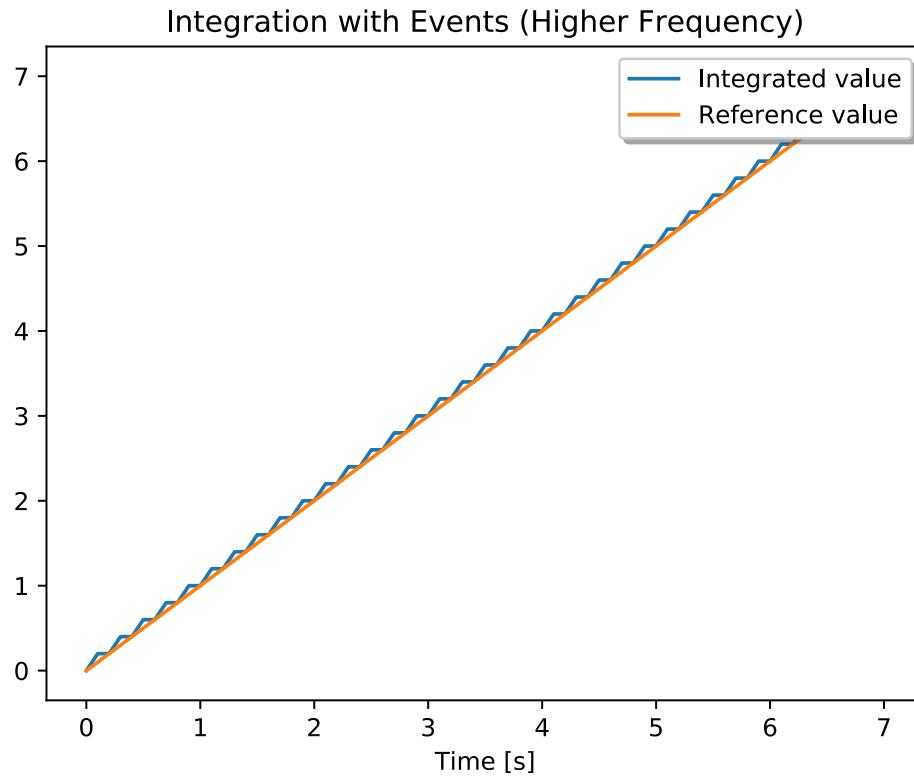
```
model WithEvents "Integrate with events"
  parameter Real freq = 1.0;
  Real x(start=0);
  Real y = time;
equation
  der(x) = if sin(2*Modelica.Constants.pi*freq*time)>0 then 2.0 else 0.0;
end WithEvents;
```

Looking at this system, we can see that half the time the derivative of x will be 2 and the other half of the time the derivative of x will be 0. So over each of these cycles, the average derivative of x should be 1. This means at the end of each cycle, x and y should be equal.

If we simulate the `WithEvents` model, we get the following results:



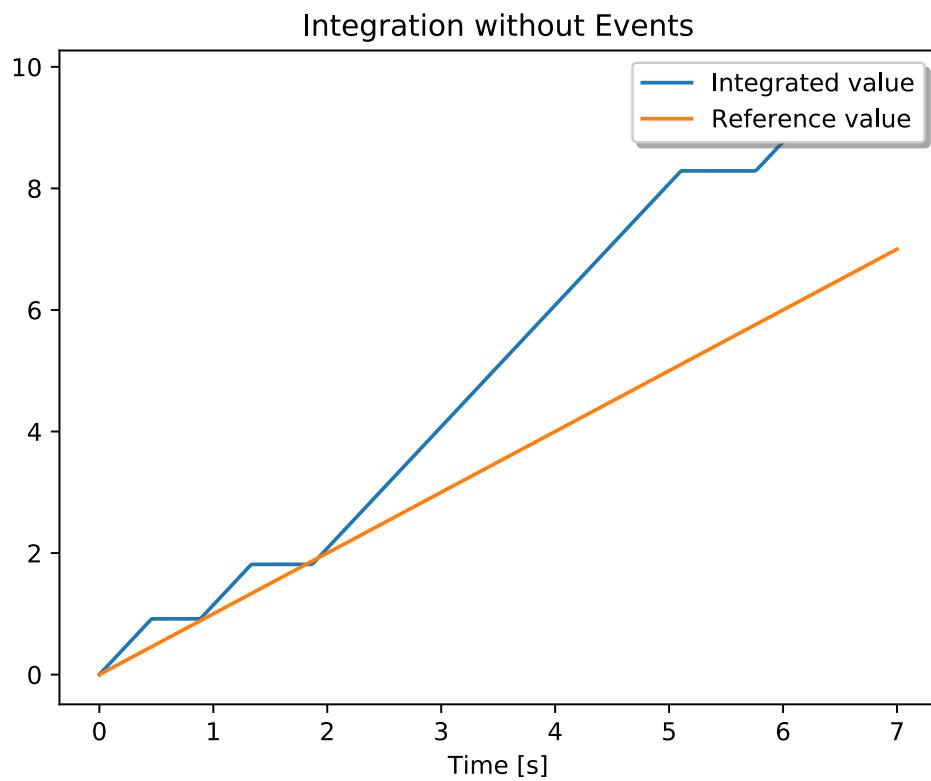
Note how, at the end of each cycle, the trajectories of x and y meet. This is a visual indication of the accuracy of the underlying integration. Even if we increase the frequency of the underlying cycle, we see that this property holds true:



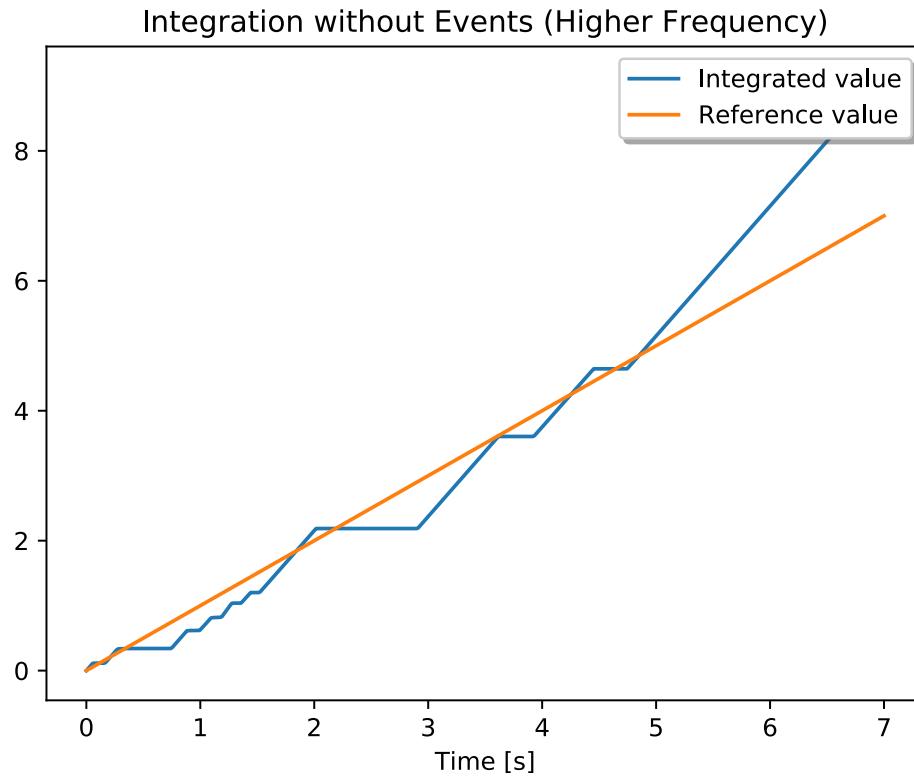
However, now let us consider the case where we use exactly the same integration parameters but suppress events by using the `noEvents` operator as follows:

```
model WithNoEvents "Integrate without events"
  parameter Real freq = 1.0;
  Real x(start=0);
  Real y = time;
equation
  der(x) = noEvent(if sin(2*Modelica.Constants.pi*freq*time)>0 then 2.0 else 0.0);
end WithNoEvents;
```

In this case, the integrator is blind to the changes in behavior. It does its best to integrate accurately but without explicit knowledge of where the behavior changes occur, it will blindly continue using the wrong value of the derivative and extrapolate well beyond the change in behavior. If we simulate the `WithNoEvents` model, using the same integrator settings, we can see how significantly different our results will be:



Note how quickly the integrator introduces some pretty significant error.



The integration settings used in these examples were chosen to demonstrate the impact that the `noEvent` operator can have on accuracy. However, the settings were admittedly chosen to accentuate these differences. Using more typical settings, the differences in the results probably would not have been so dramatic. Furthermore, the impact of using `noEvent` are impossible to predict or quantify since they will vary significantly from one solver to another. But the underlying point is clear, using the `noEvent` operator can have a significant impact on accuracy simulation results.

Switched RLC Circuit

In this section, we'll present another model, like the heat transfer model presented *earlier in this chapter* (page 42), that contains time events. In this case, we'll show how we can simulate the behavior of a switch in the context of an RLC circuit like *the one presented in the first chapter* (page 10). From the examples presented so far, nothing in this model should come as a surprise. The main motivation for this example is simply to present time events in the context of an electrical model.

Our switched RLC circuit model is as follows:

```
model SwitchedRLC "An RLC circuit with a switch"
  type Voltage=Real(unit="V");
  type Current=Real(unit="A");
  type Resistance=Real(unit="Ohm");
  type Capacitance=Real(unit="F");
  type Inductance=Real(unit="H");
  parameter Voltage Vb=24 "Battery voltage";
  parameter Inductance L = 1;
  parameter Resistance R = 100;
  parameter Capacitance C = 1e-3;
  Voltage Vs;
  Voltage V;
```

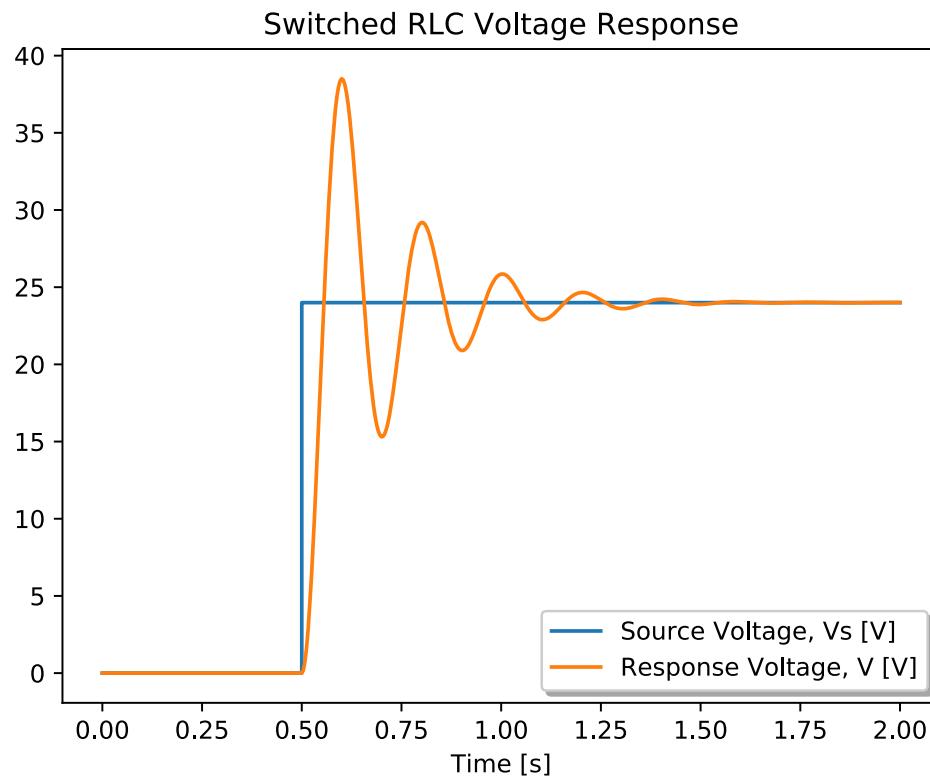
```

Current i_L;
Current i_R;
Current i_C;
equation
  Vs = if time>0.5 then Vb else 0;
  i_R = V/R;
  i_C = C*der(V);
  i_L=i_R+i_C;
  L*der(i_L) = (Vs-V);
end SwitchedRLC;

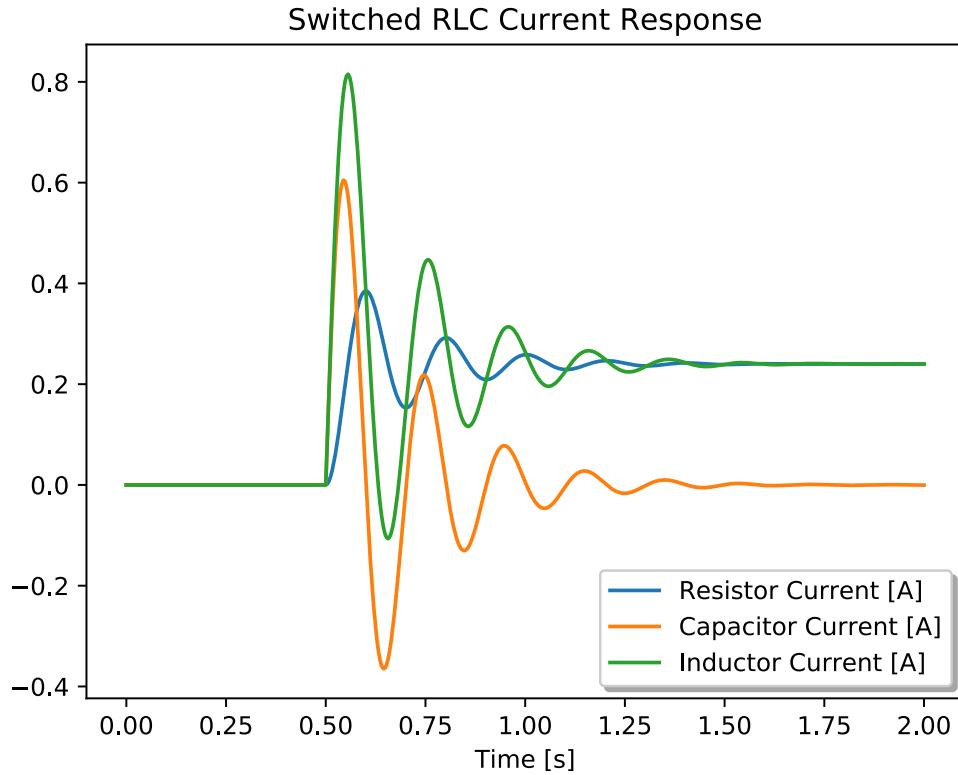
```

The time event in this model is introduced via an `if` expression. The fact that the only time-varying variable in the conditional expression is `time` means that this will trigger a time event and that the underlying numerical solver will be able to determine *a priori* when the event will occur while integrating the underlying equations.

We can see the voltage response of this model to the switched supply voltage in the following plot:



Furthermore, we can see the current response for inductor, resistor and capacitor components in this plot:



Hopefully by this point, the basic mechanisms for generating events and disturbances seem intuitive and familiar.

Speed Measurement

Baseline System

There are many applications where we need to model the interaction between continuous behavior and discrete behavior. For this section, we'll look at techniques used to measure the speed of a rotating shaft. For our discussion here, we will reuse the *mechanical example* (page 13) we discussed previously in our discussion of *Basic Equations* (page 1):

```
model SecondOrderSystem "A second order rotational system"
  type Angle=Real(unit="rad");
  type AngularVelocity=Real(unit="rad/s");
  type Inertia=Real(unit="kg.m2");
  type Stiffness=Real(unit="N.m/rad");
  type Damping=Real(unit="N.m.s/rad");
  parameter Inertia J1=0.4 "Moment of inertia for inertia 1";
  parameter Inertia J2=1.0 "Moment of inertia for inertia 2";
  parameter Stiffness c1=11 "Spring constant for spring 1";
  parameter Stiffness c2=5 "Spring constant for spring 2";
  parameter Damping d1=0.2 "Damping for damper 1";
  parameter Damping d2=1.0 "Damping for damper 2";
  Angle phi1 "Angle for inertia 1";
  Angle phi2 "Angle for inertia 2";
  AngularVelocity omega1 "Velocity of inertia 1";
  AngularVelocity omega2 "Velocity of inertia 2";
initial equation
```

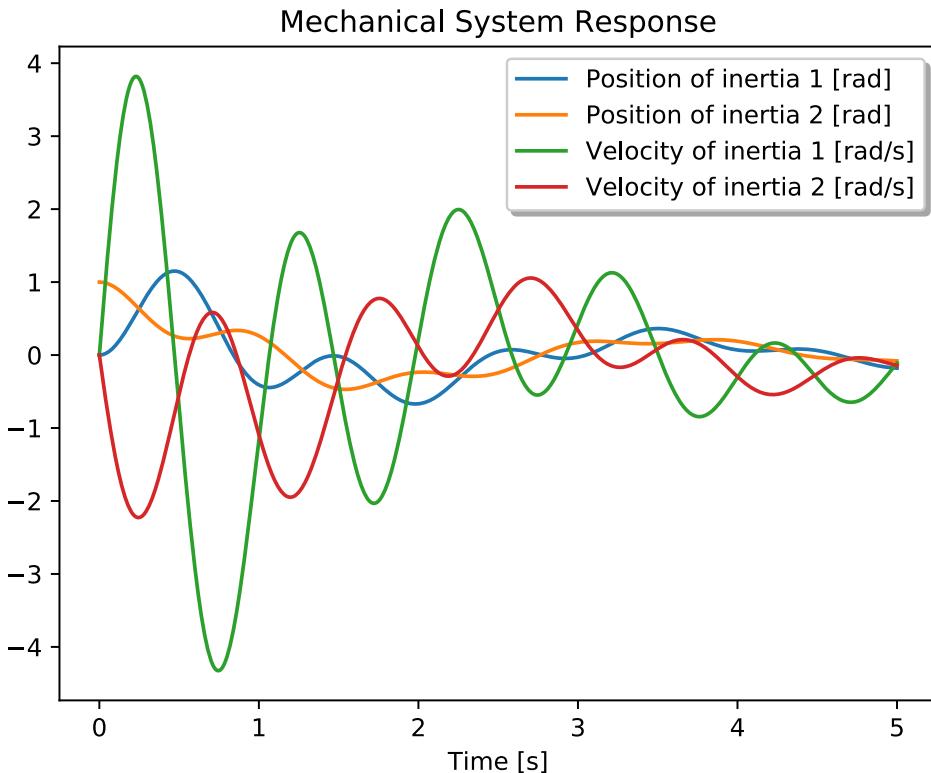
```

phi1 = 0;
phi2 = 1;
omega1 = 0;
omega2 = 0;
equation
  // Equations for inertia 1
  omega1 = der(phi1);
  J1*der(omega1) = c1*(phi2-phi1)+d1*der(phi2-phi1);
  // Equations for inertia 2
  omega2 = der(phi2);
  J2*der(omega2) = c2*(phi1-phi2)+d2*der(phi1-phi2)-c2*phi2-d2*der(phi2);
end SecondOrderSystem;

```

We will reuse this model by adding an `extends` clause to our model. This essentially imports everything from the model we are extending from. We'll talk more about the `extends` clause later when we discuss *Packages* (page 151). For now, just think of it as copying the contents of another model into the current model.

Recall the solution for the `SecondOrderSystem` model looks like this:



In this case, we are simply plotting the solution that we computed. But in a real system, we can't directly know the rotational velocity of a shaft. Instead, we have to measure it. But measurement introduces error and each measurement technique introduces different kinds of errors. In this section, we'll look at how we can model different kinds of measurement techniques.

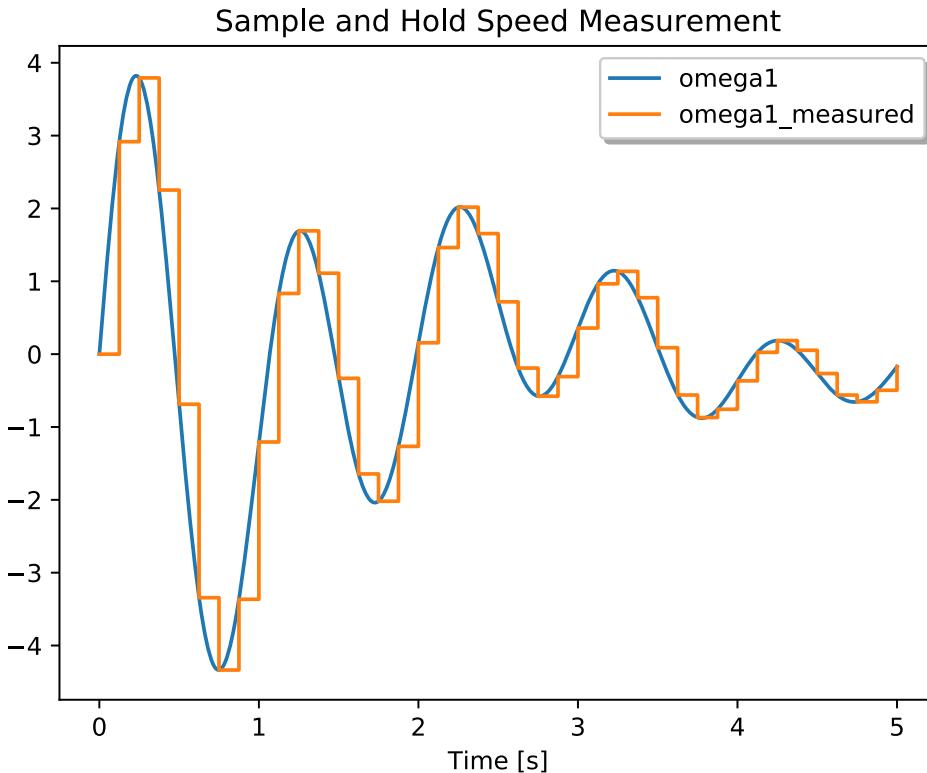
Sample and Hold

The first type of measurement we will examine is a sample and hold approach to measurement. Some speed sensors have circuits for measuring the rotational speed of the system. But instead of providing a continuous value for the speed, they sample it at a given point in time and then store it somewhere. This is called “sample and hold”. The following model demonstrates how to implement a sample and hold approach to measuring the angular velocity `omega1`:

```
model SampleAndHold "Measure speed and hold"
  extends BasicEquations.RotationalSMD.SecondOrderSystem;
  parameter Real sample_time(unit="s")=0.125;
  discrete Real omega1_measured;
equation
  when sample(0,sample_time) then
    omega1_measured = omega1;
  end when;
end SampleAndHold;
```

Note the presence of the `discrete` qualifier in the declaration of `omega1_measured`. This special qualifier indicates that the specified variable does not have a continuous solution. Instead, the value of this variable will make (only) discrete jumps during the simulation. It is not required to include the `discrete` keyword but it is useful because it provides additional information about the intent of the model that the compiler can check (*e.g.*, making sure we never request the derivative of that variable).

Let us now examine the solution generated by this model:



The important thing to note in the solution is how the measured value is piecewise-constant. This is because the value of `omega1_measured` is set only when the `when` statement becomes active. The `sample` function is a special built-in function that first becomes true at the time indicated by the first argument

(0 in this case) and then at regular intervals after that. The duration of these regular intervals is indicated by the second argument (`sample_time` in this case).

Interval Measurement

In the previous example, we weren't actually making any estimates for the speed, we were simply reporting the value of the variable `omega1` only at specific times. In other words, at the moment that we sampled `omega1` our sample was completely accurate. But by "holding" our measured value (instead of continuing to track `omega1`), we introduced some artifact in the measurement.

In the remaining examples, we will focus on techniques used to estimate the speed of a rotating shaft. In these cases, we will never make direct use of the actual speed in our measurement. Instead, we will respond to events generated by the physical system and attempt to use these events to reconstruct an estimate of the rotational speed.

The events that we will be responding to are generated by the discrete elements attached to the spinning shaft. For example, a typical way to produce these events is to use a "tooth wheel encoder". A tooth wheel encoder includes a gear on the rotating shaft. On either side of the gear, we place a light source and a light sensor. As the gear teeth pass in front of the light source, they block the light. The result is that the signal from the light sensor will include an approximate square wave. The leading edge of these square waves are the events we will be responding to.

The first estimation method we will examine computes the speed of the shaft by measuring the time interval that passes between events. Knowing that these events occur whenever the shaft has rotated by an angle of $\Delta\theta$, we can estimate the speed as:

$$\hat{\omega} = \frac{\Delta\theta}{\Delta t}$$

This technique for speed estimation can be represented in Modelica as:

```
model IntervalMeasure "Measure interval between teeth"
  extends BasicEquations.RotationalSMD.SecondOrderSystem;
  parameter Integer teeth=200;
  parameter Real tooth_angle(unit="rad")=2*3.14159/teeth;
  Real next_phi, prev_phi;
  Real last_time;
  Real omega1_measured;
initial equation
  next_phi = phi1+tooth_angle;
  prev_phi = phi1-tooth_angle;
  last_time = time;
equation
  when {phi1>=pre(next_phi),phi1<=pre(prev_phi)} then
    omega1_measured = tooth_angle/(time-pre(last_time));
    next_phi = phi1+tooth_angle;
    prev_phi = phi1-tooth_angle;
    last_time = time;
  end when;
end IntervalMeasure;
```

where `tooth_angle` represents $\Delta\theta$. Note how `tooth_angle` is not something the user needs to specify. Instead, the user specifies the number of teeth using the `teeth` parameter. The `tooth_angle` parameter is then computed using the value of `teeth` (note that while we have hand coded the value of π here, we'll learn how to avoid this later in the book when we talk about [Constants](#) (page 163)).

Let's take a close look at the `when` statement in this model:

```
equation
  when {phi1>=pre(next_phi),phi1<=pre(prev_phi)} then
    omega1_measured = tooth_angle/(time-pre(last_time));
    next_phi = phi1+tooth_angle;
```

```

prev_phi = phi1-tooth_angle;
last_time = time;
end when;

```

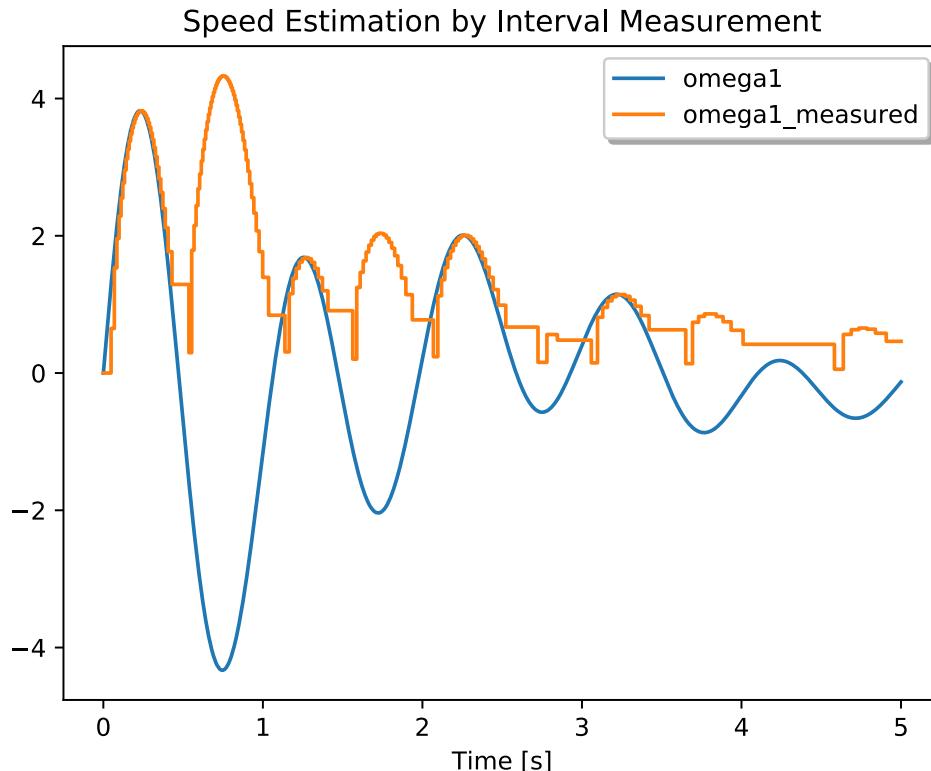
Here, we use the vector expression syntax used previously in the *Bouncing Ball* (page 47) example. Recall that the `when` statement becomes active if **any** of the conditions become true. In this case, the `when` statement becomes active if the angle, `phi1`, becomes greater than `next_phi` or less than `prev_phi`.

Another thing to note is the use of the `pre` operator throughout the `when` statement. When an event occurs in a model, there is a chance that the value of some variables may change discontinuously. During an event, while we are trying to resolve what values all the variables should have as a result of the event, the `pre` operator allows us to reference the value of a variable **prior** to the event. The `pre` operator is used in this model to refer to the previous (pre-event) values of `next_phi`, `prev_phi` and `last_time`. The `pre` operator is necessary because all of these variables are affected by the statements inside the `when` statement. So, for example, `last_time` (without the `pre` operator) refers to the value of `last_time` at the conclusion of the event while `pre(last_time)` refers to the value of `last_time` prior to any event occurring.

Use of the pre operator

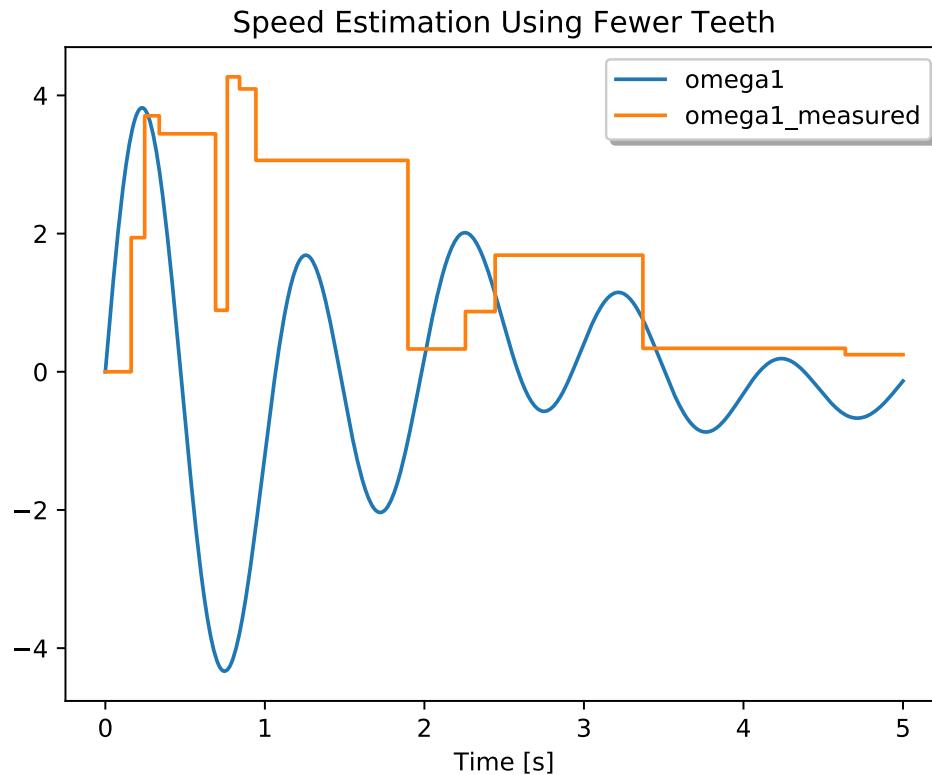
In general, if a variable changes as a result of a `when` statement becoming active, you **almost always** want to use the `pre` operator when referring to that variable in the conditional expression associated with the `when` statement (as we have done in the previous example). This makes it clear that you are responding to what was happening before the `when` statement was triggered.

Let's take a look at the speed estimates provided by this approach:

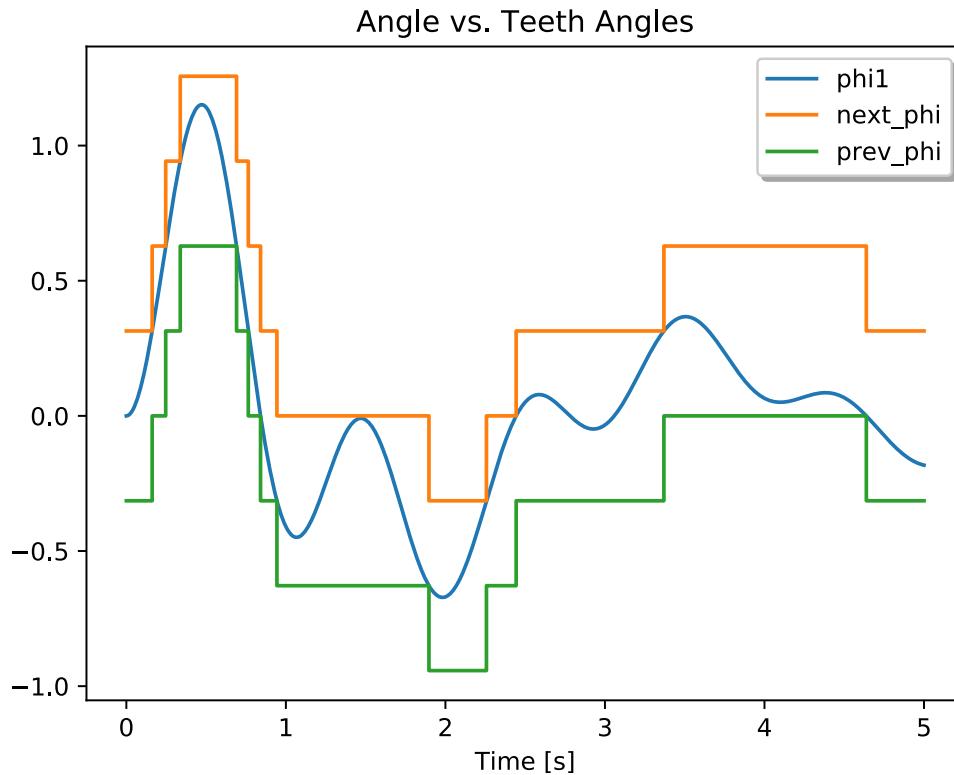


There are two important properties of this estimation algorithm that we can immediately see in these

results. The first is that the estimate is unsigned. In other words, we cannot tell from a device like a tooth wheel encoder which **direction** the shaft is rotating. Also, low speeds and changes in rotational direction can degrade the accuracy of the estimate significantly. The results are also very sensitive the number of teeth involved. If we were to reduce the number of teeth used in our encoder by setting **teeth** to 20, we'd get very different results.



To understand exactly why the measured signal is so inaccurate, it helps to consider the following plot which shows how the angle, `phi1` compares to the angles associated with the adjacent teeth, `next_phi` and `prev_phi`.



In this plot, we can clearly see how relatively low speeds and speed reversals create irregular events that introduce significant estimation error.

Pulse Counting

The interval measuring technique mentioned above requires hardware that can perform speed calculations on hardware interrupts. Another approach to estimating speed is the count how many events occur within a given (fixed) time interval and use that as an estimate of speed. Using this method, only the summation of events occurs when the events occur and the calculations are deferred to a regularly scheduled update.

Building on the previous examples in this section, the following seems like a natural way to create a model that implements this estimation technique:

```
model Counter "Count teeth in a given interval"
  extends BasicEquations.RotationalSMD.SecondOrderSystem;
  parameter Real sample_time(unit="s")=0.125;
  parameter Integer teeth=200;
  parameter Real tooth_angle(unit="rad")=2*3.14159/teeth;
  Real next_phi, prev_phi;
  Integer count;
  Real omega1_measured;
initial equation
  next_phi = phi1+tooth_angle;
  prev_phi = phi1-tooth_angle;
  count = 0;
equation
  when {phi1>=pre(next_phi),phi1<=pre(prev_phi)} then
    next_phi = phi1+tooth_angle;
    prev_phi = phi1-tooth_angle;
```

```

    count = pre(count) + 1;
end when;
when sample(0,sample_time) then
  omega1_measured = pre(count)*tooth_angle/sample_time;
  count = 0 "Another equation for count?";
end when;
end Counter;

```

However, there is a problem in this model. Note that there are actually **two** equations for `count`. Trying to compile such a model will lead to a situation where there are more equations than variables (*i.e.*, the problem is singular).

So what can we do about this? We need two different equations because the updates to `count` occur in response to different events. We could try to formulate everything under a single `when` statement, like this:

```

when {phi1>=pre(next_phi),phi1<=pre(prev_phi),sample(0,sample_time)} then
  if sample(0,sample_time) then
    omega1_measured := pre(count)*tooth_angle/sample_time;
    count :=0;
  else
    next_phi := phi1 + tooth_angle;
    prev_phi := phi1 - tooth_angle;
    count :=pre(count) + 1;
  end if;
end when;

```

But this kind of code quickly becomes hard to read. Fortunately, we can address this situation by placing all the `when` statements in an `algorithm` section.

The nature of an `algorithm` section is that it is treated as one single equation for any variables that are assigned within it. This allows multiple assignments to `count`, for example. When using an `algorithm` section, it is very important to understand that the **order** of assignment becomes important. If a conflict should arise (*e.g.*, a variable is assigned two values within the same `algorithm` section), the last one is the one that will be used. Another thing to note about `algorithm` sections is that you cannot write general equations. Instead, you must write *assignment statements*.

In this way, an `algorithm` section is very much like the way most programming languages work. The statements in the algorithm section are executed in order and each statement isn't interpreted as an equation, but rather as an assignment of an expression to a variable. The familiarity of assignment statements may make using `algorithm` sections attractive to people with a programming background who find the otherwise equation oriented aspects of Modelica disorienting and unfamiliar. But be aware that one big reason to avoid `algorithm` sections is because they interfere with the symbolic manipulation performed by the Modelica compiler. This can result in both poor simulation performance and a loss of flexibility in how you compose your models. So it is best to use an `equation` section whenever possible.

In our case, there are no significant consequences to using the `algorithm` section. Here is an example of how the previous estimation algorithm could be refactored using an `algorithm` section:

```

model CounterWithAlgorithm "Count teeth in a given interval using an algorithm"
  extends BasicEquations.RotationalSMD.SecondOrderSystem;
  parameter Real sample_time(unit="s")=0.125;
  parameter Integer teeth=200;
  parameter Real tooth_angle(unit="rad")=2*3.14159/teeth;
  Real next_phi, prev_phi;
  Integer count;
  Real omega1_measured;
initial equation
  next_phi = phi1+tooth_angle;
  prev_phi = phi1-tooth_angle;
  count = 0;
algorithm

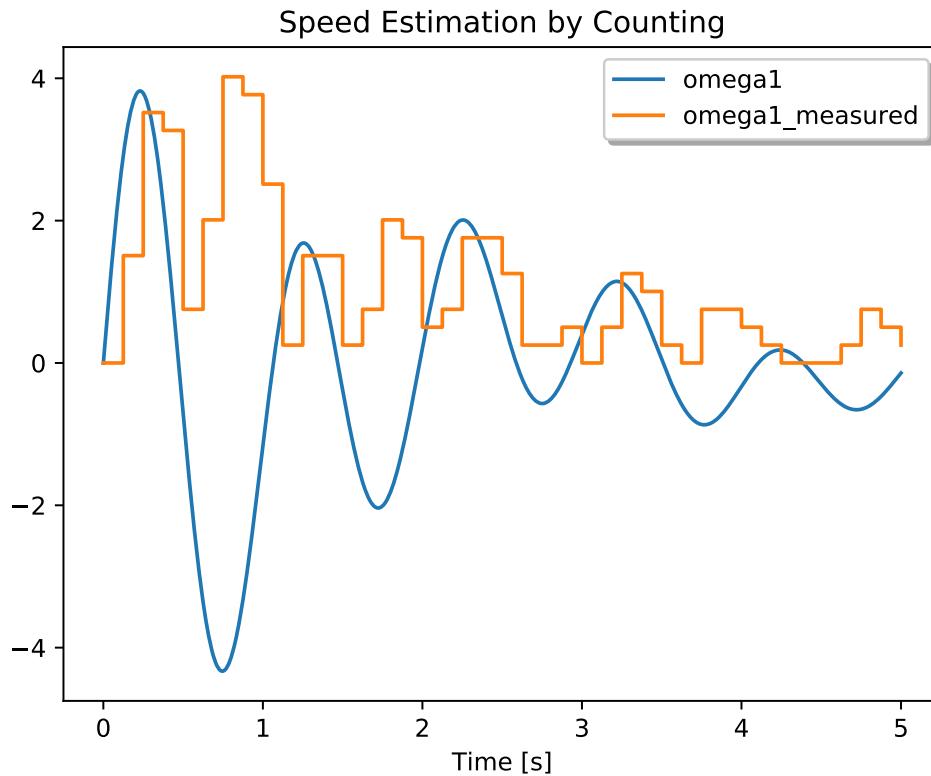
```

```

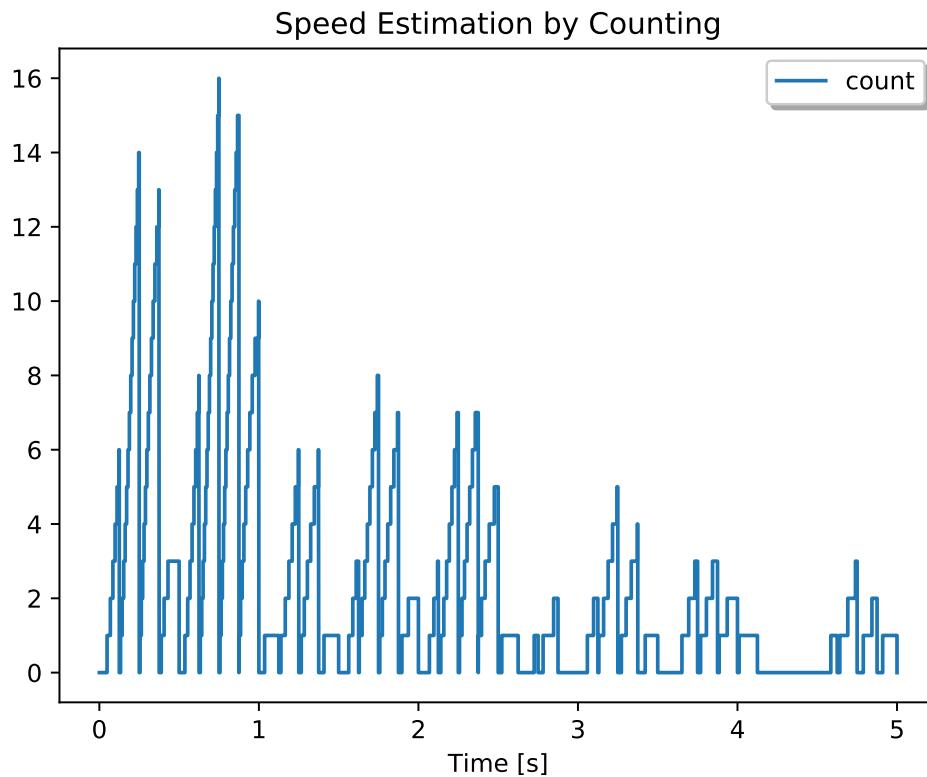
when {phi1>=pre(next_phi),phi1<=pre(prev_phi)} then
    next_phi := phi1 + tooth_angle;
    prev_phi := phi1 - tooth_angle;
    count := pre(count) + 1;
end when;
when sample(0,sample_time) then
    omega1_measured := pre(count)*tooth_angle/sample_time;
    count := 0;
end when;
end CounterWithAlgorithm;

```

The simulated results of this estimation technique can be seen in the following plot:



Again, we see that this approach cannot determine the direction of rotation. With the following plot, we can get a sense of how many events occur within each sample interval:



In general, the higher the count gets in an interval, the more accurate the estimate.

Conclusion

This section demonstrates how we can use the `when` construct to respond to physical events that occur in our system. These kinds of events and the impact they have on our system are just as important as the continuous dynamics we've covered previously. The ability to capture and respond to these physical events is an important part of why Modelica is so well suited to model complete systems since those systems frequently include both continuous and discrete behavior.

Hysteresis

In this section, we'll discuss the topic of hysteresis. This is an important concept to understand for certain types of modeling. Recall in our previous discussion of [State Event Handling](#) (page 51) we saw cases where chattering occurred. In those cases, we were able to use the `noEvent` operator to address the issue because the chattering was purely a response to numerical noise and not triggered by abrupt changes in behavior.

In this section, we will consider a slightly more extreme case. Consider the following model:

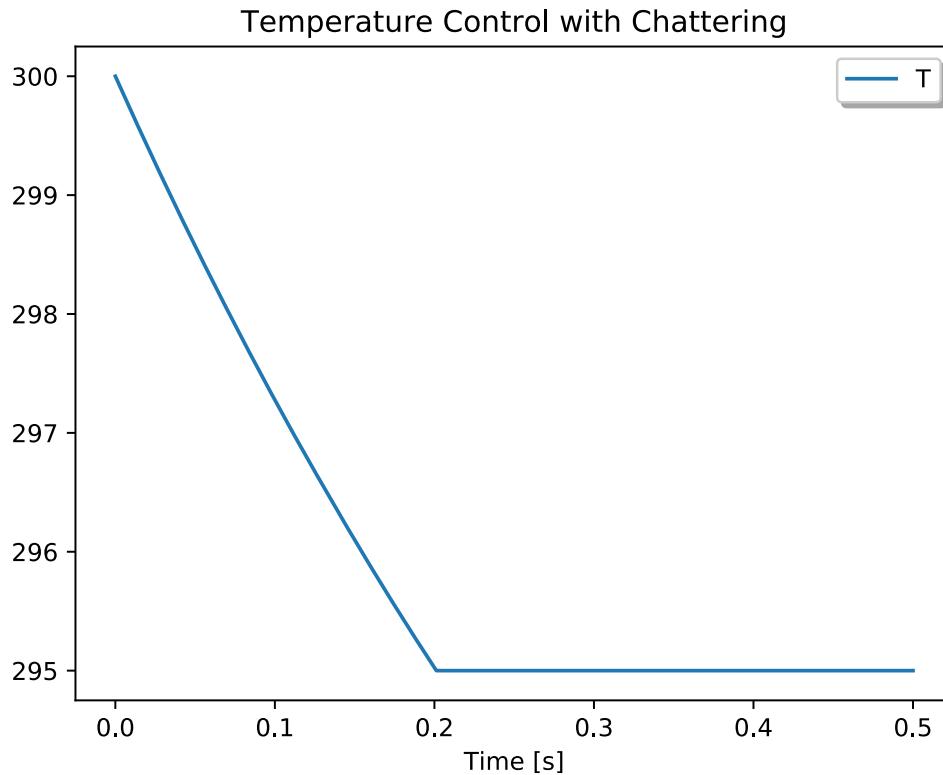
```
model ChatteringControl "A control strategy that will 'chatter'"
  type HeatCapacitance=Real(unit="J/K");
  type Temperature=Real(unit="K");
  type Heat=Real(unit="W");
  type Mass=Real(unit="kg");
  type HeatTransferCoefficient=Real(unit="W/K");
  Boolean heat "Indicates whether heater is on";
  parameter HeatCapacitance C=1.0;
```

```

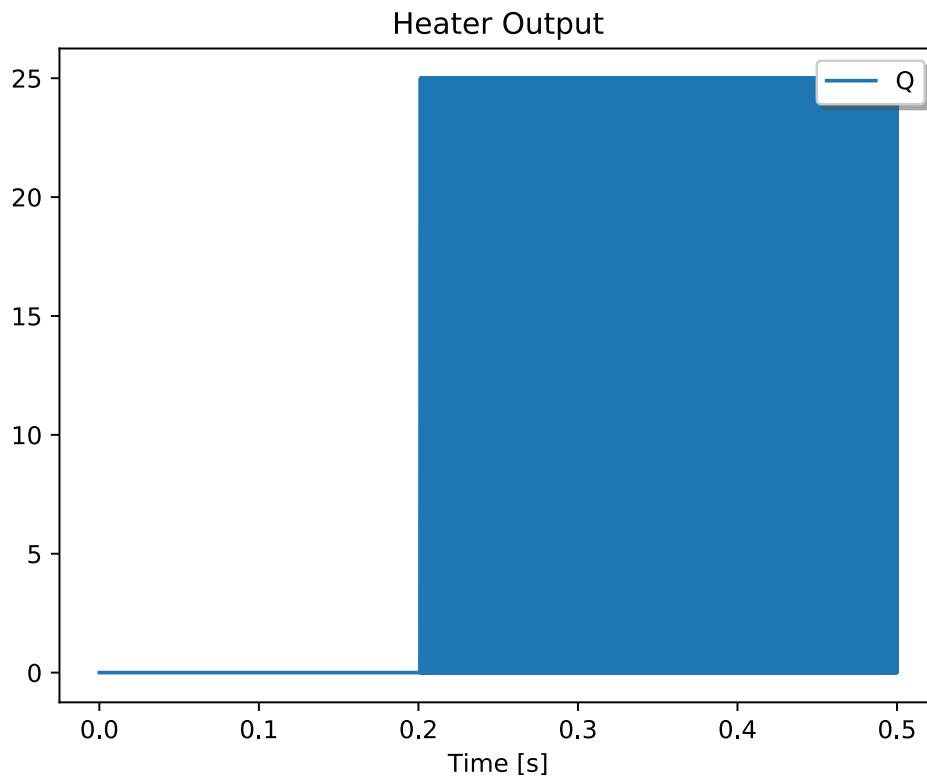
parameter HeatTransferCoefficient h=2.0;
parameter Heat Qcapacity=25.0;
parameter Temperature Tamb=285;
parameter Temperature Tbar=295;
Temperature T;
Heat Q;
initial equation
  T = Tbar+5;
equation
  heat = T<Tbar;
  Q = if heat then Qcapacity else 0;
  C*der(T) = Q-h*(T-Tamb);
end ChatteringControl;

```

If we simulate this model, we get the following results:



However, the simulation that yields these results takes a very long time to complete. The reason for such poor simulation performance can be better understood by looking at the heater output during the simulation:



What you see is that after around 0.2 seconds, the heater is constantly turning on and off. This happens so frequently, in fact, that you would have to zoom in quite a bit on the plot to see the transitions. With normal scaling, there are so many transitions that the results resemble a filled rectangle.

This is actually a real problem in control systems. If you look carefully at the way the furnace works in your own home, you will see that it does not turn on and off constantly as the temperature goes above and below the desired room temperature you have specified. Instead, it waits until the temperature gets some specified amount above or below the desired temperature before acting.

This “band” that is introduced around the desired temperature is called hysteresis. The problem with the `ChatteringControl` model is that it doesn’t have any hysteresis. Instead, it is constantly turning the heater off and on in response to minuscule changes in temperature.

The tricky thing about modeling hysteresis is that it is “stateful”. Determining the behavior of the system depends on what happened in the past. For this reason, we cannot simply use `if` statements. The reason is that `if` statements consider only the current state of the system, nothing else. To implement hysteresis, we need to use `when` statements. Consider the following model:

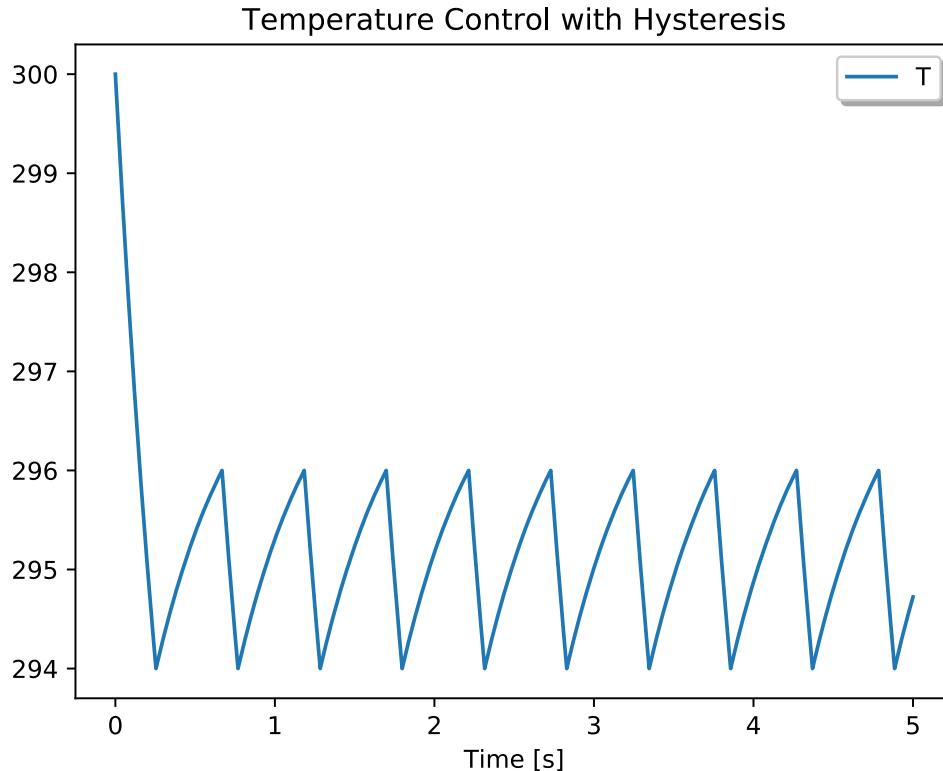
```
model HysteresisControl "A control strategy that doesn't chatter"
  type HeatCapacitance=Real(unit="J/K");
  type Temperature=Real(unit="K");
  type Heat=Real(unit="W");
  type Mass=Real(unit="kg");
  type HeatTransferCoefficient=Real(unit="W/K");
  Boolean heat(start=false) "Indicates whether heater is on";
  parameter HeatCapacitance C=1.0;
  parameter HeatTransferCoefficient h=2.0;
  parameter Heat Qcapacity=25.0;
  parameter Temperature Tamb=285;
  parameter Temperature Tbar=295;
  Temperature T;
```

```

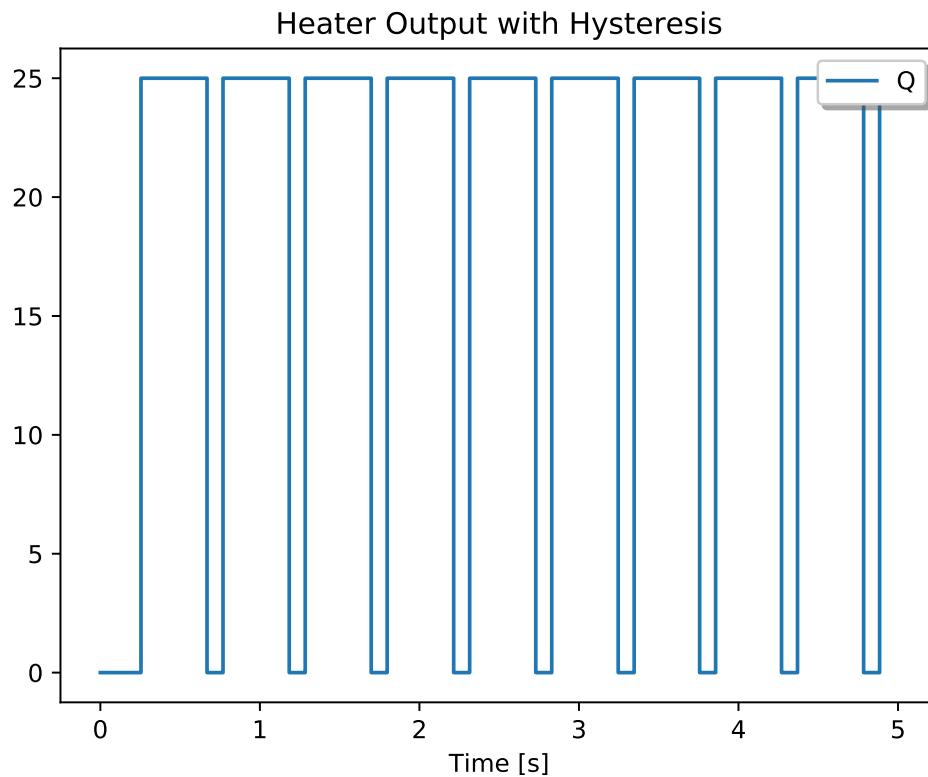
Heat Q;
initial equation
  T = Tbar+5;
  heat = false;
equation
  Q = if heat then Qcapacity else 0;
  C*der(T) = Q-h*(T-Tamb);
  when {T>Tbar+1,T<Tbar-1} then
    heat = T<Tbar;
  end when;
end HysteresisControl;

```

Examining the `when` statements, we see that the system only responds when $T > T_{\text{bar}} + 1$ becomes true or $T < T_{\text{bar}} - 1$ becomes true. **Note that nothing happens when these expressions become false.** This is why an `if` statement won't work. With an `if` statement or `if` expression, the behavior changes whenever the conditional expression changes. But with a `when` statement, the statements in the `when` statement become active **only** when the condition becomes true. If we simulate this model and look at the temperature, we see that it stays within the hysteresis band of our desired temperature.



More importantly, if we look at the heat output from the system, we see that, unlike our previous example, some time elapses between the heater turning on and the heater turning off.



The logic for implementing hysteresis can be made slightly more explicit by using an `algorithm` section (as previously discussed during our discussion on *speed estimation techniques* (page 69)).

```

model HysteresisControlWithAlgorithms "Control using algorithms"
  type HeatCapacitance=Real(unit="J/K");
  type Temperature=Real(unit="K");
  type Heat=Real(unit="W");
  type Mass=Real(unit="kg");
  type HeatTransferCoefficient=Real(unit="W/K");
  Boolean heat "Indicates whether heater is on";
  parameter HeatCapacitance C=1.0;
  parameter HeatTransferCoefficient h=2.0;
  parameter Heat Qcapacity=25.0;
  parameter Temperature Tamb=285;
  parameter Temperature Tbar=295;
  Temperature T;
  Heat Q;
initial equation
  T = Tbar+5;
  heat = false;
equation
  Q = if heat then Qcapacity else 0;
  C*der(T) = Q-h*(T-Tamb);
algorithm
  when T<Tbar-1 then
    heat :=true;
  end when;
  when T>Tbar+1 then
    heat :=false;
  end when;
end HysteresisControlWithAlgorithms;
```

Note how the two conditional expressions have been broken into two separate `when` statements. This makes it explicitly clear what causes the heat to be turned on and off. These `when` statements were placed in an `algorithm` section because they both assign to the same variable, `heat`.

Synchronous Systems

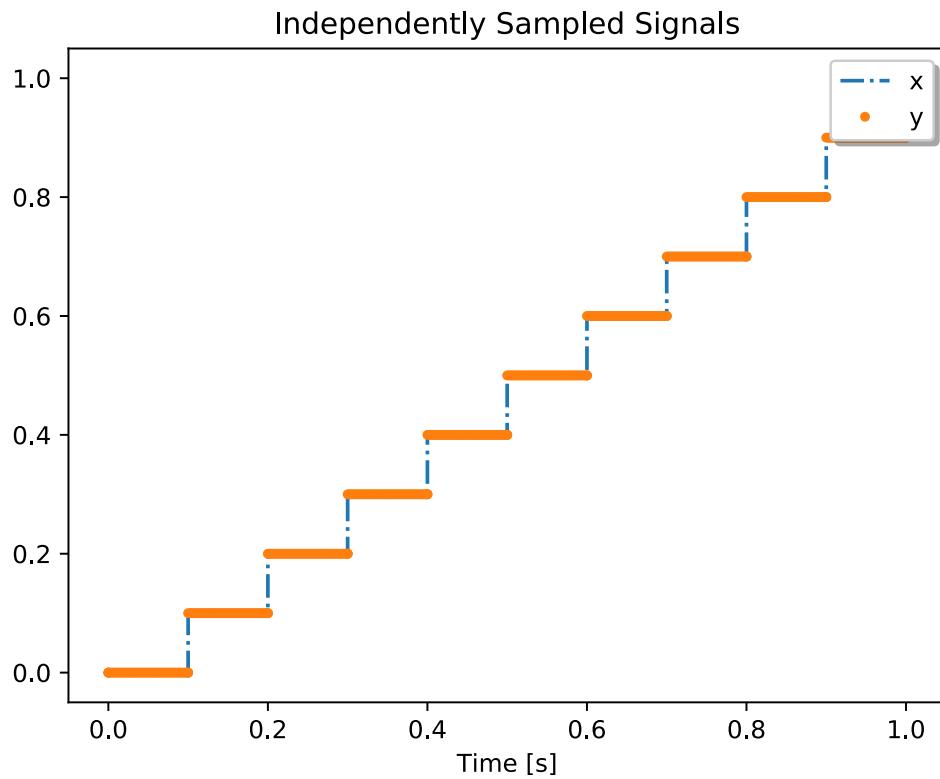
In Modelica version 3.3, new features were introduced to address concerns about non-deterministic discrete behavior [*Elmqvist*] (page 357). In this section, we'll present some examples of how these issues presented themselves before version 3.3 and show how these new features help address them.

To start, consider the following model:

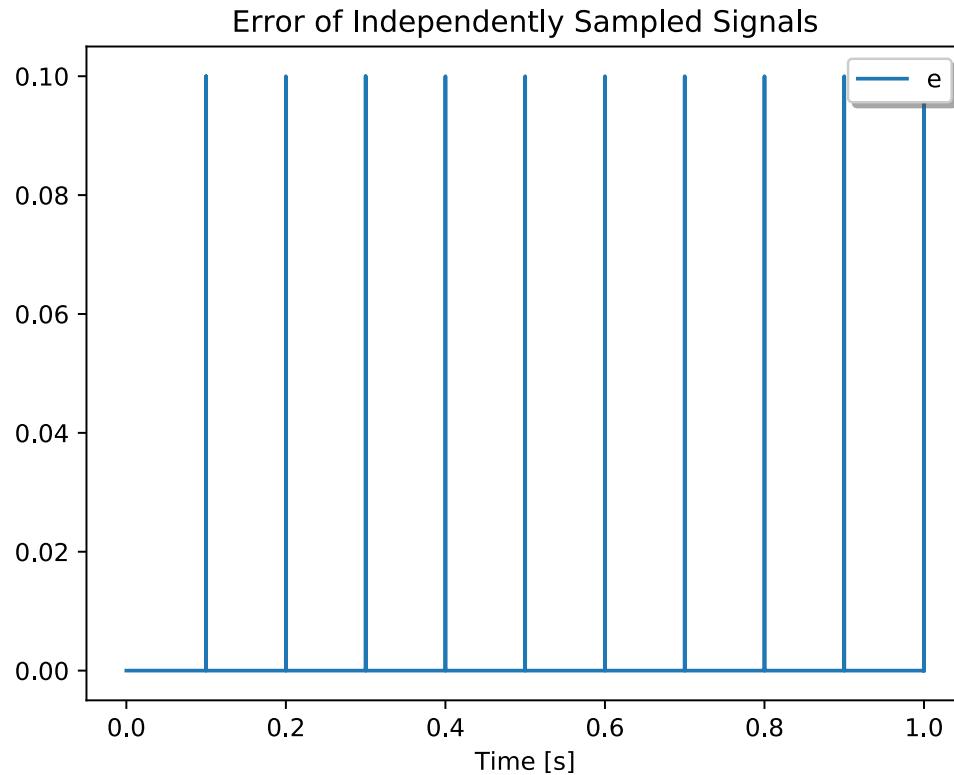
```
model IndependentSampling "Sampling independently"
  Real x "Sampled at 10Hz via one method";
  Real y "Sampled at 10Hz via another method";
  Real e "Error between x and y";
  Real next_time "Next sample for y";
equation
  when sample(0,0.1) then
    x = time;
  end when;

  when {initial(), time>pre(next_time)} then
    y = time;
    next_time = pre(next_time)+0.1;
  end when;
  e = x-y;
end IndependentSampling;
```

If you look carefully, you will see that `x` and `y` are both computed at discrete times. Furthermore, they are both sampled initially at the start of the simulation and then again every 0.1 seconds. But the question is, are they really identical? To help address this question, we include the variable `e` which measures the difference between them.



Simulating this model, we get the following trajectories for x and y . Of course, they look identical. But in order to really determine if there are any differences between them, let's plot the error value, e :



Now, let's consider the following model:

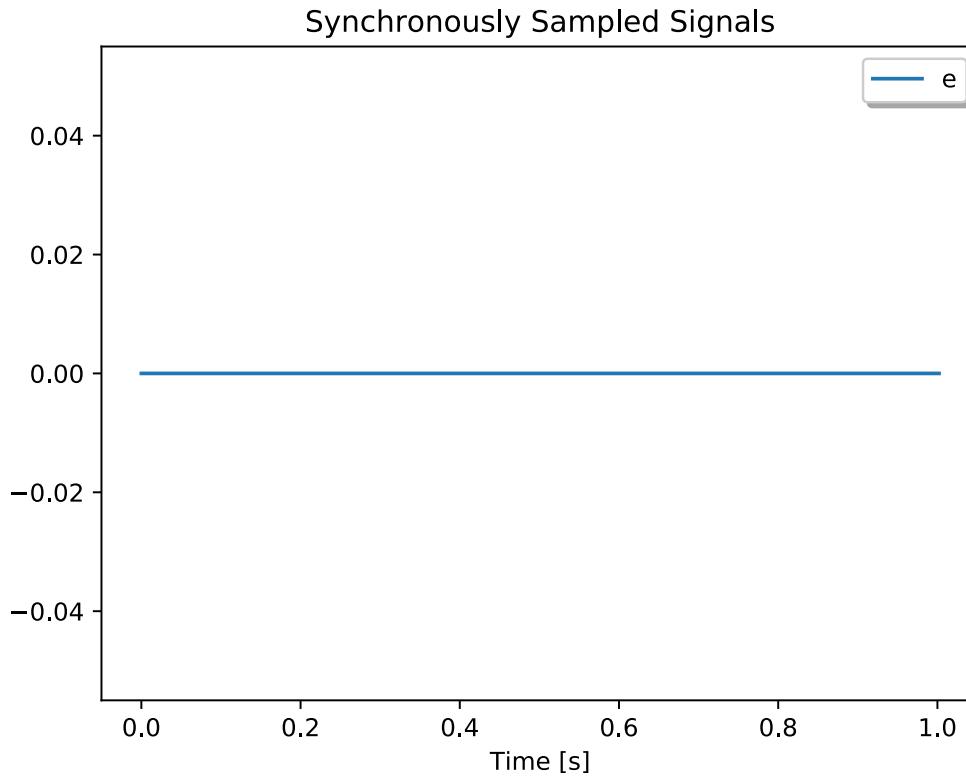
```
model SynchronizedSampling "A simple way to synchronize sampling"
  Integer tick "A clock counter";
  Real x, y;
  Real e "Error between x and y";
equation
  when sample(0,0.1) then
    tick = pre(tick)+1;
  end when;

  when change(tick) then
    x = time;
  end when;

  when change(tick) then
    y = time;
  end when;

  e = x-y;
end SynchronizedSampling;
```

Here, we set up a common signal that triggers the assignment to both variables. In this way, we can be sure that when the `tick` signal becomes true, both `x` and `y` will be assigned a value. Sure enough, if we run this model, we see that the error is always zero:



This kind of approach, where each signal is sampled based on a common “tick” (or clock), is a good way to avoid determinism issues. However, what about cases where you have one signal that samples at a higher rate than another, but you know that at certain times they should be sampled together? Consider the following example:

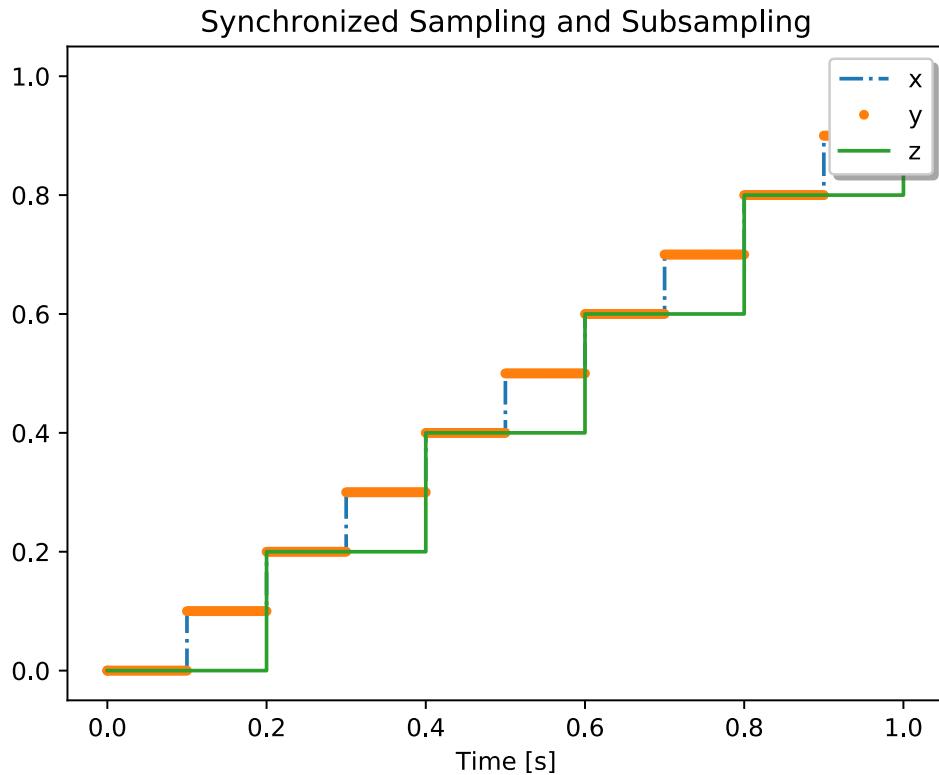
```
model SubsamplingWithIntegers "Use integers to implement subsampling"
  Integer tick "Clock counter";
  Real x, y, z;
equation
  when sample(0,0.1) then
    tick = pre(tick)+1;
  end when;

  when change(tick) then
    x = time;
  end when;

  when change(tick) then
    y = time;
  end when;

  when mod(tick-1,2)==0 then
    z = time;
  end when;
end SubsamplingWithIntegers;
```

In this case, the variable `tick` is a counter. Every time it changes, we update the values of `x` and `y`. So this much is identical to the previous models. However, we added a third signal, `z`, that is sampled only when the value of `tick` is odd. So `x` and `y` are sampled twice as often. But every time `z` is updated, we can be sure that `x` and `y` are updated at exactly the same time. Simulating this model gives us:



This is the approach taken in Modelica prior to version 3.3. But version 3.3 introduced some new features that allow us to more easily express these situations.

Consider the following model:

```
model SamplingWithClocks "Using clocks to sub and super sample"
  Real x, y, z, w;
equation
  x = sample(time, Clock(1,10));
  y = sample(time, Clock(1,10));
  z = subSample(x, 2);
  w = superSample(x, 3);
end SamplingWithClocks;
```

Now, instead of relying on a `when` statement, we use an enhanced version of the `sample` function where the first argument is an expression to evaluate to determine the sampled value and the second argument is used to tell us when to evaluate it. Let's work through these lines one by one and discuss them. First we have:

```
x = sample(time, Clock(1,10));
```

Note that we have done away with the 0.1. We no longer see any mention of the clock interval as a real number. Instead, we use the `Clock` operator to define clock interval for `x` as a rational number. This is important because it allows us to do exact comparisons between clocks. This brings us to the next line:

```
y = sample(time, Clock(1,10));
```

Again, we see the rational representation of the clock. What this means, in practice, is that the Modelica compiler can know for certain that these two clocks, `x` and `y`, are identical because they are defined in

terms of integer quantities which allow exact comparison. This means that when executing a simulation, we can know for certain that these two clocks will trigger simultaneously.

If we wanted to create a clock that was exactly half as slow as `x`, we can use the `subSample` operator to accomplish this. We see this in the definition of `z`:

```
z = subSample(x, 2);
```

Behind the scenes, the Modelica compiler can reason about these clocks. It knows that the `x` clock triggers every $\frac{1}{10}$ of a second. Using the information provided by the `subSample` operator the Modelica compiler can therefore deduce that `z` triggers every $\frac{2}{10}$ of a second. Conceptually, this means that `z` could also have been defined as:

```
z = sample(time, Clock(2,10));
```

But by defining `z` using the `subSample` operator and defining it with respect to `x` we ensure that `z` is always triggering at half the frequency of `x` regardless of how `x` is defined.

In a similar way, we can define another clock, `w` that triggers 3 times as frequently as `x` by using the `superSample` operator:

```
w = superSample(x, 3);
```

Again, we could have defined `w` directly using `sample` with:

```
w = sample(time, Clock(1,30));
```

But by using `superSample`, we can ensure that `w` is always sampling three times as fast as `x` and six times as fast as `z` (since `z` is also defined with respect to `x`).

The synchronous clock features in Modelica are relatively new. As such, they are not yet supported by all Modelica compilers. To learn more about these synchronous features and their applications see [[Elmqvist](#)] (page 357) and/or the Modelica Specification, version 3.3 or later.

1.2.2 Review

Events

In the first chapter on [Basic Equations](#) (page 1) we saw examples of how to describe continuous behavior. The equations introduced in that chapter applied at all times and the solutions to those equations were always continuous. In this chapter, we discussed the various ways in Modelica to describe discrete behavior. Events are the root cause of all discrete behavior in Modelica.

Conditional Expressions

Events are generated in one of two ways. First, they can be generated by conditional expressions. In the first few examples in this chapter, we saw that conditional expressions can trigger events. If these conditional expressions only involve the variable `time`, then we call them “time events”. The `time` variable is a built-in, global variable that is treated as an “input” to all models.

If events are generated because of conditional expressions that involve solution variables, then we call them “state events”. The important distinctions between time events and state events were discussed in the [first](#) (page 42) and [second](#) (page 47) sections of this chapter, respectively.

Conditional expressions can be created using the relational operators (`>`, `>=`, `<`, `<=`, `==`) and logical operators (`not`, `and`, `or`). As we saw in our discussion of [Event Suppression](#) (page 54), it is possible to suppress the events generated by these conditional expressions by surrounding them with the `noEvent` operator.

Frequently, these event generating conditional expressions occur in the context of an `if` statement or an `if` expression. But it should also be noted that even a simple variable assignment, *e.g.*,

```
Boolean late;
equation
  late = time>=5.0 "This will generate an event";
```

can trigger an event to be generated.

Piecewise Constructions

There is an important special case when dealing with conditional expressions. In some cases, it is useful to create an expression that is constructed piecewise. For example,

```
x = if (x<0) then 0 else x^3;
```

It is hard for a Modelica compiler to reliably determine that such a function is continuous and has continuous derivatives. For this reason, Modelica includes the `smooth` operator to explicitly express such conditions. For example, using the `smooth` operator as follows:

```
x = smooth(2, if (x<0) then 0 else x^3);
```

indicates that the expression is continuous as is and will remain continuous if differentiated up to 2 times because

$$\begin{aligned}x' &= \begin{cases} 0, & \text{for } x < 0, \\ 3x^2, & \text{otherwise,} \end{cases} \\x'' &= \begin{cases} 0, & \text{for } x < 0, \\ 6x, & \text{otherwise,} \end{cases} \\x''' &= \begin{cases} 0, & \text{for } x < 0, \\ 6, & \text{otherwise.} \end{cases}\end{aligned}$$

Hence, the function, its first and second derivatives are continuous at $x = 0$, but the third derivative is discontinuous.

Note that the `smooth` operator requires an upper bound to be specified.

Events and Functions

In addition to being generated by conditional expressions, events can also be generated by certain functions in Modelica.

Event Generating Functions

The following is a list of functions that generate events wherever the return value has a discontinuity.

Function	Description
<code>div(x,y)</code>	Algebraic quotient with fractional part discarded.
<code>mod(x,y)</code>	Modulus of <code>x/y</code>
<code>rem(x,y)</code>	Remainder from the algebraic quotient
<code>ceil(x)</code>	Smallest integer not less than <code>x</code>
<code>floor(x)</code>	Largest integer not greater than <code>x</code> (returns a <code>Real</code>)
<code>integer(x)</code>	Largest integer not greater than <code>x</code> (returns an <code>Integer</code>)
<code>initial()</code>	<code>true</code> during initialization, otherwise <code>false</code>
<code>terminal()</code>	<code>true</code> at end of simulation, otherwise <code>false</code>
<code>sample(t0,dt)</code>	Generates an event at <code>t0</code> and every <code>dt</code> seconds later
<code>edge(x)</code>	<code>true</code> only at the instant that <code>x</code> is <code>true</code>
<code>change(x)</code>	<code>true</code> whenever <code>x</code> changes value

Non-Event Generating Functions

The following is a table of functions that do **not** generate events:

Function	Description
<code>abs(x)</code>	Absolute value of <code>x</code>
<code>sign(x)</code>	Sign of <code>x</code> (-1, 0, or 1)
<code>sqrt(x)</code>	Square root of <code>x</code>
<code>min(x,y)</code>	Minimum value between <code>x</code> and <code>y</code>
<code>max(x,y)</code>	Maximum value between <code>x</code> and <code>y</code>

Event Related Operators

The following operators provide special information about event generating signals:

Function	Description
<code>pre(x)</code>	During an event, holds the value of <code>x</code> before the event
<code>previous(x)</code>	During clock tick, value of <code>x</code> during previous clock tick
<code>hold(x)</code>	Anytime, value of <code>x</code> during previous clock tick
<code>sample(expr,clock)</code>	During clock tick, value of <code>expr</code>
<code>noEvent(expr)</code>	Suppresses events generated by <code>expr</code>
<code>smooth(p,expr)</code>	Indicates <code>expr</code> can be safely differentiated <code>p</code> times.

Clock Related Operators

The following operators are used to create a manipulate clocks (event generators that trigger at regular intervals):

Function	Description
<code>Clock(i,r)</code>	A clock that fires every $\frac{i}{r}$ seconds where <code>i</code> and <code>r</code> are both of type <code>Integer</code>
<code>Clock(dt)</code>	A clock that fires every <code>dt</code> seconds where <code>dt</code> is a <code>Real</code>
<code>subSample(u,s)</code>	A clock that samples <code>s</code> times slower than the clock used to sample <code>u</code> where <code>s</code> is an <code>Integer</code>
<code>superSample(u,s)</code>	A clock that samples <code>s</code> times as fast as the clock used to sample <code>u</code> where <code>s</code> is an <code>Integer</code>

Note that the `Clock` constructor function is overloaded (*i.e.*, can take different arguments). It is worth reiterating that the synchronous clock features in Modelica are relatively new. As such, they are not

yet supported by all Modelica compilers. To learn more about these synchronous features and their applications see [\[Elmqvist\]](#) (page 357) and/or the Modelica Specification, version 3.3 or later.

If

Although it is pretty intuitive, it is worth having a short review of the syntax for `if` statements and `if` expressions. Let's start with `if` expressions because they are the simplest to explain. An `if` expression has the form:

```
if cexpr then expr1 else expr2;
```

where `cexpr` is a conditional expression (that will evaluate to a Boolean value), `expr1` is the value the expression will have if `cexpr` evaluates to `true` and `expr2` is the value the expression will have if `cexpr` evaluates to `false`.

An `if` statement has the general syntax:

```
if cond1 then
    // Statements used if cond1==true
elseif cond2 then
    // Statements used if cond1==false and cond2==true
// ...
elseif condn then
    // Statements used if all previous conditions are false
    // and condn==true
else
    // Statements used otherwise
end if;
```

It is important to note that when an `if` statement appears in an `equation` section, the number of equations must be the same regardless of which branch through the `if` statement is taken (this applies in the presence of `elseif` as well). One exception is the use of `if` within an `initial equation` or `initial algorithm` section where an `else` clause is not required since the number of equations doesn't have to be same for both branches. Another notable exception is the use of `if` within [Functions](#) (page 116) where, again, there is not requirement that the number of equations be the same across both branches.

A special case here is when you have an `if` statement that looks like this:

```
if cond then
    x = y;
else
    x = z;
end if;
```

We can see that in both branches, a value is assigned to `x`. As such, an equivalent way to write this using an `if` expression would be:

```
x = if cond then y else z;
```

The advantage of the second formulation is that it may make it easier for a tool to optimize the code generation in the case of an `if` expression.

Note: Note that conditional expressions within both `if` statements and `if` expressions have the potential to generate [Events](#) (page 82).

When

By using `when`, we can express conditions we are interested in reacting to and what we wish to do in response to them. In this section, we'll review the key ideas behind `when` statements. A `when` statement

has the following general form:

```
when expr then  
  // Statements  
end when;
```

if vs. when

In our discussion on [Hysteresis](#) (page 72), we briefly discussed the difference between an `if` statement and a `when` statement. The statements in a `when` statement become active only for an instant when the triggering conditional expression becomes true. At all other times, the `when` statement has no effect. An `if` statement or `if` expression remains active as long as the conditional expression is true. If the `if` statement or `if` expression includes an `else` clause then some branch will always be active.

Expressions

Most of the time, the expression `expr` is going to be a conditional expression and usually it will involve relational operators. Typical examples of expressions frequently used with a `when` statement would be `time>=2.0`, `x>=y+2`, `phi<=prev_phi` and so on. Recall from our discussion of the [Interval Measurement](#) (page 66) speed estimation algorithm that you should **almost always** put the `pre` operator around any variables in `expr` that also appear inside the `when` statement.

In the [Bouncing Ball](#) (page 47) example, we saw a case where `expr` was not a (scalar) conditional expression, but rather a vector of conditional expressions. Recall from that discussion that the `when` statement becomes active if **any** of the conditions in the vector of expressions becomes true.

Statements

A `when` statement is used to define new values for some variables. These new values can be assigned in one of two ways. The first is by introducing an equation of the form:

```
var = expr;
```

In this case, the variable `var` will be given the value of `expr`. Within `expr`, the `pre` operator should be used when referring to the pre-event value for a variable. Any variable assigned in this way is a discrete variable. This means that its value only changes during events. In other words, it will be a piecewise constant function. Note, a variable assigned in this way cannot be continuous over any interval in the simulation.

If you want to explicitly mark a variable as discrete, you can prefix it with the `discrete` qualifier (as we saw in the [Sample and Hold](#) (page 65) example earlier in this chapter) although this is not strictly necessary. By adding the `discrete` qualifier you ensure that the variable's value must be determined within a `when` statement.

The other way a variable can be given a value within a `when` statement, as demonstrated in the [Bouncing Ball](#) (page 47) example, is by using the `reinit` operator. In that case, the statement within the `when` statement will take the form:

```
reinit(var, expr);
```

When using the `reinit` operator, the variable, `var`, **must be a state**. In other words, its solution must arise from solving a differential equation. The use of `reinit` on such a variable has the effect of stopping the integration process, changing the value of the state (and any other states that have the `reinit` operator applied to them within the same `when` statement) and then resuming integration using what are effectively a new set of initial conditions. The values of all other states not re-initialized with the `reinit` operator remain unchanged.

algorithm Sections

One final note about `when` statements is how they interact with the “single assignment” rule in Modelica. This rule, from the specification, states that there must be exactly one equation used to determine the value of each variable. As we saw in the sections on [Speed Measurement](#) (page 63) and [Hysteresis](#) (page 72), it is sometimes necessary (or at least clearer) to express behavior in terms of multiple assignments. In those cases, if all the assignments are included within a single `algorithm` section, they are effectively combined into a single equation. However, doing so will limit the compiler’s ability to perform symbolic manipulation and, therefore, may impact simulation performance and/or reusability of the models.

It is also worth noting that if the semantics of an `algorithm` section are needed during initialization, Modelica includes an `initial algorithm` section that is analogous to the `initial equation` section discussed in the previous discussion on [Initialization](#) (page 35). The `initial algorithm` section will be applied only during the initialization phase to determine initial conditions, just like an `initial equation` section, but the `initial algorithm` section will allow multiple assignments to the same variable. The same caveats apply with respect to symbolic manipulation.

1.3 Vectors and Arrays

1.3.1 Examples

State Space

ABCD Form

Recall from our previous discussion on [Ordinary Differential Equations](#) (page 35) that we can express differential equations in the following form:

$$\begin{aligned}\dot{\vec{x}}(t) &= \vec{f}(\vec{x}(t), \vec{u}(t), t) \\ \vec{y}(t) &= \vec{g}(\vec{x}(t), \vec{u}(t), t)\end{aligned}$$

In this form, x represents the states in the system, u represents any externally specified inputs to the system and y represents the outputs of the system (*i.e.*, variables that may not be states, but can ultimately be computed from the values of the states and inputs).

There is a particularly interesting special case of these equations when the functions \vec{f} and \vec{g} depend linearly on \vec{x} and \vec{u} . In this case, the equations can be rewritten as:

$$\begin{aligned}\dot{\vec{x}}(t) &= A(t)\vec{x}(t) + B(t)\vec{u}(t) \\ \vec{y}(t) &= C(t)\vec{x}(t) + D(t)\vec{u}(t)\end{aligned}$$

The matrices in this problem are the so-called “ABCD” matrices. This ABCD form is useful because there are several interesting calculations that can be performed once a system is in this form. For example, using the A matrix, we can compute the natural frequencies of the system. Using various combinations of these matrices, we can determine several very important properties related to control of the underlying system (*e.g.*, observability and controllability).

Note that this ABCD form allows these matrices to vary with time. There is a slightly more specialized form that, in addition to being linear, is also time-invariant:

$$\begin{aligned}\dot{\vec{x}}(t) &= A\vec{x}(t) + B\vec{u}(t) \\ \vec{y}(t) &= C\vec{x}(t) + D\vec{u}(t)\end{aligned}$$

This form is often called the “LTI” form. The LTI form is important because, in addition to having the same special properties as the ABCD form, the LTI form can be used as a very simple form of “model exchange”. Historically, when someone derived the behavior equations for a given system (either by hand

or using some modeling tool), one way they could import those equations into other tools was to put them in the LTI form. This means that the model could be exchanged, shared or published as a series of matrices with either numbers or expressions in them. Today, technologies like Modelica and FMI¹² provide much better options for model exchange.

LTI Models

If someone gave us a model in LTI form, how would we express that in Modelica? Here is one way we might choose to do it:

```
model LTI
  "Equations written in ABCD form where matrices are also time-invariant"
  parameter Integer nx=0 "Number of states";
  parameter Integer nu=0 "Number of inputs";
  parameter Integer ny=0 "Number of outputs";
  parameter Real A[nx,nx]=fill(0,nx,nx);
  parameter Real B[nx,nu]=fill(0,nx,nu);
  parameter Real C[ny,nx]=fill(0,ny,nx);
  parameter Real D[ny,nu]=fill(0,ny,nu);
  parameter Real x0[nx]=fill(0,nx) "Initial conditions";
  Real x[nx] "State vector";
  Real u[nu] "Input vector";
  Real y[ny] "Output vector";
initial equation
  x = x0 "Specify initial conditions";
equation
  der(x) = A*x+B*u;
  y = C*x+D*u;
end LTI;
```

The first step in this model is to declare the parameters `nx`, `nu` and `ny`. These represent the number of states, inputs and outputs, respectively. Next, we define the matrices `A`, `B`, `C` and `D`. Because we are creating a model for a linear, time-invariant representation all of these matrices can be parameters. We know that `A`, `B`, `C` and `D` are arrays because their declarations followed by `[]`s. We know they are matrices because within the `[]`s there are two dimensions given. Finally, we see declarations for `x0`, `x`, `u` and `y`. These are also arrays. But in this case, they are vectors, since they each have only a single dimension.

Another thing to note about this model is that all parameters have been given default values. For `nx`, `nu` and `ny`, the assumption is that the number of states, inputs and outputs is zero by default. For the matrices, we assume that they are filled with zeros by default. Similarly, for initial conditions we assume that all states start the simulation with a value of zero unless otherwise specified. We shall see shortly how these assumptions make it possible for us to write very simple models by simply overriding the values for these parameters.

Vector Equations

The rest of the model should look pretty familiar by now. One thing that is important to point out is the fact that the equations in this model are all **vector** equations. An equation in Modelica can involve scalars or arrays. The only requirement is that both side of the equation have the same number of dimensions and the same size for each dimension. So in the case of the LTI model, we have the following initial equation:

```
initial equation
  x = x0 "Specify initial conditions";
```

¹² <http://fmi-standard.org>

This equation is a vector equation that expresses the fact that each element in \mathbf{x} has the corresponding value in \mathbf{x}_0 at the start of a simulation. In practice, what happens is that each element in these vectors is automatically expanded into a series of scalar equations.

Another thing that helps keep these equations readable is that Modelica has some special rules regarding [Vectorization](#) (page 112) of functions. In a nutshell, these rules say that if you have a function that works with scalars, you can automatically use it with vectors as well. If you do, Modelica will automatically apply the function to each element in the vector. So, for example, the expression `der(x)` in the LTI model is a vector where each element in the vector represents the derivative of the respective element of \mathbf{x} .

Finally, many of the typical algebraic operators like `+`, `-` and `*` have special meanings when applied to vectors and matrices. These definitions are designed so that they correspond with conventional mathematical notation. So in the LTI model, the expression `A*x` corresponds to a matrix-vector product.

LTI Examples

With all this in mind, let's revisit several of our previous examples to see how they can be represented in LTI form using our LTI model. Note that we will again use inheritance (via the `extends` keyword) to reuse the code in the LTI model.

Let's start with the [Simple First Order System](#) (page 1) we presented earlier. Using the LTI model, we can write this model as:

```
model FirstOrder "Represent der(x) = 1-x"
  extends LTI(nx=1,nu=1,A=[-1], B=[1]);
equation
  u = {1};
end FirstOrder;
```

When we extend from LTI, we only need to specify the parameter values that are different from the default values. In this case, we specify that there is one state and one input. Then we specify A and B as 1×1 matrices. Finally, since we have an input, we need to provide an equation for it. The input can, in general, be time-varying so we don't represent it as a parameter, but rather with an equation. Note that in the equation:

```
u = {1};
```

the expression `{1}` is a vector literal. This means that we are building a vector as a list of its components. In this case, the vector has only one component, 1. But we can build longer vectors using a comma separated list of expressions, *e.g.*,

```
v = {1, 2, 3*4, 5*sin(time)};
```

It is worth noting that, in addition to setting parameter values, we also can include equations in the `extends` statement. So, we could have avoided the `equation` section altogether and written the model more compactly as:

```
model FirstOrder_Compact "Represent der(x) = 1-x"
  extends LTI(nx=1,nu=1,A=[-1], B=[1], u={1});
end FirstOrder_Compact;
```

In general, including the `equation` section makes the code a bit more readable for others. But there are some circumstances where it is more convenient to include the equation as a modification in the `extends` statement.

Now let's turn our attention to the [cooling](#) (page 6) we also discussed earlier. In LTI form, we could have written the model as:

```
model NewtonCooling "NewtonCooling model in state space form"
  parameter Real T_inf=27.5 "Ambient temperature";
```

```

parameter Real T0=20 "Initial temperature";
parameter Real hA=0.7 "Convective cooling coefficient * area";
parameter Real m=0.1 "Mass of thermal capacitance";
parameter Real c_p=1.2 "Specific heat";
extends LTI(nx=1,nu=1,A=[-hA/(m*c_p)],B=[hA/(m*c_p)],x0={20});
equation
  u = {T_inf};
end NewtonCooling;

```

This model is very similar to the previous one. However, in this case, instead of putting numbers into our matrices, we've put expressions involving other parameters like m , c_p and so on. In this way, if those physical parameters are changed, the values for A and B will change accordingly.

We can take a similar approach in reformulating our previous *mechanical example* (page 13) into LTI form:

```

model RotationalSMD
  "State space version of a rotational spring-mass-damper system"
  parameter Real J1=0.4;
  parameter Real J2=1.0;
  parameter Real c1=11;
  parameter Real c2=5;
  parameter Real d1=0.2;
  parameter Real d2=1.0;
  extends LTI(nx=4, nu=0, ny=0, x0={0, 1, 0, 0},
              A=[0, 0, 1, 0;
                  0, 0, 0, 1;
                  -c1/J1, c1/J1, -d1/J1, d1/J1;
                  c1/J2, -c1/J2-c2/J2, d1/J2, -d1/J2-d2/J2]);
  equation
    u = fill(0, 0);
  end RotationalSMD;

```

Again, we compute A from physical parameters. One thing to note about this example is the construction of A . Mathematically, the A matrix is defined as:

$$A = \begin{vmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{k_1}{J_1} & \frac{k_1}{J_1} & -\frac{d_1}{J_1} & \frac{d_1}{J_1} \\ \frac{k_1}{J_2} & -\frac{k_1}{J_2} - \frac{k_2}{J_2} & \frac{d_1}{J_2} & -\frac{d_1}{J_2} - \frac{d_2}{J_2} \end{vmatrix}$$

One thing we can note about this construction of A is that the first two rows might be easier to express as a matrix of zeros and an identity matrix. In other words, it might be simpler to construct the matrix as a set of sub-matrices, *i.e.*,

$$A = \left| \begin{array}{cc|cc} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \hline -\frac{k_1}{J_1} & \frac{k_1}{J_1} & -\frac{d_1}{J_1} & \frac{d_1}{J_1} \\ \frac{k_1}{J_2} & -\frac{k_1}{J_2} - \frac{k_2}{J_2} & \frac{d_1}{J_2} & -\frac{d_1}{J_2} - \frac{d_2}{J_2} \end{array} \right|$$

In Modelica, we can construct our A matrix from sub-matrices in this way:

```

model RotationalSMD_Concat
  "State space version of a rotational spring-mass-damper system using concatenation"
  parameter Real J1=0.4;
  parameter Real J2=1.0;
  parameter Real c1=11;
  parameter Real c2=5;
  parameter Real d1=0.2;
  parameter Real d2=1.0;
  parameter Real S[2,2] = [-1/J1, 1/J1; 1/J2, -1/J2];
  extends LTI(nx=4, nu=0, ny=0, x0={0, 1, 0, 0},

```

```

A=[zeros(2, 2), identity(2);
   c1*S+[0,0;0,-c2/J2], d1*S+[0,0;0,-d2/J2]],
B=fill(0, 4, 0), C=fill(0, 0, 4),
D=fill(0, 0, 0));
equation
  u = fill(0, 0);
end RotationalSMD_Concat;

```

In the section above we do not include a representation of the Lotka-Volterra equations in LTI form. This is because the Lotka-Volterra equations, while being time-invariant, are not linear. It is worth pointing out that Modelica does not directly enforce either of these properties when using the LTI model. So it is possible to represent non-linear or time-variant models using this approach. But it would be confusing since the term LTI implies that the equations are both linear and time-invariant.

Using Components

In all of these examples so far, we've used inheritance (via `extends`) to reuse the equations from the LTI model. In general, there is a **much better** way to reuse these equations which is to **treat them as sub-components**. To see how this is done, we will recast our previous *electrical examples* (page 10) in LTI form. But this time, we'll create a named instance of the LTI model:

```

model RLC "State space version of an RLC circuit"
  parameter Real Vb=24;
  parameter Real L=1;
  parameter Real R=100;
  parameter Real C=1e-3;
  LTI rlc_comp(nx=2, nu=1, ny=2, x0={0,0},
    A=[-1/(R*C), 1/C; -1/L, 0],
    B=[0; 1/L],
    C=[1/R, 0; -1/R, 1],
    D=[0; 0]);
equation
  rlc_comp.u = {Vb};
end RLC;

```

Note that this time we do not use `extends` or inheritance of any kind. Instead, we actually declare a variable called `rlc_comp` that is of type `LTI`. Once we have finished covering all the basics of how to describe different kinds of behavior in Modelica, we'll turn our attention to how to organize all these equations into reusable *Components* (page 183). But for now, this is just a “sneak peek” of (big) things to come.

What we see in this *RLC* example is that we now have a variable called `rlc_comp` and this component, in turn, has all the parameters and variables of the LTI model inside it. So, for example, we see that our equation to specify the input, `u`, is written as:

```
rlc_comp.u = {Vb};
```

Note that this equation means that we are providing an equation for the variable `u` that is **inside** the variable `rlc_comp`. As we will see later, we can use hierarchy to manage a considerable amount of complexity that arises from complex system descriptions. The use of the `.` operation here is how we can reference variables that are organized in this hierarchical manner. Again, this will be discussed thoroughly when we introduce *Components* (page 183).

One-Dimensional Heat Transfer

Our previous discussion on *State Space* (page 87) introduced matrices and vectors. The focus was primarily on mathematical aspects of arrays. In this section, we will consider how arrays can be used to represent something a bit more physical, the one-dimensional spatial distribution of variables. We'll look

at several features in Modelica that are related to arrays and how they allow us to compactly express behavior involving arrays.

Our problems will center around the a simple heat transfer problem. Consider a one-dimensional rod like the one shown below:



Deriving Equations

Before getting into the math, there is an important point worth making. Modelica is a language for representing **lumped** systems. What this means is that the behavior must be expressed in terms of ordinary differential equations (ODEs) or in some cases differential-algebraic equations (DAEs). But Modelica does not include any means for describing partial differential equations (*i.e.*, equations that involve the gradient of variables in spatial directions). As such, in this section we will derive the ordinary differential equations that result when we discretize a rod into discrete pieces and then model the behavior of each of these discrete (lumped) pieces.

With that out of the way, let us consider the heat balance for each discrete section of this rod. First, we have the thermal capacitance of the i^{th} section. This can expressed as:

$$m_i C T_i$$

where m is the mass of the i^{th} section, C is the capacitance (a material property) and T_i is the temperature of the i^{th} section. We can further describe the mass as:

$$m_i = \rho V_i$$

where ρ is the material density and V_i is the volume of the i^{th} section. Finally, we can express the volume of the i^{th} section as:

$$V_i = A_c L_i$$

where A_c is the cross-sectional area of the i^{th} section, which is assumed to be uniform, and L_i is the length of the i^{th} section. For this example, we will assume the rod is composed of equal size pieces. In this case, we can define the segment length, L_i , to be:

$$L_i = \frac{L}{n}$$

We will also assume that the cross-sectional area is uniform along the length of the rod. As such, the mass of each segment can be given as:

$$m = \rho A_c L_i$$

In this case, the thermal capacitance of each section would be:

$$\rho A_c L_i C T_i$$

This, in turn, means that the net heat gained in that section at any time will be:

$$\rho A_c L_i C \frac{dT_i}{dt}$$

where we assume that A_c , L_i and C don't change with respect to time.

That covers the thermal capacitance. In addition, we will consider two different forms of heat transfer. The first form of heat transfer we will consider is convection from each section to some ambient temperature, T_{amb} . We can express the amount of heat lost from each section as:

$$q_h = -hA_{s_i}(T_i - T_{amb})$$

where h is the convection coefficient and A_{s_i} is the surface area of the i^{th} section. The other form of heat transfer is conduction to neighboring sections. Here there will be two contributions, one lost to the $i - 1^{th}$ section, if it exists, and the other lost to the $i + 1^{th}$ section, if it exists. These can be represented, respectively, as:

$$q_{k_{i \rightarrow i-1}} = -kA_c \frac{T_i - T_{i-1}}{L_i}$$

$$q_{k_{i \rightarrow i+1}} = -kA_c \frac{T_i - T_{i+1}}{L_i}$$

Using these relations, we know that the heat balance for the first element would be:

$$\rho A_c L_i C \frac{dT_1}{dt} = -hA_{s_i}(T_1 - T_{amb}) - kA_c \frac{T_1 - T_2}{L_i}$$

Similarly, the heat balance for the last element would be:

$$\rho A_c L_i C \frac{dT_n}{dt} = -hA_{s_n}(T_n - T_{amb}) - kA_c \frac{T_n - T_{n-1}}{L_i}$$

Finally, the heat balance for all other elements would be:

$$\rho A_c L_i C \frac{dT_i}{dt} = -hA_{s_i}(T_i - T_{amb}) - kA_c \frac{T_i - T_{i-1}}{L_i} - kA_c \frac{T_i - T_{i+1}}{L_i}$$

Implementation

We start by defining types for the various physical quantities. This will give us the proper units and, depending on the tool, allows us to do unit checking on our equations. Our type definitions are as follows:

```
type Temperature=Real(unit="K", min=0);
type ConvectionCoefficient=Real(unit="W/K", min=0);
type ConductionCoefficient=Real(unit="W.m-1.K-1", min=0);
type Mass=Real(unit="kg", min=0);
type SpecificHeat=Real(unit="J/(K.kg)", min=0);
type Density=Real(unit="kg/m3", min=0);
type Area=Real(unit="m2");
type Volume=Real(unit="m3");
type Length=Real(unit="m", min=0);
type Radius=Real(unit="m", min=0);
```

We will also define several parameters to describe the rod we are simulating:

```
parameter Integer n=10;
parameter Length L=1.0;
parameter Radius R=0.1;
parameter Density rho=2.0;
parameter ConvectionCoefficient h=2.0;
parameter ConductionCoefficient k=10;
parameter SpecificHeat C=10.0;
parameter Temperature Tamb=300 "Ambient temperature";
```

Given these parameters, we can compute the areas and volume for each section in terms of the parameters we have already defined using the following declarations:

```
parameter Area A_c = pi*R^2, A_s = 2*pi*R*L;
parameter Volume V = A_c*L/n;
```

Finally, the only array in this problem is the temperature of each section (since this is the only quantity that actually varies along the length of the rod):

```
Temperature T[n];
```

This concludes all the declarations we need to make. Now let's consider the various equations required. First, we need to specify the initial conditions for the rod. We will assume that $T_1(0) = 200$, $T_n(0) = 300$ and the initial temperatures of all other sections can be linearly interpolated between these two end conditions. This is captured by the following equation:

```
initial equation
  T = linspace(200,300,n);
```

where the `linspace` operator is used to create an array of `n` values that vary linearly between 200 and 300. Recall from our [State Space](#) (page 87) examples that we can include equations where the left hand side and right hand side expressions are vectors. This is another example of such an equation.

Finally, we come to the equations that describe how the temperature in each section changes over time:

```
equation
  rho*V*C*der(T[1]) = -h*A_s*(T[1]-Tamb)-k*A_c*(T[1]-T[2])/(L/n);
  for i in 2:(n-1) loop
    rho*V*C*der(T[i]) = -h*A_s*(T[i]-Tamb)-k*A_c*(T[i]-T[i-1])/(L/n)-k*A_c*(T[i]-T[i+1])/(L/n);
  end for;
  rho*V*C*der(T[end]) = -h*A_s*(T[end]-Tamb)-k*A_c*(T[end]-T[end-1])/(L/n);
```

The first equation corresponds to the heat balance for section 1, the last equation corresponds to the heat balance for section n and the middle equation covers all other sections. Note the use of `end` as a subscript. When an expression is used to evaluate a subscript for a given dimension, `end` represents the size of that dimension. In our case, we use `end` to represent the last section. Of course, we could use `n` in this case, but in general, `end` can be very useful when the size of a dimension is not already associated with a variable.

Also note the use of a `for` loop in this model. A `for` loop allows the loop index variable to loop over a range of values. In our case, the loop index variable is `i` and the range of values is 2 through $n - 1$. The general syntax for a `for` loop is:

```
for <var> in <range> loop
  // statements
end for;
```

where `<range>` is a vector of values. A convenient way to generate a sequence of values is to use the range operator, `::`. The value before the range operator is the initial value in the sequence and the value after the range operator is the final value in the sequence. So, for example, the expression `5:10` would generate a vector with the values 5, 6, 7, 8, 9 and 10. Note that this **includes** the values used to specify the range.

When a `for` loop is used in an equation section, each iteration of the for loop generates a new equation for each equation inside the `for` loop. So in our case, we will generate $n - 2$ equations corresponding to values of `i` between 2 and $n - 1$.

Putting all this together, the complete model would be:

```
model Rod_ForLoop "Modeling heat conduction in a rod using a for loop"
  type Temperature=Real(unit="K", min=0);
  type ConvectionCoefficient=Real(unit="W/K", min=0);
  type ConductionCoefficient=Real(unit="W.m-1.K-1", min=0);
  type Mass=Real(unit="kg", min=0);
  type SpecificHeat=Real(unit="J/(K.kg)", min=0);
```

```

type Density=Real(unit="kg/m3", min=0);
type Area=Real(unit="m2");
type Volume=Real(unit="m3");
type Length=Real(unit="m", min=0);
type Radius=Real(unit="m", min=0);

constant Real pi = 3.14159;

parameter Integer n=10;
parameter Length L=1.0;
parameter Radius R=0.1;
parameter Density rho=2.0;
parameter ConvectionCoefficient h=2.0;
parameter ConductionCoefficient k=10;
parameter SpecificHeat C=10.0;
parameter Temperature Tamb=300 "Ambient temperature";

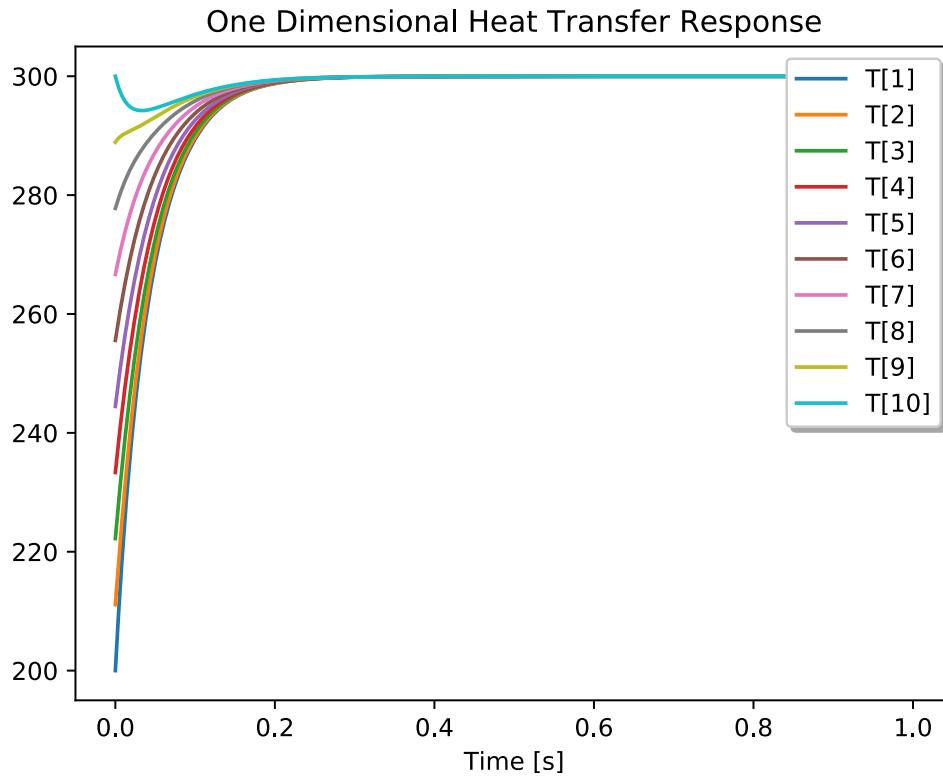
parameter Area A_c = pi*R^2, A_s = 2*pi*R*L;
parameter Volume V = A_c*L/n;

Temperature T[n];
initial equation
T = linspace(200,300,n);
equation
rho*V*C*der(T[1]) = -h*A_s*(T[1]-Tamb)-k*A_c*(T[1]-T[2])/(L/n);
for i in 2:(n-1) loop
rho*V*C*der(T[i]) = -h*A_s*(T[i]-Tamb)-k*A_c*(T[i]-T[i-1])/(L/n)-k*A_c*(T[i]-T[i+1])/(L/n);
end for;
rho*V*C*der(T[end]) = -h*A_s*(T[end]-Tamb)-k*A_c*(T[end]-T[end-1])/(L/n);
end Rod_ForLoop;

```

Note: Note that we've included `pi` as a literal constant in this model. Later in the book, we'll discuss how to properly import common *Constants* (page 163).

Simulating this model yields the following solution for each of the nodal temperatures:



Note how the temperatures are initially distributed linearly (as we specified in our `initial equation` section).

Alternatives

It turns out that there are several ways we can generate the equations we need. Each has its own advantages and disadvantages depending on the context. We'll present them here just to demonstrate the possibilities. The choice of which one they feel leads to the most understandable equations is up to the model developer.

One array feature we can use to make these equations slightly simpler is called an array comprehension. An array comprehension flips the `for` loop around so that we take a single equation and add some information at the end indicating that the equation should be evaluated for different values of the loop index variable. In our case, we can represent our equation section using array comprehensions as follows:

```
equation
  rho*V*C*der(T[1]) = -h*A_s*(T[1]-Tamb)-k*A_c*(T[1]-T[2])/(L/n);
  rho*V*C*der(T[2:n-1]) = {-h*A_s*(T[i]-Tamb)-k*A_c*(T[i]-T[i-1])/(L/n)-k*A_c*(T[i]-T[i+1])/(L/
  -n) for i in 2:(n-1)};
  rho*V*C*der(T[end]) = -h*A_s*(T[end]-Tamb)-k*A_c*(T[end]-T[end-1])/(L/n);
```

We could also combine the array comprehension with some `if` expressions to nullify contributions to the heat balance that don't necessarily apply. In that case, we can simplify the `equation` section to the point where it contains one (admittedly multi-line) equation:

```
equation
  rho*V*C*der(T) = {-h*A_s*(T[i]-Tamb)
    -(if i==1 then 0 else k*A_c/(L/n)*(T[i]-T[i-1]))
    -(if i==n then 0 else k*A_c/(L/n)*(T[i]-T[i+1])) for i in 1:n};
```

Recall, from several previous examples, that Modelica supports vector equations. In these cases, when the left hand and right hand side are vectors of the same size, we can use a single (vector) equation to represent many scalar equations. We can use this feature to simplify our equations as follows:

```
equation
  rho*V*C*der(T[1]) = -h*A_s*(T[1]-Tamb)-k*A_c*(T[1]-T[2])/(L/n);
  rho*V*C*der(T[2:n-1]) = -h*A_s*(T[i]-Tamb)-k*A_c*(T[2:n-1]-T[1:n-2])/(L/n)-k*A_c*(T[2:n-1]-
  ↵T[3:n])/(L/n);
  rho*V*C*der(T[end]) = -h*A_s*(T[end]-Tamb)-k*A_c*(T[end]-T[end-1])/(L/n);
```

Note that when a vector variable like T has a range of subscripts applied to it, the result is a vector containing the components indicated by the values in the subscript. For example, the expression $T[2:4]$ is equivalent to $\{T[2], T[3], T[4]\}$. The subscript expression doesn't need to be a range. For example, $T[\{2,5,9\}]$ is equivalent to $\{T[2], T[5], T[9]\}$.

Finally, let us consider one last way of refactoring these equations. Imagine we introduced three additional vector variables:

```
Heat Qconv[n];
Heat Qleft[n];
Heat Qright[n];
```

Then we can write these two equations (again using vector equations) to define the heat lost to the ambient, previous section and next section in the rod:

```
Qconv = {-h*A_s*(T[i]-Tamb) for i in 1:n};
Qleft = {(if i==1 then 0 else -k*A_c*(T[i]-T[i-1])/(L/n)) for i in 1:n};
Qright = {(if i==n then 0 else -k*A_c*(T[i]-T[i+1])/(L/n)) for i in 1:n};
```

This allows us to express the heat balance for each section using a vector equation that doesn't include any subscripts:

```
rho*V*C*der(T) = Qconv+Qleft+Qright;
```

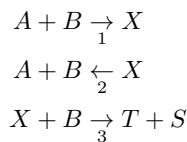
Conclusion

In this section, we've seen various ways that we can use vector variables and vector equations to represent one-dimensional heat transfer. Of course, this vector related functionality can be used for a wide range of different problem types. The goal of this section was to introduce several features to demonstrate the various options that are available to the developer when working with vectors.

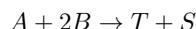
Chemical System

In this section, we'll consider a few different ways to describe the behavior of a chemical system. We'll start by building a model without using the array functionality. Then, we'll implement the same behavior using vectors. Finally, we'll implement the same model again using enumerations.

In all of our examples, we'll be building a model for the following system of reactions¹³:



It should be noted that X is simply an intermediate result of this reaction. The overall reaction can be expressed as:



¹³ <http://library.wolfram.com/infocenter/TechNotes/390/>

Using the law of mass action we can transform these chemical equations into the following mathematical ones:

$$\begin{aligned}\frac{d[A]}{dt} &= -k_1[A][B] + k_2[X] \\ \frac{d[B]}{dt} &= -k_1[A][B] + k_2[X] - k_3[B][X] \\ \frac{d[X]}{dt} &= k_1[A][B] - k_2[X] - k_3[B][X]\end{aligned}$$

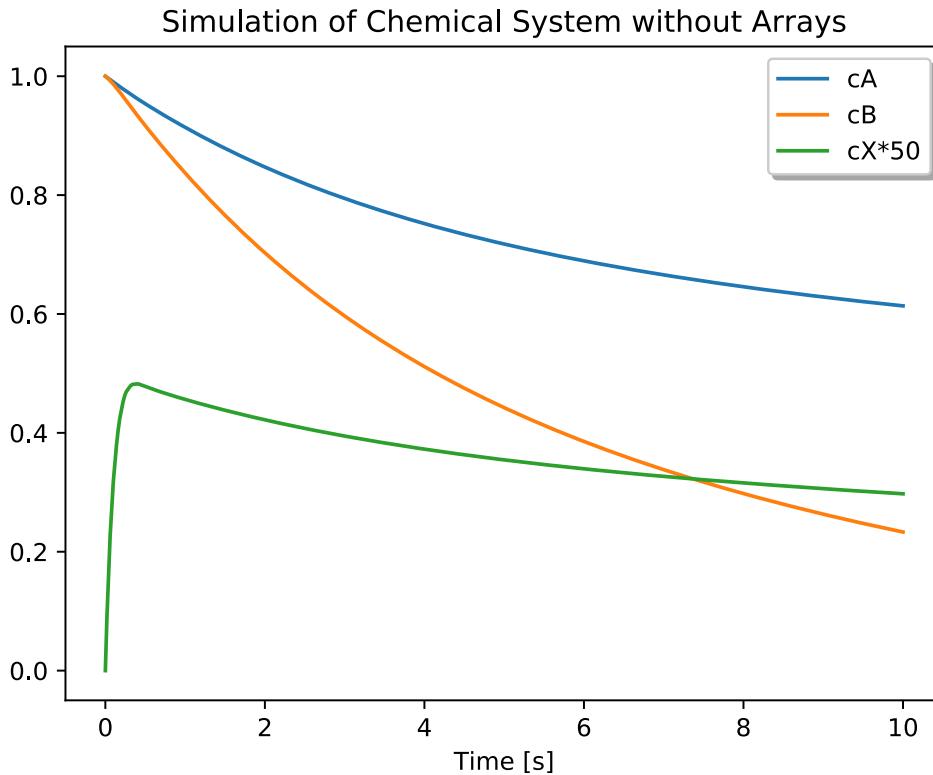
where k_1 , k_2 and k_3 are the reaction coefficients for the first, second and third reactions, respectively. These equations are derived by considering the change in each species due to each reaction involving that species. So, for example, since the first reaction $A + B \rightarrow X$ transforms molecules of A and B into molecules of X , we see the term $-k_1[A][B]$ in the balance equation for A , which represents the reduction in the amount of A as a result of that reaction. Each term in these balance equations is derived in a similar fashion.

Without Arrays

Let us start with an approach that doesn't utilize arrays at all. In this case, we simply represent the concentrations $[A]$, $[B]$ and $[X]$ by the variables cA , cB and cX as follows:

```
model Reactions_NoArrays "Modeling a chemical reaction without arrays"
  Real cA;
  Real cB;
  Real cX;
  parameter Real k1=0.1;
  parameter Real k2=0.1;
  parameter Real k3=10;
initial equation
  cA = 1;
  cB = 1;
  cX = 0;
equation
  der(cA) = -k1*cA*cB + k2*cX;
  der(cB) = -k1*cA*cB + k2*cX - k3*cB*cX;
  der(cX) = k1*cA*cB - k2*cX - k3*cB*cX;
end Reactions_NoArrays;
```

With this approach, we create an equation for the balance of each species. If we simulate this model, we get the following results:



Using Arrays

Another way to approach modeling of the chemical system is to use vectors. With this approach, we associate the species *A*, *B* and *X* with the indices 1, 2 and 3, respectively. The concentrations are mapped to the vector variable *C*. We can also cast the reaction coefficients into a vector of reaction coefficients, *k*.

With this transformation, all the equations are then transformed into vector equations:

```
model Reactions_Array "Modeling a chemical reaction with arrays"
  Real C[3];
  parameter Real k[3] = {0.1, 0.1, 10};
initial equation
  C = {1, 1, 0};
equation
  der(C) = {-k[1]*C[1]*C[2] + k[2]*C[3],
             -k[1]*C[1]*C[2] + k[2]*C[3] - k[3]*C[2]*C[3],
             k[1]*C[1]*C[2] - k[2]*C[3] - k[3]*C[2]*C[3]};
end Reactions_Array;
```

The reaction equations are non-linear, so they cannot be transformed into a completely linear form. But we could simplify them further by using a matrix-vector product. In other words, the equations:

$$\begin{aligned}\frac{d[A]}{dt} &= -k_1[A][B] + k_2[X] \\ \frac{d[B]}{dt} &= -k_1[A][B] + k_2[X] - k_3[B][X] \\ \frac{d[X]}{dt} &= k_1[A][B] - k_2[X] - k_3[B][X]\end{aligned}$$

can be transformed into the following form:

$$\frac{d}{dt} \begin{Bmatrix} [A] \\ [B] \\ [X] \end{Bmatrix} = \begin{bmatrix} -k_1[B] & 0 & k_2 \\ -k_1[B] & -k_3[X] & k_2 \\ k_1[B] & -k_3[X] & -k_2 \end{bmatrix} \begin{Bmatrix} [A] \\ [B] \\ [X] \end{Bmatrix}$$

which can then be represented in Modelica as:

```
der(C) = [-k[1]*C[2], 0, k[2];
           -k[1]*C[2], -k[3]*C[3], k[2];
           k[1]*C[2], -k[3]*C[3], -k[2]]*C;
```

The drawback of this approach is that we have to constantly keep track of which index (*e.g.*, 1, 2, or 3) corresponds to which species (*e.g.*, A, B, or X).

Using Enumerations

To address this issue of having to map back and forth from numbers to names, our third approach will utilize the `enumeration` type in Modelica. An enumeration allows us to define a set of names which we can then use to define the subscripts associated with an array. We'll define our enumeration as follows:

```
type Species = enumeration(A, B, X);
```

This defines a special type named `Species` that has exactly three possible values, A, B and X. We can then use this enumeration **as a dimension in an array** as follows:

```
Real C[Species];
```

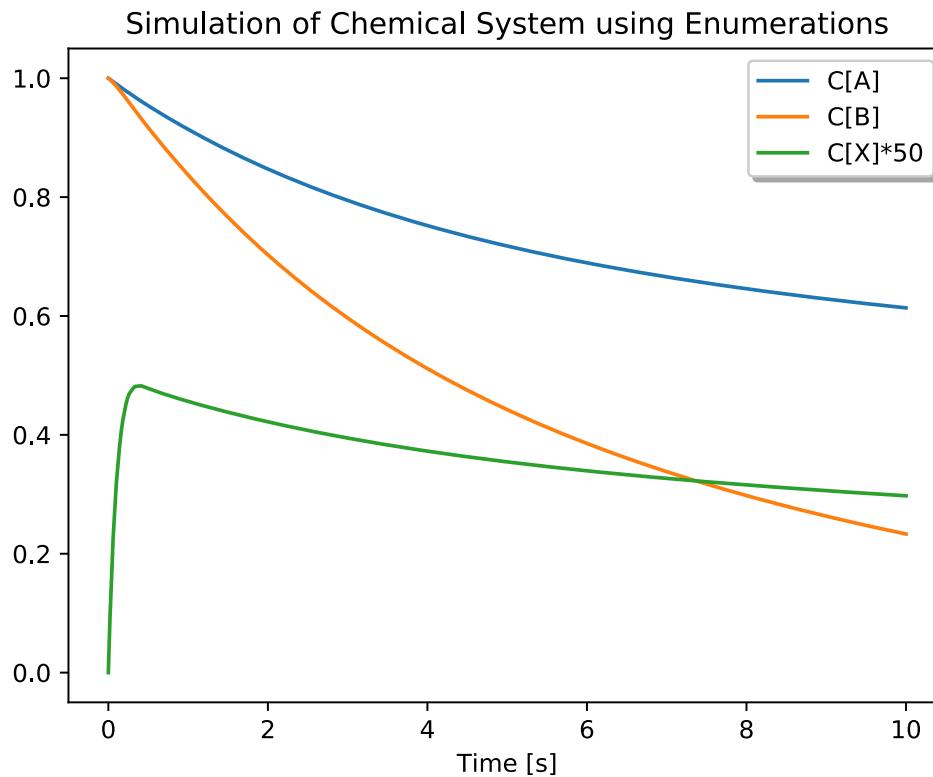
Since the `Species` type has only three possible values, this means that the vector C has exactly three components. We can then refer to the individual components of C as `C[Species.A]`, `C[Species.B]` and `C[Species.X]`.

Because it is awkward to constantly prefix each species name with `Species`, we can define a few convenient constants as follows:

```
constant Species A = Species.A;
constant Species B = Species.B;
constant Species X = Species.X;
```

In this way, we can now refer to the concentration of species A as `C[A]`. Pulling all of this together we can represent our chemical system using enumerations as:

```
model Reactions_Enum "Modeling a chemical reaction with enums"
  type Species = enumeration(
    A,
    B,
    X);
  Real C[Species] "Species concentrations";
  parameter Real k[3] = {0.1, 0.1, 10};
  constant Species A = Species.A;
  constant Species B = Species.B;
  constant Species X = Species.X;
  initial equation
    C[A] = 1.0;
    C[B] = 1.0;
    C[X] = 0.0;
  equation
    der(C[A]) = -k[1]*C[A]*C[B] + k[2]*C[X];
    der(C[B]) = -k[1]*C[A]*C[B] + k[2]*C[X] - k[3]*C[B]*C[X];
    der(C[X]) = k[1]*C[A]*C[B] - k[2]*C[X] - k[3]*C[B]*C[X];
  end Reactions_Enum;
```



Conclusion

In this chapter, we showed how a set of chemical equations could be represented with and without arrays. We also demonstrated how the `enumeration` type can be used in conjunction with arrays to make the resulting equations more readable by replacing numeric indices with names. Furthermore, this section also demonstrated how the `enumeration` type can be used not only to index the array, but also to define one or more dimensions in the declaration.

1.3.2 Review

Array Declarations

Syntax

Array declaration syntax is very simple. The syntax is the same as for a normal variable declaration except the variable name should be followed by subscripts to specify the size of each dimension of the array. The general form for an array declaration would be:

```
VariableType varName[dim1, dim2, ..., dimN];
```

where `VariableType` is a Modelica type like `Real` or `Integer`, `varName` is the name of the variable.

Integer Sizes

Normally, the dimension specifications are simply integers that indicate the size of that dimension. For example:

```
Real x[5];
```

In this case, `x` is an array of real valued numbers with only one dimension of size 5. It is possible to use parameters or constants specify the size of an array, *e.g.*,

```
parameter Integer d1=5;
constant Integer d2=2;
Real x[d1, d2];
```

Linked Dimensions

As we will see shortly when we discuss the various *Array Functions* (page 105) in Modelica, we can even use the `size` function to specify the size of one array in terms of another array. Consider the following:

```
Real x[5];
Real y[size(x,1)];
```

In this case, `y` will have one dimension of size 5. The use of the function `size(x, 1)` will return the size of dimension 1 of the array `x`. There are many applications where it is useful to express that the dimensions of different arrays are related in this way (*e.g.*, ensuring that arrays are sized such that operations like matrix multiplication are possible).

Unspecified Dimensions

There are some circumstances where we can leave the size of an array unspecified so that it can be specified by some later context. For example, we will see examples of this later when we discuss *Functions* (page 116) that have arguments which are arrays.

To indicate that the size of a given array dimension is not (yet) known, we can use the `:` symbol as the dimension. So in a declaration like this:

```
Real A[:,2];
```

we are declaring an array with two dimensions. The size of the first dimension is not specified. However, the size of the second dimension is definitively specified as 2. In effect, we have declared that `A` is a matrix with an unspecified number of rows and two columns.

Non-Integer Dimensions

Enumerations

As we saw in our *Chemical System* (page 97) examples, another way to specify the dimension for an array is with an enumeration. If an enumeration is used to specify a dimension, then the size of that dimension will be equal to the number of possible values for that enumeration. In our forthcoming discussion on *Array Indexing* (page 113), we'll see how to properly index an array that uses enumerations as dimensions.

Booleans

It is also possible to declare an array where a dimension is specified as `Boolean`, *e.g.*,

```
Real x[Boolean];
```

Array Construction

Now that we know *how to declare that a variable is an array* (page 101), the next step is filling in all the elements of those arrays. There are many different ways to construct arrays in Modelica.

Literals

Vectors

The simplest method for constructing an array is to enumerate each of the individual elements. For example, given the following parameter declaration for a variable named `x` meant to represent a vector:

```
parameter Real x[3];
```

When we use the term “vector” here, we are referring to an array that has only one subscript dimension. If we wanted to assign a value to this vector, we could do so as follows:

```
parameter Real x[3] = {1.0, 0.0, -1.0};
```

Clearly, the variable `x` is (declared to be) a vector with three real valued components. For consistency, the right hand side must also be a vector with three real valued components. Fortunately, it is. The expression `{1.0, 0.0, -1.0}` is a special syntax in Modelica for constructing vectors. We can use this syntax of a pair of {} containing a comma separated list of expressions to build vectors of any size we wish, *e.g.*,

```
parameter Real x[5] = {1.0, 0.0, -1.0, 2.0, 0.0};
```

While it is possible to use the {} notation to construct arrays of any dimension, *e.g.*,

```
parameter Real B[2,3] = {{1.0, 2.0, 3.0}, {5.0, 6.0, 7.0}};
```

Range Notation

Modelica includes a shorthand notation for constructing vectors of sequential numbers or numbers that are evenly spaced. For example, to construct a vector of integers with elements having values from 1 to 5, the following syntax can be used:

```
1:5 // {1, 2, 3, 4, 5}
```

The same syntax can be used to construct arrays of floating point numbers:

```
1.0:5.0 // {1.0, 2.0, 3.0, 4.0, 5.0}
```

Note, care should be taken when vectors of reals in this way since issues with floating point representations may result in the vector not including the final value. The following alternatives are also available (and probably more robust):

```
1.0*(1:5) // {1.0, 2.0, 3.0, 4.0, 5.0}
{1.0*i for i in 1:5} // {1.0, 2.0, 3.0, 4.0, 5.0}
```

It is also possible to construct ranges where the interval between values is not 1 by adding the “stride” between the first and last values. For example, all odd numbers between 3 and 9 can be represented as:

```
3:2:9 // {3, 5, 7, 9}
```

It is also possible to insert a stride value when dealing with floating point numbers as well. This range notation can also be used with an `enumeration` type (but a stride value is not permitted in that case).

Matrix Construction

But it is important to note that there is also a special syntax used for constructing matrices (arrays with exactly two subscript dimensions). Consider the following parameter declarations with initializer:

```
parameter Real B[2,3] = [1.0, 2.0, 3.0; 5.0, 6.0, 7.0];
```

In this case, the parameter **B** is equivalent to the following in mathematical notation:

$$B = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 5.0 & 6.0 & 7.0 \end{bmatrix}$$

As we can see in both the Modelica code and the more mathematical representation, the matrix **B** has two rows and three columns. The syntax for building arrays in this way is a bit more complicated than building vectors. Superficially, we see that while a vector is surrounded by {}, a matrix is surrounded by []. But more importantly, a mixture of commas **and semicolons** are used as delimiters. The semicolons are used to separate rows and the commas are used to separate the columns.

One nice feature about this matrix construction notation is that it is possible to embed vectors or submatrices.

Vectors

When embedding vectors, it is very important to note that **vectors are treated as column vectors**. In other words, in the context of matrix construction, a vector of size n is treated as a matrix with n rows and 1 column.

To demonstrate how this embedding is done, consider the case where we wished to construct the following matrix:

$$C = \left[\begin{array}{c|c|c|c} 2 & 1 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right]$$

We can do this concisely in Modelica by first creating each of the submatrices and then filling in **C** using these submatrices as follows:

```
parameter D[2,2] = [2, 1; 1, 2];
parameter Z[2,2] = [0, 0; 0, 0];
parameter C[6,6] = [D, Z, Z;
                  Z, D, Z;
                  Z, Z, D];
```

In other words, the , and ; delimiters work with either scalars or submatrices.

As we will see shortly, there are several different *Array Construction Functions* (page 105) that can be extremely useful when building matrices in this way.

Arrays of Any Size

So far, we've discussed vectors and matrices. But you can construct arbitrary arrays with any number of dimensions (including vectors and matrices) using by constructing them as a series of nested vectors. For example, to construct an array with three dimensions, we could simply nested a collection of vectors as follows:

```
parameter Real A[2,3,4] = { { {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 8, 7, 6} },
    { {4, 3, 2, 1},
    {8, 7, 6, 5},
    {4, 3, 2, 1} } };
```

As can be seen in this example, the inner most elements in this nested construction correspond to the right most dimension in the declaration. In other words, the array here is a vector containing two elements where each of those two elements is a vector containing three elements and each of those three elements is a vector of 4 scalars.

Array Comprehensions

So far, we've shown how to construct vectors, matrices and higher dimensional arrays by enumerating the elements contained in the array. As we can see in the case of higher dimensional arrays, these constructions can get very complicated. Fortunately, Modelica includes array comprehensions which provide a convenient syntax for programmatically constructing arrays.

The use of array comprehensions has several benefits. The first is that it is a much more compact notation. The second is that it allows us to easily express how the values in the array are tied to the various indices. The third is that it can be done in a context where an expression is required (typically providing values for variables in variable declarations). Finally, some tools may find it easier to optimize array comprehensions.

To demonstrate array comprehensions, consider the following relationship between elements in an array and the indices of the array:

$$a_{ijk} = i \ x_j \ y_k$$

where x and y are vectors. We've already seen how we could recursively define such an array using a series of nested vectors. But we have also seen how long such an expression could potentially be and how tedious it is to read and write. Using array comprehensions, we can construct the a array quite easily as:

```
parameter Real a[10,12,15] = {i*x[j]*y[k] for k in 1:15,
                                j in 1:12,
                                i in 1:10};
```

This code builds an array with 1800 elements with only a few lines of Modelica code.

Array Functions

There are a great many functions in Modelica that are related to arrays. In this section, we'll go through different categories of functions and describe how they are used.

Array Construction Functions

We already talked about [Array Construction](#) (page 103). We saw the different syntactic constructs that can be used to build vectors and matrices. Furthermore, we saw how matrices can be built from other matrices. There are several functions in Modelica that can be used for constructing vectors, matrices and higher-dimension arrays as both an alternative or complement to those previously presented.

`fill`

The `fill` function is used to create an array where each element in the array has the same value. The arguments for `fill` are:

```
fill(v, d1, d2, ..., dN)
```

where `v` is the value to be given to each element in the array and the remaining arguments are the sizes of each dimension. The elements in the resulting array will have the same type as `v`. So, to fill a 5x7 array of reals with the value 1.7, we could use the following:

```
parameter Real x[5,7] = fill(1.7, 5, 7);
```

This would result in a matrix filled as follows:

$$\begin{bmatrix} 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 \\ 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 \\ 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 \\ 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 \\ 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 & 1.7 \end{bmatrix}$$

`zeros`

When working with arrays, a common use case is to create an array that contains only zero elements. This is essentially the same functionality as the `fill` function, but since the value is known it is only necessary to specify the sizes. Using `zeros` we can initialize an array as follows:

```
parameter Real y[2,3,5] = zeros(2, 3, 5);
```

`ones`

The `ones` function is identical to the `zeros` function except that every element in the resulting array has the value 1. So, for example:

```
parameter Real z[3,5] = ones(3, 5);
```

This would result in a matrix filled as follows:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

`identity`

Another common need is to easily build an identity matrix, one whose diagonal elements are all 1 while all other elements are 0. This can be done very easily with the `identity`. The `identity` function takes a single integer argument. This argument determines the number of rows and columns in the resulting matrix. So, invoking `identity` as:

```
identity(5);
```

would produce the following matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

diagonal

The **diagonal** function is used to create a matrix where all non-diagonal elements are 0. The only argument to **diagonal** is an array containing the values of the diagonal elements. So, to create the following diagonal matrix:

$$\begin{bmatrix} 2.0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 \\ 0 & 0 & 4.0 & 0 \\ 0 & 0 & 0 & 5.0 \end{bmatrix}$$

one could use the following Modelica code:

```
diagonal({2.0, 3.0, 4.0, 5.0});
```

linspace

The **linspace** function builds a vector where the values of the elements are all linearly distributed over an interval. The **linspace** function is invoked as follows:

```
linspace(v0, v1, n);
```

where **v0** is the value of the first elements in the vector, **v1** is the last element in the vector and **n** is the total number of values in the vector. So, for example, invoking **linspace** as:

```
linspace(1.0, 5.0, 9);
```

would yield the vector:

```
{1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0}
```

Conversion Functions

The following functions provide a means to transform arrays into other arrays.

scalar

The **scalar** function is invoked as follows:

```
scalar(A)
```

where **A** is an array with an arbitrary number of dimensions as long as each dimension is of size 1. The **scalar** function returns the (only) scalar value contained in the array. For example,

```
scalar([5]) // Argument is a two-dimensional array (matrix)
```

and

```
scalar({5}) // Argument is a one-dimensional array (vector)
```

would both give the scalar value 5.

vector

The **vector** function is invoked as follows:

```
vector(A)
```

where A is an array with an arbitrary number of dimensions as long as only one dimension has a size greater than 1. The `vector` function returns the contents of the array as a vector (*i.e.*, an array with only a single dimension). So, for example, if we passed a column or row matrix, *e.g.*,

```
vector([1;2;3;4]) // Argument is a column matrix
```

or

```
vector([1,2,3,4]) // Argument is a row matrix
```

we would get back:

```
{1,2,3,4}
```

`matrix`

The `matrix` function is invoked as follows:

```
matrix(A)
```

where A is an array with an arbitrary number of dimensions as long as only two dimension have a size greater than 1. The `matrix` function returns the contents of the array as a matrix (*i.e.*, an array with only two dimensions).

Mathematical Operations

In linear algebra, there are many different types of mathematical operations that are commonly performed on vectors and matrices. Modelica provides functions to perform most of these operations. In this way, Modelica equations can be made to look very much like their mathematical counterparts in linear algebra.

Let's start with operations like addition, subtraction, multiplication, division and exponentiation. For the most part, these operations work just as they do in mathematics when applied to the various combinations of scalars, vectors and matrices. However, for completeness and reference, the following tables summarize how these operations are defined.

Explanation of Notation

Each of the operations described below involves two arguments, a and b , and a result, c . If an argument represents a scalar, it will have no subscripts. If it is a vector, it will have one subscript. If it is a matrix, it will have two subscripts. If the operation is defined for arbitrary arrays, a case will be included with three subscripts. If a given combination is not shown, then it is not allowed.

Addition (+)

Expression	Result
$a + b$	$c = a + b$
$a_i + b_i$	$c_i = a_i + b_i$
$a_{ij} + b_{ij}$	$c_{ij} = a_{ij} + b_{ij}$
$a_{ijk} + b_{ijk}$	$c_{ijk} = a_{ijk} + b_{ijk}$

Subtraction (-)

Expression	Result
$a - b$	$c = a - b$
$a_i - b_i$	$c_i = a_i - b_i$
$a_{ij} - b_{ij}$	$c_{ij} = a_{ij} - b_{ij}$
$a_{ijk} - b_{ijk}$	$c_{ijk} = a_{ijk} - b_{ijk}$

Multiplication (* and .*)

There are two types of multiplication operators. The first is the normal multiplication operator, $*$, that follows the usual mathematical conventions of linear algebra that matrix-vector products, *etc.*. The behavior of the $*$ operator is summarized in the following table:

Expression	Result
$a * b$	$c = a * b$
$a * b_i$	$c_i = a * b_i$
$a * b_{ij}$	$c_{ij} = a * b_{ij}$
$a * b_{ijk}$	$c_{ijk} = a * b_{ijk}$
$a_i * b$	$c_i = a_i * b$
$a_{ij} * b$	$c_{ij} = a_{ij} * b$
$a_{ijk} * b$	$c_{ijk} = a_{ijk} * b$
$a_i * b_i$	$c = \sum_i a_i * b_i$
$a_i * b_{ij}$	$c_j = \sum_i a_i * b_{ij}$
$a_{ij} * b_j$	$c_i = \sum_j a_{ij} * b_j$
$a_{ik} * b_{kj}$	$c_{ij} = \sum_k a_{ik} * b_{kj}$

The second type of multiplication operator is a special element-wise version, $.*$, which doesn't perform any summations and simply applies the operator element-wise to all array elements.

Expression	Result
$a .* b$	$c = a * b$
$a_i .* b_i$	$c_i = a_i * b_i$
$a_{ij} .* b_{ij}$	$c_{ij} = a_{ij} * b_{ij}$
$a_{ijk} .* b_{ijk}$	$c_{ijk} = a_{ijk} * b_{ijk}$

Division (/ and ./)

As with *Multiplication (* and .*)* (page 109), there are two division operators. The first is the normal division operator, $/$, which can be used to divide arrays by a scalar value. The following table summarizes its behavior:

Expression	Result
a/b	$c = a/b$
a_i/b	$c_i = a_i/b$
a_{ij}/b	$c_{ij} = a_{ij}/b$
a_{ijk}/b	$c_{ijk} = a_{ijk}/b$

In addition, there is also an element-wise version of the division operator, $./$, whose behavior is summarized in the following table:

Expression	Result
$a ./ b$	$c = a/b$
$a_i ./ b_i$	$c_i = a_i/b_i$
$a_{ij} ./ b_{ij}$	$c_{ij} = a_{ij}/b_{ij}$
$a_{ijk} ./ b_{ijk}$	$c_{ijk} = a_{ijk}/b_{ijk}$

Exponentiation (\wedge and $.\wedge$)

As with *Multiplication (* and .*)* (page 109) and *Division (/ and ./)* (page 109), the exponentiation operator comes in two forms. The first is the standard exponentiation operator, \wedge . The standard version can be used in two different ways. The first is to raise one scalar to the power of another (*i.e.*, $a \wedge b$). The other is to raise a square matrix to a scalar power (*i.e.*, $a_{ij} \wedge b$).

The other form of exponentiation is the element-wise form indicated with the $.\wedge$ operator. Its behavior is summarized in the following table:

Expression	Result
$a .\wedge b$	$c = a^b$
$a_i .\wedge b_i$	$c_i = a_i^{b_i}$
$a_{ij} .\wedge b_{ij}$	$c_{ij} = a_{ij}^{b_{ij}}$
$a_{ijk} .\wedge b_{ijk}$	$c_{ijk} = a_{ijk}^{b_{ijk}}$

Equality (=)

The equality operator, $=$ used to construct equations can be used with scalars as well as arrays **as long as the left hand side and right hand side have the same number of dimensions and the sizes of each dimension are the same**. Assuming this requirement is met, then the operator is simply applied element-wise. This means that the operator is applied between each element on the left hand side and its counterpart on the right hand side.

Assignment (:=)

The $:=$ (assignment) operator is applied in the same element-wise way as the *Equality (=)* (page 110) operator.

Relational Operators

All relational operators (**and**, **or**, **not**, $>$, \geq , $<$, \leq) are applied in the same element-wise way as the *Equality (=)* (page 110) operator.

transpose

The **transpose** function takes a matrix as an argument and returns a transposed version of that matrix.

outerProduct

The **outerProduct** function takes two arguments. Each argument must be a vector and they must have the same size. The function returns a matrix which represents the outer product of the two vectors.

Mathematically speaking, assume a and b are vectors of the same size. Invoking `outerProduct(a,b)` will return a matrix, c , whose elements are defined as:

$$c_{ij} = a_i * b_j$$

`symmetric`

The `symmetric` function takes a square matrix as an argument. It returns a matrix of the same size where all the elements below the diagonal of the original matrix have been replaced by elements transposed from above the diagonal. In other words,

$$b_{ij} = \text{symmetric}(a) = \begin{cases} a_{ij} & \text{if } i \leq j \\ a_{ji} & \text{otherwise} \end{cases}$$

`skew`

The `skew` function takes a vector with three components and returns the following skew-symmetric matrix:

$$\text{skew}(x) = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}$$

`cross`

The `cross` function takes two vectors (each with 3 components) and returns the following vector (with three components):

$$\text{cross}(x, y) = \begin{Bmatrix} x_2y_3 - x_3y_2 \\ x_3y_1 - x_1y_3 \\ x_1y_2 - x_2y_1 \end{Bmatrix}$$

Reduction Operators

Reduction operators are ones that reduce arrays down to scalar values.

`min`

The `min` function takes an array and returns the smallest value in the array. For example:

```
min({10, 7, 2, 11}) // 2
min([1, 2; 3, -4]) // -4
```

`max`

The `max` function takes an array and returns the largest value in the array. For example:

```
max({10, 7, 2, 11}) // 11
max([1, 2; 3, -4]) // 3
```

sum

The **sum** function takes an array and returns the sum of all elements in the array. For example:

```
sum({10, 7, 2, 11}) // 30
sum([1, 2; 3, -4]) // 2
```

product

The **product** function takes an array and returns the product of all elements in the array. For example:

```
product({10, 7, 2, 11}) // 1540
product([1, 2; 3, -4]) // -24
```

Miscellaneous Functions**ndims**

The **ndims** function takes an array as its argument and returns the number of dimensions in that array. For example:

```
ndims({10, 7, 2, 11}) // 1
ndims([1, 2; 3, -4]) // 2
```

size

The **size** function can be invoked two different ways. The first way is with a single argument that is an array. In this case, **size** returns a vector where each component in the vector corresponds to the size of the corresponding dimension in the array. For example:

```
size({10, 7, 2, 11}) // {4}
size([1, 2, 3; 3, -4, 5]) // {2, 3}
```

It is also possible to call **size** with an optional additional argument indicating a specific dimension number. In that case, it will return the size of that specific dimension as a scalar integer. For example,

```
size({10, 7, 2, 11}, 1) // 4
size([1, 2, 3; 3, -4, 5], 1) // 2
size([1, 2, 3; 3, -4, 5], 2) // 3
```

Vectorization

In this section, we've discussed the numerous functions in Modelica that are designed to work with arguments that are arrays. But a very common use case is to apply a function element-wise to every element in a vector. Modelica supports this use case through a feature called "vectorization". If a function is designed to take a scalar, but is passed an array instead, the Modelica compiler will automatically apply that function to each element in the vector.

To understand how this works, first consider a normal evaluation using the **abs** function:

```
abs(-1.5) // 1.5
```

Obviously, **abs** is normally meant to accept a scalar argument and return a scalar. But in Modelica, we can also do this:

```
abs({0, -1, 1, -2, 2}) // {0, 1, 1, 2, 2}
```

Since this function is designed for scalar, the Modelica compiler will transform:

```
abs(0, -1, 1, -2, 2)
```

into

```
{abs(0), abs(-1), abs(1), abs(-2), abs(2)}
```

In other words, it transforms the function applies to a vector of scalars into a vector a functions applied to scalar.

This feature also works functions that take multiple arguments as long as only **one** of the expected scalar arguments is a vector. To understand this slightly more complex functionality, consider the modulo function, `mod`. If applied to scalar arguments we get the following behavior:

```
mod(5, 2) // 1
```

If we turn the first argument into a vector, we get:

```
mod({5, 6, 7, 8}, 2) // {1, 0, 1, 0}
```

In other words, it transforms:

```
mod({5, 6, 7, 8}, 2)
```

into

```
{mod(5,2), mod(6,2), mod(7,2), mod(8,2)}
```

However, this vectorization does **not** apply if more than one scalar arguments is presented as a vector. For example, the following expression will be an error:

```
mod({5, 6, 7, 8}, {2, 3}) // Illegal
```

because `mod` expects two scalar arguments, but it was passed two vector arguments.

Array Indexing

We've seen many examples in this chapter showing how arrays are indexed. So it might not seem necessary to have a section devoted to discussing how to index arrays. It is true that normally you would simply reference elements in an array using integer values for each subscript. But there are enough other ways to index arrays that it is worth spending some time to talk about them.

Indices

Integers

1-index

For array dimensions specified using integers, Modelica uses indices starting with **1**. Some languages choose to use zero as the starting index, but it is important to point out from the start that Modelica follows the 1-index convention.

Obviously, the most directly way to index an array is to use an integer. For an array declared as:

```
Real x[5,4];
```

we can index elements of the array by providing an integer between 1 and 5 for the first subscript and 1 and 4 for the second subscript.

But it is worth pointing out that Modelica allows the subscripts to be vectors. To understand how vector indices work, first consider the following matrix:

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

In Modelica, such an array would be declared as follows:

```
parameter Real B[3,3] = [1, 2, 3; 4, 5, 6; 7, 8, 9];
```

Imagine we wish to extract a submatrix of B as follows:

```
parameter Real C[2,2] = [B[1,1], B[1,2]; B[2,1], B[2,2]]; // [1, 2; 4, 5];
```

We could extract the same submatrix more easily using vector subscripts as follows:

```
parameter Real C[2,2] = B[{1,2},{1,2}]; // [1, 2; 4, 5];
```

By using vector subscripts we can extract or construct arbitrary sub-arrays. This is where *Range Notation* (page 103) can be very useful. The same submatrix extraction could also be represented as:

```
parameter Real C[2,2] = B[1:2,1:2]; // [1, 2; 4, 5];
```

Enumerations

In our *Chemical System* (page 97) examples, we saw how enumerations can be used to specify array dimensions. Furthermore, we saw how the values specified by an `enumeration` type can be used to index the array. In general, for an `enumeration` like the following:

```
type Species = enumeration(A, B, X);
```

and then declare an array where that `enumeration` is used to specify a dimension, *e.g.*,

```
Real C[Species];
```

then we can use the enumeration values, `Species.A`, `Species.B` and `Species.X` as indices. For example,

```
equation  
  der(C[Species.A]) = ...;
```

Booleans

We can use the `Boolean` type in much the same way as an `enumeration`. Given an array declared with `Boolean` for a dimension:

```
Real C[5,Boolean];
```

We can then use boolean values to index that dimension, *e.g.*,

```
equation  
  der(C[1,true]) = ...;  
  der(C[1,false]) = ...;
```

```
end
```

When specifying a subscript for an array, it is legal to use `end` in the subscript expression. In this context, `end` will take on the value of the highest possible value for the corresponding array dimension. The use of `end` within expressions allows easy reference to array elements with respect to the last element rather than the first. For example, to reference the second from the last element in a vector, the expression `end-1` can be used a subscript.

Remember that `end` takes on the value of the highest possible index for the **corresponding array dimension**. So for the following array:

```
Integer B[2,4] = [1, 2, 3, 4; 5, 6, 7, 8];
```

The following expressions would evaluate as follows:

```
B[1,end]    // 4
B[end,1]    // 5
B[end,end]  // 8
B[2,end-1]  // 7
```

Slicing

There is another sophisticated way of indexing arrays in Modelica. But it doesn't make sense to talk about it just yet. We will see it later when we start our discussion of *Arrays of Component* (page 300).

Looping

```
for
```

One of the main uses of arrays is to allow code to be simplified through the use of loops. So we will conclude this chapter on arrays by introducing some basic looping constructs and showing how they are used in conjunction with array features.

In general, the `for` keyword is used to represent looping. But there are many different contexts in which `for` can be used. Several of the examples in this chapter used `for` to generate a collection of equations. When `for` is used within an equation section, any equations contained within the `for` loop are generated **for each value of the loop index variables**. In this way, we can easily generate many equations that have the same overall structure and only vary by the value of the loop index variable. The general syntax for a `for` loop in an equation section is:

```
equation
  for i in 1:n loop
    // equations
  end for;
```

Note that the loop index variable (*e.g.*, `i` in this case) **does not have to be declared**. It is also worth noting that these variables only exists within the scope of the `for` loop (not before or after the loop).

For loops can, of course, be nested. For example:

```
equation
  for i in 1:n loop
    for j in 1:n loop
      // equation
    end for;
  end for;
```

They can also appear in other contexts. For example, they can appear in `initial` `equation` sections or in *algorithm Sections* (page 87).

Another case where the `for` keyword can be seen is in our discussion of *Array Comprehensions* (page 105). In that case, the `for` construct is not used to generate equations or statements, but to populate the various elements in an array. Array comprehensions have the advantage that they may be more easily for tools to optimize.

`while`

There is another type of loop in Modelica and that is the `while` loop. The `while` loop is not used very often in Modelica. The reason is that Modelica, unlike a general purpose language, is an equation oriented language. Furthermore, it imposes a requirement that a model should include an equal number of equations and unknowns. Such a model is considered a “balanced model”.

The reason that the `while` construct is not widely used is because a balanced model requires that the number of equations is predictable (by the compiler). Because a `for` loop is bounded and the number of values of the index variable is always known (because it is always derived from a vector of possible values), the number of equations it will generate is always known. The same cannot be said of a `while` loop. As such, `while` loops are only practical in the context of *algorithm Sections* (page 87) (typically in the definition of *Functions* (page 116)).

1.4 Functions

Up to this point, we’ve used many of the built-in functions in Modelica in previous examples and particularly in our discussion of *Array Functions* (page 105). While Modelica includes an extensive set of functions for many common calculations, users still need the ability to define their own functions. The process of defining functions is the central topic of this chapter.

As always, we’ll start with several examples to motivate the need for user defined functions. Then we’ll review the important elements of user defined functions in Modelica.

1.4.1 Examples

Polynomial Evaluation

Our first example will center around using functions to evaluate polynomials. This will help use understand the basics of defining and using functions.

Computing a Line

Mathematical Background

Before diving until polynomials of arbitrary order, let’s first consider how we could use a function to evaluate points on a line. Mathematically, what we’d like to define is a function that is applied as follows:

$$y(x, x_0, y_0, x_1, y_1)$$

where x is the independent variable, (x_0, y_0) is one point that defines the line and (x_1, y_1) is the other point that defines the line. Mathematically, such a function could be defined as follows:

$$y(x, x_0, y_0, x_1, y_1) = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0$$

To reduce the number of arguments, let’s assume that combine x_0 and y_0 into a single point represented by the vector \vec{p}_0 and we combine x_1 and y_1 into a single point represented by the vector \vec{p}_1 so that the function is now invoked as:

$$\text{Line}(x, \vec{p}_0, \vec{p}_1)$$

Modelica Representation

The question now is how can we transform this mathematical relationship into a function that we can invoke from within a Modelica model. To do this, we must define a new Modelica function.

It turns out that a function definition is very similar (syntactically, at least) to a *Model Definition* (page 27). Here is the definition of our `Line` function in Modelica:

```
function Line "Compute coordinates along a line"
  input Real x      "Independent variable";
  input Real p0[2]  "Coordinates for one point on the line";
  input Real p1[2]  "Coordinates for another point on the line";
  output Real y     "Value of y at the specified x";
algorithm
  y := (p1[2]-p0[2])/(p1[1]-p0[1])*(x-p0[1])+p0[2];
end Line;
```

All the arguments to the function are prefixed with the `input` qualifier. The result of the function has the `output` qualifier. The body of the function is an `algorithm` section. The value for the return value (`y` in this case) is computed by the `algorithm` section.

So in this case, the `output` value, `y`, is computed in terms of the `input` values `x`, `p0` and `p1`. Note that there is no `return` statement in this function. Whatever the value of the `output` variable is at the conclusion of the `algorithm` section is automatically the value returned.

A couple of things to note that were discussed in previous chapters. First, note the descriptive strings on both the function itself and the arguments. These are very useful in documenting the purpose of the function and its arguments. Also note how the points use arrays to represent a two-dimensional vector and how those arrays are indexed in this example.

One troubling aspect of the `Line` model is the length and complexity of the expression used to compute `y`. It would be nice if we could break that expression down.

Intermediate Variables

In order to simplify the expression for `y`, we need to introduce some intermediate variables. We can already see that `x`, `p0` and `p1` are variables that we can use from within the function. We'd like to introduce additional variables, but they shouldn't be arguments. Instead, their values should be computed "internally" to the function. To achieve this, we create a collection of variables that are `protected`. Such variables are assumed to be computed internally by the function. Here is an example that uses `protected` to declare and compute two internal variables:

```
function LineWithProtected "The Line function with protected variables"
  input Real x      "Independent variable";
  input Real p0[2]  "Coordinates for one point on the line";
  input Real p1[2]  "Coordinates for another point on the line";
  output Real y     "Value of y at the specified x";
protected
  Real x0 = p0[1], x1 = p1[1];
  Real y0 = p0[2], y1 = p1[2];
  Real m = (y1-y0)/(x1-x0)  "Slope";
  Real b = (y0-m*x0)        "Offset";
algorithm
  y := m*x+b;
end LineWithProtected;
```

This model introduces two new variables. One variable, `m`, represents the slope of the line and the other, `b`, represents the return value for the condition when `x=0`. Having computed these two intermediate variables, the expression to evaluate `y` becomes the more easily recognized form `y := m*x+b`.

Computing a Polynomial

Mathematical Definition

Of course, our goal for this section is to create a function that can compute arbitrary polynomials. So now that we've seen a few basic functions, let us proceed with our ultimate goal. We will formulate a function that is invoked as follows:

$$p(x, \vec{c})$$

where x is again the independent variable and \vec{c} is a vector of coefficients such that our polynomial is evaluated as:

$$p(x, \vec{c}) = \sum_{i=1}^N c_i x^{N-i}$$

where N is the number of coefficients passed to the function. There are two important things to note at this point. First, **the first element in \vec{c} corresponds to the highest order term in the polynomial**. Second, we are using a notation that assumes that the elements in \vec{c} are numbered **starting from 1** to make the transition to Modelica code (where arrays are indexed starting from 1) easier.

Note that the definition for p above is easy to read and understand. But when working with floating point numbers with finite precision, it is more efficient and more accurate to use a recursive approach for evaluating the polynomial. For a 4th order polynomial, the evaluation would be:

$$p(x, \vec{c}) = ((c_1 x + c_2)x + c_3)x + c_4$$

This is more efficient because it relies on simple multiplication and addition operations and avoids performing exponentiation operations, which are more expensive. It is more accurate because exponentiation can easily trigger round-off or truncation errors in finite precision floating point representations.

Modelica Definition

Now that we've defined precisely what computations we want the function to perform, we are just left with the task of defining the function in Modelica. In this case, our polynomial evaluation function can be represented in Modelica as:

```
function Polynomial "Create a generic polynomial from coefficients"
    input Real x      "Independent variable";
    input Real c[:]   "Polynomial coefficients";
    output Real y     "Computed polynomial value";
protected
    Integer n = size(c,1);
algorithm
    y := c[1];
    for i in 2:n loop
        y := y*x + c[i];
    end for;
end Polynomial;
```

Again, all the arguments to the function have the `input` qualifier and the return value has the `output` qualifier. As with the previous example, we've defined an intermediate variable, `n`, as a convenient way to refer to the length of the coefficient vector. We also see how a `for` loop can be used to represent the recursive evaluation of our polynomial for any arbitrary order.

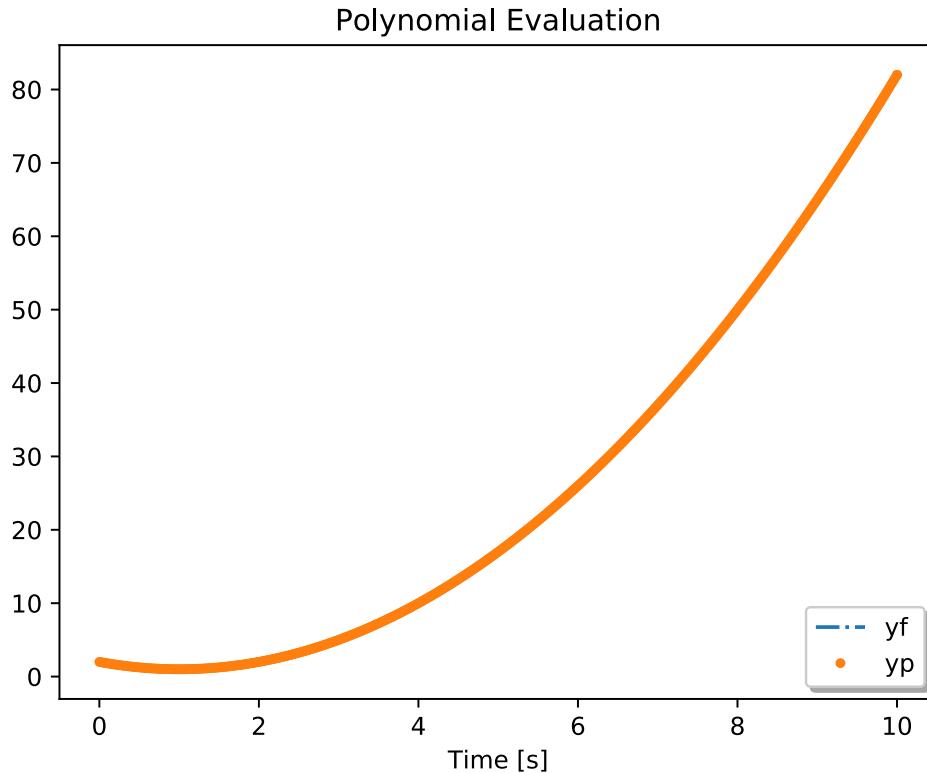
To verify that this function is working properly, let's use it in a model. Consider the following Modelica model:

```

model EvaluationTest1 "Model that evaluates a polynomial"
  Real yf;
  Real yp;
equation
  yf = Polynomial(time, {1, -2, 2});
  yp = time^2-2*time+2;
end EvaluationTest1;

```

Remember that the first element in `c` corresponds to the highest order term. If we compare a direct evaluation of the polynomial, `yp`, with one computed by our function, `yf`, we see they are identical:



Differentiation

It is completely plausible that this polynomial evaluation might be used to represent a quantity that was ultimately differentiated by the Modelica compiler. The following examples is admittedly contrived, but it demonstrates how such a polynomial might come to be differentiated in a model:

```

model Differentiation1 "Model that differentiates a function"
  Real yf;
  Real yp;
  Real d_yf;
  Real d_yp;
equation
  yf = Polynomial(time, {1, -2, 2});
  yp = time^2-2*time+2;
  d_yf = der(yf); // How to compute?
  d_yp = der(yp);
end Differentiation1;

```

Here we have the same equations for `yf`, evaluated using `Polynomial`, and `yp`, evaluated directly as a polynomial. But we've added two additional variables, `d_yf` and `d_yp` representing the derivative of `yf` and `yp`, respectively. If we attempt to compile this model the compiler is very likely to throw an error related to the equation for `d_yf`. The reason is that it has no way to compute the derivative of `yf`. This is because, unlike `yp` which is computed with a simple expression, we've hidden the details of how `yf` is computed behind the function `Polynomial`. In general, Modelica tools do not look at the implementations of functions to compute derivatives and, even if they did, determining the derivative of an arbitrary algorithm is not an easy thing to do.

So the next question is how can we deal with this situation? Won't this make it difficult to use our functions within models? Fortunately, Modelica gives us a way to specify how to evaluate the derivative of a function. This is done by adding something called an `annotation` to the function definition.

Annotations

An annotation is a piece of metadata that doesn't describe the behavior of the function directly (*i.e.*, it doesn't affect the value the function returns). Instead, annotations are used by Modelica compilers to give them "hints" about how to deal with certain situations. Annotations are always "optional" information which means tools are not required to use the information when provided. The Modelica specification defines a number of standard annotations so that they are interpreted consistently across Modelica tools.

In this case, what we need is the `derivative` annotation because it will allow us to communicate information to the Modelica compiler on how to evaluate the derivative of our function. To do this, we define a new evaluation function, `PolynomialWithDerivative`, as follows:

```
function PolynomialWithDerivative
  "Create a generic polynomial from coefficients (with derivative information)"
  input Real x      "Independent variable";
  input Real c[:]   "Polynomial coefficients";
  output Real y     "Computed polynomial value";
protected
  Integer n = size(c,1);
algorithm
  y := c[1];
  for i in 2:n loop
    y := y*x + c[i];
  end for;
  annotation(derivative=PolynomialFirstDerivative);
end PolynomialWithDerivative;
```

Note that this function is identical except for the highlighted line. In other words, all we needed to do was add the line:

```
annotation(derivative=PolynomialFirstDerivative);
```

to our function in order to explain to the Modelica compiler how to evaluate the derivative of this function. What it indicates is that the function `PolynomialFirstDerivative` should be used to evaluate the derivative of `PolynomialWithDerivative`.

Before discussing the implementation of the `PolynomialFirstDerivative` function, let's first understand, mathematically, what is required. Recall our original definition of our polynomial interpolation function:

$$p(x, \vec{c}) = \sum_{i=1}^N c_i x^{N-i}$$

Note that `p` takes two arguments. If we wish to differentiate `p` by some arbitrary variable `z`, we can use the chain rule to express the total derivative of `p` with respect to `z` as:

$$\frac{dp(x, \vec{c})}{dz} = \frac{\partial p}{\partial x} \frac{dx}{dz} + \frac{\partial p}{\partial \vec{c}} \frac{d\vec{c}}{dz}$$

We can derive the following relations from our original definition of p . First, for the partial derivative of p with respect to x we get:

$$\frac{\partial p}{\partial x} = p(x, c')$$

where c' is defined as:

$$c'_i = (N - i)c_i$$

Second, for the partial derivative of p with respect to \vec{c} we get:

$$\frac{\partial p}{\partial c_i} = p(x, \vec{d}_i)$$

where the **vector** \vec{d}_i is the i^{th} column of an $N \times N$ identity matrix.

It turns out that for efficiency reasons, it is better for the Modelica compiler to give us $\frac{dx}{dz}$ and $\frac{d\vec{c}}{dz}$ than to provide functions to evaluate $\frac{\partial p}{\partial x}$ and $\frac{\partial p}{\partial c_i}$. So, mathematically speaking, what the Modelica compiler needs is a new function that is invoked with the following arguments:

$$df(x, \vec{c}, \frac{dx}{dz}, \frac{d\vec{c}}{dz})$$

such that:

$$df(x, \vec{c}, \frac{dx}{dz}, \frac{d\vec{c}}{dz}) = \frac{df}{dz}$$

For this reason, the **derivative** annotation should point to a function that takes the same arguments as df . In our case, that function, **PolynomialFirstDerivative** would be defined as follows:

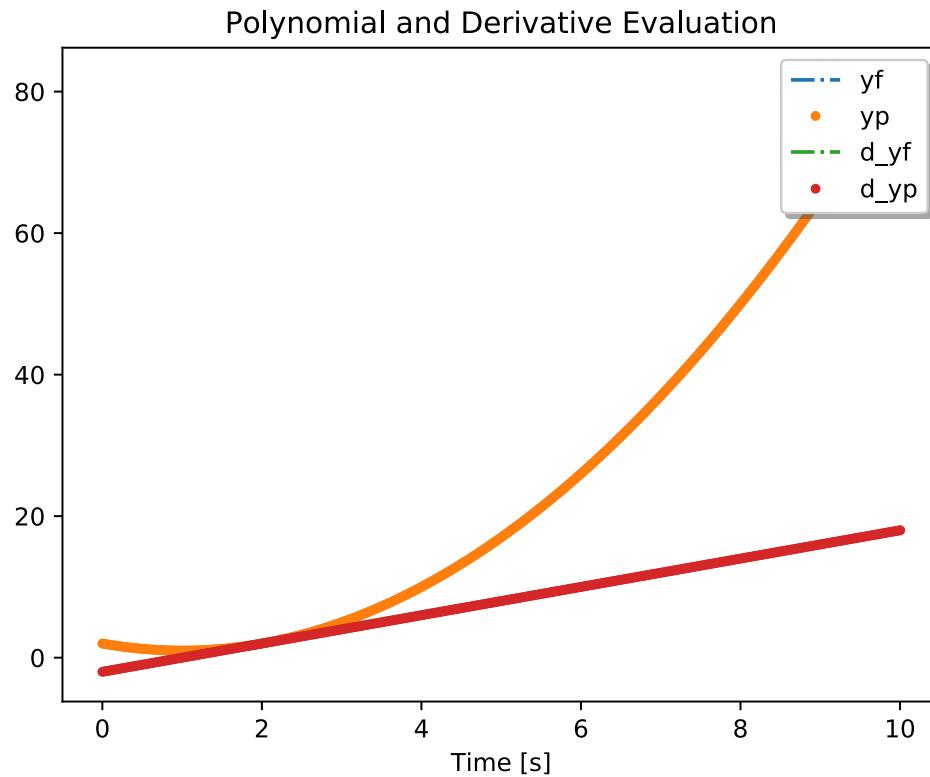
```
function PolynomialFirstDerivative
  "First derivative of the function Polynomial"
  input Real x;
  input Real c[:];
  input Real x_der;
  input Real c_der[size(c,1)];
  output Real y_der;
protected
  Integer n = size(c,1);
  Real c_diff[n-1] = {(n-i)*c[i] for i in 1:n-1};
algorithm
  y_der := PolynomialWithDerivative(x, c_diff)*x_der +
    PolynomialWithDerivative(x, c_der);
end PolynomialFirstDerivative;
```

Note how the arguments of our original function are repeated to create twice as many arguments (as we would expect). The second set of arguments represent the $\frac{dx}{dz}$ and $\frac{d\vec{c}}{dz}$ quantities, respectively. Note that the assumption is that z is a scalar so the types of the input arguments are the same. Exploiting our knowledge about the partial derivatives of a polynomial, the calculation of the derivatives is done by leveraging our polynomial evaluation function.

We can exercise all of these functions using the following model:

```
model Differentiation2 "Model that differentiates a function using derivative annotation"
  Real yf;
  Real yp;
  Real d_yf;
  Real d_yp;
equation
  yf = PolynomialWithDerivative(time, {1, -2, 2});
  yp = time^2-2*time+2;
  d_yf = der(yf);
  d_yp = der(yp);
end Differentiation2;
```

Simulating this model and comparing results, we see agreement between yf and yp as well as d_yf and d_yp :



Interpolation

In this chapter, we will example different ways of implementing a simple one dimension interpolation scheme. We'll start with an approach that is written completely in Modelica and then show an alternative implementation that combines Modelica with C. The advantages and disadvantages of each approach will then be discussed.

Modelica Implementation

Function Definition

For this example, we assume that we interpolate data in the following form:

Independent Variable, x	Dependent Variable, y
x_1	y_1
x_2	y_2
x_3	y_3
...	...
x_n	y_n

where we assume that $x_i < x_{i+1}$.

Given this data and the value for the independent variable x that we are interested in, our function should return an interpolated value for y . Such a function could be implemented in Modelica as follows:

```

function InterpolateVector "Interpolate a function defined by a vector"
    input Real x           "Independent variable";
    input Real ybar[:,2] "Interpolation data";
    output Real y         "Dependent variable";
protected
    Integer i;
    Integer n = size(ybar,1) "Number of interpolation points";
    Real p;
algorithm
    assert(x>=ybar[1,1], "Independent variable must be greater than or equal to "+String(ybar[1,
->1]));
    assert(x<=ybar[n,1], "Independent variable must be less than or equal to "+String(ybar[n,
->1]));
    i := 1;
    while x>=ybar[i+1,1] loop
        i := i + 1;
    end while;
    p := (x-ybar[i,1])/(ybar[i+1,1]-ybar[i,1]);
    y := p*ybar[i+1,2]+(1-p)*ybar[i,2];
end InterpolateVector;

```

Let's go through this function piece by piece to understand what is going on. We'll start with the argument declarations:

```

input Real x           "Independent variable";
input Real ybar[:,2] "Interpolation data";
output Real y         "Dependent variable";

```

The `input` argument `x` represents the value of the independent variable we wish to use for interpolating our function, the `input` argument `ybar` represents the interpolation data and the `output` argument `y` represents the interpolated value. The next part of the function contains:

```

protected
    Integer i;
    Integer n = size(ybar,1) "Number of interpolation points";
    Real p;

```

The part of our function includes the declaration of various `protected` variables. As we saw in our *[Polynomial Evaluation](#)* (page 116) example, these are effectively intermediate variables used internally by the function. In this case, `i` is going to be used as an index variable, `n` is the number of data points in our interpolation data and `p` represents a weight used in our interpolation scheme.

With all the variable declarations out of the way, we can now implement the `algorithm` section of our function:

```

algorithm
    assert(x>=ybar[1,1], "Independent variable must be greater than or equal to "+String(ybar[1,
->1]));
    assert(x<=ybar[n,1], "Independent variable must be less than or equal to "+String(ybar[n,
->1]));
    i := 1;
    while x>=ybar[i+1,1] loop
        i := i + 1;
    end while;
    p := (x-ybar[i,1])/(ybar[i+1,1]-ybar[i,1]);
    y := p*ybar[i+1,2]+(1-p)*ybar[i,2];

```

The first two statements are `assert` statements that verify that the value of `x` is within the interval $[x_1, x_n]$. If not, an error message will be generated explaining why the interpolation failed.

The rest of the function searches for the value of `i` such that $x_i \leq x < x_{i+1}$. Once that value of `i` has

been identified, the interpolated value is computed as simply:

$$y = p \bar{y}_{i+1,2} + (1 - p) \bar{y}_{i,2}$$

where

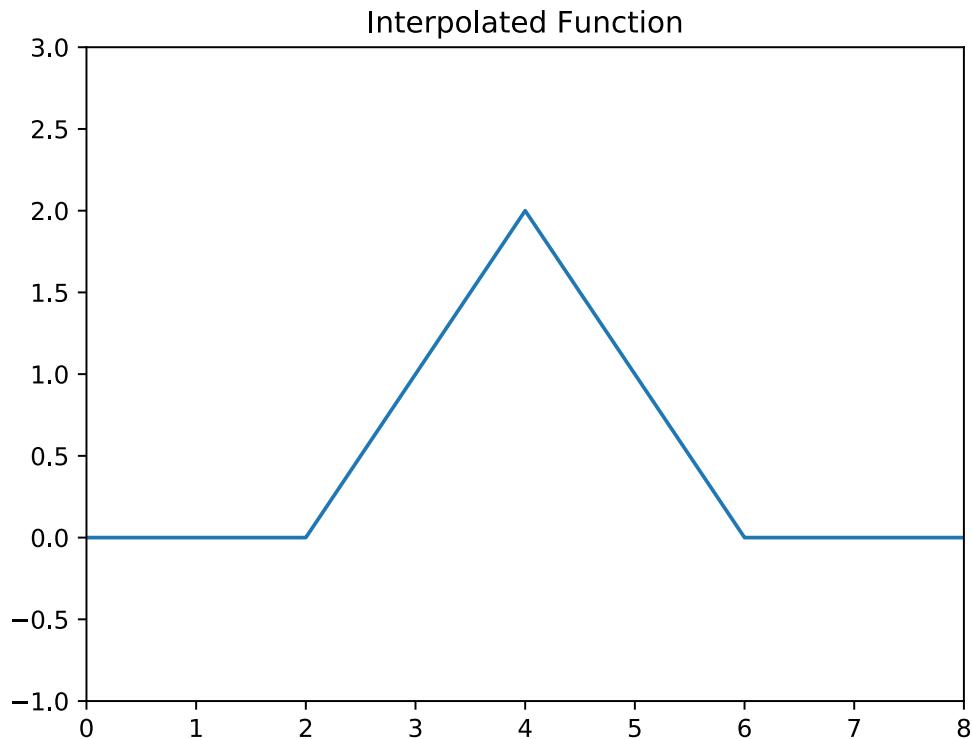
$$p = \frac{x - \bar{y}_{i,1}}{\bar{y}_{i+1,1} - \bar{y}_{i,1}}$$

Test Case

Now, let's test this function by using it from within a model. As a simple test case, let's integrate the value returned by the interpolation function. We'll use the following data as the basis for our function:

x	y
0	0
2	0
4	2
6	0
8	0

If we plot this data, we see that our interpolated function looks like this:



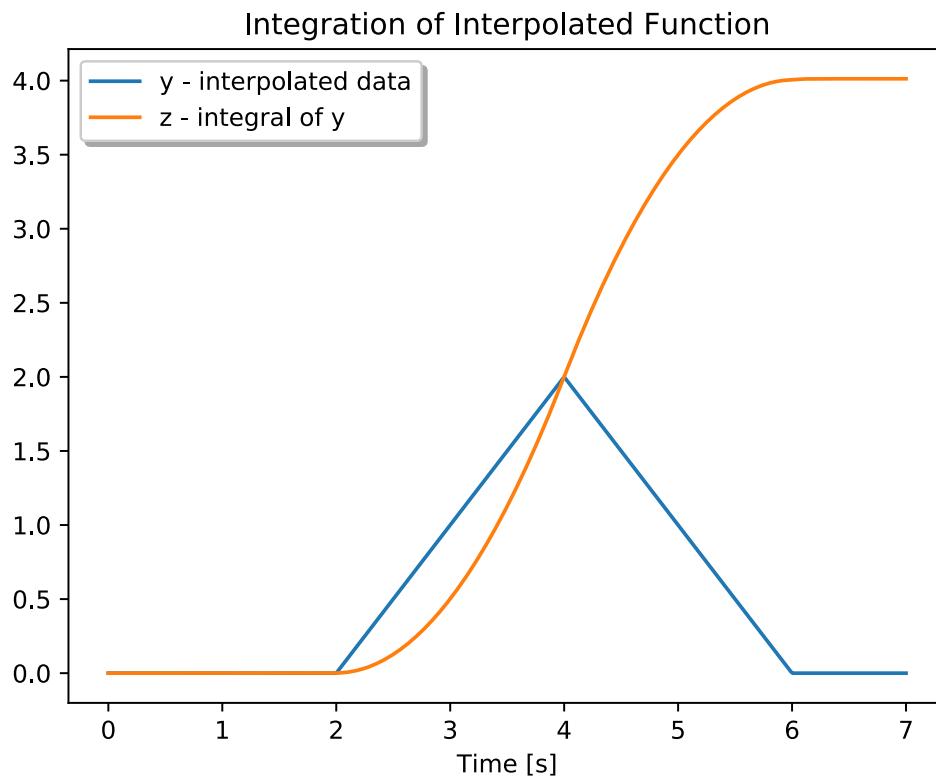
In the following model, the independent variable `x` is set equal to `time`. The sample data is then used to interpolate a value for the variable `y`. The value of `y` is then integrated to compute `z`.

```

model IntegrateInterpolatedVector "Exercises the InterpolateVector"
  Real x;
  Real y;
  Real z;
equation
  x = time;
  y = InterpolateVector(x, [0.0, 0.0; 2.0, 0.0; 4.0, 2.0; 6.0, 0.0; 8.0, 0.0]);
  der(z) = y;
annotation (experiment(StopTime=6));
end IntegrateInterpolatedVector;

```

We can see the simulated results from this model in the following plot:



There are a couple of drawbacks to this approach. The first is that the data needs to be passed around anywhere the function is used. Also, for higher dimensional interpolation schemes, the data required can be both complex (for irregular grids) and large. So it is not necessarily convenient to store the data in the Modelica source code. For example, it might be preferable to store the data in an external file. However, to populate the interpolation data from a source other than the Modelica source code, we will need to use an `ExternalObject`.

Using an ExternalObject

The `ExternalObject` type is a special type used to refer to information that is not (necessarily) represented in the Modelica source code. The main use of the `ExternalObject` type is to represent data or state that is maintained outside the Modelica source code. This might be interpolation data, as we will see in a moment, or it might represent some other software system that maintains its own state.

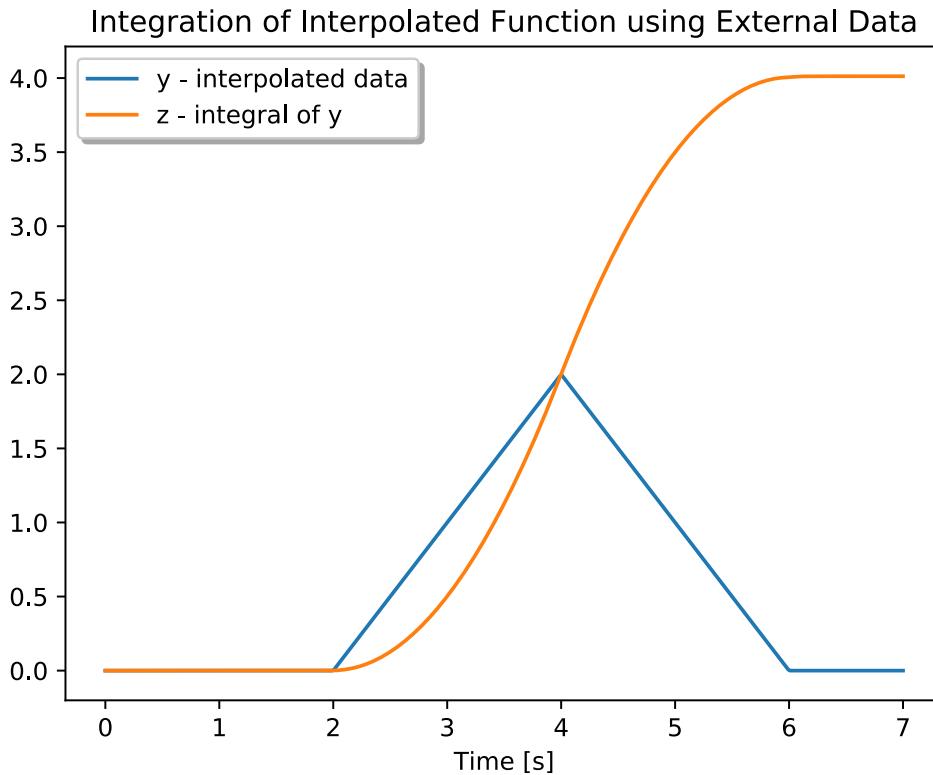
Test Case

For this example, we will flip things around and start with the test case. This will provide some useful context about how an `ExternalObject` is used. The Modelica source code for our test case is:

```
model IntegrateInterpolatedExternalVector
  "Exercises the InterpolateExternalVector"
  parameter VectorTable vector = VectorTable(ybar=[0.0, 0.0;
                                                 2.0, 0.0;
                                                 4.0, 2.0;
                                                 6.0, 0.0;
                                                 8.0, 0.0]);
  Real x;
  Real y;
  Real z;
equation
  x = time;
  y = InterpolateExternalVector(x, vector);
  der(z) = y;
  annotation (experiment(StopTime=6));
end IntegrateInterpolatedExternalVector;
```

Here the main difference between this and our previous test case is the fact that we don't pass our data directly into the interpolation function. Instead, we create a special variable `vector` whose type is `VectorTable`. We'll discuss exactly what a `VectorTable` is in a moment. But for now think of it as something that represents (only) our interpolation data. Other than the creation of the `vector` object, the rest of the model is virtually identical to the previous case except that we use the `InterpolateExternalVector` function to perform our interpolation and we pass the `vector` variable into that function in place of our raw interpolation data.

Simulating this model, we see that the results are exactly what we would expect when compared to our previous test case:



Defining an ExternalObject

To see how this most recent test case is implemented, we'll first look at how the `VectorTable` type is implemented. As mentioned previously, the `VectorTable` is an `ExternalObject` type. This is a special type in Modelica that is used to represent what is often called an “opaque” pointer. This means that the `ExternalObject` represents some data that is not directly accessible (from Modelica).

In our case, we implement our `VectorTable` type as:

```
type VectorTable "A vector table implemented as an ExternalObject"
  extends ExternalObject;
  function constructor
    input Real ybar[:,2];
    output VectorTable table;
    external "C" table=createVectorTable(ybar, size(ybar,1))
    annotation(IncludeDirectory="modelica://ModelicaByExample.Functions.Interpolation/source",
               Include="#include \"VectorTable.c\"");
  end constructor;

  function destructor "Release storage"
    input VectorTable table;
    external "C" destroyVectorTable(table)
    annotation(IncludeDirectory="modelica://ModelicaByExample.Functions.Interpolation/source",
               Include="#include \"VectorTable.c\"");
  end destructor;
end VectorTable;
```

Note that the `VectorTable` inherits from the `ExternalObject` type. An `ExternalObject` can have

two special functions implemented inside its definition, the `constructor` function and the `destructor` function. Both of these functions are seen here.

Constructor

The `constructor` function is invoked when an instance of a `VectorTable` is created (*e.g.*, the declaration of the `vector` variable in our test case). This `constructor` function is used to initialize our opaque pointer. Whatever data is required as part of that initialization process should be passed as argument to the `constructor` function. That same data should be present during instantiation (*e.g.*, the `data` argument in our declaration of the `vector` variable).

The definition of the `constructor` function is unusual because, unlike our previous examples, it does not include an `algorithm` section. The `algorithm` section is normally used to compute the return value of the function. Instead, the `constructor` function has an `external` clause. This indicates that the function is implemented in some other language besides Modelica. In this case, that other language is C (as indicated by the "C" following the `external` keyword). This tells use that the `table` variable (which is the `output` of this function and represents the opaque pointer) is returned by a C function named `createVectorTable` which is passed the contents and size of the `ybar` variable.

Following the call to `createVectorTable` is an `annotation`. This annotation tells the Modelica compiler where to find the source code for this external C function.

The essential point here is that from the point of view of the Modelica compiler, a `VectorTable` is just an opaque pointer returned by `createVectorTable`. It is not possible to access the data behind this pointer from Modelica. But this pointer can be passed to other functions, as we shall see in a minute, that are also implemented in C and which do know how to access the data represented by the `VectorTable`.

Destructor

The `destructor` function is invoked whenever the `ExternalObject` is no longer needed. This allows the Modelica runtime to clean up any memory consumed by the `ExternalObject`. An `ExternalObject` instantiated in a model will generally persist until the end of the simulation. But an `ExternalObject` declared as a `protected` variable in a function, for example, may be created and destroyed in the course of a single expression evaluation. For that reason, it is important to make sure that any memory allocated by the `ExternalObject` is released.

In general, the `destructor` function is also implemented as an external function. In this case, calling the `destructor` function from Modelica invokes the C function `destroyVectorTable` which is passed a `VectorTable` instance as an argument. Any memory associated with that `VectorTable` instance should be freed by the call to `destructor`. Again, we see the same types of annotations used to inform the Modelica compiler where to find the source code for the `destroyVectorTable` function.

External C Code

These external C functions are implemented as follows:

```
#ifndef _VECTOR_TABLE_C_
#define _VECTOR_TABLE_C_

#include <stdlib.h>
#include "ModelicaUtilities.h"

/*
   Here we define the structure associated
   with our ExternalObject type 'VectorTable'
*/
typedef struct {
    double *x; /* Independent variable values */
}
```

```

    double *y; /* Dependent variable values */
    size_t npoints; /* Number of points in this data */
    size_t lastIndex; /* Cached value of last index */
} VectorTable;

void *
createVectorTable(double *data, size_t np) {
    VectorTable *table = (VectorTable*) malloc(sizeof(VectorTable));
    if (table) {
        /* Allocate memory for data */
        table->x = (double*) malloc(sizeof(double)*np);
        if (table->x) {
            table->y = (double*) malloc(sizeof(double)*np);
            if (table->y) {
                /* Copy data into our local array */
                size_t i;
                for(i=0;i<np;i++) {
                    table->x[i] = data[2*i];
                    table->y[i] = data[2*i+1];
                }
                /* Initialize the rest of the table object */
                table->npoints = np;
                table->lastIndex = 0;
            }
            else {
                free(table->x);
                free(table);
                table = NULL;
                ModelicaError("Memory allocation error\n");
            }
        }
        else {
            free(table);
            table = NULL;
            ModelicaError("Memory allocation error\n");
        }
    }
    else {
        ModelicaError("Memory allocation error\n");
    }
    return table;
}

void
destroyVectorTable(void *object) {
    VectorTable *table = (VectorTable *)object;
    if (table==NULL) return;
    free(table->x);
    free(table->y);
    free(table);
}

double
interpolateVectorTable(void *object, double x) {
    VectorTable *table = (VectorTable *)object;
    size_t i = table->lastIndex;
    double p;

    ModelicaFormatMessage("Request to compute value of y at %g\n", x);
    if (x<table->x[0])
        ModelicaFormatError("Requested value of x=%g is below the lower bound of %g\n",
                            x, table->x[0]);
    if (x>table->x[table->npoints-1])

```

```
ModelicaFormatError("Requested value of x=%g is above the upper bound of %g\n",
    x, table->x[table->npoints-1]);

while(i<table->npoints-1&&x>table->x[i+1]) i++;
while(i>0&&x<table->x[i]) i--;

p = (x-table->x[i])/(table->x[i+1]-table->x[i]);
table->lastIndex = i;
return p*table->y[i+1]+(1-p)*table->y[i];
}

#endif
```

This is not a book on the C programming language so an exhaustive review of this code and exactly how it functions is beyond the scope of the book. But we can summarize the contents of this file as follows.

First, the `struct` called `VectorTable` is the data associated with the `VectorTable` type in Modelica. This includes not just the interpolation data (in the form of the `x` and `y` members), but also the number of data points, `npoints`, and a cached value for the last used index, `lastIndex`.

Next, we see the `createVectorTable` function which allocates an instance of the `VectorTable` structure and initializes all the data inside it. That instance is then returned to the Modelica runtime. Following the definition of `createVectorTable` is the definition of `destroyVectorTable` which effectively undoes what was done by `createVectorTable`.

Finally, we see the function `interpolateVectorTable`. This is a C function that is passed an instance of the `VectorTable` structure and a value for the independent variable and returns the interpolated value for the dependent variable. This function performs almost exactly the same function as the `InterpolateVector` function presented earlier. The Modelica runtime provides functions like `ModelicaFormatError` so that external C code can report errors. In the case of `interpolateVectorTable`, these functions are used to implement the assertions we saw previously in `InterpolateVector`. The lookup of `i` is basically the same except that instead of starting from 1 each time, it starts from the value of `i` found in the last call to `interpolateVectorTable`.

Interpolation

We've seen how `interpolateVectorTable` is defined, but so far we haven't seen where it is used. We mentioned that performs very much the same role as `InterpolateVector`, but using a `VectorTable` object to represent the interpolation data. To invoke `interpolateVectorTable` from Modelica, we simple need to define a Modelica function as follows:

```
function InterpolateExternalVector
  "Interpolate a function defined by a vector using an ExternalObject"
  input Real x;
  input VectorTable table;
  output Real y;
  external "C" y = interpolateVectorTable(table, x)
  annotation(IncludeDirectory="modelica://ModelicaByExample.Functions.Interpolation/source",
             Include="#include \"VectorTable.c\"");
end InterpolateExternalVector;
```

We mentioned previously that `VectorTable` is opaque and that Modelica code cannot access the data contained in the `VectorTable`. The Modelica function `InterpolateExternalVector` invokes its C counterpart `interpolateVectorTable` which **can** access the interpolation data and, therefore, perform the interpolation.

Discussion

As was discussed previously, the initial interpolation approach required us to pass around large amounts of unwieldy data. By implementing the `VectorTable`, we were able to represent that data by a single variable.

An important thing to note about the `ExternalObject` approach, which isn't adequately explored in our example, is that the initialization data can be completely external to the Modelica source code. For simplicity, the example code shown in this section initializes the `VectorTable` using an array of data. **But it could just as easily have passed a file name** to the initialization code. That file could then have been read by the `createVectorTable` function and the contents of the `VectorTable` structure could have been initialized using the data from that file. In many cases, this approach not only makes managing the data easier, but leveraging C allows more complex (new or existing) algorithms to be used.

The [next section](#) (page 131) includes another example of how external C code can be called from Modelica.

Software-in-the-Loop Controller

In the previous [Interpolation](#) (page 122) example, we saw external C functions could be used to manage and interpolate data. In this section, we will explore how to integrate C code for an embedded controller into a Modelica model.

When building mathematical models of physical systems in Modelica, it is sometimes useful to integrate (external) control strategies into these models. In many cases, these strategies exist as C code generated for use with an embedded controller. This example revisits the topic of [Hysteresis](#) (page 72), but with an interesting twist. Instead of implementing the hysteresis behavior in Modelica using discrete states, we will implement it in an external C function. Although this example is extremely simple, it demonstrates all the essential steps necessary to integrate external control strategies.

Physical Model

Let's start by looking at our "physical model". In this case, it is essentially just a reimplementation of the model we created previously during our discussion on [Hysteresis](#) (page 72). Our revised implementation looks like this:

```
model HysteresisEmbeddedControl "A control strategy that uses embedded C code"
  type HeatCapacitance=Real(unit="J/K");
  type Temperature=Real(unit="K");
  type Heat=Real(unit="W");
  type Mass=Real(unit="kg");
  type HeatTransferCoefficient=Real(unit="W/K");
  parameter HeatCapacitance C=1.0;
  parameter HeatTransferCoefficient h=2.0;
  parameter Heat Qcapacity=25.0;
  parameter Temperature Tamb=285;
  parameter Temperature Tbar=295;
  Temperature T;
  Heat Q;
initial equation
  T = Tbar+5;
equation
  when sample(0, 0.01) then
    Q = computeHeat(T, Tbar, Qcapacity);
  end when;
  C*der(T) = Q-h*(T-Tamb);
end HysteresisEmbeddedControl;
```

Let's take a closer look at the `equation` section:

```

equation
when sample(0, 0.01) then
  Q = computeHeat(T, Tbar, Qcapacity);
end when;
C*der(T) = Q-h*(T-Tamb);
end HysteresisEmbeddedControl;

```

The function `computeHeat` is called every 10 milliseconds to determine the amount of heat to be used. As we will see in a moment, the controller implements a “bang-bang” control strategy. That means it flips between zero heat generation and full heat generation. As we saw in our previous discussion on [Hysteresis](#) (page 72), this kind of approach can lead to “chattering”. For that reason, we put the calculation of `Q` inside a `when` statement that is executed every 10 milliseconds. This 10 millisecond interval is essentially implementing the behavior of what is normally called a “scheduler” which decides when different control strategies are executed.

Embedded Control Strategy

The Modelica function `computeHeat` used to determine how much heat should be applied to the system at any given time is implemented as:

```

impure function computeHeat "Modelica wrapper for an embedded C controller"
  input Real T;
  input Real Tbar;
  input Real Q;
  output Real heat;
  external "C" annotation (Include="#include \"ComputeHeat.c\"",
    IncludeDirectory="modelica://ModelicaByExample.Functions.ImpureFunctions/source");
end computeHeat;

```

Note the presence of the `external` keyword. This time, however, we don’t see the name of the external C function like we did in the previous examples. This means that the external C function has exactly the same name and arguments as its Modelica counterpart. Looking at the source code for the C function, we see that is the case:

```

#ifndef _COMPUTE_HEAT_C_
#define _COMPUTE_HEAT_C_

#define UNINITIALIZED -1
#define ON 1
#define OFF 0

double
computeHeat(double T, double Tbar, double Q) {
  static int state = UNINITIALIZED;
  if (state==UNINITIALIZED) {
    if (T>Tbar) state = OFF;
    else state = ON;
  }
  if (state==OFF && T<Tbar-2) state = ON;
  if (state==ON && T>Tbar+2) state = OFF;

  if (state==ON) return Q;
  else return 0;
}

#endif

```

In other words, we can save ourselves the trouble of specifying how the input and output arguments of our Modelica function map to those of the underlying C function by defining them in such a way that no mapping is truly necessary.

Note the presence of the `static` variable `state` in the C implementation of `computeHeat`. The use of the `static` keyword here indicates that the value of the variable `state` is preserved from one invocation of `computeHeat` to another. This kind of variable is quite common in embedded control strategies because they need to preserve information from one invocation of the scheduler to the next (*e.g.*, to implement hysteresis control, as we see here).

The presence of this `static` variable is a significant problem because it means that the function `computeHeat` can return **different results for the same input arguments**. Mathematically speaking, this is not a true mathematical function since a mathematical function can only depend on its input arguments. In computer science, we say such a function is “impure”. This means that each invocation of the function changes some internal memory or variable which affects that value returned by the function.

Given that such impurity is implemented in embedded control strategies **by design**, we need to be careful when using them in a mathematically oriented environment like Modelica. This is because the Modelica compiler assumes, by default, that all functions are pure and side effect free and the presence of impurity or side effects can result in very inefficient simulations, at best, or completely erroneous results, at worst.

These problems occur because the underlying solvers must compute many “candidate” solutions before they compute the “real” solution. If generating candidate solutions requires the solver to invoke functions with side effects, the solver will be unable to anticipate the effects triggered by the changes to variables it is not aware of.

It is for precisely this reason that the `impure` qualifier is applied to the definition of `computeHeat`:

```
impure function computeHeat "Modelica wrapper for an embedded C controller"
  input Real T;
```

This informs the Modelica compiler that this function **has side effects or returns a result that depends on something other than its inputs** and that it **should not** be invoked when generating candidate solutions. At first, this seems like it would completely prohibit calling the function, but that isn’t the case. Recall our integration of the control strategy:

```
equation
  when sample(0, 0.01) then
    Q = computeHeat(T, Tbar, Qcapacity);
  end when;
  C*der(T) = Q-h*(T-Tamb);
end HysteresisEmbeddedControl;
```

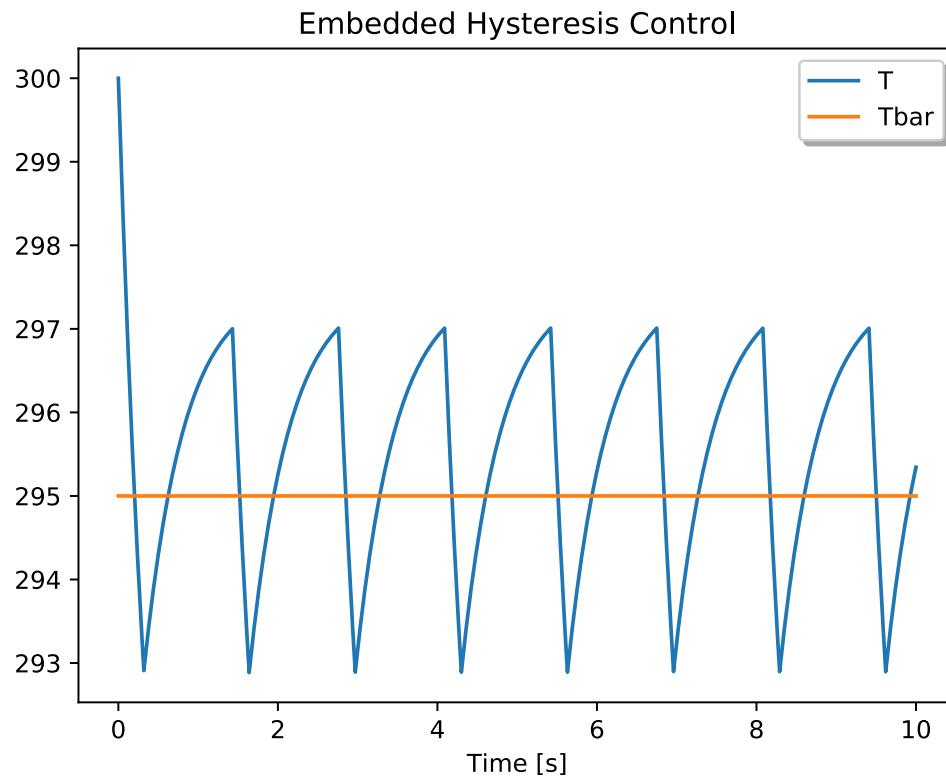
In particular, note that `computeHeat` is invoked only within the `when` statement and not as part of a “continuous” equation. As a result, we can be certain that `computeHeat` will only be invoked in response to an event but not when evaluating candidate solutions for the continuous variables.

Results

In the C function `computeHeat`, we see that these two statements implement a +/- 2 degree deadband around the setpoint:

```
if (state==OFF && T<Tbar-2) state = ON;
if (state==ON && T>Tbar+2) state = OFF;
```

It is this functionality that prevents chattering and which can be clearly observed in the simulated results for our example:



Non-Linearities

Our next example involving functions demonstrates how to address issues that come up when solving non-linear systems of equations that involve functions.

We'll start with a simple model that includes this simple mathematical relationship:

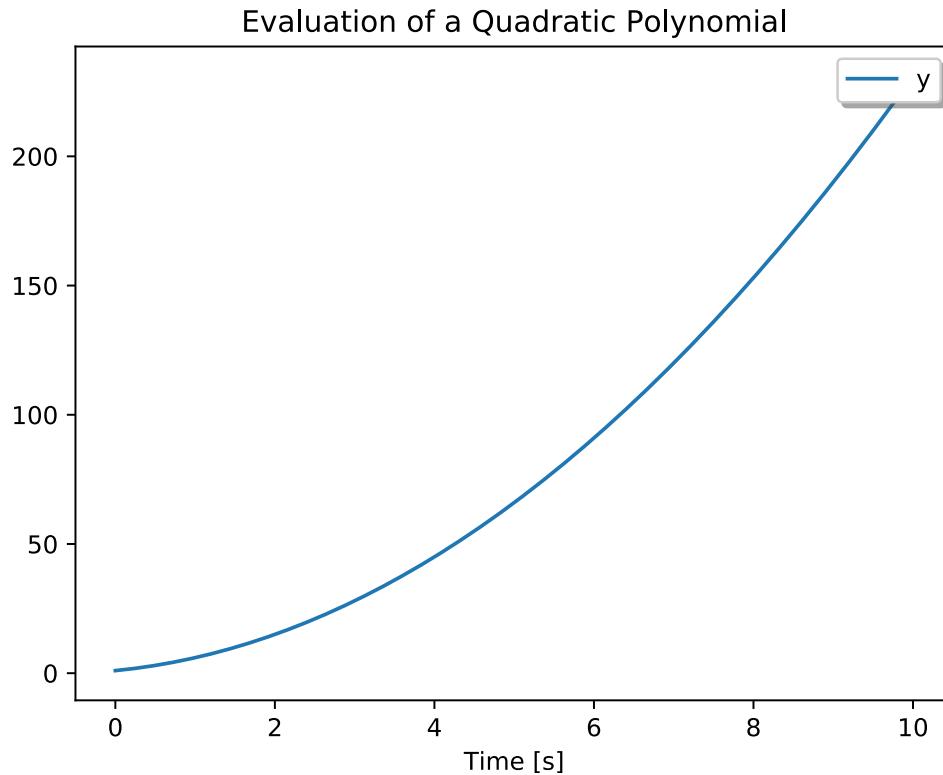
$$y = 2t^2 + 3t + 1$$

where t is time. This model can be implemented in Modelica as follows:

```
model ExplicitEvaluation
  "Model that evaluates the quadratic function explicitly"
  Real y;
equation
  y = Quadratic(2.0, 3.0, 1.0, time);
end ExplicitEvaluation;
```

where the `Quadratic` function, which we will discuss shortly, evaluates a quadratic polynomial.

Simulating this model gives the following solution for y :



So far, all of this appears quite reasonable and, based on our previous discussion on *Polynomial Evaluation* (page 116), the implementation of `Quadratic` isn't likely to hold too many surprises. However, let's make things a little more complicated. Consider the following model:

```
model ImplicitEvaluation
  "Model that requires the inverse of the quadratic function"
  parameter Real y_guess=2;
  Real y(start=y_guess);
equation
  time+1 = Quadratic(2.0, 3.0, 1.0, y);
end ImplicitEvaluation;
```

This model amounts to solving the following equation:

$$t + 5 = 2y^2 + 3y + 1$$

The important difference here is that the left hand side is known and we must compute the value of y that satisfies this equation. In other words, instead of evaluating a quadratic polynomial, like we were doing in the previous example, now we have to solve a quadratic equation.

A model that requires solving a non-linear system of equations is not remarkable by itself. Modelica compilers are certainly more than capable of recognizing and solving non-linear systems of equations (although these methods usually depend on having a reasonable initial guess in order to converge).

However, it turns out that in this case, **the Modelica compiler doesn't actually need to solve a non-linear system**. Although we couldn't know this until we saw how the `Quadratic` function is implemented:

```
function Quadratic "A quadratic function"
  input Real a "2nd order coefficient";
```

```

input Real b "1st order coefficient";
input Real c "constant term";
input Real x "independent variable";
output Real y "dependent variable";
algorithm
  y := a*x*x + b*x + c;
  annotation(inverse(x = InverseQuadratic(a,b,c,y)));
end Quadratic;

```

In particular, note the line specifying the `inverse` annotation. With this function definition, we not only tell the Modelica compiler how to evaluate the `Quadratic` function, but, through the `inverse` annotation, we are also indicating that the `InverseQuadratic` function should be used to compute `x` in terms of `y`.

The `InverseQuadratic` function is defined as follows:

```

function InverseQuadratic
  "An inverse of the quadratic function returning the positive root"
  input Real a;
  input Real b;
  input Real c;
  input Real y;
  output Real x;
algorithm
  x := sqrt(b*b - 4*a*(c - y))/(2*a);
end InverseQuadratic;

```

Note: Note that the `InverseQuadratic` function computes only the positive root in the quadratic equation. This can be both a good thing and a bad thing. By computing only a single root, we avoid the issue of having multiple solutions when we invert the quadratic relationship. However, if the negative root happens to be the one you want, this can be a problem.

In the case of our `ImplicitEvaluation` model, the Modelica compiler can then substitute this inverse function into the equations. So, where we initially had, ignoring the coefficient arguments for the moment, the following equation to solve:

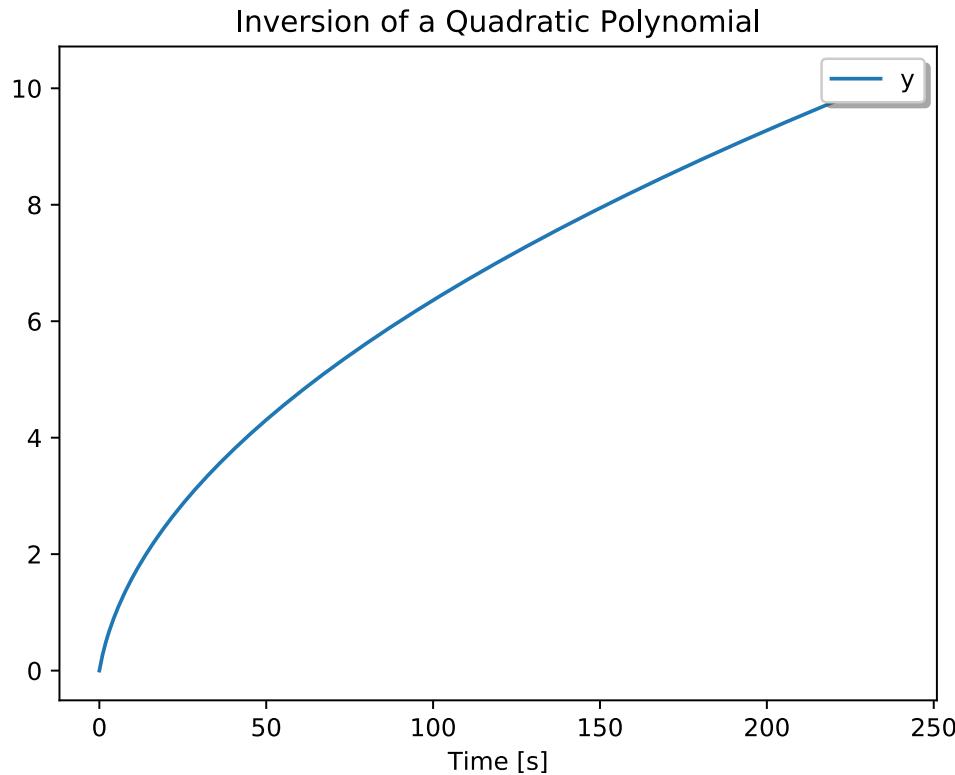
$$t + 5 = f(y)$$

which must be solved as an implicit equation for `y`, we can now solve the following **explicit** equation:

$$y = f^{-1}(t + 5)$$

by using the `InverseQuadratic` function as the inverse function.

Simulating the `ImplicitEvaluation` model we get the following solution for `y`:



Looking at this figure, we can see that, in fact, we got the correct result, but, in general, without the need to solve the non-linear system that would otherwise result from our `ImplicitEvaluation` model.

1.4.2 Review

Function Definitions

As we've already seen, Modelica includes many useful functions for describing mathematical behavior. But, inevitably, it is necessary to create new functions for specific purposes. Defining such functions is similar, syntactically, to a [Model Definition](#) (page 27).

Basic Syntax

A basic Modelica function includes one or more arguments, a return value and an `algorithm` section to compute the return value in terms of the arguments. The arguments to the function are preceded by the `input` qualifier and the return value is preceded by the `output` qualifier. For example, consider the following simple function that squares its input argument:

```
function Square
  input Real x;
  output Real y;
algorithm
  y := x*x;
end Square;
```

Here the input argument `x` has the type `Real`. The output variable `y` also has the `Real` type. Arguments and return values can be scalars or arrays (or even records, although we won't introduce records until

later (page 183)).

Intermediate Variables

For complex calculations, it is sometimes useful to define variables to hold intermediate results. Such variables must be clearly distinguished from arguments and return values. To declare such intermediate variables, make their declarations **protected**. Making the variables **protected** indicates to the Modelica compiler that these variables are not arguments or return values, but are instead used internally by the function. For example, if we wished to write a function to compute the circumference of a circle, we might utilize an intermediate variable to store the diameter:

```
function Circumference
  input Real radius;
  output Real circumference;
protected
  Real diameter := radius*2;
algorithm
  circumference := 3.14159*diameter;
end Circumference;
```

Here we see how some intermediate result or common sub-expression can be associated with an internal variable.

Default Input Arguments

In some cases, it makes sense to include default values for some input arguments. In these cases, it is possible to include a default value in the declaration of the input variable. Consider the following function to compute the potential energy of a mass in a gravitational field:

```
function PotentialEnergy
  input Real m "mass";
  input Real h "height";
  input Real g=9.81 "gravity";
  output Real pe "potential energy";
algorithm
  pe := m*g*h;
end PotentialEnergy;
```

By providing a default value for *g*, we do not force users of this function to provide a value for *g* each time. Of course, this kind of approach should only be used when there is a reasonable default value for a given argument and it should never be used if you want to force users to provide a value.

These default values have some important effects when *Calling Functions* (page 139) that we shall discuss shortly.

Multiple Return Values

Note that a function can have multiple return values (*i.e.*, multiple declarations with the **output** qualifier). For example, to consider a function that computes both the circumference and area of a circle:

```
function CircleProperties
  input Real radius;
  output Real circumference;
  output Real area;
protected
  Real diameter := radius*2;
algorithm
  circumference := 3.14159*diameter;
```

```
area := 3.14159*radius^2;
end CircleProperties;
```

Our upcoming discussion on *Calling Functions* (page 139) will cover how to address multiple return values.

Calling Functions

So far, we've covered how to define new functions. But it is also worth spending some time discussing the various ways of calling functions. In general, functions are invoked in a way that would be expected by both mathematicians and programmers, *e.g.*,

```
f(z, t);
```

Here we see the typical syntax name of the function name followed by a comma separated list of arguments surrounded by parentheses. But there are several interesting cases to discuss.

The syntax above is “positional”. That means that values in the function call are assigned to arguments based on the order. But since Modelica function arguments have names, it is also possible to call functions using named arguments. Consider the following function for computing the volume of a cube:

```
function CylinderVolume
  input Real radius;
  input Real length;
  output Real volume;
algorithm
  volume = 3.14159*radius^2*length;
end CylinderVolume;
```

When calling this function, it is important not to confuse the radius and the volume. To avoid any possible confusion regarding their order, it is possible to call the function used named arguments. In that case, the function call would look something like:

```
CylinderVolume(radius=0.5, length=12.0);
```

Named arguments are particularly useful in conjunction with default argument values. Recall the `PotentialEnergy` function introduced earlier. It can be invoked in several ways:

```
PotentialEnergy(1.0, 0.5, 9.79)      // m=1.0, h=0.5, g=9.79
PotentialEnergy(m=1.0, h=0.5, g=9.79) // m=1.0, h=0.5, g=9.79
PotentialEnergy(h=0.5, m=1.0, g=9.79) // m=1.0, h=0.5, g=9.79
PotentialEnergy(h=0.5, m=1.0)          // m=1.0, h=0.5, g=9.81
PotentialEnergy(0.5, 1.0)              // m=1.0, h=0.5, g=9.81
```

The reason named arguments are so important for arguments with default values is if a function has many arguments with default arguments, you can selectively override values for those arguments by referring to them by name.

Finally, we previously pointed out the fact that it is possible for a function to have multiple return values. But the question remains, how do we address multiple return values? To see how this is done in practice, let us revisit the `CircleProperties` function we defined earlier in this section. The following statement shows how we can reference both return values:

```
(c, a) := CircleProperties(radius);
```

In other words, the left hand side is a comma separated list of the variables to be assigned to (or equated to, in the case of an `equation` section) wrapped by a pair of parentheses.

As this discussion demonstrates, there are many different ways to call a function in Modelica.

Important Restrictions

In general, we can perform the same kinds of calculations in functions as we can in models. But there are some important restrictions.

1. Input arguments are read only - You are not allowed to assign a value to a variable which is an input argument to the function.
2. You are not allowed to reference the global variable `time` from within a function.
3. No equations or when statements - A function can have no more than one `algorithm` section and it cannot contain `when` statements.
4. The following functions cannot be invoked from a function: `der`, `initial`, `terminal`, `sample`, `pre`, `edge`, `change`, `reinit`, `delay`, `cardinality`, `inStream`, `actualStream`
5. Arguments, results and intermediate (`protected`) variables cannot be models or blocks.
6. Array sizes are restricted - Arguments that are arrays can have *Unspecified Dimensions* (page 102) and the size will be implicitly determined by the context in which the function is invoked. Results that are arrays must have their sizes specified in terms of constants or in relation to the sizes of input arguments.

One important thing to note is that functions are **not** restricted in terms of recursion (*i.e.*, a function **is** allowed to call itself).

Side Effects

In the *Software-in-the-Loop Controller* (page 131) example, we introduced external functions that had side effects. This means that the value returned by the function was not strictly a function of its arguments. Such a function is said to have “side effects”. Functions with side effects, should be qualified with the `impure` keyword. This tells the Modelica compiler that these functions cannot be treated as purely mathematical functions.

The use of `impure` functions is restricted. They can only be invoked from within a `when` statement or another `impure` function.

Function Template

Taking all of this into account, the following can be considered a generalized function definition:

```
function FunctionName "A description of the function"
  input InputType1 argName1 "description of argument1";
  ...
  input InputTypeN argNameN := defaultValueN "description of argumentN";
  output OutputType1 returnName1 "description of return value 1";
  ...
  output OutputTypeN returnNameN "description of return value N";
protected
  InterType1 intermedVarName1 "description of intermediate variable 1";
  ...
  InterTypeN intermedVarNameN "description of intermediate variable N";
  annotation(key1=value1,key2=value2);
algorithm
  // Statements that use the values of argName1..argNameN
  // to compute intermedVarName1..intermedVarNameN
  // and ultimately returnName1..returnNameN
end FunctionName;
```

Control Flow

In some cases, a function is simply a fixed set of step by step calculations. But in other cases, some kind of looping or iteration is required. In this section, we'll cover the different control structures that are allowed within a function definition.

Branching

We've already seen use cases involving `if` statements and expressions. These are, of course, allowed inside functions as well. In fact, in an `equation` section there is a restriction on `if` statements that each branch of the `if` statement (*i.e.*, under all conditions) generate the same number of equations. But that restriction does not apply in an `algorithm` section (*e.g.*, in a function definition).

Looping

In an `equation` section, looping is (just like with branching) restricted to ensure that the number of equations generated is the same regardless of the state of the system. For this reason, the only looping construct allowed in an `equation` section (and, therefore, the only one we have discussed up until now) is the `for` loop.

The syntax of a `for` loop is the same in a function as it is in any other context. It identifies an iteration variable and then assigns that iteration variable a set of values contained in a vector, *e.g.*,

```
algorithm
  for i in 1:10 loop
    // Statements
  end for;
```

There two main differences between an `equation` section and an `algorithm` section is that an `algorithm` section uses explicit assignment statements instead of equations and, since there are no equations, there are no concerns about generating a specified number of equations when using `if` or `for`.

In addition, an `algorithm` section allows us the opportunity to be more flexible by permitting the use of `while` loops as well. A `while` loop is not permitted in an `equation` section because, by its very nature, the number of iterations (and, therefore, the number of equations created in an `equation` section) is unpredictable. But this unpredictability is not an issue in an `algorithm` section.

As we already saw in the `InterpolateVector` function from our previous discussion of *Interpolation* (page 122), the syntax for a `while` loop is:

```
while x>=ybar[i+1,1] loop
  i := i + 1;
end while;
```

The main elements of the `while` loop are the condition expression that determines whether to continue looping and the statements within the `while` loop.

`break` and `return`

When iterating, it is sometimes necessary to terminate the iterations prematurely. For example, in a `for` loop, the number of iterations is normally determined by the vector of values being iterated over. But there are cases where subsequent iterations are unnecessary. Similarly, in a `while` loop, it may be convenient to have a check within the `while` loop that indicates when to terminate. In these cases, a `break` statement can be used to terminate the innermost loop.

Another issue of control flow involves when to terminate and exit from the `algorithm` section itself. There are many circumstances in which all the `output` variables have been assigned their final values. While it is always true that `if` and `else` statements can be used to prevent any further calculations and

assignments, it is often more readable to simply indicate clearly that no further calculations are needed. In such cases, the `return` statement can be used to terminate any further processing within a function's `algorithm` section. When a `return` statement is encountered, whatever values are currently associated with the `output` variables are the ones that will be returned.

External Functions

`external`

As we saw with the `InterpolateExternalVector` function in our [Interpolation](#) (page 122) related examples, it is possible to call functions not written in Modelica. Typically, such functions are written in C or Fortran. A function implemented outside Modelica does not contain an `algorithm` section. Instead, it should include an `external` statement that provides information about the external function and how to pass information to and from the function.

The minimal requirement for an externally implemented function is to include just the `external` keyword, *e.g.*,

```
external;
```

In this case, it is assumed that the function is implemented in C, that the name of the function matches the name of the Modelica "wrapper" function and that the arguments are passed in the same order as the `input` arguments to the Modelica function.

Let's consider a slightly more complex case like the one found in our `VectorTable` type shown in the [Interpolation](#) (page 122) examples:

```
function destructor "Release storage"
  input VectorTable table;
  external "C" destroyVectorTable(table)
  annotation(IncludeDirectory="modelica://ModelicaByExample.Functions.Interpolation/source
  ↵",
             Include="#include \"VectorTable.c\"");
end destructor;
```

Here we see that the implementation language of the function has been explicitly specified as "C". There are two other possible values for the implementation language, "FORTRAN 77" and "builtin". The use of "builtin" is mainly of interest to Modelica tool vendors.

We also see that the name of the function has been explicitly specified. The portion of the `external` statement that reads `destroyVectorTable(table)` specifies what data should be passed to the external function and in what order.

In some cases, it may be necessary to specify some of the values passed to the external function and to specify how the results of the function call map to the output variables. We can see this kind of information in the following function:

```
function constructor
  input Real ybar[:,2];
  output VectorTable table;
  external "C" table=createVectorTable(ybar, size(ybar,1))
  annotation(IncludeDirectory="modelica://ModelicaByExample.Functions.Interpolation/source
  ↵",
             Include="#include \"VectorTable.c\"");
end constructor;
```

Here, the external function needs to know the size of the `ybar` array since that information is not passed directly to the function. It also indicates that the result from `createVectorTable` should be assigned to the `output` variable `table`. It might seem obvious that the return value of the C function should be treated as the return value from the Modelica function, but there are cases where the `output` variables

should be passed as arguments to the function. As we will see shortly, in such cases pointers are used so that the external function can assign values to these variables.

Data Mapping

C

The following table shows how Modelica types map into native C types when passing data **into** external functions.

Modelica	C (input arguments)	C (output arguments)
Real	double	double *
Integer	int	int *
Boolean	int	int *
String	const char *	const char **
T[d1]	T' *, size_t d1	T' *, size_t d1
T[d1,d2]	T' *, size_t d1, size_t d2	T' *, size_t d1, size_t d2
T[d1,...,dn]	T' *, size_t d1, ..., size_t dn	T' *, size_t d1, ..., size_t dn
size(...)	size_t	N/A
enumeration	int	int *
record	struct *	struct *

A few additional comments about this table. First, it is assumed that all strings are null (\0) terminated. Also, in the case of arrays the type T' indicates the C type that the Modelica type T would be mapped to (using this same table). Finally, a **record** is mapped to a **struct** in C where the members of the C structure correspond in order to the members of the Modelica **record**. Types of members in **records** are mapped using the second column of this table (*i.e.*, as if they were input arguments).

For data returned by a C function, the following mapping applies:

Modelica	C
Real	double
Integer	int
Boolean	int
String	const char *
T[d1]	T' *, size_t
T[d1,d2]	T' *, size_t d1, size_t d2
T[d1,...,dn]	T' *, size_t d1, ..., size_t dn
size(...)	size_t
enumeration	int
record	struct *

Fortran

If you need to work with Fortran functions or subroutines, the following type mappings apply:

Modelica	Fortran
Real	DOUBLE PRECISION
Integer	INTEGER
Boolean	LOGICAL
T[d1]	T', INTEGER
T[d1,d2]	T', INTEGER d1, INTEGER d2
T[d1,...,dn]	T', INTEGER d1, ..., INTEGER dn
size(...)	INTEGER
enumeration	INTEGER

Two important things to note about this table. First, there is no mapping for strings or records. Second, since Fortran uses pass by reference semantics, all variables passed into and out of these functions are assumed to be pointers. For this reason, there is no distinction made whether the variable is an input or an output of the Modelica function.

Special Functions

There are a number of functions that can be called **from** external functions to interact with the Modelica runtime. For each function, the name of the function, its prototype and an explanation of the functions purpose are described.

ModelicaVFormatMessage

```
void ModelicaVFormatMessage(const char*string, va_list);
```

Output the message under the same format control as the C-function `vprintf`.

ModelicaError

```
void ModelicaError(const char* string);
```

Output the error message string (no format control). This function never returns to the calling function, but handles the error similarly to an assert in the Modelica code.

ModelicaFormatError

```
void ModelicaFormatError(const char* string, ...);
```

Output the error message under the same format control as the C-function `printf`. This function never returns to the calling function, but handles the error similarly to an assert in the Modelica code.

ModelicaVFormatError

```
void ModelicaVFormatError(const char* string, va_list);
```

Output the error message under the same format control as the C-function `vprintf`. This function never returns to the calling function, but handles the error similarly to an assert in the Modelica code.

ModelicaAllocateString

```
char* ModelicaAllocateString(size_t len);
```

Allocate memory for a Modelica string which is used as return argument of an external Modelica function. Note, that the storage for string arrays (= pointer to string array) is still provided by the calling program, as for any other array. If an error occurs, this function does not return, but calls [ModelicaError](#) (page 144).

ModelicaAllocateStringWithErrorReturn

```
char* ModelicaAllocateStringWithErrorReturn(size_t len);
```

Same as [ModelicaAllocateString](#) (page 145), except that in case of error, the function returns 0. This allows the external function to close files and free other open resources in case of error. After cleaning up resources, use [ModelicaError](#) (page 144) or [ModelicaVFormatError](#) (page 144) to signal the error.

Function Annotations

We've already discussed [Annotations](#) (page 37) in general. Modelica includes a number of standard annotations that are specifically used in conjunction with functions. These meaning of these annotations is formally defined in the Modelica Specification. In this section, we'll talk about the three general categories of annotations for functions and provide some discussion of why you need them and how to use them.

Mathematical Annotations

The first class of annotations are ones that provide additional mathematical information about the function. Because functions are written using `algorithm` sections, it is not generally possible to derive equations for the behavior of the function and many symbolic manipulations are therefore not possible. However, using the annotations in this section allows us to augment the function definition with such information.

derivative

As was saw in the ref:*polynomial-evaluation* example, there are circumstances where we would like to inform the Modelica compiler how to compute the derivative of a given function. This is done by adding the `derivative` annotation in the function.

Simple First Derivative

The basic use of the `derivative` annotation is to specify the name of another Modelica function that computes the first derivative of the function being annotated. For example:

```
function f
  input Real x;
  input Real y;
  output Real z;
  annotation(derivative=df);
  algorithm
    z := // some expression involving x and y
end f;
```

```

function df
  input Real x;
  input Real y;
  input Real dx;
  input Real dy;
  output Real dz;
algorithm
  dz := // some expression involving x, y, dx and dy
end df;

```

Note that the first arguments to the derivative function, `df`, in this case, are the same as for the original function, `f`. Those arguments are then followed by the differential versions of the input arguments to the original function. Finally, the output(s) of the derivative function are the differential versions of the output(s) of the original function. It sounds complicated, but hopefully the same code conveys how simple the construction really is.

Given such a Modelica function, the Modelica compiler can use such a function to compute various derivatives, *e.g.*,

$$\frac{df}{dv}(x, y) = df(x, y, \frac{\partial x}{\partial v}, \frac{\partial y}{\partial v})$$

Insensitivity to Some Arguments

Now consider a case where $\frac{\partial y}{\partial v}$ is zero. The derivative function will be passed this zero value or an array of zero values, if the argument was an array. That zero value will then be used in several calculations inside the derivative function. Most, if not all, of these will be multiplications and the results of those calculations will therefore be zeros. Those zeros will then be added to the final result, but will have no impact. In other words, there are many calculations that could be skipped because they won't have any impact on the result.

In such cases, Modelica offers a way to avoid these calculations. If the Modelica compiler knows *a priori* that one of the differentials is zero, it can check (among the set of `derivative` annotations) if there are any functions that compute the derivative for that case. These cases are specified using the `zeroDerivative` argument to the `derivative` annotation. So, in the case of our example function `f`, we could add the following annotation:

```

function f
  input Real x;
  input Real y;
  output Real z;
  annotation(derivative=df, derivative(zeroDerivative=y)=df_onlyx);
algorithm
  z := // some expression involving x and y
end f;

```

where `df_onlyx` would then be defined as:

```

function df_onlyx
  input Real x;
  input Real y;
  input Real dx;
  output Real dz;
algorithm
  dz := // some expression involving x, y, dx
end df_onlyx;

```

Note that the `dy` term is not included here. This function is specifically for cases where `dy` is zero. Because `dy` doesn't appear in the arguments, this function includes only those calculations involving `dx`.

Second Derivatives

There are a few more variations worth covering here. The first is how to specify what the **second** derivative of a function is. This is done by adding an **order** argument. Note that a function can have multiple **derivative** annotations, *e.g.*,

```
function f
  input Real x;
  input Real y;
  output Real z;
  annotation(derivative=df, derivative(order=2)=ddf);
algorithm
  z := // some expression involving x and y
end f;

function df
  ...
end df;

function ddf
  input Real x;
  input Real y;
  input Real dx;
  input Real dy;
  input Real ddx;
  input Real ddy;
  output Real ddz;
algorithm
  ddz := // some expression involving x, y, dx, dy,
         // ddx and ddz
end ddf;
```

Hopefully there are no real surprises here. In order to compute the second derivative, it is necessary to add an additional annotation **derivative** annotation to the original function, *i.e.*,

```
annotation(derivative=df, derivative(order=2)=ddf);
```

This additional annotation has an additional argument **order** which indicates which derivative that function computes.

Non-Real Arguments

There is one additional complication to discuss. What if the function has arguments that don't represent real numbers, *e.g.*,

```
function g
  input Real x;
  input Integer y;
  output Real z;
algorithm
  z := // some expression involving x and y
end g;
```

Here, it makes no sense to take the derivative of this function with respect to the **y** argument, since it is an integer. Any non-real argument can be ignored when formulating the derivative. So, if we wished to compute the derivative of this function, we would do it as follows:

```
function g
  input Real x;
  input Integer y;
```

```
output Real z;
annotation(derivative=dg);
algorithm
  z := // some expression involving x and y
end g;

function dg
  input Real x;
  input Integer y;
  input Real dx;
  output Real dz;
algorithm
  dz := // some expression involving x, y and dx
end dg;
```

In other words, the differential arguments only apply to arguments that are real.

inverse

During our discussion on *Non-Linearity* (page 134), we showed how the `inverse` annotation can be used to tell the Modelica compiler how to compute the inverse of a function. The goal of an inverse function is to solve explicitly for one of the current function's input arguments. As such, the `inverse` annotation contains an explicit equation involving the input and output variables of the current function, but used in conjunction with another function to explicitly compute one of the input arguments.

For example, for a Modelica function defined as follows:

```
function h
  input Real a;
  input Real b;
  output Real c;
  annotation(inverse(b = h_inv_b(a, c)));
algorithm
  c := // some calculation involving a and b
end h;
```

we see that `b` can be computed by passing `a` and `c` as arguments to the function `h_inv_b` which would be defined as follows:

```
function h_inv_b
  input Real a;
  input Real c;
  output Real b;
algorithm
  b := // some calculation involving a and c
end h_inv_b;
```

Code Generation

The next class of annotations are related to how function definitions are translated into code for simulation. These annotations allow the model developer to provide hints to the Modelica compiler on how the code generation process should be done.

Inline

The `Inline` annotation is a hint to the Modelica compiler that the statements in the function should be “inlined”. The value of the annotation is used to suggest whether inlining should be done. The default

value (if no `Inline` annotation is present) is `false`. The following is a function that uses the `Inline` annotation:

```
function SimpleCalculation
  input Real x;
  input Real y;
  output Real z;
  annotation(Inline=true);
algorithm
  z := 2*x-y;
end SimpleCalculation;
```

Here we see that the `Inline` annotation suggests that the Modelica compiler should inline the `SimpleCalculation` function. The function is inlined by replacing invocations of the function with the statements in the function that compute the output result. This is useful for functions that perform very simple calculations. In those cases, the “cost” (in CPU time) of calling the function is on the same order of magnitude as the cost of the work performed by the function. By inlining the function, the cost of the function call can be eliminated while still preserving the purpose of the function.

The `Inline` function is merely a hint to the Modelica compiler. The compiler is not obligated to inline the function. Also, the compiler’s ability to inline the function will depend on the complexity of the function. It is not necessary possible (or even desirable) to inline a function in general.

LateInline

Much like the `Inline` (page 148) annotation, the `LateInline` function tells the Modelica compiler that it would be more efficient to inline the function. The `LateInline` annotation is also assigned a Boolean value to specify whether the function should be inlined or not. The difference between the `Inline` and `LateInline` annotations is that `LateInline` indicates that inlining should be performed after symbolic manipulation has been performed. A full discussion of the potential interactions between inlining and other symbolic manipulations is beyond the scope of this book.

It should be noted that the `LateInline` annotation takes precedence over the `Inline` annotation if they are both applied to a function, *i.e.*,

Inline	LateInline	Interpretation
<code>false</code>	<code>false</code>	<code>Inline=false</code>
<code>true</code>	<code>false</code>	<code>Inline=true</code>
<code>false</code>	<code>true</code>	<code>LateInline=true</code>
<code>true</code>	<code>true</code>	<code>LateInline=true</code>

External Functions

The final class of annotations are related to functions that are defined as `external`. Such functions often depend on external include files or libraries. These annotations inform the Modelica compiler of these dependencies and where to locate them.

Include

The `Include` annotations is used whenever the code generated by a Modelica compiler requires an include statement. Typically this is required when external libraries are being referenced. The value of the `Include` annotation should be the string that should be inserted into the generated code, *e.g.*,

```
annotation(Include="#include \"mydefs.h\"");
```

Note: The value of the `Include` annotation is a string. If it included embedded strings, they need to be escaped.

IncludeDirectory

As already discussed, the `Include` (page 149) annotation allows include directives to be inserted into generated code. The `IncludeDirectory` annotation specifies what directory should be searched to find the content specified with the `Include` annotation.

The value of this annotation is a string. The string can represent a directory or it can be a URL. For example, the default value for the `IncludeDirectory` annotation is:

```
IncludeDirectory=modelica://LibraryName/Resources/Include
```

We'll explain the meaning of these `modelica:// URLs` (page 160) shortly.

Library

The `Library` annotation is used to specify any compiled libraries that a function might depend on. The value of library can be either a simple string, representing the name of the library, or an array of such strings, *i.e.*,

```
annotation(Library="somelib");
```

or

```
annotation(Library={"onelib", "anotherlib"});
```

The Modelica compiler will then use this information during the “linking” of the generated code.

LibraryDirectory

We have the same issue with `Library` that we have with `Include`. The `Library` annotation tells us what we need to add, but not where to find it. In this way, the `LibraryDirectory` annotation serves the same role as the `IncludeDirectory` (page 150) annotation. Like the `IncludeDirectory` annotation, it can also be a URL. Its default value is:

```
LibraryDirectory=modelica://LibraryName/Resources/Library
```

OBJECT-ORIENTED MODELING

2.1 Packages

So far, we've presented all of the models without any real discussion of how to properly organize them. In some cases, like the `NewtonCoolingWithTypes` example in our discussion on *adding physical type information* (page 9), it is awkward to keep everything within a single model. There are many cases where the same information gets repeated in multiple models and this makes maintaining those models very difficult.

The good news is that many of these previous examples can be greatly improved by leveraging the `package` system in Modelica. A `package` is conceptually like a directory. It holds a collection of Modelica entities. These entities can then be referenced or imported to avoid duplication.

This chapter provides several examples that demonstrate how to use the package features in Modelica. This chapter will conclude with a discussion of the *Modelica Standard Library* (page 163), which contains an enormous amount of reusable content that is available to all Modelica tools.

2.1.1 Examples

Organizing Content

Let's start by simply demonstrating how content can be organized into packages. To do this, we will revisit the *Classic Lotka-Volterra* (page 18) model. In our previous models, all variables had the type `Real`. Let's enhance that model to include types for the various quantities in the model.

We can organize those types into a package like this:

```
package Types
    type Rabbits = Real(quantity="Rabbits", min=0);
    type Wolves = Real(quantity="Wolves", min=0);
    type RabbitReproduction = Real(quantity="Rabbit Reproduction", min=0);
    type RabbitFatalities = Real(quantity="Rabbit Fatalities", min=0);
    type WolfReproduction = Real(quantity="Wolf Reproduction", min=0);
    type WolfFatalities = Real(quantity="Wolf Fatalities", min=0);
end Types;
```

The first thing to note about this Modelica code is that it uses the `package` keyword. The syntax of a `package` definition is very similar to the definition of a `model` or `function`. The main difference is that a `package` contains only definitions or constants. It cannot contain any variable declarations except those that are `constant`. In this case, we see that this `package` contains only `type` definitions.

Now let's turn our attention to the Lotka-Volterra model itself. Assuming it doesn't need to define the types itself, but can rely on the types we've just defined, it can be refactored to look as follows:

```
model LotkaVolterra "Lotka-Volterra with types"
    parameter Types.RabbitReproduction alpha=0.1;
    parameter Types.RabbitFatalities beta=0.02;
```

```

parameter Types.WolfReproduction gamma=0.4;
parameter Types.WolfFatalities delta=0.02;
parameter Types.Rabbits x0=10;
parameter Types.Wolves y0=10;
Types.Rabbits x(start=x0);
Types.Wolves y(start=y0);
equation
  der(x) = x*(alpha-beta*y);
  der(y) = -y*(gamma-delta*x);
end LotkaVolterra;

```

Notice how all the parameters and variables now have a specific type (and not just the ordinary `Real` type). Instead, we are able to associate additional information above and beyond the fact that these are continuous variables. For example, we can specify that these values should not be negative by adding the `min=0` modifier to their type definitions.

Looking at the Lotka-Volterra model by itself, it isn't obvious `where` it finds these type definitions. The Modelica compiler will use a collection of *Lookup Rules* (page 161) to lookup these definitions. We'll come to the lookup rules eventually. For now, the important point is that we have the ability to refer to things that are not in our immediate model.

Let's "zoom out" a little bit to see some additional details related to organizing models. The `Types` package we showed earlier and the `LotkaVolterra` model that references it are contained within a package called `NestedPackages` which is defined as follows:

```

within ModelicaByExample.PackageExamples;
package NestedPackages
  "An example of how packages can be used to organize things"
  package Types
    type Rabbits = Real(quantity="Rabbits", min=0);
    type Wolves = Real(quantity="Wolves", min=0);
    type RabbitReproduction = Real(quantity="Rabbit Reproduction", min=0);
    type RabbitFatalities = Real(quantity="Rabbit Fatalities", min=0);
    type WolfReproduction = Real(quantity="Wolf Reproduction", min=0);
    type WolfFatalities = Real(quantity="Wolf Fatalities", min=0);
  end Types;

  model LotkaVolterra "Lotka-Volterra with types"
    parameter Types.RabbitReproduction alpha=0.1;
    parameter Types.RabbitFatalities beta=0.02;
    parameter Types.WolfReproduction gamma=0.4;
    parameter Types.WolfFatalities delta=0.02;
    parameter Types.Rabbits x0=10;
    parameter Types.Wolves y0=10;
    Types.Rabbits x(start=x0);
    Types.Wolves y(start=y0);
  equation
    der(x) = x*(alpha-beta*y);
    der(y) = -y*(gamma-delta*x);
  end LotkaVolterra;
end NestedPackages;

```

A really important thing to note about the `NestedPackages` package is that it is contained inside another package called `PackageExamples` which is, in turn, contained within a package called `ModelicaByExample`. We know this from the `within` clause at the top:

```
within ModelicaByExample.PackageExamples;
```

Every single model that we've simulated so far in this book is contained within a package. When we showed the source code to those examples, we clipped the top line because we were not yet ready to discuss what the `within` clause was used for. But it was there in all cases.

Note that the `Types` package and the `LotkaVolterra` model don't include any kind of `within` clause.

That's because we **know** what package they are in because they are defined directly inside the `NestedPackages` package. So why does it appear immediately before the definition of `NestedPackages`? Because the `NestedPackages` package is a stand-alone file. In other words, when Modelica definitions are mapped into files and directories, we need to explicitly specify how they are related. We'll discuss the relationship between files, directories and *Package Definitions* (page 157) later. For now, the important thing to understand is that the `within` clause is simply used to specify the parent package.

Referencing Package Contents

Now that we've covered *Organizing Content* (page 151), we'll discuss how to access that content across different packages. Let's consider the following example:

```
within ModelicaByExample.PackageExamples;
model RLC "An RLC circuit referencing types from the Modelica Standard Library"
  parameter Modelica.SIunits.Voltage Vb=24 "Battery voltage";
  parameter Modelica.SIunits.Inductance L = 1;
  parameter Modelica.SIunits.Resistance R = 100;
  parameter Modelica.SIunits.Capacitance C = 1e-3;
  Modelica.SIunits.Voltage V(fixed=true, start=0);
  Modelica.SIunits.Current i_L(fixed=true, start=0);
  Modelica.SIunits.Current i_R;
  Modelica.SIunits.Current i_C;
equation
  i_R = V/R;
  i_C = C*der(V);
  i_L=i_R+i_C;
  L*der(i_L) = (Vb-V);
end RLC;
```

As we learned in the previous section, the very first line,

```
within ModelicaByExample.PackageExamples;
```

tells us that the `RLC` model is contained within the `ModelicaByExample.PackageExamples` package. As with the previous example, we are going to make use of the Modelica package system to allow us to avoid defining types directly in our model. In this way, we define the types once in one package and then we can reuse them in many places simply by referencing them.

Unlike the previous example in this chapter, we don't define any types in this example. Instead, we rely on types that are defined in the *Modelica Standard Library* (page 163). The *Modelica Standard Library* (page 163) contains many useful types, models, constants, *etc*. For this example, we'll just utilize a few of them. These types can be easily recognized because they start with `Modelica.` in the name of the type.

We look more closely at the *Lookup Rules* (page 161) later in this chapter. For now, it is sufficient to say that all the types starting with `Modelica.` exist within the `Modelica` package. In this case, all types start with `Modelica.SIunits`. `SIunits` is a package within the `Modelica` package. The purpose of the `SIunits` package is to store type definitions that conform to ISO standard quantities and units.

As can be seen in the example code, these types are referenced by their “fully qualified name”. That means that type name starts with the name of a top-level package (a package that is not contained within another package). Each `.` in the name represents a new child package. The last name in the sequence identifies that actual type being referenced.

In this case, we are using 5 different types from within the `Modelica.SIunits` package: `Voltage`, `Inductance`, `Resistance`, `Capacitance` and `Current`. These types provide information about the units for each of these types, limitations on the values of these types (*e.g.*, a capacitance cannot be less than zero), *etc*. They are defined in the *Modelica Standard Library* (page 163) as follows:

```
// Base Definitions
type ElectricPotential = Real(final quantity="ElectricPotential",
```

```

        final unit="V");
type ElectricCurrent = Real(final quantity="ElectricCurrent",
                               final unit="A");

// The types referenced in our example
type Voltage = ElectricPotential;
type Inductance = Real(final quantity="Inductance",
                       final unit="H");
type Resistance = Real(final quantity="Resistance",
                       final unit="Ohm");
type Capacitance = Real(final quantity="Capacitance",
                        final unit="F", min=0);
type Current = ElectricCurrent;

```

Apart from providing better documentation, there is an immediate benefit to associating such types with variables and that is because it enables unit consistency checking of the equations. For example, note the following equation from this example:

```
i_R = V/R;
```

Clearly, this is a statement of Ohm's law. But what if we made a mistake and accidentally wrote:

```
i_R = V*R;
```

Syntactically speaking, this equation is perfectly legal. Furthermore, if the variable `i_R`, `V` and `R` were all declared to have the type `Real`, there would be no issue with this equation. However, because we know (from the type definitions) that these variables represent a current, a voltage and a resistance, respectively, a Modelica compiler is able to determine (in a completely automatic way using the definitions shown previously) that the left and right hand sides of this equation are inconsistent with respect to physical units. In other words, by associating a physical type with variables it is possible to detect modeling errors, automatically.

Importing Physical Types

In the previous section, we learned how to reference types defined in other packages. This spares the developer from having to constantly define things in their local model. Instead, they can place definitions in packages and then reference those packages.

However, references with long fully qualified names can be tedious to type over and over again. For that reason, Modelica includes an `import` statement that allows us to use a definition as if it were defined locally.

Recall again, this example from a previous discussion on *Physical Types* (page 9):

```

within ModelicaByExample.BasicEquations.CoolingExample;
model NewtonCoolingWithTypes "Cooling example with physical types"
    // Types
    type Temperature=Real(unit="K", min=0);
    type ConvectionCoefficient=Real(unit="W/(m2.K)", min=0);
    type Area=Real(unit="m2", min=0);
    type Mass=Real(unit="kg", min=0);
    type SpecificHeat=Real(unit="J/(K.kg)", min=0);

    // Parameters
    parameter Temperature T_inf=298.15 "Ambient temperature";
    parameter Temperature T0=363.15 "Initial temperature";
    parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
    parameter Area A=1.0 "Surface area";
    parameter Mass m=0.1 "Mass of thermal capacitance";
    parameter SpecificHeat c_p=1.2 "Specific heat";

```

```
// Variables
Temperature T "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingWithTypes;
```

The previous section described how we could avoid defining these types locally by using types from the *Modelica Standard Library* (page 163). But we can also use the `import` command to import those types from the Modelica Standard Library once and then use them without having to specify their fully qualified names. The resulting code would look something like:

```
within ModelicaByExample.PackageExamples;
model NewtonCooling
  "Cooling example importing physical types from the Modelica Standard Library"
  import Modelica.SIunits.Temperature;
  import Modelica.SIunits.Mass;
  import Modelica.SIunits.Area;
  import ConvectionCoefficient = Modelica.SIunits.CoefficientOfHeatTransfer;
  import SpecificHeat = Modelica.SIunits.SpecificHeatCapacity;

  // Parameters
  parameter Temperature T_inf=300.0 "Ambient temperature";
  parameter Temperature T0=280.0 "Initial temperature";
  parameter ConvectionCoefficient h=0.7 "Convective cooling coefficient";
  parameter Area A=1.0 "Surface area";
  parameter Mass m=0.1 "Mass of thermal capacitance";
  parameter SpecificHeat c_p=1.2 "Specific heat";

  // Variables
  Temperature T "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCooling;
```

Here we have replaced the type definitions with `import` statements. Note how the highlighted lines are identical to the previous code. Let's look at two of these import statements more closely to understand what effect they have on the model. Let's start with the following import statement:

```
import Modelica.SIunits.Temperature;
```

This imports the type `Modelica.SIunits.Temperature` into the current model. By default, the name of this imported type will be the last name in the fully qualified name, *i.e.*, `Temperature`. This means that with this `import` statement present, we can simply use the type name `Temperature` and that will automatically refer back to `Modelica.SIunits.Temperature`.

Now let's look at another `import` statement:

```
import ConvectionCoefficient = Modelica.SIunits.CoefficientOfHeatTransfer;
```

The syntax here is a little bit different. In this case, the type that we are importing is `Modelica.SIunits.CoefficientOfHeatTransfer`. But instead of creating a local type based on the last name in the fully qualified name, *i.e.*, `CoefficientOfHeatTransfer` we are specifying that the local type should be `ConvectionCoefficient`. In this case, this allows us to use the name we originally used in our earliest examples. In this way, we can avoid refactoring any code that used the previous name. Another reason for specifying an alternative name (other than the default one that the Modelica compiler would normally assign) would be to avoid name collision. Imagine we wished to import two types from two different packages, *e.g.*,

```
import Modelica.SIunits.Temperature; // Celsius
import ImperialUnits.Temperature; // Fahrenheit
```

This would leave us two types both named `Temperature`. By defining an alternative name for the local alias, we could do something like this:

```
import DegK = Modelica.SIunits.Temperature; // Kelvin
import DegR = ImperialUnits.Temperature; // Rankine
```

SI Units

Note that this example imports imperial units just to demonstrate how a potential name clash might occur. But it is very bad practice to do this in practice. When using Modelica you should always use SI units and never use any other system of units. If you want to enter data or display results in other units, please use the `displayUnit` attribute discussed previously in the section on *Attributes* (page 30).

There is one last form of the `import` statement worth discussing which is the wildcard import statement. Importing units one unit at a time can be tedious. The wildcard import allows us to import **all** types from a given package at once. Recall the following earlier example:

```
within ModelicaByExample.BasicEquations.RotationalSMD;
model SecondOrderSystem "A second order rotational system"
  type Angle=Real(unit="rad");
  type AngularVelocity=Real(unit="rad/s");
  type Inertia=Real(unit="kg.m2");
  type Stiffness=Real(unit="N.m/rad");
  type Damping=Real(unit="N.m.s/rad");
  parameter Inertia J1=0.4 "Moment of inertia for inertia 1";
  parameter Inertia J2=1.0 "Moment of inertia for inertia 2";
  parameter Stiffness c1=11 "Spring constant for spring 1";
  parameter Stiffness c2=5 "Spring constant for spring 2";
  parameter Damping d1=0.2 "Damping for damper 1";
  parameter Damping d2=1.0 "Damping for damper 2";
  Angle phi1 "Angle for inertia 1";
  Angle phi2 "Angle for inertia 2";
  AngularVelocity omega1 "Velocity of inertia 1";
  AngularVelocity omega2 "Velocity of inertia 2";
initial equation
  phi1 = 0;
  phi2 = 1;
  omega1 = 0;
  omega2 = 0;
equation
  // Equations for inertia 1
  omega1 = der(phi1);
  J1*der(omega1) = c1*(phi2-phi1)+d1*der(phi2-phi1);
  // Equations for inertia 2
  omega2 = der(phi2);
  J2*der(omega2) = c2*(phi1-phi2)+d2*der(phi1-phi2);
end SecondOrderSystem;
```

We could replace these type definitions with import statements, *e.g.*,

```
import Modelica.SIunits.Angle;
import Modelica.SIunits.AngularVelocity;
import Modelica.SIunits.Inertia;
import Stiffness = Modelica.SIunits.RotationalSpringConstant;
import Damping = Modelica.SIunits.RotationalDampingConstant;
```

However, the more types we bring in, the more import statements we need to add. Instead, we could write our model as follows:

```
within ModelicaByExample.PackageExamples;
model SecondOrderSystem
  "A second order rotational system importing types from Modelica Standard Library"
  import Modelica.SIunits.*;
  parameter Angle phi1_init = 0;
  parameter Angle phi2_init = 1;
  parameter AngularVelocity omega1_init = 0;
  parameter AngularVelocity omega2_init = 0;
  parameter Inertia J1=0.4;
  parameter Inertia J2=1.0;
  parameter RotationalSpringConstant c1=11;
  parameter RotationalSpringConstant c2=5;
  parameter RotationalDampingConstant d1=0.2;
  parameter RotationalDampingConstant d2=1.0;
  Angle phi1;
  Angle phi2;
  AngularVelocity omega1;
  AngularVelocity omega2;
initial equation
  phi1 = phi1_init;
  phi2 = phi2_init;
  omega1 = omega1_init;
  omega2 = omega2_init;
equation
  omega1 = der(phi1);
  omega2 = der(phi2);
  J1*der(omega1) = c1*(phi2-phi1)+d1*der(phi2-phi1);
  J2*der(omega2) = c1*(phi1-phi2)+d1*der(phi1-phi2)-c2*phi2-d2*der(phi2);
end SecondOrderSystem;
```

Note the highlighted `import` statement. This single (wildcard) import statement imports all definitions from `Modelica.SIunits` into the current model. With wildcard imports, there is no option to “rename” the types. They will have exactly the name locally as they have in the named package.

Before using wildcard imports, be sure to read [this caveat](#) (page 162).

In this chapter, we’ve seen how `import` statements can be used to import types from other packages. As it turns out, `import` statements are not always that useful. When models are being developed within a graphical modeling environment, tools generally use the least ambiguous and most explicit method for reference types: using fully qualified names. After all, when using a graphical tool the length of the name is not an issue because it doesn’t need to be typed. This also avoids issues with name lookup, naming conflicts, etc.

2.1.2 Review

Package Definitions

Basic Syntax

As we learned in this chapter, a `package` is a Modelica entity that allows us to organize definitions (including definitions of other packages). The syntax definition of a `package` has a lot in common with other Modelica definitions. The general syntax for a `package` is:

```
package PackageName "Description of package"
  // A package can contain other definitions or variables with the
  // constant qualifier.
end PackageName;
```

A package definition can be prefixed by the `encapsulated` qualifier. We'll discuss that more when we examine Modelica's *Lookup Rules* (page 161).

Packages can also be nested, *e.g.*,

```
package OuterPackage "A package that wraps a nested package"
  // Anything contained in OuterPackage
  package NestedPackage "A nested package"
    // Things defined inside NestedPackage
  end NestedPackage;
end OuterPackage;
```

In fact, nesting of packages is very common and allows us to represent complex taxonomies.

Directory Storage

Although it is possible to build an entire library of Modelica definitions in a single file as a series of nested packages, this is undesirable for at least two reasons. The first is that the resulting file would be quite hard to read based on its length and the degree of indenting that would be required. The second is that from the standpoint of version control, it is much better to break things into smaller files to help avoid any merge conflicts.

Stored in a Single File

There are several ways that Modelica source code can be mapped to a file system. The simplest way is to store everything in a file. Such a file should have a `.mo` suffix. Such a file might contain only a single model definition or it might contain a deeply nested hierarchy of packages or anything in between.

Stored as a Directory

As we already discussed, storing everything in one file is usually not a good idea. The alternative is to map Modelica definitions into a directory structure. A package can be stored as a directory by creating a directory **with the same name as the package**. Then, inside that directory, there must be a file called `package.mo` that stores the definition of the package, **but not any nested definitions**. The nested definitions can be stored either as single files (as described above) or as directories representing packages (as described in this paragraph). The following diagram attempts to visualize a sample directory layout:

```
/RootPackage          # Top-level package stored as a directory
  package.mo           # Indicates this directory is a package
  NestedPackageAsFile.mo # Definitions stored in one file
  /NestedPackageAsDir   # Nested package stored as a directory
    package.mo           # Indicates this directory is a package
```

The `package.mo` file associated with the package named `RootPackage` would look something like this:

```
within;
package RootPackage
  // only annotations can be stored in a package.mo
end RootPackage;
```

There are two important things to note here. First, the `within` clause should be present, but empty. This indicates that this package is not contained in any other packages. In addition, the definitions of `NestedPackageAsFile` and `NestedPackageAsDir` are not (and cannot be) present. Those must be stored outside the `package.mo` file.

Similarly, the `package.mo` file associated with the `NestedPackageAsDir` package would look like this:

```
within RootPackage;
package NestedPackageAsDir
  // only annotations can be stored in a package.mo
end NestedPackageAsDir;
```

Again, there should be no definitions contained in this package, only annotations. The `within` clause is slightly different, reflecting the fact that `NestedPackageAsDir` belongs to the `RootPackage` package.

Finally, the `NestedPackageAsFile.mo` file would look something like this:

```
within RootPackage;
package NestedPackageAsFile
  // The following can be stored here including:
  // * constants
  // * nested definitions
  // * annotations
end NestedPackageAsFile;
```

The `within` clause is the same as for the `NestedPackageAsDir` package definition, but since we are storing this package as a single file, nested definitions for constants, models, functions, *etc.* are allowed here as well.

Ordering for Directories

When all definitions are stored within a single file, the order they appear in the file indicates the order they should appear when visualized (*e.g.*, in a package browser). But when they are stored on the file system, there is no implied ordering. For this reason, an optional `package.order` file can be included alongside the `package.mo` file to specify an ordering. The file is simply a list of the names for nested entities, one per line. So, for example, if we wanted to impose an ordering on this sample package structure, the file system would be populated as follows:

<code>/RootPackage</code> <code> package.mo</code> <code> package.order</code> <code> NestedPackageAsFile.mo</code> <code>/NestedPackageAsDir</code> <code> package.mo</code> <code> package.order</code>	<code># Top-level package stored as a directory</code> <code># Indicates this directory is a package</code> <code># Specifies an ordering for this package</code> <code># Definitions stored in one file</code> <code># Nested package stored as a directory</code> <code># Indicates this directory is a package</code> <code># Specifies an ordering for this package</code>
--	--

In the absence of a `package.order` file, a Modelica tool would probably simply sort packages alphabetically. But if we wanted to order the contents of the `RootPackage` in reverse alphabetical order, the `package.order` file in the `RootPackage` directory would look like this:

```
NestedPackageAsFile
NestedPackageAsDir
```

This would specify to the Modelica tool that `NestedPackageAsFile` should come before `NestedPackageAsDir`.

Versioning

MODELICAPATH

Most Modelica tools allow the user to open a file either by specifying the full path name of the file or by using a file selection dialog to open it. But it can be tedious to find and load lots of different files each time you use a tool. For this reason, the Modelica specification defines a special environment variable called `MODELICAPATH` that the user can use to specify the location of the source code they want the tool to be able to automatically locate.

The MODELICAPATH environment variable should contain a list of directories to search. On Windows, that list should be separated by a ; and under Unix it should be separated by a :. When the Modelica compiler comes across a package it has not already loaded, it will search the directories listed by the MODELICAPATH environment variable looking for either a matching file or directory. For example, if the MODELICAPATH was defined as (assuming Unix conventions):

```
/home/mtiller/Dir1:/home/mtiller/Dir2
```

and the compiler was looking for a package called `MyLib`, it would first look in `/home/mtiller/Dir1` for either a package named `MyLib.mo` (stored as a single file) or a directory named `MyLib` that contained a `package.mo` file that defined a package named `MyLib`. If neither of those could be found, it would then search the `/home/mtiller/Dir2` directory (for the same things).

modelica:// URLs

In many cases, it is useful to include non-Modelica files along with a Modelica package. These non-Modelica files might contain data, scripts, images, etc. We call these non-Modelica files “resources”. Now that we’ve covered how Modelica definitions are mapped to a file system, we can introduce an extremely useful feature in Modelica which is the use of URLs to refer to the location of these resources.

For example, when we discussed *External Functions* (page 149), we introduced several annotations that specified the location of such resources. Specifically, the `IncludeDirectory` and `LibraryDirectory` annotations specified where the Modelica compiler should look for include and library files, respectively. As was briefly mentioned then, the default values for these annotations started with `modelica://` `LibraryName/Resources`. Such a URL allows us to define the location of resources **relative to a given Modelica definition on the file system**. Let us revisit the directory structure we discussed earlier, but with some resource files added:

```
/RootPackage          # Top-level package stored as a directory
  package.mo           # Indicates this directory is a package
  package.order         # Specifies an ordering for this package
  NestedPackageAsFile.mo # Definitions stored in one file
/NestedPackageAsDir    # Nested package stored as a directory
  package.mo           # Indicates this directory is a package
  package.order         # Specifies an ordering for this package
  datafile.mat          # Data specific to this package
/Resources              # Resources are stored here by convention
  logo.jpg             # An image file
```

If we have a model that needs the data contained in `NestedPackageAsDir`, we can use the following URL to reference it:

```
modelica://RootPackage/NestedPackageAsDir/datafile.mat
```

Such a URL starts with `modelica://`. This is our way of indicating that the resource being referenced is with respect to a Modelica model and not, for example, something to be fetched over the network. The `//` is then followed by the fully qualified name of a Modelica definition except that each component is separated by a `/` instead of a `..`. The Modelica compiler will interpret this as the name of the directory that contains that definition. Finally, the last element in the URL names the file to be used.

As another example, if we wished to reference the `logo.jpg` file in the `Resources` package, we would use the following URL:

```
modelica://RootPackage/Resources/logo.jpg
```

It is a common convention to store resources related to a library in a nested package named `Resources` (hence the default values for `IncludeDirectory` and `LibraryDirectory`).

Lookup Rules

Recall the following example from our discussion on *Organizing Content* (page 151):

```
within ModelicaByExample.PackageExamples;
package NestedPackages
  "An example of how packages can be used to organize things"
  package Types
    type Rabbits = Real(quantity="Rabbits", min=0);
    type Wolves = Real(quantity="Wolves", min=0);
    type RabbitReproduction = Real(quantity="Rabbit Reproduction", min=0);
    type RabbitFatalities = Real(quantity="Rabbit Fatalities", min=0);
    type WolfReproduction = Real(quantity="Wolf Reproduction", min=0);
    type WolfFatalities = Real(quantity="Wolf Fatalities", min=0);
  end Types;

  model LotkaVolterra "Lotka-Volterra with types"
    parameter Types.RabbitReproduction alpha=0.1;
    parameter Types.RabbitFatalities beta=0.02;
    parameter Types.WolfReproduction gamma=0.4;
    parameter Types.WolfFatalities delta=0.02;
    parameter Types.Rabbits x0=10;
    parameter Types.Wolves y0=10;
    Types.Rabbits x(start=x0);
    Types.Wolves y(start=y0);
    equation
      der(x) = x*(alpha-beta*y);
      der(y) = -y*(gamma-delta*x);
  end LotkaVolterra;
end NestedPackages;
```

When we discussed *Referencing Package Contents* (page 153), the example that was presented used fully qualified names for all the types it referenced. But the example above doesn't. We see, from the `LotkaVolterra` model that the `Wolves` type is referenced as:

```
parameter Types.Wolves y0=10;
```

And not as:

```
parameter ModelicaByExample.PackageExamples.NestedPackages.Types.Wolves y0=10;
```

In other words, we didn't use the fully qualified name. But the `LotkaVolterra` model compiles just fine. So how is it that the Modelica compiler knows which definition of `Wolves` to use?

The answer involves “name lookup” in Modelica. Name lookup in Modelica involves searching for the named definition. Type names in Modelica are generally qualified (although not necessarily **fully** qualified) names. This means they may contain a `.` in them, *e.g.*, `Modelica.SIunits.Voltage`. In order to locate the matching definition associated with a name, the Modelica compiler starts by looking for the **first** name in the qualified name, *e.g.*, `Modelica`. It searches for a matching definition in the following order:

1. Look for a matching name among builtin types
2. Look in the current definition for a nested definition with a matching name (include inherited definitions)
3. Look in the current definition for an imported definition with a matching name (do not include inherited imports)
4. Look in the parent package of the current definition for a nested definition with a matching name (including inherited definitions)
5. Look in the parent package for an imported definition with a matching name (not including inherited imports)

6. Look in each successive parent (using the same approach) until either:
 - The parent package has the `encapsulated` qualifier, in which case the search terminates.
 - There are no more parent packages, in which case you search for a match among root level packages.

If the given name cannot be found after searching all these locations, then the search fails and the type cannot be resolved. If the search succeeds, then we've located the definition of the **first name** in the qualified name. If the name is not qualified (*i.e.*, it does not have a `.` in the name), then we are done. However, if it does have other components in the name, these must be nested definitions contained within the definition returned by the search. If nested definitions cannot be found for all remaining components in a qualified name, then the search fails and the type cannot be resolved.

At first, this might sound very complicated. However, most of the time these rules are not very important. The reason is that, as we discussed previously, most graphical Modelica environments will use fully qualified names. Most type names in Modelica code will either reference local definitions or will be specified with fully qualified names.

Duplicate Names

You should always avoid having nested packages with the same name as a top-level package. The reason this is a problem is that the lookup rules search up through the package hierarchy. As a result, they will find the nested definition before the root level one. This means that lookup of fully qualified names (ones that are referenced relative to the root of the package tree) will fail because they will find the nested definition first.

Importing

As we saw previously, there are basically three forms of importing. In all cases, the `import` statement creates an “alias” within the definition that refers to a name defined elsewhere.

The first form simply imports a definition by its fully qualified name, *e.g.*:

```
import Modelica.SIunits.Temperature;
```

The result of such an import is that references to the name `Temperature` are mapped to the fully qualified name `Modelica.SIunits.Temperature`. In other words, the alias introduced by the `import` statement is `Temperature` and it maps to the definition found at `Modelica.SIunits.Temperature`. With this form of import, the name of the alias always matches the last element in the name of what is being imported (*e.g.*, `Temperature`).

In some cases, we want the alias that is introduced to be different from the last element of the imported name. In this case, we can explicitly introduce an alternative name for the alias, *e.g.*,

```
import DegK = Modelica.SIunits.Temperature; // Kelvin
```

After such an import, we can use the alias `DegK` to refer to `Modelica.SIunits.Temperature`. Providing alternative names avoids name collisions or simply makes the model more readable.

Finally, it is possible to import all definitions within a package into the current scope. This is done with a wildcard import. For example, to import all the definitions in the `Modelica.SIunits` package, we would use the following `import` statement:

```
import Modelica.SIunits.*;
```

Such an import would create as many aliases as there are definitions in `Modelica.SIunits`. The only option available is for each alias to be named the same as the definition in the imported package (*i.e.*, it isn't possible to assign an alternative name for the alias).

Wildcards considered harmful

These types of wildcard imports are dangerous because, as mentioned, there is no option to rename a type. As a consequence, two or more wildcard imports in the same model could create name clashes. Furthermore, explicit imports (or fully qualified types) make it very easy to backtrack and locate the definition associated with a given type. Wildcards make this very difficult because it is not clear what types are imported from what packages.

If you want to import multiple entries from the same package you can use a special syntax.

```
import Modelica.SIunits.{Temperature, Length};
```

This avoids the repetition of multiple imports while avoiding the problems of wildcards. Note that this feature was introduced in Modelica Language Specification Version 3.3.

Modelica Standard Library

The power of packages in Modelica is the ability to take commonly needed types, models, functions, *etc.* and organize them into packages for reuse later by simply referencing them (rather than repeating them). But there is still a repetition problem if every user is expected to make their own packages of commonly needed definitions. For this reason, the Modelica Association maintain a package called the Modelica Standard Library. This library includes definitions that are generally useful to scientists and engineers.

In this section, we will provide an overview of the Modelica Standard Library so readers are aware of what they can expect to find and utilize from this library. This is not an exhaustive tour and because the Modelica Standard Library is constantly being updated and improved, it may not reflect the latest version of the library. But it covers the basics and hopefully provides readers with an understanding of how to locate useful definitions.

Constants

The Modelica Standard Library contains definitions of some common physical and machine constants. The library is small enough that we can include the source code for this package directly. The following represents the `Modelica.Constants` package from version 3.2.2 of the Modelica Standard Library (with a few cosmetic changes):

```
within Modelica;
package Constants
    "Library of mathematical constants and constants of nature (e.g., pi, eps, R, sigma)"

    import SI = Modelica.SIunits;
    import NonSI = Modelica.SIunits.Conversions.NonSIunits;

    // Mathematical constants
    final constant Real e=Modelica.Math.exp(1.0);
    final constant Real pi=2*Modelica.Math.asin(1.0); // 3.14159265358979;
    final constant Real D2R=pi/180 "Degree to Radian";
    final constant Real R2D=180/pi "Radian to Degree";
    final constant Real gamma=0.57721566490153286060
    "see http://en.wikipedia.org/wiki/Euler\_constant";

    // Machine dependent constants
    final constant Real eps=ModelicaServices.Machine.eps
        "Biggest number such that 1.0 + eps = 1.0";
    final constant Real small=ModelicaServices.Machine.small
        "Smallest number such that small and -small are representable on the machine";
    final constant Real inf=ModelicaServices.Machine.inf
```

```

"Biggest Real number such that inf and -inf are representable on the machine";
final constant Integer Integer_inf=ModelicaServices.Machine.Integer_inf
"Biggest Integer number such that Integer_inf and -Integer_inf are representable on the
machine";

// Constants of nature
// (name, value, description from http://physics.nist.gov/cuu/Constants/)
final constant SI.Velocity c=299792458 "Speed of light in vacuum";
final constant SI.Acceleration g_n=9.80665
"Standard acceleration of gravity on earth";
final constant Real G(final unit="m3/(kg.s2)") = 6.67408e-11
"Newtonian constant of gravitation";
final constant SI.FaradayConstant F = 9.648533289e4 "Faraday constant, C/mol";
final constant Real h(final unit="J.s") = 6.626070040e-34 "Planck constant";
final constant Real k(final unit="J/K") = 1.38064852e-23 "Boltzmann constant";
final constant Real R(final unit="J/(mol.K)") = 8.3144598 "Molar gas constant";
final constant Real sigma(final unit="W/(m2.K4)") = 5.670367e-8
"Stefan-Boltzmann constant";
final constant Real N_A(final unit="1/mol") = 6.022140857e23
"Avogadro constant";
final constant Real mue_0(final unit="N/A2") = 4*pi*1.e-7 "Magnetic constant";
final constant Real epsilon_0(final unit="F/m") = 1/(mue_0*c*c)
"Electric constant";
final constant NonSI.Temperature_degC T_zero=-273.15
"Absolute zero temperature";

```

Noteworthy definitions are those for `pi`, `e`, `g_n` and `eps`.

The first two, `pi` and `e`, are mathematical constants representing π and e , respectively. Having these constants available not only avoids having to provide your own numerical value for these (irrational) constants, but by using the version defined in the Modelica Standard Library, you get a value that has the highest possible precision.

The next constant, `g_n`, is a physical constant representing the gravitational constant on earth (for computing things like potential energy, *i.e.*, mgh).

Finally, `eps` is a machine constant that represents a “small number” for whatever computing platform is being used.

SI Units

As we discussed previously, the use of units not only makes your code easier to understand by associating concrete units with parameters and variables, it also allows unit consistency checking to be performed by the Modelica compiler. For this reason it is very useful to associate physical types with parameters and variables whenever possible.

The `Modelica.SIunits` package is very large and full of physical units that are rarely used. They are included for completeness in adhering to the ISO 31-1992 specification. The following are examples of how common physical units are defined in the `SIunits` package:

```

type Length = Real (final quantity="Length", final unit="m");
type Radius = Length(min=0);
...
type Velocity = Real (final quantity="Velocity", final unit="m/s");
type AngularVelocity = Real(final quantity="AngularVelocity",
                           final unit="rad/s");
...
type Mass = Real(quantity="Mass", final unit="kg", min=0);
type Density = Real(final quantity="Density", final unit="kg/m3",
                     displayUnit="g/cm3", min=0.0);
type MomentOfInertia = Real(final quantity="MomentOfInertia",
                            final unit="kg.m2");

```

```

...
type Pressure = Real(final quantity="Pressure", final unit="Pa",
                     displayUnit="bar");
...
type ThermodynamicTemperature = Real(
  final quantity="ThermodynamicTemperature",
  final unit="K",
  min = 0.0,
  start = 288.15,
  nominal = 300,
  displayUnit="degC")
"Absolute temperature (use type TemperatureDifference for relative temperatures)";
type Temperature = ThermodynamicTemperature;
type TemperatureDifference = Real(final quantity="ThermodynamicTemperature",
                                   final unit="K");

```

Models

The Modelica Standard Library includes many different domain specific libraries inside of it. This section provides an overview of each of these domains and discusses how models in each domain are organized.

Blocks

The Modelica Standard Library includes a collection of models for building causal, block-diagram models. The definitions for these models can be found in the `Modelica.Blocks` package. Examples of components that can be found in this library include:

- Input connectors (`Real`, `Integer` and `Boolean`)
- Output connectors (`Real`, `Integer` and `Boolean`)
- Gain block, summation blocks, product blocks
- Integration and differentiation blocks
- Deadband and hysteresis blocks
- Logic and relational operation blocks
- Mux and demux blocks

The `Blocks` package contains a wide variety of blocks for performing operations on signals. Such blocks are typically used for describing the function of control systems and strategies.

Electrical

The `Modelica.Electrical` package contains sub-packages specifically related to analog, digital and multi-phase electrical systems. It also includes a library of basic electrical machines as well. In this library you will find components like:

- Resistors, capacitors, inductors
- Voltage and current actuators
- Voltage and current sensors
- Transistor and other semiconductor related models
- Diodes and switches
- Logic gates
- Star and Delta connections (multi-phase)

- Synchronous and Asynchronous machines
- Motor models (DC, permanent magnet, *etc.*)
- Spice3 models

Mechanical

The `Modelica.Mechanics` library contains three main libraries.

Translational

The translational library contains component models used for modeling one-dimensional translational motion. This library contains components like:

- Springs, dampers and backlashes
- Masses
- Sensors and actuators
- Friction

Rotational

The rotational library contains component models used for modeling one-dimensional rotational motion. This library contains components like:

- Springs, dampers and backlashes
- Inertias
- Clutches and Brakes
- Gears
- Sensors and Actuators

MultiBody

The multibody library contains component models used for modeling three-dimensional mechanical systems. This library contains components like:

- Bodies (including associated inertia tensors and 3D CAD geometry)
- Joints (*e.g.*, prismatic, revolute, universal)
- Sensors and Actuators

Fluids and Media

There are two packages within the Modelica Standard Library associated with modeling fluid systems. The first is `Modelica.Media` which is a library of property models for various media like:

- Ideal gases (based on NASA Glenn coefficient data)
- Air (dry, reference, moist)
- Water (simple, salt, two-phase)
- Generic incompressible fluids
- R134a (tetrafluoroethane) refrigerant

These property models provide functions for computing fluid properties like enthalpy, density and specific heat ratios for a variety of pure fluids and mixtures.

In addition to these medium models, the Modelica Standard Library also includes the `Modelica.Fluid` library which is a library of components to describe fluid devices, for example:

- Volumes, tanks and junctions
- Pipes
- Pumps
- Valves
- Pressure losses
- Heat exchangers
- Sources and ambient conditions

Magnetics

The `Modelica.Magnetic` library contains two sub-packages. The first is the `FluxTubes` package which is used to construct models of lumped networks of magnetic components. This includes components to represent the magnetic characteristics of basic cylindrical and prismatic geometries as well as sensors and actuators. The other is the `FundamentalWave` library which is used to model electrical fields in rotating electrical machinery.

Thermal

The `Modelica.Thermal` package has two sub-packages:

HeatTransfer

The `HeatTransfer` is for modeling heat transfer in lumped solids. Models in this library can be used to build lumped thermal network models using components like:

- Lumped thermal capacitances
- Conduction
- Convection
- Radiation
- Ambient conditions
- Sensors

FluidHeatFlow

Normally, the `Modelica.Fluid` and `Modelica.Media` libraries should be used to model thermo-fluid systems because they are capable of handling a wide range of problems involving complex media and multiple phases. However, for a certain class of simpler problems, the `FluidHeatFlow` library can be used to build simple flow networks of thermo-fluid systems.

Utilities

The `Utilities` library provides support functionality to other libraries and model developers. It includes several sub-packages for dealing with non-mathematical aspects of modeling.

Files

The `Modelica.Utilities.Files` library provides functions for accessing and manipulating a computers file system. The following functions are provided by the `Files` package:

- `list` - List contents of a file or directory
- `copy` - Copy a file or directory
- `move` - Move a file or directory
- `remove` - Remove a file or directory
- `createDirectory` - Create a directory
- `exist` - Test whether a given file or directory already exists
- `fullPathName` - Determine the full path to a named file or directory
- `splitPathName` - Split a file name by path
- `temporaryFileName` - Return the name of a temporary file that does not already exist.
- `loadResource` - Convert a *Modelica URL* (page 160) into an absolute file system path (for use with functions that don't accept Modelica URLs).

Streams

The `Streams` package is used reading and writing data to and from the terminal or files. It includes the following functions:

- `print` - Writes data to either the terminal or a file.
- `readFile` - Reads data from a file and return a vector of strings representing the lines in the file.
- `readLine` - Reads a single line of text from a file.
- `countLines` - Returns the number of lines in a file.
- `error` - Used to print an error message.
- `close` - Closes a file.

Strings

The `Strings` package contains functions that operate on strings. The general capabilities of this library include:

- Find the length of a string
- Constructing and extracting strings
- Comparing strings
- Parsing and searching strings

System

The `System` package is used to interact with the underlying operating system. It includes the following functions:

- `getWorkingDirectory` - Get the current working directory.
- `setWorkingDirectory` - Set the current working directory.
- `getEnvironmentVariable` - Get the value of an environment variable.

- `setEnvironmentVariable` - Set the value of an environment variable.
- `command` - Pass a command to the operating system to execute.
- `exit` - Terminate execution.

2.2 Connectors

2.2.1 Introduction

Component-Oriented Modeling

Before diving into some examples, there is a bit of background required in order to understand what a **connector** in Modelica is, why it is used and the mathematical basis for their formulations. We'll cover these points first before proceeding.

So far, we've talked primarily about how to model behavior. What we've seen so far are models composed largely of textual equations. But from this point forward, we will be exploring how to create reusable component models. So instead of writing Ohm's law whenever we wish to model a resistor, we'll instead add an instance of a **resistor** component model to our system.

Until now, the models that we've shown have all been self contained. All the behavior of our entire system was captured in a single model and represented by textual equations. But this approach does not scale well. What we really want is the ability to create reusable component models.

But before we can dive into how to create these components, we need to first discuss how we will be connecting them together. The component models that we will be building during the remainder of the book are still represented by a `model` definition. What sets them apart from the models we have seen so far is that they will feature **connectors**.

A **connector** is a way for one model to exchange information with another model. As we'll see, there are different ways that we may wish to exchange this information and this chapter will focus on explaining the various semantics that can be used to describe connectors in Modelica.

Acausal Connections

In order to understand one specific class of connector semantics, it is first necessary to understand a bit more about acausal formulations of physical systems. An acausal approach to physical modeling identifies two distinct classes of variables.

The first class of variables we will discuss are “across” variables (also called *potential* or *effort* variables). Differences in the values of across variables across a component are what trigger components to react. Typical examples of across variables, that we will be discussing shortly, are temperature, voltage and pressure. Differences in these quantities typically lead to dynamic behavior in the system.

The second class of variables we will discuss are “through” variables (also called *flow* variables). Flow variables normally represent the flow of some conserved quantity like mass, momentum, energy, charge, etc. These flows are usually the result of some difference in the across variables across a component model. For example, current flowing through a resistor is in response to a voltage difference across the two sides of the resistor. As we will see in many of the examples to come, there are many different types of relationships between the through and across variables (Ohm's law being just one of many).

Sign Conventions

It is extremely important to know that Modelica follows a convention that a positive value for a through variable represents flow of the conserved quantity **into** a component. We'll repeat this convention several times later in the book (especially once we begin our discussion of how to build models of *Components* (page 183)).

The next section will define through and across variables for a number of basic engineering domains.

2.2.2 Examples

Simple Domains

In this section, we'll discuss relatively simple engineering domains. These are ones where a `connector` deals with only one through and one across variable. Conceptually, this means that only one conserved quantity is involved with that connector.

The following table covers four different engineering domains. In each domain, we see the choice of through and across variables that we will be using along with the SI units for those quantities.

Domain	Through Variable	Across Variable
Electrical	Current [A]	Voltage [V]
Thermal	Heat [W]	Temperature [K]
Translational	Force [N]	Position [m]
Rotational	Torque [N.m]	Angle [rad]

You may have seen a similar table before with slightly different choices. For example, you will sometimes see velocity (in m/s) chosen as the across variable for translational motion. The choices above are guided by two constraints.

The first constraint is that the through variable should be the time derivative of some conserved quantity. The reason for this constraint is that the through variable will be used to formulate generalized conservation equations in our system. As such, it is essential that the through variables be conserved quantities.

The second constraint is that the across variable should be the lowest order derivative to appear in any of our constitutive or empirical equations in the domain. So, for example, we chose position for translational motion because position is used in describing the behavior of a spring (*i.e.*, Hooke's law). If we had chosen velocity (the derivative of position with respect to time), then we would have been in the awkward situation of trying to describe the behavior of a spring in terms of velocities, not positions. An essential point here is that **differentiation is lossy**. If we know position, we can easily express velocity. But if we only know velocity, we cannot compute position without knowing an additional integration constant. This is why we want to work with across variables that have not been overly differentiated.

Now let's look at each domain individually.

Electrical

We can define a `connector` for the electrical domain as follows:

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

Here we see that the variable `v` in this connector represents the voltage and the variable `i` represents the current.

Note the presence of the `flow` qualifier in the declaration of `i`, the current. The `flow` qualifier is what tells the Modelica compiler that `i` is the through variable. Recall from our discussion on *Acausal Connections* (page 169) that the `flow` variable should be the time derivative of a conserved quantity. We can see that this connector follows that convention, since `Current` is the time derivative of charge (and charge is a conserved quantity).

Note the absence of any qualifier in the declaration of `v`, the voltage. A variable without any qualifier is assumed to be the `across` variable. You will find a more complete discussion about *Variables* (page 178) (including the various qualifiers that can be applied to them) later in this chapter.

The interested reader may wish to jump ahead to our discussion of *Electrical Components* (page 192) to see how we build on the connector definition to create electrical circuit components.

Thermal

A connector for modeling lumped heat transfer isn't much different from an electrical connector:

```
connector Thermal
  Modelica.SIunits.Temperature T;
  flow Modelica.SIunits.HeatFlowRate q;
end Thermal;
```

Instead of `Voltage` and `Current`, this connector includes `Temperature` and `HeatFlowRate`. While the names are different, the overall structure is essentially the same. The `connector` includes one through variable (`q`, indicated by the presence of the `flow` qualifier) and one across variable (`T`, indicated by the lack of any qualifier). Again, we see that the type of the variable with the `flow` qualifier, `HeatFlowRate`, is the time derivative of a conserved quantity, energy.

An example of how such a connector can be used to build components for lumped thermal network modeling can be found in the upcoming discussion on *Heat Transfer Components* (page 183). If you feel comfortable with this `connector` definition, feel free to jump ahead. It would still be a good idea to read the *Review* (page 177) section of the *Connectors* (page 169) chapter at some point.

Translational

To model translational motion, we would define a connector as follows:

```
connector Translational
  Modelica.SIunits.Position x;
  flow Modelica.SIunits.Force f;
end Translational;
```

Again, we see the same basic structure as before. The connector contains one through variable, `f`, and one across variable `x`. Note that, although this is a one-dimensional mechanical connector, the physical types are specific to translational motion and distinct from the physical types used from the *Rotational* (page 171) case to be presented next.

An important fact about mechanical connectors that is often overlooked is that the `flow` variable is the time derivative of a conserved quantity. For example, in the case of translational motion the `flow` variable, `f`, is a force and force is the time derivative of (linear) momentum and momentum is a conserved quantity.

Rotational

For systems whose motion is constrained to be rotational, the following Modelica `connector` definition should be used:

```
connector Rotational
  Modelica.SIunits.Angle phi;
  flow Modelica.SIunits.Torque tau;
end Rotational;
```

Here we see that the across variable is `phi` (representing the angular displacement) and the through variable is `torque`. As with all previous examples, the `flow` variable is the time derivative of a conserved quantity. In this case, that conserved quantity is angular momentum.

SimpleDomains

All the connectors defined in this section are grouped, for future reference, into a single package:

```
within ModelicaByExample.Connectors;
package SimpleDomains "Examples of connectors for simple domains"
    connector Electrical
        Modelica.SIunits.Voltage v;
        flow Modelica.SIunits.Current i;
    end Electrical;

    connector Thermal
        Modelica.SIunits.Temperature T;
        flow Modelica.SIunits.HeatFlowRate q;
    end Thermal;

    connector Translational
        Modelica.SIunits.Position x;
        flow Modelica.SIunits.Force f;
    end Translational;

    connector Rotational
        Modelica.SIunits.Angle phi;
        flow Modelica.SIunits.Torque tau;
    end Rotational;
end SimpleDomains;
```

Fluid Connectors

One area that Modelica has been widely used in is the modeling of various types of fluid systems. We saw in the previous section how to create connectors for various *Simple Domains* (page 170). But what makes Modelica so compelling for fluid systems is the ability to create more complex connectors involving multiple conserved quantities simultaneously. Such connectors are essential for modeling fluid systems, where a single connector might involve the flow of mass, momentum, energy and/or species. Such cases require the definition of rich connectors types.

We'll start this section with a discussion of basic connectors very similar to the ones used for *Simple Domains* (page 170). But we will conclude with a connector that is fundamentally different from the previous examples because it describes a connector that involves the conservation of both mass and energy.

Incompressible Fluids

Modeling of incompressible fluids is very useful in a number of engineering applications, most notably hydraulically actuated systems. We'll start by presenting a simple connector that can be used to model incompressible systems, but with some important caveats.

Consider the following connector definition:

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

As we saw in our discussion of *Simple Domains* (page 170), we see the familiar pattern of an across variable and a through variable. In this case the across variable is *p* (the pressure) and the through variable is *q* (the volumetric flow rate). But this connector is different from all the previous examples because the *flow* variable is **not** the time derivative of a conserved quantity, since volume is not a conserved quantity.

This connector works **as long as the fluid being modeled is incompressible**. To understand why, consider the following equation:

$$q_1 + q_2 + q_3 + q_4 = 0$$

where q_1 , q_2 , q_3 and q_4 represent volumetric flow terms (*i.e.*, each has units of m^3/s). In general, this equation does not qualify as a conservation equation because volume is (again, in general) not conserved. However, if we know that each of these flows is an incompressible fluid, then we can multiply the entire equation by the density of that incompressible fluid, *i.e.*,

$$\rho q_1 + \rho q_2 + \rho q_3 + \rho q_4 = 0$$

Now each of these terms has units of kg/s which is a conservation equation because mass is a conserved quantity. However, **if you use this connector definition with a fluid that has any significant degree of compressibility, you will get the wrong answer**.

Such a connector definition is useful for relatively simple incompressible fluid flow networks because it can frequently describe the behavior of the system without having to specify (or know) the density of the working fluid. However, this kind of approach is inherently limiting so it should only be used in situations where it solves more problems than it creates.

Compressible Fluids

While the previous connector definition should be strictly used for *Incompressible Fluids* (page 172), the following connector is more general:

```
connector GenericFluid
  Modelica.SIunits.Pressure p;
  flow Modelica.SIunits.MassFlowRate m_flow;
end GenericFluid;
```

This connector can be used for **both** incompressible or compressible fluids. This is because it doesn't make any inherent assumptions about the compressibility of the fluid. Note that the across variable, p , is still pressure, but the through variable, m_flow , is a mass flow rate. As such, the through variable conforms to the convention that a through variable should be the time derivative of a conserved quantity (in this case, mass). So there are no implicit assumptions in this connector, which is why it can be used to model fluid flow of both compressible and incompressible fluids.

This connector isn't really fundamentally different from the connectors associated with *Simple Domains* (page 170), but it appears in this section because it is a stepping stone to the next example.

Thermo-Fluid Modeling

So far in this section, we've presented a connector for incompressible fluid systems, *Incompressible*, and a more general connector, *GenericFluid*. But in both of these cases, the only conserved quantity considered was mass. Nowhere in these previous connectors is there any reference to or allowance for modeling the **temperature** of the fluid.

There are many applications where the temperature of the working fluid is critical. In some cases, the temperature changes the density of the working fluid. In other cases, the temperature may trigger a phase change (*e.g.*, from liquid to gas). Temperature can also affect other critical properties of the fluid like viscosity, which have a significant impact on the performance of, for example, lubrication systems. As a result, those previous connector definitions would be inadequate for modeling systems where the temperature of the working fluid had any significant impact on the system behavior.

To predict the temperature of a working fluid, it is necessary to track the energy that flows with the fluid as it flows through a network. To do this, the connector definition must be augmented to include energy, alongside mass, as a conserved quantity that flows through the connector. The following connector definition does just that:

```
connector ThermoFluid
  Modelica.SIunits.Pressure p;
  flow Modelica.SIunits.MassFlowRate m_flow;
  Modelica.SIunits.Temperature T;
  flow Modelica.SIunits.HeatFlowRate q;
end ThermoFluid;
```

Note that this connector includes **two** variables that have the **flow** qualifier, **m_flow** and **q**. These represent the flow of mass and energy, respectively. Each of these is paired with an across variable. One of those across variables is the pressure, **p**, just as we saw in the previous connectors in this section. The other across variable, **T**, is the temperature of the working fluid.

Block Connectors

So far, all the connectors that have been presented have been acausal. This means that they consist of pairs of through and across variables. Such connectors are the basis for modeling physical interactions (ones where conserved quantities are exchanged between components). But there are other types of interactions and other modeling formalisms that can be represented in Modelica.

Block connectors are used to model the flow of information through a system. Here we are not concerned with physical quantities, like current, which might flow in one direction for a while and then reverse direction. Here we will consider how to model signals where some components produce information and others consume it (and then, in turn, produce other information). In this kind of situation, we frequently refer to such signals as “input signals” and “output signals”.

To model such interactions, we can use connector definitions like these:

```
within ModelicaByExample.Connectors;
package BlockConnectors "Connectors for block diagrams"
  connector RealInput = input Real;
  connector RealOutput = output Real;
  connector IntegerInput = input Integer;
  connector IntegerOutput = output Integer;
  connector BooleanInput = input Boolean;
  connector BooleanOutput = output Boolean;
end BlockConnectors;
```

It should be pretty obvious from these definitions that, for example, the **BooleanInput** connector is used to identify a **Boolean** input signal and **RealOutput** identifies a **Real** output signal.

We'll see how to utilize these connector definitions later when we discuss *Block Diagram Components* (page 236).

Graphical Connectors

Text vs. Graphics

Until now, we've discussed Modelica as a purely textual language. But the reality is that Modelica models are primarily built **graphically**. From this point on, graphics will play a greater and greater role in the way we visualize complex Modelica models.

This visualization starts with the connectors. The textual component of the Modelica models will always be present. Variables and equations will always be declared as we've shown already, in textual form. But as we will repeatedly see as we move forward, the **annotation** feature in Modelica can be used to associate a graphical appearance with many different entities in Modelica.

The first kind of graphical association we will introduce will be the graphics associated with a **connector**. More specifically, we'll introduce how to associate graphics with the definition of a **connector**. These graphics will then be used whenever the connector is instantiated in a diagram (something we'll discuss in greater detail when we discuss *Components* (page 183)).

Icon Annotations

When we associate an annotation with a definition, we place it in the definition, but do not associate it with any declarations or other entities in the definition. Instead, it is just another element of the definition. To demonstrate this, consider the following electrical pin connector definitions:

```
within ModelicaByExample.Connectors;
package Graphics
  connector PositivePin
    Modelica.SIunits.Voltage v;
    flow Modelica.SIunits.Current i;
    annotation (
      Icon(graphics={
        Ellipse(
          extent={{-100,100},{100,-100}},
          lineColor={0,0,255},
          fillColor={85,170,255},
          fillPattern=FillPattern.Solid),
        Rectangle(
          extent={{-10,58},{10,-62}},
          fillColor={0,128,255},
          fillPattern=FillPattern.Solid,
          pattern=LinePattern.None),
        Rectangle(
          extent={{-60,10},{60,-10}},
          fillColor={0,128,255},
          fillPattern=FillPattern.Solid,
          pattern=LinePattern.None,
          lineColor={0,0,0}),
        Text(
          extent={{-100,-100},{100,-140}},
          lineColor={0,0,255},
          fillColor={85,170,255},
          fillPattern=FillPattern.Solid,
          textString="%name")),
      Documentation(info="

This connector is used to represent the &quot;positive&quot; pins on electrical components. This does not imply that the voltage at this pin needs to be positive or even greater than voltages on <a href=\"modelica://ModelicaByExample.Connectors.Graphics.NegativePin\">&quot;negative&quot; pins</a>. It is simply a convention used to distinguish different connectors on components (particularly those with only two pins).</p>

"));
    end PositivePin;

  connector NegativePin
    Modelica.SIunits.Voltage v;
    flow Modelica.SIunits.Current i;
    annotation (
      Icon(graphics={
        Ellipse(
          extent={{-100,100},{100,-100}},
          lineColor={0,0,255},
          fillColor={85,170,255},
          fillPattern=FillPattern.Solid),
        Rectangle(
          extent={{-60,10},{60,-10}},
          fillColor={0,128,255},
          fillPattern=FillPattern.Solid,
          pattern=LinePattern.None,
          lineColor={0,0,0}),
        Text(

```

```

extent={{-100,-100},{100,-140}},
lineColor={0,0,255},
fillColor={85,170,255},
fillPattern=FillPattern.Solid,
textString="%name"}),
Documentation(info=<html>
<p>This pin and
<a href=\"modelica://ModelicaByExample.Connectors.Graphics.PositivePin\">
its counterpart</a> are documented in
<a href=\"modelica://ModelicaByExample.Connectors.Graphics.PositivePin\">
PositivePin</a>.</p>
</html>"));
end NegativePin;
end Graphics;

```

Note the length of each of these definitions. The length is due almost entirely to the annotations present in these definitions. Apart from the annotations, the `PositivePin` and `NegativePin` definitions are identical to the `Electrical` connector definition presented in the discussion on *Simple Domains* (page 170).

The reason we've chosen to define two electrical pin connectors is so that they can be made graphically distinct. An instance of the `PositivePin` connector looks like:



while an instance of `NegativePin` looks like:



Let's look in greater detail at the `Icon` annotation in the `PositivePin` definition:

```

Icon(graphics=
  Ellipse(

```

```

extent={{-100,100},{100,-100}},
lineColor={0,0,255},
fillColor={85,170,255},
fillPattern=FillPattern.Solid),
Rectangle(
extent={{-10,58},{10,-62}},
fillColor={0,128,255},
fillPattern=FillPattern.Solid,
pattern=LinePattern.None),
Rectangle(
extent={{-60,10},{60,-10}},
fillColor={0,128,255},
fillPattern=FillPattern.Solid,
pattern=LinePattern.None,
lineColor={0,0,0}),
Text(
extent={{-100,-100},{100,-140}},
lineColor={0,0,255},
fillColor={85,170,255},
fillPattern=FillPattern.Solid,
textString="%name")),

```

We will be discussing *Graphical Annotations* (page 178) shortly. But let's take a quick look at what is going on in these definitions. We can see that the `Icon` annotation contains another variable `graphics`. The `graphics` variable is assigned a vector of graphical elements. We see that this vector graphical elements includes an `Ellipse` (used to render the circle in the icon), two `Rectangle` elements (used to render the “+” sign) and a `Text` element. Note the `textString` field in the `Text` element contains `%name`. There are a number of substitution patterns that can appear in a graphical annotation. This particular one will be filled in with the instance name whenever a variable is declared with the type `PositivePin`. So, for example, the following declarations:

```
PositivePin p;
```

would be rendered graphically with `%name` replaced with `p`. In this way, the textual names assigned to connectors in a diagram always match the name of the underlying connector declarations in a model.

Graphical Annotations (page 178) will be reviewed in detail later in this chapter and we will see many more uses of them as we transition from strictly textual representations of models to implementations that incorporate graphical rendering as well.

2.2.3 Review

Connector Definitions

Syntax

As we have seen several times already, Modelica definitions share a considerable amount of syntactic similarity. This is just as true with `connector` definitions.

The general syntax for a connector definition is:

```
connector ConnectorName "Description of the connector"
  // Declarations for connector variables
end ConnectorName;
```

Unlike a `model` or `function`, a `connector` is not allowed to include any behavior. So there can never be an `equation` or `algorithm` section present in a `connector`.

Variables

Causal Variables

In our previous discussion of [Block Connectors](#) (page 174), we showed that variables within a Modelica **connector** definition can have a causality associated with them. If the signal is expected to be computed externally, then the variable should have the **input** qualifier associated with it. If, on the other hand, a signal is expected to be computed internally (and then transmitted to other components), it should have the **output** qualifier associated with it.

Acausal Variables

In our discussion of [Simple Domains](#) (page 170) and [Fluid Connectors](#) (page 172), we saw numerous examples of **connector** definitions that included through and across variables. These variables always occurred in pairs with the through variable being prefixed by the **flow** qualifier while the across variable had no qualifier associated with it.

As we will see in the coming chapters, such connector definitions are very convenient when modeling physical systems because they enable the Modelica compiler to automatically generate conservation equations for networks of components. Furthermore, they allow quantities like, mass, momentum, energy, charge, species and so on to flow **bi-directionally** through a network.

Parameters

A variable in a **connector** definition can also have the **parameter** qualifier associated with it. This qualifier means the same thing that it meant when we first discussed [Parameters](#) (page 28), *i.e.*, the value of the variable cannot change during a simulation. A **parameter** variable is frequently used in connector definitions to indicate the size of an array contained in the connector.

Final Remarks

It should be noted that a **connector** definition can mix causal, acausal and parameter variables all in the same connector. In fact, a variable in a connector **can itself be a connector** as well. This richness of expressiveness in Modelica allows users to model a range of different types of interactions and choose, on a variable by variable basis, the semantics that make the most sense for each potential interaction.

Graphical Annotations

Although this section appears in the chapter on [Connectors](#) (page 169), this topic applies to graphical annotations associated with model definitions in general. So the information presented here will be a useful reference with respect to many aspects of Modelica.

Graphical Layers

When describing the appearance of a Modelica entity, there are two different representations to choose from. One is called the “icon” representation and the other is called the “diagram” representation. In Modelica, the icon representation is used when viewing something from the “outside”. Generally, the icon includes some distinctive visual representation along with additional information about that entity added via [Substitutions](#) (page 182) (which we will be covering shortly).

The “diagram” representation, on the other hand, is used to represent the view of a component from the “inside”. The diagram representation is generally used to include additional graphical documentation about the Modelica component that would be too detailed for the “icon” view.

A definition's graphical appearance in an “icon” layer is specified by the `Icon` annotation (briefly touched on in our *Graphical Connectors* (page 174) discussion earlier). Not surprisingly, a definition's graphical appearance in the “diagram” layer is specified by the `Diagram` annotation. Both of these are annotations that appear directly in the definition and are not associated with existing elements like declarations or `extends` clauses.

Generally speaking, most definitions include an “icon” representation, but only a few bother to include a “diagram” representation. However, it turns out that despite being rendered in different contexts, the specification of graphical appearance is identical between them.

Use of `Icon` in examples

For the remainder of the book, we will show examples of graphical annotations using the `Icon` annotation. These examples could equally be applied to a `Diagram` annotation, but since the `Icon` annotation is more common, all further examples regarding graphical annotations will appear exclusively in the context of the `Icon` annotation.

Common Graphical Definitions

The following definitions will be referenced throughout this section:

```
type DrawingUnit = Real(final unit="mm");
type Point = DrawingUnit[2] "{x, y}";
type Extent = Point[2]
    "Defines a rectangular area {{x1, y1}, {x2, y2}}";
type Color = Integer[3](min=0, max=255) "RGB representation";
constant Color Black = zeros(3);
type LinePattern = enumeration(None, Solid, Dash, Dot, DashDot, DashDotDot);
type FillPattern = enumeration(None, Solid, Horizontal, Vertical,
    Cross, Forward, Backward,
    CrossDiag, HorizontalCylinder,
    VerticalCylinder, Sphere);
type BorderPattern = enumeration(None, Raised, Sunken, Engraved);
type Smooth = enumeration(None, Bezier);
type Arrow = enumeration(None, Open, Filled, Half);
type TextStyle = enumeration(Bold, Italic, UnderLine);
type TextAlignement = enumeration(Left, Center, Right);

record FilledShape "Style attributes for filled shapes"
    Color lineColor = Black "Color of border line";
    Color fillColor = Black "Interior fill color";
    LinePattern pattern = LinePattern.Solid "Border line pattern";
    FillPattern fillPattern = FillPattern.None "Interior fill pattern";
    DrawingUnit lineThickness = 0.25 "Line thickness";
end FilledShape;
```

In addition, many of the annotations we will be discussing include a set of common elements represented by the following record definition:

```
partial record GraphicItem
    Boolean visible = true;
    Point origin = {0, 0};
    Real rotation(quantity="angle", unit="deg")=0;
end GraphicItem;
```

For annotations representing graphical elements, we will extend from this `GraphicItem` to make the presence of these common elements explicitly clear.

Icon and Diagram Annotations

The elements that should appear in the icon layer of a model are described by the following data:

```
record Icon "Representation of the icon layer"
  CoordinateSystem coordinateSystem(extent = {{-100, -100}, {100, 100}});
  GraphicItem[:] graphics;
end Icon;
```

where the coordinate system data is defined as:

```
record CoordinateSystem
  Extent extent;
  Boolean preserveAspectRatio=true;
  Real initialScale = 0.1;
  DrawingUnit grid[2];
end CoordinateSystem;
```

In other words, the `Icon` annotation includes information about the coordinate system contained in the definition of `coordinateSystem` and it also includes a list of graphical items stored in `graphics`. The definition of the `Diagram` annotation is identical:

```
record Diagram "Representation of the diagram layer"
  CoordinateSystem coordinateSystem(extent = {{-100, -100}, {100, 100}});
  GraphicItem[:] graphics;
end Diagram;
```

Graphical Items

There are a number of different graphical items that are defined in the specification that can be used in constructing the `graphics` vector associated with either the `Icon` or `Diagram` annotations. Their definitions are presented here for reference.

Line

```
record Line
  extends GraphicItem;
  Point points[:];
  Color color = Black;
  LinePattern pattern = LinePattern.Solid;
  DrawingUnit thickness = 0.25;
  Arrow arrow[2] = {Arrow.None, Arrow.None} "{start arrow, end arrow}";
  DrawingUnit arrowSize=3;
  Smooth smooth = Smooth.None "Spline";
end Line;
```

Polygon

```
record Polygon
  extends GraphicItem;
  extends FilledShape;
  Point points[:];
  Smooth smooth = Smooth.None "Spline outline";
end Polygon;
```

Rectangle

```
record Rectangle
  extends GraphicItem;
  extends FilledShape;
  BorderPattern borderPattern = BorderPattern.None;
  Extent extent;
  DrawingUnit radius = 0 "Corner radius";
end Rectangle;
```

Ellipse

```
record Ellipse
  extends GraphicItem;
  extends FilledShape;
  Extent extent;
  Real startAngle(quantity="angle", unit="deg")=0;
  Real endAngle(quantity="angle", unit="deg")=360;
end Ellipse;
```

Text

```
record Text
  extends GraphicItem;
  extends FilledShape;
  Extent extent;
  String textString;
  Real fontSize = 0 "unit pt";
  String fontName;
  TextStyle textStyle[:];
  Color textColor=lineColor;
  TextAlignment horizontalAlignment = TextAlignment.Center;
end Text;
```

Bitmap

```
record Bitmap
  extends GraphicItem;
  Extent extent;
  String fileName "Name of bitmap file";
  String imageSource "Base64 representation of bitmap";
end Bitmap;
```

Inheriting Graphical Annotations

When one model definition inherits from another, the graphical annotations are inherited by default. However, this behavior can be controlled by annotating the `extends` clause with the following data (for the icon and diagram layers, respectively):

```
record IconMap
  Extent extent = {{0, 0}, {0, 0}};
  Boolean primitivesVisible = true;
end IconMap;
```

```
record DiagramMap
  Extent extent = {{0, 0}, {0, 0}};
  Boolean primitivesVisible = true;
end DiagramMap;
```

In both cases, the `extent` data allows the location of the inherited graphical elements to be adjusted. Setting `primitivesVisible` to `false` will suppress the rendering of inherited graphical elements.

Substitutions

When working with the `Text` (page 181) annotation, the `textString` field can contain substitution patterns. The following substitution patterns are supported:

- `%name` - This pattern will be replaced by the name of the instance of the given definition.
- `%class` - This pattern will be replaced by the name of this definition.
- `%<name>` where `<name>` is a parameter name - This pattern will be replaced by the `value` of the named parameter.
- `%%` - This pattern will be replaced by `%`.

Putting It All Together

Having discussed all these aspects of graphical annotations, let us review the icon definitions presented during our discussion of *Graphical Connectors* (page 174).

```
Icon(graphics={
  Ellipse(
    extent={{-100,100},{100,-100}},
    lineColor={0,0,255},
    fillColor={85,170,255},
    fillPattern=FillPattern.Solid),
  Rectangle(
    extent={{-10,58},{10,-62}},
    fillColor={0,128,255},
    fillPattern=FillPattern.Solid,
    pattern=LinePattern.None),
  Rectangle(
    extent={{-60,10},{60,-10}},
    fillColor={0,128,255},
    fillPattern=FillPattern.Solid,
    pattern=LinePattern.None,
    lineColor={0,0,0}),
  Text(
    extent={{-100,-100},{100,-140}},
    lineColor={0,0,255},
    fillColor={85,170,255},
    fillPattern=FillPattern.Solid,
    textString="%name"))},
```

Here we see the `annotation` associated with the `PositivePin` definition is a model annotation. Furthermore, we can see the `Icon` data associated with this annotation includes a list of graphical items. The first graphical item is an `Ellipse` (page 181) annotation. That is followed by two `Rectangle` (page 181) annotations and finally a `Text` (page 181) (which also makes use of the *Substitutions* (page 182) we discussed earlier).

Note how the data being presented in this `annotation` lines up with the data described in the record definitions we discussed earlier.

2.3 Components

When most people think of Modelica, they think of the component-oriented approach that it enables. As such, how to build such component models is probably the single most important topic to cover in this book.

Until now, we've focused primarily on how to describe the mathematical behavior (both continuous and discrete). Now it is time to understand how that behavior can be wrapped up into reusable component models. The fact that these component models are reusable means that, once written and tested, the same code can be used over and over. This kind of reuse saves development time, avoids errors and simplifies maintenance.

2.3.1 Examples

Heat Transfer Components

We'll start our discussion of component models by building some component models in the heat transfer domain. These models will allow us to recreate the models we saw *previously* (page 6), but this time using component models to represent each of the various effects. Investing the time to make component models will then allow us to easily combine the underlying physical behavior to create models for a wide variety of thermal systems.

Thermal Connectors

In our previous discussion on *Simple Domains* (page 170) we described how a thermal connector could be described. For the component models in this section, we will utilize the thermal connector models from the Modelica Standard Library. These connectors are defined as follows:

```
within Modelica.Thermal.HeatTransfer;
package Interfaces "Connectors and partial models"
    partial connector HeatPort "Thermal port for 1-dim. heat transfer"
        Modelica.SIunits.Temperature T "Port temperature";
        flow Modelica.SIunits.HeatFlowRate Q_flow
            "Heat flow rate (positive if flowing from outside into the component)";
    end HeatPort;

    connector HeatPort_a "Thermal port for 1-dim. heat transfer (filled rectangular icon)"
        extends HeatPort;
        annotation(...,
            Icon(coordinateSystem(preserveAspectRatio=true,
                extent={{-100,-100},{100,100}}),
            graphics={Rectangle(
                extent={{-100,100},{100,-100}},
                lineColor={191,0,0},
                fillColor={191,0,0},
                fillPattern=FillPattern.Solid)}));
    end HeatPort_a;

    connector HeatPort_b "Thermal port for 1-dim. heat transfer (unfilled rectangular icon)"
        extends HeatPort;
        annotation(...,
            Icon(coordinateSystem(preserveAspectRatio=true,
                extent={{-100,-100},{100,100}}),
            graphics={Rectangle(
                extent={{-100,100},{100,-100}},
                lineColor={191,0,0},
```

```

        fillColor={255,255,255},
        fillPattern=FillPattern.Solid)}));
end HeatPort_b;
end Interfaces;
```

Careful inspection of these connector definitions shows that `HeatPort_a` and `HeatPort_b` are identical in terms of their content to the `HeatPort` model. The only difference is that `HeatPort_a` and `HeatPort_b` have distinguishing graphical icons.

The component models presented in the remainder of this section will utilize these connector definitions.

Component Models

When building component models, the goal is to create component models that implement (only) one physical effect (*e.g.*, capacitance, convection). By implementing component models in this way, we will see that they can then be combined in any infinite number of different configurations without the need to add any more equations. This kind of reuse of equations makes the model developer more productive and avoids opportunities to introduce errors.

Thermal Capacitance

Our first component model will be a model of lumped thermal capacitance with uniform temperature distribution. The equation we wish to associate with this component model is:

$$C\dot{T} = Q_{flow}$$

The Modelica model (with the `Icon` annotation removed) representing this equation is quite simple:

```

within ModelicaByExample.Components.HeatTransfer;
model ThermalCapacitance "A model of thermal capacitance"
  parameter Modelica.SIunits.HeatCapacity C "Thermal capacitance";
  parameter Modelica.SIunits.Temperature T0 "Initial temperature";
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a node
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
initial equation
  node.T = T0;
equation
  C*der(node.T) = node.Q_flow;
end ThermalCapacitance;
```

where `C` is the thermal capacitance and `T0` is the initial temperature.

Note the presence of the `node` connector in this model. This is how the `ThermalCapacitance` component model interacts with the “outside world”. We will use the temperature at the `node`, `node.T` to represent the temperature of the thermal capacitance. The `flow` variable, `node.Q_flow`, represents the flow of heat `into` the thermal capacitance. We can see this when looking at the equation for the thermal capacitance:

$$C*der(node.T) = node.Q_flow;$$

Note that when `node.Q_flow` is positive, the temperature of the thermal capacitance, `node.T`, will increase. This confirms that we have followed the Modelica convention that `flow` variables on a connector represent a flow of the conserved quantity, heat in this case, into the component (a more thorough discussion of [Accounting](#) (page 254) will be presented shortly).

Using this model alone, we can already build a simple “system” model as follows:

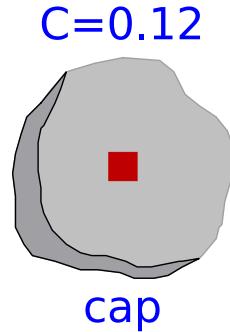
```

within ModelicaByExample.Components.HeatTransfer.Examples;
model Adiabatic "A model without any heat transfer"
  ThermalCapacitance cap(C=0.12, T0(displayUnit="K") = 363.15)
```

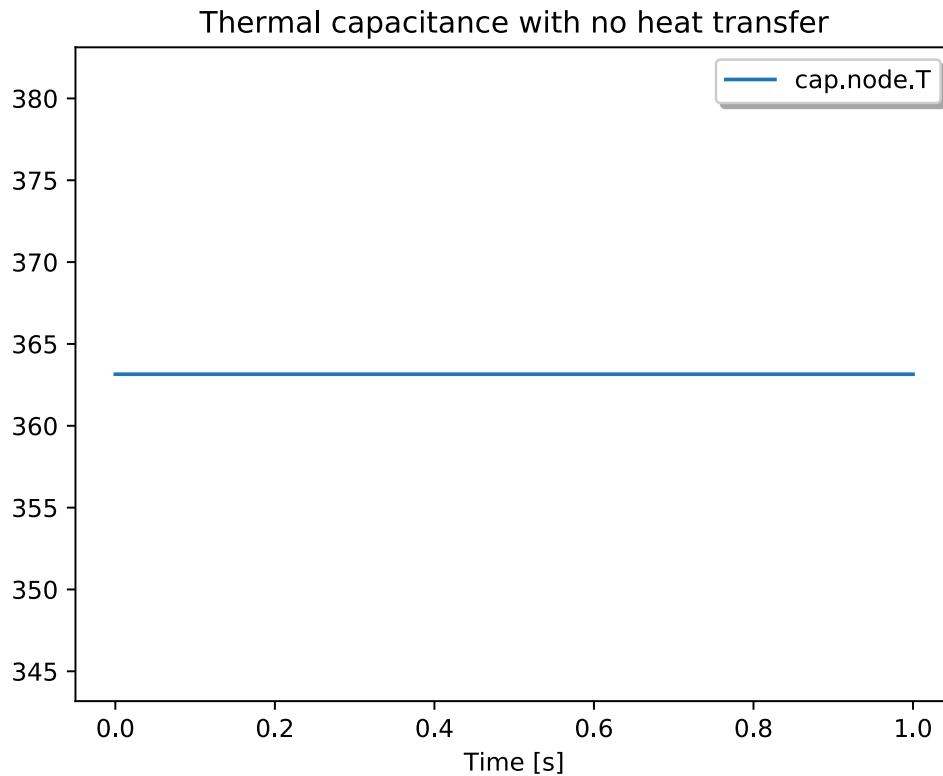
```
"Thermal capacitance component"
annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));
end Adiabatic;
```

This model contains only the thermal capacitance element (as indicated by the declaration of the variable `cap` of type `ThermalCapacitance`) and no other heat transfer elements (*e.g.*, conduction, convection, radiation). Ignore the `Placement` annotation for the moment, we'll provide a complete explanation in a later section on [Component Model Annotations](#) (page 262).

Using the graphical annotations in the model (some of which were left out of the previous listing) it can be rendered as:



Since no heat enters or leaves the thermal capacitance component, `cap`, the temperature of the capacitance remains constant as shown in the following plot:



ConvectionToAmbient

To quickly add some heat transfer, we could define another component model to represent heat transfer to some ambient temperature. Such a model could be represented in Modelica (again, without the `Icon` annotation) as follows:

```
within ModelicaByExample.Components.HeatTransfer;
model ConvectionToAmbient "An overly specialized model of convection"
  parameter Modelica.SIunits.CoefficientOfHeatTransfer h;
  parameter Modelica.SIunits.Area A;
  parameter Modelica.SIunits.Temperature T_amb "Ambient temperature";
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
equation
  port_a.Q_flow = h*A*(port_a.T-T_amb) "Heat transfer equation";
end ConvectionToAmbient;
```

This model includes parameters for the heat transfer coefficient, `h`, the surface area, `A` and the ambient temperature, `T_amb`. This model is attached to other heat transfer elements through the connector `port_a`.

Again, we must pay close attention to the sign convention. Recall from our previous discussion of *Thermal Capacitance* (page 184) that Modelica follows a sign convention that a positive value for a `flow` variable represents flow into the component. In particular, let's take a close look at the equation in the `ConvectionToAmbient` model:

```
port_a.Q_flow = h*A*(port_a.T-T_amb) "Heat transfer equation";
```

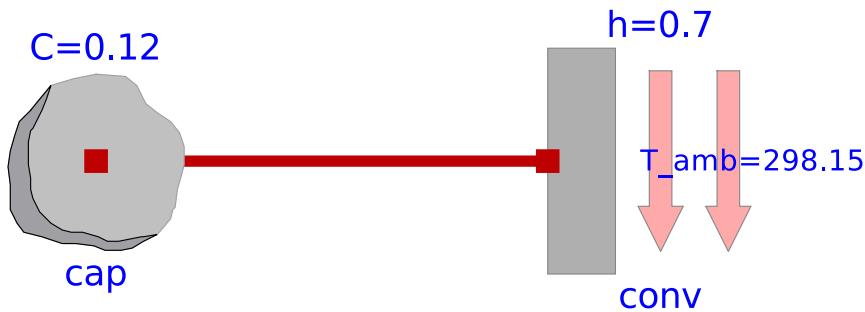
Note that when `port_a.T` is greater than `T_amb`, the sign of `port_a.Q_flow` is positive. That means heat is flowing **into** this component. In other words, when `port_a.T` is greater than `T_amb`, this component will **take heat away** from `port_a` (and, conversely, when `T_amb` is greater than `port_a.T`, it will **inject heat into** `port_a`).

Having such a component model available enables us to combine it with the `ThermalCapacitance` model and simulate a system just like we modeled in *some of our earlier heat transfer examples* (page 6) using the following Modelica code:

```
within ModelicaByExample.Components.HeatTransfer.Examples;
model CoolingToAmbient "A model using convection to an ambient condition"

  ThermalCapacitance cap(C=0.12, T0(displayUnit="K") = 363.15)
    "Thermal capacitance component"
    annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));
  ConvectionToAmbient conv(h=0.7, A=1.0, T_amb=298.15)
    "Convection to an ambient temperature"
    annotation (Placement(transformation(extent={{20,-10},{40,10}})));
equation
  connect(cap.node, conv.port_a) annotation (Line(
    points={{-20,0},{20,0}},
    color={191,0,0},
    smooth=Smooth.None));
end CoolingToAmbient;
```

In this model, we see two components have been declared, `cap` and `conv`. The parameters for each of these components are also specified when they are declared. The following is a schematic for the `CoolingToAmbient` model:



But what is really remarkable about this model is the equation section:

```
equation
  connect(cap.node, conv.port_a) annotation (Line(
    points={{-20,0},{20,0}},
    color={191,0,0},
    smooth=Smooth.None));
```

This statement introduces one of the most important features in Modelica. Note that statement appears within an `equation` section. While the `connect` operator looks like a function, it is much more than that. It represents the equations that should be generated to model the interaction between the two specified connectors, `cap.node` and `conv.port_a`.

In this context, a connection does two important things. The first thing it does is to generate an equation that equates the “across” variables on either connector. In this case, that means the following equation:

```
cap.node.T = conv.port_a.T "Equating across variables";
```

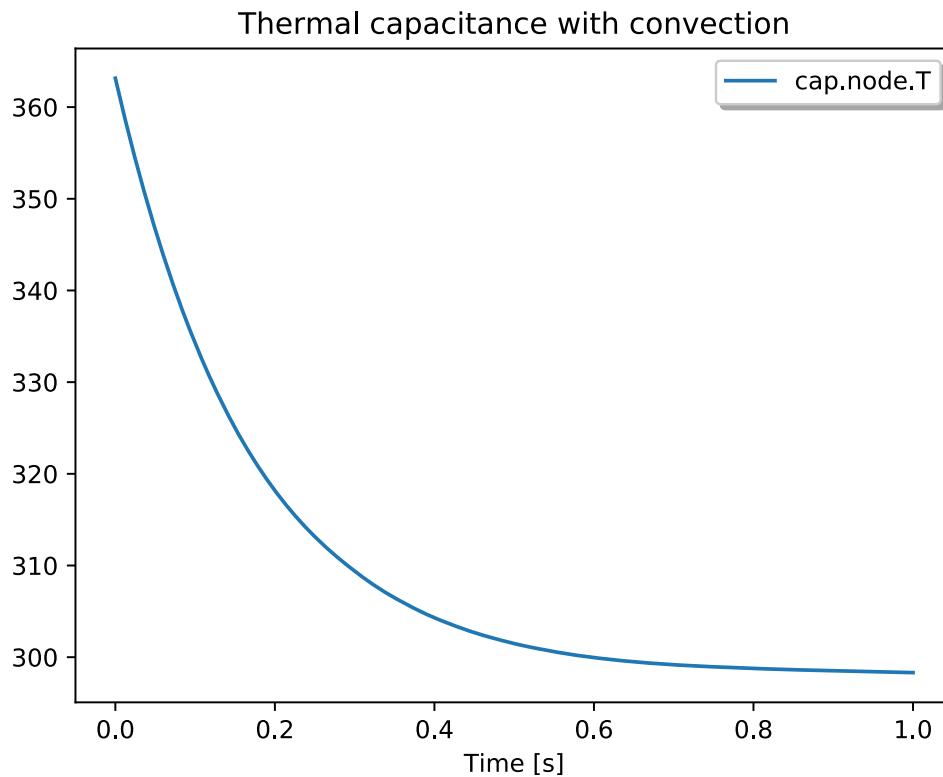
In addition, a connection generates an equation for all the through variables as well. The equation that is generated is a conservation equation. You can think of this conservation equation as a generalization of Kirchoff’s current law to any conserved quantity. Basically, it represents the fact that the connection itself has no “storage” ability and that whatever amount of the conserved quantity, in this case heat, that flows out of one component must go into the other(s). So in this case, the `connect` statement will generate the following equation with respect to the `flow` variables:

```
cap.node.Q_flow + conv.port_a.Q_flow = 0 "Sum of heat flows must be zero";
```

Note the sign convention here. All the `flow` variables are summed. We will examine more complex cases shortly where multiple components are interacting. But in this simple case, with only two components, we see clearly that if one value for `Q_flow` is positive, the other must be negative. In other words, if heat is flowing out of one component, it must be flowing into another. These conservation equations ensure that we have a proper accounting of conserved quantities throughout our network and that no amount of the conserved quantity gets “lost”.

A very simple way to summarize the behavior of a connection, in the context of a thermal problem, is to **think of a connection as a perfectly conducting element with no thermal capacitance**.

If we simulate the `CoolingToAmbient` model above, we get the following temperature trajectory:



Digging Deeper

There is one slight issue with the `CoolingToAmbient` model. We mentioned earlier that when building component models it is best to isolate each individual physical effect to its own component. But we've actually lumped two different effects into one component. As we will see in a moment, this limits the reusability of the component models. But first, let's refactor the code to separate these effects out and then we'll revisit the system level model using these new components.

Convection

The first new component is a `Convection` model. In this case, we won't make any assumptions about the temperature at either end. Instead, we'll only assume that each is connected to something with an appropriate thermal connector. The result is a model like this one:

```
within ModelicaByExample.Components.HeatTransfer;
model Convection "Modeling convection between port_a and port_b"
  parameter Modelica.SIunits.CoefficientOfHeatTransfer h;
  parameter Modelica.SIunits.Area A;
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_b port_b
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
equation
  port_a.Q_flow + port_b.Q_flow = 0 "Conservation of energy";
  port_a.Q_flow = h*A*(port_a.T-port_b.T) "Heat transfer equation";
end Convection;
```

This model contains two equations. The first equation:

```
port_a.Q_flow + port_b.Q_flow = 0 "Conservation of energy";
```

represents the fact that this component does not store heat. The equation enforces the constraint that whatever heat flows in from one connector must flow out from the other (which is exactly the same behavior we saw from the `connect` statement earlier in this section). The next equation:

```
port_a.Q_flow = h*A*(port_a.T-port_b.T) "Heat transfer equation";
```

captures the heat transfer relationship for convection by expressing the relationship between the flow of heat through this component and the temperatures on either end.

Number of equations

All our previous models had one connector and one equation. The `Convection` model has two connectors. As a result, it has two equations. A simple rule of thumb is that you need as many equations as connectors. But keep in mind that this rule of thumb assumes that you are using connectors with only one through variable on them and no “internal variables” in your model (*e.g.*, `protected` variables). The upcoming section on [Component Models](#) (page 252) will provide a more comprehensive discussion on determining how many equations a component requires. Specifically, it will provide guidance on how to build so-called [Balanced Components](#) (page 257).

AmbientCondition

Now that we have the convection model, we need something to represent the ambient conditions. We need something like a thermal capacitance model, but one that maintains a constant temperature. Imagine if we took the `ThermalCapacitance` model and gave a very large value for its capacitance, `C`. Then we’d have something that changed temperature very slowly. But what we want is something that doesn’t change temperature at all, as if it had a `C` value that was infinitely large.

This kind of model comes up frequently and it is commonly called an “infinite reservoir” model. Typically, such a model is characterized by the fact that no matter how much of the conserved quantity (heat in this case) flows into or out of the component, the across variable remains constant. In an electrical context, such a model would represent electrical ground. In a mechanical context, it would represent a mechanical ground (something that didn’t change position, regardless of how much force was applied).

We will represent our ambient conditions using the `AmbientConditions` model:

```
within ModelicaByExample.Components.HeatTransfer;
model AmbientCondition
  "Model of an \"infinite reservoir\" at some ambient temperature"
  parameter Modelica.SIunits.Temperature T_amb "Ambient temperature";
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a node annotation (
    Placement(transformation(extent={{-10,-10},{10,10}}), iconTransformation(
      extent={{-10,-10},{10,10}})));
  equation
    node.T = T_amb;
end AmbientCondition;
```

Since we are talking about the heat transfer domain, this model is an infinite reservoir for heat and no matter how much heat flows into or out of this component, its temperature remains the same.

Flexibility

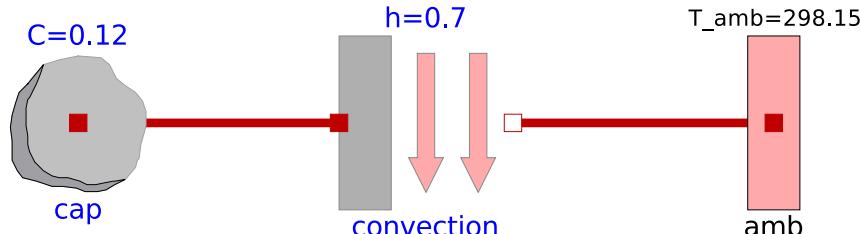
Using these new `Convection` and an `AmbientCondition` models, we can reconstruct our simple system level heat transfer model using the following:

```

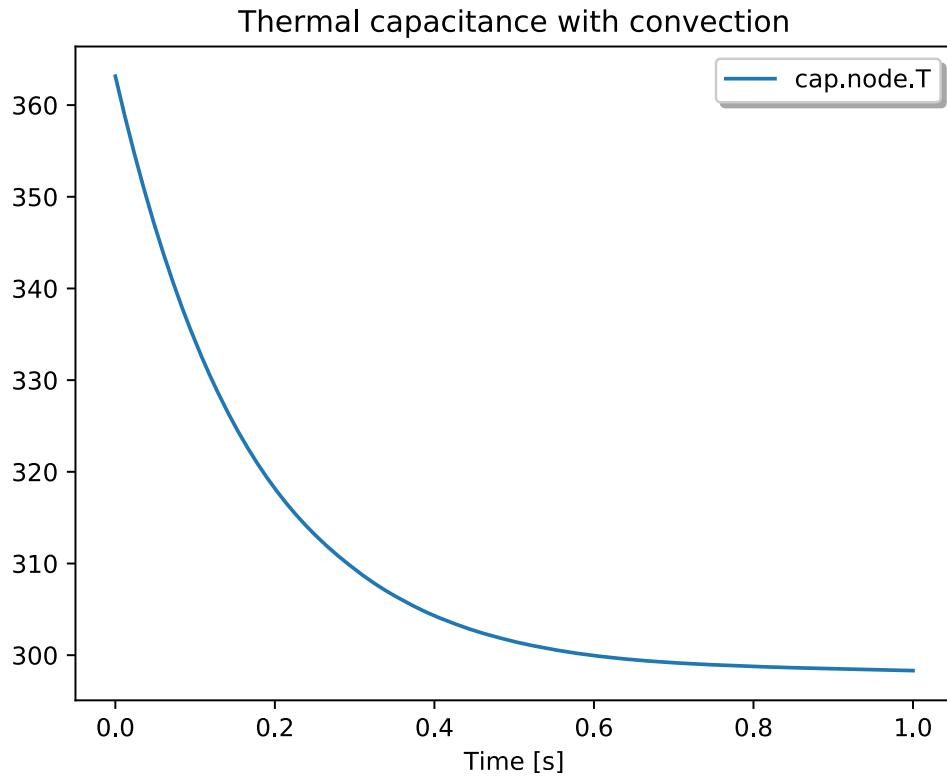
within ModelicaByExample.Components.HeatTransfer.Examples;
model Cooling "A model using generic convection to ambient conditions"
  ThermalCapacitance cap(C=0.12, T0(displayUnit="K") = 363.15)
    "Thermal capacitance component"
    annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));
  Convection convection(h=0.7, A=1.0)
    annotation (Placement(transformation(extent={{10,-10},{30,10}}));
  AmbientCondition amb(T_amb(displayUnit="K") = 298.15)
    annotation (Placement(transformation(extent={{50,-10},{70,10}}));
equation
  connect(convection.port_a, cap.node) annotation (Line(
    points={{10,0},{-20,0}},
    color={191,0,0},
    smooth=Smooth.None));
  connect(amb.node, convection.port_b) annotation (Line(
    points={{60,0},{30,0}},
    color={191,0,0},
    smooth=Smooth.None));
end Cooling;

```

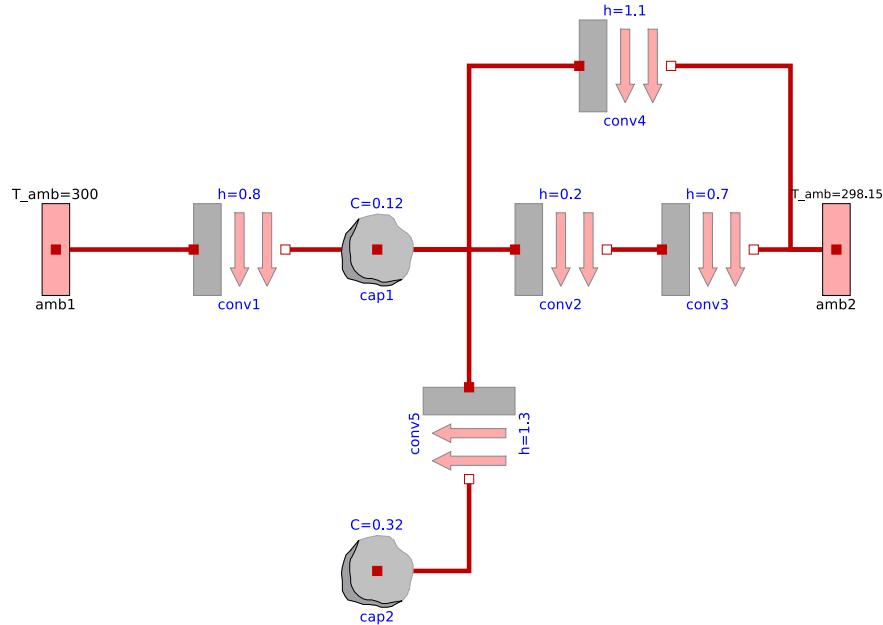
When rendered, the model looks like this:

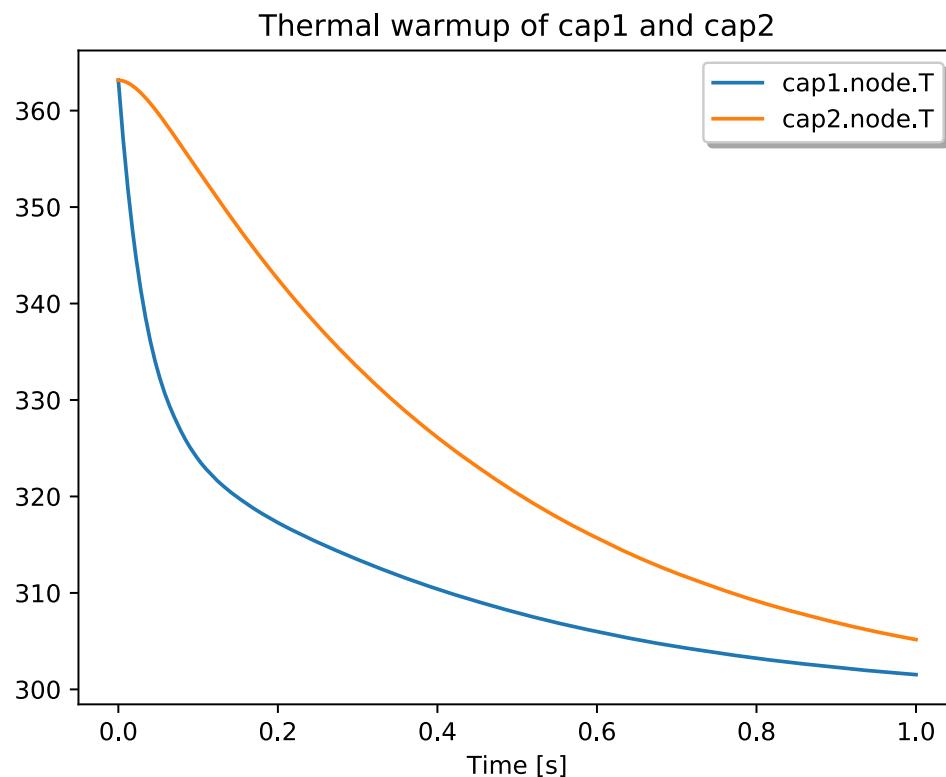


This may not seem like much of an improvement. Although we went to the trouble to break up the `ConvectionToAmbient` model into individual `Convection` and `AmbientTemperature` models, we still end up with the same fundamental behavior, *i.e.*,



The big benefit of breaking down `ConvectionToAmbient` into `Convection` and `AmbientTemperature` models is the ability to recombine them in different ways. The following schematic is just one example of how the handful of fundamental components we've constructed so far can be rearranged to form an entirely new (and more complex) model:





In fact, with these components we can now make **arbitrarily complex** networks of components and still never have to worry about formulating the associated equations that describe their dynamics. Everything that is required to do this has already been captured in our component models. This allows us to focus on the process of creating and designing our system and leave the tedious, time-consuming and error prone work of manipulating equations behind.

Electrical Components

The previous section discussed how to create component models in the heat transfer domain. Now let's turn our attention to how to construct some basic electrical components and then use them to simulate the kinds of systems we saw in our previous *electrical example* (page 10).

In this section we will implement the basic electrical component models **twice**. The first time through, we will implement each component without any regard to the others. But the second time through, we'll see how we can use the inheritance mechanism in Modelica to make our lives a little easier.

But in both cases, we'll use the same connector definitions. In our discussion of *Simple Domains* (page 170), we saw how to construct an electrical connector. As with the previous section on heat transfer, the examples in this section will rely on the connector definitions from the Modelica Standard Library. Those connector definitions look like this:

```

connector PositivePin "Positive pin of an electric component"
  Modelica.SIunits.Voltage v "Potential at the pin";
  flow Modelica.SIunits.Current i "Current flowing into the pin";
end PositivePin;

connector NegativePin "Negative pin of an electric component"
  Modelica.SIunits.Voltage v "Potential at the pin";

```

```

flow Modelica.SIunits.Current i "Current flowing into the pin";
end NegativePin;

```

Basic Component Models

Given these `connector` definitions, it is relatively straightforward to construct a resistor model. The goal of a resistor model is to encapsulate the relationship between the voltage across the resistor and the current through the resistor using Ohm's law. The following model represents how one might expect such a resistor model to look:

```

within ModelicaByExample.Components.Electrical.VerboseApproach;
model Resistor "A resistor model"
  parameter Modelica.SIunits.Resistance R;
  Modelica.Electrical.Analog.Interfaces.PositivePin p
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Electrical.Analog.Interfaces.NegativePin n
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Voltage v = p.v-n.v;
equation
  p.i + n.i = 0 "Conservation of charge";
  v = p.i*R "Ohm's law";
end Resistor;

```

In the same way, we might create inductor and capacitor models as follows:

```

within ModelicaByExample.Components.Electrical.VerboseApproach;
model Inductor "An inductor model"
  parameter Modelica.SIunits.Inductance L;
  Modelica.Electrical.Analog.Interfaces.PositivePin p
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Electrical.Analog.Interfaces.NegativePin n
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Voltage v = p.v-n.v;
equation
  p.i + n.i = 0 "Conservation of charge";
  L*der(p.i) = p.v;
end Inductor;

```

```

within ModelicaByExample.Components.Electrical.VerboseApproach;
model Capacitor "A capacitor model"
  parameter Modelica.SIunits.Capacitance C;
  Modelica.Electrical.Analog.Interfaces.PositivePin p
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Electrical.Analog.Interfaces.NegativePin n
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Voltage v = p.v-n.v;
equation
  p.i + n.i = 0 "Conservation of charge";
  C*der(v) = p.i;
end Capacitor;

```

The important thing to notice about these models is the amount of common code shared between them. In software development, this kind of redundancy is frowned upon. In fact, a common software maxim is “Redundancy is the root of all evil”. The reason this redundancy is a problem is partly because you are doing the same work multiple times, but also because this code needs to be *maintained* as well. When you repeat code and then find a mistake in that code, you have to fix it everywhere.

The DRY Principle

This issue of redundancy is an important one. So let's revisit building models of resistors, inductors and capacitors with the goal of reducing the amount of repeated code. In software, there is something called the *DRY principle* where DRY stands for “Don’t Repeat Yourself”. So our next step is to make our resistor, capacitor and inductor models DRY.

The key to eliminating redundant code is to identify all the common code between these models and create a **partial** model that we can inherit from. We highlighted the common lines previously. Now we can capture them in their own model as follows:

```
within ModelicaByExample.Components.Electrical.DryApproach;
partial model TwoPin "Common elements of two pin electrical components"
  Modelica.Electrical.Analog.Interfaces.PositivePin p
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Electrical.Analog.Interfaces.NegativePin n
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));

  Modelica.SIunits.Voltage v = p.v-n.v;
  Modelica.SIunits.Current i = p.i;
equation
  p.i + n.i = 0 "Conservation of charge";
end TwoPin;
```

In summary, we've extracted the declarations for **p**, **n** and **v** from the previous models into this model. We've also included a variable, **i**, to represent the current flowing from pin **p** to pin **n**. Finally, the conservation of charge equation is also included.

Creating such a model then allows us to create a much more succinct resistor model as follows:

```
within ModelicaByExample.Components.Electrical.DryApproach;
model Resistor "A DRY resistor model"
  parameter Modelica.SIunits.Resistance R;
  extends TwoPin;
equation
  v = i*R "Ohm's law";
end Resistor;
```

There are several things to notice about this **Resistor** model. The first is how much shorter it is. This is because we inherit the electrical pins, the conservation of charge equation and the variables **v** and **i** from the **TwoPin** model. Another thing to notice is that, by leverage the definitions of **v** and **i**, Ohm's law looks just like it would if you saw it in a text book.

We can give the same treatment to our inductor and capacitor models:

```
within ModelicaByExample.Components.Electrical.DryApproach;
model Capacitor "A DRY capacitor model"
  parameter Modelica.SIunits.Capacitance C;
  extends TwoPin;
equation
  C*der(v) = i;
end Capacitor;
```

```
within ModelicaByExample.Components.Electrical.DryApproach;
model Inductor "A DRY inductor model"
  parameter Modelica.SIunits.Inductance L;
  extends TwoPin;
equation
  L*der(i) = v;
end Inductor;
```

Again, we see that the models are much more succinct. Ultimately, factoring out common code in this way means that the component models are easier to write and easier to maintain.

Circuit Model

So far, we've only built component models. In order to create a circuit model we first need to define a few more component models. Specifically, we need to create a step voltage source model:

```
within ModelicaByExample.Components.Electrical.DryApproach;
model StepVoltage "A DRY step voltage source"
  parameter Modelica.SIunits.Voltage V0;
  parameter Modelica.SIunits.Voltage Vf;
  parameter Modelica.SIunits.Time stepTime;
  extends TwoPin;
equation
  v = if time>=stepTime then Vf else V0;
end StepVoltage;
```

Note how the `StepVoltage` model also leverages the `TwoPin` model. We will also need a ground model which we model as follows:

```
within ModelicaByExample.Components.Electrical.DryApproach;
model Ground "Electrical ground"
  Modelica.Electrical.Analog.Interfaces.PositivePin ground "Ground pin"
    annotation (Placement(transformation(extent={{-10,70},{10,90}})));
  equation
    ground.v = 0;
end Ground;
```

The `Ground` model has only one pin so it cannot inherit from `TwoPin`. Recall how we described the *AmbientCondition* (page 189) model from our discussion on *Heat Transfer Components* (page 183) an infinite reservoir. The `Ground` model serves a very similar purpose. No matter how much current flows in to or out of an electrical ground, the voltage remains zero.

Having defined all these components, we can now create a circuit model as follows:

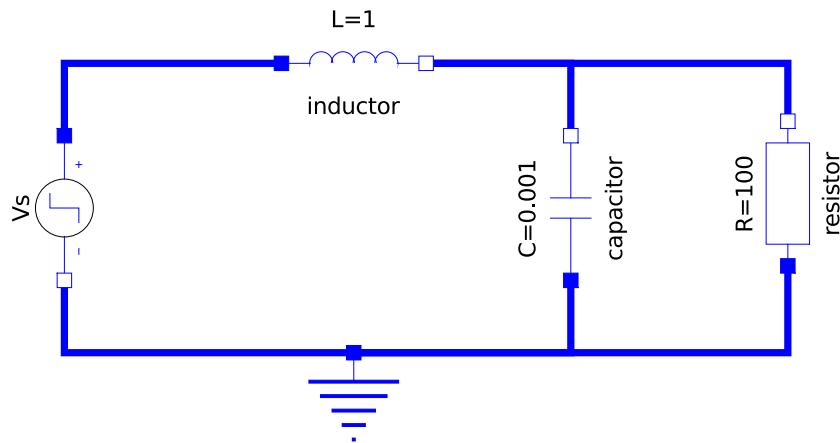
```
within ModelicaByExample.Components.Electrical.Examples;
model SwitchedRLC "Recreation of the switched RLC circuit"
  DryApproach.StepVoltage Vs(V0=0, Vf=24, stepTime=0.5)
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=270,
      origin={-40,0})));
  DryApproach.Inductor inductor(L=1, i(fixed=true, start=0))
    annotation (Placement(transformation(extent={{-10,10},{10,30}})));
  DryApproach.Capacitor capacitor(C=1e-3, v(fixed=true, start=0))
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=90,
      origin={30,0})));
  DryApproach.Resistor resistor(R=100) annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90,
    origin={60,2})));
  DryApproach.Ground ground
    annotation (Placement(transformation(extent={{-10,-38},{10,-18}})));
equation
  connect(inductor.n, resistor.n) annotation (Line(
    points={{10,20},{60,20},{60,12}},
    color={0,0,255},
    smooth=Smooth.None));
  connect(capacitor.n, inductor.n) annotation (Line(
    points={{30,10},{30,20},{10,20}},
    color={0,0,255},
    smooth=Smooth.None));
  connect(inductor.p, Vs.p) annotation (Line(
```

```

points={{-10,20},{-40,20},{-40,10}},
color={0,0,255},
smooth=Smooth.None));
connect(capacitor.p, ground.ground) annotation (Line(
points={{30,-10},{30,-20},{0,-20}},
color={0,0,255},
smooth=Smooth.None));
connect(resistor.p, ground.ground) annotation (Line(
points={{60,-8},{60,-20},{0,-20}},
color={0,0,255},
smooth=Smooth.None));
connect(Vs.n, ground.ground) annotation (Line(
points={{-40,-10},{-40,-20},{0,-20}},
color={0,0,255},
smooth=Smooth.None));
end SwitchedRLC;

```

The schematic diagram for this model is rendered as:



Basic Rotational Components

In this section, we'll show how to create basic components for modeling one-dimensional rotational systems. We'll build on our discussion of rotational connectors and show how they can be used to define the interfaces for basic rotational components. Finally, we'll show how those rotational components can then be assembled into a system model that replicates the behavior of the equation-based version of the same system presented in the first chapter.

Component Models

In the first chapter, we considered [A Mechanical Example](#) (page 13) modeled strictly in terms of equations (*i.e.*, without component models). In this section, we will start by recreating that system model using components. To do this, we first have to define models for the fundamental components we require. These will consist of models for an inertia, a spring, a damper and a mechanical ground.

As in [the previous section](#) (page 192), we will first define the component models using verbose formulations and then we will revisit these definitions and attempt to factor out common code to avoid repetition across component models.

Coordinate Systems

The method for creating these models will be very similar to how we previously created component models in the heat transfer and electrical domains. But before we start building component models, we should first discuss one of the complexities associated with mechanical systems, coordinate systems.

In the mechanical domain, the conserved quantity we will be tracking is momentum. What makes momentum different from the conserved quantities we've already covered, heat and charge, is that it is directional. Since we are only concerning ourselves with the one dimensional case here, the consequence of this directionality is that momentum is a signed quantity (*i.e.*, it can be positive or negative).

Consider a rotating mass with a moment of inertia, J . If the angular position of the inertia is represented by φ , then the angular velocity of the inertia, ω , is defined as:

$$\omega = \dot{\varphi}$$

Obviously, a positive value of ω will result in an increase in φ over time. Furthermore, the angular acceleration of the inertia, α , is defined as:

$$\alpha = \dot{\omega}$$

As with the angular velocity, we can see that a positive value for α will result in an increase in the angular velocity. Finally, the angular momentum of this rotating inertia is defined as $J\omega$ and we know from Euler's laws of motion that (assuming J is a constant):

$$J \frac{d\omega}{dt} = \tau$$

From this relationship, it is clear that a positive value for the torque, τ , will increase the amount of momentum stored in the mass.

The point of presenting all these relationships is to underscore the sign conventions associated with φ , ω , α and τ . They are all tied to the fundamental definition of what a positive angular position is. **Whatever direction causes φ to increase is the same direction that corresponds to a positive velocity, a positive acceleration and a positive torque.**

Rotational Inertia

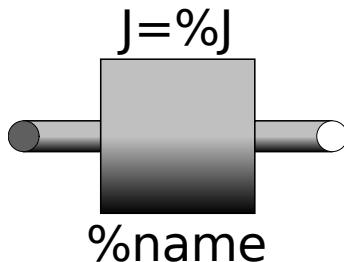
With this discussion about sign conventions and coordinate systems out of the way, we can start creating our component models. We'll start with the inertia model:

```
within ModelicaByExample.Components.Rotational.VerboseApproach;
model Inertia "Rotational inertia without inheritance"
  parameter Modelica.SIunits.Inertia J;
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Angle phi_rel;
  Modelica.SIunits.Torque tau;
  Modelica.SIunits.AngularVelocity w;
equation
  // Variables
  phi_rel = flange_a.phi-flange_b.phi;
  tau = flange_a.tau;
  w = der(flange_a.phi);
  // Conservation of angular momentum (includes storage)
  J*der(w) = flange_a.tau + flange_b.tau;
```

```
// Kinematic constraint (inertia is rigid)
phi_rel = 0;
annotation ( Icon(graphics={
```

extent={{-100,90},{100,50}},

The **Inertia** model includes two “flanges”, one on either end. The significance of these flanges is made clearer from the icon of the **Inertia** model:



In other words, the **Inertia** model includes a flange on either end. You can think of this model as a shaft with connectors on either end.

Now, the fundamental equation we wish to capture in the **Inertia** model is:

```
J*der(w) = flange_a.tau + flange_b.tau;
```

This is basically expressing the fact that the increase in momentum stored within the inertia is equal to the sum of the torques applied to the inertia. Recall, from our previous discussions on [Acausal Connections](#) (page 169), that the sign convention for flow variables on connectors (`flange_a.tau` and `flange_b.tau` in this case) is that a positive value represents a flow of the conserved quantity into the component model. The fact that `flange_a` and `flange_b` have the same sign convention means that the **Inertia** model is symmetric (*i.e.*, it can be flipped over and it doesn't change the behavior).

However, this equation refers to the internal variables `w` (which represents ω) and `tau` so we need to include declarations and definitions for those variables as well.

Spring Model

Next, let us consider the definition of a spring model:

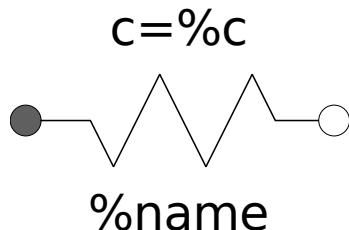
```
within ModelicaByExample.Components.Rotational.VerboseApproach;
model Spring "Rotational spring without inheritance"
  parameter Modelica.SIunits.RotationalSpringConstant c;
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Angle phi_rel;
  Modelica.SIunits.Torque tau;
equation
  // Variables
  phi_rel = flange_a.phi-flange_b.phi;
  tau = flange_a.tau;

  // No storage of angular momentum
  flange_a.tau + flange_b.tau = 0;

  // Hooke's law
```

```
tau = c*phi_rel;
end Spring;
```

The icon for our spring model is rendered as:



Like the `Inertia` model, the `Spring` model has two connectors, one on each end. It also defines many of the same internal variables. Ultimately, the behavior of the spring comes down to this equation:

```
tau = c*phi_rel;
```

In fact, apart from this equation and the parameter `c`, much of the content in the `Spring` model is the same as the content in the `Inertia` model.

Damper Model

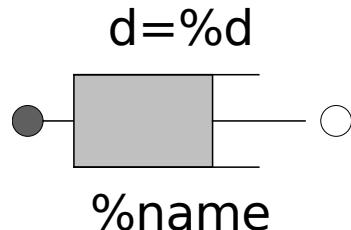
The `Damper` model is also very similar to the `Spring` model. Again, the main differences are the parameter (`d` in this case) and one equation:

```
within ModelicaByExample.Components.Rotational.VerboseApproach;
model Damper "Rotational damper without inheritance"
  parameter Modelica.SIunits.RotationalDampingConstant d;
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Angle phi_rel;
  Modelica.SIunits.Torque tau;
equation
  // Variables
  phi_rel = flange_a.phi-flange_b.phi;
  tau = flange_a.tau;

  // No storage of angular momentum
  flange_a.tau + flange_b.tau = 0;

  // Damping relationship
  tau = d*der(phi_rel);
end Damper;
```

The icon for the `Damper` model is rendered as:



DRY Component Models

We already have models for an inertia, a spring and a damper. The only model we are missing in order to complete our *dual spring mass damper system* (page 13) is a model of mechanical ground. But before we complete that model, let's take a moment to revisit the models we've already created with the goal of factoring out the large amount of code shared between these models. As in *the previous section* (page 192), let's take the time to apply the DRY (Don't Repeat Yourself) principle.

Common Code

It is worth noting that because the Modelica Standard Library has an extensive collection of rotational components, it was forced to deal with this issue of redundant code almost from the start. However, we will not be using the `partial` models from the Modelica Standard Library here simply because they are designed to deal with many other cases that are not relevant in this context. As a result, its complexity (although necessary) makes it unsuitable pedagogically.

But one thing we will preserve from the Modelica Standard Library is the need for multiple `partial` models. This need arises from the fact that, unlike in our previous discussion of *Electrical Components* (page 192), our rotational component models share different amounts of code with each other.

What is common to all of our models is the existence of two flange connectors, `flange_a` and `flange_b`. However, while the `Inertia` model has the capacity to store angular momentum, the `Spring` and `Damper` models do not. As a result, the conservation equations are different among these components.

Let's start with the elements that are common to all three models. These are represented by the following `TwoFlange` model:

```
within ModelicaByExample.Components.Rotational.Interfaces;
partial model TwoFlange
  "Definition of a partial rotational component with two flanges"

  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.SIunits.Angle phi_rel;
equation
  phi_rel = flange_a.phi-flange_b.phi;
end TwoFlange;
```

In addition to defining the two flanges, `flange_a` and `flange_b`, this model also defines the relative angle between these flanges, *i.e.*, `phi_rel`. Of course, this model is also marked as `partial` since it is missing any description of the component's behavior.

We could have all three models inherit from this model. But then we would still have some redundant equations between our `Spring` and `Damper` model. So we will instead create a slightly more specialized version of the `TwoFlange` model to represent compliant models that do not store momentum:

```

within ModelicaByExample.Components.Rotational.Interfaces;
partial model Compliant "A compliant rotational component"
  extends ModelicaByExample.Components.Rotational.Interfaces.TwoFlange;
protected
  Modelica.SIunits.Torque tau;
equation
  tau = flange_a.tau;
  flange_a.tau + flange_b.tau = 0
    "Conservation of angular momentum (no storage)";
end Compliant;

```

The `Compliant` model adds on additional internal variable (to represent the torque that passes through the component from `flange_a` to `flange_b`) and an equation indicating that no angular momentum is stored by the component.

With these base classes defined, let us quickly revisit the various component model definitions to see how much more succinct they can be made by using inheritance.

Rotational Inertia

Leveraging the `TwoFlanges` model, our `Inertia` model can be simplified to:

```

within ModelicaByExample.Components.Rotational.Components;
model Inertia "A rotational inertia model"
  parameter Modelica.SIunits.Inertia J;
  extends ModelicaByExample.Components.Rotational.Interfaces.TwoFlange;
  Modelica.SIunits.AngularVelocity w "Angular Velocity"
    annotation(Dialog(group="Initialization", showStartAttribute=true));
  Modelica.SIunits.Angle phi "Angle"
    annotation(Dialog(group="Initialization", showStartAttribute=true));
equation
  phi = flange_a.phi;
  w = der(flang_a.phi) "velocity of inertia";
  phi_rel = 0 "inertia is rigid";
  J*der(w) = flange_a.tau + flange_b.tau
    "Conservation of angular momentum with storage";
end Inertia;

```

Spring Model

In the same way, inheriting from the `Compliant` model our `Spring` model can be much more compactly represented as:

```

within ModelicaByExample.Components.Rotational.Components;
model Spring "A rotational spring component"
  parameter Modelica.SIunits.RotationalSpringConstant c;
  extends ModelicaByExample.Components.Rotational.Interfaces.Compliant;
equation
  tau = c*phi_rel "Hooke's Law";
end Spring;

```

Damper Model

Likewise, the `Damper` model is similarly simplified:

```

within ModelicaByExample.Components.Rotational.Components;
model Damper "A rotational damper"

```

```

parameter Modelica.SIunits.RotationalDampingConstant d;
extends ModelicaByExample.Components.Rotational.Interfaces.Compliant;
equation
  tau = d*der(phi_rel) "Damping relationship";
end Damper;

```

Mechanical Ground

Finally, we can complete the one model remaining in order to complete our *dual spring mass damper system* (page 13). The mechanical ground model is defined as follows:

```

within ModelicaByExample.Components.Rotational.Components;
model Ground "Mechanical ground"
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    annotation (Placement(transformation(extent={{-10,50},{10,70}})));
  equation
    flange_a.phi = 0;
end Ground;

```

Dual Spring Mass Damper System

Finally, we have all the parts we need in order to reconstruct the example we saw in the first chapter. Using the various components already defined in this section, the Modelica code for our component based system model looks like this:

```

within ModelicaByExample.Components.Rotational.Examples;
model SMD
  Components.Ground ground annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90,
    origin={76,0})));
  Components.Damper damper2(d=1)
    annotation (Placement(transformation(extent={{30,10},{50,30}})));
  Components.Spring spring2(c=5)
    annotation (Placement(transformation(extent={{28,-30},{48,-10}})));
  Components.Inertia inertia2(
    J=1,
    phi(fixed=true, start=1),
    w(fixed=true, start=0))
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  Components.Damper damper1(d=0.2)
    annotation (Placement(transformation(extent={{-50,10},{-30,30}})));
  Components.Spring spring1(c=11)
    annotation (Placement(transformation(extent={{-50,-30},{-30,-10}})));
  Components.Inertia inertia1(
    J=0.4,
    phi(fixed=true, start=0),
    w(fixed=true, start=0))
    annotation (Placement(transformation(extent={{-90,-10},{-70,10}})));
equation
  connect(ground.flange_a, damper2.flange_b) annotation (Line(
    points={{70,0},{66,0},{66,0},{60,0},{60,20},{50,20}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(ground.flange_a, spring2.flange_b) annotation (Line(
    points={{70,0},{60,0},{60,-20},{48,-20}},
    color={0,0,0},
    smooth=Smooth.None));

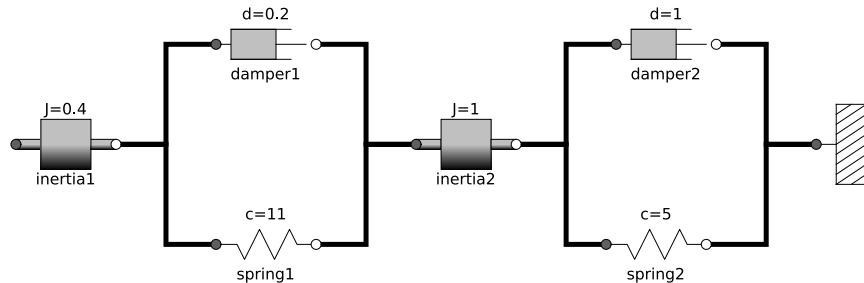
```

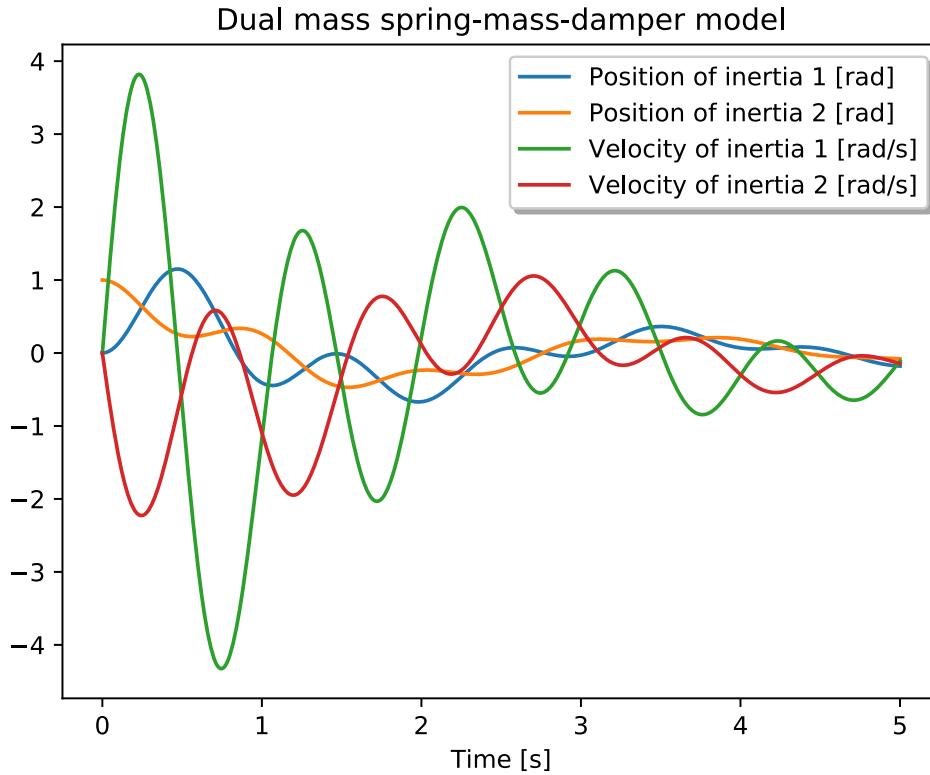
```

connect(damper2.flange_a, inertia2.flange_b) annotation (Line(
    points={{30,20},{20,20},{20,0},{10,0}},
    color={0,0,0},
    smooth=Smooth.None));
connect(spring2.flange_a, inertia2.flange_b) annotation (Line(
    points={{28,-20},{20,-20},{20,0},{10,0}},
    color={0,0,0},
    smooth=Smooth.None));
connect(inertia2.flange_a, damper1.flange_b) annotation (Line(
    points={{-10,0},{-20,0},{-20,20},{-30,20}},
    color={0,0,0},
    smooth=Smooth.None));
connect(inertia2.flange_a, spring1.flange_b) annotation (Line(
    points={{-10,0},{-20,0},{-20,-20},{-30,-20}},
    color={0,0,0},
    smooth=Smooth.None));
connect(damper1.flange_a, inertia1.flange_b) annotation (Line(
    points={{-50,20},{-60,20},{-60,0},{-70,0}},
    color={0,0,0},
    smooth=Smooth.None));
connect(spring1.flange_a, inertia1.flange_b) annotation (Line(
    points={{-50,-20},{-60,-20},{-60,0},{-70,0}},
    color={0,0,0},
    smooth=Smooth.None));
end SMD;

```

The diagram for this model, when rendered, looks like this:





This completes our discussion of basic rotational components. But there is quite a bit more to say about rotational components in the next section on [Advanced Rotational Components](#) (page 204).

Advanced Rotational Components

In the previous section, we discussed [Basic Rotational Components](#) (page 196) and showed how to build a system model from basic components. In this section we will demonstrate how to incorporate event handling, which we will use when modeling a backlash. Furthermore, we'll also show how to use parameter values to effect the interface of a component.

Modeling Backlash

Let's start our exploration of more advanced component models by looking at a rotational backlash element. The equation for a backlash model is very simple:

$$\tau = \begin{cases} c(\Delta\varphi - \frac{b}{2}) & \text{if } \Delta\varphi > \frac{b}{2} \\ c(\Delta\varphi + \frac{b}{2}) & \text{if } \Delta\varphi < -\frac{b}{2} \\ 0 & \text{otherwise} \end{cases}$$

In Modelica (where $\Delta\varphi$ is `phi_rel`), this component can be described as follows:

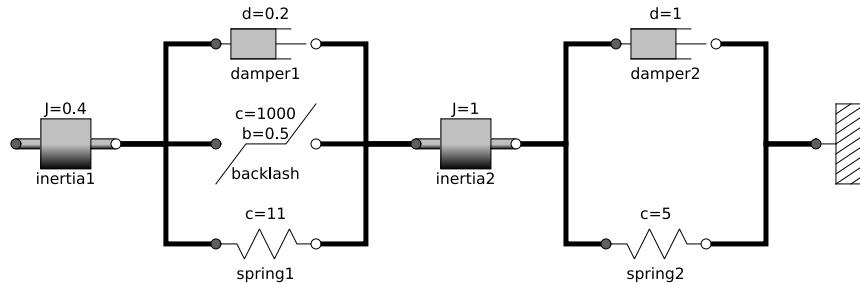
```
within ModelicaByExample.Components.Rotational.Components;
model Backlash "A rotational backlash model"
  parameter Modelica.SIunits.RotationalSpringConstant c "Torsional stiffness";
  parameter Modelica.SIunits.Angle b(final min=0) "Total lash";
  extends ModelicaByExample.Components.Rotational.Interfaces.Compliant;
equation
```

```

if phi_rel>b/2 then
  tau = c*(phi_rel-b/2);
elseif phi_rel<-b/2 then
  tau = c*(phi_rel+b/2);
else
  tau = 0 "In the lash region";
end if;
end Backlash;

```

We can add an instance of this backlash model into our previous model by placing it in parallel with the spring and the damper, *i.e.*,



If we use the inheritance mechanism in Modelica, the resulting Modelica model is quite simple:

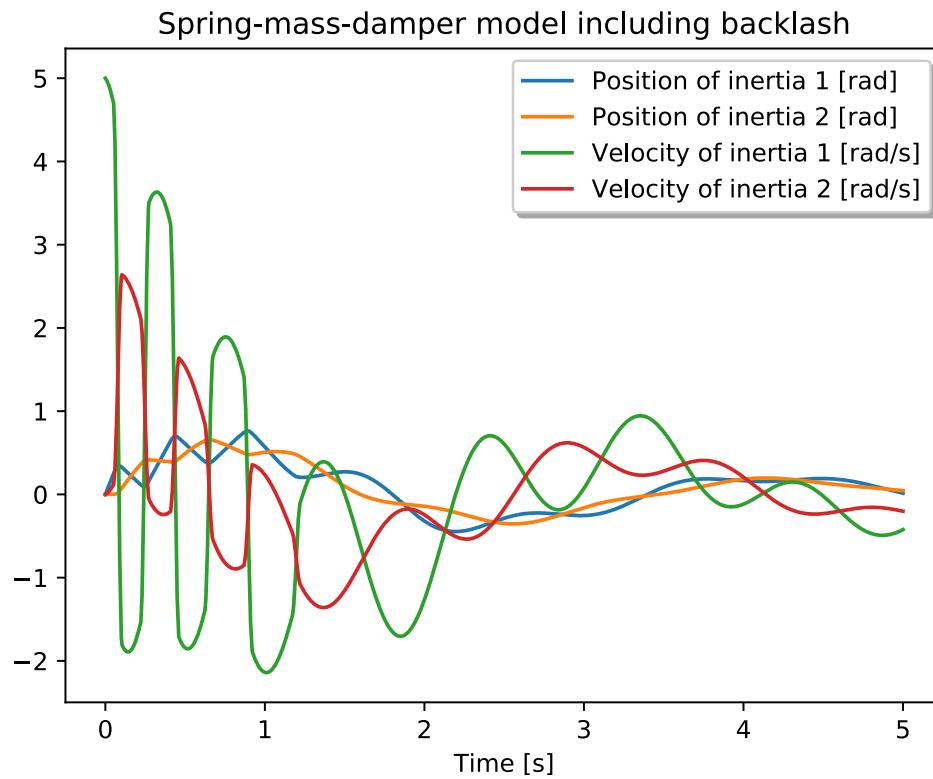
```

within ModelicaByExample.Components.Rotational.Examples;
model SMD_WithBacklash "The spring-mass-damper system with backlash"
  extends SMD(inertia2(phi(fixed=true, start=0)), inertia1(phi(fixed=true, start=0), w(start=5));
  Components.Backlash backlash(c=1000, b(displayUnit="rad") = 0.5)
  annotation (Placement(transformation(extent={{-50,-10},{-30,10}})));
equation
  connect(inertia1.flange_b, backlash.flange_a) annotation (Line(
    points={{-70,0},{-50,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(backlash.flange_b, inertia2.flange_a) annotation (Line(
    points={{-30,0},{-10,0}},
    color={0,0,0},
    smooth=Smooth.None));
end SMD_WithBacklash;

```

In this case, if the relative angle between `inertia1` and `inertia2` is more than 0.5 radians (*i.e.*, the value of `b` in our backlash instance), then the torque from the backlash element will be introduced.

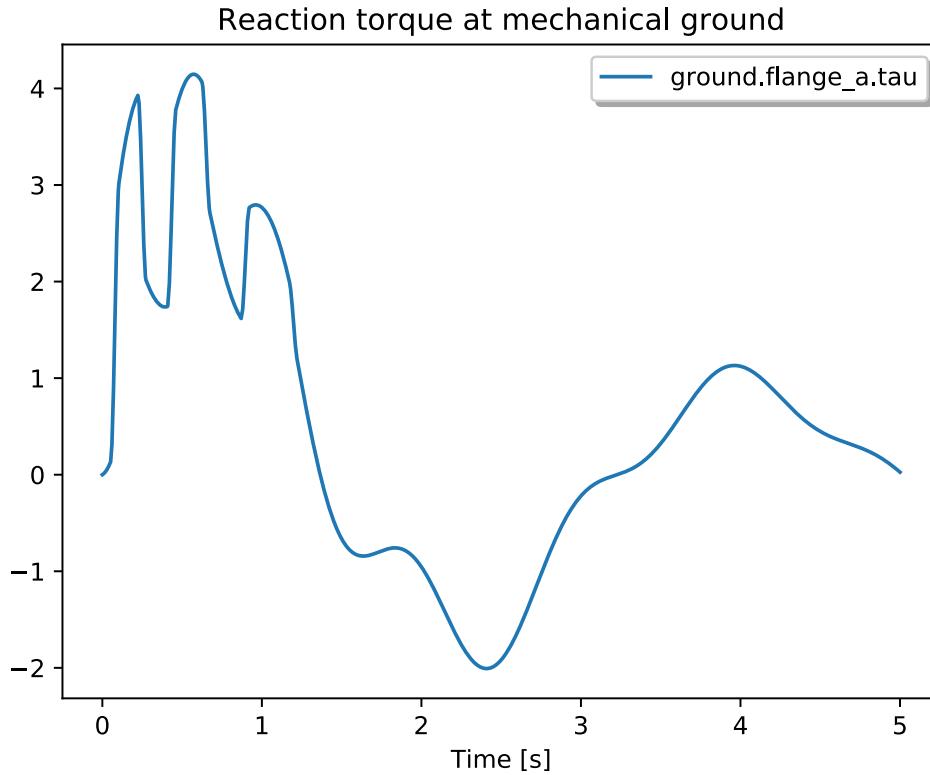
If we simulate this model, we can see the impact that the backlash's presence has on the response of the system:



Another thing worth looking at (which we will delve into much more in the next topic) is the force felt by the mechanical ground element. Looking at our schematic, it is clear that the role of the mechanical ground element is to fix the angular position on one side of our system. An equation to constrain the motion (or lack of motion, in this case) of a point in the system is called a *kinematic constraint*.

When a kinematic constraint is imposed on a system, the component imposing the constraint must generate some kind of force or torque in order to affect the motion of the system. This is called a reaction force or reaction torque.

The following plot shows the reaction torque that the mechanical ground element must impose on the system in order to fix the angular position:



Grounding and Reaction Torques

As we saw in the previous example, the behavior of the mechanical ground element is such that it must exert a reaction torque on the system to constrain the motion of the system. In this section, we will examine this effect a bit closer.

To demonstrate some of the complexities of kinematic constraints, we need to create a mechanical gear model. In this model, we will ignore the inertia of the gear elements, efficiency losses in the gear and any backlash that might exist between the teeth in the gear. Recall our discussion about *Digging Deeper* (page 188) earlier in this chapter where we mentioned that component models should focus on individual physical effects. That same principle applies here. Inertia, friction and backlash can all be modeled as individual effects (as we've already seen in this chapter). There is no need to lump them into our gear model. Instead, we will focus only on the relationship between gear input speed and output speed.

In a typical system dynamics class, the equations that describe the behavior of a gear are derived as follows. First, we start with the understanding that a gear introduces a relationship between the input speed and the output speed, *i.e.*,

$$\omega_a = R\omega_b$$

where R is the gear ratio. Recall that we assume the gear to be perfectly efficient. This means that the power going into the gear must equal the power going out which we can express mathematically as:

$$\tau_a \omega_a + \tau_b \omega_b = 0$$

Note we are using the Modelica sign conventions here where a positive value for the flow of a conserved quantity means a flow into the component. In this case, $\tau_a \omega_a$ is the flow of mechanical power into the gear from `flange_a` and $\tau_b \omega_b$ is the flow of mechanical power into the gear from `flange_b`. Therefore, their sum must be zero, since our gear model does not include the inertia of the gear elements and, therefore, no way to store energy or momentum within the gear model.

Given these two facts, we can substitute the relationship between the speeds into the relationship between the powers and get:

$$\tau_a R \omega_b + \tau_b \omega_b = 0$$

This allows us to cancel out ω_b from the equation and rearranging terms gives us:

$$\tau_b = -R\tau_a$$

Such a derivation will probably look very familiar to most engineers. But it is important to recognize that there is something missing here. More specifically, there is something implicitly assumed that is not necessarily a reasonable assumption.

To understand the issue, let's first consider Euler's second law:

$$J\ddot{\varphi} = \sum_i \tau_i$$

In other words, the sum of the torques on a body should be equal to the amount of angular momentum being accumulated by the body. Recall that our gear model doesn't include the inertia of the gear elements. As such, it has no capacity to store energy or angular momentum. If that is the case the previous equation simplifies to:

$$\sum_i \tau_i = 0$$

Our gear has only two external torques, τ_a and τ_b . Using the relationships we derived earlier, we know that their sum is:

$$\tau_a + \tau_b = \tau_a - R\tau_a = \tau_a(1 - R) = 0$$

This equation implies that for any gear ratio, R , not equal to 1.0, the torque at `flange_a` (and consequently, the torque at `flange_b` as well) must be zero. But this cannot be correct if our gear is to function as a gear.

What this mathematical relationship is showing us about the physical behavior of the system is more clearly demonstrated in this relationship:

$$\tau_a - R\tau_a = 0$$

The first term, τ_a is the torque entering the gear from `flange_a`. The second term, $R\tau_a$ is the torque entering the gear from `flange_b`. This equation tells us that these two torques will never sum to zero (for $R \neq 1$). It appears that we have proven, mathematically, that $\tau_a = 0$. But in fact, what we are really demonstrating is that there is an **imbalance** in the equation. This imbalance is the result of having forgotten something in our formulation. What is missing is the **reaction torque**.

If you aren't already familiar with this issue, you might be puzzled about where this reaction torque comes from. After all, we have only two mechanical connections to this gear and we have expressions for the torque at both of these points. But along the way, there was an implicit assumption that the **housing of the gear is grounded**. In reality, a gear has three mechanical connections. The third connection is the one between the housing of the gear and whatever the gear is mounted to. If the housing is connected to a mechanical ground, then our equations so far are correct as we can capture the behavior of such a (grounded) gear as follows:

```
within ModelicaByExample.Components.Rotational.Components;
model GroundedGear "An ideal non-reversing gear with grounded housing"
  parameter Real ratio "Ratio of phi_a/phi_b";
  extends Interfaces.TwoFlange;
equation
  ratio*flange_a.tau + flange_b.tau = 0 "No storage";
  flange_a.phi = ratio*flange_b.phi "Kinematic constraint";
annotation (Icon(graphics={
```

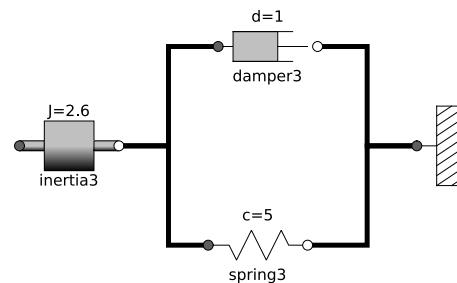
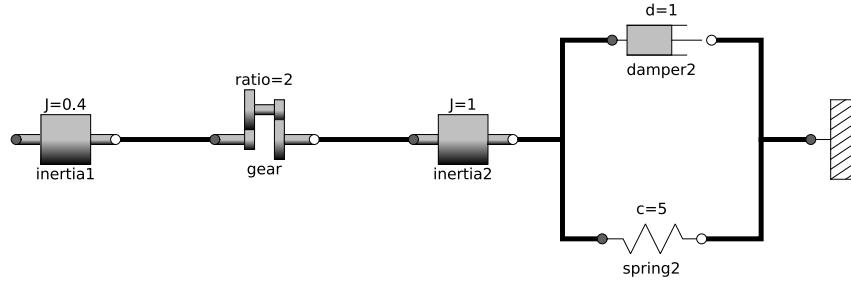
```

Rectangle(
  extent={{-100,10},{-40,-10}},
  lineColor={0,0,0},
  fillPattern=FillPattern.HorizontalCylinder,
  fillColor={192,192,192}),
Rectangle(
  extent={{-40,20},{-20,-20}},
  lineColor={0,0,0},
  fillPattern=FillPattern.HorizontalCylinder,
  fillColor={192,192,192}),
Rectangle(
  extent={{-40,100},{-20,20}},
  lineColor={0,0,0},
  fillPattern=FillPattern.HorizontalCylinder,
  fillColor={192,192,192}),
Rectangle(
  extent={{-20,70},{20,50}},
  lineColor={0,0,0},
  fillPattern=FillPattern.HorizontalCylinder,
  fillColor={192,192,192}),
Rectangle(
  extent={{20,80},{40,39}},
  lineColor={0,0,0},
  fillPattern=FillPattern.HorizontalCylinder,
  fillColor={192,192,192}),
Rectangle(
  extent={{20,40},{40,-40}},
  lineColor={0,0,0},
  fillPattern=FillPattern.HorizontalCylinder,
  fillColor={192,192,192}),
Rectangle(
  extent={{40,10},{100,-10}},
  lineColor={0,0,0},
  fillPattern=FillPattern.HorizontalCylinder,
  fillColor={192,192,192}),
Text(
  extent={{-100,140},{100,100}},
  lineColor={0,0,0},
  fillColor={255,255,255},
  fillPattern=FillPattern.Solid,
  textString="ratio=%ratio"),
Text(
  extent={{-100,-40},{100,-80}},
  lineColor={0,0,0},
  fillColor={255,255,255},
  fillPattern=FillPattern.Solid,
  textString="%name")));
end GroundedGear;

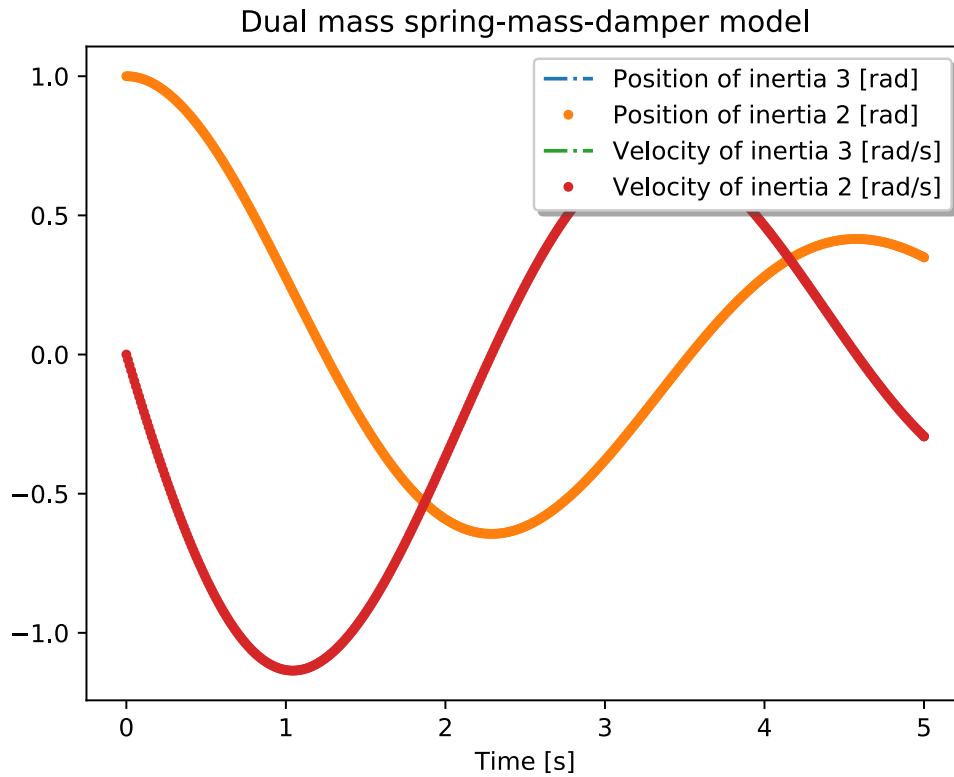
```

Note that instead of using the relationship $\omega_a = R\omega_b$ in the `GroundedGear` model, we instead used the relationship $\varphi_a = R\varphi_b$. This is actually more accurate since, once assembled, the teeth of the gear really do constrain the angular positions of the two shafts. Furthermore, there may be some applications (*e.g.* stepper motors) where preserving this relationship between positions, and not just velocities, could be very important.

Using the `GroundedGear` model, we can then build a system model using this gear as follows:



Note this system has two parallel mechanisms. The first one uses the gear model we just developed. The second one replaces the assembly of the gear and inertias with a single inertia. This single inertia was specifically chosen to have the “*effective inertia*” of the assembly. As a result, when we simulate this system, we see that `inertia2` and `inertia3` have the same response:



Comparison

As previously mentioned, the issue with the `GroundedGear` model is that it is implicitly assumed to be grounded. This assumption may not always be a reasonable one (*e.g.*, in an automotive transmission where gears are generally connected to compliant mounts). To understand how much different the response of a system can be between grounded and ungrounded gears we will first create a more complete gear model that is not implicitly grounded and then compare its performance, side by side, with gears that are grounded.

Without the implicit assumption that the housing of the gear is grounded, the kinematic relationship between the two shafts and the housing is more completely expressed as:

$$(1 - R)\varphi_h = \varphi_a - R\varphi_b$$

Although it is beyond the scope of this discussion, we can derive the following two equations using conservation of energy and conservation of momentum:

$$\tau_b = -R\tau_a\tau_h = -(1 - R)\tau_a$$

Combining these relationships and adding an additional mechanical connector to represent the housing, we get the following Modelica model for an ideal gear:

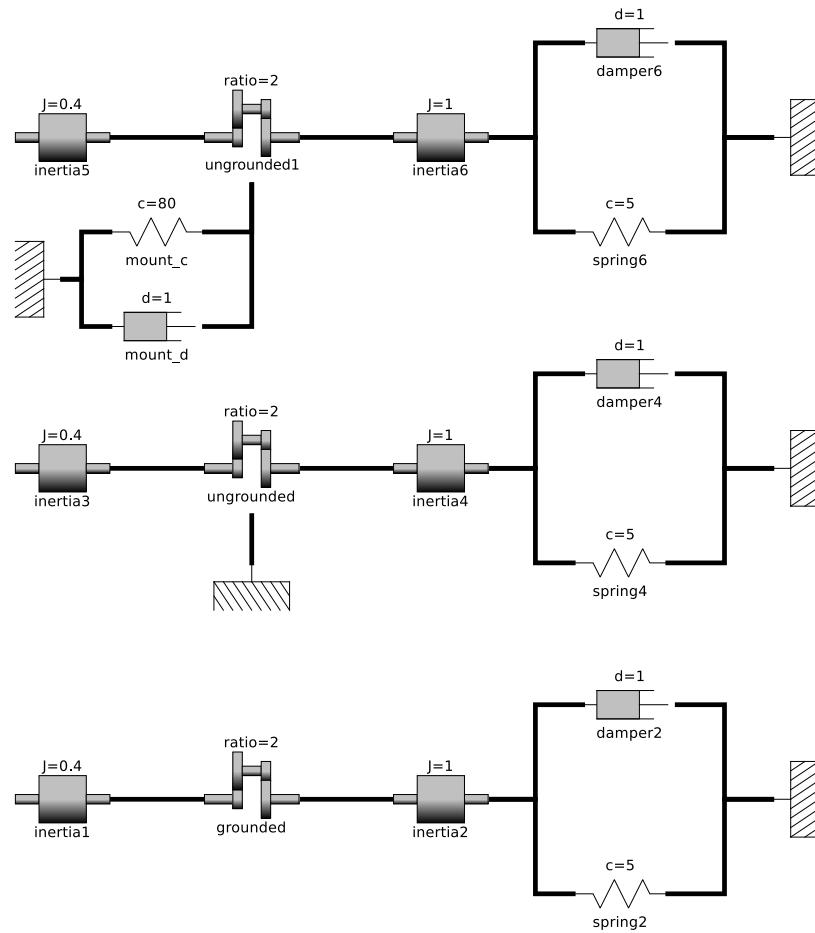
```
within ModelicaByExample.Components.Rotational.Components;
model UngroundedGear "An ideal non-reversing gear with a free housing"
  parameter Real ratio "Ratio of phi_a/phi_b";
  extends Interfaces.TwoFlange;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b housing
    "Connection for housing"
  annotation (Placement(transformation(extent={{-10,-110},{10,-90}})));
```

```

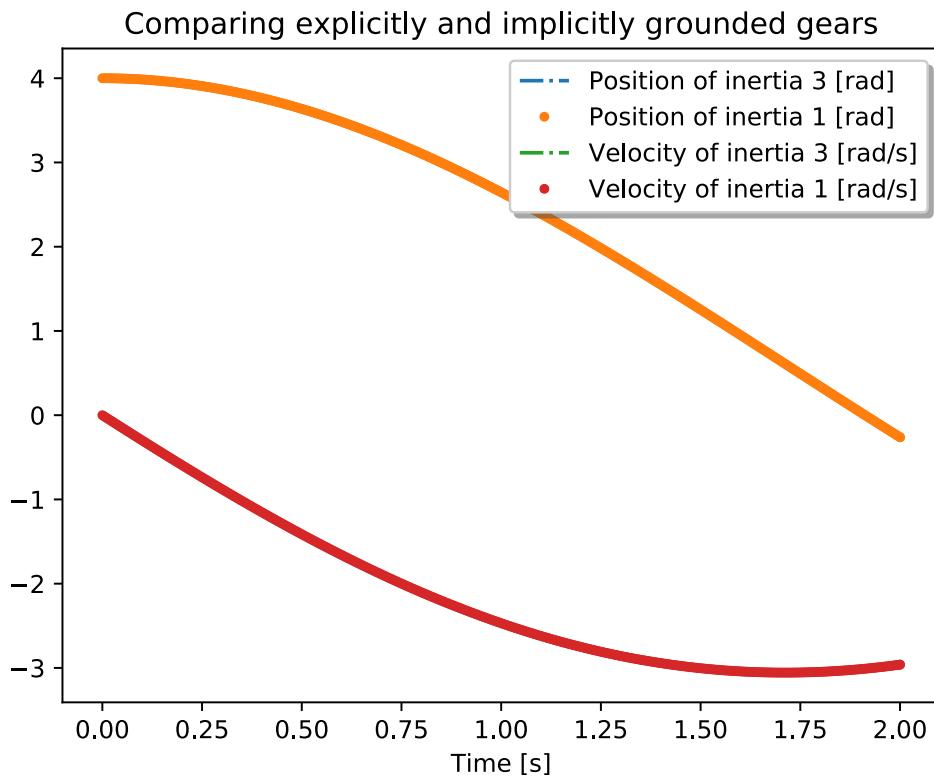
equation
  (1-ratio)*housing.phi = flange_a.phi - ratio*flange_b.phi;
  flange_b.tau = -ratio*flange_a.tau;
  housing.tau = -(1-ratio)*flange_a.tau;
end UngroundedGear;

```

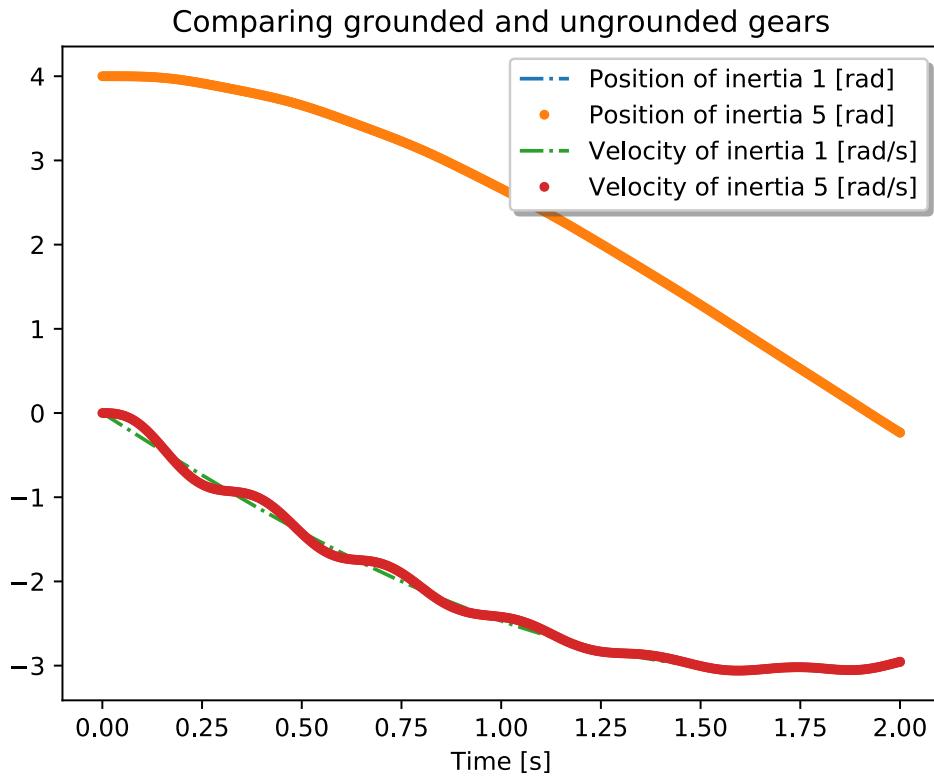
Now, let us build a system model with three different mechanisms. In each mechanism the parameters for the gear, inertia, spring and damper are all identical. The only difference is whether we use an implicit grounded gear, an explicitly grounded gear or a gear that is not directly connected to ground, but is instead connected through a very stiff mounting system. The schematic for our system looks like this when rendered:



The first thing we would expect is that the response of the mechanism with the implicitly grounded gear should be identical to the response of the mechanism with the explicitly grounded gear. This is verified by the following plot:



But the question still remains, how much difference would it make if we assumed that a gear was implicitly grounded when, in fact, it wasn't? This is clearly demonstrated in the following plot:



Optional Ground Connector

So far in our discussion of rotational systems, we've created two different gear models, `GroundedGear`, which is implicitly grounded, and `UngroundedGear`, which includes a mechanical connector for the housing. Ultimately, the equations used by these two components are quite similar and there is a considerable amount of common code between them. As we've talked about before, redundancy like this should be avoided.

One way that we can avoid redundancy in this case is to combine these two models. It might seem like this is impractical since they have very different underlying assumptions and, more importantly, **different interfaces** (*i.e.*, different connectors). Nevertheless, it is possible to combine these models by making use of something called a *conditional declaration*.

Consider the following `ConfigurableGear` model:

```
within ModelicaByExample.Components.Rotational.Components;
model ConfigurableGear
  "An ideal non-reversing gear which can be free or grounded"
  parameter Real ratio "Ratio of phi_a/phi_b";
  parameter Boolean grounded(start=false) "Set if housing should be grounded";
  extends Interfaces.TwoFlange;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b housing(phi=housing_phi,
    tau = -flange_a.tau-flange_b.tau) if not grounded "Connection for housing"
    annotation (Placement(transformation(extent={{-10,-110},{10,-90}})));
protected
  Modelica.SIunits.Angle housing_phi;
equation
  if grounded then
    housing_phi = 0;
  end if;
```

```
(1-ratio)*housing_phi = flange_a.phi - ratio*flange_b.phi;
flange_b.tau = -ratio*flange_a.tau;
end ConfigurableGear;
```

In particular, notice that the declaration of `housing` ends with `if not grounded`. When `if` appears at the end of a declaration, it indicates that the variables only exists if the condition following the `if` is true. So when the `grounded` parameter is true, there is no `housing` connector in this model. Furthermore, the equations included, as modifications, in the declaration of `housing` (*i.e.*, `phi=housing_phi` and `tau=-flange_a.tau-flange_b.tau`) also disappear with the declaration.

Meanwhile, in the `equation` section, we see that the `if` statement there provides an additional equation, `housing_phi=0`, in the case when the model is grounded. This is necessary because the variable `housing_phi` is always present (*i.e.*, there is no `if` at the end of its declaration) so there must be an equation for it.

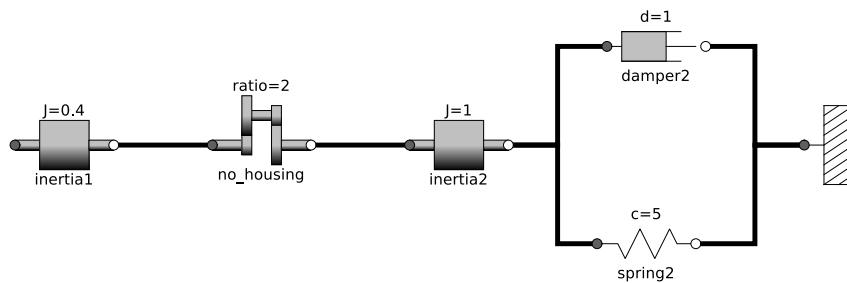
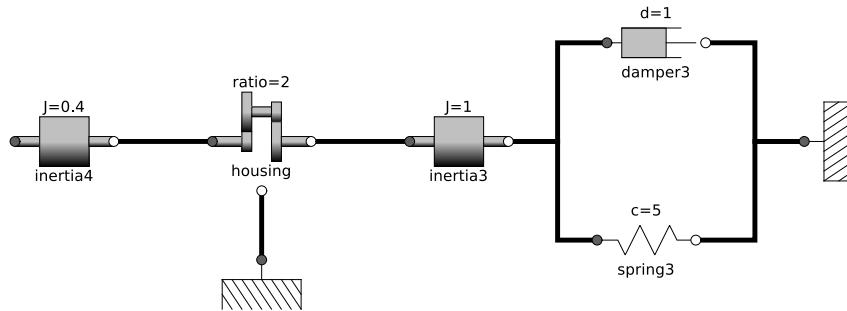
To understand more completely what is going on here, recall that the number of equations required by a component model is equal to the number of flow variables across all the component's connectors + the number of (non-parameter) variables declared in the model.

The following table summarizes how these things add up for the case where `grounded` is true and the case where it isn't:

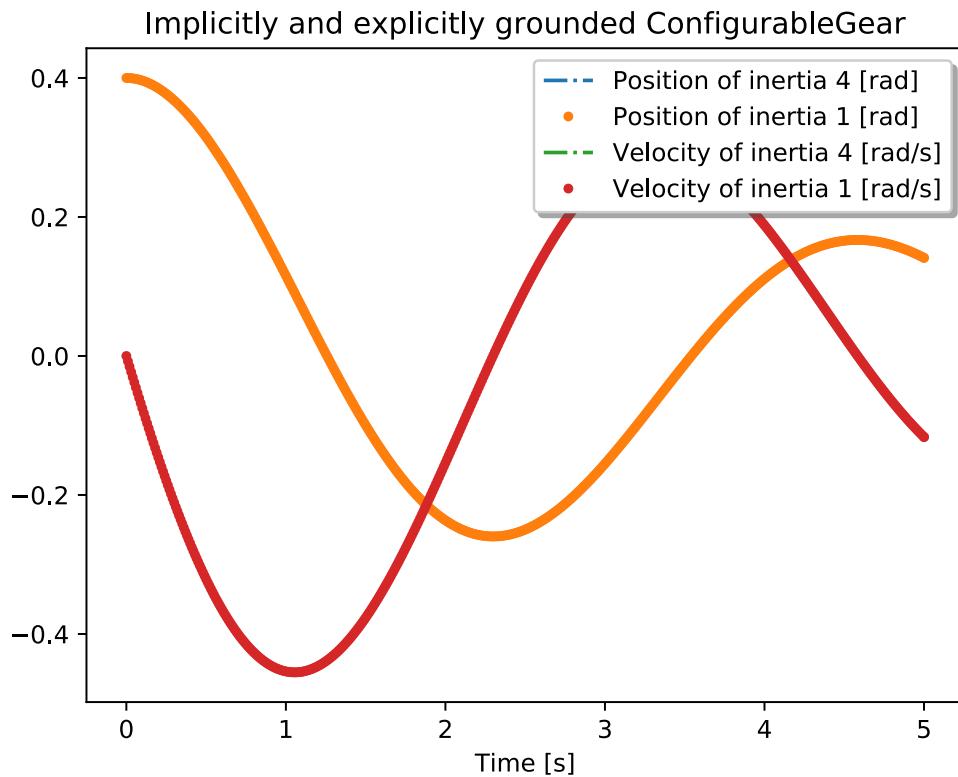
Quantity	<code>grounded==true</code>	<code>grounded==false</code>
Number of flow variables	2	3
Number of variables	1 (<code>housing_phi</code>)	1 (<code>housing_phi</code>)
Equations required	3	4
Equations in declarations	0	2 (from <code>housing</code>)
Equations in <code>equation</code> section	3	2
Equations provided	3	4

When using conditional declarations, it is very important to make sure that the number of equations provided balances with the number of equations required for all possible conditions. In this case, we have only two conditions to concern ourselves with and we can clearly see from this table that we have met this requirement in both cases.

The following model demonstrates how we can now use the `ConfigurableGear` models as both an implicitly and explicitly grounded gear:



And, as we would expect, the response for `inertia1` and `inertia4` are identical:



Lotka-Volterra Equations Revisited

In this section, we will revisit the *Lotka-Volterra Systems* (page 18) discussed in the first chapter. However, this time we will create system models from individual components. After recreating the behavior shown in the first chapter, we'll expand the set of effects we consider and reconfigure these component models into other system models that demonstrate different dynamics.

Classic Lotka-Volterra

We'll start by looking at the classic Lotka-Volterra system. In order to create such a system using component models, we will require models to represent the population of both rabbits and foxes as well as models for reproduction, starvation and predation.

Connectors

However, as we learned during our discussion of *Connectors* (page 169), before we can start building component models we first need to formally define the information that will be exchanged by interacting components by defining connectors. The `connector` we will use in this section is the `Species` connector and it is defined as follows:

```
within ModelicaByExample.Components.LotkaVolterra.Interfaces;
connector Species "Used to represent the population of a specific species"
    Real population "Animal population";
    flow Real rate "Flows that affect animal population";
end Species;
```

This connector definition is interesting because these definitions do not come from engineering. Instead, they really arise from ecology. In this case, our across variable is `population` which represents the actual number of animals of a particular species. Our through variable, indicated by the presence of the `flow` qualifier, is `rate` which represents the rate at which new animals “enter” the component that this connector is attached to.

Regional Population

To track the population of a given species in one region, we'll use the `RegionalPopulation` model. The model has several noteworthy aspects so we'll present the model piece by piece starting with:

```
within ModelicaByExample.Components.LotkaVolterra.Components;
model RegionalPopulation "Population of animals in a specific region"
    encapsulated type InitializationOptions = enumeration(
        Free "No initial conditions",
        FixedPopulation "Specify initial population",
        SteadyState "Population initially in steady state");
```

The first two lines are as expected. But after that we see that this model defines a type called `InitializationOptions`. The type definition is qualified with the `encapsulated` keyword. This is necessary because the type is being defined within a model and not a package. Modelica has a rule that if we wish to refer to this type from outside the `model` definition, the type definition must be prefixed by `encapsulated`. We can see from the definition of this `enumeration` that it defines three distinct values: `Free`, `FixedPopulation` and `SteadyState`. We'll see how these values will be used shortly.

The first declarations in our `RegionalPopulation` model is:

```
parameter InitializationOptions init=InitializationOptions.Free
annotation(choicesAllMatching=true);
```

Note that the first parameter, `init`, utilizes the `InitializationOptions` enumeration both to specify its type (the enumeration itself) and its initial value, `Free`. Also note the presence of the `choicesAllMatching` annotation. We'll talk more about this annotation later in this chapter, when we are reviewing the concepts introduced here, and in subsequent chapters.

The next declaration is:

```
parameter Real initial_population
annotation(Dialog(group="Initialization",
enable=init==InitializationOptions.FixedPopulation));
```

The `initial_population` parameter is used to represent the initial value for the population in this region at the start of the simulation. However, as we will see shortly in the equation section, this value is only used if the value of `init` is set to `FixedPopulation`. For this reason, the `enable` annotation on this `parameter` is set to `init==InitializationOptions.FixedPopulation`. This annotation is used to inform Modelica tools of this relationship. This information can then be taken into account when building graphical user interfaces (*e.g.*, a parameter dialog) associated with this model.

It is also worth noting the presence of the `Dialog` annotation in the definition of `initial_population`. This annotation allows the model developer to organize parameters into categories, in this case "Initialization". Tools generally use such information to help structure parameter dialogs.

The last public declaration in the model is for the `connector` instance that allows interactions with other components:

```
Interfaces.Species species
annotation (Placement(transformation(extent={{-10,90},{10,110}}),
iconTransformation(extent={{-10,90},{10,110}})));
```

Here we again see the *Component Placement* (page 261) annotation which we will again defer talking about for the moment. This leaves the last declaration, which happens to be `protected`:

```
protected
Real population(start=10) = species.population "Population in this region";
```

This variable represents the number of animals in this region. It is given a non-zero `start` value to avoid the trivial solutions we saw in our earlier discussion of *Steady State Initialization* (page 22) of Lotka-Volterra systems. We can also see, from this declaration, that this declaration equates the local variable, `population`, with the value of the across variable on the `species` connector, `species.population`. In effect, the `population` variable is an alias for the expression `species.population`.

Now that we have the declarations out of the way, let's look at the equations associated with the `RegionalPopulation` model:

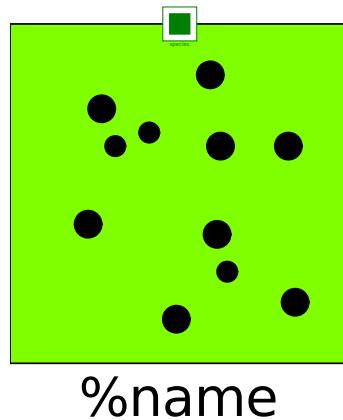
```
initial equation
if init==InitializationOptions.FixedPopulation then
  population = initial_population;
elseif init==InitializationOptions.SteadyState then
  der(population) = 0;
else
end if;
equation
  der(population) = species.rate;
  assert(population>=0, "Population must be greater than or equal to zero");
end RegionalPopulation;
```

The `initial equation` demonstrates the significance that the value of the `init` parameter has on the behavior of this component. In the case that the `init` value is equal to the `FixedPopulation` value in the `InitializationOptions` enumeration, an equation is introduced specifying that the value of the `population` variable at the start of the simulation is equal to the `initial_population` parameter. If, on the other hand, the value of `init` is equal to the `SteadyState` value of the enumeration, then an equation is introduced specifying that the rate of population change at the start of the simulation must be zero. Obviously, if `init` is equal to `Free` (the last remaining possibility), no initial equation is introduced.

Within the `equation` section, we see that the rate at which the `population` changes is equal to the value of the `flow` variable on the `species` connector, `species.rate`. Again, recall the sign convention that a positive value for a `flow` variable means a flow into the component and the fact that this equation is consistent with that sign convention (*i.e.*, a positive value for `species.rate` will have the effect of increasing the `population` within the region).

The last thing worth noting about the model is the presence of the `assert` call in the `equation` section. This is used to define a model boundary (*i.e.*, a point beyond which the equations in the model are not valid). It is used to enforce the constraint that the population within a region cannot be less than zero. If a solution is ever found where the constraint is violated, the resulting error message will be “Population must be greater than zero”.

This model also has an Icon annotation associated with the model definition. As usual, the Icon annotation is not included in the source listing. But when this component model is rendered within a system model, its icon will look like this:



Reproduction

The first real effect we will examine is reproduction. As we know from our previous discussion, the growth in a given population due to reproduction is proportional to the number of animals of that species in a given region. As a result, we can describe reproduction very succinctly as:

```
within ModelicaByExample.Components.LotkaVolterra.Components;
model Reproduction "Model of reproduction"
  extends Interfaces.SinkOrSource;
  parameter Real alpha "Birth rate proportionality constant";
equation
  growth = alpha*species.population "Growth is proportional to population";
end Reproduction;
```

where `alpha` is the proportionality constant. However, the simplicity and clarity of this model is due mostly to the inheritance of the `SinkOrSource` model in much the same way that our “DRY” *Electrical Components* (page 192) benefited from inheriting the `TwoPin` model.

The `SinkOrSource` model is a starting point for any model that either creates or destroys animals in a population. It is defined as follows:

```
within ModelicaByExample.Components.LotkaVolterra.Interfaces;
partial model SinkOrSource "Used to describe single species effects"

  Species species
    annotation (Placement(transformation(extent={{-10,90},{10,110}})));
  protected
    Real growth "Growth in the population (if positive)";
    Real decline "Decline in the population (if positive)";
```

```

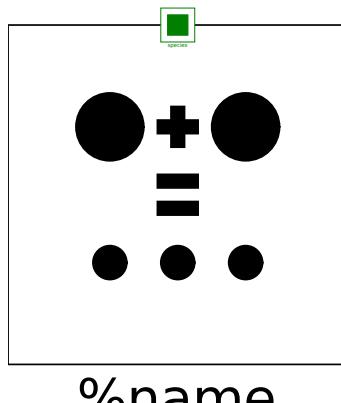
equation
  decline = -growth;
  species.rate = decline;
end SinkOrSource;

```

To understand these equations it is first necessary to understand that any model that `extends` from `SinkOrSource` will generally be connected to a `RegionalPopulation` instance (but will not, itself, `be` a `RegionalPopulation` model). This means that if the flow variable `species.rate` in such an instance is positive, it will have the effect of pulling animals **out** of the `RegionalPopulation` model. Looking at the `SinkOrSource` model in this way, we can see that the variable `decline` is simply an alias for `species.rate`. In other words, when `decline` has a positive value, `species.rate` will have a positive value and, therefore, any `RegionalPopulation` that this `SinkOrSource` instance is connected to will suffer a drain on its population. Conversely, the `growth` variable is positive when `species.rate` is negative. In that case, the connected `RegionalPopulation` model will see an increase in species population.

By defining the `SinkOrSource` model and inheriting from it, much of this complexity is hidden. As a result, models like `Reproduction` can have equations written in a way that make their behavior more intuitive, *e.g.*, `growth = alpha*species.population`.

Although not shown, the Icon for the `Reproduction` model is rendered as:



`%name`

Starvation

Just like the `Reproduction` model just described, the `Starvation` model also inherits from the `SinkOrSource` model. However, its behavior with respect to the `decline` variable, is described as follows:

```

within ModelicaByExample.Components.LotkaVolterra.Components;
model Starvation "Model of starvation"
  extends Interfaces.SinkOrSource;
  parameter Real gamma "Starvation coefficient";
equation
  decline = gamma*species.population
  "Decline is proportional to population (competition)";
end Starvation;

```

Predation

The last effect we need to consider before building a system model to represent the classic Lotka-Volterra behavior is a model for predation.

Recall our previous discussion of the `SinkOrSource` model and the potential confusion associated with sign conventions. The `SinkOrSource` model was designed to work with effects that only interacted with

a single `RegionalPopulation` (since it had only one `Species` connector). In order to address the same potential sign convention confusion for effects that involve interactions between two different regional populations, the following partial model, `Interaction` was defined:

```
within ModelicaByExample.Components.LotkaVolterra.Interfaces;
partial model Interaction "Used to describe interactions between two species"
  Species a "Species A"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Species b "Species B"
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Real a_growth "Growth in population of species A (if positive)";
  Real a_decline "Decline in population of species A (if positive)";
  Real b_growth "Growth in population of species B (if positive)";
  Real b_decline "Decline in population of species B (if positive)";
equation
  a_decline = -a_growth;
  a.rate = a_decline;
  b_decline = -b_growth;
  b.rate = b_decline;
end Interaction;
```

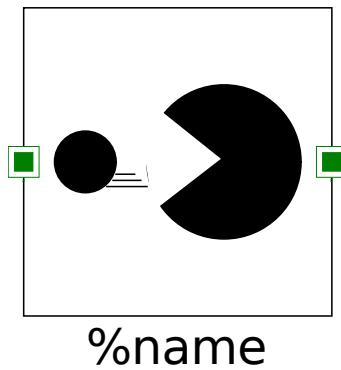
Again, we have the concepts of growth and decline variables. However this time we have two version of each. One is associated with the `a` connector and the other is associated with the `b` connector.

Using these definitions, we can define `Predation` very simply as:

```
within ModelicaByExample.Components.LotkaVolterra.Components;
model Predation "Model of predation"
  extends Interfaces.Interaction;
  parameter Real beta "Prey (Species A) consumed";
  parameter Real delta "Predators (Species B) fed";
equation
  b_growth = delta*a.population*b.population;
  a_decline = beta*a.population*b.population;
end Predation;
```

This model captures the effect that the growth in the “B” (predator) population is proportional to the product of the predator and prey populations. Similarly, the decline in the “A” (prey) population is also proportional the product of the predator and prey populations (although with a different proportionality constant).

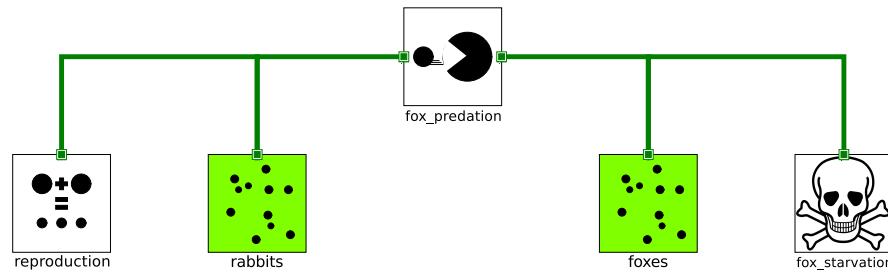
Although not shown, the Icon for the `Predation` model is rendered as:



Note that the `Predation` model is asymmetric. The `b` connector should be connected to the predator population and the `a` connector should be connected to the prey population. This is reinforced by the image and asymmetry of the icon itself.

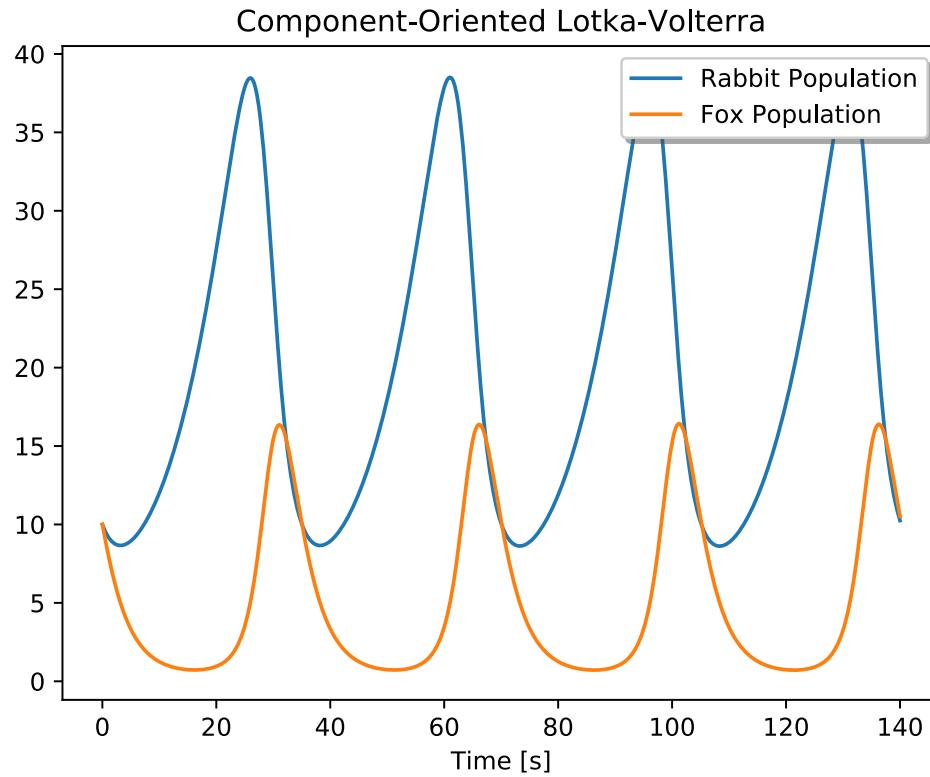
Classic System Model

With all of these components in hand, we can very easily construct a component-oriented version of the classic Lotka-Volterra behavior by dragging and dropping the components into the following system configuration:



Here we see that the **Starvation** model is attached to the **foxes** population while the **Reproduction** model is attached to the **rabbits** population. The **Predation** model is connected to both populations with the **a** (prey) connector attached to the **rabbits** and the **b** (predator) connector attached to the **foxes**.

As we can see from the following plot, the behavior of this system is identical to the one presented in our earlier discussion of *Lotka-Volterra Systems* (page 18):



Introducing a Third Species

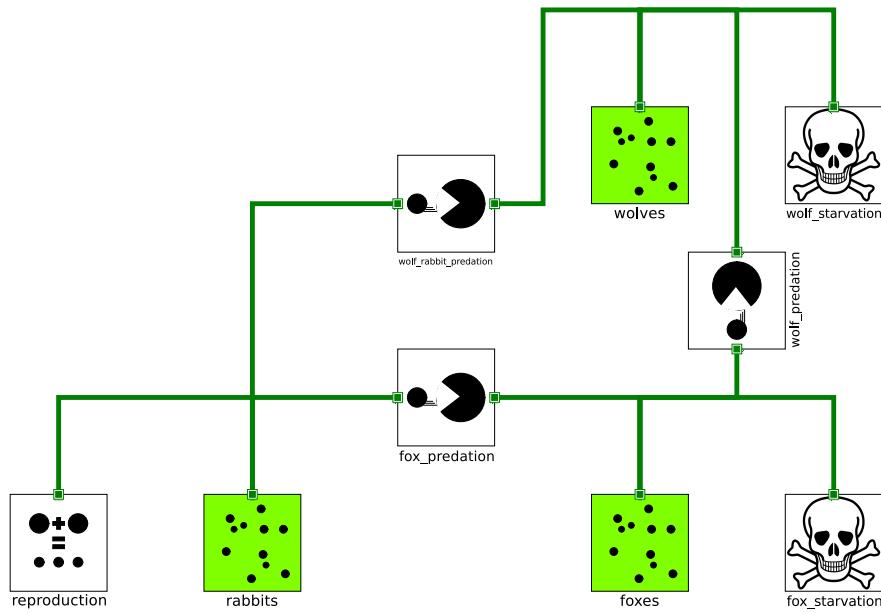
As we will see over and over again, there is an initial investment in building component models over simply typing the equations that they encapsulate. But there is also a significant “payoff” to this process because of the schematic based system composition that is then possible as a result. In the context of the Lotka-Volterra example, this is exemplified by the addition of a third species, wolves, to the classic Lotka-Volterra system.

Adding Wolves

The creation of a model with a third species does not require any additional component models to be defined. Instead, we can reuse not only our existing models for Predation, Starvation and RegionalPopulation, but we can also reuse the `ClassicLotkaVolterra` model itself:

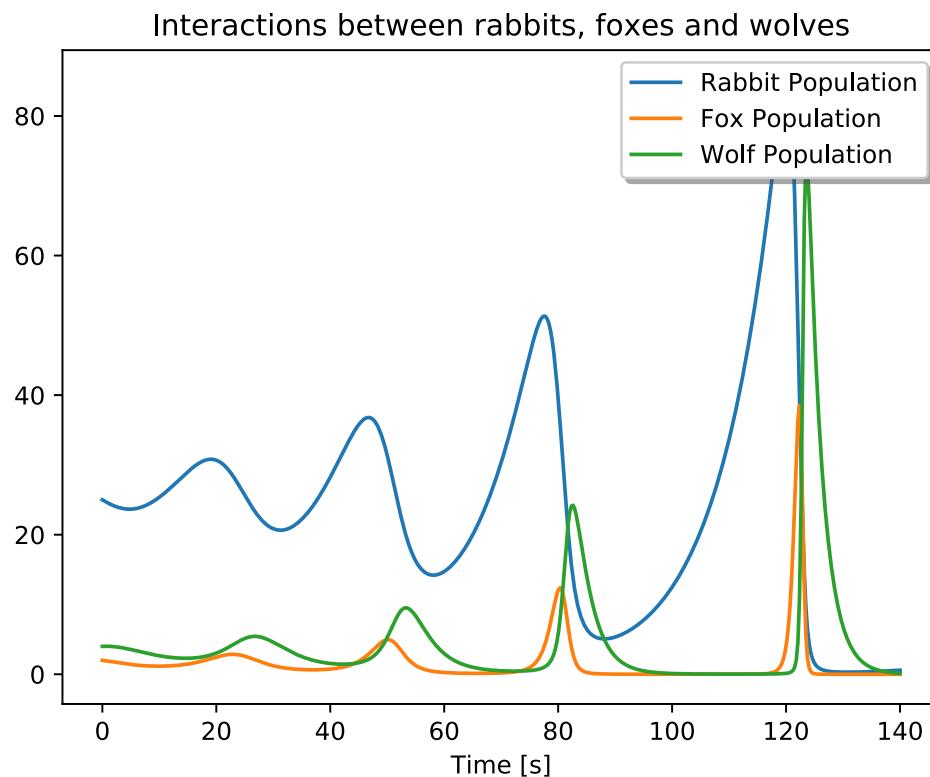
```
within ModelicaByExample.Components.LotkaVolterra.Examples;
model ThirdSpecies "Adding a third species to Lotka-Volterra"
  import ModelicaByExample.Components.LotkaVolterra.Components.RegionalPopulation.
  ↵InitializationOptions.FixedPopulation;
  extends ClassicLotkaVolterra(rabbits(initial_population=25), foxes(initial_population=2));
  Components.RegionalPopulation wolves(init=FixedPopulation, initial_population=4)
    annotation (Placement(transformation(extent={{30,40},{50,60}})));
  Components.Starvation wolf_starvation(gamma=0.4)
    annotation (Placement(transformation(extent={{70,40},{90,60}})));
  Components.Predation wolf_predation(beta=0.04, delta=0.08) "Wolves eating Foxes"
    annotation (Placement(transformation(extent={{-10,-10},{10,10}}), rotation=90, origin={60,
  ↵-20}));
  Components.Predation wolf_rabbit_predation(beta=0.02, delta=0.01) "Wolves eating rabbits"
    annotation (Placement(transformation(extent={{-10,30},{10,50}})));
equation
  connect(wolf_predation.b, wolves.species) annotation (Line(
    points={{60,30},{60,80},{40,80},{40,60},{40,60}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(wolf_rabbit_predation.a, rabbits.species) annotation (Line(
    points={{-10,40},{-40,40},{-40,-20}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(wolf_predation.a, foxes.species) annotation (Line(
    points={{60,10},{60,0},{40,0},{40,-20}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(wolf_starvation.species, wolves.species) annotation (Line(
    points={{80,60},{80,80},{40,80},{40,60}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(wolves.species, wolf_rabbit_predation.b) annotation (Line(
    points={{40,60},{40,80},{20,80},{20,40},{10,40}},
    color={0,127,0},
    smooth=Smooth.None));
  annotation (experiment(StopTime=100, Tolerance=1e-006));
end ThirdSpecies;
```

Such a model would not typically be created by typing in the source code you see above. Instead, within a graphical development environment it would take less than a minute to assemble such a system by augmenting the existing `ClassicLotkaVolterra` model. When visualized, the schematic for the resulting system is rendered as:



Resulting Dynamics

By creating such a model, we can quickly explore the differences in system dynamics between the classic model and this three species model. The following plot shows how these three species interact:

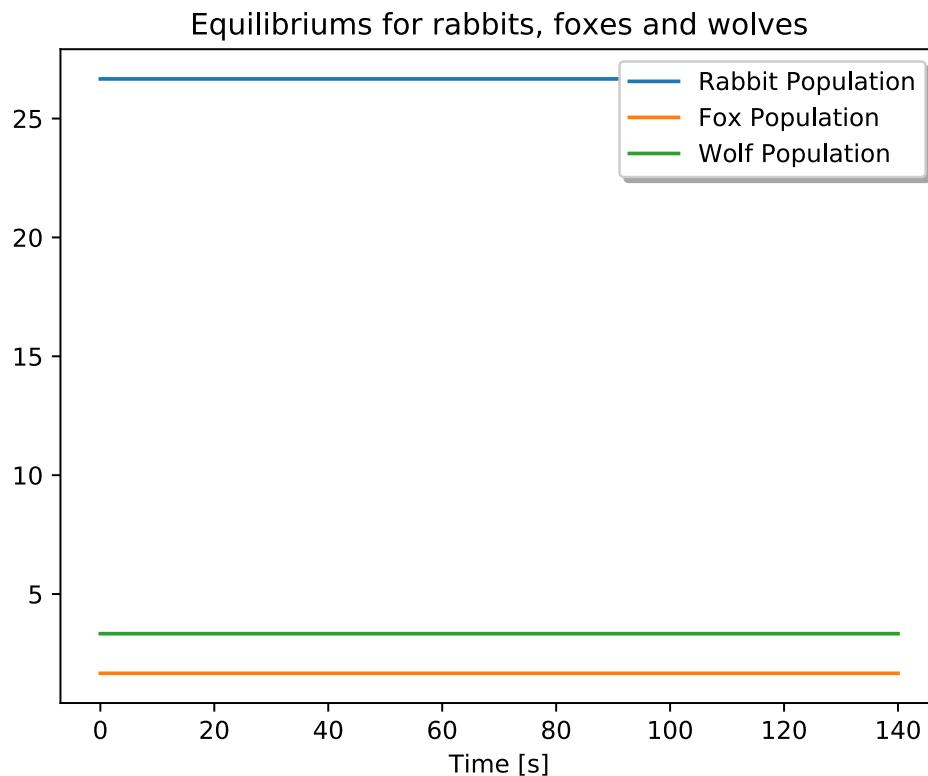


By using the `init` parameter in the various `RegionalPopulation` instances, we can also quickly create

a model to solve for the equilibrium population levels for all three species:

```
within ModelicaByExample.Components.LotkaVolterra.Examples;
model ThreeSpecies_Quiescent "Three species in a quiescent state"
  import ModelicaByExample.Components.LotkaVolterra.Components.RegionalPopulation.
  <InitializationOptions.SteadyState;
  extends ThirdSpecies(
    rabbits(init=SteadyState, population(start=30)),
    foxes(init=SteadyState, population(start=2)),
    wolves(init=SteadyState, population(start=4)));
  annotation (experiment(StopTime=100, Tolerance=1e-006));
end ThreeSpecies_Quiescent;
```

All that is required in this model is to extend from the `ThirdSpecies` model and modify the `init` parameter for each of the underlying species populations. Simulating this model gives us the equilibrium population level for each species:



From an ecological standpoint, we can already make an interesting observation about this system. If we start it from a non-equilibrium condition the system quickly goes unstable. In other words, the introduction of wolves into the otherwise stable eco-system involving only rabbits and foxes has a significant impact on the population dynamics.

Speed Measurement Revisited

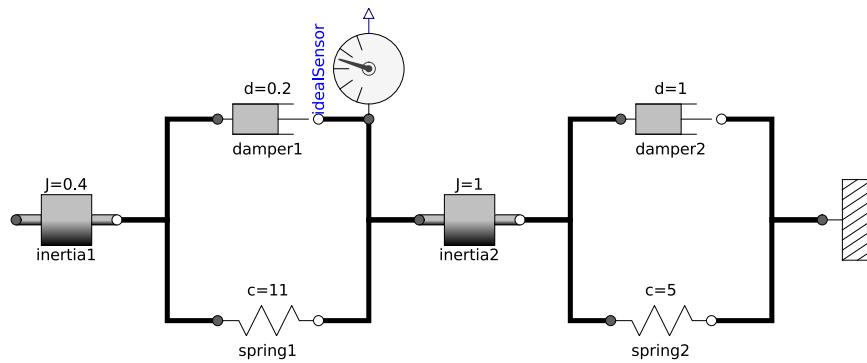
Recall our previous discussion on [Speed Measurement](#) (page 63). That discussion took place before we had introduced the idea of building reusable components. As a result, adding equations associated with the various speed measurement techniques was awkward.

In this section, we'll revisit that topic and all of the various speed estimation techniques discussed there.

As before, we assume that we have an existing model of the “plant” (the system whose speeds we wish to measure). But this time, we’ll create reusable component models for different speed estimation algorithms and add them to the plant model as graphical components.

Plant Model

We’ll start with our “plant model”. It is identical in behavior to the system we used in our previous discussion of *Speed Measurement* (page 63) and when rendered as a schematic, looks like this:

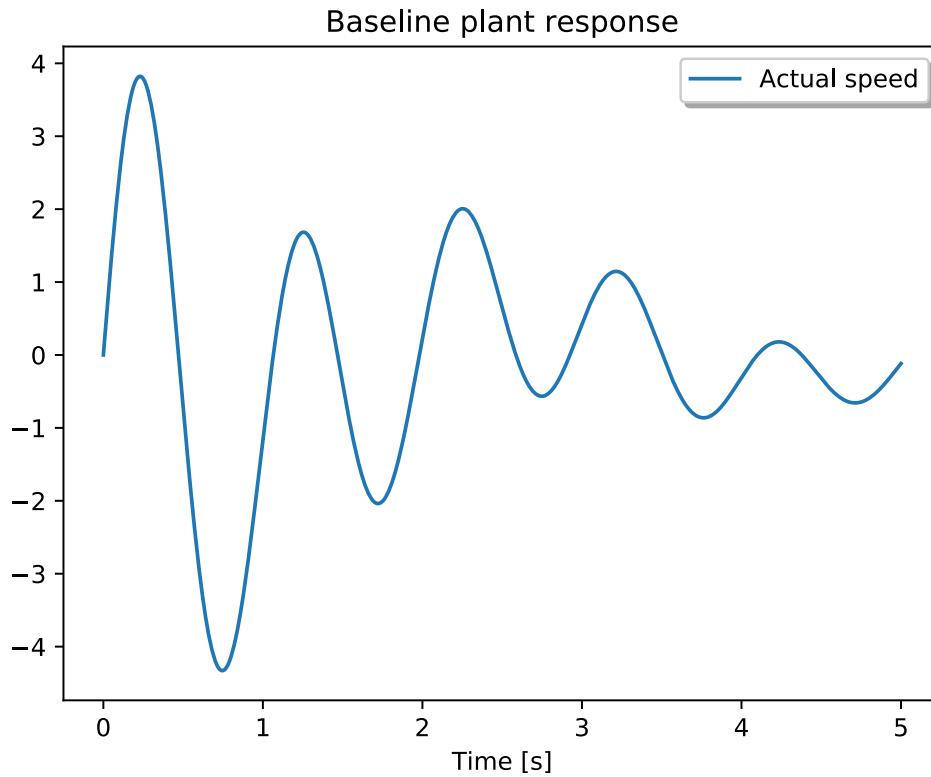


The source code for the underlying Modelica model is:

```
within ModelicaByExample.Components.SpeedMeasurement.Examples;
model Plant "The basic plant model"
  extends ModelicaByExample.Components.Rotational.Examples.SMD;
  Components.IdealSensor idealSensor
  annotation (Placement(transformation(extent={{-10,-10},{10,10}}),
    rotation=90, origin={-20,30}));
  equation
    connect(idealSensor.flange, inertia1.flange_a) annotation (Line(
      points={{-20,20}, {-20,0}, {-10,0}}, color={0,0,0},
      smooth=Smooth.None));
    annotation(experiment(StopTime=10, Tolerance=1e-006));
end Plant;
```

In the remaining sections we will create models that use different approximation techniques to measure the speed of the `inertia1` component in our plant model.

Before we look at the different speed approximation methods, let’s have a look at the actual speed response from our plant model.



Note that this is exactly the same response we saw when we initially covered this topic.

Sample and Hold Sensor

Previously, we discussed the *Sample and Hold* (page 65) approach to speed measurement. Here we will revisit this topic, but encapsulate the speed approximation in a reusable sensor model. The following model implements the “sample and hold” approximation to speed measurement:

```
within ModelicaByExample.Components.SpeedMeasurement.Components;
model SampleHold "A sample-hold ideal speed sensor"
  extends Interfaces.SpeedSensor;
  parameter Modelica.SIunits.Time sample_time;
initial equation
  w = der(flangе.φ);
equation
  when sample(0, sample_time) then
    w = der(flangе.φ);
  end when;
end SampleHold;
```

Behaviorally, there is no difference between this estimation technique and our previous implementation of *Sample and Hold* (page 65). But our approach is different this time because we have wrapped that estimation technique in a reusable component model.

We have once again saved ourselves some trouble by utilizing a **partial** model to represent code that will be common across our various sensor models. As we can see from the definition of the **SpeedSensor** model:

```
within ModelicaByExample.Components.SpeedMeasurement.Interfaces;
partial model SpeedSensor "The base class for all of our sensor models"
  extends Modelica.Mechanics.Rotational.Interfaces.PartialAbsoluteSensor;
  Modelica.Blocks.Interfaces.RealOutput w "Sensed speed"
  annotation (Placement(transformation(extent={{100,-10},{120,10}})));
end SpeedSensor;
```

We can see from the `SpeedSensor` model that the output signal is named `w`. But we also see that `SpeedSensor` inherits from another model in the Modelica Standard Library, `PartialAbsoluteSensor`. The `PartialAbsoluteSensor` model is defined as:

```
partial model PartialAbsoluteSensor
  "Partial model to measure a single absolute flange variable"
  extends Modelica.Icons.RotationalSensor;

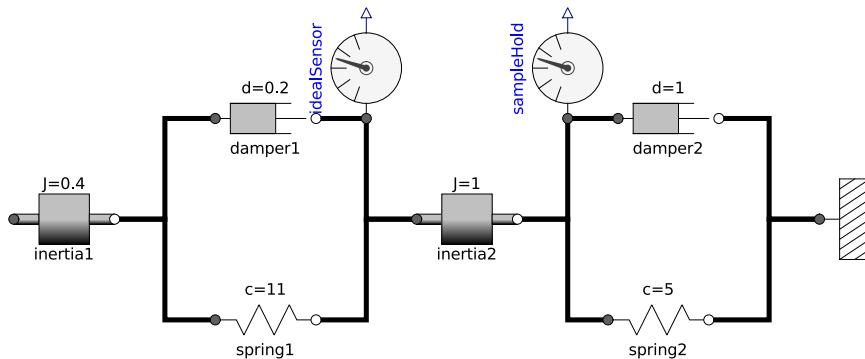
  Flange_a flange "Flange from which speed will be measured"
  annotation(Placement(transformation(extent={{-110,-10},{-90,10}}), rotation=0));
equation
  0 = flange.tau;
end PartialAbsoluteSensor;
```

In addition to providing a nice icon, the `PartialAbsoluteSensor` model features a rotational connector, `flange`. Furthermore, the model assumes that the sensor model is completely passive (*i.e.*, it has no impact on the system it is sensing) since it applies zero torque at the connection point.

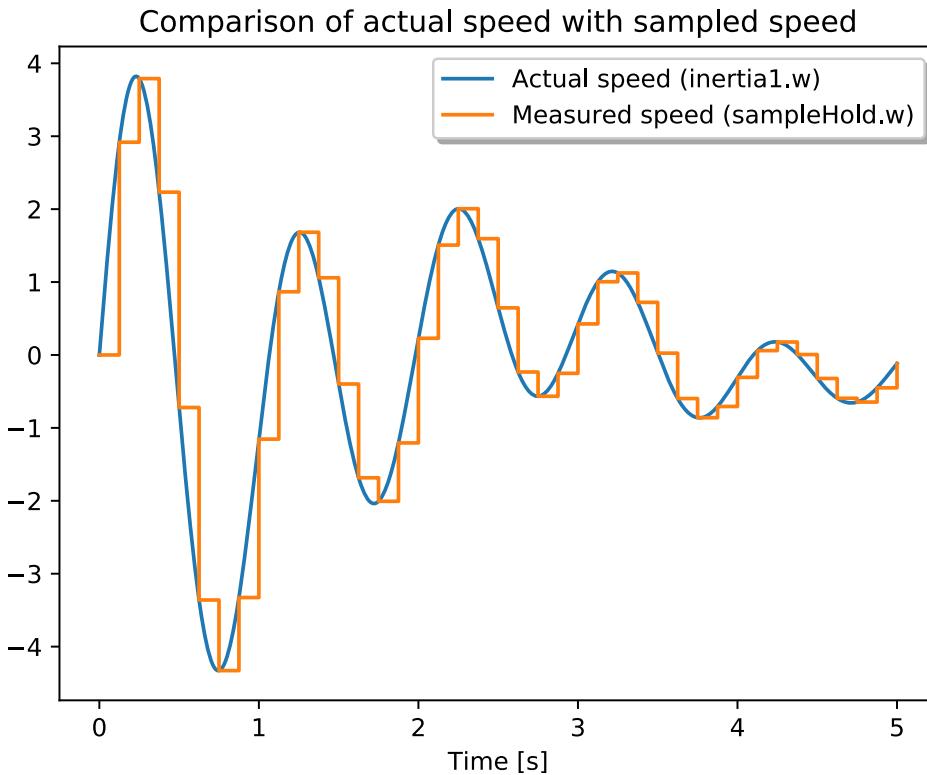
To test this model, we simply extend from our `Plant` model, add an instance of the `SampleHold` sensor and connect it to the `inertia`, *e.g.*,

```
within ModelicaByExample.Components.SpeedMeasurement.Examples;
model PlantWithSampleHold "Comparison between ideal and sample-hold sensor"
  extends Plant;
  Components.SampleHold sampleHold(sample_time=0.125)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}}),
                           rotation=90, origin={20,30}));
equation
  connect(sampleHold.flange, inertia1.flange_b) annotation (Line(
    points={{20,20},{20,0},{10,0}}, color={0,0,0},
    smooth=Smooth.None));
  annotation (experiment(StopTime=10, Tolerance=1e-006));
end PlantWithSampleHold;
```

When assembled, our final system looks like this:



If we simulate this system for 5 seconds, we can compare the actual speed of the inertia with the signal returned from our sensor:



These results are identical to the results from our previous discussion of the [Sample and Hold](#) (page 65) approach.

Interval Measurement

Now let us turn our attention to the [Interval Measurement](#) (page 66) technique. Again, we will create a reusable component model by extending from our `Sensor` model. This time, the implementation will use the time between teeth to estimate speed:

```
within ModelicaByExample.Components.SpeedMeasurement.Components;
model IntervalMeasure
  "Estimate speed by measuring interval between encoder teeth"
  extends Interfaces.SpeedSensor;
  parameter Integer teeth;
  Real next_phi, prev_phi;
  Real last_time;
protected
  parameter Modelica.SIunits.Angle tooth_angle=2*Modelica.Constants.pi/teeth;
initial equation
  next_phi = flange.phi+tooth_angle;
  prev_phi = flange.phi-tooth_angle;
  last_time = time;
equation
  when {flange.phi>=pre(next_phi), flange.phi<=pre(prev_phi)} then
    w = tooth_angle/(time-pre(last_time));
    next_phi = flange.phi+tooth_angle;
    prev_phi = flange.phi-tooth_angle;
    last_time = time;
```

```

end when;
end IntervalMeasure;

```

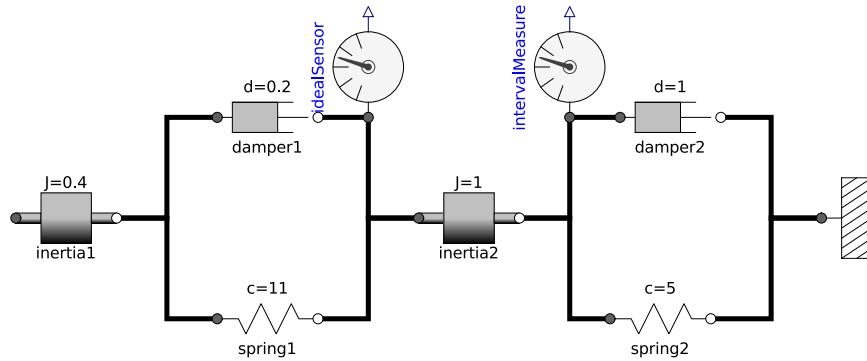
Adding this sensor to our plant model is as simple as creating the following Modelica model:

```

within ModelicaByExample.Components.SpeedMeasurement.Examples;
model PlantWithIntervalMeasure
  "Comparison between ideal and an interval measuring sensor"
  extends Plant;
  Components.IntervalMeasure intervalMeasure(teeth=200)
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}}, rotation=90,
      origin={20,30})));
  equation
    connect(intervalMeasure.flange, inertia1.flange_b) annotation (Line(
      points={{20,20},{20,0},{10,0}}, color={0,0,0},
      smooth=Smooth.None));
    annotation (experiment(StopTime=10, Tolerance=1e-006));
  end PlantWithIntervalMeasure;

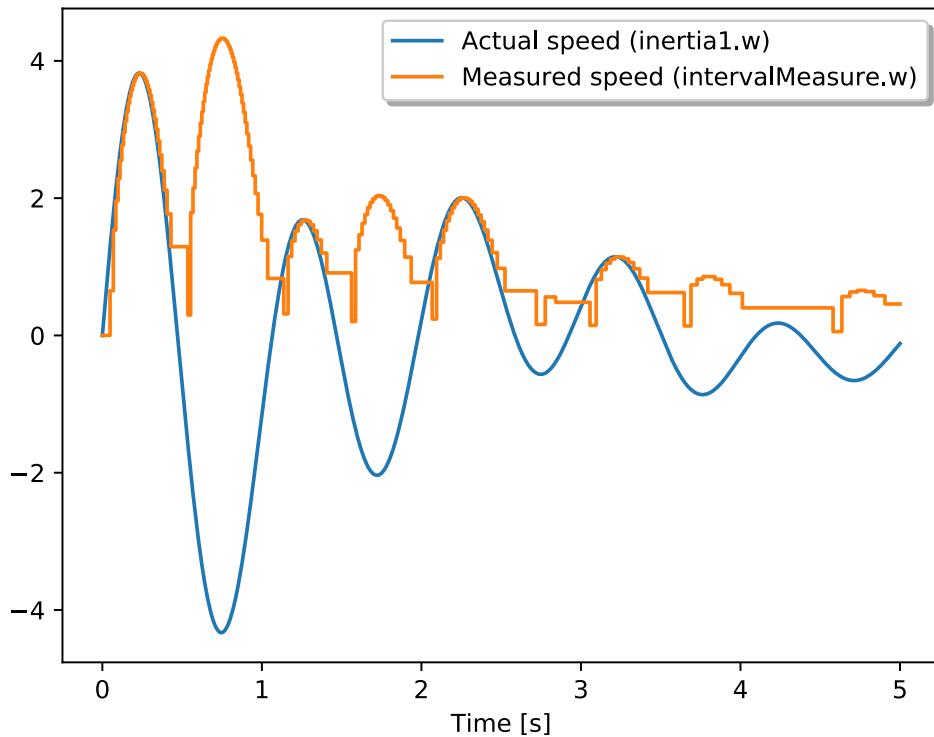
```

When assembled, our system model looks like this:

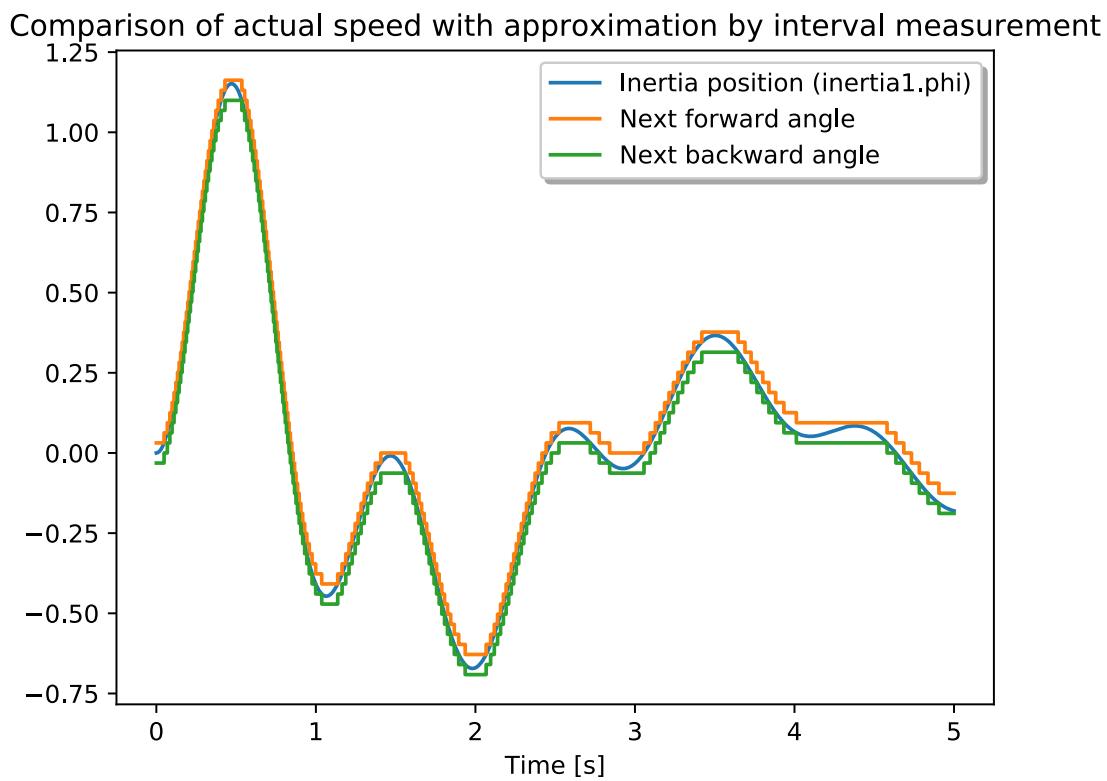


Simulating this system, we get the following results for estimated speed:

Comparison of actual speed with approximation by interval measurement

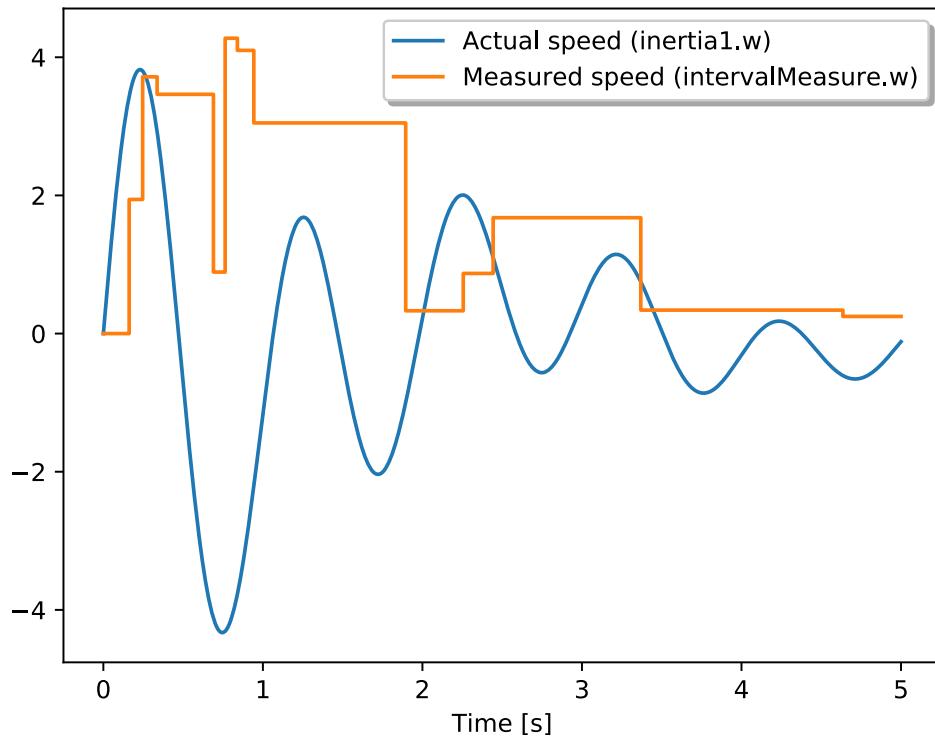


As we saw in our previous discussion of the *Interval Measurement* (page 66) technique, the quality of the estimated signal is severely reduced if we reduce the number of teeth. The previous plot used a sensor with 200 teeth per rotation. If we plot the shaft angle with respect to the bracketing teeth angles, we see that the shaft cannot move very far without triggering a measurement:



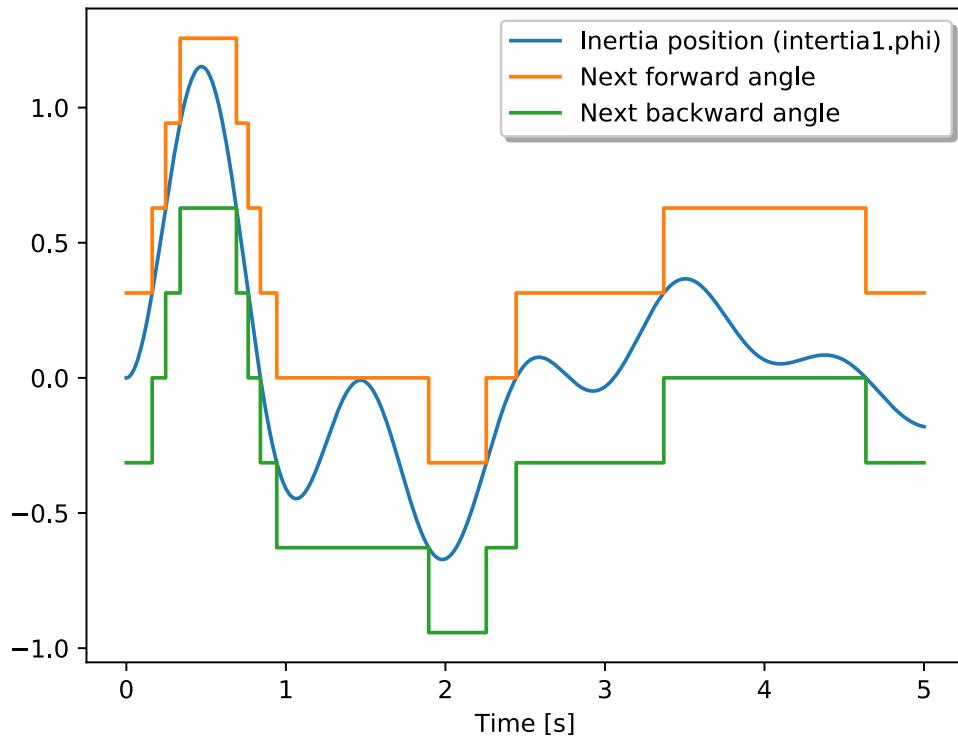
On the other hand, if we reduce the number of teeth per rotation down to 20, we get the following results:

Comparison of actual speed with approximation by interval measurement



Plotting the teeth angles that bracket the current shaft angle, we see that crossings are far less frequent, and, as a result the accuracy of the measurement is greatly reduced:

Comparison of actual speed with approximation by interval measurement



Again, we can validate our component-oriented sensor implementations by noting that these results are identical to the results presented during our previous discussion of the *Interval Measurement* (page 66) technique.

Pulse Counter

Finally, we have the *Pulse Counting* (page 69) approach. Again, the estimation technique can be wrapped in a reusable component that extends from the base `Sensor` model:

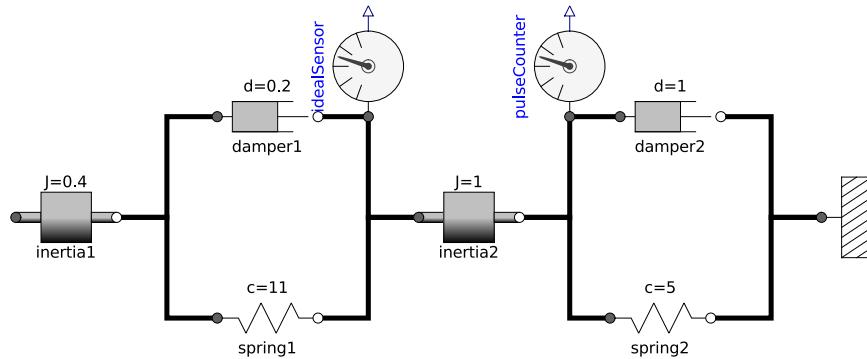
```
within ModelicaByExample.Components.SpeedMeasurement.Components;
model PulseCounter "Compute speed using pulse counting"
  extends Interfaces.SpeedSensor;
  parameter Modelica.SIunits.Time sample_time;
  parameter Integer teeth;
  Modelica.SIunits.Angle next_phi, prev_phi;
  Integer count;
protected
  parameter Modelica.SIunits.Angle tooth_angle=2*Modelica.Constants.pi/teeth;
initial equation
  next_phi = flange.phi+tooth_angle;
  prev_phi = flange.phi-tooth_angle;
  count = 0;
algorithm
  when {flange.phi>=pre(next_phi), flange.phi<=pre(prev_phi)} then
    next_phi := flange.phi + tooth_angle;
    prev_phi := flange.phi - tooth_angle;
    count := pre(count) + 1;
  end when;
  when sample(0,sample_time) then
```

```
w := pre(count)*tooth_angle/sample_time;
count := 0 "Another equation for count?";
end when;
end PulseCounter;
```

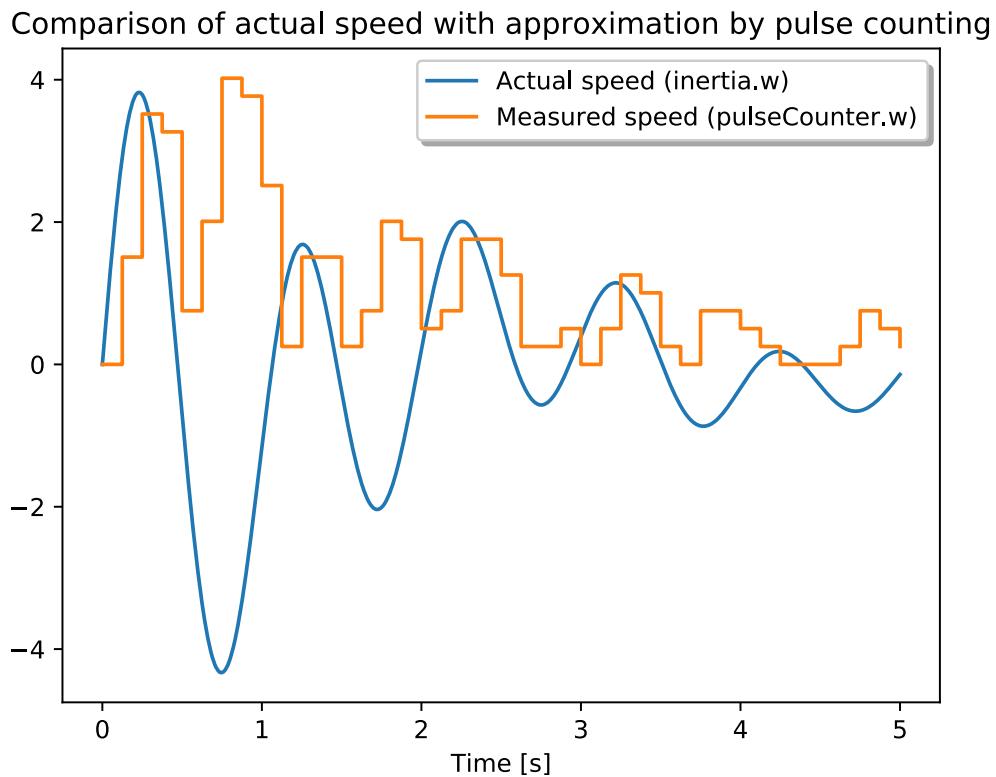
and then added to our overall Plant model:

```
within ModelicaByExample.Components.SpeedMeasurement.Examples;
model PlantWithPulseCounter
  "Comparison between ideal and pulse counting sensor"
  extends Plant;
  Components.PulseCounter pulseCounter(sample_time=0.125, teeth=200)
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}}, rotation=90,
      origin={20,30})));
  equation
    connect(pulseCounter.flange, inertia1.flange_b) annotation (Line(
      points={{20,20},{20,0},{10,0}}, color={0,0,0},
      smooth=Smooth.None));
    annotation (experiment(StopTime=10, Tolerance=1e-006));
  end PlantWithPulseCounter;
```

The resulting system, when rendered, looks like this:



Simulating the system, we see that the results are the same as in our previous discussion of [Pulse Counting](#) (page 69):



Conclusion

The discussion of *Speed Measurement* (page 63) earlier in the book went into great detail about the various measurement techniques that can be used to estimate the speed of a rotating shaft. The purpose of this section was not to revisit that discussion but rather to show how the estimation techniques presented earlier can benefit greatly from the component-oriented features in Modelica. As we have seen, all of these techniques can be nicely encapsulated in component models. The result is that utilizing these different estimation techniques is now as easy as dragging and dropping one of these sensor models into a system and attaching it to the rotating element whose speed you wish to measure.

Block Diagram Components

So far, the focus of this chapter has been on acausal modeling. But Modelica also supports causal formalisms. The main reason for the emphasis on acausal modeling is that it lends itself very well to the modeling of physical systems. It enables models of physical systems to be assembled schematically rather than mathematically while also automatically formulating conservation equations to ensure proper bookkeeping.

Block diagrams are a different way of approach modeling. The emphasis in block diagram is on creating component models that represent a wide range of mathematical operations. The operations are then performed on (generally time-varying) signals and yield, in turn, other signals. In fact, we will introduce a special kind of `model` in this section, called a `block`, that is restricted to having only `input` and `output` signals.

In this section, we'll first look at how to construct causal blocks representing some basic mathematical operations. We'll then see how those blocks can be used in two different ways. The first way will be to model a simple physical system. We'll include some discussion contrasting the causal and acausal

approaches. The second way to use such blocks is to model control systems. As we'll see, using blocks to construct control systems is a much more natural fit for the block diagram formalism.

Fortunately, Modelica allows both causal and acausal approaches and, as we'll see shortly, even allows them to be mixed. The result is that Modelica allows the model developer to use whichever approach works best in a given context.

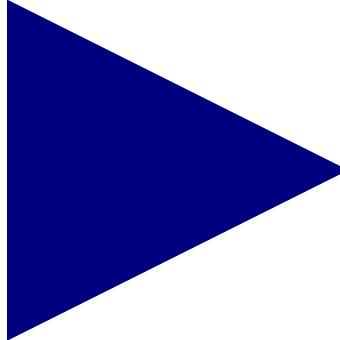
Blocks

Modelica Standard Library

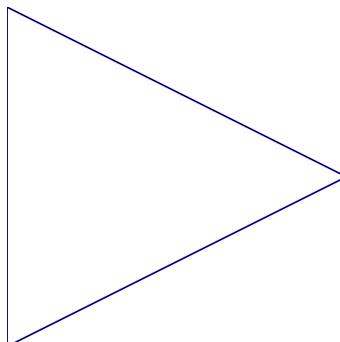
In this section, we will leverage several definitions from the Modelica Standard Library starting with the connectors:

```
connector RealInput = input Real "'input Real' as connector";
connector RealOutput = output Real "'output Real' as connector";
```

As the names suggest, `RealInput` and `RealOutput` are connectors for representing real valued input and output signals respectively. When drawn in a diagram, the `RealInput` connector takes the form of a blue solid triangle:



The `RealOutput` connector is a blue triangle outline:



We will leverage the Modelica Standard Library for several different `partial` block definitions. The first `partial` definition we'll use is the `S0`, or “single output”, definition:

```
partial block S0 "Single Output continuous control block"
  extends Modelica.Blocks.Icons.Block;

  RealOutput y "Connector of Real output signal" annotation (Placement(
    transformation(extent={{100,-10},{120,10}}, rotation=0)));
end S0;
```

Obviously, this definition is used for blocks that have a single output. By convention, this output signal is named y . Another definition we'll use is the **SISO** or “single input, single output” block:

```
partial block SISO "Single Input Single Output continuous control block"
  extends Modelica.Blocks.Icons.Block;

  RealInput u "Connector of Real input signal" annotation (Placement(
    transformation(extent={{-140,-20},{-100,20}}, rotation=0)));
  RealOutput y "Connector of Real output signal" annotation (Placement(
    transformation(extent={{100,-10},{120,10}}, rotation=0)));
end SISO;
```

This model adds an input signal, u , in addition to the output signal, y . Finally, for blocks with multiple inputs, the **MISO** block defines the input signal, u , as a vector:

```
partial block MISO "Multiple Input Single Output continuous control block"
  extends Modelica.Blocks.Icons.Block;
  parameter Integer nin=1 "Number of inputs";
  RealInput u[nin] "Connector of Real input signals" annotation (Placement(
    transformation(extent={{-140,-20},{-100,20}}, rotation=0)));
  RealOutput y "Connector of Real output signal" annotation (Placement(
    transformation(extent={{100,-10},{120,10}}, rotation=0)));
end MISO;
```

The parameter `nin` is used to specify the number of inputs to the block.

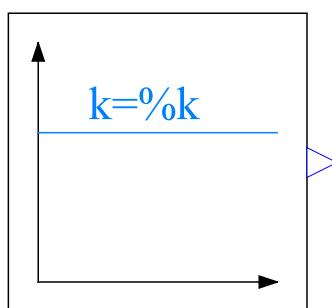
It is worth pointing out that all of the blocks we are about to define are also available in the Modelica Standard Library. But we'll create our own versions here just to demonstrate how such models can be defined.

Constant

Probably the simplest block we can imagine is one that simply outputs a constant value. Since this model has a single output, it extends from the **SO** block:

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Constant "A constant source"
  parameter Real k "Constant output value";
  extends Icons.Axes;
  extends Interfaces.SO;
equation
  y = k;
end Constant;
```

When rendered, the block looks like this:

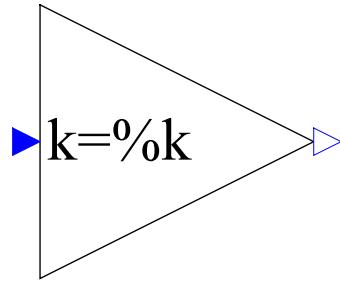


Gain

Another simple **block** is a “gain block” which multiplies an input signal by a constant to compute its output signal. Such a block will have a single input signal and a single output signal. As such, it extends from the SISO model as follows:

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Gain "A gain block model"
  extends Interfaces.SISO;
  parameter Real k "Gain coefficient";
equation
  y = k*u;
end Gain;
```

When rendered, the block looks like this:



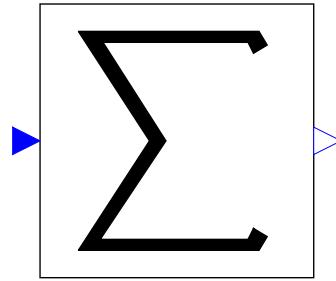
Sum

The **Sum** block can operate on an arbitrary number of input signals. For this reason, it extends from the MISO block:

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Sum "Block that sums its inputs"
  extends Interfaces.MISO;
equation
  y = sum(u);
end Sum;
```

The **Sum** block uses the *sum* (page 112) function to compute the sum over the array of input signals, *u*, to compute the output signal *y*.

When rendered, the block looks like this:

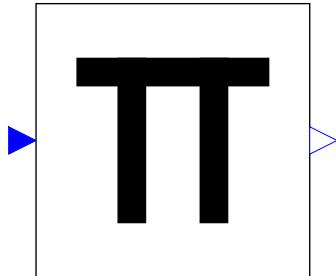


Product

The **Product** block is nearly identical to the **Sum** block except that it uses the *product* (page 112) function:

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Product "Block that outputs the product of its inputs"
  extends Interfaces.MISO;
equation
  y = product(u);
end Product;
```

When rendered, the block looks like this:



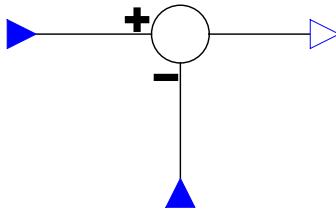
Feedback

The **Feedback** block differs from the previous blocks because it doesn't extend from any definitions in the Modelica Standard Library. Instead, it explicitly declares all of the connectors it uses:

```
within ModelicaByExample.Components.BlockDiagrams.Components;
block Feedback "A block to compute feedback terms"
  Interfaces.RealInput u1
    annotation (Placement(transformation(extent={{-120,-10},{-100,10}})));
  Interfaces.RealInput u2 annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90,
    origin={0,-110})));
  Interfaces.RealOutput y
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
equation
  y = u1-u2;
end Feedback;
```

The output of the **Feedback** block is the difference between the two **input** signals **u1** and **u2**.

When rendered, the block looks like this:



Integrator

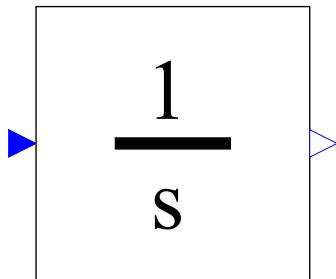
The **Integrator** block is another **SISO** block that integrates the input signal, **u**, to compute the output signal, **y**. Since this block performs an integral, it requires an initial condition which is specified using the parameter **y0**:

```

within ModelicaByExample.Components.BlockDiagrams.Components;
block Integrator
  "This block integrates the input signal to compute the output signal"
  parameter Real y0 "Initial condition";
  extends Interfaces.SISO;
initial equation
  y = y0;
equation
  der(y) = u;
end Integrator;

```

When rendered, the block looks like this:



Systems

Now that we've created this assortment of blocks, we'll explore how they can be used to model a couple of example systems. As we'll see, the suitability of causal `block` components varies from application to application.

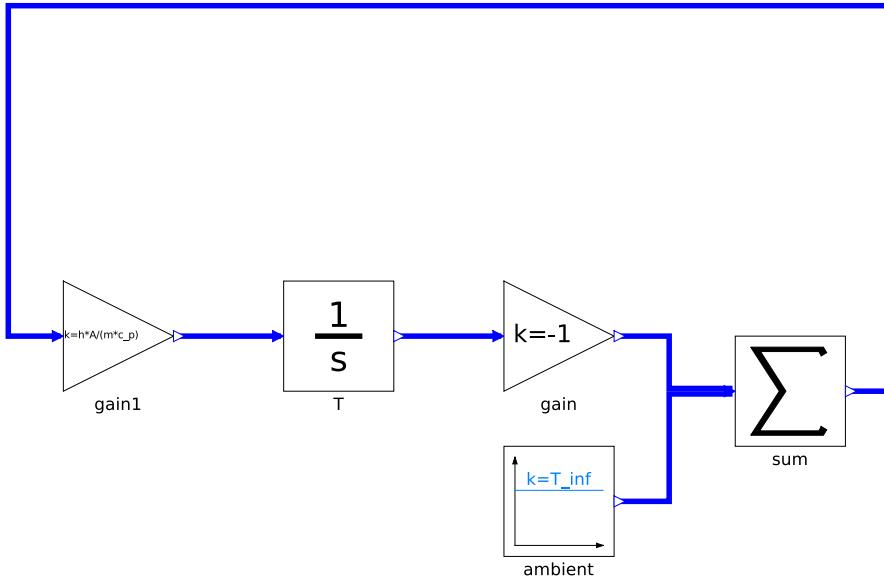
Cooling Example

The first system that we will model using our `block` definitions is the heat transfer example we presented *earlier in this chapter* (page 183). However, this time, instead of using acausal components to build our model, we'll build it up in terms of the mathematical operations associated with our `block` definitions.

Since these blocks represent mathematical operations, let us first revisit the equation associated with this example:

$$mc_p \dot{T} = hA(T_{\infty} - T)$$

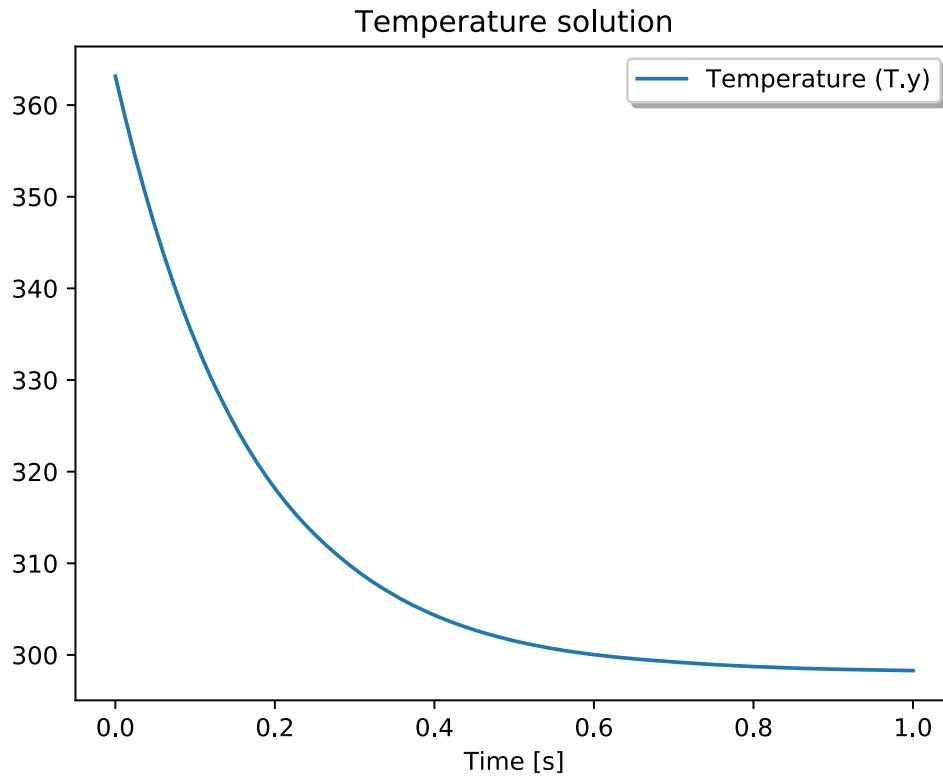
The following block diagram will solve for the temperature profile, T :



The Modelica source code for this example is:

```
within ModelicaByExample.Components.BlockDiagrams.Examples;
model NewtonCooling "Newton cooling system modeled with blocks"
  import Modelica.SIunits.Conversions.from_degC;
  parameter Real h = 0.7 "Convection coefficient";
  parameter Real A = 1.0 "Area";
  parameter Real m = 0.1 "Thermal mass";
  parameter Real c_p = 1.2 "Specific heat";
  parameter Real T_inf = from_degC(25) "Ambient temperature";
  Components.Integrator T(y0=from_degC(90))
    annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));
  Components.Gain gain(k=-1)
    annotation (Placement(transformation(extent={{10,-10},{30,10}}));
  Components.Constant ambient(k=T_inf)
    annotation (Placement(transformation(extent={{10,-40},{30,-20}}));
  Components.Sum sum(nin=2)
    annotation (Placement(transformation(extent={{52,-20},{72,0}}));
  Components.Gain gain1(k=h*A/(m*c_p))
    annotation (Placement(transformation(extent={{-70,-10},{-50,10}}));
equation
  connect(T.y, gain1.u) annotation (Line(
    points={{-9,0},{9,0}}, color={0,0,255},
    smooth=Smooth.None));
  connect(sum.y, gain1.u) annotation (Line(
    points={{73,-10},{80,-10},{80,60},{-80,60},{-80,0},{-71,0}},
    color={0,0,255}, smooth=Smooth.None));
  connect(gain.y, sum.u[2]) annotation (Line(
    points={{31,0},{40,0},{40,-9.5},{51,-9.5}},
    color={0,0,255}, smooth=Smooth.None));
  connect(ambient.y, sum.u[1]) annotation (Line(
    points={{31,-30},{40,-30},{40,-10.5},{51,-10.5}},
    color={0,0,255}, smooth=Smooth.None));
  connect(gain1.y, T.u) annotation (Line(
    points={{-49,0},{-31,0}},
    color={0,0,255}, smooth=Smooth.None));
end NewtonCooling;
```

The temperature, T , is represented in this model by the variable $T.y$. Simulating this system, we get the following solution for the temperature:



As we can see, the solution is exactly the same as it has been for all previous incarnations of this example.

So far, we've seen this particular problem formulated three different ways. The first formulation described the mathematical structure using a single equation. The second formulation used acausal component models of individual physical effects to represent the same dynamics. Finally, we have this most recent block diagram formulation. But the real question is, **which ones of these approaches is the most appropriate for this particular problem?**

There are really two extreme cases to consider. If we wanted to solve only this one particular configuration of this problem with a single thermal capacitance convecting heat to some infinite ambient reservoir, the *equation based version* (page 6) would probably be the best choice since the behavior of the entire problem can be expressed by the single equation:

```
m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
```

Such an equation can be typed in very quickly. In contrast, the component based versions would require the user to drag, drop and connect component models which would invariably take longer.

However, if you intend to create variations of the problem combining different modes of heat transfer, different boundary conditions, etc., then the acausal version is better. This is because while some investment is required to create the component models, they can be reconfigured almost trivially.

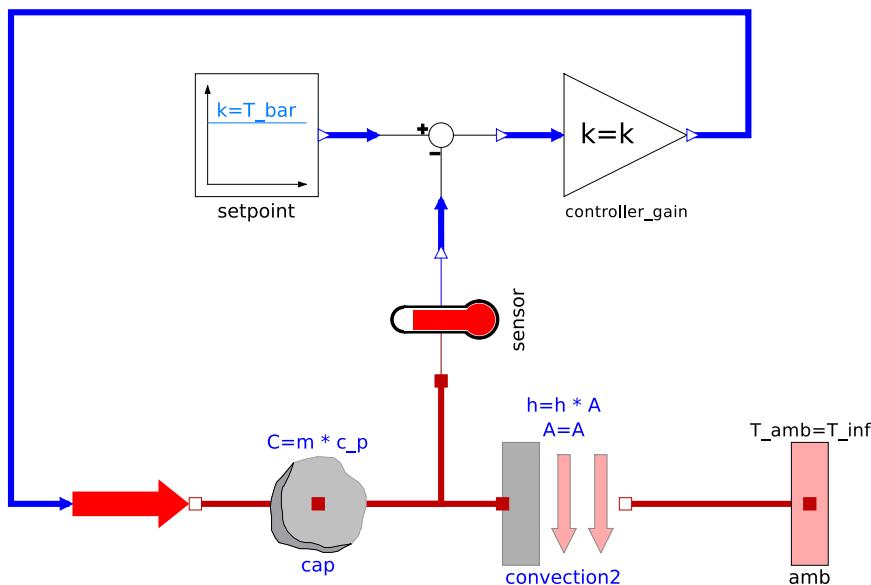
One might say the same is true for the block diagram version of the model (*i.e.*, that it can be trivially reconfigured), **but that is not the case**. The block diagram version of the model is a **mathematical** representation of the problem, not a schematic based formulation. If you create variations of this heat transfer problem that specify alternative boundary conditions, add more thermal inertias or include additional modes of heat transfer, the changes to a schematic will be simple. However, for a block diagram formulation **you will need to completely reformulate the block diagram**. This is because the resulting mathematical equations might be very different when expressed in state-space form. One of the big advantages of the acausal, schematic based approach is that the Modelica compiler will translate

the textbook equations into state-space form automatically. This saves a great deal of tedious, time-consuming and error prone work on the part of the model developer and this is precisely why the acausal approach is preferred.

Thermal Control

For the next example, we'll mix both causal components, the blocks we've developed in this section, with acausal components, the *Heat Transfer Components* (page 183) developed earlier in this chapter. This will prove to be a powerful combination, since it allows us to represent the physical components schematically, but allows us to express the control strategy mathematically.

Here is a schematic diagram showing how both approaches can be combined:



When modeling a physical system together with a control system, the physical components and effects will use an acausal formulation. The components representing the control strategy will typically use a causal formulation. What bridges the gap between these two approaches are the sensors and actuators. The sensors measure some aspect of the system (temperature in this example) and the actuators apply some “influence” over the system (in this case, a heat flux).

The actuator models will generally have signals as inputs combined with an acausal connection to the physical system (through which the “influence”, like a force or an electric current, will be applied). This is reversed for the sensor models where the causal connectors will be outputs and the acausal connectors will be used to “sense” some aspect of the physical system (like a voltage, temperature, *etc.*).

Our example model can be expressed in Modelica as:

```

within ModelicaByExample.Components.BlockDiagrams.Examples;
model MultiDomainControl
    "Mixing thermal components with blocks for sensing, actuation and control"

    import Modelica.SIunits.Conversions.from_degC;

    parameter Real h = 0.7 "Convection coefficient";
    parameter Real A = 1.0 "Area";
    parameter Real m = 0.1 "Thermal maass";
    parameter Real c_p = 1.2 "Specific heat";
    parameter Real T_inf = from_degC(25) "Ambient temperature";
    parameter Real T_bar = from_degC(30.0) "Desired temperature";

```

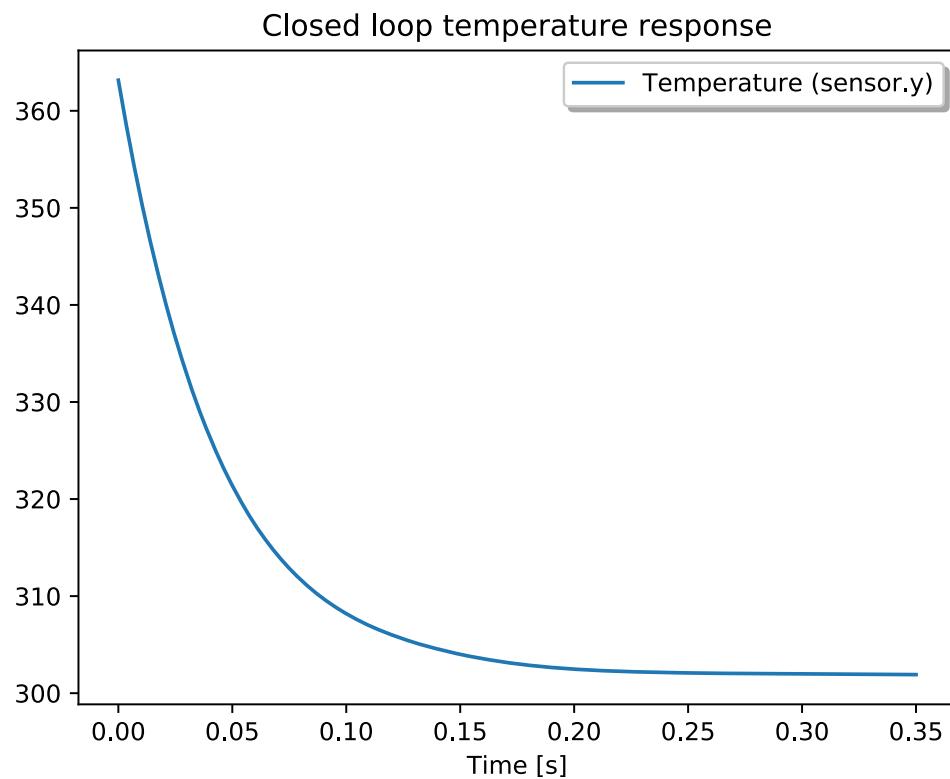
```

parameter Real k = 2.0 "Controller gain";

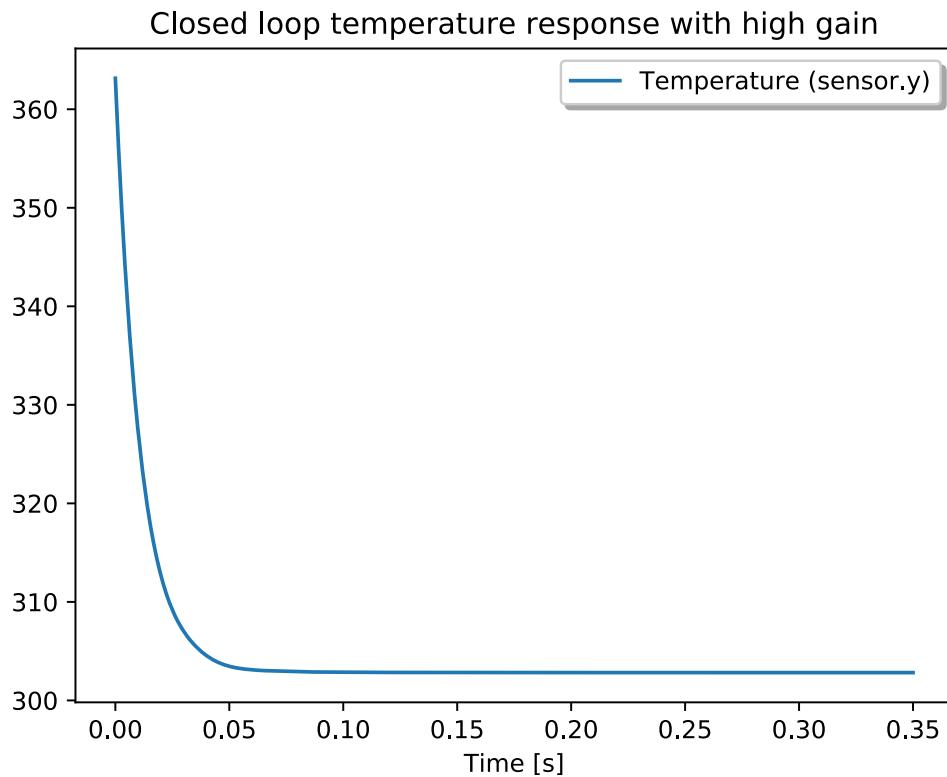
Components.Constant setpoint(k=T_bar)
  annotation (Placement(transformation(extent={{-40,30},{-20,50}})));
Components.Feedback feedback
  annotation (Placement(transformation(extent={{-10,30},{10,50}})));
Components.Gain controller_gain(k=k) "Gain for the proportional control"
  annotation (Placement(transformation(extent={{20,30},{40,50}}));
HeatTransfer.ThermalCapacitance cap(C=m*c_p, T0 = from_degC(90))
  "Thermal capacitance component"
  annotation (Placement(transformation(extent={{-30,-30},{-10,-10}}));
HeatTransfer.Convection convection2(h=h, A=A)
  annotation (Placement(transformation(extent={{10,-30},{30,-10}}));
HeatTransfer.AmbientCondition
  amb(T_amb(displayUnit="K") = T_inf)
  annotation (Placement(transformation(extent={{50,-30},{70,-10}}));
Components.IdealTemperatureSensor sensor annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  rotation=90,
  origin={0,10}));
Components.HeatSource heatSource
  annotation (Placement(transformation(extent={{-60,-30},{-40,-10}}));
equation
  connect(setpoint.y, feedback.u1) annotation (Line(
    points={{-19,40},{-11,40}}, color={0,0,255},
    smooth=Smooth.None));
  connect(feedback.y, controller_gain.u) annotation (Line(
    points={{10,40},{19,40}}, color={0,0,255},
    smooth=Smooth.None));
  connect(convection2.port_a, cap.node) annotation (Line(
    points={{10,-20},{-20,-20}}, color={191,0,0},
    smooth=Smooth.None));
  connect(amb.node, convection2.port_b) annotation (Line(
    points={{60,-20},{30,-20}}, color={191,0,0},
    smooth=Smooth.None));
  connect(sensor.y, feedback.u2) annotation (Line(
    points={{0,21},{0,24.5},{0,24.5},{0,29}}, color={0,0,255},
    smooth=Smooth.None));
  connect(heatSource.node, cap.node) annotation (Line(
    points={{-40,-20},{-20,-20}}, color={191,0,0},
    smooth=Smooth.None));
  connect(controller_gain.y, heatSource.u) annotation (Line(
    points={{41,40},{50,40},{50,60},{-70,60},{-70,-20},{-61,-20}}, color={0,0,255},
    smooth=Smooth.None));
  connect(sensor.node, cap.node) annotation (Line(
    points={{0,0},{0,0},{0,-20},{-20,-20}}, color={191,0,0},
    smooth=Smooth.None));
end MultiDomainControl;

```

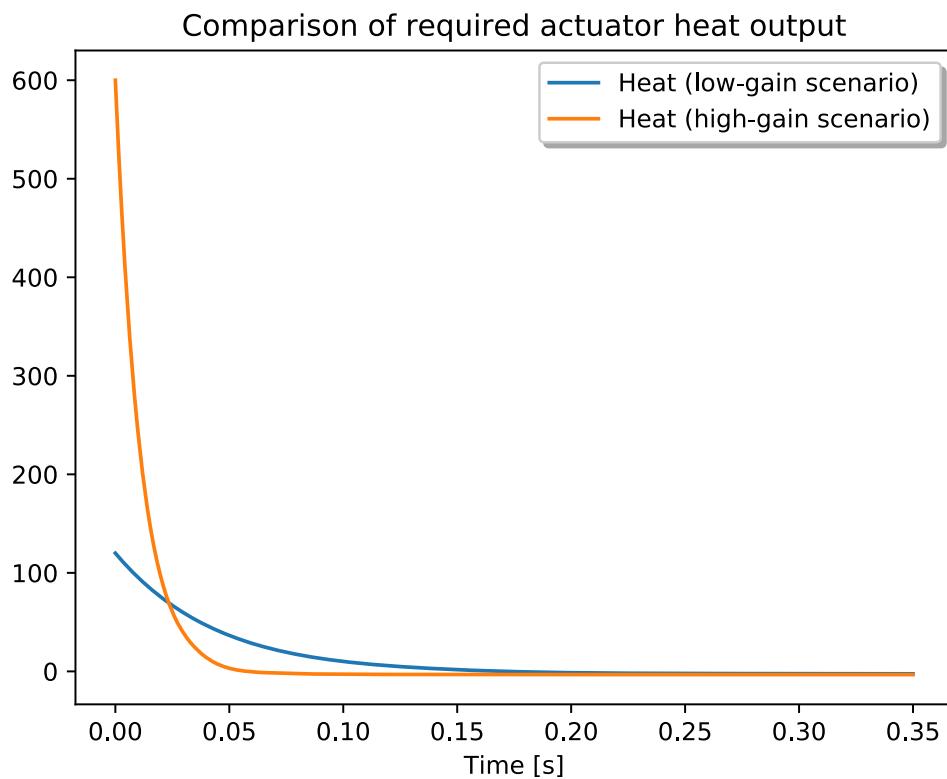
Looking at the model, we can see that the initial temperature is 90 °C and the ambient temperature is 25 °C. In addition, the setpoint temperature (the desired temperature) is 30 °C. So unlike our previous examples where the system temperature eventually came to rest at the ambient temperature, this system should approach the setpoint temperature due to the influence of the control system. Simulating this system, we get the following temperature response:



We can increase the “gain” of the controller, k , and we see a different response:



However, we can see from the following plot that much more heat output was required from our actuator in order to achieve the faster response in the second case:



This is just a very simple example of how combining physical response with control allows model developers to explore how overall system performance is impacted by both physical and control strategy design.

Conclusion

In this section, we've seen how to define causal `block` components and use them to model both the physical and control related behavior. We've even seen how these causal components can be combined with acausal components to yield a “best of both worlds” combination where control features are implemented with causal components while physical components use acausal components.

Chemical Components

For our last example of component oriented modeling, we'll return to the *Chemical System* (page 97) we discussed in the chapter on *Vectors and Arrays* (page 87). However, this time we will create component models for the various effects and show how connections in Modelica automatically ensure conservation of species.

Species

The species we will be dealing with in this example are defined by the following enumeration:

```
within ModelicaByExample.Components.ChemicalReactions.ABX;
type Species = enumeration(A, B, X);
```

Note that this definition exists within a package called `ABX`. This indicates that the component models are designed to work with this three species system involving `A`, `B` and `X`.

Mixture

Also contained in the `ABX` package is the following `connector` definition (which can be found in the `Interfaces` sub-package):

```
within ModelicaByExample.Components.ChemicalReactions.ABX.Interfaces;
connector Mixture
  Modelica.SIunits.Concentration C[Species];
  flow ConcentrationRate R[Species];
end Mixture;
```

Here we see that our `Mixture` connector uses concentrations as the across variables and a concentration rate as the flow variable. Although the `flow` variable in this case is not strictly the flow of a conserved quantity, it will suffice in this example since all reactions are contained within the same volume.

Note that both `C` and `R` in this connector are arrays where the subscript is given by an `enumeration` type. We saw earlier how *Enumerations* (page 102) can be used in this way.

Solution

Our first component model is used to track the concentration of the various chemical species within a control volume. As alluded to earlier, since all reactions occur within the same volume, we don't need to actually specify the size of the control volume.

The `Solution` model is quite simple. Like the `RegionalPopulation` model discussed *earlier in this chapter* (page 217), the rate of change of the across variable associated with its sole connector is equal to the `flow` variable on that same connector:

```
within ModelicaByExample.Components.ChemicalReactions.ABX.Components;
model Solution "A mixture of species A, B and X"
  Interfaces.Mixture mixture
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
    Modelica.SIunits.Concentration C[Species]=mixture.C
      annotation(Dialog(group="Initialization",showStartAttribute=true));
  equation
    der(mixture.C) = mixture.R;
end Solution;
```

Reactions

Reaction

As we saw previously, this system has three reactions. Each of the specific reactions we'll examine extend from the following partial model:

```
within ModelicaByExample.Components.ChemicalReactions.ABX.Interfaces;
partial model Reaction "A reaction potentially involving species A, B and X"
  parameter Real k "Reaction coefficient";
  Mixture mixture
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  protected
    ConcentrationRate consumed[Species];
    ConcentrationRate produced[Species];
    Modelica.SIunits.Concentration C[Species] = mixture.C;
  equation
```

```

consumed = -produced;
mixture.R = consumed;
end Reaction;
```

We see that each reaction has a reaction coefficient, k , and a `Mixture` connector, `mixture`, that ultimately connects it to the `Solution` where the reaction is to take place. The internal vector variables `consumed` and `produced` play a role that is similar to the `decline` and `growth` variables in the `SinkOrSource` discussed earlier in this chapter (page 219) (*i.e.*, they allow us to write contributions from individual reactions using an intuitive terminology).

A+B->X

The first complete reaction model we will consider is the one that turns one molecule of A and one molecule of B into one molecule of X. Using the `Reaction` model, we can model this reaction as follows:

```

model 'A+B->X' "A+B -> X"
  extends Interfaces.Reaction;
protected
  Interfaces.ConcentrationRate R = k*C[Species.A]*C[Species.B];
equation
  consumed[Species.A] = R;
  consumed[Species.B] = R;
  produced[Species.X] = R;
end 'A+B->X';
```

The first thing to note about this model is that it is composed of non-alphanumeric characters. Specifically, the name of this model contains +, - and >. This is permitted in Modelica as long as the name is quoted using single quote characters. The rate of the reaction, R , is used in conjunction with the `consumed` and `produced` variables inherited from the `Reaction` model to create equations that clearly describe both the reactants and the products in this reaction.

A+B<-X

The next reaction we will consider is one that takes one molecule of X and transforms it (back) into one molecule of A and one molecule of B. This is the reverse of the previous reaction. The Modelica code for this reaction would be:

```

model 'A+B<-X' "A+B <- X"
  extends Interfaces.Reaction;
protected
  Interfaces.ConcentrationRate R = k*C[Species.X];
equation
  produced[Species.A] = R;
  produced[Species.B] = R;
  consumed[Species.X] = R;
end 'A+B<-X';
```

Again, the equations convey clearly which species are reactants (*i.e.*, are consumed in the reaction) and which are the products (*i.e.*, those species that are produced in the reaction).

X+B->T+S

Finally, our last reaction converts molecules of X and B into molecules of T and S:

```

model 'X+B->T+S' "X+B->T+S"
  extends Interfaces.Reaction;
protected
```

```

Interfaces.ConcentrationRate R = k*C[Species.B]*C[Species.X];
equation
  consumed[Species.A] = 0;
  consumed[Species.B] = R;
  consumed[Species.X] = R;
end 'X+B->T+S';

```

We do not track the concentration of the T and S species since they are simply byproducts and do not participate in any other reactions. This model follows the same familiar pattern as before with the exception that the A species is not involved.

System

We can combine the `Solution` model along with the various reaction models as follows:

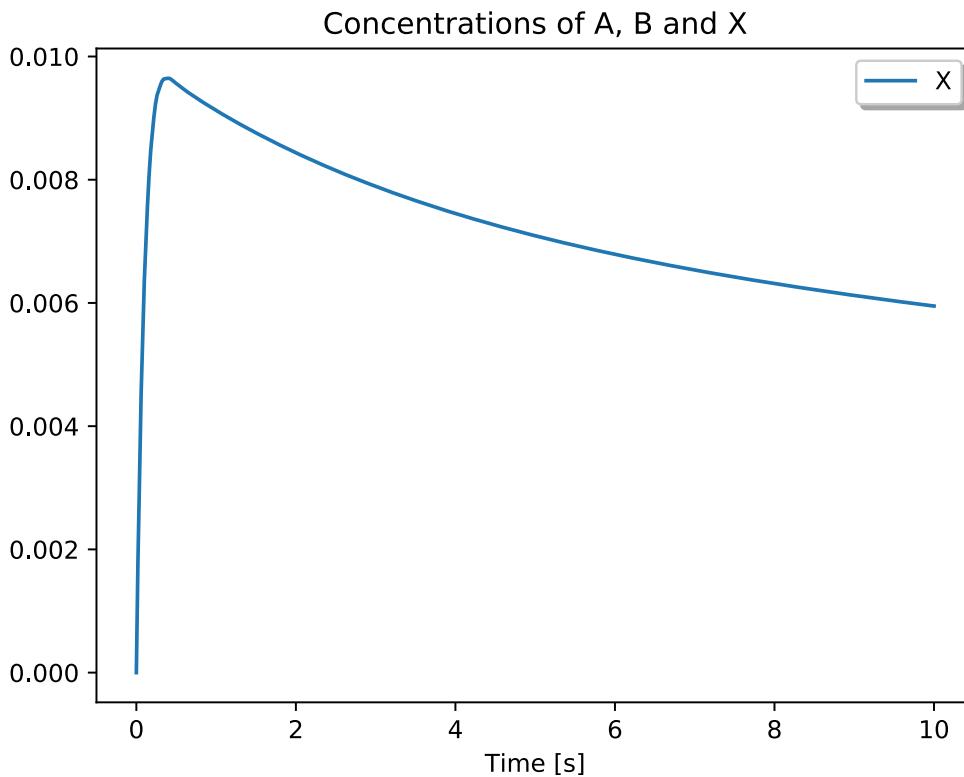
```

within ModelicaByExample.Components.ChemicalReactions.Examples;
model ABX_System "Model of simple two reaction system"
  ABX.Components.Solution solution(C(each fixed=true, start={1,1,0}));
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  ABX.Components.'A+B->X' 'A+B->X'(k=0.1)
  annotation (Placement(transformation(extent={{30,30},{50,50}})));
  ABX.Components.'A+B<-X' 'A+B<-X'(k=0.1)
  annotation (Placement(transformation(extent={{30,-10},{50,10}})));
  ABX.Components.'X+B->T+S' 'X+B->T+S'(k=10)
  annotation (Placement(transformation(extent={{30,-50},{50,-30}})));
equation
  connect('A+B<-X'.mixture, solution.mixture) annotation (Line(
    points={{30,0},{10,0}}, color={0,0,0},
    smooth=Smooth.None));
  connect('X+B->T+S'.mixture, solution.mixture) annotation (Line(
    points={{30,-40},{20,-40},{20,0},{10,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect('A+B->X'.mixture, solution.mixture) annotation (Line(
    points={{30,40},{20,40},{20,0},{10,0}},
    color={0,0,0},
    smooth=Smooth.None));
end ABX_System;

```

Note how modifications to the `solution` component are used to set the initial concentration of species within the `solution` component. Also, the reaction coefficients are specified via modifications to each of the reaction components. Finally, each of the reaction components is attached to the `solution.mixture` connector.

Simulating this system for 10 seconds yields the following concentration trajectories:



Conclusion

From our earlier discussion of this chemical system, you may recall that the resulting system of equations was:

$$\begin{aligned}\frac{d[A]}{dt} &= -k_1[A][B] + k_2[X] \\ \frac{d[B]}{dt} &= -k_1[A][B] + k_2[X] - k_3[B][X] \\ \frac{d[X]}{dt} &= k_1[A][B] - k_2[X] - k_3[B][X]\end{aligned}$$

Each equation represents the accumulation of a particular species and each term on the right hand side of those equations is computing the net flow of that particular species into the control volume. Constructing this system by hand for even a relatively small number of participating species is rife with opportunities to introduce errors. By using a component oriented approach instead, we never had to assemble such a system of equations. As a result, these equations were generated automatically. By automating this process, we can avoid many potential errors and the time required to identify and fix them.

2.3.2 Review

Component Models

In this section, we'll summarize how component models are different from the previous models we've created. This discussion will be broken into two parts. The first part will focus on acausal modeling and how it provides a framework for schematic-based, component-oriented modeling where conservation equations are automatically generated and enforced. The second part will provide an overview of how the topics in this chapter impact, mostly syntactically, the definition of component models.

However, before we dive into that discussion, it is worth taking some time to talk about terminology. In this chapter, we've created two different types of models. The first type represent individual effects (*e.g.*, resistors, capacitors, springs, dampers). The other type represent more complex assemblies (*e.g.*, circuits, mechanisms).

Before we discuss some of the differences between these different types of models, let's introduce some terminology so we can refer to them precisely. A *component model* is a model that is used to encapsulate equations into a reusable form. By creating such a model, an instance of the component can be used in place of the equations it contains. A *subsystem model* is a model that is composed of components or other subsystems. In other words, it doesn't (generally) include equations. Instead, it represents an assembly of other components. Typically, these subsystem models are created by dragging, dropping and connecting component and other subsystem models schematically. While component models are "flat" (they don't contain other components or subsystems, only equations), subsystem models are hierarchical.

We'll often refer to a subsystem model as a *system model*. A system model is a model that we expect to simulate. In simulating it, the Modelica compiler will traverse the hierarchy of the model and note all the variables and equations present throughout the hierarchy. These are the variables and equations that will be used in the simulation. Of course, in order for there to be a unique solution, the system model (like any non-partial model), must be *balanced* (page 257).

Note that a subsystem model *can* include equations. There is no rule against it in Modelica. But most of the time models tend to be composed either of equations or other components/subsystems. It is actually a good idea to avoid putting equations in models containing subcomponents or subsystems because doing so means that some information about the model will be "invisible" when looking at a diagram of the subsystem. One possible exception to this could be the use of `initial equation` sections in subsystems.

With that discussion of terminology out of the way, let's dive into discussions about component models.

Acausal Modeling

We'll start with a discussion about acausal modeling. We touched on this topic very briefly in the chapter on *Connectors* (page 169). Here we will provide a more comprehensive discussion about acausal modeling.

Composability

There are two very big advantages to acausal modeling. The first is composability. In this context, composability means the ability to drag, drop and connect component instances in nearly any configuration we wish without having to concern ourselves with "compatibility". This is because acausal connectors are designed around the idea of physical compatibility, not causal compatibility. This is possible because acausal connector definitions focus on physical information exchanged, not the direction that information flows. The result is that we can create component models around the idea of physical interactions without requiring any *a priori* knowledge about the nature (*i.e.*, directionality) of the information exchange.

But there are other implications to this composability. Not only can we easily create systems by dragging, dropping and connecting components, but we can also easily reconfigure them. Replacing a voltage source in an electrical circuit with a current source can have a profound impact on the mathematical representation of that system (*e.g.*, if the system is represented as a block diagram). But such a change has no significant impact when using an acausal approach. Although the underlying mathematical representation still changes, sometimes profoundly, there is no impact on the user, because that representation is generated automatically as part of the compilation process.

Finally, another aspect of composability is in the support for multi-domain systems. In fact, Modelica not only supports different engineering domains (electrical, thermal, hydraulic), it supports multiple modeling formalisms. Model developers have created libraries for block diagrams, state charts, petri nets, *etc.* Instead of requiring special tools or editors in each case, all of these different domains and formalisms can be freely combined in Modelica as appropriate.

Accounting

Connectors

The other advantage of acausal modeling is the amount of automatic “accounting” performed with this approach. To understand exactly what accounting is performed, let’s consider the following rotational connector definitions from the Modelica Standard Library:

```
connector Flange_a "1-dim. rotational flange of a shaft (filled square icon)"
  Modelica.SIunits.Angle phi "Absolute rotation angle of flange";
  flow Modelica.SIunits.Torque tau "Cut torque in the flange";
  annotation(Icon(/* Filled gray circle */));
end Flange_a;

connector Flange_b "1-dim. rotational flange of a shaft (filled square icon)"
  Modelica.SIunits.Angle phi "Absolute rotation angle of flange";
  flow Modelica.SIunits.Torque tau "Cut torque in the flange";
  annotation(Icon(/* Gray circular outline */));
end Flange_b;
```

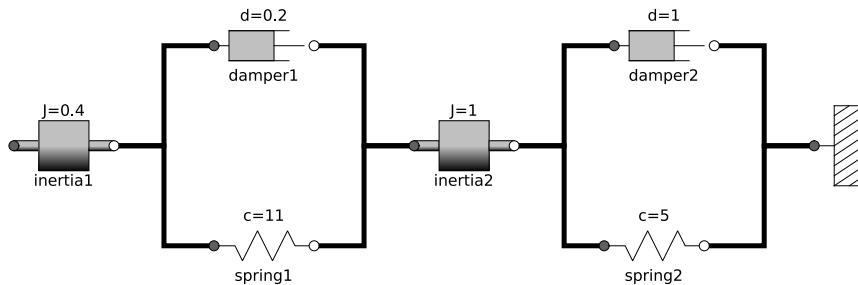
As we’ve discussed previously, an acausal connector includes two different types of variables, across variables and through variables. The through variable is indicated by the presence of the `flow` qualifier. In the case of the Rotational connector, the across variable is `phi`, the angular position, and the through variable is `tau`, the torque.

Sign Conventions

Also recall from our previous discussion that Modelica models should observe the following convention: a positive value for the `flow` variable on a connector represents the flow of that quantity **into** the component that the connector is connected to. This is an important sign convention not only because it make sure all the accounting is correct, but it also helps with composability as well by allowing (inherently symmetric) components like springs, dampers, *etc.* to be flipped over and still function identically.

Connection Sets

Before we can get into the details of the accounting performed by the compiler, we need to introduce the concept of a *connection set*. To demonstrate what a connection set is, consider the following schematic:



Note that there are 8 connections in this model:

```
equation
  connect(ground.flange_a, damper2.flange_b);
  connect(ground.flange_a, spring2.flange_b);
  connect(damper2.flange_a, inertia2.flange_b);
```

```

connect(spring2.flange_a, inertia2.flange_b);
connect(inertia2.flange_a, damper1.flange_b);
connect(inertia2.flange_a, spring1.flange_b);
connect(damper1.flange_a, inertia1.flange_b);
connect(spring1.flange_a, inertia1.flange_b);

```

If two connect statements have one connector in common, **they belong to the same connection set**. If a connector is not connected to any other connectors, then it belongs to a connection set that includes only itself. Using this rule, we can organize the connectors into connection sets as follows:

- Connection Set #1
 - ground.flange_a
 - damper2.flange_b
 - spring2.flange_b
- Connection Set #2
 - damper2.flange_a
 - spring2.flange_a
 - inertia2.flange_b
- Connection Set #3
 - inertia2.flange_a
 - damper1.flange_b
 - spring1.flange_b
- Connection Set #4
 - inertia1.flange_b
 - damper1.flange_a
 - spring1.flange_a
- Connection Set #5
 - inertia1.flange_a

Note that these connection sets appear from right to left in the diagram. It may be useful to take the time to match the connectors in the diagram with those listed in the connection sets to understand what a connection set intuitively is. Note that the `flange_a` connectors are filled circles whereas the `flange_b` ones are only outlined.

Generated Equations

This is where the “accounting” starts. For each connection **set**, special equations are automatically generated. The first set of automatic equations are related to the across variables. We need to impose the constraint, mathematically speaking, that all across variables must have the same value. Furthermore, we also introduce an equation that states that the sum of all through variables in the connection set must sum to zero.

In the case of the connection sets above, the following equations will be automatically generated:

```

// Connection Set #1
// Equality Equations:
ground.flange_a.phi = damper2.flange_b.phi;
damper2.flange_b.phi = spring2.flange_b.phi;
// Conservation Equation:
ground.flange_a.tau + damper2.flange_b.tau + spring2.flange_b.tau = 0;

```

```

// Connection Set #2
// Equality Equations:
damper2.flange_a.phi = spring2.flange_a.phi;
spring2.flange_a.phi = inertia2.flange_b.phi;
// Conservation Equation:
damper2.flange_a.tau + spring2.flange_a.tau + inertia2.flange_b.tau = 0;

// Connection Set #3
// Equality Equations:
inertia2.flange_a.phi = damper1.flange_b.phi;
damper1.flange_b.phi = spring1.flange_b.phi;
// Conservation Equation:
inertia2.flange_a.tau + damper1.flange_b.tau + spring1.flange_b.tau = 0;

// Connection Set #4
// Equality Equations:
inertia1.flange_b.phi = damper1.flange_a.phi;
damper1.flange_a.phi = spring1.flange_a.phi;
// Conservation Equation:
inertia1.flange_b.tau + damper1.flange_a.tau + spring1.flange_a.tau = 0;

// Connection Set #5
// Equality Equations: NONE
// Conservation Equation:
inertia1.flange_a.tau = 0;

```

Note that for an empty connection set (*i.e.*, Connection Set #5), there is only one across variable in the set, so no equality equations are generated. The conservation equation is still generated but it contains only one term. So it amounts to a statement that nothing can flow out of an unconnected connector. This makes intuitive physical sense as well.

What does all this mean physically? In the case of an electrical connection, this implies that each connection can be treated as a “perfect short” between the connectors. In the case of a mechanical system, connections are treated as perfectly rigid shafts with zero inertia. The bottom line is that a connection means that the across variables on each connector will be equal and that any conserved quantity that leaves one component must enter another one. Nothing can get lost or stored between components.

Conservation

There are two important consequences to these equations. The first is that the `flow` variable is automatically conserved. Typical `flow` variables are current, torque, mass flow rate, etc. Since these are all the time derivative of a conserved quantity (*i.e.*, charge, angular momentum and mass, respectively), such equations are automatically conserving these quantities.

But something else is being implicitly conserved as well. Specifically, **we can ensure that energy is conserved** as well. For all of these domains, the power flow through a connector can be represented by the product of the through variable and either the across variable or a derivative of the across variable. As a result, for each domain we can easily derive a power conservation equation from the equations automatically generated for the connection set. From our example above, we know that in the first connection set we have the following equations:

```

ground.flange_a.phi = damper2.flange_b.phi;
damper2.flange_b.phi = spring2.flange_b.phi;
ground.flange_a.tau + damper2.flange_b.tau + spring2.flange_b.tau = 0;

```

If we multiply the last equation by `der(ground.flange_a.phi)`, the angular velocity of the `ground.flange_a` connector, we get:

```
der(ground.flange_a.phi)*ground.flange_a.tau
+ der(ground.flange_a.phi)*damper2.flange_b.tau
+ der(ground.flange_a.phi)*spring2.flange_b.tau = 0;
```

However, we also know that all the across variables in the connection set are equal. As a result, their derivatives must also be equal. This means that we can substitute any one of them for another. Making two such substitutions yields:

```
der(ground.flange_a.phi)*ground.flange_a.tau
+ der(damper2.flange_b.phi)*damper2.flange_b.tau
+ der(spring2.flange_b.phi)*spring2.flange_b.tau = 0;
```

The first term in the equation above is the power flowing into the `ground` component through `flange_a`. The second term is the power flowing into the `damper2` component through `flange_b`. The last term is the power flowing into the `spring2` component through `flange_b`. Since these represent the power flowing through all connectors in the connection set, this implies that power is conserved by that connection set (*i.e.*, all power that flows out of one component must flow into another, nothing is lost or stored).

Balanced Components

If we look carefully at the previous discussion on equations generated involving acausal variables in connection sets, we'll see something very interesting. But to see it, we first need to review a few things we've learned about connectors and connector sets:

1. A connection can only belong to one connection set.
2. As we learned in our previous discussion on *Acausal Variables* (page 178), for every through variable in a connector (*i.e.*, a variable declared with the `flow` qualifier), there must be a matching across variable (*i.e.*, a variable without any qualifier).
3. The number of equations generated in a connection set is equal to the number of connectors in the connection set multiplied by the number of through-across pairs in the connector.

Remember that acausal variables come in pairs. Equations for half of those variables (one per pair) will be generated automatically via connections. That means the remaining half of the equations must come from the component models themselves.

Keep in mind that this discussion is focused only on acausal variables in connectors. We also need to take into account two other cases:

1. Variables declared within a component model (as opposed to on a connector).
2. Causal variables on connectors (*i.e.*, those qualified by either `input` or `output`).

Modelica requires that any non-partial model be balanced. But what does that mean? It means that the component should provide the proper number of equations (no more and no less than necessary). The question is how to compute the number of equations required?

We already have a start based on our discussion about acausal variables. Since half of the equations needed for acausal variables come from generated equations, the other half must come from within component models containing these connectors. Specifically, the component must provide one equation for every through-across pair in each of its connectors. In addition, it should also provide one equation for every variable on its connectors that has the `output` qualifier (note, the component does not have to provide equations for any variables on its connectors with the `input` qualifier). The rationale here is that a component can assume that all `input` signals are known (specified externally) and that it is responsible for computing any `output` signals it advertises. Finally, any (non-parameter) variable declared within the component must also have an equation.

In summary, the number of equations that a component must provide is the sum of:

1. The number of through-across pairs across all its connectors
2. The number of non-parameter variables declared in the component model.

3. The number of **output** variables across all its connectors.

Note that these equations can (and frequently do) originate in a **partial** model that is inherited.

If the number of equations provided by a component equals the number of equations required, then the component model is said to be **balanced**.

Component Definitions

In this chapter we've discussed how to create component models. Fundamentally, nothing has changed since we first discussed what a *Model Definition* (page 27) should include. But it is worth emphasizing a few things about component models.

Blocks

First, in the discussion on *Block Diagram Components* (page 236) we introduced the idea of a **block**. A **block** is a special kind of **model** where the connectors contain only **input** and **output** signals.

Conditional Variables/Connectors

Another thing we saw in our discussion of the *Optional Ground Connector* (page 214) was the ability to make a declaration conditional. The expression on which the conditional declaration depends cannot change as a function of time (*i.e.*, the variable cannot appear and disappear during the simulation). Instead, it must be a function of parameters and constants so that the compiler or simulation runtime can determine whether the variable should be present prior to simulation. As we saw, the syntax for such a declaration is:

```
VariableType variableName(/* modifications */) if conditional_expression;
```

In other words, by including the **if** keyword and a conditional expression immediately after the name of the variable (and any modifications that are applied to the variable), we can make the declaration of that variable conditional. When the conditional expression is **true**, the conditional variable will be present. When it is **false**, it will not be present.

Model Limitations

assert

To understand how to enforce model limitations, we must first explain the **assert** function. The syntax of a call to the **assert** function is:

```
assert(conditional_expression, "Explanation of failure", assertLevel);
```

where **conditional_expression** is an expression that yields either **true** or **false**. A value of **false** indicates a failure of the assertion. We'll discuss the consequences of that momentarily. The second argument must be a **String** that describes the reason that the assertion failed. The last argument, **assertLevel**, is of type **AssertionLevel** (which was defined in our previous discussion on **enumerations**). This last argument is **optional** and has the default value of **AssertionLevel.error**.

Now that we know how to use the **assert** function, let's examine the consequences of assertions during simulation to understand why they are important.

Defining Model Limitations

When creating a component `model` (or any `model`, for that matter), it is useful to incorporate any limitations on the equations in a model by including them directly in the model. This is done by adding `assert` calls in either the `equation` or `algorithm` section. As their name implies, these assertions assert that certain conditions must always be true.

If the equations within a model are only accurate or applicable under certain conditions, it is essential that these conditions be included in the model via assertions. Otherwise, the model may silently yield an incorrect solution. If not uncovered, this could lead to bad decisions based on model solutions. If it is uncovered, it will undermine the trust people have in the models. So always try to capture such model limitations.

It is worth taking a moment to understand what impact such an assertion has during simulation. Part of the simulation process is the generation of so-called *candidate solutions*. These solutions may, or may not, end up being actual solutions. They are usually generated as the underlying solvers propose solutions and then check to make sure that the solutions are accurate to within some numerical tolerance. Those candidate solutions that are found to be inaccurate are typically refined in some way until a sufficiently accurate solution is found.

If a candidate solution violates an assertion, then it is automatically considered to be inaccurate. The violated assertion will automatically trigger the refinement process in an attempt to find a solution that is more accurate and, hopefully, doesn't violate the assertion. However, if these refinement processes lead to a solution that is sufficiently accurate (*i.e.*, satisfies the accuracy requirements to within the acceptable tolerance), but that solution still violates any assertions in the system, then the simulation environment will do one of two things. If the `level` argument in the `assert` call is `AssertionLevel.error` then the simulation is terminated. If, on the other hand, the `level` argument is `AssertionLevel.warning`, then the assertion description will be used to generate a warning message to the user. How this message is delivered is specific to each simulation environment. Recall that the default value for the `level` argument (if none is provided in the call to `assert`) is `AssertionLevel.error`.

System Models

The next chapter will provide an in-depth discussion about *Subsystems* (page 264). For now, we'll only discuss the handful of topics related to subsystems that we've seen so far.

Connections

One distinction that we've seen in this chapter between component and subsystem models is that subsystem models include `connect` statements. To explore how the `connect` statement works, let's revisit the `MultiDomainControl` example from the discussion on *Thermal Control* (page 244). If we strip away all the annotations (which we will discuss shortly), we get a model that looks like this:

```
within ModelicaByExample.Components.BlockDiagrams.Examples;
model MultiDomainControl
  "Mixing thermal components with blocks for sensing, actuation and control"

  import Modelica.SIunits.Conversions.from_degC;

  parameter Real h = 0.7 "Convection coefficient";
  parameter Real m = 0.1 "Thermal maass";
  parameter Real c_p = 1.2 "Specific heat";
  parameter Real T_inf = from_degC(25) "Ambient temperature";
  parameter Real T_bar = from_degC(30.0) "Desired temperature";
  parameter Real k = 2.0 "Controller gain";

  Components.Constant setpoint(k=T_bar);
  Components.Feedback feedback;
  Components.Gain controller_gain(k=k);
```

```

HeatTransfer.ThermalCapacitance cap(C=m*c_p, T0 = from_degC(90));
HeatTransfer.Convection convection2(h=h);
HeatTransfer.AmbientCondition amb(T_amb(displayUnit="K") = T_inf);
Components.IdealTemperatureSensor sensor;
Components.HeatSource heatSource;
equation
  connect(setpoint.y, feedback.u1);
  connect(feedback.y, controller_gain.u);
  connect(convection2.port_a, cap.node);
  connect(amb.node, convection2.port_b);
  connect(sensor.y, feedback.u2);
  connect(heatSource.node, cap.node);
  connect(controller_gain.y, heatSource.u);
  connect(sensor.node, cap.node);
end MultiDomainControl;

```

During our earlier discussion on [Acausal Modeling](#) (page 253), we talked about equations that are generated for **acausal** variables in a connector. But the impact of a `connect` statement depends on the nature of the variables being connected. The `MultiDomainControl` model is useful because it isn't restricted to acausal connections.

Before we consider the specific connections in the `MultiDomainControl` model, let's first elaborate on what the `connect` statement actually does. There are some complex cases that arise, but for the sake of simplicity and pedagogy, we'll only discuss the basic case here.

A `connect` statement connects exactly two connectors. It then "pairs up" variables across each connector **by name**. In other words, it takes each variable in one connector and pairs it up with the variable of the same name in the other connector.

For each pair, the compiler first checks to make sure that the two corresponding variables have the same type (*e.g.*, `Real`, `Integer`). But what equations are generated and what additional restrictions exists depend on what qualifiers have been applied to the variables. The following list covers all the essential cases:

- **Through variables** - These are variables with the `f!ow` qualifier. As we covered in our previous discussion on [Acausal Modeling](#) (page 253), a conservation equation is generated for all variables in the connection set.
- **Parameters** - A variable that includes the `parameter` qualifier does not generate any equations. Instead, it generates an `assert` call that ensures that the values are identical between the two variables. This is useful when a connector includes `Integer` parameters that specify the size of arrays in the connector, for example, because it asserts the arrays are the same size.
- **Inputs** - A variable that has the `input` qualifier can only be paired with a variable that has an `input` or an `output` qualifier. Assuming this requirement is met, an equation will be generated simply equating these two variables.
- **Outputs** - A variable that has the `output` qualifier can only be paired with a variable that has the `input` qualifier (*i.e.*, two outputs can never be connected). As with the case for input variables, an equality relationship is generated for such a connection.
- **Across variables** - These are variables that lack any qualifiers (unlike the previous cases). As we covered in our previous discussion on [Acausal Modeling](#) (page 253), a series of equations will be generated equating all the across variables in the connection set.

In our discussion of [Block Diagram Components](#) (page 236), we describe the `input` and `output` qualifiers as "causal". In fact, the `input` and `output` qualifiers do not actually specify the order in which calculations are performed. As discussed above, they just enforce restrictions on how the variables can be connected. In addition to the restriction already mentioned, there is one additional restriction that, within a connection set, there can only be one `output` signal (for obvious reasons).

In our `MultiDomainControl` model, we can see several of these cases covered. For example,

```
connect(setpoint.y, feedback.u1);
```

Here, an output signal, `setpoint.y`, is connected to an input signal, `feedback.u1`. So this is a connection involving only causal signals. On the other hand, we have connections like this:

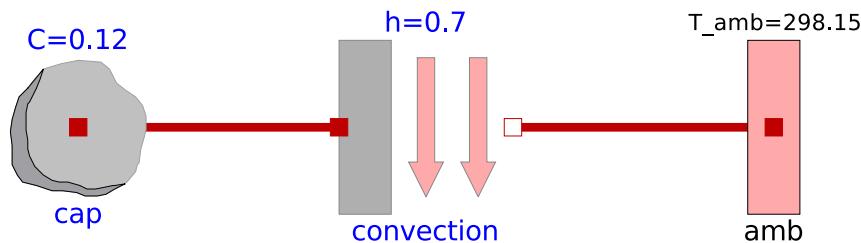
```
connect(heatSource.node, cap.node);
```

This will lead to the types of conservations equations *discussed earlier* (page 253).

In summary, a `connect` statement is a way to generate equations that automatically manages complex tasks (like generation of conservation and continuity equations) while at the same time checking to make sure that the connection makes sense (*e.g.*, that the variables have the same type).

Diagrams

In this chapter, we showed how Modelica subsystem models can be represented graphically, *e.g.*,



All the information required to generate such a diagram is contained in the Modelica model. While this information has been visible in some of the Modelica code listings in this chapter, we haven't really discussed what information is stored and where.

To render a subsystem diagram, three pieces of information are needed:

- The icon to use to represent each component.
- The location of each component.
- The path for each connection

Component Icon

The icon used for each component is simply whatever drawing primitives are included in the `Icon` annotation for that component's definition. The details of the `Icon` annotation were covered in our previous discussion of *Graphical Annotations* (page 178).

Component Placement

Now that we know what to draw for each component, we need to know where to draw it. This is where the `Placement` annotation comes in. This annotation appeared in many of the examples in this chapter, *e.g.*,

```
Convection convection(h=0.7, A=1.0)
annotation (Placement(transformation(extent={{10,-10},{30,10}})));
```

The `Placement` annotation simply establishes a rectangular region in which to draw the icon associated with each component. As with other *Graphical Annotations* (page 178), we can describe the `Placement` annotation in terms of a `record` definition:

```
record Placement
  Boolean visible = true;
  Transformation transformation "Placement in the diagram layer";
  Transformation iconTransformation "Placement in the icon layer";
end Placement;
```

The `visible` field serves the same purpose as it does in the `GraphicItem` annotation we discussed earlier, *i.e.*, it is used to control whether the component is rendered or not.

The `transformation` field defines how the icon is rendered in a schematic diagram and the `iconTransformation` defines how it is rendered if it is considered part of the `subsystem`'s icon. Generally, the `iconTransformation` is only defined for connectors since these are typically the only components that appear in the icon representation.

The `Transformation` annotation, which is defined as follows:

```
record Transformation
  Point origin = {0, 0};
  Extent extent;
  Real rotation(quantity="angle", unit="deg")=0;
end Transformation;
```

The `rotation` field indicates how many degrees the component's icon should be rotated and the `origin` field indicates the point around which this rotation should occur. Finally, the `extent` field indicates the size of the region the icon will be rendered into.

Connection Rendering

Finally, we have the third topic, rendering the connections. Again, the annotations that govern how connections are rendered have appeared in many examples. Now, finally, we'll explain what that information represents. Consider the following `connect` statement from our *Thermal Control* (page 244) example:

```
connect(controller_gain.y, heatSource.u) annotation (Line(
  points={{41,40},{50,40},{50,60},{-70,60},{-70,-20},{-61,-20}},
  color={0,0,255},
  smooth=Smooth.None));
```

Note that the `connect` statement is followed by an annotation. In particular, note that this is a `Line` annotation. We already discussed the *Line* (page 180). The annotation data is the same in this context as it was then.

Component Model Annotations

Several of the annotation that appeared in the examples from this chapter have been explained previously (*e.g.*, in our discussions on *Graphical Annotations* (page 178) and *Diagrams* (page 261)). Here we'll run through those annotations that have not yet been explained and discuss their purpose.

`defaultComponentName`

Type: Model Annotation

The `defaultComponentName` annotation is used within a model definition to define the default name that an instance of that model should have. This is used by graphical tools to assign an initial name to components when they are dragged into a diagram.

`defaultComponentPrefixes`

Type: Model Annotation

Where the `defaultComponentName` annotation defines the default name used when a component is dragged into a diagram, the `defaultComponentPrefixes` defines any **qualifiers** that should automatically be included in the declaration of the component. The value of this annotation should be a **string** that is a space separated list of the qualifiers.

When a component is instantiated, graphical tools will find the definition associated with that component and look to see if a value has been provided for the `defaultComponentPrefixes` annotation. If so, it will extract the qualifiers listed in that string and immediately add them as qualifiers to that component's declaration.

`Dialog`

Type: Declaration Annotation

The `Dialog` annotation is used to help organize variables (typically **parameters**) in the context of a graphical user interface. It provides additional information, beyond what is necessary to compile the model, that instructs graphical tools what content to include in component dialogs.

The structure of a `Dialog` annotation can be represented by the following **record** definition:

```
record Dialog
  parameter String tab = "General";
  parameter String group = "Parameters";
  parameter Boolean enable = true;
  parameter Boolean showStartAttribute = false;
  parameter String groupImage = "";
  parameter Boolean connectorSizing = false;
end Dialog;
```

The `tab` field is a string that indicates the name of the tab that this variable should be organized under. The `group` field specifies the name of a “group” **within the specified tab** in which the variable should be placed. The `enable` field should be given an expression that indicates when the parameter should be shown. The `showStartAttribute` field can be used to incorporate the `start` attributes value (for this variable) into the user interface so the user can easily specify the value of the `start` attribute. The `groupImage` field allows the user to specify an image to associate with the group named by the `group` field. Finally, the `connectorSizing` is only useful in declarations for integer parameters that are used to specify the size of arrays of connectors. In such circumstances, if the value of the `connectorSizing` field is `true`, the graphical environment **may** update the value of the annotated parameter in response to any action that impacts the necessary size of that connector.

`DynamicSelect`

Type: Declaration Annotation

The `DynamicSelect` annotation is used to specify how annotation values can depend on a simulated solution. For example, the `DynamicSelect` annotation can be used to adjust the color of a component icon in response to a change in temperature. The `DynamicSelect` has two values associated with it, *i.e.*,

```
DynamicSelect(static_value, dynamic_value)
```

The first is the “static” value. This value is used when either no simulation results are available or in the case that the specific tool does not support linking simulation results to annotations. The second value is the “dynamic” value. This is an expression, typically involving variables in the scope in which the annotated declaration appears, which is evaluated based on simulation results.

`preferredView`

Type: Definition Annotation

The `preferredView` annotation is used to describe what particular “view” of a given definition should be shown when that model is selected within a graphical tool. Possible values for this annotation are:

- “info” - Show any documentation associated with this definition.
- “text” - Show the Modelica code associated with this definition.
- “diagram” - Show the schematic diagram associated with this definition.

A common use for the `preferredView` annotation is to created a package specifically for documentation. In this case, the package includes a `Documentation` annotation and the `preferredView` annotation is set to `info` (thus causing the documentation to be shown when the definition is visited).

`unassignedMessage`

Type: Declaration Annotation

The value of the `unassignedMessage` annotation is a string. If an equation cannot be found to compute a value for the annotated declaration, the string value given to the `unassignedMessage` annotation may be presented as a diagnostic message by the compiler.

2.4 Subsystems

2.4.1 Examples

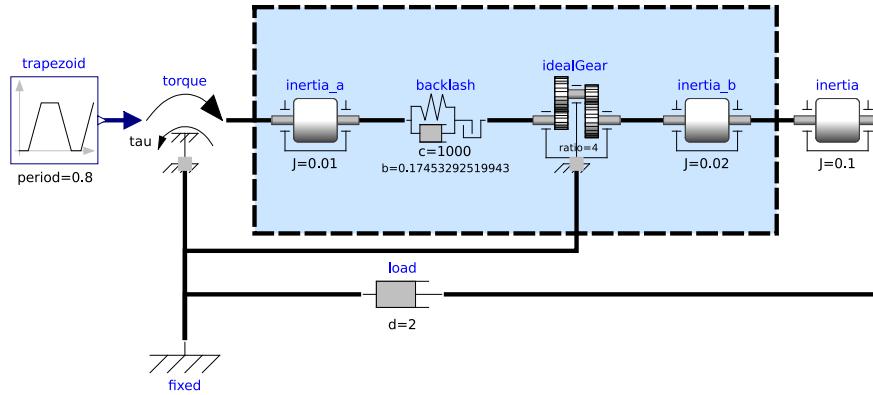
Gear Assembly

In this section, we’ll take a close look at how to model a simple gear. We’ll consider things like the inertia of each gear element, the backlash that exists between the teeth and, of course, the kinematic relationship between the two rotating shafts. We’ll first show an example how a “flat” model of such an assembly would be created and then we’ll look at how this flat model can be refactored into a reusable subsystem model that can be used across a wide ranging of applications.

We’ve mentioned several times up until now that it is usually a good idea to create component models that model just one physical effect, *e.g.*, inertia, compliance, resistance, convection, *etc.* This is true when we are modeling at the component level. But many real world subsystems are a mixture of all of these effects. The way we address this in Modelica is to build reusable subsystem models. Of course, we don’t “reinvent the wheel” by adding the equations for all these effects into our subsystem model. Instead, we reuse the component models we’ve already developed. In the end, the subsystem model ends up being nothing more than an assembly of pre-existing component models arranged in a specific configuration. Furthermore, we will show how parameters used to describe the components can be “wired up” to parameters of the subsystem.

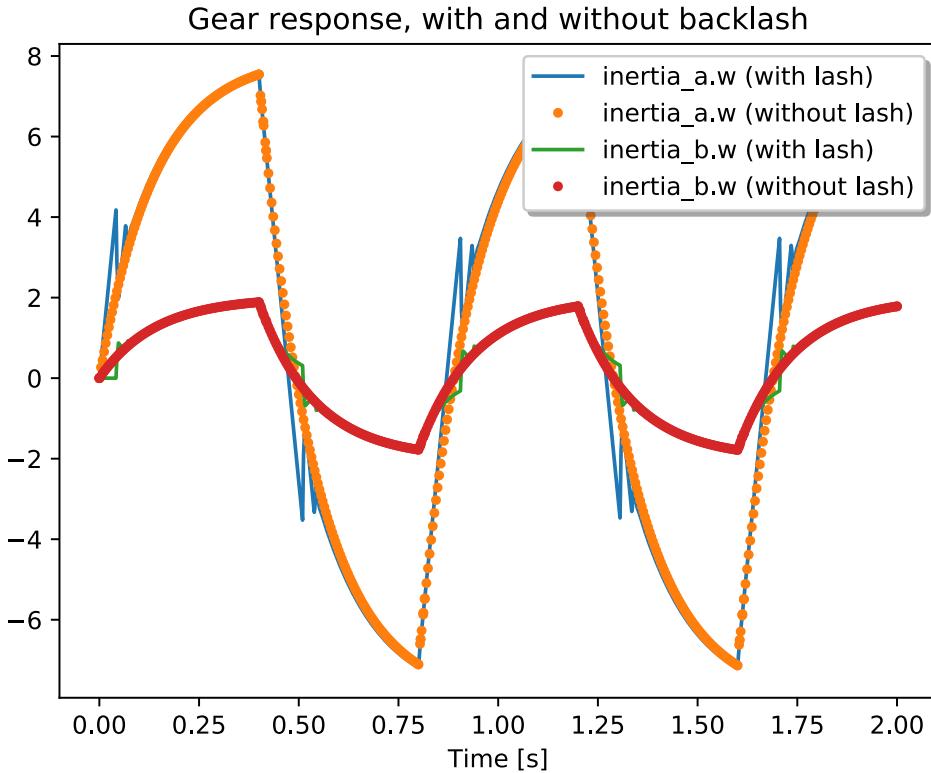
Flat Version

If we were unfamiliar with the ability to create reusable subsystem models in Modelica, we might start by building a Modelica model that looked like this one:



This model includes two essential components. Part of the model, inside the dashed line, represents how the gear itself is being modeled. It includes the inertia for each gear element, the backlash between the gear teeth and the kinematic relationship between the two shafts. Each of these is represented by an individual component model. The other part of the model, outside the dashed line, represents the specific scenario/experiment we are performing. This includes a torque profile to be applied to the gear and the downstream load that is being driven by the gear.

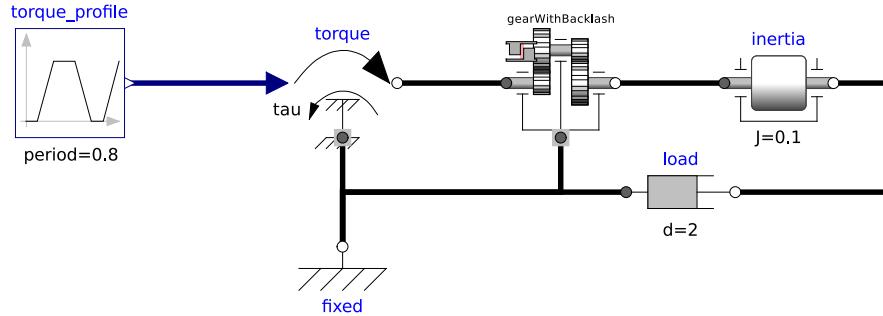
If we simulate this system, we get the following response:



The important thing to understand about this system is that the particular assembly of components inside the dashed line are likely to be repeated in any gear related application. In fact, they may be repeated multiple times in a model of something like an automotive transmission.

Hierarchical Version

So, in order to avoid redundancy (the reasons for which have already been discussed), we should create a reusable subsystem model of the components within the dashed line. In such a case, our schematic diagram would then look something like this:



In this case, the collection of components used to represent the gear are replaced by a single instance in the diagram layer. This is possible because all the component models that make up the gear model have been assembled into the following subsystem model:

```
within ModelicaByExample.Subsystems.GearSubsystemModel.Components;
model GearWithBacklash "A subsystem model for a gear with backlash"
  extends Modelica.Mechanics.Rotational.Icons.Gear;
  import Modelica.Mechanics.Rotational.Components.*;

  parameter Boolean useSupport(start=true);
  parameter Modelica.SIunits.Inertia J_a
    "Moment of inertia for element connected to flange 'a'";
  parameter Modelica.SIunits.Inertia J_b
    "Moment of inertia for element connected to flange 'b'";
  parameter Modelica.SIunits.RotationalSpringConstant c
    "Backlash spring constant (c > 0 required)";
  parameter Modelica.SIunits.RotationalDampingConstant d
    "Backlash damping constant";
  parameter Modelica.SIunits.Angle b=0
    "Total backlash as measured from flange_a side";
  parameter Real ratio
    "Transmission ratio (flange_a.phi/flange_b.phi, once backlash is cleared)";

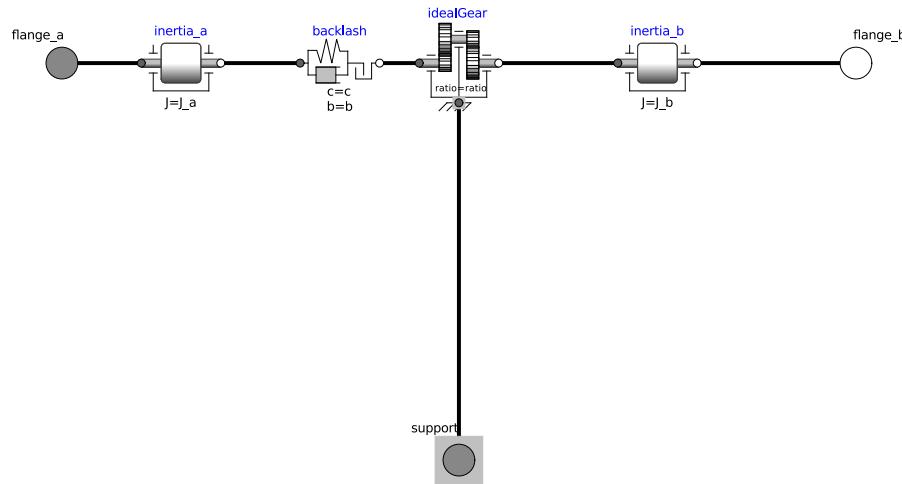
protected
  Inertia inertia_a(final J=J_a) "Inertia for the element 'a'"
    annotation (Placement(transformation(extent={{-80,-10},{-60,10}})));
  Inertia inertia_b(final J=J_b)
    annotation (Placement(transformation(extent={{40,-10},{60,10}}));
  ElastoBacklash backlash(final c=c, final d=d, final b=b) "Backlash as measured from flange_a"
    annotation (Placement(transformation(extent={{-40,-10},{-20,10}}));
  IdealGear idealGear(final useSupport=useSupport, final ratio=ratio)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}}));
public
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}}));
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
    annotation (Placement(transformation(extent={{90,-10},{110,10}}));
  Modelica.Mechanics.Rotational.Support support if useSupport
    annotation (Placement(transformation(extent={{-10,-110},{10,-90}}));
equation
  connect(flange_a, inertia_a.flange_a)
    annotation (Line(points={{-80,0},{-100,0}},
      color={0,0,0}, smooth=Smooth.None));
```

```

connect(inertia_b.flange_b, flange_b)
annotation (Line(points={{60,0},{100,0}},
    color={0,0,0}, smooth=Smooth.None));
connect(idealGear.support, support)
annotation (Line(points={{0,-10},{0,-100}},
    color={0,0,0}, smooth=Smooth.None));
connect(idealGear.flange_b, inertia_b.flange_a)
annotation (Line(points={{10,0},{40,0}},
    color={0,0,0}, smooth=Smooth.None));
connect(backlash.flange_a, inertia_a.flange_b)
annotation (Line(points={{-40,0},{-60,0}},
    color={0,0,0}, smooth=Smooth.None));
connect(backlash.flange_b, idealGear.flange_a)
annotation (Line(points={{-20,0},{-10,0}},
    color={0,0,0}, smooth=Smooth.None));
end GearWithBacklash;

```

When rendered, we see the diagram for the `GearWithBacklash` model looks like this:

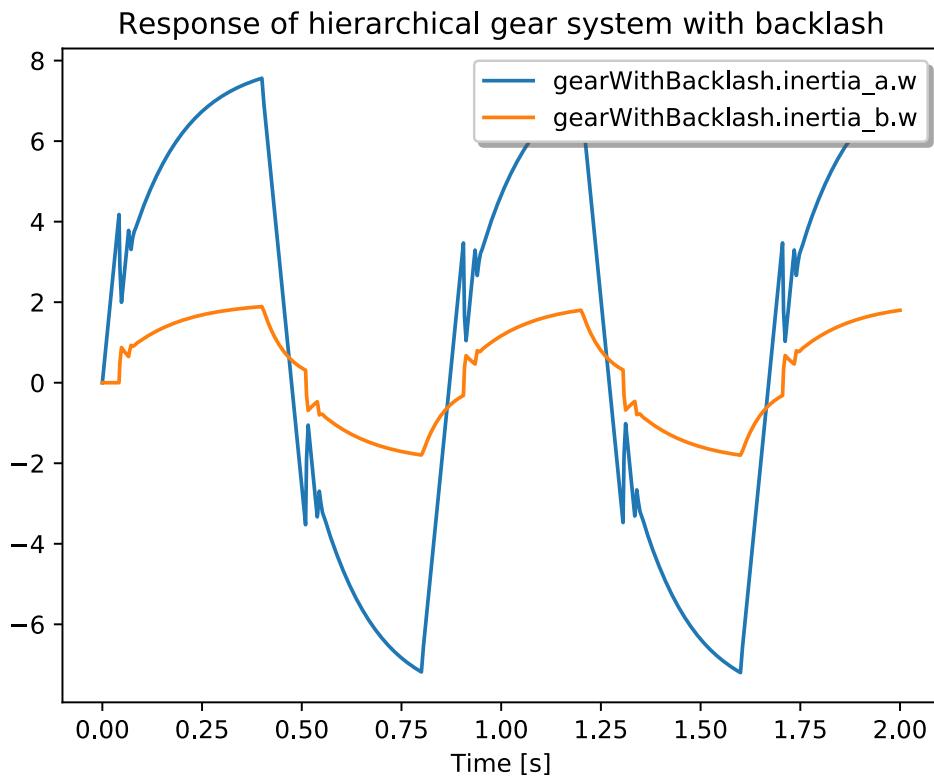


There is quite a bit going on in this model. First, note the presence of the `useSupport` parameter. This is used to determine whether to include the *Optional Ground Connector* (page 214) we discussed in the previous chapter.

Also note that all the subcomponents (`inertia_a`, `inertia_b`, `backlash` and `idealGear`) are `protected`. Only the connectors (`flange_a`, `flange_b` and `support`) and the parameters (`J_a`, `J_b`, `c`, `d`, `b`, `ratio`) are `public`. The idea here is that the only thing that the user needs to be aware of (or should even be able to access) are the connectors and the parameters. Everything else is an implementation detail. The `protected` elements of a model cannot be referenced from outside. This prevents models from breaking if the internal details (which the user should not require any knowledge of anyway) were to change.

Also note how many of the parameters, *e.g.*, `c`, are specified at the subsystem level and then assigned to parameters lower down in the hierarchy (often in conjunction with the `final` qualifier). In this way, parameters of the components can be collected at the subsystem level so users of this model will see all relevant parameters in one place (at the subsystem level). This is called *Propagation* (page 302) and we will be discussing it in greater detail later in the chapter.

As we can see in the following plot, the results are identical when compared to the “flat” version presented previously:



Conclusion

We've already seen how component models can be used to turn equations into reusable components. This avoids the tedious, time-consuming and error prone process of manually entering equations over and over again. This same principle applies when we find ourselves constantly building the same assembly of component models into similar assemblies. We can use this subsystem model approach to create reusable assemblies of components and parameterize them such that the assembly can be used over and over again where the only changes required are parametric.

Lotka-Volterra with Migration

In this section, we will once again revisit the Lotka-Volterra models to understand how we can build on the work we've already done of creating reusable component models. We will now take the next step and create reusable models of individual geographical regions (each with the usual population dynamics) and then connect those geographical regions together with migration models.

Two Species Region

The models in this section will all make use of the following model that represents a region consisting of two populations, one of rabbits and the other foxes, with the usual Lotka-Volterra population dynamics. The Modelica source code for the model is:

```
within ModelicaByExample.Subsystems.LotkaVolterra.Components;
model TwoSpecies "Lotka-Volterra two species configuration"
  // Import several component models from ModelicaByExample.Components.LotkaVolterra
  import ModelicaByExample.Components.LotkaVolterra.Components.RegionalPopulation;
```

```

import ModelicaByExample.Components.LotkaVolterra.Components.Reproduction;
import ModelicaByExample.Components.LotkaVolterra.Components.Starvation;
import ModelicaByExample.Components.LotkaVolterra.Components.Predation;

parameter Real alpha=0.1 "Birth rate";
parameter Real gamma=0.4 "Starvation coefficient";
parameter Real initial_rabbit_population=10 "Initial rabbit population";
parameter Real initial_fox_population=10 "Initial fox population";
parameter Real beta=0.02 "Prey (rabbits) consumed";
parameter Real delta=0.02 "Predators (foxes) fed";

ModelicaByExample.Components.LotkaVolterra.Interfaces.Species rabbits
  "Population of rabbits in this region"
  annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
ModelicaByExample.Components.LotkaVolterra.Interfaces.Species foxes
  "Population of foxes in this region"
  annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  RegionalPopulation rabbit_population(
    initial_population=initial_rabbit_population,
    init=RegionalPopulation.InitializationOptions.FixedPopulation) "Rabbit population"
    annotation (Placement(transformation(extent={{-50,-60},{-30,-40}}));
  Reproduction reproduction(alpha=alpha) "Reproduction of rabbits"
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      origin={-80,-50})));
  RegionalPopulation fox_population(
    init=RegionalPopulation.InitializationOptions.FixedPopulation,
    initial_population=initial_fox_population)
    annotation (Placement(transformation(extent={{30,-60},{50,-40}}));
  Starvation fox_starvation(gamma=gamma) "Starvation of foxes"
    annotation (Placement(transformation(extent={{70,-60},{90,-40}}));
  Predation fox_predation(beta=beta, delta=delta)
    "Foxes eating rabbits"
    annotation (Placement(transformation(extent={{-10,-30},{10,-10}}));
equation
  connect(reproduction.species, rabbit_population.species)
    annotation (Line(
      points={{-80,-40},{-80,-20},{-40,-20},{-40,-40}},
      color={0,127,0},
      smooth=Smooth.None));
  connect(fox_predation.a, rabbit_population.species)
    annotation (Line(
      points={{-10,-20},{-40,-20},{-40,-40}},
      color={0,127,0},
      smooth=Smooth.None));
  connect(fox_starvation.species, fox_population.species)
    annotation (Line(
      points={{80,-40},{80,-20},{40,-20},{40,-40}},
      color={0,127,0},
      smooth=Smooth.None));
  connect(fox_population.species, fox_predation.b)
    annotation (Line(
      points={{40,-40},{40,-20},{10,-20}},
      color={0,127,0},
      smooth=Smooth.None));
  connect(rabbit_population.species, rabbits)
    annotation (Line(
      points={{-40,-40},{-40,0},{-100,0}},
      color={0,127,0},
      smooth=Smooth.None));
  connect(fox_population.species, foxes)
    annotation (Line(

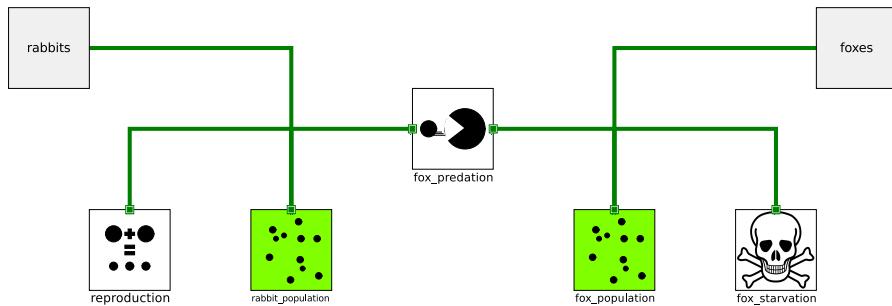
```

```

points={{40,-40},{40,0},{100,0}},
color={0,127,0},
smooth=Smooth.None));
end TwoSpecies;

```

The diagram for this component is rendered as:



This model will be used as the basis for the regional population dynamics in subsequent models presented in this section.

Unconnected Regions

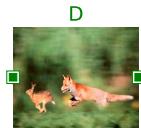
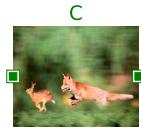
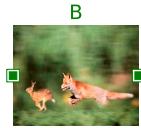
We'll start by building a model that consists of four unconnected regions. The Modelica source code for such a model is quite simple:

```

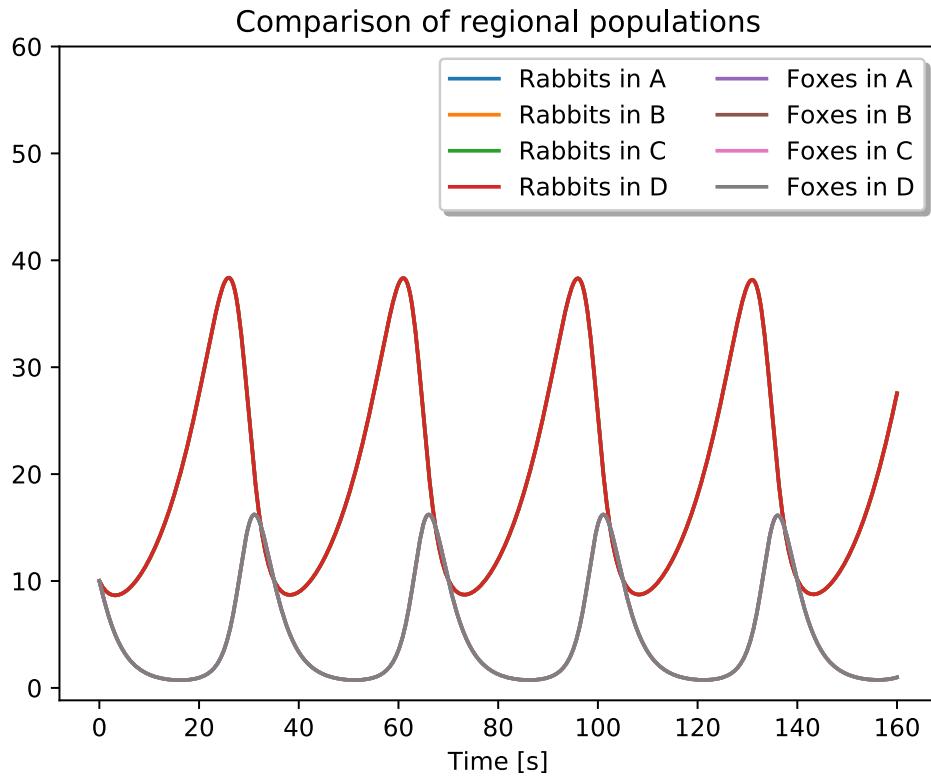
within ModelicaByExample.Subsystems.LotkaVolterra.Examples;
model UnconnectedPopulations "Several unconnected regional populations"
  Components.TwoSpecies A "Region A"
    annotation (Placement(transformation(extent={{-10,80},{10,100}})));
  Components.TwoSpecies B "Region B"
    annotation (Placement(transformation(extent={{-10,20},{10,40}})));
  Components.TwoSpecies C "Region C"
    annotation (Placement(transformation(extent={{-10,-40},{10,-20}})));
  Components.TwoSpecies D "Region D"
    annotation (Placement(transformation(extent={{-10,-100},{10,-80}})));
    annotation (experiment(StopTime=40, Intervals=0.008));
end UnconnectedPopulations;

```

The diagram for this model is equally simple:



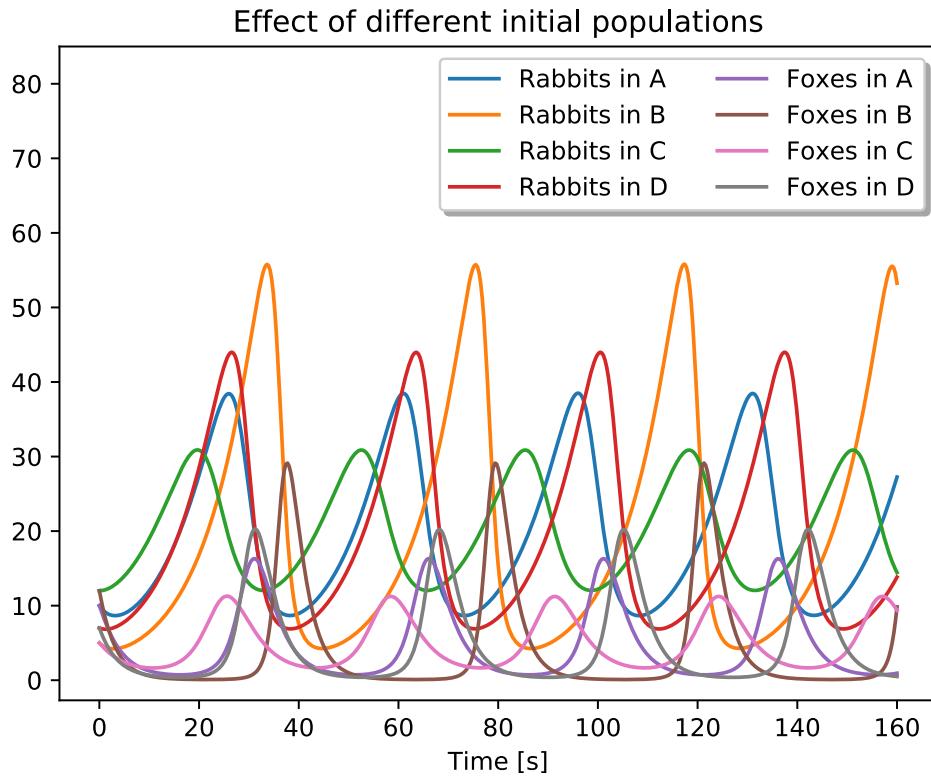
If we simulate this model, each population should follow the same trajectory since their initial conditions are identical. The following plot shows that this is, in fact, the case:



In a moment, we'll look at the effects of migration. But in order to fully appreciate the effect that migration has, we'll need to introduce some differences in the evolution of the different regions. So let's modify the initial conditions of the `UnconnectedPopulations` model to introduce some regional variation:

```
within ModelicaByExample.Subsystems.LotkaVolterra.Examples;
model InitiallyDifferent "Multiple regions with different initial populations"
  extends UnconnectedPopulations(
    B(initial_rabbit_population=5, initial_fox_population=12),
    C(initial_rabbit_population=12, initial_fox_population=5),
    D(initial_rabbit_population=7, initial_fox_population=7));
end InitiallyDifferent;
```

Simulating this model, we see that each region has a slightly different population dynamic:



Migration

Now that we have simulated the population dynamics in four **unconnected** regions, it would be interesting to note the impact that migration might have on these dynamics.

Consider the following Modelica model for migration:

```
within ModelicaByExample.Subsystems.LotkaVolterra.Components;
model Migration "Simple 'diffusion' based model of migration"
  parameter Real rabbit_migration=0.001 "Rabbit migration rate";
  parameter Real fox_migration=0.005 "Fox migration rate";
  ModelicaByExample.Components.LotkaVolterra.Interfaces.Species rabbit_a
    "Rabbit population in Region A"
    annotation (Placement(transformation(extent={{-70,90},{-50,110}})));
  ModelicaByExample.Components.LotkaVolterra.Interfaces.Species rabbit_b
    "Rabbit population in Region B"
    annotation (Placement(transformation(extent={{-70,-110},{-50,-90}})));
  ModelicaByExample.Components.LotkaVolterra.Interfaces.Species fox_a
    "Fox population in Region A"
    annotation (Placement(transformation(extent={{50,90},{70,110}})));
  ModelicaByExample.Components.LotkaVolterra.Interfaces.Species fox_b
    "Fox population in Region B"
    annotation (Placement(transformation(extent={{50,-110},{70,-90}})));
equation
  rabbit_a.rate = (rabbit_a.population-rabbit_b.population)*rabbit_migration;
  rabbit_a.rate + rabbit_b.rate = 0 "Conservation of rabbits";
  fox_a.rate = (fox_a.population-fox_b.population)*fox_migration;
  fox_a.rate + fox_b.rate = 0 "Conservation of foxes";
  annotation ( Icon(graphics={
    Rectangle(
      width=100,
      height=50,
      fill=white,
      stroke=black,
      strokeWidth=1px
    )
  })

```

```

        extent={{-100,100},{100,-100}},
        lineColor={0,127,0},
        fillColor={255,255,255},
        fillPattern=FillPattern.Solid),
    Text(
        extent={{-100,20},{100,-20}},
        lineColor={0,127,0},
        fillColor={255,255,255},
        fillPattern=FillPattern.Solid,
        textString="%name",
        origin={-120,0},
        rotation=90),
    Bitmap(extent={{-84,82},{-36,-82}}, fileName=
        "modelica://ModelicaByExample/Resources/Images/rabbit.png"),
    Bitmap(extent={{18,80},{94,-84}}, fileName=
        "modelica://ModelicaByExample/Resources/Images/fox.png")));
end Migration;

```

This model looks at the population of both rabbits and foxes in the connected regions and specifies a rate of migration that is proportional to the difference in population between the regions. In other words, if there are more rabbits in one region than another, the rabbits will move from the more populated region to the less population region. This is effectively a “diffusion” model of migration and does not necessarily have a basis in ecology. It is introduced simply as an example of how we could add additional effects, on top of those implemented in each region, to change the population dynamics between regions.

If we connect our previously unconnected regions with migration paths, *e.g.*,

```

within ModelicaByExample.Subsystems.LotkaVolterra.Examples;
model WithMigration "Connect populations by migration"
  extends InitiallyDifferent;
  Components.Migration migrate_AB "Migration from region A to region B"
    annotation (Placement(transformation(extent={{-10,50},{10,70}})));
  Components.Migration migrate_BC "Migration from region B to region C"
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  Components.Migration migrate_CD "Migration from region C to region D"
    annotation (Placement(transformation(extent={{-10,-70},{10,-50}})));
equation
  connect(migrate_CD.rabbit_b, D.rabbits) annotation (Line(
    points={{-6,-70},{-6,-76},{-20,-76},{-20,-90},{-10,-90}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(migrate_CD.rabbit_a, C.rabbits) annotation (Line(
    points={{-6,-50},{-6,-44},{-20,-44},{-20,-30},{-10,-30}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(migrate_BC.rabbit_b, C.rabbits) annotation (Line(
    points={{-6,-10},{-6,-16},{-20,-16},{-20,-30},{-10,-30}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(migrate_CD.fox_b, D.foxes) annotation (Line(
    points={{6,-70},{6,-76},{20,-76},{20,-90},{10,-90}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(migrate_BC.fox_b, C.foxes) annotation (Line(
    points={{6,-10},{6,-16},{20,-16},{20,-30},{10,-30}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(migrate_CD.fox_a, C.foxes) annotation (Line(
    points={{6,-50},{6,-44},{20,-44},{20,-30},{10,-30}},
    color={0,127,0},
    smooth=Smooth.None));
  connect(migrate_BC.fox_a, B.foxes) annotation (Line(

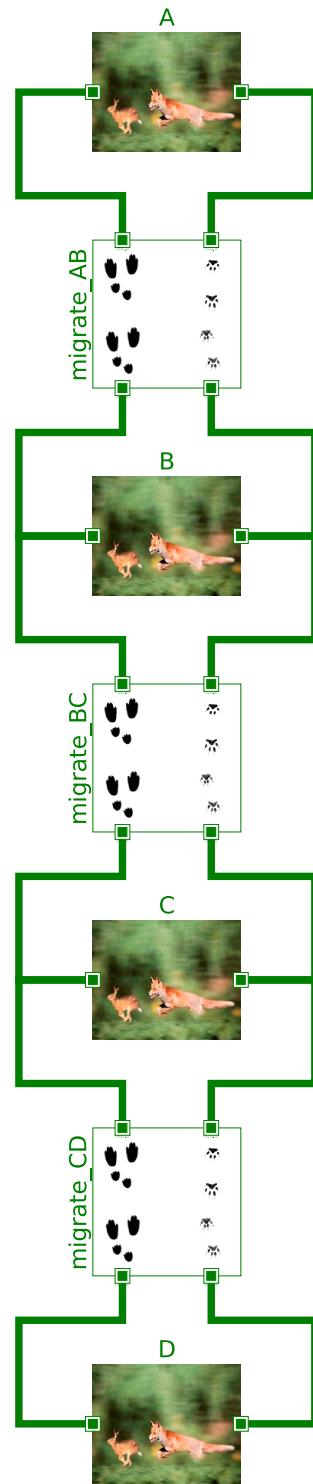
```

```

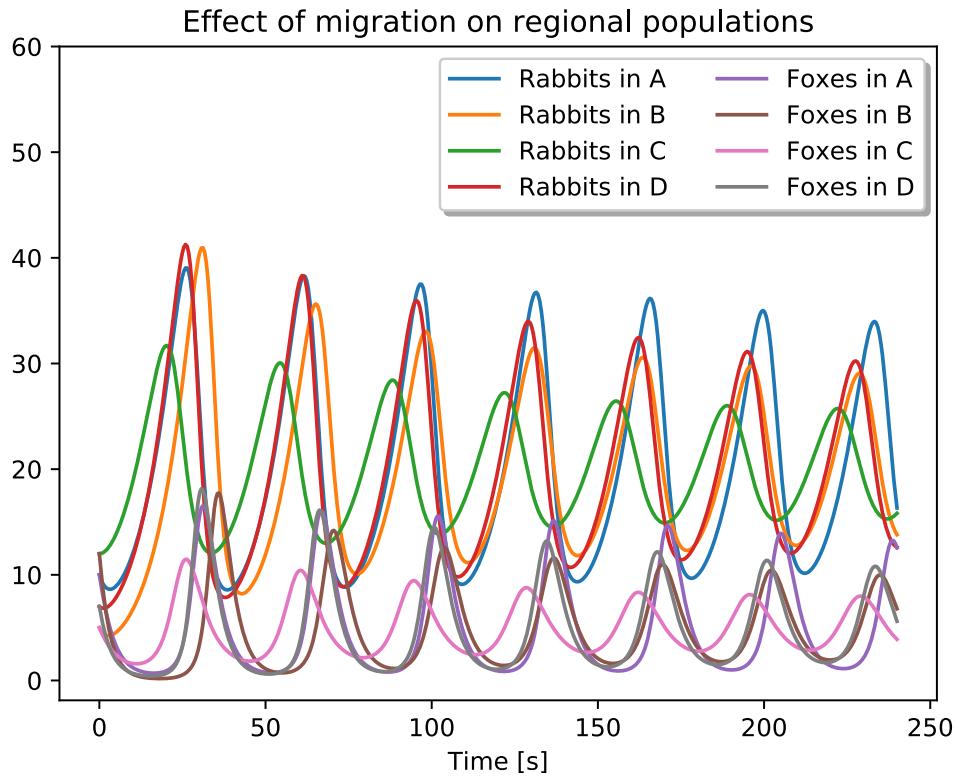
points={{6,10},{6,16},{20,16},{20,30},{10,30}},
color={0,127,0},
smooth=Smooth.None);
connect(migrate_BC.rabbit_a, B.rabbits) annotation (Line(
    points={{-6,10}, {-6,16}, {-20,16}, {-20,30}, {-10,30}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_AB.fox_b, B.foxes) annotation (Line(
    points={{6,50},{6,50},{6,44},{20,44},{20,30},{10,30}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_AB.rabbit_b, B.rabbits) annotation (Line(
    points={{-6,50}, {-6,44}, {-20,44}, {-20,30}, {-10,30}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_AB.fox_a, A.foxes) annotation (Line(
    points={{6,70},{6,76},{20,76},{20,90},{10,90}},
    color={0,127,0},
    smooth=Smooth.None));
connect(migrate_AB.rabbit_a, A.rabbits) annotation (Line(
    points={{-6,70}, {-6,76}, {-20,76}, {-20,90}, {-10,90}},
    color={0,127,0},
    smooth=Smooth.None));
end WithMigration;

```

the resulting system diagram becomes:



Simulating this system, we see that the population dynamics in the different regions start off out of sync, but eventually stabilize into repeating patterns:



Conclusion

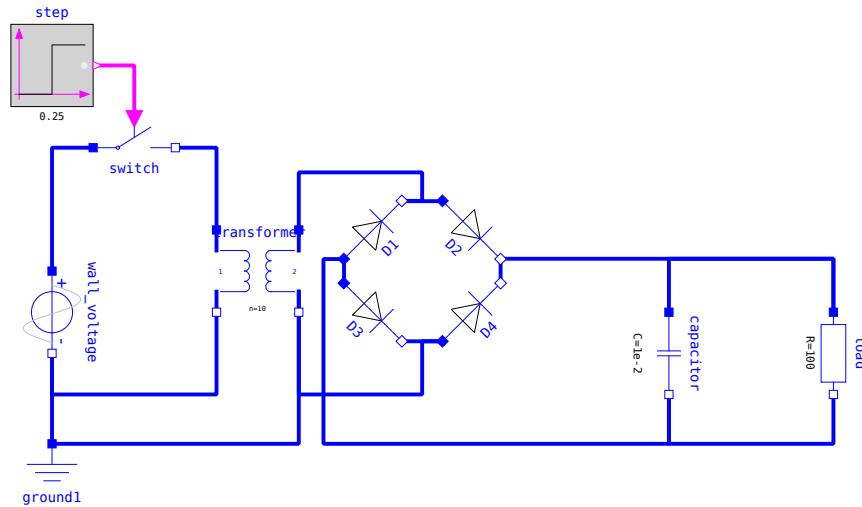
Earlier, we turned the Lotka-Volterra equations into components representing predation, starvation and reproduction. In this section, we were able to use those component models to build up subsystem models to represent the population dynamics in a particular region and then link those subsystems together into a hierarchical system model that also captured the effects of migration between these distinct regions.

DC Power Supply

In this section, we'll consider how a DC power supply model could be implemented in Modelica. We'll show, once again, how a flat system model can be refactored to make use of a reusable subsystem model.

Flat Power Supply Model

In this case, our flat system model will be the DC power supply circuit shown here:



Implemented in Modelica, this model looks like this:

```

within ModelicaByExample.Subsystems.PowerSupply.Examples;
model FlatCircuit "A model with power source, AC-DC conversion and load in one diagram"
  import Modelica.Electrical.Analog;
  Analog.Sources.SineVoltage wall_voltage(V=120, freqHz=60)
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=270, origin={-90,0}}));
  Analog.Ideal.IdealClosingSwitch switch(Goff=0)
    annotation (Placement(transformation(extent={{-80,30},{-60,50}})));
  Analog.Ideal.IdealTransformer transformer(Lm1=1, n=10, considerMagnetization=false)
    annotation (Placement(transformation(extent={{-50,0},{-30,20}})));
  Analog.Ideal.IdealDiode D1(Vknee=0, Ron=1e-5, Goff=1e-5)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=45, origin={-12,20})));
  Analog.Basic.Capacitor capacitor(C=1e-2)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=270, origin={60,-10})));
  Analog.Basic.Resistor load(R=100)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=270, origin={100,-10})));
  Modelica.Blocks.Sources.BooleanStep step(startTime=0.25)
    annotation (Placement(transformation(extent={{-100,50},{-80,70}})));
  Analog.Ideal.IdealDiode D2(Vknee=0, Ron=1e-5, Goff=1e-5)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=-45, origin={12,20})));
  Analog.Ideal.IdealDiode D3(Vknee=0, Ron=1e-5, Goff=1e-5)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=-45, origin={-12,0})));
  Analog.Ideal.IdealDiode D4(Vknee=0, Ron=1e-5, Goff=1e-5)
    annotation (Placement(
      transformation(

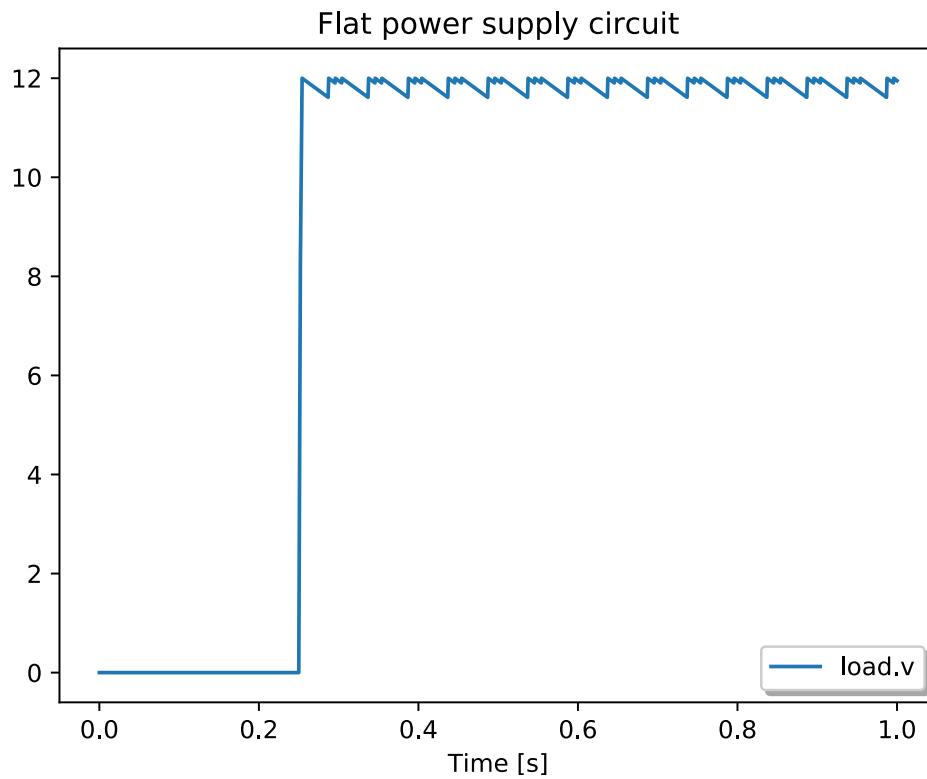
```

```

        extent={{-10,-10},{10,10}},
        rotation=45, origin={12,0})));
Analog.Basic.Ground ground1
annotation (Placement(transformation(extent={{-100,-52},{-80,-32}})));
equation
connect(D1.p, D3.p) annotation(
    Line(points = {{-19, 13}, {-19, 13}, {-19, 7}, {-19, 7}}, color = {0, 0, 255}));
connect(load.p, capacitor.p) annotation (Line(
    points = {{100, 0}, {100, 13}, {60, 13}, {60, 0}}, color={0,0,255}));
connect(load.p, D2.n) annotation (Line(
    points = {{100, 0}, {100, 13}, {19, 13}, {19, 12.9289}, {19.0711, 12.9289}}, color={0,0,255}));
connect(D1.n, D2.p) annotation(
    Line(points = {{-5, 27}, {5, 27}, {5, 27}, {5, 27}}, color = {0, 0, 255}));
connect(switch.p, wall_voltage.p) annotation (Line(
    points={{-80,40},{-90,40},{-90,10}}, color={0,0,255}, smooth=Smooth.None));
connect(switch.n, transformer.p1) annotation (Line(
    points={{-60,40},{-50,40},{-50,15}}, color={0,0,255}, smooth=Smooth.None));
connect(step.y, switch.control) annotation (Line(
    points={{-79,60},{-70,60},{-70,47}}, color={255,0,255}, smooth=Smooth.None));
connect(D3.n, D4.p) annotation (Line(
    points={{-4.92893,-7.07107},{-2.46446,-7.07107},{-2.46446,-7.07107},{1.09406e-006,-7.07107},{1.09406e-006,-7.07107},{4.92893,-7.07107}}, color={0,0,255}, smooth=Smooth.None));
connect(D2.n, D4.n) annotation (Line(
    points={{19.0711,12.9289},{19.0711,11.4644},{19.0711,11.4644},{19.0711,10},{19.0711,7.07107},{19.0711,7.07107}}, color={0,0,255}, smooth=Smooth.None));
connect(transformer.p2, D1.n) annotation (Line(
    points={{-30,15},{-30,34},{0,34},{0,27.0711},{-4.92893,27.0711}}, color={0,0,255}, smooth=Smooth.None));
connect(D4.p, transformer.n2) annotation (Line(
    points={{4.92893,-7.07107},{0,-7.07107},{0,-20},{-30,-20},{-30,5}}, color={0,0,255}, smooth=Smooth.None));
connect(wall_voltage.n, transformer.n1) annotation (Line(
    points={{-90,-10},{-90,-20},{-50,-20},{-50,5}}, color={0,0,255}, smooth=Smooth.None));
connect(wall_voltage.n, ground1.p) annotation (Line(
    points={{-90,-10},{-90,-32}}, color={0,0,255}, smooth=Smooth.None));
connect(transformer.n2, ground1.p) annotation (Line(
    points={{-30,5},{-30,-32},{-90,-32}}, color={0,0,255}, smooth=Smooth.None));
connect(D1.p, capacitor.n) annotation (Line(
    points={{-19.0711,12.9289},{-24,12.9289},{-24,-32},{60,-32},{60,-20}}, color={0,0,255}, smooth=Smooth.None));
connect(load.n, capacitor.n) annotation (Line(
    points={{100,-20},{100,-32},{60,-32},{60,-20}}, color={0,0,255}, smooth=Smooth.None));
end FlatCircuit;

```

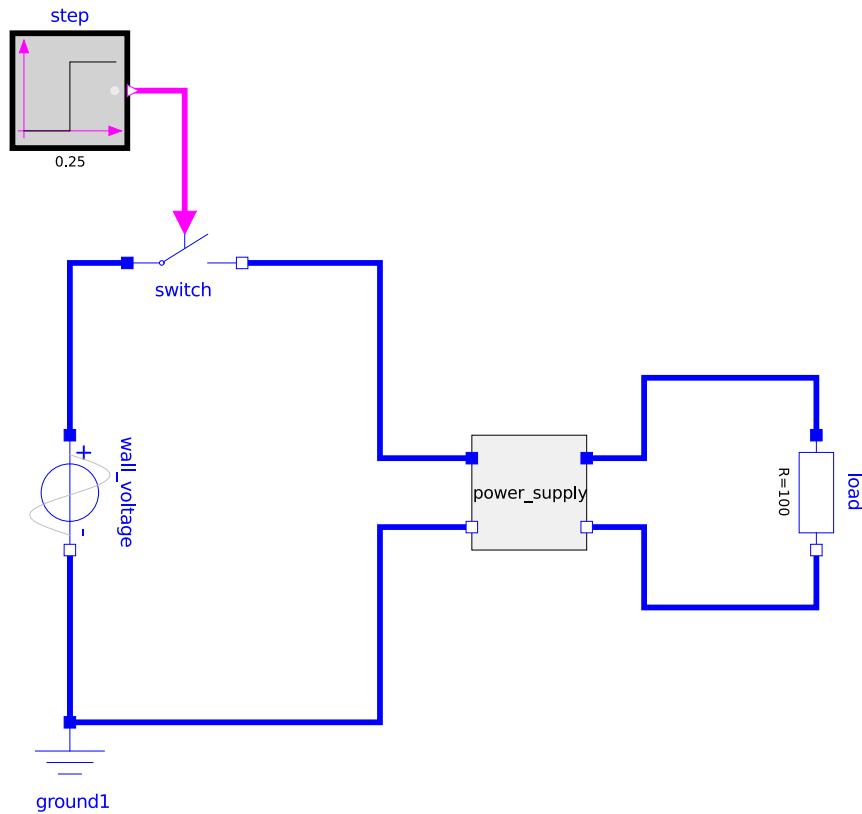
This kind of power supply works by taking an AC input voltage (120V at 60Hz), rectifying it and then passing it through a low-pass filter. If we simulate this model, we see the following voltage across the load resistor:



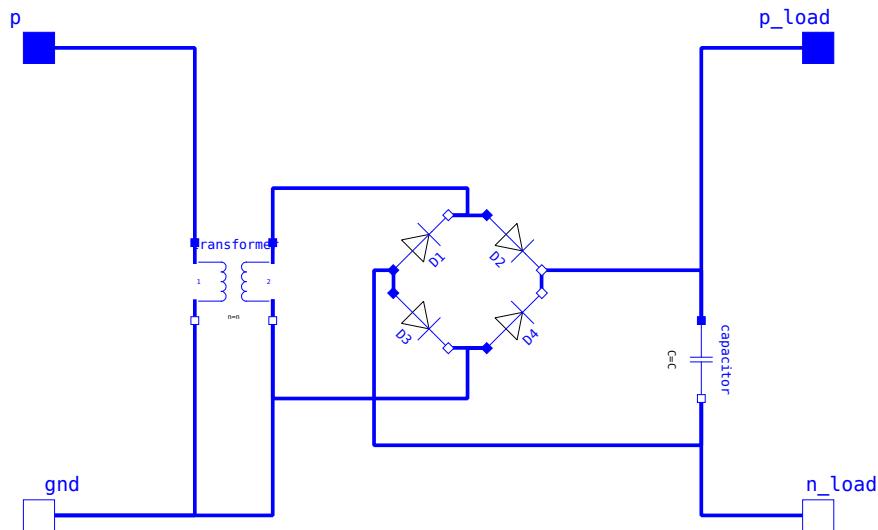
Note that our target here is an output voltage of 12 volts. However, the greater the load on the power supply, the lower the quality of the output signal will be. In this particular simulation, the load is initially zero (because the switch to the power supply is open). But when the switch is closed and current begins to flow through the load (the resistor named `load`), we start to see some artifact.

Hierarchical Power Supply

Once again, we'll improve upon the flat version of our system by taking some collection of components and organizing them into a subsystem model. Our system level circuit then becomes:



This model uses the `BasicPowerSupply` model whose diagram is shown here:



The Modelica source code for this reusable power supply subsystem model is:

```
within ModelicaByExample.Subsystems.PowerSupply.Components;
model BasicPowerSupply "Power supply with transformer and rectifier"
  import Modelica.Electrical.Analog;
```

```

parameter Modelica.SIunits.Capacitance C=1e-2
  "Filter capacitance"
  annotation(Dialog(group="General"));
parameter Modelica.SIunits.Conductance Goff=1e-5
  "Backward state-off conductance (opened diode conductance)"
  annotation(Dialog(group="General"));
parameter Modelica.SIunits.Resistance Ron=1e-5
  "Forward state-on differential resistance (closed diode resistance)"
  annotation(Dialog(group="General"));
parameter Real n=10
  "Turns ratio primary:secondary voltage"
  annotation(Dialog(group="Transformer"));
parameter Boolean considerMagnetization=false
  "Choice of considering magnetization"
  annotation(Dialog(group="Transformer"));
parameter Modelica.SIunits.Inductance Lm1=1e-2
  "Magnetization inductance w.r.t. primary side"
  annotation(Dialog(group="Transformer", enable=considerMagnetization));

Analog.Interfaces.NegativePin gnd
  "Pin to ground power supply"
  annotation (Placement(transformation(extent={{-110,-70},{-90,-50}})));
Analog.Interfaces.PositivePin p
  "Positive pin on supply side"
  annotation (Placement(transformation(extent={{-110,50},{-90,70}})));
Analog.Interfaces.PositivePin p_load
  "Positive pin for load side"
  annotation (Placement(transformation(extent={{90,50},{110,70}})));
Analog.Interfaces.NegativePin n_load
  "Negative pin for load side"
  annotation (Placement(transformation(extent={{90,-70},{110,-50}})));
protected
  Analog.Ideal.IdealTransformer transformer(
    final n=n, final considerMagnetization=considerMagnetization,
    final Lm1=Lm1)
    annotation (Placement(transformation(extent={{-60,-10},{-40,10}})));
  Analog.Ideal.IdealDiode D1(final Vknee=0, final Ron=Ron, final Goff=Goff)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=45,
        origin={-2,10})));
  Analog.Basic.Capacitor capacitor(C=C)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=270,
        origin={70,-20})));
  Analog.Ideal.IdealDiode D2(final Vknee=0, final Ron=Ron, final Goff=Goff)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=-45,
        origin={22,10})));
  Analog.Ideal.IdealDiode D3(final Vknee=0, final Ron=Ron, final Goff=Goff)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=-45,
        origin={-2,-10})));
  Analog.Ideal.IdealDiode D4(final Vknee=0, final Ron=Ron, final Goff=Goff)
    annotation (Placement(
      transformation(

```

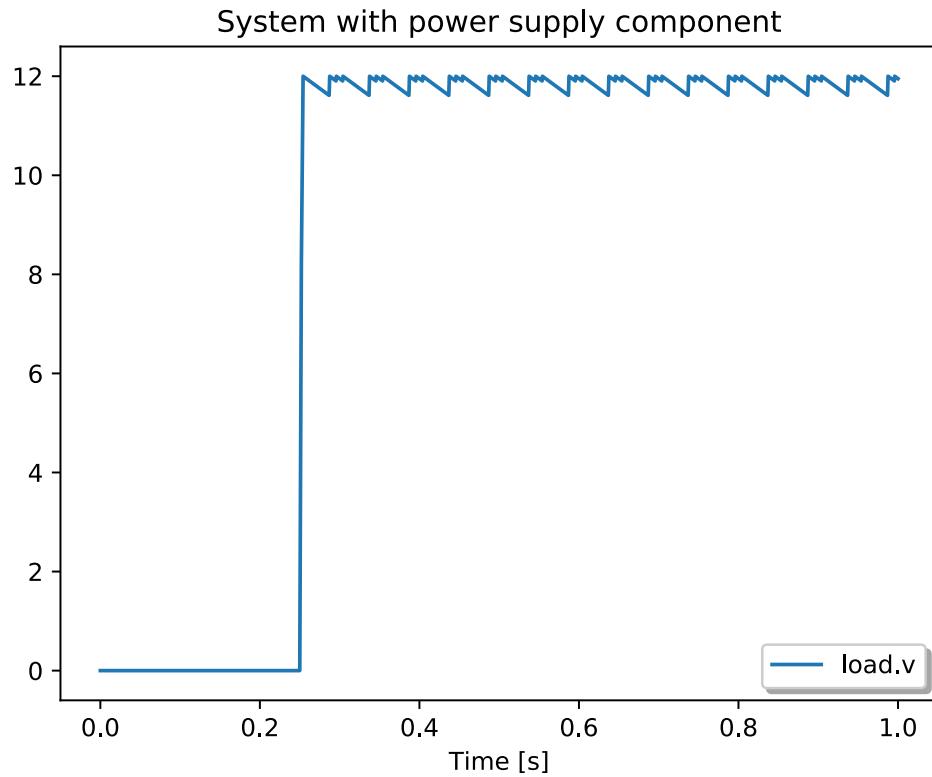
```

        extent={{-10,-10},{10,10}},
        rotation=45,
        origin={22,-10})));
equation
  connect(D1.p, capacitor.n) annotation (Line(
    points = {{-9.07107, 2.92893}, {-8.5355, 2.92893}, {-8.5355, 2.92893}, {-14, 2.92893}, {-14, -42}, {70, -42}, {70, -30}}, color={0,0,255}));
  connect(transformer.p2, D1.n) annotation(
    Line(points = {{-40, 5}, {-40, 24}, {10, 24}, {10, 17.0711}, {5.07107, 17.0711}}, color =
{0, 0, 255}));
  connect(capacitor.p, D2.n) annotation(
    Line(points = {{70, -10}, {70, 2.92893}, {29.0711, 2.92893}}, color = {0, 0, 255}));
  connect(D2.n, D4.n) annotation(
    Line(points = {{29, 3}, {29, 3}, {29, -3}, {29, -3}}, color = {0, 0, 255}));
  connect(D1.p, D3.p) annotation(
    Line(points = {{-9, 3}, {-9, 3}, {-9, -3}, {-9, -3}}, color = {0, 0, 255}));
  connect(D3.n, D4.p) annotation(
    Line(points = {{5, -17}, {15, -17}, {15, -17}, {15, -17}}, color = {0, 0, 255}));
  connect(D1.n, D2.p) annotation(
    Line(points = {{5, 17}, {15, 17}, {15, 17}, {15, 17}}, color = {0, 0, 255}));
  connect(D4.p, transformer.n2) annotation (Line(
    points={{14.9289,-17.0711},{10,-17.0711},{10,-30},{-40,-30},{-40,-5}}, color={0,0,255},
    smooth=Smooth.None));
  connect(transformer.n1, gnd) annotation (Line(
    points={{-60,-5},{-60,-60},{-100,-60}}, color={0,0,255},
    smooth=Smooth.None));
  connect(transformer.n2, gnd) annotation (Line(
    points={{-40,-5},{-40,-60},{-100,-60}}, color={0,0,255},
    smooth=Smooth.None));
  connect(transformer.p1, p) annotation (Line(
    points={{-60,5},{-60,60},{-100,60}}, color={0,0,255},
    smooth=Smooth.None));
  connect(capacitor.p, p_load) annotation (Line(
    points={{70,-10},{70,60},{100,60}}, color={0,0,255},
    smooth=Smooth.None));
  connect(capacitor.n, n_load) annotation (Line(
    points={{70,-30},{70,-60},{100,-60}}, color={0,0,255},
    smooth=Smooth.None));
end BasicPowerSupply;

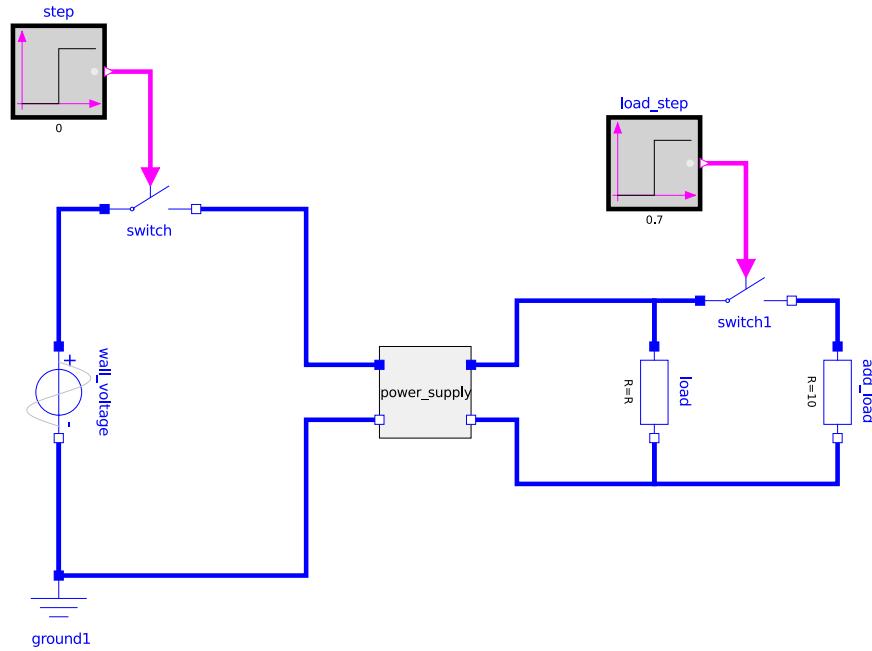
```

There are a couple of interesting things to note about this model. First, we see the same organizational structure as we have before where parameters and connectors are made `public` while the internal components are `protected`. We can also see the use of the *Dialog* (page 263) annotation to organize parameters into distinct groupings (in this case, "General" and "Transformer"). We can also see the use of the `enable` annotation in conjunction with `considerMagnetization` parameter to selectively expose the `Lm1` parameter only in the cases where `considerMagnetization` is true.

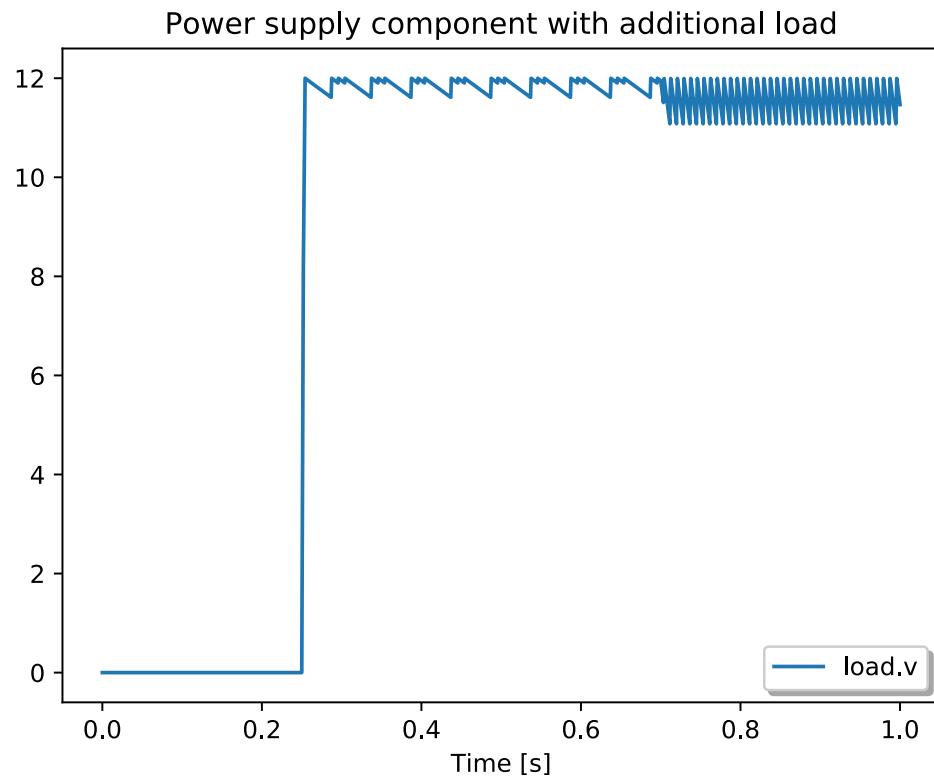
Using our hierarchical system model we get, as expected, exactly the same results as we got for the flat version:



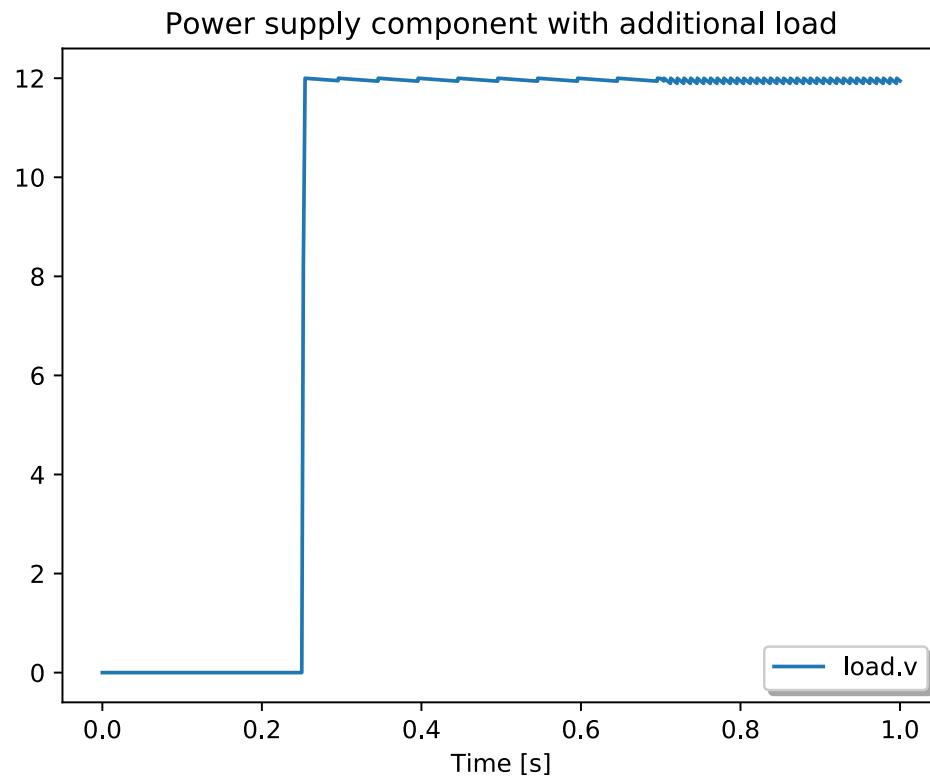
We can augment our system model to include an additional load (that comes online after some delay):



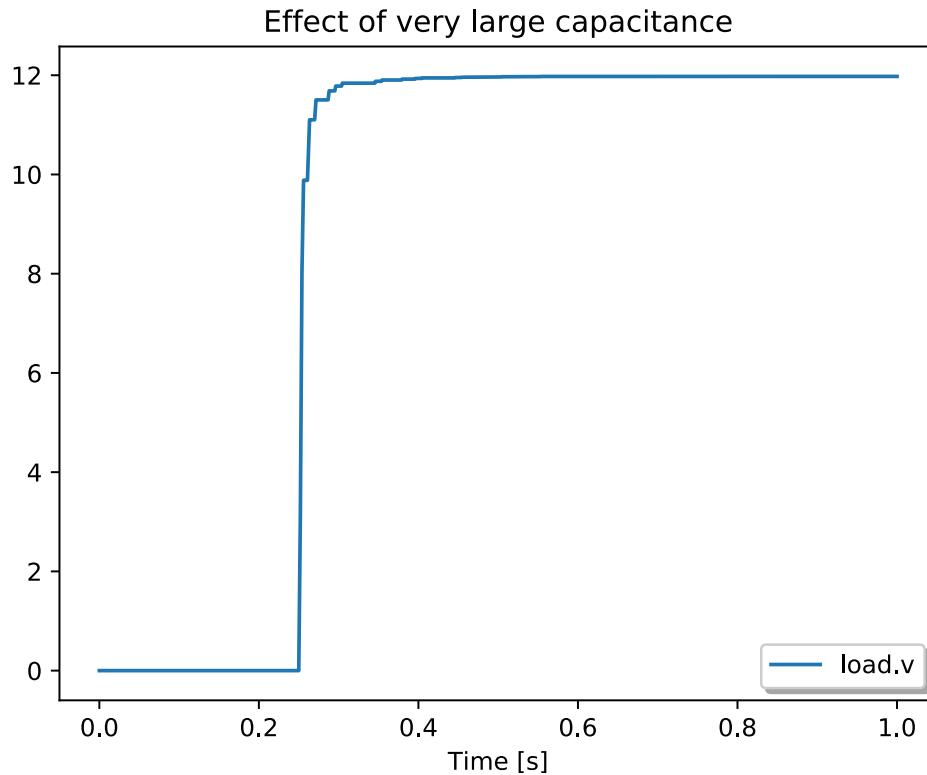
In that case, if we simulate the model we can see the impact that additional load will have on the quality of power supply output:



By increasing the capacitance in the power supply, we can reduce the amplitude of the voltage fluctuations, *e.g.*,



However, if we increase the capacitance level too much, we will find that the power supply output is very slow to respond to load changes, *e.g.*,



Conclusion

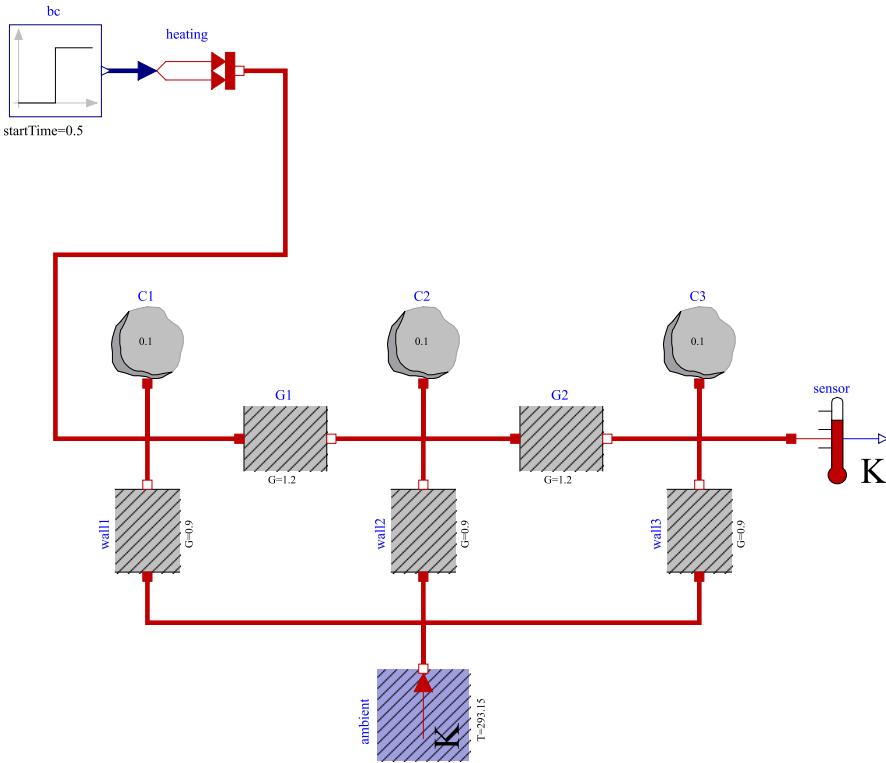
This example illustrates, once again, how a collection of components can be organized into a reusable subsystem model. This example follows the best practice of placing parameters and connectors in the `public` section of the resulting subsystem model while keeping the internal components in the `protected` section.

Spatially Distributed Heat Transfer

Our next example of creating reusable subsystems will introduce a slight twist. In this section we will not only demonstrate how to create a reusable subsystem model, as in the previous examples from this chapter, but that subsystem model will use an array of `component` instances where the size of that array can be used to control the spatial resolution of the results. This is similar to the kind of model presented earlier in [One-Dimensional Heat Transfer](#) (page 91).

Flat System

Let's start, as usual, with a flat system level model like the one shown below:



This model consists of a collection of thermal capacitances with thermal conductors in between them. In practical terms, the thermal capacitances could represent segments of a metal pipe. The axial conduction could represent heat conduction axially along the length of the rod. The radial conduction could be the heat lost through some insulating material like plastic. In this example, there are 3 such thermal capacitances and 5 thermal conductors. On the left side, heat is applied to the system and on the right side a temperature sensor measures how the temperature of the rightmost thermal capacitance changes. So the example applies heat at one end and monitors the temperature increase at the other end.

When implemented in Modelica, the model looks like this:

```
within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model FlatRod "Modeling a heat transfer in a rod in a without subsystems"
  Modelica.Thermal.HeatTransfer.Sources.PrescribedHeatFlow heating
    "Heating actuator"
    annotation (Placement(transformation(extent={{-60,50},{-40,70}})));
  Modelica.Blocks.Sources.Step bc(height=10, startTime=0.5) "Heat profile"
    annotation (Placement(transformation(extent={{-90,50},{-70,70}})));
  Modelica.Thermal.HeatTransfer.Components.HeatCapacitor C1(C=0.1, T(fixed=true))
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      origin={-60,2})));
  Modelica.Thermal.HeatTransfer.Components.ThermalConductor G1(G=1.2)
    annotation (Placement(transformation(extent={{-40,-30},{-20,-10}}));
  Modelica.Thermal.HeatTransfer.Components.HeatCapacitor C2(C=0.1, T(fixed=true))
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      origin={0,2})));
  Modelica.Thermal.HeatTransfer.Components.ThermalConductor G2(G=1.2)
    annotation (Placement(transformation(extent={{20,-30},{40,-10}}));
  Modelica.Thermal.HeatTransfer.Components.HeatCapacitor C3(C=0.1, T(fixed=true))
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      origin={60,2})));
```

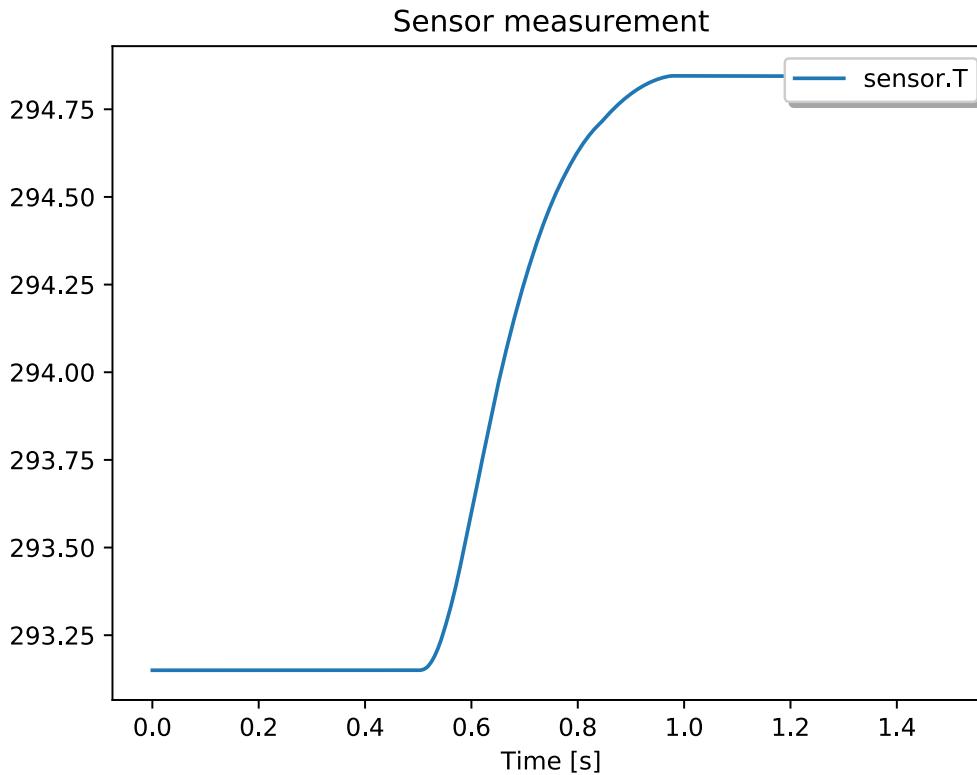
```

Modelica.Thermal.HeatTransfer.Sensors.TemperatureSensor sensor
  annotation (Placement(transformation(extent={{80,-30},{100,-10}})));
Modelica.Thermal.HeatTransfer.Components.ThermalConductor wall1(G=0.9)
  annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90, origin={-60,-40})));
Modelica.Thermal.HeatTransfer.Sources.FixedTemperature ambient(T=293.15)
  "Ambient temperature" annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90, origin={0,-80})));
Modelica.Thermal.HeatTransfer.Components.ThermalConductor wall2(G=0.9)
  annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90, origin={0,-40})));
Modelica.Thermal.HeatTransfer.Components.ThermalConductor wall3(G=0.9)
  annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90, origin={60,-40})));
equation
  connect(bc.y, heating.Q_flow) annotation (Line(
    points={{-69,60}, {-60,60}},
    color={0,0,127}, smooth=Smooth.None));
  connect(C1.port, G1.port_a) annotation (Line(
    points={{-60,-8}, {-60,-20}, {-40,-20}},
    color={191,0,0}, smooth=Smooth.None));
  connect(C2.port, G2.port_a) annotation (Line(
    points={{0,-8}, {0,-20}, {20,-20}},
    color={191,0,0}, smooth=Smooth.None));
  connect(G2.port_b, C3.port) annotation (Line(
    points={{40,-20}, {60,-20}, {60,-8}},
    color={191,0,0}, smooth=Smooth.None));
  connect(G1.port_b, C2.port) annotation (Line(
    points={{-20,-20}, {0,-20}, {0,-8}},
    color={191,0,0}, smooth=Smooth.None));
  connect(heating.port, C1.port) annotation (Line(
    points={{-40,60}, {-30,60}, {-30,20}, {-80,20}, {-80,-20}, {-60,-20}, {-60,-8}},
    color={191,0,0}, smooth=Smooth.None));
  connect(wall1.port_b, C1.port) annotation (Line(
    points={{-60,-30}, {-60,-8}},
    color={191,0,0}, smooth=Smooth.None));
  connect(wall1.port_a, ambient.port) annotation (Line(
    points={{-60,-50}, {-60,-60}, {0,-60}, {0,-70}},
    color={191,0,0}, smooth=Smooth.None));
  connect(wall2.port_a, ambient.port) annotation (Line(
    points={{0,-50}, {0,-50}, {0,-70}},
    color={191,0,0}, smooth=Smooth.None));
  connect(wall3.port_a, ambient.port) annotation (Line(
    points={{60,-50}, {60,-60}, {0,-60}, {0,-70}},
    color={191,0,0}, smooth=Smooth.None));
  connect(wall3.port_b, C3.port) annotation (Line(
    points={{60,-30}, {60,-8}},
    color={191,0,0}, smooth=Smooth.None));
  connect(sensor.port, C3.port) annotation (Line(
    points={{80,-20}, {60,-20}, {60,-8}},
    color={191,0,0}, smooth=Smooth.None));
  connect(C2.port, wall2.port_b) annotation (Line(
    points={{0,-8}, {0,-19}, {0,-30}, {0,-30}},
    color={191,0,0},
    smooth=Smooth.None));
end FlatRod;

```

Simulating this system, we can see the temperature response of the rightmost thermal capacitance in

the following plot:



Segmented Rod Subsystem

In our flat system model, we have 3 thermal capacitances and 5 conductances. This configuration represents a rod that has been divided into 3 equal segments and the conductance that occurs between those segments as well as between each segment and some ambient conditions. In theory, we can divide a rod into N equal segments with $N - 1$ conduction paths between them and N conduction paths between each segment and the ambient conditions.

Our particular configuration was for $N = 3$. But we can create a subsystem model where N is a parameter of the subsystem. In other words, we can create a subsystem model that is divided into N equal segments. But to do so, we cannot simply drag and drop the capacitances into the model and connect them with conductances because we don't know the exact number.

Instead, we'll use an array of **components** to represent this collection of capacitances and conductances. The resulting Rod model can be written in Modelica as follows:

```
within ModelicaByExample.Subsystems.HeatTransfer.Components;
model Rod "Modeling discretized rod"
  import HTC=Modelica.Thermal.HeatTransfer.Components;

  parameter Integer n(start=2,min=2) "Number of rod segments";
  parameter Modelica.SIunits.Temperature T0 "Initial rod temperature";
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port_a
    "Thermal connector to ambient"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_b port_b
    "Thermal connector for rod end 'b'"
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));


  
```

```

parameter Modelica.SIunits.HeatCapacity C
  "Total heat capacity of element (= cp*m)";
parameter Modelica.SIunits.ThermalConductance G_wall
  "Thermal conductivity of wall";
parameter Modelica.SIunits.ThermalConductance G_rod
  "Thermal conductivity of rod";
Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a ambient
  "Thermal connector to ambient"
  annotation (Placement(transformation(extent={{-10,-110},{10,-90}})));
protected
  HTC.HeatCapacitor capacitance[n](each final C=C/n, each T(start=T0, fixed=true))
  annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    origin={-30,20})));
  HTC.ThermalConductor wall[n](each final G=G_wall/n)
  annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90, origin={-30,-20})));
  HTC.ThermalConductor rod_conduction[n-1](each final G=G_rod*(n-1))
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
equation
  for i in 1:n loop
    connect(capacitance[i].port, wall[i].port_b) "Capacitance to walls";
    connect(wall[i].port_a, ambient) "Walls to ambient";
  end for;
  for i in 1:n-1 loop
    connect(capacitance[i].port, rod_conduction[i].port_a)
    "Capacitance to next conduction";
    connect(capacitance[i+1].port, rod_conduction[i].port_b)
    "Capacitance to prev conduction";
  end for;
  connect(capacitance[1].port, port_a) "First capacitance to rod end";
  connect(capacitance[n].port, port_b) "Last capacitance to (other) rod end";
end Rod;

```

There are several interesting things to note about this model. First, the number of segments the rod will be divided into is represented by the `n` parameter:

```
parameter Integer n(start=2,min=2) "Number of rod segments";
```

The parameter `n` is then used in the following declarations to specify the number of capacitance and conductance elements in the rod:

```

HTC.HeatCapacitor capacitance[n](each final C=C/n, each T(start=T0, fixed=true))
  annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    origin={-30,20})));
  HTC.ThermalConductor wall[n](each final G=G_wall/n)
  annotation (Placement(transformation(
    extent={{-10,-10},{10,10}},
    rotation=90, origin={-30,-20})));
  HTC.ThermalConductor rod_conduction[n-1](each final G=G_rod*(n-1))
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));

```

Note that if we wish to apply a modification, *e.g.*, $G=G_{\text{rod}}/n$ to every component in an array of components, we can use the `each` qualifier on the modification. We'll discuss the `each` qualifier and how to apply modifications to arrays of components later in this chapter in the section on *Modifications* (page 301).

Now that we've declared our component arrays, we can then wire together the capacitances and conductances using `for` loops in an `equation` section:

```

for i in 1:n loop
  connect(capacitance[i].port, wall[i].port_b) "Capacitance to walls";
  connect(wall[i].port_a, ambient) "Walls to ambient";
end for;
for i in 1:n-1 loop
  connect(capacitance[i].port, rod_conduction[i].port_a)
    "Capacitance to next conduction";
  connect(capacitance[i+1].port, rod_conduction[i].port_b)
    "Capacitance to prev conduction";
end for;

```

We also need to connect the ends of the rod to the external connectors so that the rod can be connected to other models:

```

connect(capacitance[1].port, port_a) "First capacitance to rod end";
connect(capacitance[n].port, port_b) "Last capacitance to (other) rod end";

```

In this way, we are able to create a segmented rod model with an arbitrary number of equally divided segments.

Spatial Resolution

Now that we have our parameterized Rod model, we can look at how the number of segments in the rod impacts the response we see. Ultimately, what we should see is that as the number of segments gets larger (or, as the size of the segments gets smaller), we should converge on a solution.

We'll start by considering a model where $n=3$, *i.e.*,

```

within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model ThreeSegmentRod "Modeling a heat transfer using 3 segment rod subsystem"
  Modelica.Thermal.HeatTransfer.Sources.PrescribedHeatFlow heating
    "Heating actuator"
    annotation (Placement(transformation(extent={{-60,50},{-40,70}})));
  Modelica.Blocks.Sources.Step bc(height=10, startTime=0.5) "Heat profile"
    annotation (Placement(transformation(extent={{-90,50},{-70,70}}));
  Modelica.Thermal.HeatTransfer.Sensors.TemperatureSensor sensor
    annotation (Placement(transformation(extent={{80,-10},{100,10}}));
  Modelica.Thermal.HeatTransfer.Sources.FixedTemperature ambient(T=293.15)
    "Ambient temperature" annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=90, origin={0,-80})));
  Components.Rod rod(n=3, C=0.3, G_wall=2.7, T0=293.15, G_rod=1.2)
    annotation (Placement(transformation(extent={{-20,-20},{20,20}}));
equation
  connect(bc.y, heating.Q_flow) annotation (Line(
    points={{-69,60}, {-60,60}},
    color={0,0,127}, smooth=Smooth.None));
  connect(heating.port, rod.port_a) annotation (Line(
    points={{-40,60}, {-30,60}, {-30,0}, {-20,0}},
    color={191,0,0},
    smooth=Smooth.None));
  connect(rod.ambient, ambient.port) annotation (Line(
    points={{0,-20}, {0,-70}, {0,-70}},
    color={191,0,0},
    smooth=Smooth.None));
  connect(rod.port_b, sensor.port) annotation (Line(
    points={{20,0}, {80,0}},
    color={191,0,0},
    smooth=Smooth.None));
end ThreeSegmentRod;

```

We can then extend this model to create additional models with more segments, *e.g.*,

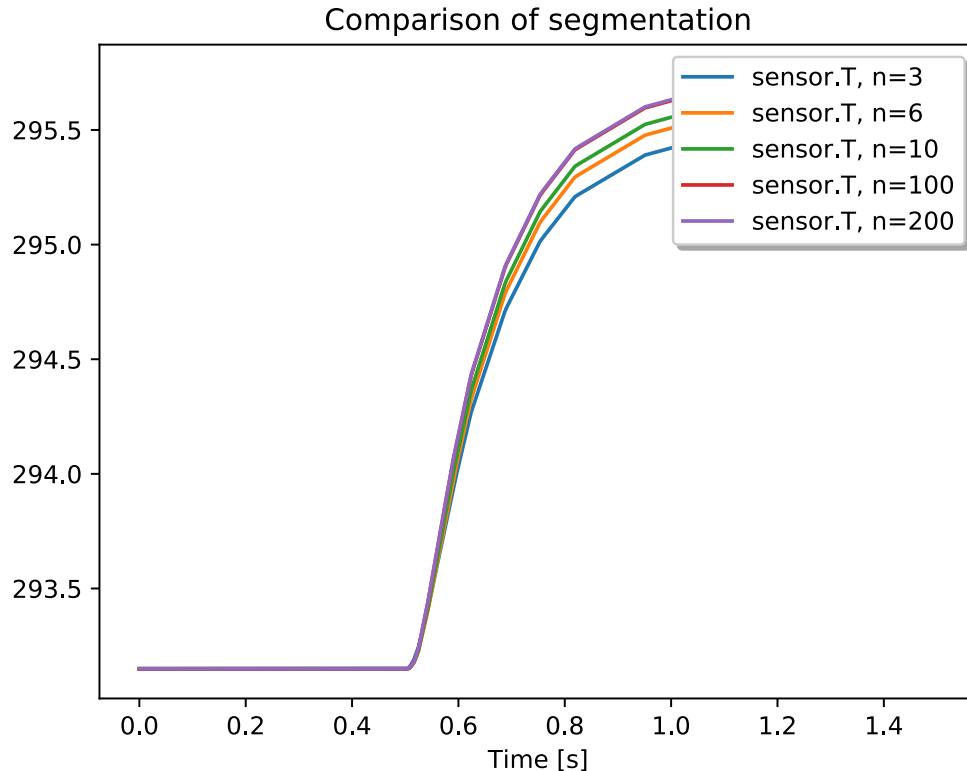
```
within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model SixSegmentRod "Rod divided into 6 pieces"
  extends ThreeSegmentRod(rod(n=6));
end SixSegmentRod;
```

```
within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model TenSegmentRod
  extends SixSegmentRod(rod(n=10));
end TenSegmentRod;
```

```
within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model OneHundredSegmentRod "Rod divided into 100 pieces"
  extends ThreeSegmentRod(rod(n=100));
end OneHundredSegmentRod;
```

```
within ModelicaByExample.Subsystems.HeatTransfer.Examples;
model TwoHundredSegmentRod
  extends OneHundredSegmentRod(rod(n=200));
end TwoHundredSegmentRod;
```

If we simulate all of these cases, we see that as n gets larger, they appear to converge to a common solution and that $n=10$ seems to provide a reasonable solution without the need to introduce a large number of superfluous components:



Conclusion

In this section, we've seen how we can build assemblies of arbitrary size using arrays of components and `for` loops to connect them together.

Harmonic Motion of Pendulums

In this section, we will recreate an interesting experiment [*Berg*] (page 357) involving pendulums. If we create a series of pendulums with different natural frequencies and then start them all at the same position, what we will see is that they will oscillate at different frequencies. But if we remove any energy dissipation from the system, they will eventually all “reunite” at their initial position.

In general, the period between these “reunion” events is the least common multiple of the periods of all the pendulums in the system. We can choose the lengths of the pendulums to achieve a specific period between these “reunions”.

Pendulum Model

In this section, we will use arrays of components to build our subsystem model just like we did in our *Spatially Distributed Heat Transfer* (page 287) example. But before we can create an array of pendulums, we need to first have a model of a single pendulum.

```
within ModelicaByExample.Subsystems.Pendula;
model Pendulum "A single individual pendulum"
  import Modelica.Mechanics.MultiBody.Parts;
  import Modelica.Mechanics.MultiBody.Joints;

  parameter Modelica.SIunits.Position x;
  parameter Modelica.SIunits.Mass m "Mass of mass point";
  parameter Modelica.SIunits.Angle phi "Initial angle";
  parameter Modelica.SIunits.Length L "String length";
  parameter Modelica.SIunits.Diameter d=0.01;

  Parts.Fixed ground(r={0,0,x}, animation=false)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=270, origin={0,60})));
  Parts.PointMass ball(m=m, sphereDiameter=5*d)
    annotation (Placement(transformation(extent={{-10,-90},{10,-70}})));
  Parts.BodyCylinder string(density=0, r={0,L,0}, diameter=d)
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=90,
      origin={0,-30})));
  Joints.Revolute revolute(phi(fixed=true, start=phi),
    cylinderDiameter=d/2, animation=false)
    annotation (Placement(
      transformation(
        extent={{-10,-10},{10,10}},
        rotation=90,
        origin={0,20})));
equation
  connect(string.frame_a, ball.frame_a) annotation (Line(
    points={{0,-40},{0,-40},{0,-80}},
    color={95,95,95},
    thickness=0.5,
    smooth=Smooth.None));
  connect(revolute.frame_b, ground.frame_b) annotation (Line(
    points={{0,30},{0,40},{0,40},{0,50}},
```

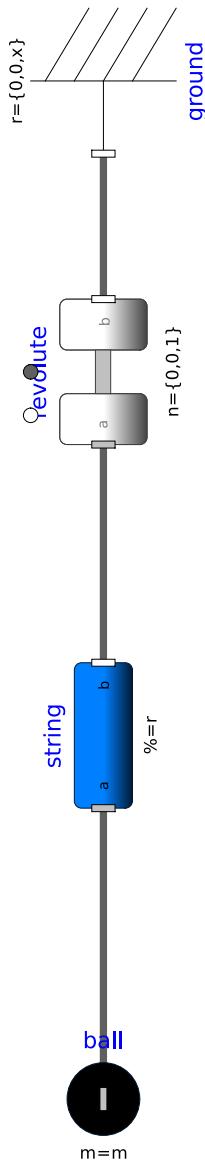
```

color={95,95,95},
thickness=0.5,
smooth=Smooth.None));
connect(revolute.frame_a, string.frame_b) annotation (Line(
points={{0,10},{0,10},{0,-20},{0,-20}},
color={95,95,95},
thickness=0.5,
smooth=Smooth.None));
end Pendulum;

```

This particular model is using the `Modelica.Mechanics.MultiBody` library. This library not only includes many useful models of parts and joints commonly found in mechanisms, but it also includes information about how to render those parts. This allows Modelica tools to transform the simulated results of such a model directly into a 3D animation of the system.

The components of the pendulum can be rendered as follows:



System Model

Now that we have an individual pendulum model, we can build a system of pendulums. If we want a system of n pendulums where the period for a complete cycle of the system is T seconds, we compute the length of the i^{th} pendulum as:

$$l_i = g_n \frac{T}{2\pi(X + (n - i))}$$

where g_n is Earth's gravitational constant, n is the number of pendulums, T is the period of one complete cycle of the system and X is the number of oscillations of the longest pendulum over T seconds.

In Modelica, we could build such a system as follows:

```
within ModelicaByExample.Subsystems.Pendula;
model System "A system of pendula"
  import Modelica.Constants.g_n;
  import Modelica.Constants.pi;

  parameter Integer n=15 "Number of pendula";
  parameter Modelica.SIunits.Position x[n] = linspace(0,(n-1)*0.05,n);
  parameter Modelica.SIunits.Time T = 54;
  parameter Modelica.SIunits.Time X = 30;
  parameter Modelica.SIunits.Length lengths[n] = { g_n*(T/(2*pi*(X+(n-i))))^2 for i in 1:n};
  parameter Modelica.SIunits.Angle phi0 = 0.5;

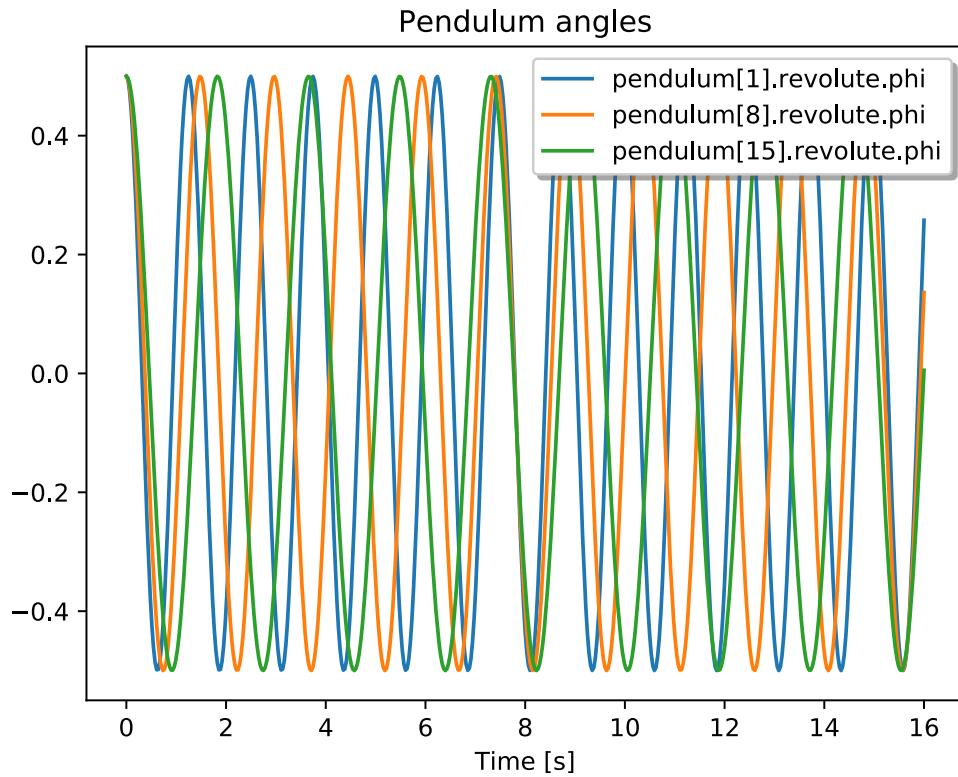
  Pendulum pendulum[n](x=x, each m=1, each phi=phi0, L=lengths)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  inner Modelica.Mechanics.MultiBody.World world
    annotation (Placement(transformation(extent={{-80,-60},{-60,-40}})));
end System;
```

There are two declarations of interest here. The first is the declaration of the `world` component. This is needed to provide a frame of reference for the system as well as some key environmental parameters like the gravitational constant to use in the system. In any given model there should only be one `world` component in the system. Another very interesting declaration is the `pendulum` component:

```
Pendulum pendulum[n](x=x, each m=1, each phi=phi0, L=lengths)
```

Because `pendulum` is an array of `n` components, there will be `n` values for the `x`, `m`, `phi` and `L` parameters associated with these pendulums. For example, if `n=3`, then the model will have 3 values for `x`: `pendulum[1].x`, `pendulum[2].x` and `pendulum[3].x`. In the declaration of `pendulum`, we handle this in different ways for different parameters. In the case of `m`, we give each pendulum the same value with the modification `each m=1`. However, in the case of `L` (and `x`), we supply an array of values, `L=lengths` used to initialize the parameters where the values in the `lengths` array are computed using the equation for pendulum lengths we introduced earlier. We will give a more complete discussion on how to apply modifications to arrays of components *later in this chapter* (page 301).

If we simulate this system, we get the following solution for the trajectory of each of the pendulums:



As we can see from this plot, although the pendulums oscillate at different frequencies, their positions coincide at regular intervals. In fact, every 54 seconds they all simultaneously return to their initial positions.

This phenomenon can be more clearly visualized in three dimensions¹⁴.

Conclusion

In this section, we have seen how arrays of components can be used, declared and modified. In this particular case, this allows us to specify the number of pendulums in our system and then simulate them to observe the peculiar behavior observed when we choose their lengths according to the equation specified earlier.

2.4.2 Review

Subsystem Interface

The emphasis of this chapter was on how component models could be organized into reusable subsystems. As we saw in numerous examples in this chapter, there is a common pattern that emerges. Let's review various aspects of subsystem models to understand this pattern better.

Parameters

Generally, parameters for a subsystem should be `public`. Normally, all declarations in any `model` or `block` definition will be public unless they come after the `protected` keyword. In such cases, it is still

¹⁴ <https://youtu.be/cc7ajqhflVM>

possible to declare something as public by adding the `public` keyword and placing the declarations after it. In other words:

```
model PublicAndProtected
  Real x; // This is public (because that is the default)
protected
  Real y1; // This is protected
  Real y2; // This is *also* protected
public
  Real z1; // This is public
  Real z2; // This is *also* public
equation
  // ...
end PublicAndProtected;
```

In the examples from this chapter, it was common to see the parameters either at the start of the definition (where they are `public` by default) or in a collection of declarations explicitly marked `public`.

Obviously, parameters are made public so that users of the subsystem can have access to them. We will see shortly, in our discussions on [Modifications](#) (page 301) and [Propagation](#) (page 302), how the values of these parameters should be cascaded to lower level components. But for now, the main point is to recognize that parameter declarations are part of the subsystem pattern.

Connectors

In some sense, connectors are what differentiates a subsystem from a system model. A system model is something that is complete and ready to simulate and, as such, has no connectors because it does not expect to be influenced by anything external. A subsystem can encapsulate a large hierarchy of components (and ultimately, equations). But the fact that it includes connectors indicates that it is really meant to be used as part of some larger system. Furthermore, as we have seen in the examples of this chapter, because these connectors are meant to be connected to, they should be `public`.

Hierarchical Connections

Because subsystems are typically composed exclusively of components or other subsystems, any physical interaction with that subsystem is usually redirected to some nested component or subsystem. We've seen this pattern many times in this chapter where connectors at the subsystem boundary really act as "proxies" for internal connectors. A simple example of this can be seen in the `GearWithBacklash` model:

```
connect(flange_a, inertia_a.flange_a)
```

Note that this `connect` statement connects `flange_a`, a connector instance that belongs to the subsystem, with `inertia_a.flange_a`, a connector instance belonging to the subcomponent `inertia_a`. This is a common pattern in subsystem models and it can be easily recognized because one of the connectors named in the `connect` statement includes a `.` and the other one does not. All the "internal" connections that wire together internal components directly to other internal components will have a `.` for both of the named connectors in the `connect` statement, *e.g.*,

```
connect(idealGear.flange_b, inertia_b.flange_a)
```

Equation Generation for Hierarchical Connections

In this book, the point has been made repeatedly (*e.g.*, in our discussion of [Acausal Connections](#) (page 169)) that the sign convention for `flow` variables is such that a positive value for the `flow` variable represents a flow into the component or subsystem. At the same time, we also pointed out that [Connections](#) (page 259) generate equations that sum all the corresponding `flow` variables in the connection set to zero.

But, when dealing with hierarchical subsystem definitions there is a modification to this rule. For a subsystem, the sign convention for `flow` variables remains intact. So, a positive value for the `flow` variable on a connector still represents a flow of a conserved quantity into that component. And it is still the case that a conservation equation will be generated for each `flow` variable in a connection set. However, in that conservation equation, the sign for `flow` variables on connectors belonging to internal components will have the opposite sign as `flow` variables on connectors belonging to the subsystem itself.

To understand the implications of this, let us consider the following two `connect` statements from the `GearWithBacklash` model:

```
connect(flange_a, inertia_a.flange_a)
annotation (Line(points={{-80,0},{-100,0}},
color={0,0,0}, smooth=Smooth.None));
connect(idealGear.flange_b, inertia_b.flange_a)
annotation (Line(points={{10,0},{40,0}},
color={0,0,0}, smooth=Smooth.None));
```

From these equations, we get the following two connection sets:

- **Connection Set #1:** `flange_a, inertia_a.flange_a`
- **Connection Set #2:** `idealGear.flange_b, inertia_b.flange_a`

In each of these connection sets, there is a `flow` variable, `tau`. Using the rules for *Connections* (page 259) described previously, we might expect the following two equations to be generated for the `flow` variables on these connectors:

```
flange_a.tau + inertia_a.flange_a.tau = 0;
idealGear.flange_b.tau + inertia_b.flange_a = 0;
```

However, in our previous discussion on *Connections* (page 259), all the components were at the same hierarchical level (*i.e.*, as `idealGear.flange_b` and `inertia_b.flange_a` are). But with subsystem models, this isn't always the case. And, as described a moment ago, for cases where the connections are at different levels, we need to introduce a sign difference for contributions at different levels. Taking that into account, the actual equations that will be generated will be:

```
-flange_a.tau + inertia_a.flange_a.tau = 0;
idealGear.flange_b.tau + inertia_b.flange_a = 0;
```

Note the minus sign in front of the `flange_a.tau`. Remember that `flange_a` is meant to be acting as a proxy for `inertia_a.flange_a`. If that is the case, then by changing the sign of `flange_a.tau`, the first equation above can be transformed into:

```
inertia_a.flange_a.tau = flange_a.tau;
```

In other words, by introducing the sign change on `flange_a.tau`, any conserved quantity flows in through `flange_a` also flows into `inertia_a.flange_a`, which is exactly what we expect from this kind of “proxy” relationship.

Implementation

A subsystem model is typically just a wrapper around a collection of components. As we've discussed already in this section, the parameters and connectors exposed by the subsystem are `public` because that is how users of the subsystem will interact with it.

The actual internal details of the subsystem represent the implementation of the subsystem. This implementation is generally a collection of components and other subsystems that are connected together with one or more of their connectors connected to the subsystem's connectors.

Normally, these implementation details are best hidden from the end user of the subsystem. To accomplish this, all `non-parameter` declarations in a subsystem are typically marked as `protected`. There are two main reasons for doing this. First, it hides implementation details from the users of the subsystem

model. This has the effect of simplifying the interface down to just parameters and connectors and avoids mixing things the user really needs to know with things that they do not need to know (or even should not know). Another reason to make the implementation details `protected` is to provide the flexibility to improve or refactor the implementation in the future. If users are allowed to reference the implementation details, that means they will then (perhaps even unintentionally) become `dependent` on them. As a result, if they change in the future it will break the end user's models.

Arrays of Component

Several of the examples in this chapter used arrays of components. Arrays of components are useful when the user might want to “scale” the number of components used using a parameter (as we saw in our discussions of both *Spatially Distributed Heat Transfer* (page 287) and *Harmonic Motion of Pendulums* (page 294)).

Creating an array of components is really no different from creating an array of scalar variables like we did in our previous discussion on *Vectors and Arrays* (page 87). As we can see in this example,

```
HTC.HeatCapacitor capacitance[n] (each final C=C/n, each T(start=T0, fixed=true))
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  origin={-30,20})));
HTC.ThermalConductor wall[n] (each final G=G_wall/n)
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}},
  rotation=90, origin={-30,-20})));
HTC.ThermalConductor rod_conduction[n-1] (each final G=G_rod*(n-1))
annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
```

the syntax for creating an array of components is identical to the syntax used for other types. All that is required is to follow the name of the variable being declared by a set of dimensions.

However, unlike scalars, components have other declarations inside them. So whenever an array of components is created, the structure of that component is replicated for each component in the array. Modelica imposes a restriction that in every array, every element must be the same type. This may seem obvious, but that is partly because we haven't discussed `replaceable` components yet. We'll learn more about `replaceable` components in the next chapter when we talk about Modelica's *Configuration Management* (page 346) features. But for now we will simply point out that it is not possible to `redeclare` only one element in an array.

As we touched on briefly in our discussion of *Harmonic Motion of Pendulums* (page 294), when we make *Modifications* (page 301) to arrays of components, each variable inside the component is implicitly treated as an array. For example, consider the following `record` definition:

```
record Point
  Real x;
  Real y;
  Real z;
end Point;
```

If we were to declare an array of `Point` components, *e.g.*, `Point p[5]`, any reference to `p.x` is treated as an array of 5 `Real` variables, *i.e.*, `p[1].x`, `p[2].x`, `p[3].x`, `p[4].x` and `p[5].x`. This is called *slicing*. The bottom line is that if we leave off a subscript (*e.g.*, `p.x`), it gets “pushed to the end” (or more technically, it is left “unbound” and can be “bound” later). Also, if a subscript is supplied as a range (*e.g.*, `: 1:end`, `2:3`), then the expression resolves to a subset of the array corresponding to the indices in the range. All of this holds even for arrays of components containing arrays of components and so on.

The following example, demonstrates some of the more common cases:

```
record Vector3D
  Real x[3];
end Vector3D;
```

```

model ArrayExample
  Point p[2];
  Point q[2,3];
  Vector3D v[4];
equation
  p.x = {1, 2}; // p[1].x = 1, p[2].x = 2
  q[:,3].y = {4, 5}; // q[1,3].y = 4, q[2, 3].y = 5;
  q.x = [1, 2, 3; 4, 5, 6] // q[1,1].x = 1,
                            // q[1,2].x = 2,
                            // q[2,3].x = 6
  v.x[1] = {1, 2, 3, 4}; // v[1].x[1] = 1, v[2].x[1] = 2,
                        // v[3].x[1] = 3, v[4].x[1] = 4
  v[:,].x[1] = {1, 2, 3, 4}; // v[1].x[1] = 1, v[2].x[1] = 2,
                            // v[3].x[1] = 3, v[4].x[1] = 4
  v[2:3].x[1] = {2, 3}; // v[2].x[1] = 2, v[3].x[1] = 3
  v[1].x = {1, 2, 3}; // v[1].x[1] = 1, v[1].x[2] = 2,
                      // v[1].x[3] = 3
end ArrayExample;

```

Modifications

Previously, we've seen examples of modifications applied to variables. In some cases, these modifications are applied to *Attributes* (page 30) of built-in types, *e.g.*,

```
Real x(start=2, min=1);
```

In other cases, they have been applied to `model` instances to change the values of parameters for that particular instance, *e.g.*,

```
StepVoltage Vs(V0=0, Vf=24, stepTime=0.5);
```

But it is also worth pointing out that such modifications can reach down deeper into the hierarchy than simply one level. For example, consider the previous example involving a `StepVoltage` component. We could also have made a modification to the `min` attribute associated with the `Vf` parameter in the `Vs` instance of the `StepVoltage` model as follows:

```
StepVoltage Vs(V0=0, Vf(min=0), stepTime=0.5);
```

But what if we wanted to change an attribute of the `Vf` parameter **and** give it a value? The syntax for such a modification is:

```
StepVoltage Vs(V0=0, Vf(min=0)=24, stepTime=0.5);
```

An important case worth discussion, with regards to modifications, is how modifications are performed on **arrays** of components. Imagine we had an array of `StepVoltage` components declared as follows:

```
StepVoltage Vs[5];
```

As we saw in our discussion of *Arrays of Component* (page 300), this is not only legal Modelica, but it can be useful to represent a collection of components within a subsystem. If we want to give the parameter `Vf` a value, we have two choices. The first is to specify an array of values, *e.g.*,

```
StepVoltage Vs[5](Vf={24,26,28,30,32});
```

This assigns the values in the vector `{24,26,28,30,32}` to `Vs[1].Vf`, `Vs[2].Vf`, `Vs[3].Vf`, `Vs[4].Vf` and `Vs[5].Vf`, respectively. The other choice we have is to give the same value to every element in the array. We could use this same array initialization syntax, *e.g.*,

```
StepVoltage Vs[5](Vf={24,24,24,24,24});
```

The problem comes when the number of elements in an array is defined by a `parameter`, *e.g.*,

```
parameter Integer n;
StepVoltage Vs[n](Vf/* ??? */);
```

If we tried to initialize `Vf` with a literal array (*e.g.*, `{24,24,24}`), then it won't adapt to changes in `n`. To address this situation, we could use the `fill` (page 105) function:

```
parameter Integer n;
StepVoltage Vs[n](Vf=fill(24, n));
```

This is an acceptable solution. But imagine if we wanted to modify both the value of `Vf` and the `min` attribute inside `Vf`? We'd end up with something like this:

```
parameter Integer n;
StepVoltage Vs[n](Vf(min=fill(0,n))=fill(24, n));
```

With nested modifications, this kind of thing can get complicated quickly. Fortunately, Modelica includes a feature to deal with such situations. By placing the `each` keyword in front of a modification, that modification is applied to every instance, *e.g.*,

```
parameter Integer n;
StepVoltage Vs[n](each Vf(min=0)=24);
```

Modifications are an essential part of modeling because they allow us to modify the parameter values down through the hierarchy. As you can see from the examples in this section, Modelica provides many features to make applying modifications to hierarchies simple and powerful.

Propagation

When building subsystem models, it is extremely common for a subsystem to contain parameters that it then propagates or cascades down to its components. For example, consider the following system model used in our discussion of *Basic Rotational Components* (page 196):

```
within ModelicaByExample.Components.Rotational.Examples;
model SMD
  Components.Damper damper2(d=1);
  Components.Ground ground;
  Components.Spring spring2(c=5);
  Components.Inertia inertia2(J=1,
    phi(fixed=true, start=1),
    w(fixed=true, start=0));
  Components.Damper damper1(d=0.2);
  Components.Spring spring1(c=11);
  Components.Inertia inertia1(
    J=0.4,
    phi(fixed=true, start=0),
    w(fixed=true, start=0));
equation
  // ...
end SMD;
```

If we wanted to use this model in different contexts where the values of the component parameters, like `d`, might vary, we could make `d` a parameter at the subsystem level and then propagate it down into the hierarchy using a modification. The result would look something like this:

```
within ModelicaByExample.Components.Rotational.Examples;
model SMD
```

```

import Modelica.SIunits.*;
parameter RotationalDampingConstant d;
Components.Damper damper2(d=d);
// ...

```

There is one complication here. It is possible for a user to come along and change the value of `damper2.d` instead of modifying the `d` parameter in the SMD model. To avoid having the `d` parameter and the `damper2.d` parameter from getting out of sync (having different values), we can permanently bind them using the `final` qualifier:

```

within ModelicaByExample.Components.Rotational.Examples;
model SMD
  import Modelica.SIunits.*;
  parameter RotationalDampingConstant d;
  Components.Damper damper2(final d=d);
// ...

```

By adding the `final` qualifier, we are indicating that it is no longer possible to modify the value of `damper2.d`. Any modifications must be made to `d` only.

Giving all of the “hard-wired” numerical values in the SMD model the same treatment, we would end up with a highly reusable model like this:

```

within ModelicaByExample.Components.Rotational.Examples;
model SMD
  import Modelica.SIunits.*;

  parameter RotationalDampingConstant d1, d2;
  parameter RotationalSpringConstant c1, c2;
  parameter Inertia J1, J2;
  parameter Angle phi1_init=0, phi2_init=0;
  parameter AngularVelocity w1_init=0, w2_init=0;

  Components.Damper damper2(final d=d2);
  Components.Ground ground;
  Components.Spring spring2(final c=c2);
  Components.Inertia inertia2(
    final J=J2,
    phi(fixed=true, final start=phi2_init),
    w(fixed=true, final start=w2_init));
  Components.Damper damper1(final d=d1);
  Components.Spring spring1(final c=c1);
  Components.Inertia inertia1(
    final J=J1,
    phi(fixed=true, final start=phi1_init),
    w(fixed=true, final start=w1_init));
equation
  // ...
end SMD;

```

If we wanted to use a specific set of parameter values, we could do it in one of two ways. One way would be to extend the parameterized model above and include a modification in the `extends` statement, *e.g.*,

```

model SpecificSMD
  extends SMD(d2=1, c2=5, J2=1,
              d1=0.5, c1=11, J1=0.4,
              phi1_init=1);

```

Note that we did not need to include modifications for the values of `phi2_init`, `w1_init` and `w2_init`, since those parameters were declared with default values. In general, **default values for parameters should only be used when those defaults are reasonable for the vast majority of cases**. The reason for this is that if a parameter has no default value most Modelica compilers will generate a warning

alerting you that a value is required. But if a default value is there, it will silently use the default value. If that default value is not reasonable or typical, then you will silently introduce an unreasonable value into your model.

But returning to the topic of propagation, the other approach that could be used would be to instantiate an instance of the SMD model and use modifications on the declared variable to specify parameter values, *e.g.*,

```
SMD mysmd(d2=1, c2=5, J2=1,
           d1=0.5, c1=11, J1=0.4,
           phi1_init=1);
```

We'll defer the discussion on which of these approaches is better until the upcoming chapter on *Architectures* (page 304).

2.5 Architectures

At the start of this book, we looked at how to write equations and transform those into simulations. That, by itself, is very interesting because we were able to avoid having to worry about how we would solve the resulting linear and non-linear systems of equations or how to integrate the resulting differential equations. However, writing complex systems in terms of individual equations does not scale well.

So, we then explored the features of Modelica that allow us to create component models so that we could reuse these equations without having to take the time to write them out in every context where they would be used. Not only did this allow us to compose systems from pre-defined (and presumably tested) component models, it also allowed us, through the use of Modelica's standard graphical annotations, to compose and represent systems graphically.

This too has scalability issues because building complex models strictly from components requires a great deal of dragging, dropping and connecting of components. Furthermore, system models can become large and complex without any kind of hierarchy. This is yet another limitation to scalability. To address this issue, we examined how to define reusable subsystem models that, instead of containing equations to be reused, contained reusable assemblies of components and other subsystems. In this way, we could drag, drop and connect common assemblies of components into reusable assemblies. This minimized the amount of dragging, dropping and connecting that was required to build complex system models.

Each step in this progression has shown how to reduce the amount of tedious, time consuming and potentially error prone work we need to perform in order to build system models. This chapter represents the last step along this progression. Here, we will learn about architectures. Architectures are models where a collection of subsystems have been **pre-connected** and the composition of the system is done by simply selecting specific implementations (models) for each subsystem in our system. In this way, not only do we not need to supply equations, we don't even need to drag, drop and connect components or subsystems. Instead, we only need to choose the specific model to use for each particular subsystem.

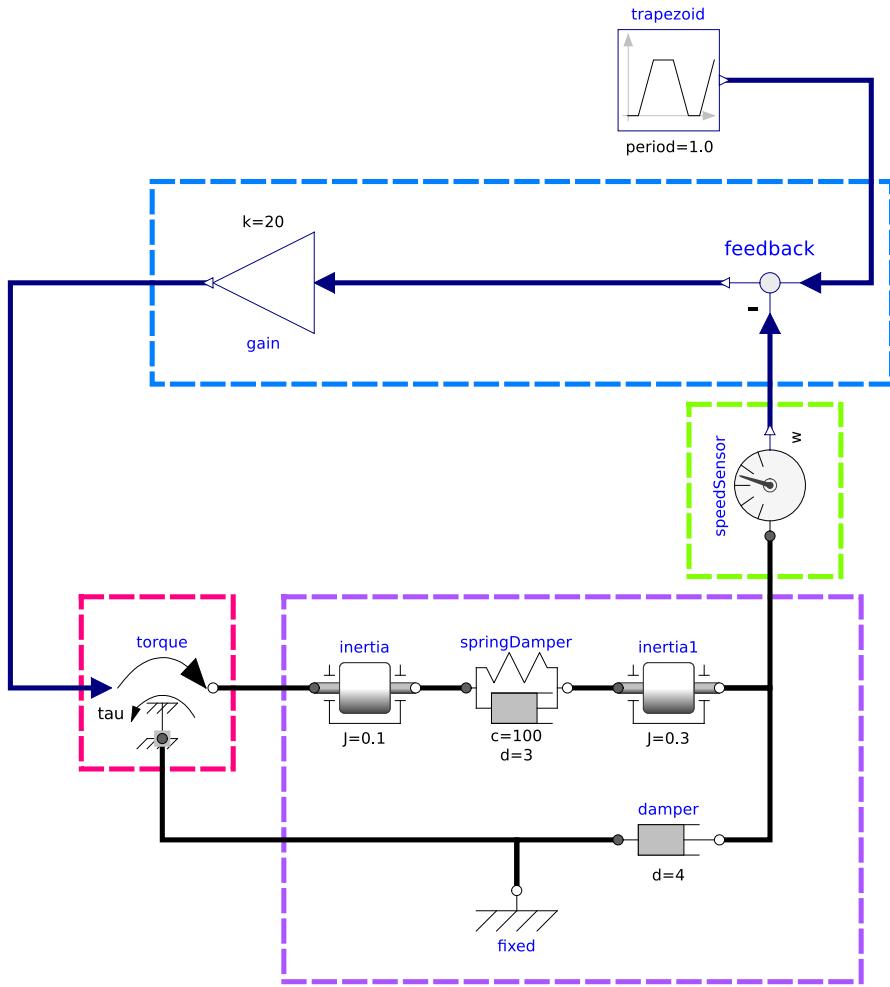
2.5.1 Examples

Sensor Comparison

Let us start our study of architectures with an example that is similar to one presented in my previous book [[Tiller2001](#)] (page 357). In it, we will consider the performance of a control system using several different sensor models.

Flat System

Our system schematic is structured as follows:



All the components within the magenta box represent the plant model. In this case, it is a simple rotational system involving two rotating inertias connected via a spring and a damper. One of the inertias is connected to a rotational ground by an additional damper. The green box identifies the sensor in the system. The sensor is used to measure the speed of one of the rotating shafts. Similarly, the magenta box identifies the actuator. The actuator applies a torque to the other shaft (the one whose speed is not measured). Finally, all the components in the blue box represent the control system, which tries to keep the measured speed as close as possible to the setpoint supplied by the signal generator at the top of the diagram.

We can represent this using the following Modelica code:

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
model FlatSystem "A rotational system with no architecture"
  Modelica.Mechanics.Rotational.Components.Fixed fixed
    annotation (Placement(transformation(extent={{10,-90},{30,-70}})));
  Modelica.Mechanics.Rotational.Components.Inertia inertia(J=0.1)
    annotation (Placement(transformation(extent={{-20,-50},{0,-30}})));
  Modelica.Mechanics.Rotational.Components.Inertia inertia1(J=0.3)
    annotation (Placement(transformation(extent={{40,-50},{60,-30}})));
  Modelica.Mechanics.Rotational.Sources.Torque torque(useSupport=true)
    annotation (Placement(transformation(extent={{-60,-50},{-40,-30}}));
  Modelica.Mechanics.Rotational.Components.SpringDamper springDamper(c=100, d=3)
    annotation (Placement(transformation(extent={{10,-50},{30,-30}}));
  Modelica.Mechanics.Rotational.Components.Damper damper(d=4)
    annotation (Placement(transformation(extent={{40,-80},{60,-60}}));
```

```

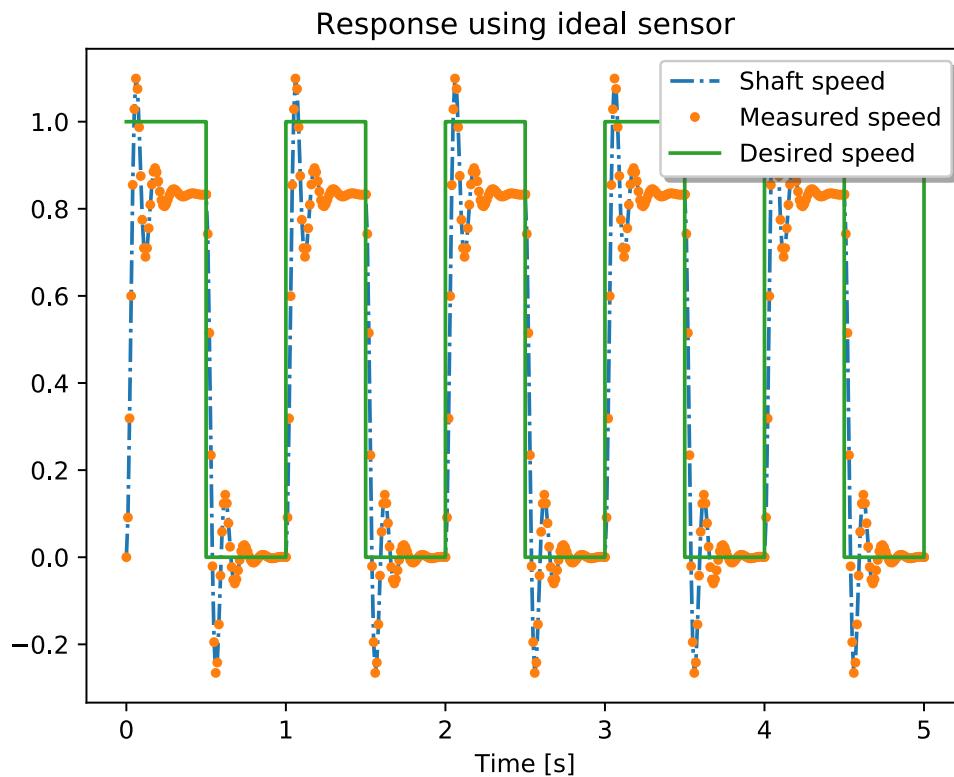
Modelica.Mechanics.Rotational.Sensors.SpeedSensor speedSensor annotation (
    Placement(transformation(
        extent={{-10,-10},{10,10}}, rotation=90, origin={70,0})));
Modelica.Blocks.Math.Feedback feedback annotation (Placement(transformation(
    extent={{10,-10},{-10,10}}, origin={70,40})));
Modelica.Blocks.Sources.Trapezoid trapezoid(period=1.0)
    annotation (Placement(transformation(extent={{40,70},{60,90}})));
Modelica.Blocks.Math.Gain gain(k=20) annotation (Placement(transformation(
    extent={{-10,-10},{10,10}}, rotation=180, origin={-30,40})));
equation
    connect(springDamper.flange_a, inertia.flange_b) annotation (Line(
        points={{10,-40},{0,-40}},
        color={0,0,0}, smooth=Smooth.None));
    connect(springDamper.flange_b, inertia1.flange_a) annotation (Line(
        points={{30,-40},{40,-40}},
        color={0,0,0}, smooth=Smooth.None));
    connect(torque.support, fixed.flange) annotation (Line(
        points={{-50,-50},{-50,-70},{20,-70},{20,-80}},
        color={0,0,0}, smooth=Smooth.None));
    connect(damper.flange_b, inertia1.flange_b) annotation (Line(
        points={{60,-70},{70,-70},{70,-40},{60,-40}},
        color={0,0,0}, smooth=Smooth.None));
    connect(damper.flange_a, fixed.flange) annotation (Line(
        points={{40,-70},{20,-70},{20,-80}},
        color={0,0,0}, smooth=Smooth.None));
    connect(torque.flange, inertia.flange_a) annotation (Line(
        points={{-40,-40},{-20,-40}},
        color={0,0,0}, smooth=Smooth.None));
    connect(speedSensor.flange, inertia1.flange_b) annotation (Line(
        points={{70,-10},{70,-40},{60,-40}},
        color={0,0,0}, smooth=Smooth.None));
    connect(feedback.y, gain.u) annotation (Line(
        points={{61,40},{-18,40}},
        color={0,0,127}, smooth=Smooth.None));
    connect(gain.y, torque.tau) annotation (Line(
        points={{-41,40},{-80,40},{-80,-40},{-62,-40}},
        color={0,0,127}, smooth=Smooth.None));
    connect(trapezoid.y, feedback.u1) annotation (Line(
        points={{61,80},{90,80},{90,40},{78,40}},
        color={0,0,127}, smooth=Smooth.None));
    connect(speedSensor.w, feedback.u2) annotation (Line(
        points={{70,11},{70,32}},
        color={0,0,127}, smooth=Smooth.None));
annotation (
    Diagram(graphics={
        Rectangle(
            extent={{-52,60},{94,20}}, lineColor={0,128,255},
            pattern=LinePattern.Dash, lineThickness=0.5),
        Rectangle(
            extent={{54,16},{84,-18}}, lineColor={128,255,0},
            pattern=LinePattern.Dash, lineThickness=0.5),
        Rectangle(
            extent={{-66,-22},{-36,-56}}, lineColor={255,0,128},
            pattern=LinePattern.Dash, lineThickness=0.5),
        Rectangle(
            extent={{-26,-22},{88,-98}}, lineColor={170,85,255},
            pattern=LinePattern.Dash, lineThickness=0.5}),
        experiment(StopTime=4));
end FlatSystem;

```

Note that for this particular model, the sensor component is an instance of the SpeedSensor model

found in `Modelica.Mechanics.Rotational.Sensors` which is an “ideal” sensor that reports the exact solution trajectory. In other words, it does not introduce any kind of measurement artifact.

In fact, simulating such a system we see that using accurate speed measurements, the control system is still not quite able to follow the speed trace:



We'll discuss why in a moment. But the problem clearly isn't measurement artifact since the measured speed is exactly equal to the actual shaft speed.

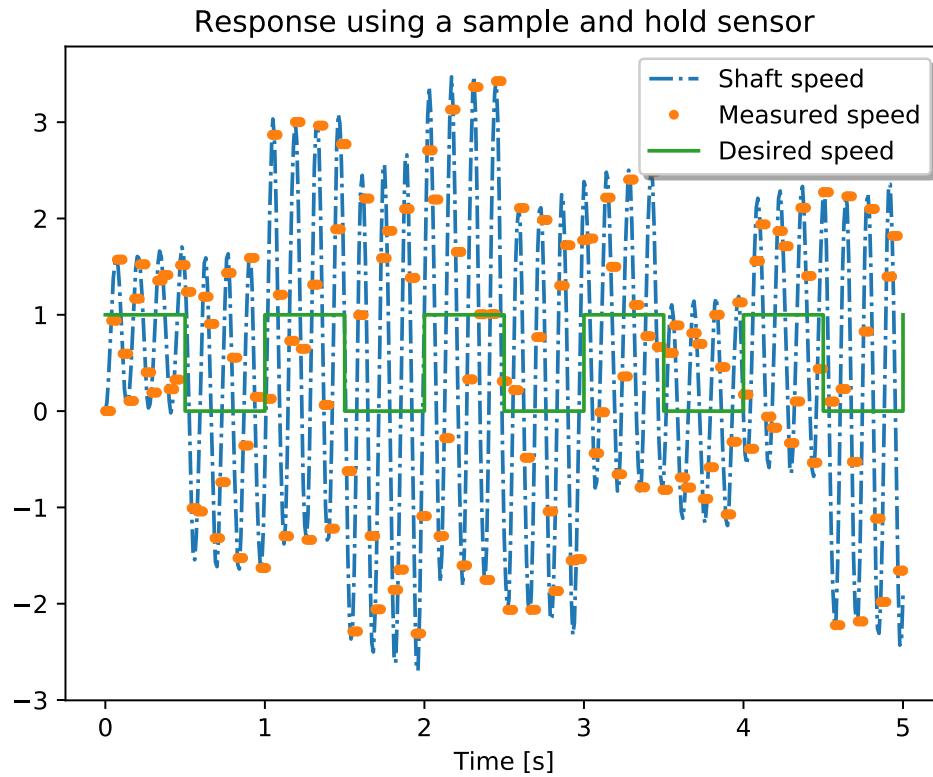
Now, imagine we wanted to use a more realistic sensor model, like the *Sample and Hold Sensor* (page 227) model developed previously, to see what additional impact measurement artifact might have on our system performance. One way we could do that would be to replace the following lines in the `FlatSystem` model:

```
Modelica.Mechanics.Rotational.Sensors.SpeedSensor speedSensor annotation (
  Placement(transformation(
    extent={{-10,-10},{10,10}}, rotation=90, origin={70,0})));
Modelica.Blocks.Math.Feedback feedback annotation (Placement(transformation(
```

with these lines:

```
Components.SpeedMeasurement.Components.SampleHold speedSensor(sample_time=0.036)
annotation (Placement(transformation(
  extent={{-10,-10},{10,10}}, rotation=90, origin={70,0})));
Modelica.Blocks.Math.Feedback feedback annotation (Placement(transformation(
```

Note that **the only change being made here is the type of the speedSensor component**. Simulating this system, we would see the following performance for the control system:

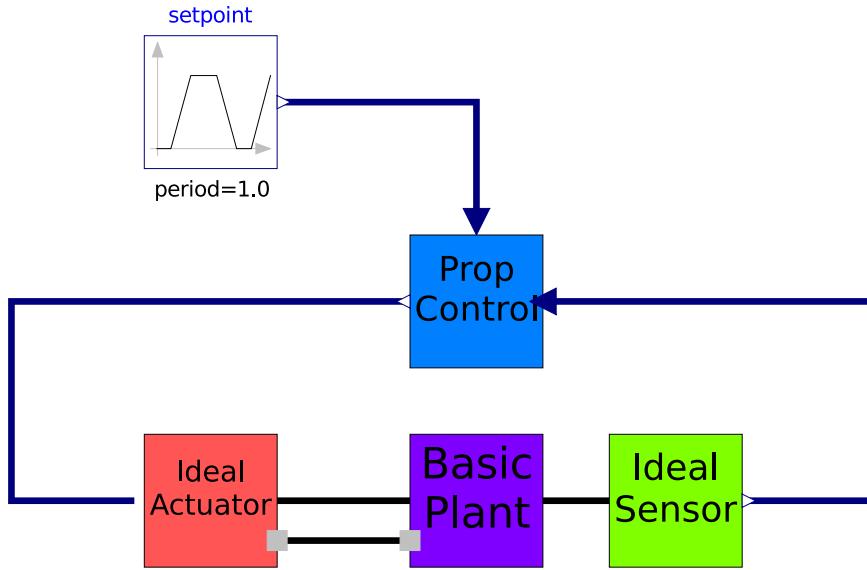


In this case, we can see that things are going from bad to worse. While we were initially unable to track the desired speed closely, now (as a result of the measurement artifact) our system has become unstable.

Hierarchical System

At this point, we'd like to explore this performance issue a bit more to understand how characteristics of the sensor (*e.g.*, `sample_time`) impact performance and potentially what kinds of improvements might ultimately be required for the control system itself.

If we plan on substituting sensors, actuators and control strategies our first step should be to organize our system into those subsystems. Doing so, we end up with the following system model:



Here we see only four subsystems at the system level. They correspond to the subsystems we mentioned a moment ago. Our Modelica model is now much simpler because the only declarations we now have are:

```

Implementation.BasicPlant plant
  annotation (Placement(transformation(extent={{-10,-40},{10,-20}})));
Implementation.IdealActuator actuator
  annotation (Placement(transformation(extent={{-50,-40},{-30,-20}})));
Implementation.IdealSensor sensor
  annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
Implementation.ProportionalController controller
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
Modelica.Blocks.Sources.Trapezoid setpoint(period=1.0)
  annotation (Placement(transformation(extent={{-50,20},{-30,40}})));
  
```

Each subsystem (plant, actuator, sensor and controller) is implemented by a subsystem in the `Implementations` package. This is an improvement because it means that if we wanted to change controller models, we could simply change the type associated with the `controller` subsystem and a different implementation would be used. This is definitely an improvement over having to delete the existing controller components, drag in new ones and then connect everything back up (correctly!).

But switching our sensor model to a sample and hold version still involves editing the text of our model, *i.e.*,

```

Implementation.BasicPlant plant
  annotation (Placement(transformation(extent={{-10,-40},{10,-20}})));
Implementation.IdealActuator actuator
  annotation (Placement(transformation(extent={{-50,-40},{-30,-20}})));
Implementation.SampleHoldSensor sensor(sample_time=0.01)
  annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
Implementation.ProportionalController controller
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
Modelica.Blocks.Sources.Trapezoid setpoint(period=1.0)
  annotation (Placement(transformation(extent={{-50,20},{-30,40}})));
  
```

There are still a couple of issues with this solution. First, recall that we changed the implementation to be used by changing the name of the type. This raises the question “What types *can* be used here?” What if I change the type of `sensor` to `BasicPlant`? That wouldn’t make any sense. But we don’t really know that by looking at the model. But the **bigger** problem is that to create a model that we

end up with two models that are almost identical. As we learned *earlier in the book* (page 200), it is always useful to keep in mind the DRY (Don't Repeat Yourself) principle. In these models, we see a lot of redundancy.

Dealing with Redundancy

Imagine we start with our hierarchical model using the `IdealSensor` model:

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
model HierarchicalSystem "Organizing components into subsystems"
    Implementation.BasicPlant plant
        annotation (Placement(transformation(extent={{-10,-40},{10,-20}})));
    Implementation.IdealActuator actuator
        annotation (Placement(transformation(extent={{-50,-40},{-30,-20}})));
    Implementation.IdealSensor sensor
        annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
    Implementation.ProportionalController controller
        annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
    Modelica.Blocks.Sources.Trapezoid setpoint(period=1.0)
        annotation (Placement(transformation(extent={{-50,20},{-30,40}}));
equation
    connect(actuator.shaft, plant.flange_a) annotation (Line(
        points={{-30,-30},{-10,-30}},
        color={0,0,0},
        smooth=Smooth.None));
    connect(actuator.housing, plant.housing) annotation (Line(
        points={{-30,-36},{-10,-36}},
        color={0,0,0},
        smooth=Smooth.None));
    connect(plant.flange_b, sensor.shaft) annotation (Line(
        points={{10,-30},{20,-30}},
        color={0,0,0},
        smooth=Smooth.None));
    connect(controller.command, actuator.tau) annotation (Line(
        points={{-11,0},{-70,0},{-70,-30},{-52,-30}},
        color={0,0,127},
        smooth=Smooth.None));
    connect(sensor.w, controller.measured) annotation (Line(
        points={{41,-30},{60,-30},{60,0},{10,0}},
        color={0,0,127},
        smooth=Smooth.None));
    connect(setpoint.y, controller.setpoint) annotation (Line(
        points={{-29,30},{0,30},{0,12}},
        color={0,0,127},
        smooth=Smooth.None));
end HierarchicalSystem;
```

Now, we want to create a variation of this model where the `sensor` component can assume different types.

Previously, we dealt with redundancy by using inheritance. We can certainly use inheritance to pull the subsystems from `HierarchicalSystem` into another model and then change *parameters*, e.g.,

```
model Variation1
    extends HierarchicalSystem(setpoint(startTime=1.0));
end Variation1;
```

But we don't want to change parameters, we want to change **types**. If we could somehow “overwrite” previous choices, we could do something like this:

```
model Variation2 "What we'd like to do (but cannot)"
    extends HierarchicalSystem(setpoint(startTime=1.0));
```

```
Implementation.SampleHoldSensor sensor
annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
end Variation2;
```

This is essentially what we want to do. But this isn't legal in Modelica and there are a couple of other problems with this approach. The first is that the original model developer might not want to allow such changes. The second problem is that we end up with two `sensor` components (of different types, no less). Even if it truly "overwrites" any previous declaration of the `sensor` component, another problem is that we might type the name wrong for the variable name and end up with two sensors. Finally, we still don't have a way to know if it even makes sense to change `sensor` into a `SampleHoldSensor`. In this case it does, but how do we ensure that in general?

Fortunately, there **is** a way to do almost exactly what we are attempting here. But in order to address these other complications, we need to be a bit more rigorous about it.

We need to start by indicating that that component is allowed to be replaced. We can do this by declaring the `sensor` as follows:

```
replaceable Implementation.IdealSensor sensor
annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
```

The use of the `replaceable` keyword indicates that when we inherit from the model, that variable's type can be changed (or, "redeclared"). But remember that this model also has a statement like this one:

```
connect(plant.flange_b, sensor.shaft);
connect(sensor.w, controller.measured);
```

This introduces a complication. What happens if we replace the `sensor` component with something that **does not have** a `w` connector on it? In that case, this `connect` statement will generate an error. In this case, we say that the two sensor models are not **plug-compatible**. A model X is plug-compatible with a model Y if for every **public** variable in Y, there is a corresponding public variable in X with the same name. Furthermore, every such variable in X must itself be plug-compatible with its counterpart in Y. This ensures that if you change a component of type Y into a component of type X and everything you need (parameters, connectors, etc) will still be there and will still be compatible. **However, please note** that if X is plug-compatible with Y, this **does not** imply that Y is plug-compatible with X (as we will see in a moment).

Plug compatibility is important because, in general, we want to ensure that any redeclaration that we make is "safe". The way we do that is to require that whatever type we decide to change our `sensor` component to is plug compatible with the original type. In this case, that means that it has to have a `w` connector (and that connector, in turn, must be plug compatible with the `w` that was there before) and it has to have a `shaft` connector (which, again, must be plug compatible with the previous `shaft`).

So the question then is, does our `SampleHoldSensor` implementation satisfy this requirement plug compatibility requirement? Is it plug-compatible with the `IdealSensor` model? First, let's look at the `IdealSensor` model:

```
within ModelicaByExample.Architectures.SensorComparison.Implementation;
model IdealSensor "Implementation of an ideal sensor"
  Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
    "Flange of shaft from which sensor information shall be measured"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
protected
  Modelica.Mechanics.Rotational.Sensors.SpeedSensor idealSpeedSensor
    "An ideal speed sensor" annotation (Placement(transformation(
      extent={{-10,-10},{10,10}}));
```

The public interface of this component consists only of the two connectors `w` and `shaft`. Looking at the `SampleHoldSensor` model:

```

within ModelicaByExample.Architectures.SensorComparison.Implementation;
model SampleHoldSensor "Implementation of a sample hold sensor"
  parameter Modelica.SIunits.Time sample_time(min=Modelica.Constants.eps);
  Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
    "Flange of shaft from which sensor information shall be measured"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
protected
  Components.SpeedMeasurement.Components.SampleHold sampleHoldSensor(
    sample_time=sample_time)
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));

```

we see that its public interface also contains the connectors `w` and `shaft`. Furthermore, they are exactly the same type as the connectors on the `IdealSensor` model. For this reason, the `SampleHoldSensor` model is plug-compatible with the `IdealSensor` model so we should be able to replace an `IdealSensor` instance with a `SampleHoldSensor` instance and our `connect` statements will still be valid.

So, if our `HierarchicalSystem` model were declared as follows:

```

within ModelicaByExample.Architectures.SensorComparison.Examples;
model HierarchicalSystem "Organizing components into subsystems"
  replaceable Implementation.BasicPlant plant
    annotation (Placement(transformation(extent={{-10,-40},{10,-20}})));
  replaceable Implementation.IdealActuator actuator
    annotation (Placement(transformation(extent={{-50,-40},{-30,-20}})));
  replaceable Implementation.IdealSensor sensor
    annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
  replaceable Implementation.ProportionalController controller
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  replaceable Modelica.Blocks.Sources.Trapezoid setpoint
    annotation (Placement(transformation(extent={{-50,20},{-30,40}})));
  // ...
end HierarchicalSystem;

```

Then we can achieve our original goal of creating a variation of this model without repeating ourselves as follows:

```

model Variation3 "DRY redeclaration"
  extends HierarchicalSystem(
    redeclare Implementation.SampleHoldSensor sensor
  );
end Variation3;

```

There are several things worth noting about this model. The first is that the syntax of a redeclaration is just like a normal declaration except it is preceded by the `redeclare` keyword. Also note that the redeclaration is part of the `extends` clause. Specifically, it is a modification, like any other modification, in the `extends` clause. If we wanted to both redeclare the `sensor` component and change the `startTime` parameter of our setpoint, they would both be modifications of the `extends` clause, *e.g.*,

```

model Variation3 "DRY redeclaration"
  extends HierarchicalSystem(
    setpoint(startTime=1.0),
    redeclare Implementation.SampleHoldSensor sensor
  );
end Variation3;

```

Constraining Types

Recall, from earlier in this section, that the public interface for the `SampleHoldSensor` model included:

```
parameter Modelica.SIunits.Time sample_time=0.01;
Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft;
Modelica.Blocks.Interfaces.RealOutput w;
```

and that the `IdealSensor` public interface contained only:

```
Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft;
Modelica.Blocks.Interfaces.RealOutput w;
```

If redeclarations are restricted in such a way that the redeclared type has to be plug-compatible with the original type, then we could run into the following problem. What if our initial model for our system used the `SampleHoldSensor` sensor, *i.e.*,

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
model InitialSystem "Organizing components into subsystems"
  replaceable Implementation.BasicPlant plant;
  replaceable Implementation.IdealActuator actuator;
  replaceable Implementation.SampleHoldSensor sensor;
  replaceable Implementation.ProportionalController controller;
  replaceable Modelica.Blocks.Sources.Trapezoid setpoint;
equation
  // ...
  connect(plant.flange_b, sensor.shaft);
  connect(sensor.w, controller.measured);
  // ...
end InitialSystem;
```

Imagine further that we then wanted to redeclare the `sensor` component to be an `IdealSensor`, *e.g.*,

```
model Variation4
  extends InitialSystem(
    setpoint(startTime=1.0),
    redeclare Implementation.IdealSensor sensor // illegal
  );
end Variation4;
```

Now we have a problem. The problem is that our original `sensor` component has a parameter called `sample_time`. But, we are trying to replace it with something that does not have that parameter. In other words, the `IdealSensor` model is **not** plug-compatible with the `SampleHoldSensor` model because it is missing something, `sample_time`, that the original model, `SampleHoldSensor`, had.

But when we look at source code of the `InitialSystem` model, we see that the `sample_time` parameter was never used. So there is no real reason why we couldn't switch the type. For this reason, Modelica includes the notion of a *constraining type*.

The important thing to understand about redeclarations is that there are really two important types associated with the original declaration. The first type is what the type of the original declaration was. The second type is what the type *could be* and still work. This second type is called the constraining type because as long as any redeclaration is plug-compatible with the constraining type, the model should still work. So in our `InitialSystem` model above, the type of the original declaration was `SampleHoldSensor`. But the model will still work as long as any new type is plug-compatible with `IdealSensor`.

When we indicate that a component is `replaceable`, we can indicate the constraining type by adding a `constrainedby` clause at the end, *e.g.*,

```
replaceable Implementation.SampleHoldSensor sensor
  constrainedby Implementation.IdealSensor;
```

This declaration says that the `sensor` component can be redeclared by anything that is plug-compatible with the `IdealSensor` model, **but** if it isn't redeclared, then **by default** it should be declared as a `SampleHoldSensor` sensor. For this reason, the original type used in the declaration, `SampleHoldSensor`, is called the **default type**.

Recall that our original definition of the `InitialSystem` model didn't specify a constraining type. It only specified the initial type. In that case, the default type and the constraining type are assumed to be the initial type.

We will continue using this same system architecture in the next section when we discuss how to develop such a system model using a top-down, *Architecture Driven Approach* (page 314).

Architecture Driven Approach

So far in this section, we started with a flat approach and gradually adapted it to use the architectural features of Modelica. Let's start to build out our system from the top down using an architectural approach where we define the structure of our system first and then add specific subsystem implementations after that.

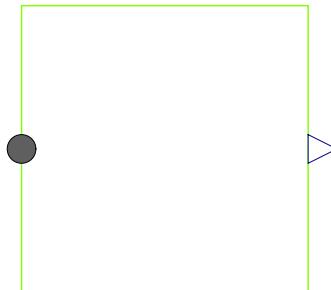
Interfaces

We'd like to start by building our top level architecture model that describes the subsystems we have and how they are connected. However, we need **something** to put into this architecture to represent our subsystems. We don't (yet) want to get down to the job of creating implementations for all of our subsystem models. So where do we start?

The answer is that we start by describing the interfaces of our subsystems. Remember, from earlier in this section, that redeclarations depend on a constraining type and that all implementations must conform to that constraining type? Well an interface is essentially a formal definition of the constraining type (*i.e.*, the expected public interface) but without any implementation details. Since it has no implementation details (*i.e.*, no equations or sub-components), it will end up being a **partial** model. But that's fine for our purposes.

Let's start by considering the interface for our `sensor` subsystem. We have already discussed the public interface in detail. Here is a Modelica definition and icon for that interface:

```
within ModelicaByExample.Architectures.SensorComparison.Interfaces;
partial model Sensor "Interface for sensor"
  Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
    "Flange of shaft from which sensor information shall be measured"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
    annotation (Placement(transformation(extent={{100,-10},{120,10}}));
    annotation (Icon(graphics={Rectangle(extent={{-100,100},{100,-100}}, lineColor={128,255,0}})));
end Sensor;
```

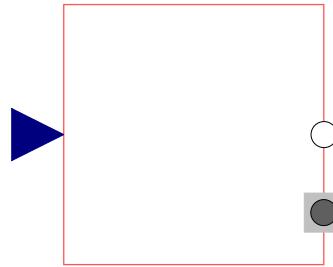


A few things to note about this model definition. The first is, as we mentioned a moment ago, that this model is **partial**. This is because it has connectors but no equations to help solve for the variables on those connectors. Another thing worth noting is the fact that it contains annotations. The annotations associated with the connector declarations specify how those connectors should be rendered. These annotations will be inherited by any model that `extends` from this one. So it is important to choose

locations that will make sense across all implementations. It also includes an `Icon` annotation. This essentially defines a “backdrop” for the final implementations icon. In other words, any model that extends from this model will be able to draw **additional** graphics on top of what is defined by the `Sensor` model.

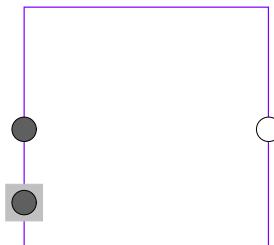
The interface definition for the actuator model is almost identical except that it includes two rotational connectors, one to apply the commanded torque to and the other that handles the reaction torque. If, for example, our actuator model were an electric motor, the former would be the connector for the rotor and the latter would be the connector for the stator. Otherwise, it is very similar to our `Sensor` definition:

```
within ModelicaByExample.Architectures.SensorComparison.Interfaces;
partial model Actuator "Interface for actuator"
  Modelica.Mechanics.Rotational.Interfaces.Flange_b shaft "Output shaft"
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Support housing
    "Connection to housing"
    annotation (Placement(transformation(extent={{90,-70},{110,-50}})));
  Modelica.Blocks.Interfaces.RealInput tau "Input torque command"
    annotation (Placement(transformation(extent={{-140,-20},{-100,20}}));
  annotation (Icon(graphics={Rectangle(extent={{-100,100},{100,-100}}, lineColor={255,85,85}))));
end Actuator;
```



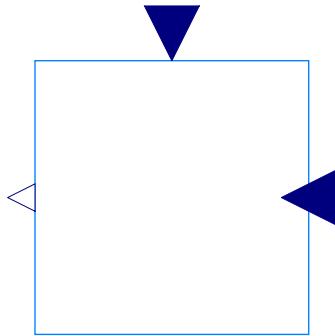
The `Plant` interface has three rotational connectors. One for the “input” side (where the actuator will be connected), one for the “output” side (where the sensor will be connected) and the last one for the “support” side (so it can be “mounted” to something):

```
within ModelicaByExample.Architectures.SensorComparison.Interfaces;
partial model Plant "Interface for plant model"
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
    "Output shaft of plant"
    annotation (Placement(transformation(extent={{90,-10},{110,10}}));
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    "Input shaft for plant"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}}));
  Modelica.Mechanics.Rotational.Interfaces.Support housing
    "Connection to mounting"
    annotation (Placement(transformation(extent={{-110,-70},{-90,-50}}));
  annotation (Icon(graphics={Rectangle(extent={{-100,100},{100,-100}}, lineColor={128,0,255})}));
end Plant;
```



Finally, we have the `Controller` interface definition:

```
within ModelicaByExample.Architectures.SensorComparison.Interfaces;
partial model Controller "Interface for controller subsystem"
  Modelica.Blocks.Interfaces.RealInput setpoint "Desired system response"
    annotation (Placement(transformation(
      extent={{-20,-20},{20,20}},
      rotation=270,
      origin={0,120})));
  Modelica.Blocks.Interfaces.RealInput measured "Actual system response"
    annotation (Placement(transformation(
      extent={{-20,-20},{20,20}},
      rotation=180,
      origin={100,0})));
  Modelica.Blocks.Interfaces.RealOutput command "Command to send to actuator"
    annotation (Placement(transformation(
      extent={{-10,-10},{10,10}},
      rotation=180,
      origin={-110,0})));
    annotation (Icon(graphics={Rectangle(extent={{-100,100},{100,-100}}, lineColor={0,128,255})}));
end Controller;
```



Regardless of how it is implemented (*e.g.*, proportional control, PID control), the `Controller` will require an input connector for the desired setpoint, `setpoint`, another input connector for the measured speed, `measured` and an output connector to send the commanded torque, `command` to the actuator.

Architecture

With the interfaces specified, our architecture model can be written as follows:

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
partial model SystemArchitecture
  "A system architecture built from subsystem interfaces"

  replaceable Interfaces.Plant plant
    annotation (choicesAllMatching=true,
    Placement(transformation(extent={{-10,-50},{10,-30}})));
  replaceable Interfaces.Actuator actuator
    annotation (choicesAllMatching=true,
    Placement(transformation(extent={{-50,-50},{-30,-30}})));
  replaceable Interfaces.Sensor sensor
    annotation (choicesAllMatching=true,
    Placement(transformation(extent={{30,-50},{50,-30}})));
  replaceable Interfaces.Controller controller
    annotation (choicesAllMatching=true,
    Placement(transformation(extent={{-10,-10},{10,10}})));
```

```

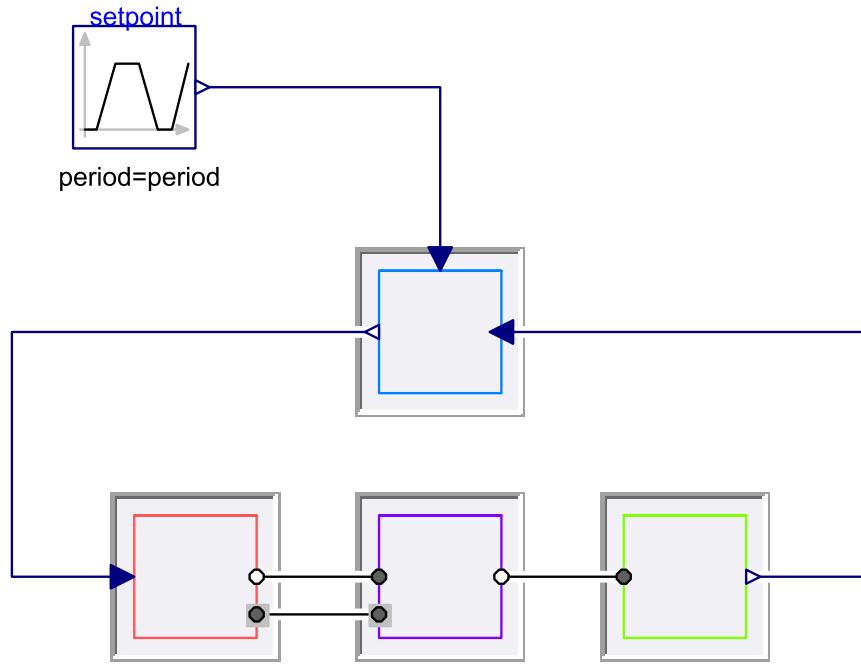
Modelica.Blocks.Sources.Trapezoid setpoint(period=1.0)
  annotation (choicesAllMatching=true,
    Placement(transformation(extent={{-60,30},{-40,50}})));
equation
  connect(actuator.shaft, plant.flange_a) annotation (Line(
    points={{-30,-40},{-10,-40}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(actuator.housing, plant.housing) annotation (Line(
    points={{-30,-46},{-10,-46}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(plant.flange_b, sensor.shaft) annotation (Line(
    points={{10,-40},{30,-40}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(controller.measured, sensor.w) annotation (Line(
    points={{10,0},{70,0},{70,-40},{51,-40}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(controller.command, actuator.tau) annotation (Line(
    points={{-11,0},{-70,0},{-70,-40},{-52,-40}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(setpoint.y, controller.setpoint) annotation (Line(
    points={{-39,40},{0,40},{0,12}},
    color={0,0,127},
    smooth=Smooth.None));
end SystemArchitecture;

```

We can see from the Modelica code that our architecture consists of four `replaceable` subsystems: `plant`, `actuator`, `sensor` and `setpoint`. Each of these declarations includes only one type. As we learned earlier in this section, that type will be used as both the default type and the constraining type (which is what we want). This model also includes all the connections between the subsystems. In this way, it is a complete description of the subsystems and the connections between them. **All that is missing are the implementation choices for each subsystem.**

Note that our `SystemArchitecture` model is, itself, `partial`. This is precisely because the default types for all the subsystems are `partial` and we have not yet specified implementations. In other words, this model has no implementation so it cannot (yet) be simulated. We indicate this by marking it as `partial`.

Like our interface specifications, this model also contains graphical annotations. This is because, in addition to specifying the subsystems, we are also specifying the locations of the subsystems and the paths of the connections. When rendered, our system architecture looks like this:



Implementations

Now that we have the interfaces and the architecture, we need to create some implementations that we can then “inject” into the architecture. These implementations can choose to either inherit from the existing interfaces (thus avoiding redundant code) or simply ensure that the declarations in the implementation make it plug-compatible with the interface. Obviously, inheriting from the interface is, in general, a much better approach. But we have included examples of both simply to demonstrate that it isn’t strictly required to inherit from the interface.

Here are some of the implementations we’ll be using over the remainder of this section. It is important to keep in mind that although these models include many lines of Modelica source code, they can be very quickly implemented in a graphical Modelica environment (*i.e.*, one does not normally type in such models by hand).

Plant Models

```

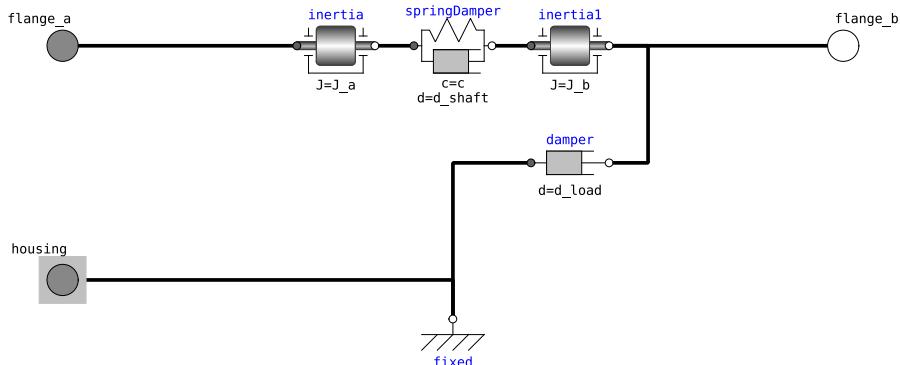
within ModelicaByExample.Architectures.SensorComparison.Implementation;
model BasicPlant "Implementation of the basic plant model"
  parameter Modelica.SIunits.Inertia J_a=0.1 "Moment of inertia";
  parameter Modelica.SIunits.Inertia J_b=0.3 "Moment of inertia";
  parameter Modelica.SIunits.RotationalSpringConstant c=100 "Spring constant";
  parameter Modelica.SIunits.RotationalDampingConstant d_shaft=3
    "Shaft damping constant";
  parameter Modelica.SIunits.RotationalDampingConstant d_load=4
    "Load damping constant";

  Modelica.Mechanics.Rotational.Interfaces.Support_housing
    "Connection to mounting"
    annotation (Placement(transformation(extent={{-110,-70},{-90,-50}})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a
    "Input shaft for plant"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b
  
```

```

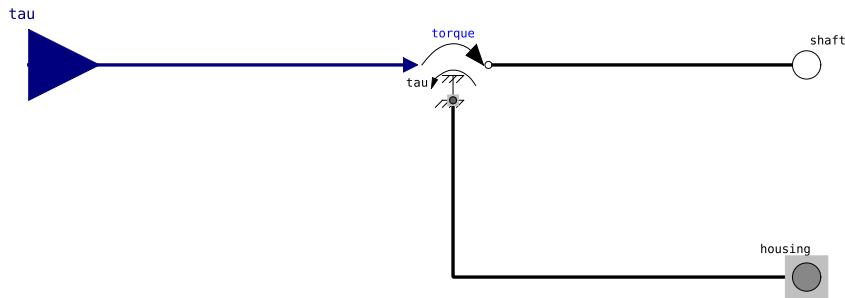
"Output shaft of plant"
annotation (Placement(transformation(extent={{90,-10},{110,10}})));
protected
  Modelica.Mechanics.Rotational.Components.Fixed fixed
    annotation (Placement(transformation(extent={{-10,-80},{10,-60}})));
  Modelica.Mechanics.Rotational.Components.Inertia inertia(J=J_a)
    annotation (Placement(transformation(extent={{-40,-10},{-20,10}})));
  Modelica.Mechanics.Rotational.Components.Inertia inertia1(J=J_b)
    annotation (Placement(transformation(extent={{20,-10},{40,10}})));
  Modelica.Mechanics.Rotational.Components.SpringDamper springDamper(c=c, d=
    d_shaft)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  Modelica.Mechanics.Rotational.Components.Damper damper(d=d_load)
    annotation (Placement(transformation(extent={{20,-40},{40,-20}})));
equation
  connect(springDamper.flange_a, inertia.flange_b) annotation (Line(
    points={{-10,0},{-20,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(springDamper.flange_b, inertia1.flange_a) annotation (Line(
    points={{10,0},{20,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(damper.flange_b, inertia1.flange_b) annotation (Line(
    points={{40,-30},{50,-30},{50,0},{40,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(damper.flange_a, fixed.flange) annotation (Line(
    points={{20,-30},{0,-30},{0,-70}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(inertia1.flange_b, flange_b) annotation (Line(
    points={{40,0},{100,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(inertia.flange_a, flange_a) annotation (Line(
    points={{-40,0},{-100,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(fixed.flange, housing) annotation (Line(
    points={{0,-70},{0,-60},{-100,-60}},
    color={0,0,0},
    smooth=Smooth.None));
end BasicPlant;

```



Actuator Models

```
within ModelicaByExample.Architectures.SensorComparison.Implementation;
model IdealActuator "An implementation of an ideal actuator"
  Modelica.Mechanics.Rotational.Interfaces.Flange_b shaft "Output shaft"
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
  Modelica.Mechanics.Rotational.Interfaces.Support housing
    "Connection to housing"
    annotation (Placement(transformation(extent={{90,-70},{110,-50}})));
  Modelica.Blocks.Interfaces.RealInput tau "Input torque command"
    annotation (Placement(transformation(extent={{-140,-20},{-100,20}})));
protected
  Modelica.Mechanics.Rotational.Sources.Torque torque(useSupport=true)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
equation
  connect(torque.flange, shaft) annotation (Line(
    points={{10,0},{100,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(torque.support, housing) annotation (Line(
    points={{0,-10},{0,-60},{100,-60}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(torque.tau, tau) annotation (Line(
    points={{-12,0}, {-120,0}},
    color={0,0,127},
    smooth=Smooth.None));
end IdealActuator;
```

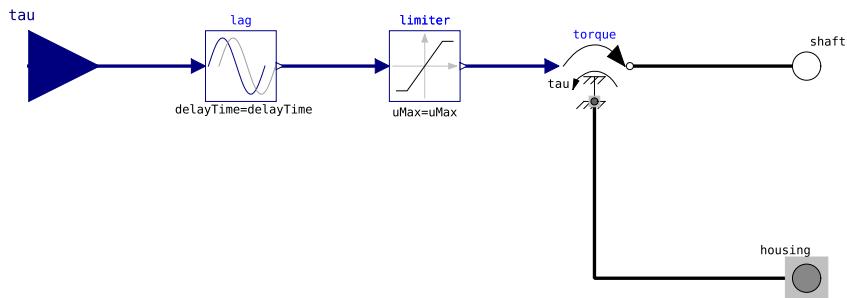


```
within ModelicaByExample.Architectures.SensorComparison.Implementation;
model LimitedActuator "An actuator with lag and saturation"
  extends Interfaces.Actuator;
  parameter Modelica.SIunits.Time delayTime
    "Delay time of output with respect to input signal";
  parameter Real uMax "Upper limits of input signals";
protected
  Modelica.Mechanics.Rotational.Sources.Torque torque(useSupport=true)
    annotation (Placement(transformation(extent={{30,-10},{50,10}})));
  Modelica.Blocks.Nonlinear.Limiter limiter(uMax=uMax)
    annotation (Placement(transformation(extent={{-18,-10},{2,10}})));
  Modelica.Blocks.Nonlinear.FixedDelay lag(delayTime=delayTime)
    annotation (Placement(transformation(extent={{-70,-10},{-50,10}})));
equation
  connect(torque.flange, shaft) annotation (Line(
    points={{50,0},{100,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(torque.support, housing) annotation (Line(
```

```

points={{40,-10},{40,-60},{100,-60}},
color={0,0,0},
smooth=Smooth.None));
connect(limiter.y, torque.tau) annotation (Line(
    points={{3,0},{28,0}},
    color={0,0,127},
    smooth=Smooth.None));
connect(lag.u, tau) annotation (Line(
    points={{-72,0}, {-120,0}},
    color={0,0,127},
    smooth=Smooth.None));
connect(lag.y, limiter.u) annotation (Line(
    points={{-49,0}, {-20,0}},
    color={0,0,127},
    smooth=Smooth.None));
end LimitedActuator;

```



Controller Models

```

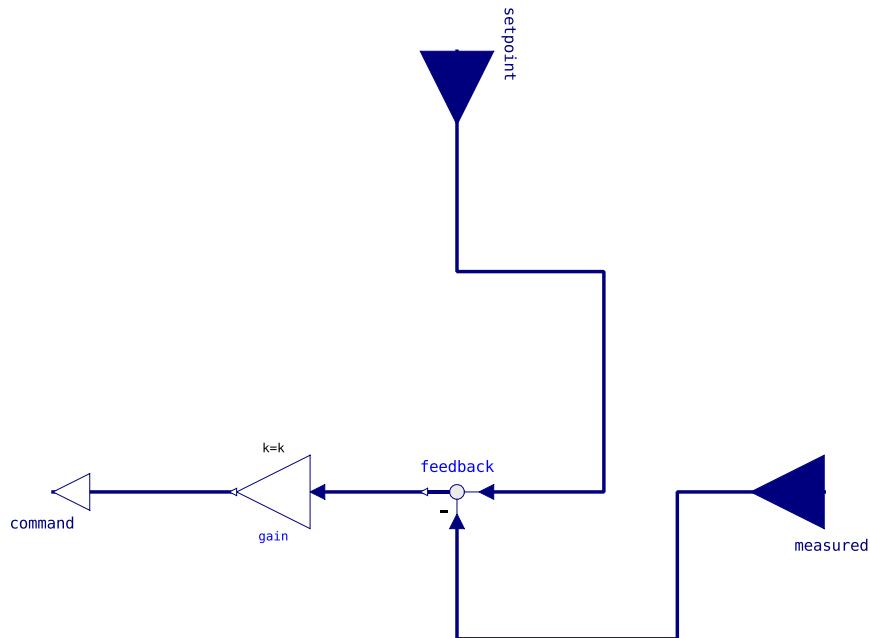
within ModelicaByExample.Architectures.SensorComparison.Implementation;
model ProportionalController "Implementation of a proportional controller"
    parameter Real k=20 "Controller gain";
    Modelica.Blocks.Interfaces.RealInput setpoint "Desired system response"
        annotation (Placement(transformation(
            extent={{-20,-20},{20,20}},
            rotation=270, origin={0,120})));
    Modelica.Blocks.Interfaces.RealInput measured "Actual system response"
        annotation (Placement(transformation(
            extent={{-20,-20},{20,20}},
            rotation=180, origin={100,0})));
    Modelica.Blocks.Interfaces.RealOutput command "Command to send to actuator"
        annotation (Placement(transformation(
            extent={{-10,-10},{10,10}},
            rotation=180, origin={-110,0})));
protected
    Modelica.Blocks.Math.Gain gain(k=k)
        annotation (Placement(transformation(
            extent={{-10,-10},{10,10}},
            rotation=180, origin={-50,0})));
    Modelica.Blocks.Math.Feedback feedback
        annotation (Placement(transformation(
            extent={{10,-10},{-10,10}})));
equation
    connect(feedback.y, gain.u) annotation (Line(
        points={{-9,0},{2,0},{2,0},{-38,0}},
        color={0,0,127}, smooth=Smooth.None));
    connect(feedback.u1, setpoint) annotation (Line(

```

```

points={{8,0},{40,0},{40,60},{0,60},{0,120}},
color={0,0,127},
smooth=Smooth.None));
connect(gain.y, command) annotation (Line(
points={{-61,0}, {-80.5,0}, {-80.5,0}, {-110,0}},
color={0,0,127},
smooth=Smooth.None));
connect(measured, feedback.u2) annotation (Line(
points={{100,0},{60,0},{60,-40},{0,-40},{0,-8}},
color={0,0,127}, smooth=Smooth.None));
end ProportionalController;

```

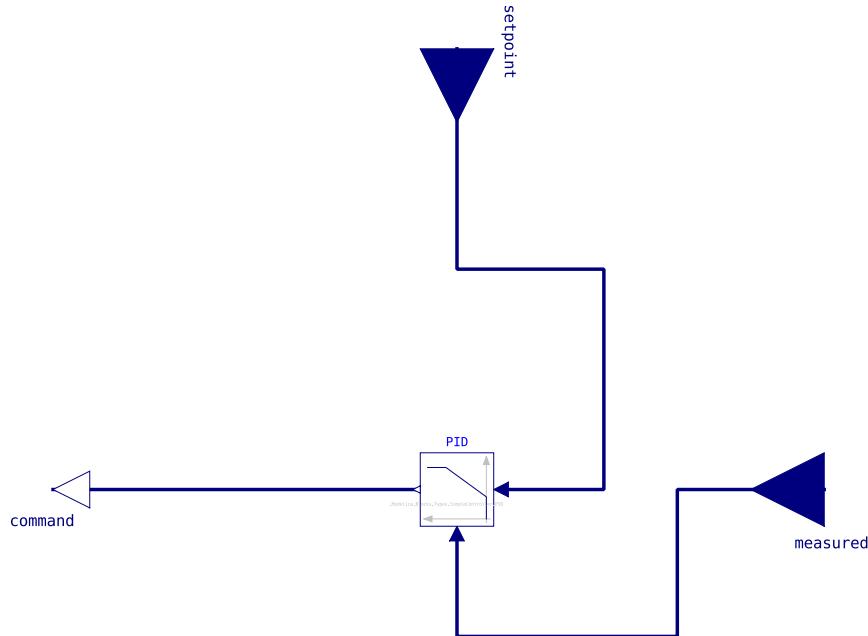


```

within ModelicaByExample.Architectures.SensorComparison.Implementation;
model PID_Controller "Controller subsystem implemented using a PID controller"
  extends Interfaces.Controller;
  parameter Real k "Gain of controller";
  parameter Modelica.SIunits.Time Ti "Time constant of Integrator block";
  parameter Modelica.SIunits.Time Td "Time constant of Derivative block";
  parameter Real yMax "Upper limit of output";
protected
  Modelica.Blocks.Continuous.LimPID PID(k=k, Ti=Ti, Td=Td, yMax=yMax)
    annotation (Placement(transformation(
      extent={{10,-10},{-10,10}})));
equation
  connect(setpoint, PID.u_s) annotation (Line(
    points={{0,120},{0,60},{40,60},{40,0},{12,0}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(measured, PID.u_m) annotation (Line(
    points={{100,0},{60,0},{60,-40},{0,-40},{0,-12}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(PID.y, command) annotation (Line(
    points={{-11,0}, {-110,0}},
    color={0,0,127}),

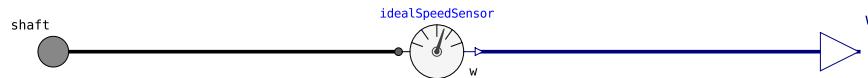
```

```
smooth=Smooth.None));
end PID_Controller;
```



Sensor Models

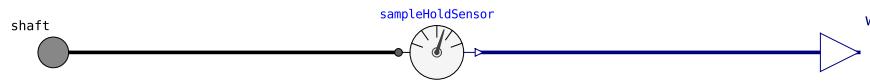
```
within ModelicaByExample.Architectures.SensorComparison.Implementation;
model IdealSensor "Implementation of an ideal sensor"
  Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
    "Flange of shaft from which sensor information shall be measured"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
protected
  Modelica.Mechanics.Rotational.Sensors.SpeedSensor idealSpeedSensor
    "An ideal speed sensor" annotation (Placement(transformation(
      extent={{-10,-10},{10,10}}));
equation
  connect(idealSpeedSensor.flange, shaft) annotation (Line(
    points={{-10,0}, {-100,0}},
    color={0,0,0},
    smooth=Smooth.None));
  connect(idealSpeedSensor.w, w) annotation (Line(
    points={{11,0}, {110,0}},
    color={0,0,127},
    smooth=Smooth.None));
end IdealSensor;
```



```

within ModelicaByExample.Architectures.SensorComparison.Implementation;
model SampleHoldSensor "Implementation of a sample hold sensor"
  parameter Modelica.SIunits.Time sample_time(min=Modelica.Constants.eps);
  Modelica.Mechanics.Rotational.Interfaces.Flange_a shaft
    "Flange of shaft from which sensor information shall be measured"
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  Modelica.Blocks.Interfaces.RealOutput w "Absolute angular velocity of flange"
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
protected
  Components.SpeedMeasurement.Components.SampleHold sampleHoldSensor(
    sample_time=sample_time)
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
equation
  connect(sampleHoldSensor.w, w) annotation (Line(
    points={{11,0},{110,0}},
    color={0,0,127},
    smooth=Smooth.None));
  connect(sampleHoldSensor.flange, shaft) annotation (Line(
    points={{-10,0}, {-100,0}},
    color={0,0,0},
    smooth=Smooth.None));
end SampleHoldSensor;

```



Variations

Baseline Configuration

With these implementations at hand, we can create a number of different implementations of our complete system. For example, to implement the behavior of our original `FlatSystem` model, we simply extend from the `SystemArchitecture` model and redeclare each of the subsystems so that the implementations match the subsystem implementations in `FlatSystem`, *i.e.*,

```

within ModelicaByExample.Architectures.SensorComparison.Examples;
model BaseSystem "System architecture with base implementations"
  extends SystemArchitecture(
    redeclare replaceable Implementation.ProportionalController controller,
    redeclare replaceable Implementation.IdealActuator actuator,
    redeclare replaceable Implementation.BasicPlant plant,
    redeclare replaceable Implementation.IdealSensor sensor);
end BaseSystem;

```

Here we see the real power of Modelica for specifying configurations. Note how each redeclaration includes the `replaceable` qualifier. By doing so, we ensure that subsequent redeclarations are possible.

If we had wanted our `SystemArchitecture` model to specify these implementations as the default but still use the interfaces as the constraining types, we could have declared the subsystems in `SystemArchitecture` as follows:

```

replaceable Implementation.BasicPlant plant constrainedby Interfaces.Plant
  annotation (choicesAllMatching=true,
  Placement(transformation(extent={{-10,-50},{10,-30}})));
replaceable Implementation.IdealActuator actuator constrainedby
  Interfaces.Actuator
  annotation (choicesAllMatching=true,
  Placement(transformation(extent={{-50,-50},{-30,-30}})));

```

```

replaceable Implementation.IdealSensor sensor constrainedby Interfaces.Sensor
annotation (choicesAllMatching=true,
Placement(transformation(extent={{30,-50},{50,-30}})));
replaceable Implementation.ProportionalController controller constrainedby
Interfaces.Controller
annotation (choicesAllMatching=true,
Placement(transformation(extent={{-10,-10},{10,10}})));
replaceable Modelica.Blocks.Sources.Trapezoid setpoint(period=1.0) constrainedby
Modelica.Blocks.Interfaces.SO
annotation (choicesAllMatching=true,
Placement(transformation(extent={{-60,30},{-40,50}})));

```

Variation1

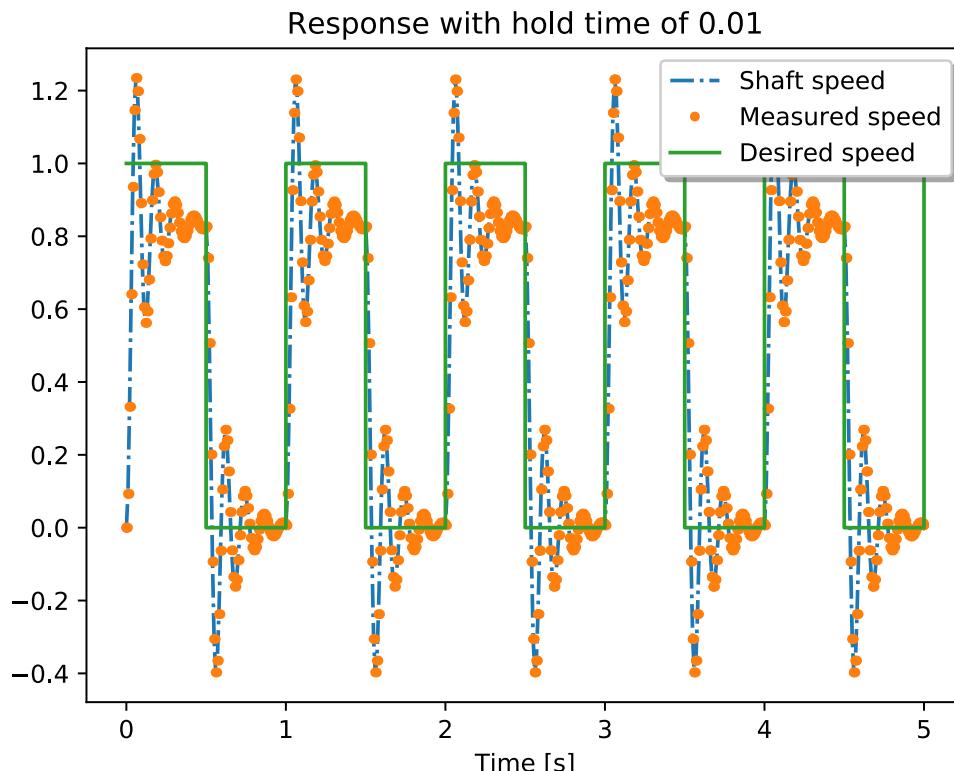
If we wish to create a variation of the `BaseSystem` model, we can use inheritance and modifiers to create them, e.g.,

```

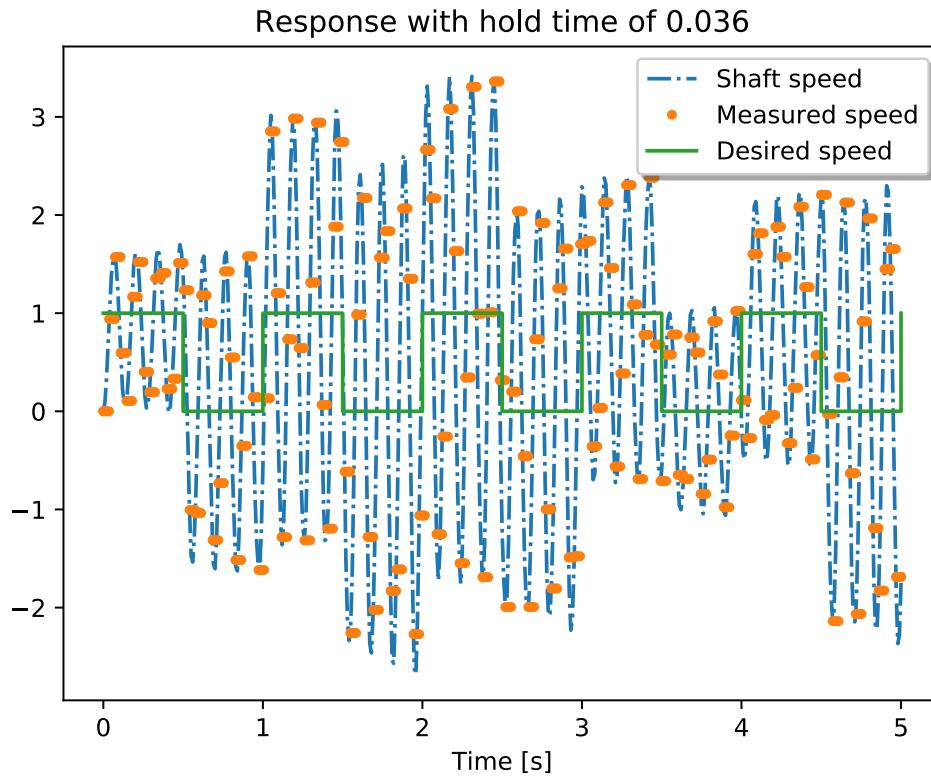
within ModelicaByExample.Architectures.SensorComparison.Examples;
model Variant1 "Creating sample-hold variant using system architecture"
  extends BaseSystem(redeclare replaceable
    Implementation.SampleHoldSensor sensor(sample_time=0.01));
end Variant1;

```

Note how this model extends from `BaseSystem` configuration but then changes **only** the `sensor` model. If we simulate this system, we see that the performance matches that of our original *Flat System* (page 304).



However, if we instead create a configuration with the longer hold time, we find that the system becomes unstable (exactly as it did in the *Flat System* (page 304) as well):

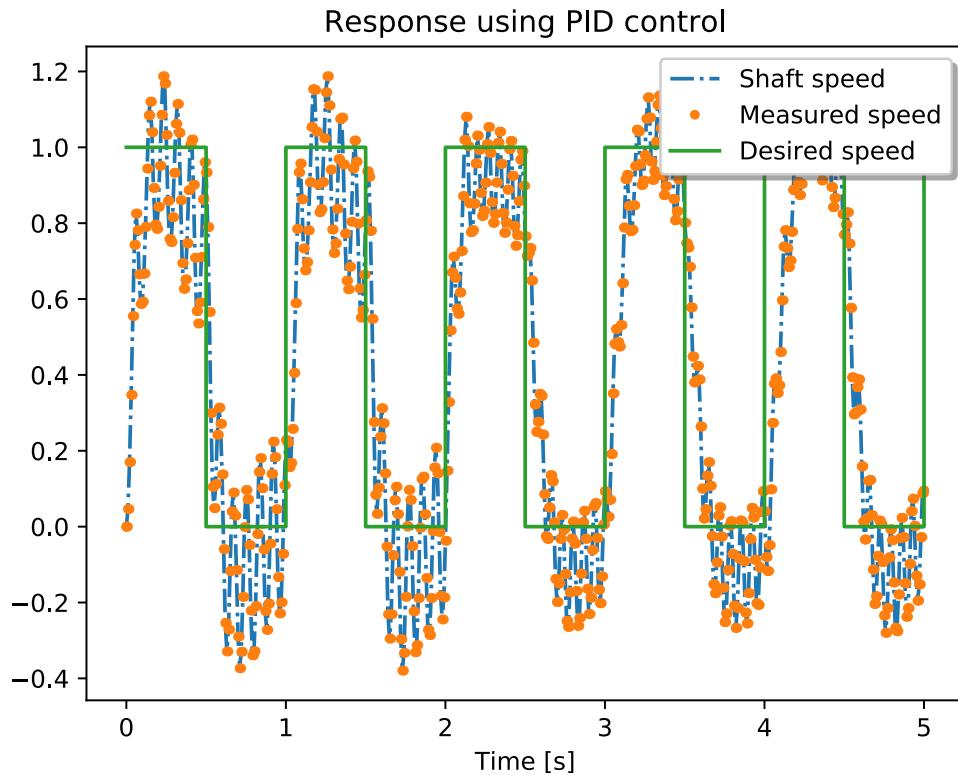


Variation2

Note how, even with an adequate sensor, the controller in our `Variant1` configuration seems to be converging to the wrong steady state speed. This is because we are only using a proportional gain controller. However, if we extend from the `Variant1` model and add a PID controller and a more realistic actuator with limitations on the amount of torque it can supply, *i.e.*,

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
model Variant2 "Adds PID control and realistic actuator subsystems"
  extends Variant1(
    redeclare replaceable Implementation.PID_Controller controller(
      yMax=15, Td=0.1, k=20, Ti=0.1),
    redeclare replaceable Implementation.LimitedActuator actuator(
      delayTime=0.005, uMax=10));
end Variant2;
```

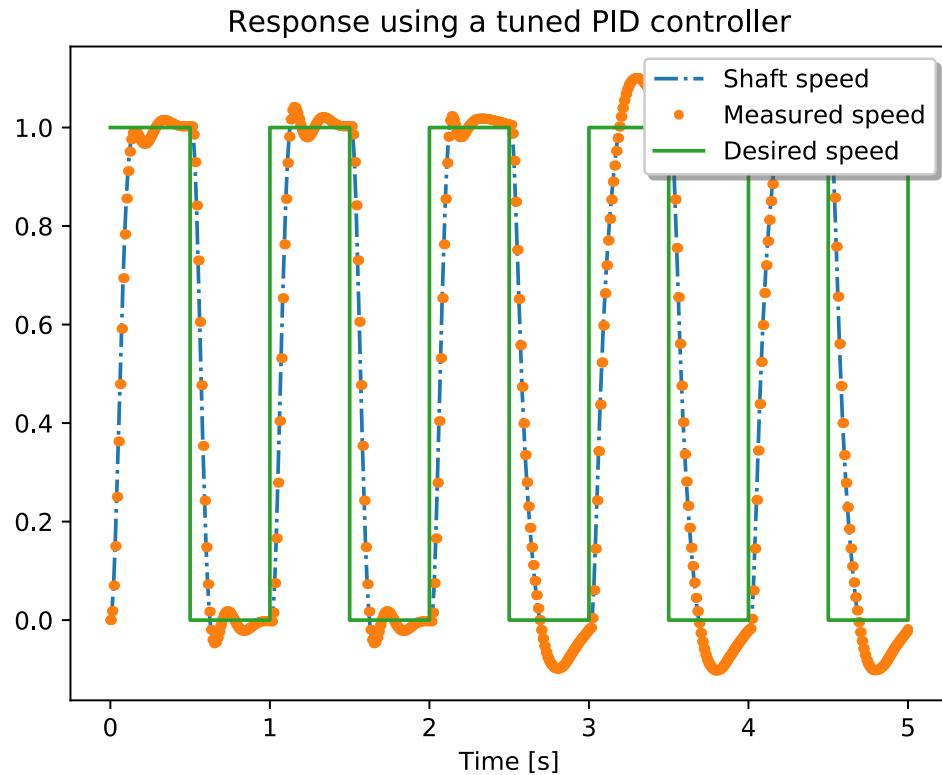
we will get the following simulation results:



Furthermore, if we take some time to tune the gains in the PID controller, *i.e.*,

```
within ModelicaByExample.Architectures.SensorComparison.Examples;
model Variant2_tuned "A tuned up version of variant 2"
  extends Variant2(
    controller(yMax=50, Ti=0.07, Td=0.01, k=4),
    actuator(uMax=50),
    sensor(sample_time=0.01));
end Variant2_tuned;
```

Then, we will get even better simulation results:



Conclusion

This concludes our discussion of this particular architecture. The key point in this example is that by using architectures, we make it very easy to explore alternative configurations for our system. But in addition to being easier (in the sense that we are able to do these things very quickly), we are also able to ensure a degree of correctness as well because no additional connecting is required for each configuration. Instead, the user simply needs to specify which implementation they would like to use for each subsystem and, even then, the Modelica compiler can check to make sure that their choices are plug-compatible with the constraining types used to specify the architecture.

Thermal Control

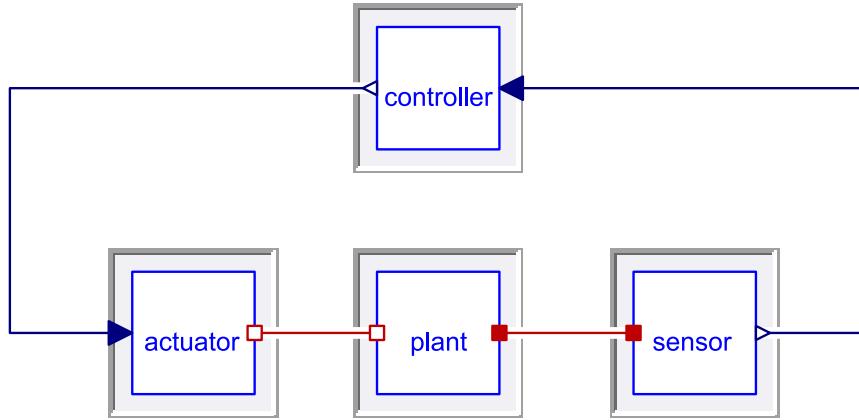
In this chapter, we'll consider another system that includes a plant, controller, sensor and actuator. The application will be thermal control of a three zone house. The plant will be the house itself, the sensor will be a temperature sensor and the actuator will be the furnace in the house. Using these models, we will explore a few different control strategies.

We'll also follow the architecture driven approach to building the system that we followed in the previous section. However, we'll start using one set of interfaces and then, after discussing their limitations, restart taking a different approach that will provide us with greater flexibility.

Initial Approach

Architecture

Let's start with the following architecture:



Here we see the same basic pieces we saw in the previous section, *i.e.*, a plant model, a sensor, a controller and an actuator. In fact, this is a pretty typical architecture. In some cases, people may break down the plant model into a few subsystems and/or include multiple controllers and control loops. But many closed loop system control problems follow this same basic structure.

What tends to change from application to application are the specific signals exchanged between these parts. In this case, we can see from the architecture schematic that our interface definitions are defined such that:

- The actuator receives a commanded temperature and then injects heat through a thermal connection to the plant
- The sensor model also has a thermal connector (to the plant) and an output signal containing the measured temperature.
- The plant has two thermal connections. One represents where the furnace heat is added to the system and the other is where the sensor is located.
- The controller takes the measured temperature (from the sensor) as an input and outputs a commanded heat output (to the actuator)

The Modelica code for this base system looks like this:

```

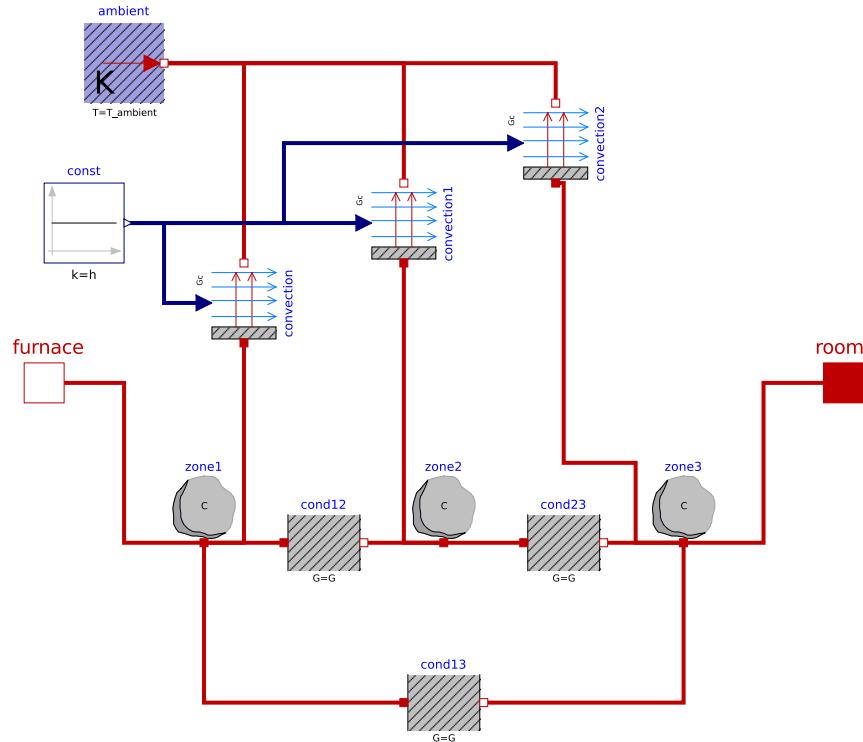
within ModelicaByExample.Architectures.ThermalControl.Architectures;
partial model BaseArchitecture "A basic thermal architecture"
  replaceable Interfaces.PlantModel plant
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  replaceable Interfaces.ControlSystem controller
    annotation (Placement(transformation(extent={{-10,30},{10,50}})));
  replaceable Interfaces.Sensor sensor
    annotation (Placement(transformation(extent={{32,-10},{52,10}})));
  replaceable Interfaces.Actuator actuator
    annotation (Placement(transformation(extent={{-50,-10},{-30,10}})));
equation
  connect(plant.room, sensor.room) annotation (Line(
    points={{10,0},{32,0}},
    color={191,0,0}, smooth=Smooth.None));
  connect(sensor.temperature, controller.temperature) annotation (Line(
    points={{53,0},{70,0},{70,40},{12,40}},
    color={0,0,127}, smooth=Smooth.None));
  connect(actuator.furnace, plant.furnace) annotation (Line(
    points={{-30,0},{-10,0}},
    color={191,0,0}, smooth=Smooth.None));
  connect(controller.heat, actuator.heat) annotation (Line(
    points={{-11,40},{-70,40},{-70,0},{-52,0}});
  
```

```
color={0,0,127}, smooth=Smooth.None));
end BaseArchitecture;
```

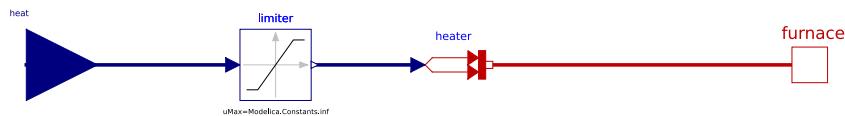
Initial Implementations

Plant

Our plant model looks like this:

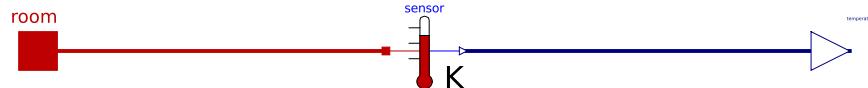


Here we can see that the furnace heat is added in one zone while the temperature is measured in a different zone. Furthermore, there is an additional zone between the actuator and sensor zones. The furnace model itself is a simple heat source:



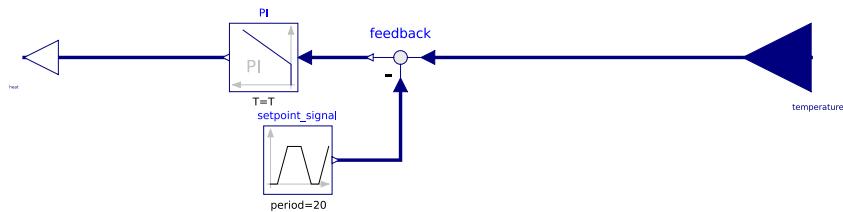
This actuator takes a commanded heat level as an input and then injects that amount of heat into the zone it is connected to.

The sensor is similarly simple:



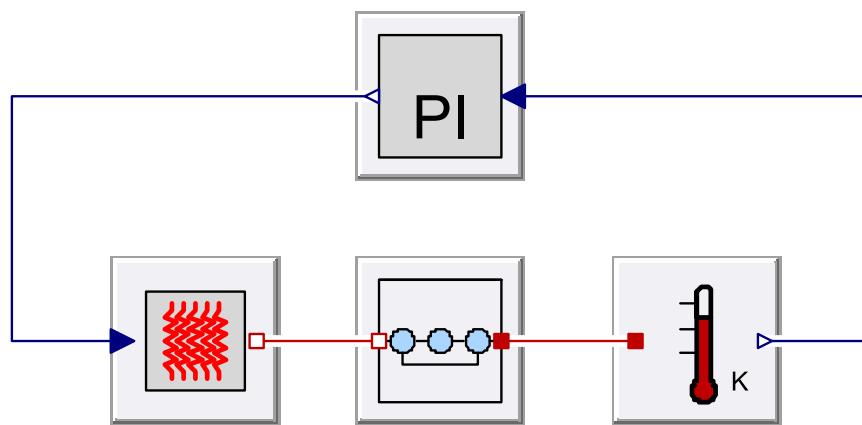
This sensor doesn't introduce any artifact. Instead, it provides the exact temperature in the zone as a continuous output signal.

We will use the following PI controller to control the temperature:



Initial Results

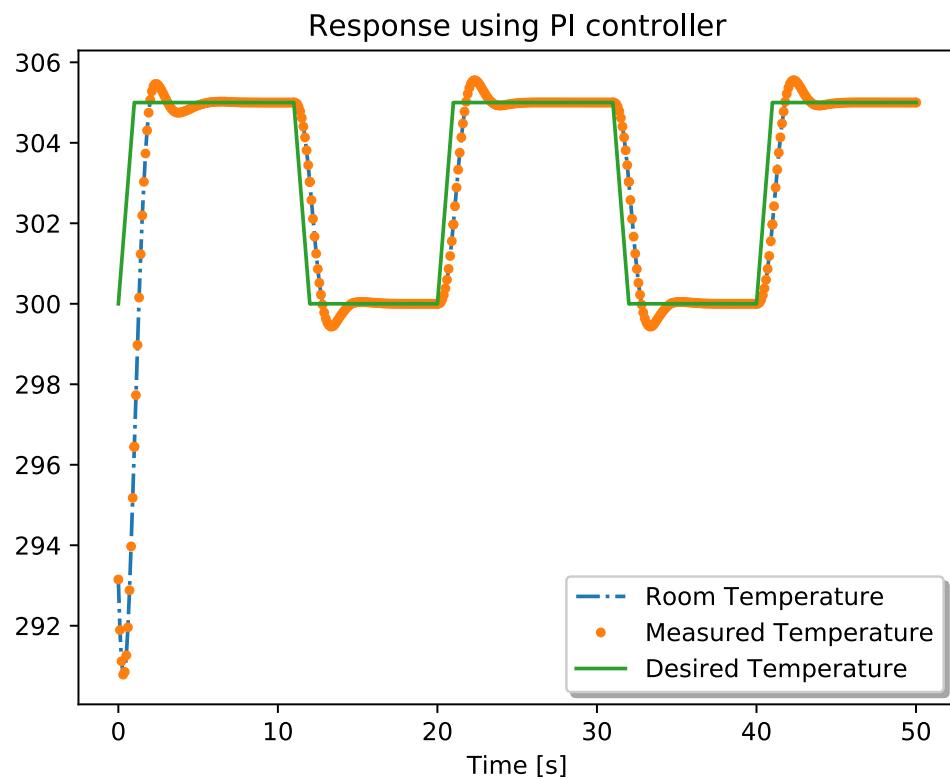
Populating our architecture with these implementations, our model now looks like this:



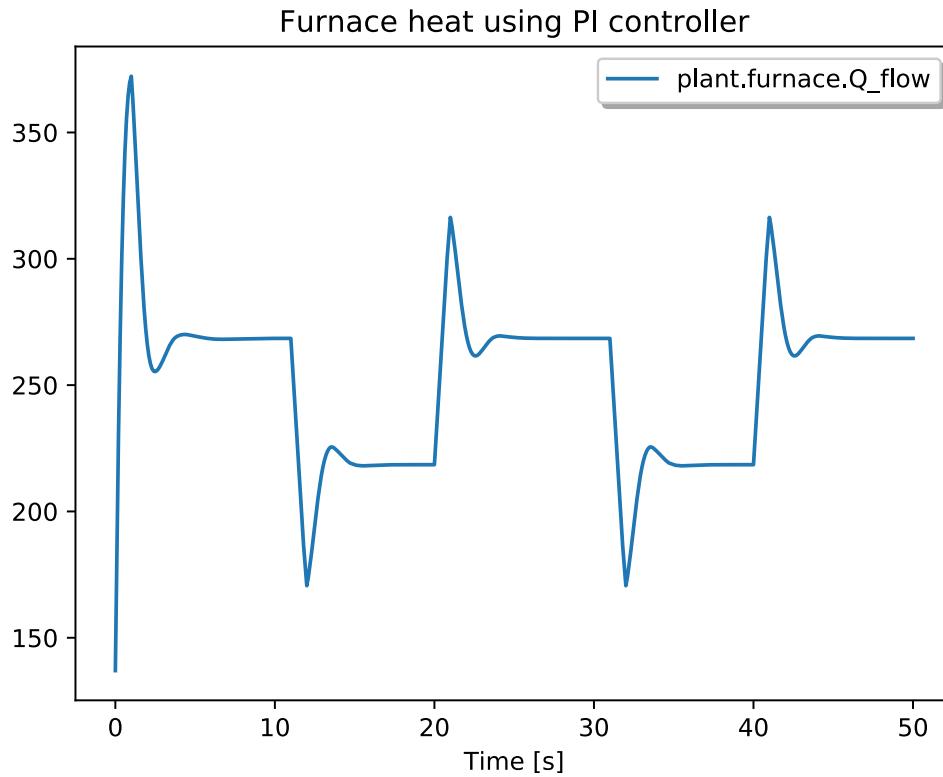
Note how the icons for the various subsystems have changed. This is because when we perform a `redeclare`, the icon for the new type associated with that subsystem is used. The Modelica code for this system is:

```
within ModelicaByExample.Architectures.ThermalControl.Examples;
model BaseModel "Base model using a conventional architecture"
  extends Architectures.BaseArchitecture(
    redeclare Implementations.ThreeZonePlantModel plant(
      C=2, G=1, h=2, T_ambient=278.15),
    redeclare
      ModelicaByExample.Architectures.ThermalControl.Implementations.ConventionalPIControl
        controller(setpoint=300, T=1, k=20),
      redeclare Implementations.ConventionalActuator actuator,
      redeclare Implementations.ConventionalSensor sensor);
end BaseModel;
```

If we simulate this system, we get the following results:



As we can see, this approach works very well. The furnace heat required to achieve this degree of control is:

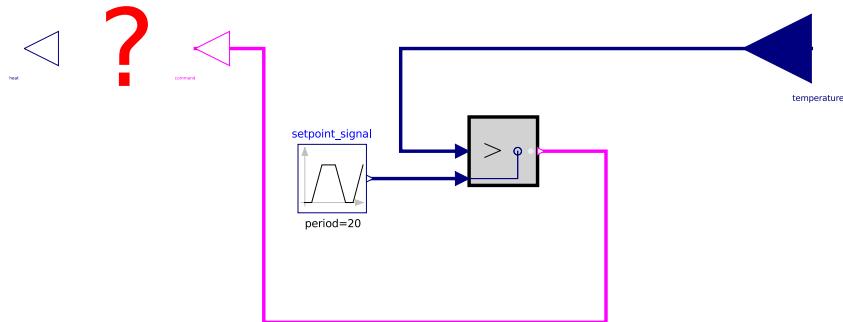


Bang Bang Control

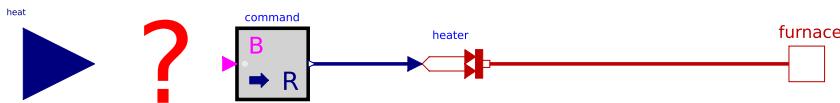
So far, this approach seems like it has been quite successful. We have a nice architecture that we can use to consider different actuators, sensors, controllers or even plant models. The control system we've developed seems to do a fairly good job of controlling our plant.

But one thing worth noting is that the furnace heat required in this case is continuous. However, home heating systems do not typically use this type of control strategy. Instead, they tend to use something called “bang-bang” control where the furnace is either “off” or “on”.

We have this flexible architecture, so perhaps to address this situation, we should create an implementation of our controller and actuator models where the controller command is a boolean indicating whether the furnace should be on or off. However, if we start this process, we quickly run into the following problem:



Note that the output from our controller is Boolean value but the commanded heat signal from our ControlSystem interface requires a Real value. We have the same problem on the actuator side:



The interface supplies an actuator that is a Real value but again we see that if our furnace expects an “on” or “off” command, we have a mismatch.

So the question then becomes, **how can we handle situations where choosing different subsystems requires us to have different interfaces?**

Expandable Approach

The solution to this problem is **expandable** connector definitions. With this approach, our subsystem interface would be the same regardless of whether the control strategy generates a Boolean or Real. What changes is the contents of the **connector** instances.

To understand how these **expandable** connectors work, we’ll reformulate our architecture to include **expandable** connectors and then see how these can be used for both continuous and “bang-bang” approaches.

Expandable Connectors

The key feature that allows us to make more flexible architectures is the **expandable connector**. For example, previously we defined the **Actuator** interface as:

```

within ModelicaByExample.Architectures.ThermalControl.Interfaces;
partial model Actuator "Actuator subsystem interface"

  Modelica.Blocks.Interfaces.RealInput heat "Heating command" annotation (
    Placement(transformation(
      extent={{-20,-20},{20,20}},
      origin={-120,0})));
  
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_b furnace
    "Connection point for the furnace"
    annotation (Placement(transformation(extent={{90,-10},{110,10}})));
end Actuator;
  
```

This interface contains two connectors, the **heat** connector and the **furnace** connector. The **furnace** connector is a thermal connector that allows the furnace to interact thermally with the plant. The **heat** connector is a Real valued input signal that comes from the controller and specifies the desired heat output level. The fact that this is a Real valued signal was the problem when we wanted to switch to a type of control that required a Boolean signal. To address this, we’ll use the following interface definition for our actuators:

```

within ModelicaByExample.Architectures.ThermalControl.Interfaces;
partial model Actuator_WithExpandableBus
  "Actuator subsystem interface with an expandable bus"

  ExpandableBus bus
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
  
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_b furnace
  
```

```

"Connection point for the furnace"
annotation (Placement(transformation(extent={{90,-10},{110,10}})));
end Actuator_WithExpandableBus;

```

Here we see the furnace connector is still present. But the heat connector is gone. Instead, it has been replaced by a new connector instance, bus, of type `ExpandableBus`. The connector definition for `ExpandableBus` is:

```

within ModelicaByExample.Architectures.ThermalControl.Interfaces;
expandable connector ExpandableBus "An example of an expandable bus connector"
end ExpandableBus;

```

In other words, **it is empty**. But what is significant is the presence of the `expandable` qualifier. If a given bus is required to always have certain signals, they should be listed within the connector definition. But the fact that there are no variables or sub-connectors listed in the `ExpandableBus` class means there is no minimum requirement for information to be carried on the bus. But the bus can be **expanded** to include additional information.

Note that there is no formal definition of a “bus” in Modelica. The term is often used in such contexts to connote a connector that is carrying multiple pieces of information.

Expandable connectors work in a special way. The signals on an expandable bus **are determined by the connections themselves**. By connecting something to the expandable bus, a signal is implicitly added to that connector. Then the Modelica compiler looks at all the connectors in a connection set and expands each one so that they match. We'll go into more details about this process once we get to the point where we have some implementation models to discuss.

The interface for the plant model is unaffected by the use of `expandable` connectors (since none of these expandable connectors are associated with the plant model), but the interfaces for the sensor and controller are changed as follows:

```

within ModelicaByExample.Architectures.ThermalControl.Interfaces;
partial model Sensor_WithExpandableBus
  "Sensor subsystem interface using an expandable bus"

  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a room
  "Thermal connection to room"
  annotation (Placement(transformation(extent={{-110,-10},{-90,10}}));

  ModelicaByExample.Architectures.ThermalControl.Interfaces.ExpandableBus bus
  annotation (Placement(transformation(extent={{90,-10},{110,10}}));
end Sensor_WithExpandableBus;

```

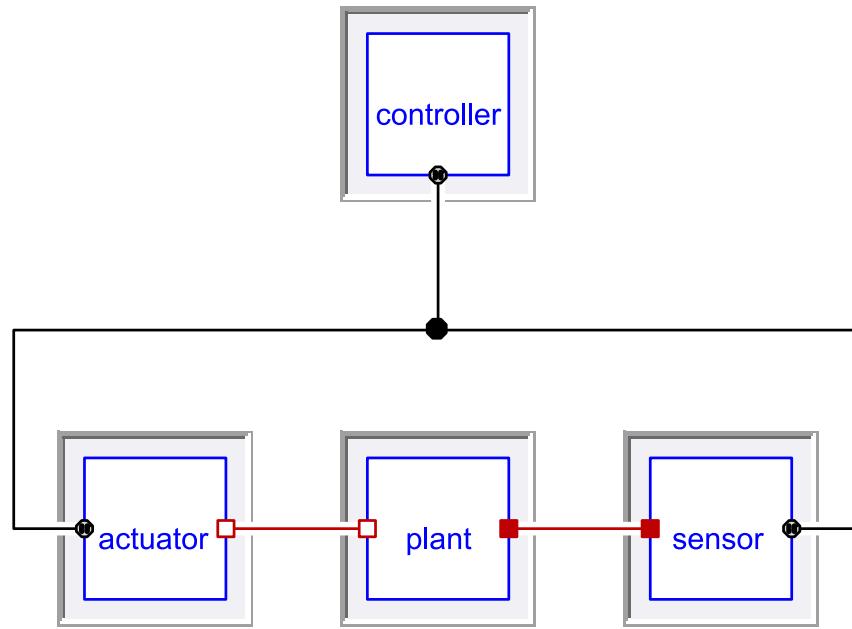
```

within ModelicaByExample.Architectures.ThermalControl.Interfaces;
partial model ControlSystem_WithExpandableBus
  "Control system interface using an expandable bus connector"
  ExpandableBus bus annotation (Placement(transformation(extent={{-10,-110},{10,-90}})),
    iconTransformation(extent={{-10,-110},{10,-90}}));
end ControlSystem_WithExpandableBus;

```

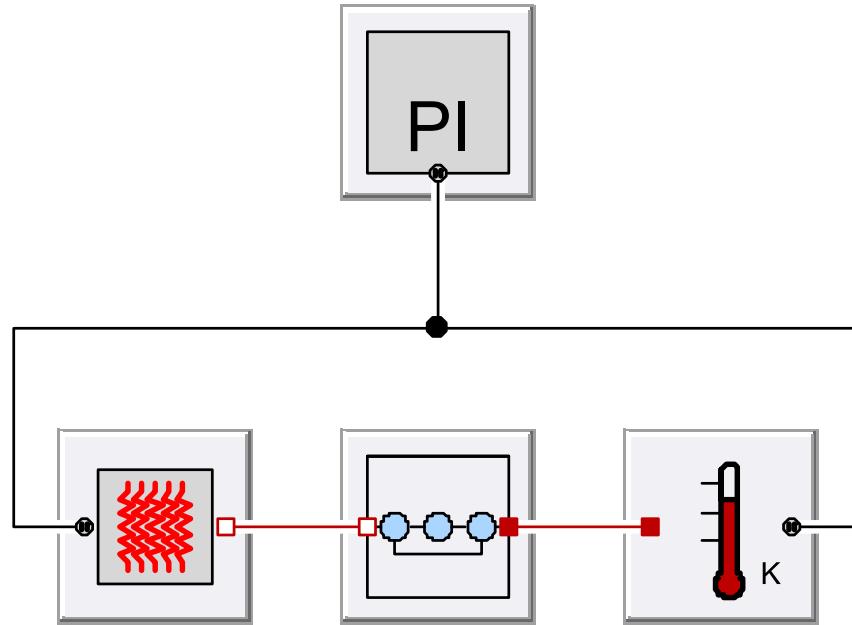
Note how simple the controller interface has become. This is because with an `expandable` connector, we can put the temperature measurement received from the sensor and the heat command sent to the actuator **on the same bus**. As such, we only need one connector. A developer may still choose to use multiple buses simply to organize signals, to make them more representative of the physical partitioning or to avoid confusion. Here we will use a single connector simply to demonstrate that this is now possible.

Using expandable connectors, we can create the following revised architecture:

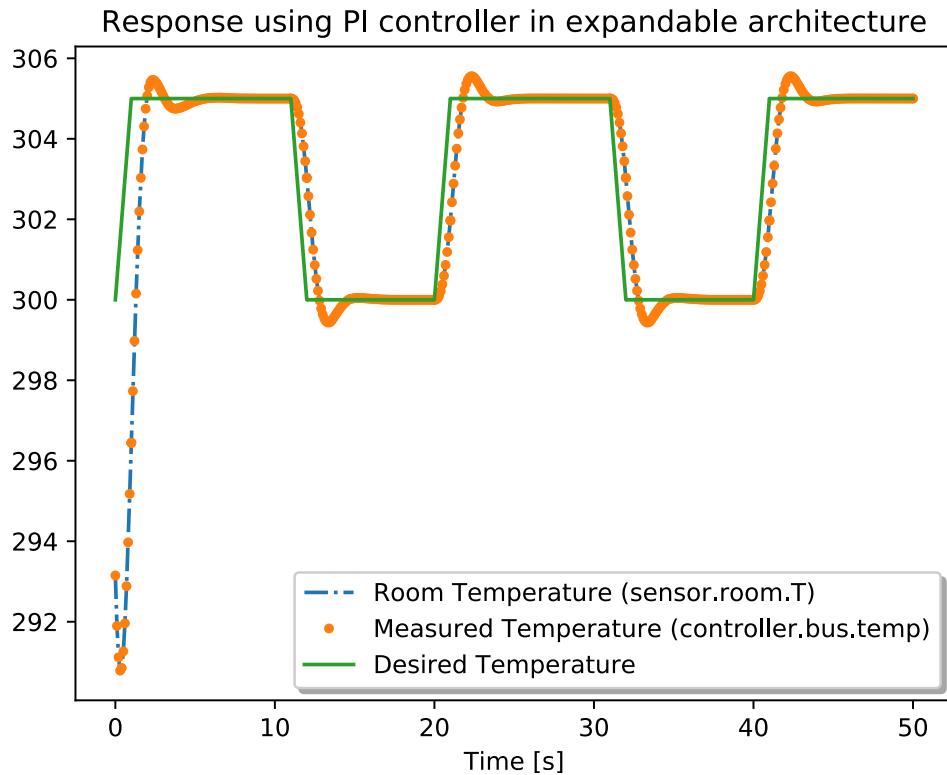


Expandable Implementations

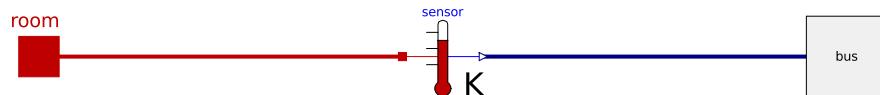
With this more flexible architecture, let's first recreate our original configuration with the continuous control system:



If we plot the results from this system, we get the following response:



Note that the measured temperature corresponds to the signal `controller.bus.temp` where `bus` is an instance of the expandable connector. Further recall that the `ExpandableBus` definition **didn't contain a signal called temperature**. So the question is, how did it get on the connector? The answer lies in the implementation of the sensor model. The diagram for the sensor model looks like this:



The corresponding Modelica code is:

```
within ModelicaByExample.Architectures.ThermalControl.Implementations;
model TemperatureSensor "Temperature sensor using an expandable bus"
  extends Interfaces.Sensor_WithExpandableBus;
protected
  Modelica.Thermal.HeatTransfer.Sensors.TemperatureSensor sensor
  annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
equation
  connect(sensor.T, bus.temperature) annotation (Line(
    points={{10,0},{100,0}}, 
    color={0,0,127}, 
    smooth=Smooth.None));
  connect(room, sensor.port) annotation (Line(
    points={{-100,0}, {-10,0}}, 
    color={191,0,0}, 
    smooth=Smooth.None));
end TemperatureSensor;
```

Of particular importance is the highlighted line.

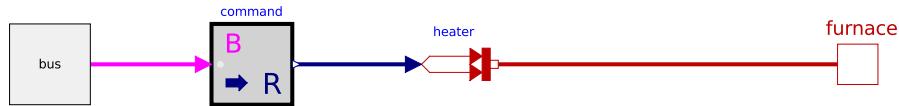
In the diagram, we can see that the output signal from the temperature sensor component is connected to the bus. But when we look at the `connect` statement, it is more than just connected to the bus. It is connected to something named `temperature` inside the bus. This `temperature` connector doesn't exist in the definition of `ExpandableBus`. Instead, **it is created by the connect statement itself!** This is precisely what the `expandable` qualifier allows.

In general, we don't want all connectors to be `expandable`. In cases where we know *a priori* the names and types of all signals, it is better to list them explicitly. This allows the Modelica compiler to make several important checks to ensure the correctness of the model. It is worth noting that by adding the `expandable` qualifier on a connector, the risk of accidentally creating an unintended signal (*e.g.*, as a result of a typing error) becomes a possibility that would otherwise be caught by the compiler.

Reconfiguration

Now that we've shown how we can use the expandable approach to model the continuous control version of our system, let's turn our attention to the "bang-bang" version.

We've already seen the temperature sensor subsystem configured to work with the `expandable` connector. What remains is the controller and actuator models. The actuator model diagram looks like this:



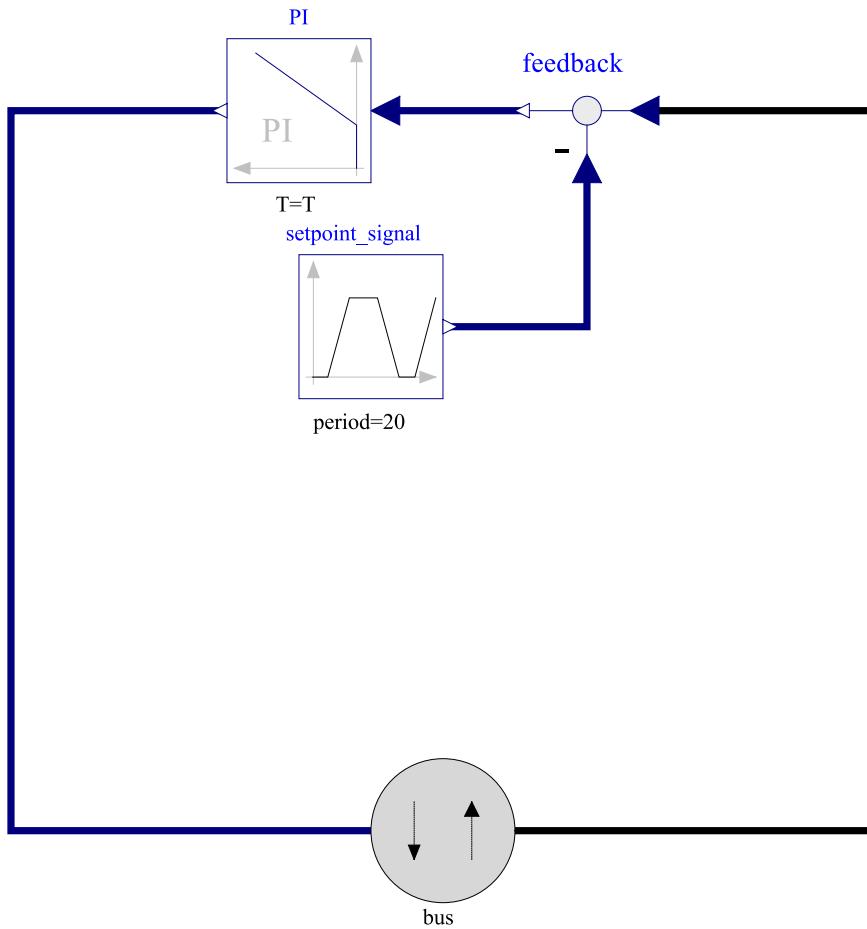
Again, looking at the Modelica code is important to see how the signals on the `bus` connector are referenced:

```

within ModelicaByExample.Architectures.ThermalControl.Implementations;
model OnOffActuator "On-off actuator implemented with an expandable bus"
  extends Interfaces.Actuator_WithExpandableBus;
  parameter Real heating_capacity "Heating capacity of actuator";
protected
  Modelica.Thermal.HeatTransfer.Sources.PrescribedHeatFlow heater
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
  Modelica.Blocks.Math.BooleanToReal command(realTrue=heating_capacity,
    realFalse=0)
    annotation (Placement(transformation(extent={{-60,-10},{-40,10}})));
equation
  connect(heater.port, furnace) annotation (Line(
    points={{10,0},{100,0}}, color={191,0,0},
    smooth=Smooth.None));
  connect(command.y, heater.Q_flow) annotation (Line(
    points={{-39,0}, {-10,0}}, color={0,0,127},
    smooth=Smooth.None));
  connect(command.u, bus.heat_command) annotation (Line(
    points={{-62,0}, {-100,0}}, color={255,0,255},
    smooth=Smooth.None));
end OnOffActuator;
  
```

Again, note the emphasized line. It references something called `heat_command` on the `bus` connector. Again, that signal doesn't exist in the definition of `ExpandableBus`, but it is implicitly created simply because it is referenced in the highlighted `connect` statement.

From the sensor model, we see that the measured temperature is added to the `bus` connector as a `Real` signal named `temperature`. From the actuator model, we see that the command expected by the actuator from the controller is a `Boolean` signal named `heat_command`. As such, we should expect to see both of these signals used by the controller model. The diagram for the controller looks like this:



But the diagram doesn't include sufficient detail to know the precise names of the signals being referenced on the `bus` connector. For that, we need to look at the actual source code:

```

within ModelicaByExample.Architectures.ThermalControl.Implementations;
model ExpandablePIControl "PI controller implemented with an expandable bus"
  extends Interfaces.ControlSystem_WithExpandableBus;
  parameter Real setpoint "Desired temperature";
  parameter Real k=1 "Gain";
  parameter Modelica.SIunits.Time T "Time Constant (T>0 required)";
protected
  Modelica.Blocks.Sources.Trapezoid setpoint_signal(
    amplitude=5, final offset=setpoint, rising=1,
    width=10, falling=1, period=20)
    annotation (Placement(transformation(extent={{-20,-40},{0,-20}})));
  Modelica.Blocks.Math.Feedback feedback
    annotation (Placement(transformation(extent={{30,-10},{10,10}}));
  Modelica.Blocks.Continuous.PI PI(final T=T, final k=-k)
    annotation (Placement(transformation(extent={{-10,-10},{-30,10}})));
equation
  connect(setpoint_signal.y, feedback.u2)
    annotation (Line(
      points={{1,-30},{20,-30},{20,-8}},
      color={0,0,127}, smooth=Smooth.None));
  connect(PI.u,feedback.y) annotation (Line(
    points={{-8,0},{11,0}},
    color={0,0,127}, smooth=Smooth.None));
  connect(bus.temperature, feedback.u1) annotation (Line(

```

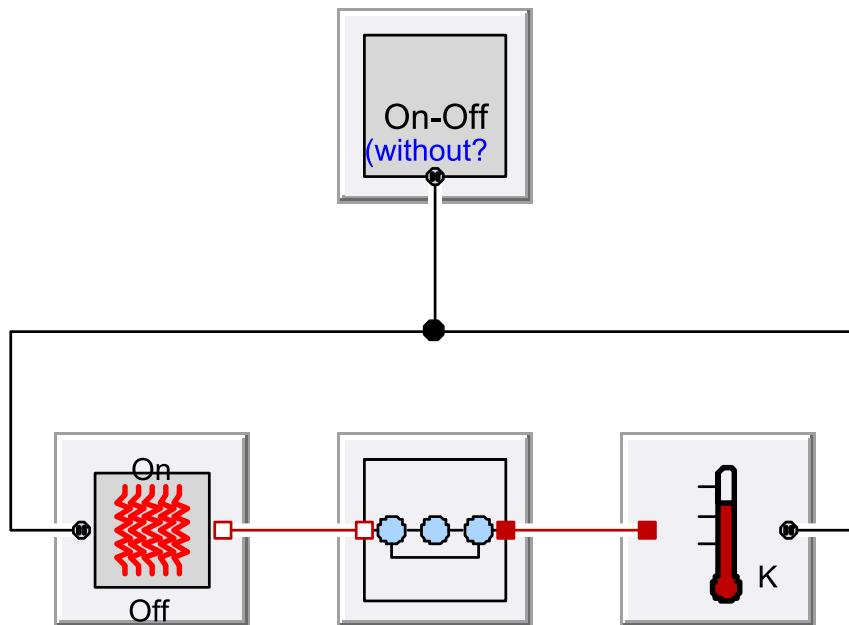
```

points={{0,-100},{60,-100},{60,0},{28,0}},
color={0,0,0}, smooth=Smooth.None));
connect(PI.y, bus.heat) annotation (Line(
points={{-31,0}, {-60,0}, {-60,-100}, {0,-100}},
color={0,0,127}, smooth=Smooth.None));
end ExpandablePIControl;

```

Again, note the highlighted lines. Not only do these `connect` statements implicitly add the `heat` and temperature signals to the bus connector, **those names match** the names that the sensor and actuator models expect.

Pulling all of these subsystems together, we get the following diagram for our system:

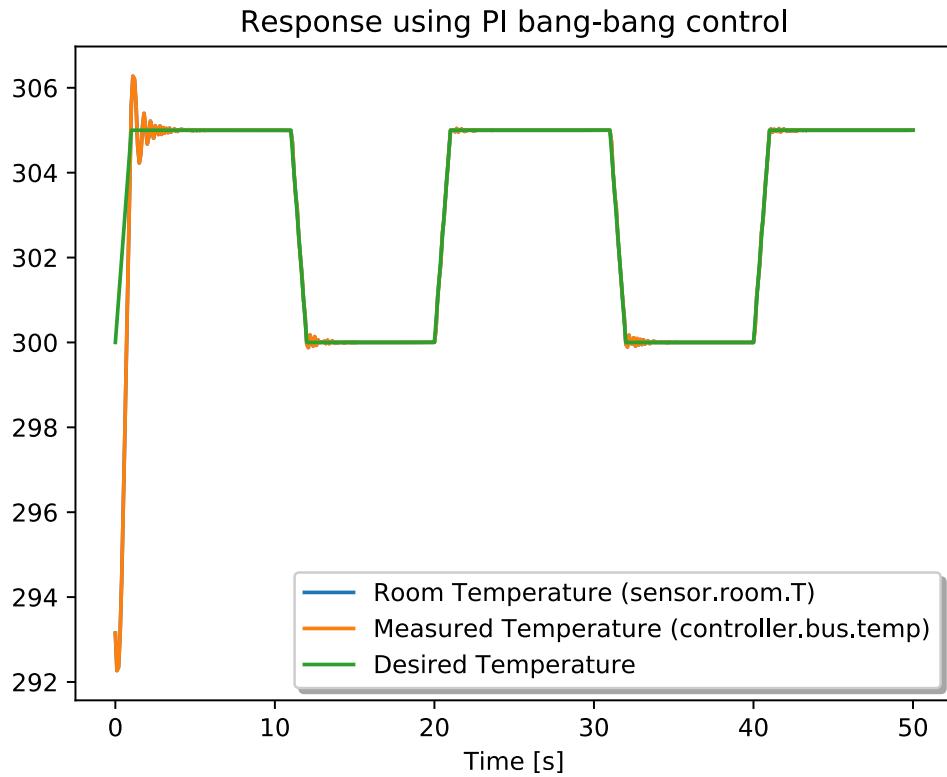


The source code for our system model is quite simple:

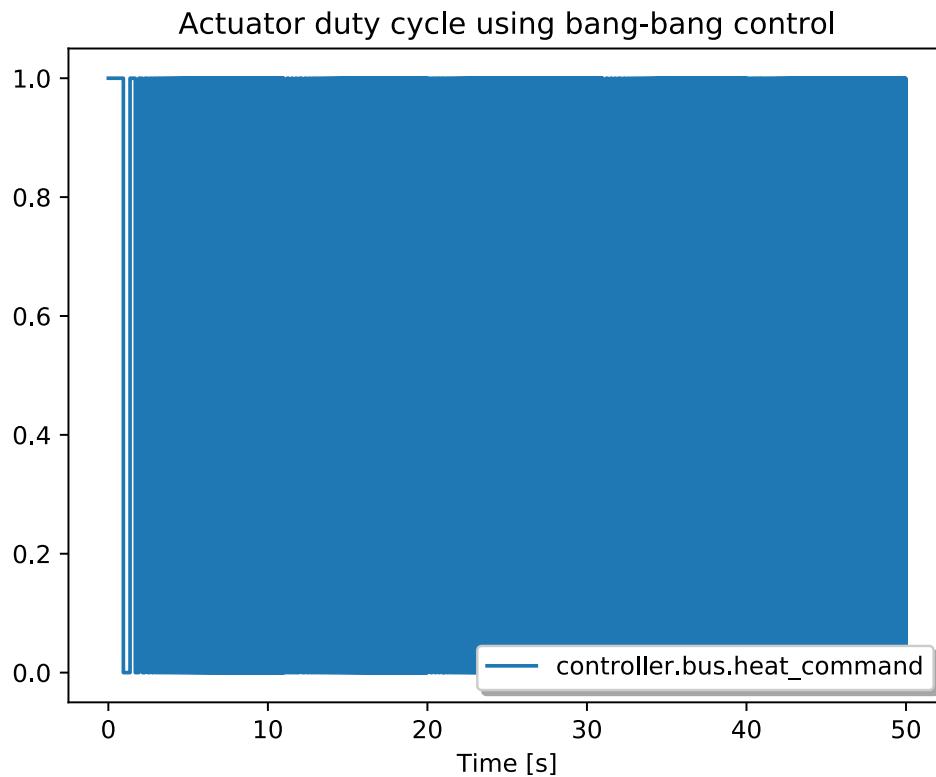
```

within ModelicaByExample.Architectures.ThermalControl.Examples;
model OnOffVariant "Variation with on-off control"
  extends ExpandableModel(
    redeclare replaceable
      Implementations.OnOffActuator actuator(heating_capacity=500),
    redeclare replaceable
      Implementations.OnOffControl controller(setpoint=300));
end OnOffVariant;

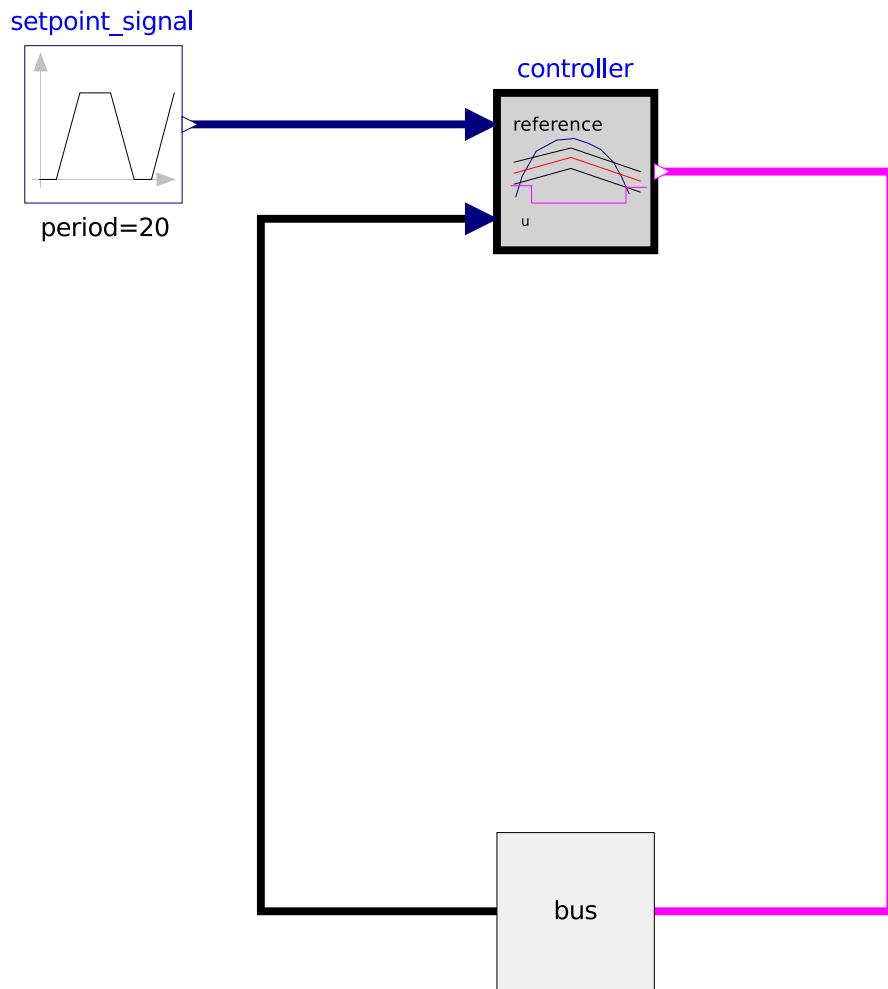
```



However, there is still one remaining issue with these models and it can be seen more clearly if we look at the duty cycle of the furnace:



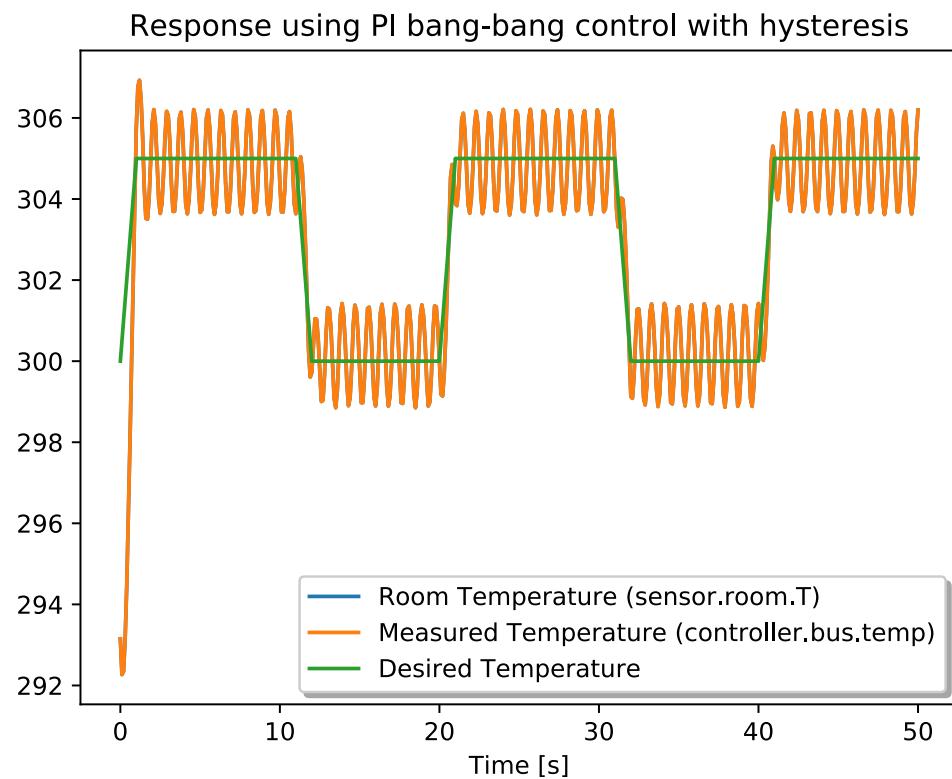
This is exactly the same issue we demonstrated in the previous section on [Hysteresis](#) (page 72). It is precisely the fact that our control strategy lacks any hysteresis that we see the furnace constantly turning on and off. If we add hysteresis, our controller model becomes:



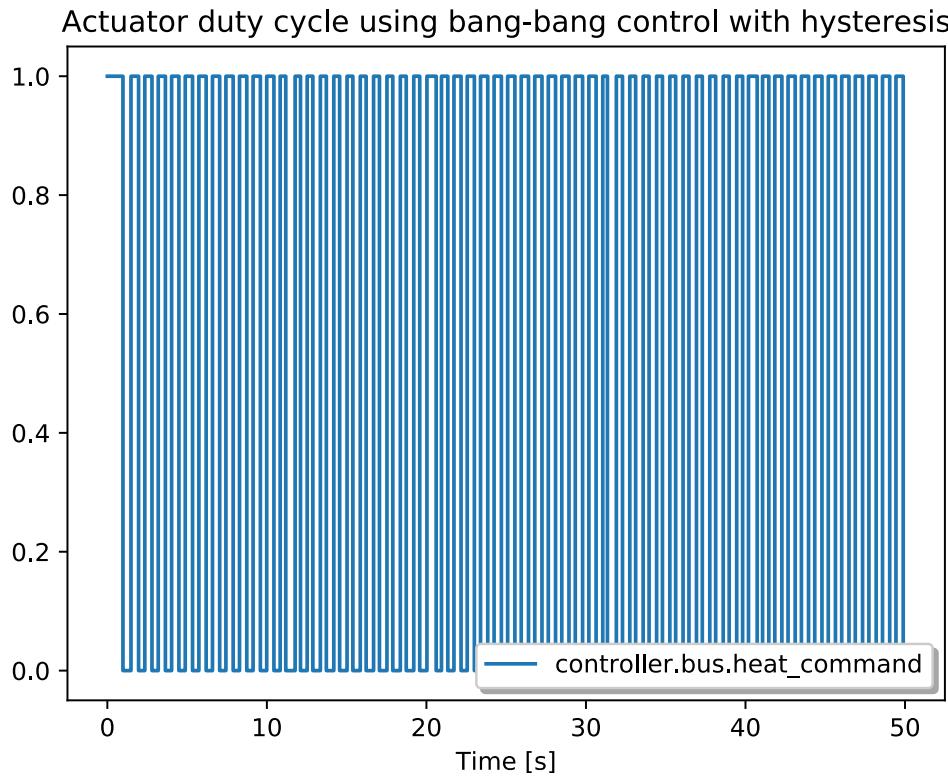
Nothing else has changed. We will use the same sensor and actuator models and we still use the same bus signals since this is still a bang-bang controller. So the only change to our system level model (compared to the `OnOffVariant` model) is the use of a different controller model. As we can see, these configuration management features in Modelica do a nice job of conveying our configuration choices in our system level model:

```
within ModelicaByExample.Architectures.ThermalControl.Examples;
model HysteresisVariant "Using on-off controller with hysteresis"
  extends OnOffVariant(redeclare Implementation.OnOffControl_WithHysteresis
    controller(setpoint=300, bandwidth=1));
end HysteresisVariant;
```

Using hysteresis control, our simulation results look like this:



But the most important difference is the fact that the hysteresis doesn't lead to the kind of chattering we saw in our previous bang-bang controller:



Conclusion

This is the second example of how we can use the configuration management features in Modelica to take an architecturally based approach to building system models. This architectural approach is very useful when there are many variations of the same architecture that require analysis. Using the `redeclare` feature, it is possible to easily substitute alternative designs for each subsystem or to consider more or less detail in any given subsystem as necessary for any given engineering analysis.

In this particular example, we saw how an `expandable` connector can provide greater flexibility than a standard connector. However, it also comes with some risk because the type checking normally done by the Modelica compiler is less rigorous.

2.5.2 Review

Interfaces and Implementations

Conceptual Definitions

In both of the examples we presented in this chapter, we used interface definitions as part of the architecture definition process. The term “interface” doesn’t come from Modelica itself, it is a term common among computer languages. In Modelica, we can think of interfaces as models that define all the details of the model **that are externally visible**. You can think of an interface as a “shell” without any internal details. For this reason, interface models are almost always marked as `partial`.

Another important concept is that of an “implementation”. This is another term borrowed from the world of computer languages. Whereas an interface is used to simply describe the externally visible aspects of a model, an implementation includes internal details as well. It includes the information required to

actually implement that interface. In some cases, it may only constitute a partial implementation (in which case it should also be marked as **partial**). In other cases, it may represent the architecture of a particular subsystem where further implementation details are pushed one additional level down in the model hierarchy (another case of a **partial** model). But most of the time, these implementations will be complete (**non-partial**) models for a particular subsystem.

Plug-Compatibility

The most important thing we need to consider when we talk about interfaces and implementations is the notion of **plug-compatibility**. As we already discussed in our elaboration of the *Sensor Comparison* (page 304) example, a model X is plug-compatible with a model Y if for every **public** variable in Y, there is a corresponding public variable in X with the same name. Furthermore, every such variable in X must itself be plug-compatible with its counterpart in Y. This ensures that if you change a component of type Y into a component of type X that everything you need (parameters, connectors, etc) will still be there and will still be compatible. **However, please note** that if X is plug-compatible with Y, this **does not** imply that Y is plug-compatible with X (as we will see in a moment).

Generally speaking, most cases where we concern ourselves with plug-compatibility revolve around whether a given implementation is plug-compatible with a given interface. As we've seen in these examples (and we will review shortly), the configuration management features in Modelica hinge on the relationship between interfaces and implementations and the process by which configuration management is performed is centered around plug-compatibility.

Conclusion

The bottom line is that it is very useful to not only think in terms of interface and implementation models, but also to create models to formally define interfaces and distinguish them from implementations, since these will be very useful when creating architecture driven models.

Configuration Management

`replaceable`

What really enables the configuration management features in Modelica is the **replaceable** keyword. It is used to identify components in a model whose type can be changed (or “redeclared”) in the future. One way to think about **replaceable** is that it allows the model developer to define “slots” in the model that are either “blank” to begin with (where an interface model is the original type in the declaration) or at least “configurable”.

The advantage of using the **replaceable** keyword is that it allows new models to be created without having to reconnect anything. This not only imposes a structural framework on future models (to ensure naming conventions are followed, interfaces are common, *etc.*), it also helps avoid potential errors by eliminating an error prone task from the model development process, *i.e.*, creating connections.

To make a component replaceable the only thing that is necessary is to add the **replaceable** keyword in front of the declaration, *i.e.*,

```
replaceable DeclaredType variableName;
```

where **DeclaredType** is the initial type for a variable named **variableName**. In such a declaration, the **variableName** variable can be given a new type (we will discuss how very shortly). But any new type used for **variableName** must be *plug-compatible* (page 346).

constrainedby

As we just mentioned, by default the new type of any `replaceable` component must be plug-compatible with the initial type. But this doesn't have to be the case. As our earlier discussion on *Constraining Types* (page 312) pointed out, it is possible to specify both a default type for the variable to have and a separate constraining type that any new type needs to be compatible with.

Specifying an alternative constraining type requires the use of the `constrainedby` keyword. The syntax for using the `constrainedby` keyword is:

```
replaceable DefaultType variableName constrainedby ConstrainingType;
```

where `variableName` is again the name of the variable being declared, `DefaultType` represents the type of `variableName` of the type of `variableName` is never changed and `ConstrainingType` indicates the constraining type. As mentioned previously, any new type attributed to the `variableName` variable must be plug-compatible with the constraining type. But, of course, the `DefaultType` must also be plug-compatible with the constraining type.

constrainedby vs. extends

Older versions of Modelica didn't include the `constrainedby` keyword. Instead, the `extends` keyword was used instead. But it was felt that inheritance and plug-compatibility were distinct enough that a separate keyword would be less confusing. So don't be confused if you are looking at Modelica code and the keyword `extends` is used where you would expect to see the `constrainedby` keyword (*i.e.*, following a `replaceable` declaration).

redeclare

So now we know that by using the `replaceable` keyword, we can change the type of a variable in the future. Changing the type is called "redeclaring" the variable (*i.e.*, to have a different type). For this reason, it is fitting that the keyword used to indicate a redeclaration is `redeclare`. Assume that we have the following system model:

```
model System
  Plant plant;
  Controller controller;
  Actuator actuator;
  replaceable Sensor sensor;
end System;
```

In this `System` model, only the `sensor` is `replaceable`. So the types of each of the other subsystems (*i.e.*, `plant`, `controller` and `actuator`) cannot be changed.

If we wanted to extend this model, but use a different model for the `sensor` subsystem, we would use the `redeclare` keyword as follows:

```
model SystemVariation
  extends System(
    redeclare CheapSensor sensor
  );
end SystemVariation;
```

What this tells the Modelica compiler is that in the context of the `SystemVariation` model, the `sensor` subsystem should be an instance of the `CheapSensor` model, not the (otherwise default) `Sensor` model. However, the `CheapSensor` model (or any other type chosen during redeclaration) **must be plug-compatible with that variable's constraining type**.

The syntax of a `redeclare` statement is really exactly the same as a normal declaration except that it is preceded by the `redeclare` keyword. Obviously, any variable that is redeclared had to be declared in the first place (*i.e.*, you cannot use this syntax to declare a variable, only to *redeclare* it).

It is **very important** to understand that when you redeclare a component, the new redeclaration supersedes the previous one. For example, after the following redeclaration:

```
redeclare CheapSensor sensor;
```

the `sensor` component is **no longer replaceable**. This is because the new declaration doesn't include the `replaceable` keyword. As a result, it is as if it was never there. If we wanted the component to remain replaceable, the redeclaration would need to be:

```
redeclare replaceable CheapSensor sensor;
```

Furthermore, if we choose to make the redeclared variable replaceable, we also have the option **to redeclare the constraining type**, like this:

```
redeclare replaceable CheapSensor sensor constrainedby NewSensorType;
```

However, the original constraining type still plays a role even in this case because the type `NewSensorType` must be plug-compatible with the original constraining type. In the terminology of programming languages, we can narrow the type (reducing the set of things that are plug-compatible), but we can never widen the type (which would make things that were previously not plug-compatible now plug-compatible).

Earlier, when discussing *Arrays of Component* (page 300), we made the point that it was not possible to redeclare individual elements in arrays. Instead, a redeclaration must be applied to the entire array. In other words, if we declare something initially as:

```
replaceable Sensor sensors[5];
```

It is then possible to redeclare the array, *e.g.*,

```
redeclare CheapSensor sensors[5];
```

But the important point is that the redeclaration affects every element of the `sensors` array. There is no way to redeclare only one element.

Modifications

One important complexity that comes with replaceability is what happens to modifications in the case of a redeclaration. To understand the issue, consider the following example.

```
replaceable SampleHoldSensor sensor(sample_time=0.01)
  constrainedby Sensor;
```

Now, what happens if we were to redeclare the `sensor` as follows:

```
redeclare IdealSensor sensor;
```

Is the value for `sample_time` lost? We would hope so since the `IdealSensor` model probably doesn't have a parameter called `sample_time` to set.

But let's consider another case:

```
replaceable Resistor R1(R=100);
```

Now imagine we had another resistor model, `SensitiveResistor` that was plug-compatible with `Resistor` (*i.e.*, it had a `parameter` called `R`) but included an additional parameter, `dRdT`, indicating the (linear) sensitivity of the resistance to temperature. We might want to do something like this:

```
redeclare SensitiveResistor R1(dRdT=0.1);
```

What happens to `R` in this case? In this case, we would actually like to preserve the value of `R` so it persists across the redeclaration. Otherwise, we'd need to restate it all the time, *i.e.*,

```
redeclare SensitiveResistor R1(R=100, dRdT=0.1);
```

and this would violate the DRY principle. The result would be that any change in the original value of `R` would be overridden by any redeclarations.

So, we've seen two cases valid use cases. In one case, we don't want a modification to persist following a redeclaration and in the other we would like the modification to persist. Fortunately, Modelica has a way to express both of these. The normal Modelica semantics take care of the first case. If we redeclare something, all modifications from the original declaration are erased. But what about the second case? In that case, the solution is to **apply the modifications to the constraining type**. So for our resistor example, our original declaration would need to be:

```
replaceable Resistor R1 constrainedby Resistor(R=100);
```

Here we explicitly list both the default type `Resistor` and the constraining type `Resistor(R=100)` separately because the constraining type now includes a modification. By moving the modification to the constraining type, **that modification will automatically be applied to both the original declaration and any subsequent redeclarations**. So in this case, the resistor instance `R1` will have an `R` value of 100 even though the modification isn't directly applied after the variable name. But furthermore, if we perform the redeclaration we discussed previously, *i.e.*,

```
redeclare SensitiveResistor R1(dRdT=0.1);
```

the `R=100` modification will automatically be applied here as well.

In summary, if you want a modification to apply only to a specific declaration and not in subsequent redeclarations, apply it after the variable name. If you want it to persist through subsequent redeclarations, apply it to the constraining type.

Redefinitions

It turns out that the `replaceable` keyword can also be associated with *definitions*, not just declarations. The main use of this feature is to be able to change the type of **multiple** components at once. For example, imagine a circuit model with several different resistor components:

```
model Circuit
  Resistor R1(R=100);
  Resistor R2(R=150);
  Resistor R4(R=45);
  Resistor R5(R=90);
  // ...
equation
  connect(R1.p, R2.n);
  connect(R1.n, R3.p);
  // ...
end Circuit;
```

Now imagine we wanted one version of this model with ordinary `Resistor` components and the other where each resistor was an instance of the `SensitiveResistor` model. One way we could achieve this would be to define our `Circuit` as follows:

```
model Circuit
  replaceable Resistor R1 constrainedby Resistor(R=100);
  replaceable Resistor R2 constrainedby Resistor(R=150);
  replaceable Resistor R4 constrainedby Resistor(R=45);
```

```

replaceable Resistor R5 constrainedby Resistor(R=90);
// ...
equation
  connect(R1.p, R2.n);
  connect(R1.n, R3.p);
// ...
end Circuit;

```

But in that case, our circuit with `SensitiveResistor` components would be defined as:

```

model SensitiveCircuit
  extends Circuit(
    redeclare SensitiveResistor R1(dRdT=0.1),
    redeclare SensitiveResistor R2(dRdT=0.1),
    redeclare SensitiveResistor R3(dRdT=0.1),
    redeclare SensitiveResistor R4(dRdT=0.1)
  );
end SensitiveCircuit;

```

Note that we don't have to specify resistance values because the modifications that set the resistance were applied to the constraining type in our `Circuit` model. But, it is a bit tedious that we have to change each individual resistor and specify `dRdT` over and over again even though they are all the same value. However, Modelica gives us a way to do them all at once. The first step is to define a local type within the model like this:

```

model Circuit
  model ResistorModel = Resistor;
  ResistorModel R1(R=100);
  ResistorModel R2(R=150);
  ResistorModel R4(R=45);
  ResistorModel R5(R=90);
// ...
equation
  connect(R1.p, R2.n);
  connect(R1.n, R3.p);
// ...
end Circuit;

```

What this does is establish `ResistorModel` as a kind of alias for `Resistor`. This by itself doesn't help us with changing the type of each resistor easily. But making `ResistorModel` `replaceable` does:

```

model Circuit
  replaceable model ResistorModel = Resistor;
  ResistorModel R1(R=100);
  ResistorModel R2(R=150);
  ResistorModel R4(R=45);
  ResistorModel R5(R=90);
// ...
equation
  connect(R1.p, R2.n);
  connect(R1.n, R3.p);
// ...
end Circuit;

```

If our `Circuit` is defined in this way, we can create the `SensitiveCircuit` model as follows:

```

model SensitiveCircuit
  extends Circuit(
    redeclare model ResistorModel = SensitiveResistor(dRdT=0.1)
  );
end SensitiveCircuit;

```

All our resistor components are still of type `ResistorModel`, we didn't have to redeclare any of them. What we `did do` was redefine what a `ResistorModel` is by changing its definition to `SensitiveResistor(dRdT=0.1)`. Note that the modification `dRdT=0.1` will be applied to all components of type `ResistorModel`. Technically, this isn't a redeclaration of a component's type, it is a redefinition of a type. But we reuse the `redeclare` keyword.

Interestingly, with these redefinitions we still have the notion of a default type and a constraining type. The general syntax for a redefinable type is:

```
replaceable model AliasType = DefaultType(...) constrainedby ConstrainingType(...);
```

Just as with a replaceable component, any modifications associated with the default type, `DefaultType`, are only applied in the case that `AliasType` isn't redefined. But, any modification associated with the constraining type, `ConstrainingType`, will persist across redefinitions. Furthermore, `AliasType` must always be plug compatible with the constraining type.

Although this aspect of the language is less frequently used, compared to replaceable components, it can save time and help avoid errors in some cases.

Choices

This section has focused on configuration management and we've learned that the constraining type controls what options are available when doing a `redeclare`. If a single model developer creates an architecture and all compatible implementations, then they have a very good sense of what potential configurations will satisfy the constraining types involved.

But what if you are using an architecture developed by someone else? How can you determine what possibilities exist? Fortunately, the Modelica specification includes a few standard annotations that help address this issue.

choices

The `choices` annotation allows the original model developer to associate a list of modifications with a given declaration. The very simplest use case for this could be to specify values for a given parameter:

```
parameter Modelica.SIunits.Density rho
annotation(choices(choice=1.1455 "Air",
choice=992.2 "Water"));
```

In this case, the model developer has listed several possible values that the user might want to give to the `rho` parameter. Each choice is a modification to be applied to the `rho` variable. This information is commonly used by graphical Modelica tools to provide users with intelligent choices.

This feature can just as easily be used in the context of configuration management. Consider the following example:

```
replaceable IdealSensor sensor constrainedby Sensor
annotation(
choices(
choice(redeclare SampleHoldSensor sensor
"Sample and hold sensor"),
choice(redeclare IdealSensor sensor
"An ideal sensor")));
```

Again, the model developer is embedding a set of possible modifications along with the declaration. These `choice` values can also be used by graphical tools to provide a reasonable set of choices when configuring a system.

choicesAllMatching

But one problem here is that it is not only tedious to have to explicitly list all of these choices, but the set of possibilities might change. After all, other developers (besides the original model developer) might come along and create implementations that satisfy a given constraining type. How about giving users the option of seeing **all** legal options when configuring their system?

Fortunately, Modelica includes just such an annotation. It is the `choicesAllMatching` annotation. By setting the value of this annotation to `true` on a given declaration (or `replaceable` definition), this instructs the tool to find all possible legal options and present them through the user interface. For example,

```
replaceable IdealSensor sensor constrainedby Sensor  
annotation(choicesAllMatching=true);
```

By adding this annotation, the tool knows to find all legal redeclarations when a user is reconfiguring their models through the graphical user interface. This can increase the usability of architecture based models **enormously** because it presents users with the full range of options at their disposal with trivial effort on the part of the model developer.

Conclusion

In this section, we've discussed the configuration management features in Modelica. As with other aspects of the Modelica language, the goals here are the same: promote reuse, increase productivity and ensure correctness. Modelica includes many powerful options for redeclaring components and redefining types. By combining this with the `choicesAllMatching` annotation, models can be built to support a large combination of possible configurations using clearly defined choice points.

Expandable Connectors

Definition

As we saw in the *Thermal Control* (page 328) example from this chapter, when building architectures it can be difficult to define an interface that covers all important configurations. One option, in this case, is to use `expandable` connectors.

An expandable connector uses exactly the same syntax as a normal connector definition (see previous discussion on *Connector Definitions* (page 177)) with the exception that the `expandable` qualifier is added before the definition, *e.g.*,

```
expandable connector ConnectorName "Description of the connector"  
  // Declarations for connector variables  
end ConnectorName;
```

We can ignore the `expandable` qualifier and just treat this connector like a normal connector if we want. The assumption is that components that use such a connector as a normal connector will provide the correct number of equations such that any component models are *Balanced Components* (page 257).

If, however, a user connects to a named element that is **not** part of the connector definition, this causes a new element to be added to that particular instance of the connector. The type of this new element will be identical to the connector it is connected to. Ultimately, that element will be added to any other connector instances in the *connection set* (page 254) as well.

In this way, an `expandable` connector definition can be “grown” to add additional signals not in the original definition. In this way, any interface definition with an expandable connector has the flexibility to exchange more information than just what was in the initial connector definition. As we saw in the *Thermal Control* (page 328) example, this can be very useful when the interface can vary significantly depending on the choice of subsystems.

By far, the most common use case for expandable connectors is to add additional `input` and `output` signals to a connector. This frequently occurs because the information required when pairing controllers up with sensors and actuators can change depending on the implementation details of those subsystems. For example, an internal combustion engine with cam phasing will require a commanded cam angle from the controller. But an engine without this feature will not.

However, the `expandable` connector can also be used with acausal sub-connectors to add additional points of physical interaction between subsystems. For example, one subsystem may not include thermal effects while another one does. The one that does could include a thermal acausal connector to allow other subsystems to interact thermally with it.

Conclusion

Expandable connector definitions define a *minimum* set of variables that should appear on the connector. It is assumed that all component models will be balanced with respect to those variables in the connector definition. In addition, additional variables can be added to an `expandable` connector as long as the overall system model is still balanced. Expandable connectors are typically used in conjunction with interface definitions to define a minimum interface within an architecture that can be expanded by the choice of implementations.

CHAPTER
THREE

INDICES AND TABLES

- genindex

BIBLIOGRAPHY

- [Lotka] Lotka, A.J., "Contribution to the Theory of Periodic Reaction", J. Phys. Chem., 14 (3), pp 271–274 (1910)
- [Volterra] Volterra, V., Variations and fluctuations of the number of individuals in animal species living together in Animal Ecology, Chapman, R.N. (ed), McGraw-Hill, (1931)
- [Guldberg] C.M. Guldberg and P. Waage,"Studies Concerning Affinity" C. M. Forhandlinger: Videnskabs-Selskabet i Christiana (1864), 35
- [Elmqvist] "Fundamentals of Synchronous Control in Modelica", Hilding Elmqvist, Martin Otter and Sven-Erik Mattsson <http://www.ep.liu.se/ecp/076/001/ecp12076001.pdf>
- [Berg] Richard E. Berg, "Pendulum waves: A demonstration of wave motion using pendula" <http://dx.doi.org/10.1119/1.16608>
- [Tiller2001] Michael M. Tiller, "Introduction to Physical Modeling with Modelica" <http://www.amazon.com/Introduction-Physical-Modeling-International-Engineering/dp/0792373677>

INDEX

A

ABCD, 87
algorithm
 in a function, 117
algorithm section, 70
annotation, 5, 120
 associated with
 declarations, 38
 definitions, 38
 equation, 38
 extends, 38
 models, 38
 defaultComponentName, 262
 defaultComponentPrefixes, 263
 derivative, 120
 Dialog, 263
 Documentation, 40
 DynamicSelect, 263
 Evaluate, 41
 experiment, 5, 41
 HideResult, 41
 Placement, 261
 preferredView, 264
 unassignedMessage, 264
annotations
 Icon, 175
 standard annotations
 derivative, 145
 Include, 149
 IncludeDirectory, 150
 Inline, 148
 inverse, 148
 LateInline, 149
 Library, 150
 LibraryDirectory, 150
array comprehensions, 96
arrays
 assignment, 110
 equations, 110
 mathematical operations
 addition, 108
 division, 109
 exponentiation, 110
 multiplication, 109
 subtraction, 109
of components, 300

 relational operators, 110
assert, 219, 258, 259
assertion levels, 29
AssertionLevel, 29
assertions, 259
attribute, 33
attribute modification, 33
attributes, 9, 30

B

balanced models, 257
block, 258
Boolean, 29
break, 141
bus, 335

C

camel case, 27
candidate solutions, 259
composition, 25
conditional declaration, 214
connect, 186, 260
 flow, 254
 input, 260
 output, 260
 parameter, 260
connection set, 254
connector, 177
connectors
 expandable, 334
constant, 28
constraining types, 313
cross, 111

D

default type, 313
der, 1
derived types, 9, 29
descriptive strings, 2
diagonal, 107
differential algebraic equations, 92
discrete, 29, 65
displayUnit attribute, 30

E

encapsulated, 161, 217

enumeration, **29**, 217, 248
events, **82**
expandable
 caveats, **338**
expandable connectors, **334**
extends, **26**, **27**, 33, 64
external, **142**
ExternalObject, 125

F

fill, **105**
final, **303**
fixed attribute, **30**
flow, **170**
FMI, **87**
for, **141**
fully qualified name, 153
function arguments, 117
functions
 cross, **111**
 diagonal, **107**
 differentiating, 120
 fill, **105**
 for, **141**
 identity, **106**
 linspace, **107**
 matrix, **108**
 max (vector), **111**
 min (vector), **111**
 ndims, **112**
 ones, **106**
 outerProduct, **110**
 product, **112**
 scalar, **107**
 size, **112**
 skew, **111**
 sum, **112**
 symmetric, **111**
 transpose, **110**
 vector, **107**
 vectorization, **112**
while, **141**
zeros, **106**

H

hierarchical, **33**
Hooke's law, **14**

I

icons
 associating with definitions, **175**
identity, **106**
if, **85**
if expression, **45**
if statement, **43**
import, **162**
infinite reservoir, **189**
inheritance, **25**, **27**

initial equation, 3, **35**
initialization, **35**
input, **117**
Integer, **29**

K

kinematic constraint, **206**

L

linspace, **107**

M

matrix, **108**
 addition, **108**
 division, **109**
 exponentiation, **110**
 multiplication, **109**
 subtraction, **109**
 transpose, **110**
matrix-vector products, **109**
max, **46**
max (vector), **111**
max attribute, **30**
min (vector), **111**
min attribute, **10**, **30**
model, **1**, **27**
model limitations, **259**
MODELICAPATH, **159**
models
 balanced, **257**
 equations
 number, **257**
modification, **33**
modifications, **16**, **26**, **33**
 in the context of redeclarations, 348

N

ndims, **112**
nominal attribute, **30**
number of equations required, **189**

O

ones, **106**
ordinary differential equations, **92**
outer product, **110**

P

packages
 ordering within, **159**
parameter, **7**, **28**
parametric variability, **34**
partial, **194**
partial differential equations, **92**
product, **112**
protected, **297**
public, **297**

Q

quantity attribute, [30](#)

R

reaction torque, [206](#)

Real, [1](#), [29](#)

record, [36](#)

record constructor, [37](#)

redeclare, [312](#)

reinit, [86](#)

replaceable, [311](#)

return, [141](#)

S

scalar, [107](#)

side effects, [133](#)

size, [112](#)

skew, [111](#)

slicing, [300](#)

start attribute, [20](#), [30](#)

state events, [42](#), [82](#)

state-space form, [12](#)

states, [35](#)

stateSelect attribute, [30](#)

String, [29](#)

sum, [112](#)

T

time, [82](#)

time events, [42](#), [82](#)

transpose, [110](#)

type, [10](#), [29](#)

U

unit attribute, [9](#), [30](#)

V

vector, [107](#)

addition, [108](#)

assignment, [110](#)

division, [109](#)

equations, [110](#)

exponentiation, [110](#)

multiplication, [109](#)

subtraction, [109](#)

vectorization, [112](#)

vectors

outer product, [110](#)

relational operators, [110](#)

W

when, [85](#)

while, [141](#)

Z

zeros, [106](#)