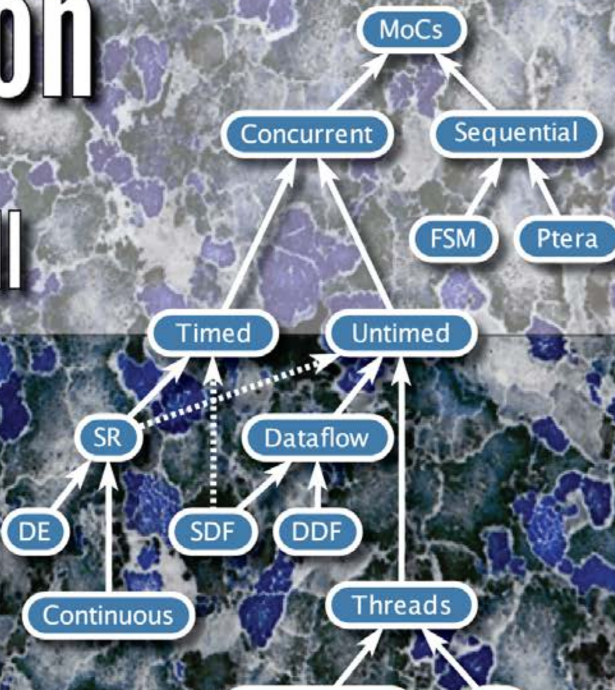
System Design, Modeling, and Simulation

Using Ptolemy II



Claudius Ptolemaeus, Editor

First Edition, Version 1.02

ISBN: 978-1-304-42106-7

Please cite this book as:

Claudius Ptolemaeus, Editor,
System Design, Modeling, and Simulation using Ptolemy II, Ptolemy.org, 2014.
http://ptolemy.org/systems.

Contents

Preface

My last written work was published nearly 1,900 years ago. I am pleased to come out of retirement to give voice to a project that I'm proud to have named after me, the Ptolemy Project. Like much of my prior work in astronomy and geography, this project deals with complex systems. Like most of my prior writings, this text compiles the thinking and contributions of many people.

The motions of the planets, the sun, the earth, and the moon, which I studied in my work *The Almagest*, are concurrent interacting processes. They are *deterministic*, not subject to the whims of the gods. More accurately, the models that I developed, as well as those of many of my successors, deliberately ignore any effects that the gods might capriciously impose. These models focus instead on precisely matching observed behavior, and more importantly, on predicting behavior. The Ptolemy Project similarly studies concurrent processes and focuses on deterministic models.

Ideally, an intellectual quest moves human knowledge from superstition and unfounded beliefs to logic and measurement. What we now call "science," particularly in the study of natural systems, is deeply rooted in the scientific method, where we form hypotheses, design experiments, and draw conclusions about the hypotheses based on the experiments. To be able to make measurements, of course, the artifact or process being measured must exist in some form. In my earlier studies, this was not an issue, since the sun, earth, moon, and planets already existed. Engineering disciplines, which focus on human-constructed artifacts and processes, however, study systems *that do not yet exist*. Nevertheless, the scientific method can and is applied in engineering design. Engineers construct simulations and prototypes of systems, formulate hypotheses, and perform experiments to test those hypotheses.

Because of the focus on artifacts and processes that do not yet exist, engineering design should not be based solely on the scientific method. The goal of experiments is to improve our understanding of the artifact or process being designed. But we have to create the artifact or process before we can perform experiments. Being forced to create something before we understand it dooms our design to roots in superstition and unfounded beliefs.

An important part of a science, quite complementary to the scientific method, is the construction of models. Models are abstractions of the physical reality, and the ability of a model to lend insight and predict behavior may form the centerpiece of a hypothesis that is to be validated (or invalidated) by experiment. The construction of models is itself more an engineering discipline than a science. It is not, fundamentally, the study of a system that preexists in nature; it is instead the human-driven construction of an artifact that did not previously exist. A model itself must be engineered.

Good models can even reduce the need for measurement, and therefore reduce the dependence on the scientific method. Once we have a model of the motions of the planets, for example, that we know accurately predicts their positions, there is less need to measure their positions. The role of measurements changes from determining the positions of the planets to improving the models of their motions and to detecting capricious actions of the gods (something that engineers call "fault detection").

In both science and engineering, models can be iteratively improved. My own geocentric model of the universe required much iterative refinement to bring it into close conformance with experimental observation of the motions of the planets. I am quite proud of the predictive power of the resulting models, and also of the way predictions based on these models could be mechanized by an *astrolabe*. I nevertheless acknowledge that my esteemed colleague Nicolaus Copernicus provides a better model for the motion of the planets. His model is conceptually simpler, once you make the leap that the center of our *observable* universe, the ground we stand on, need not be the center of a *model* of the universe. Indeed, we have much more freedom with models than with the physical world, as models need not be constrained by nature. Nevertheless, my models were the best ones available for nearly 1400 years.

The Ptolemy Project is indeed a study of models of systems. The systems, however, are quite different from the ones I focused on in the past. Those were given to me by nature, but the ones in this book are created by humans. In this book, the purpose of modeling is to improve systems, and there is nothing I could have done to improve the planetary system given to us by nature.

In short, in engineering, as opposed to science, models play a role in the *design* of the systems being modeled. As with science, models can be improved, but unlike science, so can the systems being modeled.

Even more interestingly, in engineering, unlike science, the choice of models affects the systems being modeled. Two engineers, given the same goals, may come up with radically different designs and realizations of a system simply because they started with radically different *models* of the system. Moreover, these two engineers may have come up with different models simply because they started with different tools for constructing models. An engineer who constructs models with pencil and paper will likely come up with very different models than those of an engineer who starts with a software modeling tool. As a consequence, they will likely come up with very different system designs.

This text collects an astonishingly rich variety of modeling tools and techniques for complex systems. Some of these will undoubtedly be improved upon in the future, like my own epicycles, a modeling complexity that was rendered unnecessary by Copernicus. Nevertheless, the goal of this book is to offer to engineers the best modeling techniques available today, with the hope that this will lead to the best system designs achievable today. Tomorrow, we can be sure we will be able to do still better.

Claudius Ptolemaeus

How to Use This Book

The book is intended for use by engineers and scientists who need to model a wide variety of systems or who want to understand how to model complex, heterogeneous systems. Such systems include, for example, mechanical systems, electrical systems, control systems, biological systems, and – most interestingly – heterogeneous systems that combine elements from these (and other) domains. The book assumes a familiarity with simulation and modeling tools and techniques, but does not require in-depth background in the subject matter.

The book emphasizes modeling techniques that have been realized in Ptolemy II. Ptolemy II is an open-source simulation and modeling tool intended for experimenting with system design techniques, particularly those that involve combinations of different types of models. It was developed by researchers at UC Berkeley, and over the last two decades it has evolved into a complex and sophisticated tool used by researchers around the world. This book uses Ptolemy II as the basis for a broad discussion of system design, modeling, and simulation techniques for hierarchical, heterogeneous systems. But it also uses Ptolemy II to ensure that the discussions are not abstract and theoretical. All of the techniques are backed by a well-designed and well-tested software realization. More detailed descriptions of Ptolemy's underlying software architecture and the finer points of its operation and underlying theory are found in sidebars, references, and links.

The electronic version of the book provides a number of links to material both within the book and on external websites. Furthermore, most of the illustrations showing models provide a hyperlink that enables you to view, download, edit, and run the model shown in the figure. Such figures are indicated in caption with a link reading "[online]" that will first take you to a web page that enables browsing the model and downloading it to run on your own Ptolemy II installation (available from `http://ptolemy.org`). This web page also provides a link to a **Java Web Start** deployment of Ptolemy II, which circumvents the steps of downloading and installing the software. The Java Web Start deployment, however, will not work if your browser does not have Java enabled or if you are reading on a machine that does not have Java capability (such as an iPad). For the print version of the book, to get to the models marked with "online," go to `http://ptolemy.org/systems`.

The book is organized in three parts. First is an introduction. The first chapter outlines the principles behind the styles of modeling that are covered in this book, and it gives a tour of the multiple models of computation (MoCs) that are described. The second chapter is a tutorial on using Ptolemy II via its graphical editor, Vergil. This chapter is a good starting point for the impatient reader who wants to just get directly into building models.

The second part, Chapters 3 through 11, covers models of computation. Each chapter covers one or a small family of related models of computation, explaining how they work, how to build models using them, and what sorts of models are well matched to the MoCs.

The third part covers capabilities of Ptolemy II that span models of computation. Chapter 12, perhaps the most important one for readers who are interested in extending Ptolemy II or in writing their own actors in Java, describes the software architecture. Ptolemy is open-source software, with well-documented code that is meant to be read, and this chapter serves as a good orientation for readers who would like to look at the code and build on it. Chapter 13 describes the expression language that is used for specifying parameter values for models and for giving custom functions to actors. Chapter 17 describes the capabilities of the signal plotters that are included in the Ptolemy II standard library. Chapter 14 describes the type system in Ptolemy II; it is a sophisticated type system designed to minimize the burden on the model builder (by emphasizing type inference rather than type declarations) while maximizing safety by providing a strong type system. Chapter 15 describes ontologies, which extend the type system with an ability to associate units, dimensions, and concepts with data values in models. Again, the emphasis is on inference and safety. Finally, Chapter 16 describes the web interfaces included with Ptolemy II. Specifically, it explains the capabilities for exporting web pages from models and for building web servers and services into models.

At the end of the book is an extensive bibliography (where most entries provide hyperlinks to the documents that are cited), a comprehensive actor index (listing all the actors in the Ptolemy II standard library), and an extensive index with more than 4,000 entries.

Acknowledgements

The modeling techniques described in this book have been developed in the Ptolemy Project at the University of California at Berkeley over many years. The roots of the project date back to the 1980s. The first germ came from Messerschmitt (1984), who built a software framework called **Blosim** (for Block Simulator) for simulating signal processing systems. Messerschmitt's PhD student, Edward A. Lee, inspired by Blosim, developed the synchronous dataflow (SDF) model of computation (Lee, 1986; Lee and Messerschmitt, 1987b) and a scheduling theory for it (Lee and Messerschmitt, 1987a). Lee and his students then developed the Lisp-based software tool called **Gabriel** (Lee et al., 1989), with which they developed and refined the SDF model of computation. In the early 1990s, Lee and Messerschmitt set out to develop an object-oriented block diagram framework called **Ptolemy** (Buck et al., 1994) (now called **Ptolemy Classic**). Later in the 1990s, Lee and his group started a complete redesign called **Ptolemy II** based on the brand-new programming language Java (Eker et al., 2003). Ptolemy II developed the key ideas of actor-oriented design (Lee et al., 2003) and hierarchical heterogeneity.

The trajectory of the software in many ways reflects the evolution of computing during this time period. Blosim was written in C. Gabriel was written in Lisp. The first Ptolemy system, now called Ptolemy Classic, was written in C++. The next version, Ptolemy II, was written in Java. Each of these transitions reflected an effort to leverage the best available technology to solve real design problems. Lisp made sense over C because of its robustness and suitability for specification of complex decision logic. C++ made sense over Lisp because of its much better developed (at the time) notions of object-oriented design (particularly inheritance and polymorphism). Java made sense over C++ because of its portable support for multithreading and user interfaces. And, perhaps most importantly, switching languages periodically made sense because it forced the team to redesign the system, leveraging lessons learned.

This text is based heavily on the Java-based Ptolemy II. Nevertheless, a great deal of credit goes to the designers of Ptolemy Classic (Buck et al., 1994), most particularly Joseph Buck, Soonhoi Ha, Edward A. Lee, and David Messerschmitt.

People

Many people have contributed to this book and to the Ptolemy II software used to illustrate the concepts in the book. In addition to the authors of individual chapters, people who had major influence on the book include Shuvra S. Bhattacharyya, David Broman, Adam

Cataldo, Chihhong Patrick Cheng, Daniel Lazaro Cuadrado, Johan Eker, Brian L. Evans, Teale Fristoe, Chamberlain Fong, Shanna-Shaye Forbes, Edwin E. Goei, Jeff Jensen, Bart Kienhuis, Rowland R. Johnson, Soonhoi Ha, Asawaree Kalavade, Phil Lapsley, Bilung Lee, Seungjun Lee, Isaac Liu, Eleftherios D. Matsikoudis, Praveen K. Murthy, Hiren Patel, José Luis Pino, Jan Reineke, Sonia Sachs, Farhana Sheikh, Sun-Inn Shih, Gilbert C. Sih, Sean Simmons, S. Sriram, Richard S. Stevens, Juergen Teich, Neil E. Turner, Jeffrey C. Tsay, Brian K. Vogel, Kennard D. White, Martin Wiggers, Michael C. Williamson, Michael Wirthlin, Zoltan Kemenczy, Ye (Rachel) Zhou, and Jia Zou. Christopher Brooks has had overall responsibility for the software, and more than anyone is responsible for its high quality. Other contributors are given at `http://ptolemy.org/people`.

We also extend particular thanks to Jennifer White, who provided extensive editorial help, improving the the writing and clarity throughout the book. Others providing significant editorial help include Yishai Feldman, William Lucas, Aviral Shrivastava, Ben Zhang, and Michael Zimmer.

Funding

The work described in this text has received funding from a variety of sources over many years. In particular, the National Science Foundation (NSF) supported much of the work described in this book under the following projects:

- CCR-0225610: ITR: *Foundations of Hybrid and Embedded Systems and Software*
- CNS-0931843: Large: CPS: *ActionWebs*
- CNS-1035672: CPS: Medium: *Timing-Centric Software*
- CNS-0647591: CSR-SGER: Cyber-Physical Systems — *Are Computing Foundations Adequate?*

In addition, key pieces were sponsored by:

- The Army Research Laboratory (ARL), Cooperative Agreement Number W911NF-11-2-0038: *Disciplined Design of Systems of Systems*, as part of the Office of Secretary of Defense (OSD) Software-Intensive Systems Producibility Initiative.
- The Army Research Laboratory (ARL), Cooperative Agreement Number W911NF-07-2-0019: *Scalable Composition of Subsystems*, as part of the Office of Secretary of Defense (OSD) Software-Intensive Systems Producibility Initiative.

- The Naval Research Labs (NRL), Cooperative Agreement Number N00173-12-1-G015: *Software Producibility for Systems of Systems*, as part of the Office of Secretary of Defense (OSD) Software-Intensive Systems Producibility Initiative.
- The Air Force Research Lab (AFRL), Contract FA8750-11-C-0023: *Extensible Modeling and Analysis Framework*.
- The Air Force Office of Scientific Research (AFOSR) under the Multi-University Research Initiative (MURI) grant #FA9550-06-0312: *High-Confidence Design of Adaptive, Distributed Embedded Control Systems (HCDDES)*.

Additional funding was provided by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which over several years has received support from the following companies:

- Agilent
- Bosch
- Lockheed-Martin
- National Instruments
- Thales
- Toyota

Support for work on modeling of heterogeneous systems was also provided by the iCyPhy (industrial cyber-physical systems) research consortium, sponsored by IBM and United Technologies.

Key support for work described in Chapters 10 and 16 was provided by the Focus Center Research Program (FCRP), a Semiconductor Research Corporation (SRC) program funded by MARCO and DARPA, under the following projects:

- The *Multiscale Systems Center* (MuSyC).
- The *TerraSwarm Research Center*, one of six centers supported by the STARnet phase of the FCRP.

The views and conclusions contained in this text are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsors. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Part I

Getting Starting

This part of this book introduces system design, modeling, and simulation. First, Chapter 1 outlines the guiding principles for disciplined modeling of heterogeneous systems. It gives a high-level overview of the models of computation that are described in detail in Part II. It also provides a highly simplified case study (an electric power system) that illustrates the roles that various models of computation play in complex system design.

Chapter 2 provides a tutorial on using Ptolemy II through its graphical user interface, Vergil. One of the goals of this book is to enable the reader to exploit the open-source Ptolemy II system to conduct experiments in system design. This chapter is intended to provide enough information to make the reader a competent user of Ptolemy II. To extend Ptolemy II, the reader is referred to Part III.

1

Heterogeneous Modeling

Edward A. Lee

Contents

Many of today's engineered systems combine heterogeneous and often complex subsystems. A modern car, for example, may combine a complex engine, electronic control units (**ECUs**), traction control systems, body electronics (for controlling windows and door locks), entertainment systems, climate control and ventilation, and a variety of safety subsystems (such as airbags). Each subsystem may be realized with a combination of software, electronics, and mechanical parts. Engineering such complex systems is quite challenging, in part because even the smallest subsystems span multiple engineering disciplines.

These complex systems also challenge the design tools that engineers use to specify, design, simulate, and analyze systems. It is no longer sufficient to sketch a mechanical structure and write down a few equations describing the interactions of the mechanical parts. Neither is it sufficient to rely entirely on software tools for 3D modeling of mechanical parts or tools for model-based design of software systems. The complex interplay across domains (mechanics, software, electronics, communication networks, chemistry, fluid dynamics, and human factors) reduce the usefulness of tools that address only a single domain.

The focus of this book is on **cyber-physical systems** (**CPS**) (Lee, 2008a, 2010a; Lee and Seshia, 2011), which combine computing and networking with physical dynamics. Cyber-physical systems require model combinations that integrate the continuous dynamics of physical processes (often described using differential equations) with models of software. Diverse models are most useful in applications where timed interactions between components are combined with conventional algorithmic computations.[1] They can also be used in traditional software systems that have concurrent[2] interactions between algorithmic components.

[1] An **algorithm** is a finite description of a sequence of steps to be taken to solve a problem. Physical processes are rarely structured as a sequence of steps; rather, they are structured as continuous interactions between concurrent components.

[2] **Concurrency**, from the Latin verb *concurrere* meaning "run together," is often taken in computer science to mean the arbitrary interleaving of two or more sequences of steps. However, this is a rather specialized interpretation of a basic concept. In this book, we take concurrency to mean simultaneous operation, with no implication of either interleaving nor sequences of steps. In particular, two continuous processes can operate concurrently without being directly representable as sequences of steps. Consider, for example, a resistive heating element immersed in a vat of water. Increasing the current through the heating element will cause the temperature of the water to rise. The electrical flow is one continuous process, as is the temperature of the water, and these processes are interacting. But neither process is reasonably representable as a sequence of steps, nor is the overall process an interleaving of such steps.

Sidebar: About the Term "Cyber-Physical Systems"

The term "cyber-physical systems" emerged around 2006, when it was coined by Helen Gill at the National Science Foundation in the United States. We are all familiar with the term "**cyberspace**," attributed William Gibson, who used the term in the novel *Neuromancer* to refer to the medium of computer networks used for communication between humans. We may be tempted to associate the term cyberspace with CPS, but the roots of the term CPS are older and deeper. It would be more accurate to view the terms "cyberspace" and "cyber-physical systems" as stemming from the same root, "**cybernetics**," rather than viewing one as being derived from the other.

The term "cybernetics" was coined by Norbert Wiener (Wiener, 1948), an American mathematician who had a huge impact on the development of control systems theory. During World War II, Wiener pioneered technology for the automatic aiming and firing of anti-aircraft guns. Although the mechanisms he used did not involve digital computers, the principles involved are similar to those used today in a huge variety of computer-based feedback control systems. Wiener derived the term from the Greek $\kappa\upsilon\beta\varepsilon\rho\nu\eta\tau\eta\varsigma$ (kybernetes), meaning helmsman, governor, pilot, or rudder. The metaphor is apt for control systems.

Wiener described his vision of cybernetics as the conjunction of control and communication. His notion of control was deeply rooted in closed-loop feedback, where the control logic is driven by measurements of physical processes, and in turn drives the physical processes. Even though Wiener did not use digital computers, the control logic is effectively a computation, and therefore cybernetics is the conjunction of physical processes, computation, and communication.

Wiener could not have anticipated the powerful effects of digital computation and networks. The fact that the term "cyber-physical systems" may be ambiguously interpreted as the conjunction of cyberspace with physical processes, therefore, helps to underscore the enormous impact that CPS will have. CPS leverages a phenomenal information technology that far outstrips even the wildest dreams of Wiener's era.

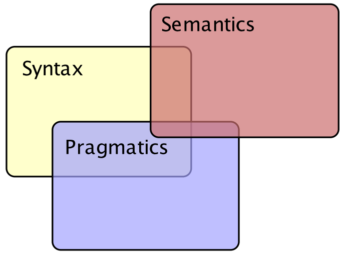


Figure 1.1: Today's design tools involve complex combinations of syntax, semantics, and pragmatics.

1.1 Syntax, Semantics, and Pragmatics

At the time of this writing, we are in the midst of a dramatic transformation in engineering tools and techniques, which are evolving to enable us to adapt to increasing system complexity and heterogeneity. In the past, entire industries were built around providing design tools for a single engineering domain, such as digital circuits, software, 3D mechanical design, and heating and ventilation systems. Today we see a growing consolidation and combination of design tools; individual tools often expand into tool suites and provide capabilities outside of their traditional domain. This evolution often entails significant growing pains, where poorly integrated capabilities yield frustratingly unexpected behaviors. Tool integration often results in "frankenware,"[3] brittle combinations of mostly incompatible tools that are extremely difficult to maintain and use effectively in combination.

Moreover, tools that have traditionally worked well within a relatively narrow domain are not as effective when used in broader domains. Today's sophisticated design tools involve complex combinations of **syntax** (how a design is represented), **semantics** (what a design means and how it works), and **pragmatics** (Fuhrmann and von Hanxleden, 2008) (how an engineer visualizes, edits, and analyzes a design). When tools are used in domains for which they were not originally designed or in combination with other tools, awkwardness may arise from incompatible syntaxes, poorly understood semantics, and inconsistent human interfaces.

[3]The term "frankenware" is due to Christopher Brooks.

Incompatibilities in syntax may arise because the structure of designs is intrinsically different (software syntax has very little in common with 3D volumes, for example). But all too often, it arises because the tools were developed in different engineering communities using different techniques. Similarly, the pragmatics of tools, such as how design files are managed and how changes are tracked, are often starkly different by historical accident. Differences in semantics are often accidental as well, sometimes arising from simple misunderstandings. Semantics may not be intuitively obvious in a different domain. A block diagram, for example, may mean something completely different to a control engineer as to a software engineer.

This book examines key concepts in heterogeneous modeling using Ptolemy II, an open-source modeling and simulation tool.[4] In contrast to most other design tools, Ptolemy II was developed from the outset to address heterogeneous systems. A key goal of the Ptolemy Project (an ongoing research effort at UC Berkeley) has been to minimize the accidental differences in syntax, semantics, and pragmatics between domains, and maximize the interoperability of designs expressed in different domains. As a consequence, Ptolemy II provides a useful laboratory for experimenting with design technologies for cyber-physical systems.

Ptolemy II integrates four distinct classes of syntaxes: block diagrams, bubble-and-arc diagrams, imperative programs, and arithmetic expressions. These syntaxes are complementary, and enable Ptolemy to address a variety of design domains. Block diagrams are used to express concurrent compositions of communicating components; bubble-and-arc diagrams are used to express sequencing of states or modes; imperative programs are used to express algorithms; and arithmetic expressions are used to express functional numeric computations.

Ptolemy II also integrates a number of semantic domains. For block diagrams, in particular, there are many distinct semantics possible. Connections between blocks represent interactions between components in a design, but what type of interaction? Is it an asynchronous message (like sending a letter)? Is it a rendezvous communication (like making a phone call)? Is it a clocked update of data (as in a synchronous digital circuit)? Does time play a role in the interaction? Is the interaction discrete or continuous? To enable heterogeneous modeling, Ptolemy II has been designed to support all of them, and is extensible to support more.

[4] Ptolemy II is available for download at `http://ptolemy.org`.

1.2 Domains and Models of Computation

A **semantic domain** in Ptolemy II, often just called a **domain**, defines the "laws of physics" for the interaction between components in a design. It provides the rules that govern concurrent execution of the components and the communication between components (such as those described above). A collection of such rules is called a **model of computation** (**MoC**). We will use the terms "model of computation" and "domain" (nearly) interchangeably, though technically we think of a domain as being an *implementation* of a MoC. The MoC is an abstract model, whereas the domain is its concrete implementation in software.

The rules that constitute a model of computation fall into three categories. The first set of rules specifies what constitutes a component. In this book, a component is generally an actor, to be defined more precisely below. The second set of rules specifies the execution and concurrency mechanisms. Are actors invoked in order? Simultaneously? Nondeterministically? The third specifies the communication mechanisms. How do actors exchange data?

Each of the MoCs discussed in this book has many possible variants, many of which have been realized in other modeling tools. In this book, we focus only on MoCs that have been realized in Ptolemy II and that have well understood and documented semantics.[5] For further context, we also provide brief descriptions and pointers to other useful MoCs that have not been realized in Ptolemy II, but have been realized in other tools.

To support the design of heterogeneous systems, Ptolemy II domains (and models of computation) interoperate with one another. This requires a level of agreement between semantic domains that is rare when tools are developed separately and then later integrated. The principles behind interoperation of domains in Ptolemy II are described in a number of papers (Eker et al., 2003; Lee et al., 2003; Goderis et al., 2009; Lee, 2010b; Tripakis et al., 2013). In this book, we focus on the practical aspects of domain interoperability, not on the theory.

Using a single, coherent software system lets us focus on domain interoperation rather than on less important incompatibilities that typically arise in tool integration. For, example, the Ptolemy II type system (which defines the types of data that can be used with various computational components) is shared by all domains, by the state machine notation,

[5]In the electronic version of this book, most illustrations of models provide a hyperlink in the caption that enables you to browse the model online. If you are reading the book on a Java-capable machine, then you can edit and execute the models shown in most of the figures.

and by the expression language. The domains are all capable of inferring and verifying appropriate data types; this functionality works seamlessly across heterogeneous models with multiple domains. Similarly, domains that include a notion of time in their semantics share a common representation of time and a (multiform) model of time. Finally, the same graphical editor spans domains, and the same XML schema is used to store design data. These agreements remove many of the practical obstacles to heterogeneous composition of models. They allow us to focus on the benefits of heterogeneous integration – most importantly, the ability to choose the domain that best matches the problem, even when the design is heterogeneous.

1.3 The Role of Models in Design

This book provides a framework for understanding and building models in Ptolemy II and, more broadly, for understanding key issues in modeling and simulating complex heterogeneous systems. This topic is broad enough that no single volume could possibly cover all of the techniques that could be useful to system designers. In this book, we focus on models that describe **dynamics**, or how a system or subsystem evolves in time. We do not cover techniques that focus primarily on the static structure of designs (such as UML class diagrams for software or 3D volumetric modeling). As a consequence, all of the models in this book are executable. We call the execution of a model a **simulation**.

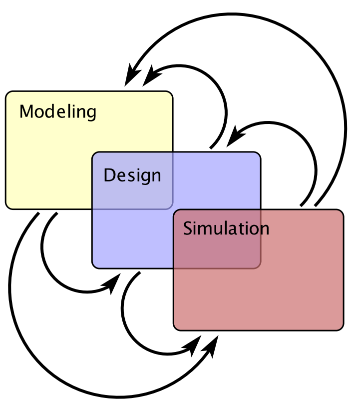


Figure 1.2: Iterative process of modeling, design, and simulation.

Figure 1.2 shows three major parts of the process of implementing systems: modeling, design, and simulation. **Modeling** is the process of gaining a deeper understanding of a system through imitation. Models imitate the system and reflect properties of the system. Models of dynamics specify *what* a system does; that is, how it reacts to stimulus from its environment, and how it evolves over time. **Design** is the structured creation of artifacts (such as software components) to implement specific functionality. It specifies *how* a system will accomplish the desired functionality. **Simulation** shows how models behave in a particular environment. Simulation is a simple form of design analysis; its goal is to lend insight into the properties of the design and to enable **testing** of the design. The models we discuss in this book can also be subjected to much more elaborate forms of analysis, including formal verification. In its most general form, **analysis** is the process of gaining a deeper understanding of a system through dissection, or partitioning into smaller, more readily analyzed pieces. It specifies *why* a system does what it does (or fails to do) what a model says it should do. We leave all analysis techniques except simulation to other texts.

As suggested in Figure 1.2, the three parts of the design process overlap, and the process iterates between them. Normally, the design process begins with modeling, where the goal is to understand the problem and to develop solution strategies.

Modeling plays a central role in modern design processes. The key principle of **model-based design** is to maximally leverage modeling to construct better designs. To be effective, models must be reasonably faithful, of course, but they also must be understandable and analyzable. To be understandable and analyzable, a model needs to have a clear meaning (a clear semantics).

Models are expressed in some **modeling language**. For example, a procedure may be expressed in Java or in C, so these programming languages are in fact modeling languages for procedures. A modeling language has a **strong semantics** if models expressed in the language have a clear and unambiguous meaning. Java, for example, has a stronger semantics than C, as illustrated by the following example.

Example 1.1: Suppose that the arguments to a procedure are of type *int*. In Java, this data type is well defined, but not in C. In C, *int* may represent a 16-bit integer or a 32-bit integer, for example. The behavior of the procedure may be quite different depending on which implementation is provided. Particularly, overflow occurs more easily with 16-bit integers than with 32-bit integers.

Sidebar: Models vs. Realizations of Systems

Models must be used with caution. The **Kopetz principle** (named after Prof. Dr. Hermann Kopetz of TU-Vienna, who taught us this principle), paraphrased, is: *Many properties that we assert about systems (determinism, timeliness, reliability) are in fact not properties of an implemented system, but rather properties of a model of the system.*

Golomb (1971) emphasizes understanding the distinction between a model and thing being modeled, famously stating "you will never strike oil by drilling through the map!" In no way, however, does this diminish the value of a map! Consider **determinism**. A model is **determinate** if it produces a uniquely defined output for each particular input. It is **nondeterminate** if there are multiple possible outputs for any particular input. Although this seems like a simple definition, there are many subtleties. What do we mean by a "particular input?" Does the time at which the input arrives matter? What do we mean by a "uniquely defined output?" Should we consider how the system behaves when its implementation hardware fails?

Any statement about the determinism of a physical "implemented" system is fundamentally a religious or philosophical assertion, not a scientific one. We may assert that no real physical system is determinate. How will it behave when it is crushed, for example? Or we may conversely assert that everything in the physical world is preordained, a concept that we find farfetched, difficult to refute, and not very useful.

For models, however, we can make definitive assertions about their determinism. For example, a procedure defined in a programming language may be determinate in that the returned value of the procedure depends only on the arguments. No actual realization of the procedure is actually determinate in an absolute sense (the hardware may fail and no returned value will be produced at all). The procedure is a **model** defined within a **formal framework** (the semantics of the language). It models the execution of a machine abstractly, omitting information. The time at which the inputs are provided makes no difference to the model, so time is not part of what we mean by a "particular input." The inputs and outputs are just data, and the procedure defines the relationship between the inputs and outputs. This point about models is supported by Box and Draper (1987), who state "Essentially, all models are wrong, but some are useful." The usefulness of a model depends on the model **fidelity**, the degree to which a model accurately imitates the system being modeled. But models are *always* an approximation.

Many popular modeling languages based on block diagrams have quite **weak semantics**. It is common, for example, for modeling languages to adopt a block diagram notation without giving precise meaning to the lines drawn between blocks; they vaguely represent the fact that components interact. (For examples, see the sidebar on page 14.) Modeling languages with weak semantics are harder to analyze. Their value lies instead in their ability to informally communicate design concepts among humans.

1.4 Actor Models

Ptolemy II is based on a class of models called **actor-oriented models**, or more simply, **actor models**. **Actors** are components that execute concurrently and share data with each other by sending messages via ports.

Example 1.2: Consider, for example, the Ptolemy model shown in Figure 1.3. This model shows three actors, each of which has one port. Actor A sends messages to actors B and C via its port (the Relation diamond indicates that the output from A goes to both B and C).

The sum of all of the messages communicated via a port is referred to as a **signal**. The **Director** block in the example specifies the domain (and hence the model of computation). Most of this book is devoted to explaining the various domains that have been realized in Ptolemy II.

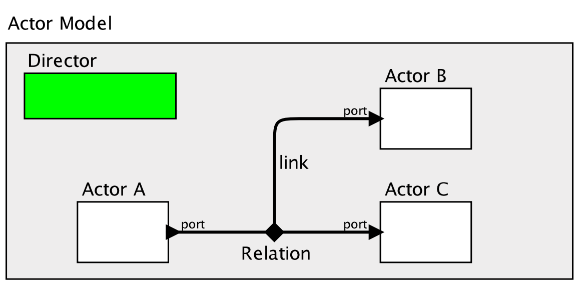


Figure 1.3: Visual rendition of a simple actor model.

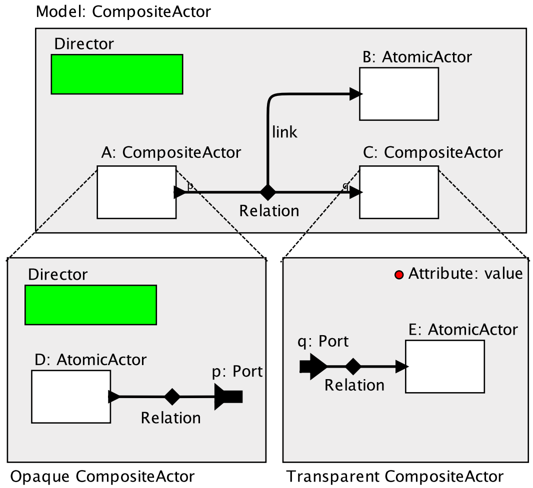


Figure 1.4: A hierarchical actor model consisting of a top-level composite actor and two submodels, each of which is also a composite actor.

1.5 Model Hierarchy

Models of complex systems are often complex. There is an art (the art of **model engineering**) to constructing good models of complex systems. A good model of a complex system provides relatively simple views of the system in order to faciliate easier understanding and analysis. A key approach to creating models with simplified views is to use modeling **hierarchy**, where what appears to be single component in one model is, internally, another model.

A hierarchical actor model is shown in Figure 1.4. It is an elaboration of Figure 1.3 where actors A and C are revealed to be themselves actor models. An **atomic actor** (where **atomic** comes from the ancient Greek **atomos**, meaning indivisible), is one that is not internally defined as an actor model. A **composite actor**, in contrast, is itself a composition of other actors. The ports p and q in the figure bridge the levels of hierarchy. A communication from D, for example, will eventually arrive at actor E after traversing ports and levels of the hierarchy.

1.6 Approaches to Heterogeneous Modeling

There are many approaches to heterogeneous modeling (Brooks et al., 2008). In **multi-view modeling**, distinct and separate models of the same system are constructed to model different aspects of a system. For example, one model may describe dynamic behavior, while another describes physical design and packaging. In **amorphous heterogeneity**, distinct modeling styles are combined in arbitrary ways within the same model without the benefit of structure. For example, some component interactions in a model may use rendezvous messaging (where both a sender and a receiver must be ready before a communication can occur), while others use asynchronous message passing (where the receiver receives the communication at some indeterminate time after the sender sends it). In **hierarchical multimodeling**, hierarchical compositions of distinct modeling styles are combined to take advantage of the unique capabilities and expressiveness of each style.

Sidebar: About the Term "Actors"

Our notion of actor-oriented modeling is related to the term "actor" as introduced in the 1970's by Hewitt to describe the concept of autonomous reasoning agents (Hewitt, 1977). The term evolved through the work of Agha and others to describe a formalized model of concurrency (Agha et al., 1997). Agha's actors each have an independent thread of control and communicate via asynchronous message passing. The term "actor" was also used in Dennis's dataflow models (Dennis, 1974) of discrete atomic computations that react to the availability of inputs by producing outputs sent to other actors.

In this book, the term "actor" embraces a larger family of models of concurrency. They are often more constrained than general message passing and do not necessarily conform with a dataflow semantics. Our actors are still conceptually concurrent, but unlike Agha's actors, they need not have their own thread of control. Unlike Dennis' actors, they need not be triggered by input data. Moreover, although communication is still achieved through some form of message passing, it need not be asynchronous.

Actors are *components* in systems and can be compared to **objects**, software components in object-oriented design. In prevailing object-oriented languages (such as Java, C++, and C#), the interfaces to objects are primarily **methods**, which are procedures that modify or observe the state of objects. By contrast, the actor interfaces are primarily **ports**, which send and receive data. They do not imply the same sequential transfer of control that procedures do, and hence they are better suited to concurrent models.

Sidebar: Actors in UML, SysML, and MARTE

The **Object Management Group** (**OMG**) has standardized a number of notations that relate strongly to the block diagram syntax common in actor models. The actor models in this book relate to **composite structure diagrams** of UML 2 (the second version of the **unified modeling language**) (Bock, 2006; Booch et al., 1998), or more directly its derivative **SysML** (Object Management Group (OMG), 2008a). The **internal block diagram** notation of SysML, particularly with the use of flow ports, is closely related to actor models. In SysML, the actors are called "blocks." (The term "actor" is used in UML for another purpose.)

SysML, however, emphasizes how model diagrams are rendered (their visual syntax), and leaves many details open about what the diagrams mean and how the models operate (their semantics). For example, although the SysML declares that "flow ports are intended to be used for asynchronous, broadcast, or send-and-forget interactions" (Object Management Group (OMG), 2008a), there is nothing like an MoC in SysML. Different SysML tools may give different behavior to flow ports and still be compliant with the standard. A single SysML model may represent multiple designs, and the behavior of the model may depend on the tools used to interpret the model. The emphasis of SysML is on standardizing the notation, not the meaning of the notation.

In contrast, the emphasis in Ptolemy II is on the semantics of models, rather than on how they are rendered (visually or otherwise). The visual notation is incidental, and in fact is not the only representation for a Ptolemy II model. Ptolemy II directors give models a very specific meaning. This concrete meaning ensures that a model means the same thing to different observers, and enables interoperation of heterogeneous models.

MARTE (modeling and analysis of real-time and embedded systems) puts more emphasis than SysML on the behavior of models (Object Management Group (OMG), 2008b). It avoids "constraining" the execution semantics, making the standard flexible, enabling representation of many prevalent real-time modeling techniques. In contrast, the emphasis in Ptolemy II is less on capturing existing design practices, and more on providing precise and well-defined models of system behavior. MARTE, interestingly, includes a multiform time model (André et al., 2007) not unlike that supported by Ptolemy II.

An example familiar to software engineers is Statecharts (Harel, 1987), which hierarchically combines synchronous concurrent composition with finite state machines. Another example of hierarchical modeling is **cosimulation**, where two distinct simulation tools are combined using a standardized interface such as the Simulink S function interface or the functional mockup interface (**FMI**) from the Modelica Association.[6] It is also possible to support heterogeneous modeling by creating very flexible or underspecified modeling frameworks that can be adapted to cover the models of interest. The downside of this approach is weak semantics. The goal of Ptolemy II is to achieve strong semantics, yet embrace heterogeneity and provide mechanisms for heterogeneous models to interact concurrently.

As shown in Figure 1.4, one can partition a complex model into a hierarchical tree of nested submodels. At each level, the submodels can be joined together to form a network of interacting actors. Ptolemy II constrains each level of the hierarchy to be locally homogeneous, using a common model of computation. These homogeneous networks can then be hierarchically combined to form a larger heterogeneous model. The advantage to this approach is that each part of the system can be modeled using the model of computation that provides the best match for its processing requirements — yet each model of computation provides strong semantics to ensure that it is relatively easy to understand, analyze, and execute.

In Ptolemy II, a director defines the semantics of a model. In Figure 1.4, there are two directors. The one at the top level defines the interaction between actors A, B, and C. Since C does not internally contain a director, the same top-level director governs the interactions with actor E. Actor C is called a **transparent composite actor**; its contained model is visible to its director.

In contrast, actor A internally contains another director. That inside director governs the interaction of actors within the sub model (in this simple example, there is only one such actor, but there could be more). Actor A is called an **opaque composite actor**, and its contents are hidden from A's outside director. To distinguish the two directors, we call the outside director the **executive director**. To the executive director, actor A looks just like an atomic actor. But internally, A contains another model.

The directors at different levels of the hierarchy need not implement the same MoC. Opaque composite actors, therefore, are Ptolemy's way of realizing hierarchical multimodeling and cosimulation.

[6] `http://www.functional-mockup-interface.org`

Sidebar: Plurality of Models

Occam's razor is a principle in science and engineering that encourages selection of those theories and hypotheses that require the fewest assumptions, postulates, or entities to explain a given phenomenon. The principle can be expressed as "entities must not be multiplied beyond necessity" (*entia non sunt multiplicanda praeter necessitatem*) or as "plurality should not be posited without necessity" (*pluralitas non est ponenda sine necessitate*) (Encyclopedia Britannica, 2010). The principle is attributed to 14th-century English logician, theologian and Franciscan friar William of Ockham.

Despite its compelling value, the principle has limitations. Immanuel Kant, for example, felt a need to moderate the effects of Occam's razor, stating "the variety of beings should not rashly be diminished." (*entium varietates non temere esse minuendas*) (Smith, 1929). Einstein allegedly remarked, "everything should be made as simple as possible, but not simpler" (Shapiro, 2006).

When applied to design techniques, Occam's razor biases us towards using fewer and simpler design languages and notations. However, experience indicates that both redundancy and diversity can be beneficial. For example, there is benefit to using UML class diagrams even if the information they represent is already encoded in a C++ program. There is also value in UML use-case diagrams, which express concepts that are not encoded in the C++ program and are also not (directly) represented in the UML class diagram. The three representations serve different purposes, though they represent the same underlying process.

The fact that many different notations are used in UML and its derivatives runs counter to the principle in Occam's razor. Ironically, the unified modeling language (**UML**) originated in the 1990s to *reduce* the diversity of notations used to express object-oriented software architectures (Booch et al., 1998). So what is gained by this anti-razor?

Design of software systems is essentially a creative process; engineers create programs that did not previously exist. Occam's razor should be applied only cautiously to creative processes, because creativity often flourishes when there are multiple media with which to achieve the desired effect. UML facilitates the creative process by offering more abstract notations than C++ source code, and these notations encourage experimentation with design and communication of design ideas.

Sidebar: About Heterogeneous Models

Some authors use the term **multi-paradigm modeling** to describe approaches that mix models of computation (Mosterman and Vangheluwe, 2004). Ptolemy II focuses on techniques that combine actors with multi-paradigm modeling. An early systematic approach to such mixed models was realized in Ptolemy Classic (Buck et al., 1994), the predecessor to Ptolemy II (Eker et al., 2003). Influenced by the Ptolemy approach, SystemC is capable of realizing multiple MoCs (Patel and Shukla, 2004; Herrera and Villar, 2006). So are ModHel'X (Hardebolle and Boulanger, 2007) and ForSyDe (Jantsch, 2003; Sander and Jantsch, 2004).

Another approach supports mixing concurrency and communication mechanisms without the structural constraints of hierarchy (Goessler and Sangiovanni-Vincentelli, 2002; Basu et al., 2006). A number of other researchers have tackled the problem of heterogeneity in creative ways (Burch et al., 2001; Feredj et al., 2009).

It is also possible to use **tool integration**, where different modeling tools are combined either through interchange languages or through co-simulation (Liu et al., 1999; University of Pennsylvania MoBIES team, 2002; Gu et al., 2003; Karsai et al., 2005). This approach is challenging, however, and yields fragile tool chains. Many tools lack documentation on how and where they can be extended to enable cross-tool integration; implementing and maintaining integration requires considerable effort. Challenges include API incompatibilities, unstable or undocumented APIs, unclear semantics, syntactic incompatibilities, and unmaintainable code bases.Tool integration proves to be a painful way to accomplish heterogeneous design. A better approach is to focus on the semantics of interoperation, rather than on the software problems of tool integration. Good software architectures for interoperation will emerge only from a good understanding of the semantics of interoperation.

In Ptolemy, each model contains a director that specifies the MoC being used and provides either a code generator or an interpreter for the MoC (or both). An interesting alternative is given by "42" (Maraninchi and Bhouhadiba, 2007), which integrates a custom MoC with the model.

Sidebar: Tools Supporting Heterogeneous Models

Several widely used tools provide fixed combinations of a few MoCs. Commercial tools include Simulink/StateFlow (from The MathWorks), which combines continuous- and discrete-time actor models with finite-state machines, and LabVIEW (from National Instruments), which combines dataflow actor models with finite-state machines and a time-driven MoC. Statemate (Harel et al., 1990) and SCADE (Berry, 2003) combine finite-state machines with a synchronous/reactive formalism (Benveniste and Berry, 1991). Giotto (Henzinger et al., 2001) and TDL (Pree and Templ, 2006) combine FSMs with a time-driven MoC. Several hybrid system modeling and simulation tools combine continuous-time dynamical systems with FSMs (Carloni et al., 2006).

The Y-chart approach supports heterogeneous modeling and is popular for hardware-software codesign (Kienhuis et al., 2001). This approach separates modeling of the hardware implementation from modeling of application behavior (a form of multi-view modeling), and provides mechanisms for bringing these disparate models together. These mechanisms allow developers to trade off hardware cost and complexity with software design. Metropolis is a particularly elegant tool for this purpose (Goessler and Sangiovanni-Vincentelli, 2002). It introduces a "quantity manager" that mediates interactions between the desired functionality and the resources required to implement that functionality.

Modelica (Fritzson, 2003; Modelica Association, 2009) also has actor-like semantics in the sense that components are concurrent and communicate via ports, but the ports are neither inputs nor outputs. Instead, the connections between ports declare equation constraints on variables. This approach has significant advantages, particularly for specifying physical models based on differential-algebraic equations (**DAE**s). However, the approach also appears to be harder to combine heterogeneously with other MoCs.

DESTECS (design support and tooling for embedded control software) is a tool supported by a consortium from academia and industry that has a focus on fault-tolerant embedded systems (Fitzgerald et al., 2010). This tool integrates continuous-time models made in 20-sim (Broenink, 1997) and discrete-event models in VDM (Vienna Development Method) (Fitzgerald et al., 2008). DESTECS synchronizes time and passes variables between the two tools.

1.7 Models of Time

Some models of computation have a notion of **time**. Specifically, this means that communication between actors and computation performed by actors occurs on a logical time line. Even more specifically, this means that there is a notion of two actions (communication or computation) being either ordered in time (one occurs before the other) or being simultaneous. A notion of time may also have a metric, meaning (loosely) that the time gap between two actions may be measured.

A key mechanism that Ptolemy II provides for interoperability of domains is a coherent notion of time. This mechanism has proven effective even for combining models of computation that have no notion of time (such as dataflow models and finite state machines), with models of computation that depend strongly on time (such as discrete-event models and continuous-time models). In this section, we outline key features of this mechanism.

1.7.1 Hierarchical Time

The model hierarchy discussed in Sections 1.5 and 1.6 is central to the management of time. Typically, only the top-level director advances time. Other directors in a model obtain the current model time from their enclosing director. If the top-level director does not implement a timed model of computation, then time does not advance. Hence, timed models always contain a top-level director that implements a timed model of computation.

Timed and untimed models of computation may be interleaved in the hierarchy. As we will discuss later, however, there are certain combinations that do not make sense, while other combinations are particularly useful, particularly the modal models discussed in Chapter 8.

Time can also advance non-uniformly in a model. In the modal models of Chapter 8, the advancement of time can be temporarily suspended in a submodel (Lee and Tripakis, 2010). More generally, as explained in Chapter 10, time may also progress at different rates at different levels of the hierarchy. This feature is particularly useful for modeling distributed systems where maintaining a perfectly coherent uniform time base is not physically possible. It is referred to as **multiform time**, and it enables highly realistic models that explicitly recognize that time can only be imperfectly measured.

1.7.2 Superdense Time

In addition to providing multiform time, Ptolemy II provides a model of time known as **superdense time** (Manna and Pnueli, 1993; Maler et al., 1992; Lee and Zheng, 2005; Cataldo et al., 2006). A superdense time value is a pair (t, n), called a **time stamp**, where t is the **model time** and n is a **microstep** (also called an **index**). The model time represents the time at which some event occurs, and the microstep represents the sequencing of events that occur at the same model time. Two time stamps (t, n_1) and (t, n_2) can be interpreted as being **simultaneous** (in a weak sense) even if $n_1 \neq n_2$. A stronger notion of **simultaneity** would require the time stamps to be equal (both in model time and microstep). An example illustrates the value of superdense time.

Example 1.3: To understand the role of the microstep, consider Newton's cradle, a toy with five steel balls suspended by strings, shown in Figure 1.5. If you lift the first ball and release it, it strikes the second ball, which does not move. Instead, the fifth ball reacts by rising.

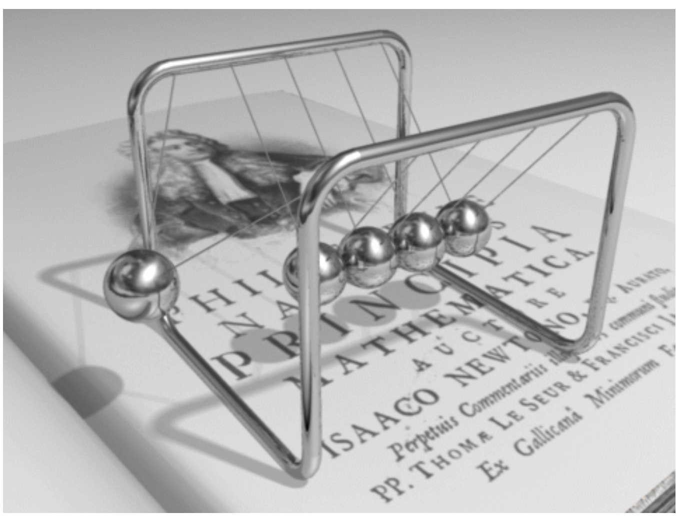


Figure 1.5: Newton's cradle. Image by Dominique Toussaint, made available under the terms of the GNU Free Documentation License, Version 1.2 or later.

Consider the momentum p of the second ball as a function of time. The second ball does not move, so its momentum must be everywhere zero. But the momentum of the first ball is somehow transferred to the fifth ball, passing through the second ball. So the momentum cannot be always zero.

Let $\mathbb{R}$ represent the real numbers. Let $p\colon \mathbb{R} \to \mathbb{R}$ be a function that represents the momentum of this second ball, and let τ be the time of the collision. Then

$$p(t) = \begin{cases} P & \text{if } t = \tau \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

for some constant P and for all $t \in \mathbb{R}$. Before and after the instant of time τ, the momentum of the ball is zero, but at time τ, it is not zero. Momentum is proportional to velocity, so

$$p(t) = Mv(t),$$

where M is the mass of the ball. Hence, combining with (1.1),

$$v(t) = \begin{cases} P/M & \text{if } t = \tau \\ 0 & \text{otherwise.} \end{cases} \tag{1.2}$$

The position of a mass is the integral of its velocity,

$$x(t) = x(0) + \int_0^t v(\tau)d\tau,$$

where $x(0)$ is the initial position. The integral of the function given by (1.2) is zero at all t, so the ball does not move, despite having a non-zero momentum at an instant.

The above physical model mostly works to describes the physics, but has two flaws. First, it violates the basic physical principle of conservation of momentum. At the time of the collision, all three middle balls will simultaneously have non-zero momentum, so seemingly, aggregate momentum has magically increased. Second, the model cannot be directly converted into a discrete representation.

A **discrete** representation of a signal is a sequence of values that are ordered in time (for mathematical details, see the sidebar on page 334). Any such representation of the momentum in (1.1) or velocity in (1.2) is ambiguous. If the sequence does not include the value at the time of the collision, then the representation does not capture the fact that momentum is transferred through the ball. If the representation does include the value at the time of the collision, then the representation is indistinguishable from a representation of a signal that has a non-zero momentum over some interval of time, and therefore models a ball that does move. In such a discrete representation, there is no semantic distinction between an instantaneous event and a rapidly varying continuous event.

Superdense time solves both problems. Specifically, the momentum of the second ball can be unambiguously represented by a sequence of samples where $p(\tau, 0) = 0$, $p(\tau, 1) = P$, and $p(\tau, 2) = 0$, where τ is the time of the collision. The third ball has non-zero momentum only at superdense time $(\tau, 2)$. At the time of the collision, each ball first has zero momentum, then non-zero, then zero again, all in an instant. The event of having non-zero momentum is weakly simultaneous for all three middle balls, but not strongly simultaneous. Momentum is conserved, and the model is unambiguously discrete.

One could argue that the physical system is not actually discrete. Even well-made steel balls will compress, so the collision is actually a continuous process, not a discrete event. This is true, but when building models, we do not want the modeling formalism to force us to construct models that are more detailed than is appropriate. Such a model of Newton's cradle would be far more sophisticated, and the resulting non-linear dynamics would be far more difficult to analyze. The fidelity of the model would improve, but at a steep price in understandability and analyzability.

The above example shows that physical processes that include instantaneous events are better modeled using functions of the form $p\colon \mathbb{R} \times \mathbb{N} \to \mathbb{R}$, where $\mathbb{N}$ represents the natural numbers, rather than the more conventional $p\colon \mathbb{R} \to \mathbb{R}$. The latter is adequate for continuous processes, but not for discrete events. At any time $t \in \mathbb{R}$, the signal p has a sequence of values, ordered by their microsteps. This signal cannot be misinterpreted as a rapidly varying continuous signal.

We say that two time stamps (t_1, n_1) and (t_2, n_2) are **weakly simultaneous** if $t_1 = t_2$, and **strongly simultaneous** if, in addition, $n_1 = n_2$.

Thus we can represent causally-related, but weakly simultaneous events. A signal may have two *distinct* events at with time stamps (t, n_1) and (t, n_2), where $n_1 \neq n_2$. A signal may therefore include weakly simultaneous, but distinct, events. Two distinct signals may contain strongly simultaneous events, but a single signal cannot contain two distinct strongly simultaneous events. This model of time unambiguously represents discrete events, discontinuities in continuous-time signals, and sequences of zero-time events in discrete signals.

Superdense time is ordered lexicographically (like a dictionary), which means that $(t_1, n_1) < (t_2, n_2)$ if either $t_1 < t_2$, or $t_1 = t_2$ and $n_1 < n_2$. Thus, an event is considered to occur before another if its model time is less or, if the model times are the same, if its microstep is lower. Time stamps are a particular realization of **tags** in the tagged-signal model of Lee and Sangiovanni-Vincentelli (1998).

1.7.3 Numeric Representation of Time

Computers cannot perfectly represent real numbers, so a time stamp of form $(t, n) \in \mathbb{R} \times \mathbb{N}$ is not realizable. Many software systems approximate a time t using a double-precision floating point number. But such a representation has two serious disadvantages. First, the **precision** of a number (how close it is to the next smaller or large representable number) depends on its magnitude. Thus, as time increases in such systems, the precision with which time is represented decreases. Second, addition and subtraction can introduce quantization errors in such a representation, so it is not necessarily true that $(t_1+t_2)+t_3 = t_1 + (t_2 + t_3)$. This significantly weakens the semantic notion of simultaneity, since whether two events are (weakly or strongly) simultaneous may depend on how their time stamps were computed.

Ptolemy II solves this problem by making the **time resolution** a single, global constant. Model time is given as $t = mr$, where m is an arbitrarily large integer, and the time resolution r is a double-precision floating point number. The multiple m is realized as a Java BigInteger (an arbitrarily large integer), so it will never overflow. The time resolution r, a *double*, is a parameter shared by all the directors in a model. A model, therefore, has the same time resolution throughout its hierarchy and throughout its execution, no matter how big time gets. Moreover, addition and subtraction of time values does not suffer quantization errors. By default, the time resolution is $r = 10^{-10}$, which may represent one tenth of a nanosecond. Then, for example, $m = 10^{11}$ represents 10 seconds.

In Ptolemy II, the microstep n in a time stamp (t, n) is represented as an *int*, a 32-bit integer. The microstep, therefore, is vulnerable to overflow. Such overflow may be prevented by avoiding models that have chattering Zeno behavior, as discussed in Chapter 7.

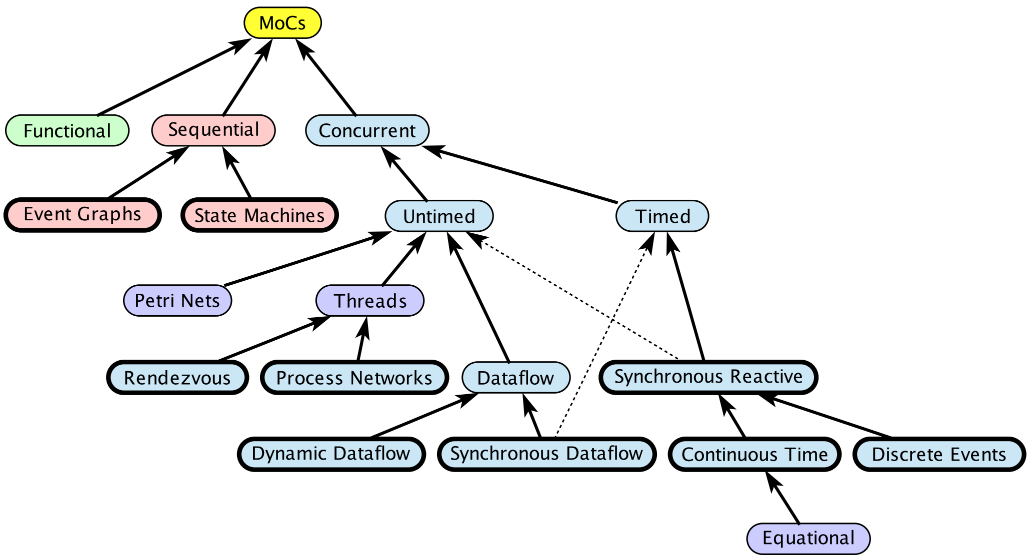


Figure 1.6: Summary of the relationship between models of computation. The ones with bold outlines are covered in detail in this Book.

1.8 Overview of Domains and Directors

In Ptolemy II an implementation of a model of computation is called a domain.[7] In this section, we briefly describe domains that have been realized in Ptolemy II. This is not a complete list; the intent is to show the diversity of the models of computation under study. These and other domains are described in subsequent chapters in more detail. Figure 1.6 summarizes the relationships between these domains.

All of the domains discussed here ensure determinism unless the model explicitly specifies nondeterministic behavior. That is, nondeterminism, if desired, must be explicitly built into the models; it does not arise accidentally from weak semantics in the modeling framework. A domain is said to be determinate if the signals sent between actors —

[7] The term "domain" comes from a fanciful notion in astrophysics, that there are regions of the universe with different sets of laws of physics. A model of computation represents the "laws of physics" of the submodel it governs.

including data values carried by messages, their order, and their time stamps — do not depend on arbitrary scheduling decisions, despite the concurrency in the model. Ensuring determinism is far from trivial in concurrent MoCs, and providing reasonable nondeterminate mechanisms is also challenging. The goal is that, when a model includes nondeterminate behavior, it should be explicitly specified by the builder of the model; it should not appear accidentally, nor should it surprise the user.

Dataflow. Ptolemy II includes several dataflow domains, described in Chapter 3. The execution of an actor in dataflow domains consists of a sequence of **firings**, where each firing occurs as a reaction to the availability of input data. A firing is a (typically small) computation that consumes the input data and produces output data.

The synchronous dataflow (SDF) domain (Lee and Messerschmitt, 1987b) is particularly simple, and is possibly the most used domain of all. When an actor is executed in SDF, it consumes a fixed amount of data from each input port, and produces a fixed amount of data to each output port. An advantage of the SDF domain is that (as described in Chapter 3) the potential for deadlock and boundedness can be statically checked, and schedules (including parallel schedules) can be statically computed. Communication in this domain is realized with **first-in, first-out** (**FIFO**) queues with fixed finite capacity, and the execution order of components is statically scheduled. SDF can be timed or untimed, though it is usually untimed, as suggested in Figure 1.6.

In contrast, the dynamic dataflow (DDF) domain is more flexible than SDF and computes schedules on the fly. In DDF, the capacity of the FIFO queues is not bounded. DDF is useful when communication patterns between actors are dependent on the data that is passed between actors.

Dataflow models are ideal for representing **streaming** systems, where sequences of data values flow in relatively regular patterns between components. Signal processing systems, such as audio and video systems, for example, are a particularly good match.

Process Networks. In the process network (PN) domain, described in Chapter 4, actors represent concurrent processes that communicate by (conceptually infinite capacity) FIFO queues (Lee and Parks, 1995). Writing to the queues always succeeds immediately, while reading from an empty queue blocks the reader process. The simple blocking-read, nonblocking-write strategy ensures the determinacy of the model (Kahn and MacQueen, 1977). Nevertheless, we have extended the model to support certain forms of nondeterminism. Each actor executes in its own Java thread, so on multicore machines they can execute in parallel. This domain is untimed. The PN domain realizes a generalization of dataflow where instead of discrete firings, actors represent continually executing processes (Lee and Matsikoudis, 2009).

PN is suitable for describing concurrent processes that communicate asynchronously by sending messages to one another. Messages are eventually delivered in the same order they are sent. Message delivery is presumed to be reliable, so the sender does not expect nor receive any confirmation. This domain has a "send and forget" flavor.

PN also provides a relatively easy way to get parallel execution of models. Each actor executes in its own thread, and most modern operating systems will automatically map threads onto available cores. Note that if the actors are relatively fine-grained, meaning that they perform little computation for each communication, then the overhead of multithreading and inter-thread communication may overwhelm the performance advantages of parallel execution. Thus, model builders should expect performance advantages only for coarse-grained models.

Rendezvous. The Rendezvous domain, also described in Chapter 4, is similar to PN in that actors represent concurrent processes. However, unlike PN's "send and forget" semantics, in the Rendezvous domain, actors communicate by atomic instantaneous data exchanges. When one actor sends data to another, the sender will block until the receiver is ready to receive. Similarly, when one actor attempts to read input data, it will block until the sender of the data is ready to send the data. As a consequence, the process that first reaches a rendezvous point will stall until the other process reaches the same rendezvous point (Hoare, 1978). It is also possible in this domain to create multi-way rendezvous, where several processes must all reach the rendezvous point before any process can continue. Like PN, this domain is untimed, supports explicit nondeterminism, and can transparently leverage multicore machines.

The Rendezvous domain is particularly useful for modeling asynchronous resource contention problems, where a single resource is shared by multiple asynchronous processes.

Synchronous-Reactive. The synchronous-reactive (SR) domain, described in Chapter 5, is based on the semantics of synchronous languages (Benveniste and Berry, 1991; Halbwachs et al., 1991; Edwards and Lee, 2003a). The principle behind synchronous languages is simple, although the consequences are profound. Execution follows "ticks" of a global "clock." At each tick, each variable (represented visually in Ptolemy II by the wires that connect the blocks) may or may not have a value. Its value (or absence of

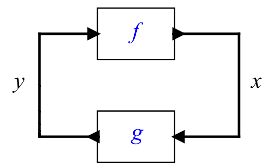


Figure 1.7: A simple feedback system.

value) is given by an actor whose output port is connected to the wire. The actor realizes a function that maps the values at its input ports to the values at its output ports (the function can vary from tick to tick). For example, in Figure 1.7, the variables x and y at a particular tick are related by

$$x = f(y), \text{ and } y = g(x).$$

The task of the domain's director is to find, at each tick, values of x and y that solve these equations. This solution is called a **fixed point**. The SR domain is by default untimed, but it can optionally be timed, in which case there is a fixed time interval between ticks.[8]

The SR domain is similar to dataflow and PN in that actors send streams of data to one other. Unlike dataflow, however, the streams are synchronized; at a tick of the clock, every communication path either has a message, or the message is unambiguously absent. Dataflow models, by contrast, are more asynchronous; a message may be "absent" simply because it hasn't arrived yet due to an accident of scheduling. To prevent nondeterminism, PN and dataflow have no semantic notion of an "absent" input. Inputs always have messages (or will have messages, in which case the actor is required to wait for the messages to arrive).

SR is well suited to situations with more complex control flow, where an actor may take different actions depending on whether a message is present or not. By synchronizing actions, the domain handles these scenarios without nondeterminism. SR is less concurrent than dataflow or PN, since each tick of the clock must be tightly orchestrated. As a consequence, it is harder to execute in parallel.

Finite-State Machine. The finite state machine (FSM) domain, described in Chapter 6, is the only domain discussed here that is not concurrent. The components in this domain

[8] If you need a variable time interval between ticks, you can accomplish this by placing an SR model within a DE model.

are not actors, but rather represent states, and the relations represent not communication paths, but rather transitions between states. Transitions have guards that determine when state transitions occur.

An FSM can be used to define the behavior of an actor used in any of the other domains. The actor can have any number of input and output ports. When that actor executes, the FSM reads the inputs, evaluates the guards to determine which transition to take, and produces outputs as specified on the selected transition. FSMs can also have local variables whose values can be modified by transitions (providing a model of computation that is known as an extended state machine).

An FSM can also be used to create a rich class of hierarchical models known as modal models, discussed in Chapter 8 (Lee and Tripakis, 2010). In a modal model, states of an FSM contain submodels that process inputs and produce outputs. Each state of the FSM represents a mode of execution, and the mode refinement defines the behavior in that mode. The mode refinement is a submodel with its own director that is active only when the FSM is in the corresponding state. When a submodel is not active, its local time does not advance, as explained above in Section 1.7.1.

Discrete Event. In the discrete-event (DE) domain, described in Chapter 7, actors communicate through events placed on a time line. Each event has a value and a time stamp, and actors process events in chronological order. The output events produced by an actor are required to be no earlier in time than the input events that were consumed. In other words, actors in DE are causal.

The execution of this model uses a global event queue. When an actor generates an output event, the event is slotted into the queue according to its time stamp. During each iteration of a DE model, the events with the smallest time stamp are removed from the global event queue, and their destination actor is fired. The DE domain supports simultaneous events. At each time where at least one actor fires, the director computes a fixed point, similar to SR (Lee and Zheng, 2007). DE is closely related to the well-known DEVS (discrete event system specification) formalism (Zeigler et al., 2000), which is widely used for simulating large, complex systems. The semantics of the Ptolemy II variant of DE is given by Lee (1999).

DE is well suited for modeling the behavior of complex systems over time. It can model networks, digital hardware, financial systems, and human organizational systems, for example. Chapter 10 shows how DE can be extended to leverage multiform time.

Continuous time. The Continuous time domain (Lee and Zheng, 2005), described in Chapter 9, models ordinary differential equations (ODEs), while also supporting discrete events. Special actors that represent *integrators* are connected in feedback loops in order to represent the ODEs. Each connection in this domain represents a continuous-time function, and the components denote the relations between these functions.

The Continuous model computes solutions to ODEs using numerical methods. As with SR and DE, at each instant, the director computes a fixed point for all signal values (Lee and Zheng, 2007). In each iteration, time is advanced by an amount determined by the ODE solver. To advance time, the director chooses a time stamp with the help of a solver and speculatively executes actors through this time step. If the time step is sufficiently small (key events such as level crossings, mode changes, or requested firing times are not skipped over, and the numerical integration is sufficiently accurate), then the director commits the time increment.

The Continuous director interoperates with all other timed Ptolemy II domains. Combining it with FSMs yields a particular form of modal model known as a hybrid system (Lee and Zheng, 2005; Lee, 2009). Combinations with discrete-event and synchronous/reactive domains are also useful (Lee and Zheng, 2007).

Ptera. The Ptera domain, described in Chapter 11, realizes a variant of event graphs. In Ptera, the components are not actors. Instead, the components are events, and the connections between components are triggering relations for events. A Ptera model represents how events in a system can trigger other events. Ptera is a timed model, and like FSM, it can be used to define the behavior of an actor to be used in another domain. In addition, events can be composites in that they have actions associated with them that are themselves defined by a submodel specified using another domain. Ptera is useful for specifying timed behaviors where input events may trigger chain reactions.

1.9 Case Study

Cyber-physical systems are intrinsically heterogeneous. CPS models, therefore, benefit from being able to combine models of computation. In this section, we walk through an example that uses several models of computation. The example is highly simplified, but with a little imagination, it is easy to see how the model can evolve to become an accurate and complete model of a large complex system. In particular, the large complex system we have in mind is an electric power system in a smart grid or on a vehicle (such as an airplane or advanced ground vehicle). In such a system, there are multiple sources of electric power (windmills, solar panels, turbines, backup generators, etc.) that must be coordinated to provide power to a multiplicity of loads. The system includes controllers that regulate the generators to keep voltages and frequencies near constant, and supervisory controllers that connect and disconnect loads and generators to provide services and handle faults to protect equipment. Such a system also includes networks whose dynamics may affect the overall behavior of the system.

Here, we illustrate a highly simplified version of such a system to show how the various MoCs come into play.

Example 1.4: A simplified model of a gas-powered generator that may be connected to and disconnected from a load is shown in Figure 1.8. This is a continuous-time model, as indicated by the Continuous director, which is discussed in Chapter 9. The model has two inputs, a *drive* signal, and a *loadAdmittance*. The output is a *voltage* signal. In addition, the model has three parameters, a time constant T, an output impedance Z, and a drive limit L. The model gives the output voltage of a generator over time as the generator gets more or less gas (specified by the *drive* input), and as the load varies (as specified by the *loadAdmittance* input).

This model exhibits simplified linear and nonlinear dynamics. The nonlinear dynamics is realized by the Limiter actor (see the sidebar on page 57), which limits

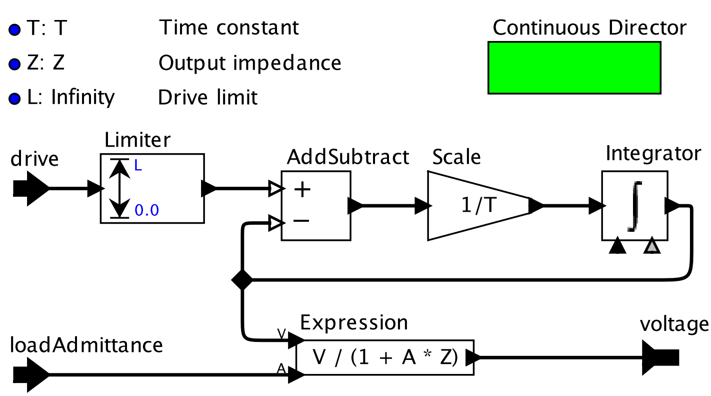


Figure 1.8: Simplified model of a gas-powered generator. [online]

the *drive* input. In particular, if the *drive* input becomes negative, it sets the drive to zero (you cannot extract gas from a generator). It also saturates the *drive* input at an upper bound given by the parameter L, which defaults to `Infinity`, meaning that there is no saturation (this generator can accept an arbitrarily large drive input).

The linear dynamics in this model is given by the small feedback loop, which includes an AddSubtract actor, a Scale actor, and an Integrator. If the output of the limiter is D, then this loop gives a value V that satisfies the following ordinary differential equation,

$$\frac{dV}{dt} = \frac{1}{T}(D - V)\,,$$

where both D and V are functions of time (see Chapter 9 to understand how this model yields the above equation).

For our purposes here, understanding this equation is not important, since this part of such a model would typically be constructed by a mechanical engineer who is an expert in such models, but we can nevertheless make some intuitive observations. First, if $D = V$, then the derivative is zero, so the generator is stable and will produce an unchanging output. Second, when $D \neq V$, the feedback loop adjusts the value of V to make it closer to D. If $D > V$, then this equation makes the derivative of V positive, which means that V will increase. If $D < V$, then the derivative is negative, so V will decrease. In fact, the output V will converge to D exponentially with time constant T. A **time constant** is the amount of time that an exponential signal takes to reach $1 - 1/e \approx 63.2\%$ of its final (asymptotic) value.

The last part of the model is the part that models the effect of the load. This effect is modeled by the Expression actor (see Section 13.2.4), which uses Ohm's law to calculate the output voltage as a function of the value V (representing the generator's effort), the output impedance Z, and the load admittance A. An electrical engineer would recognize this calculation as the realization of a simple voltage divider.

For our purposes, it is sufficient to notice that if $A = 0$ (there is no load) or $Z = 0$ (the generator is an ideal voltage source with no output impedance), then the voltage output is equal to the effort V. A real generator, however, will have a non-zero output impedance. As the load admittance A increases from zero, the output voltage will drop.

The above model is about the simplest interesting model of **continuous dynamics**. To integrate this model with digital controllers, we could wrap the model in another one that defines the discrete interfaces, as shown next.

Example 1.5: The Generator model of Figure 1.8 is wrapped to provide a discrete interface in Figure 1.9. Here, the *drive* and *loadAdmittance* inputs go to instances of the ZeroOrderHold actor. These inputs, therefore, can be provided as discrete events rather continuous-time signals. The ZeroOrderHold actor converts these discrete events into continuous-time signals by holding the value constant between arrivals of events (see Section 9.2).

The output voltage goes through a PeriodicSampler actor (see Section 9.2), which produces discrete events that are samples of the output voltage. The sample period is a parameter P of the model.

This model exposes the time constant T and output impedance Z, but hides the drive limit L. Of course, the model designer could make other choices about which parameters to expose.

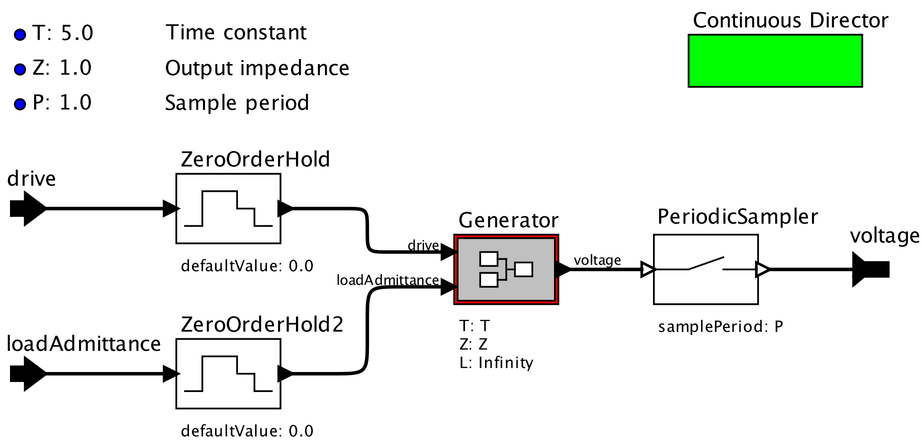


Figure 1.9: The Generator model of Figure 1.8 wrapped to provide a discrete interface. [online]

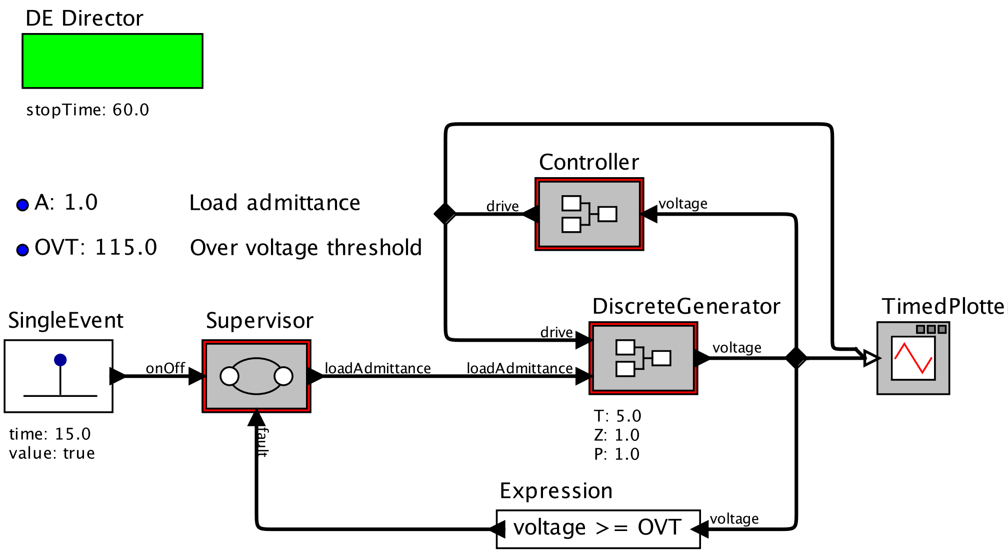


Figure 1.10: A discrete-event model with a generator, a controller, and an over-voltage protector. [online]

A continuous-time model may be embedded within a discrete-event model (see Chapter 7), as illustrated next.

Example 1.6: The DiscreteGenerator model of Figure 1.9 is embedded in a discrete-event model in Figure 1.10. This model has two parameters, the load admittance A and an over-voltage threshold OVT. The time constant T of the DiscreteGenerator is set to 5.0. This model includes two other components that we will explain below, a Supervisor, which provides the over-voltage protection, and a Controller, which regulates the *drive* input of the DiscreteGenerator based on measurements of the output voltage.

In addition, this model includes a simple test scenario, where a SingleEvent actor (see sidebar on page 241) requests that a load be connected at time 15.0, and a TimedPlotter actor (see Chapter 17), which displays the results of a run of the model, as shown in Figure 1.11.

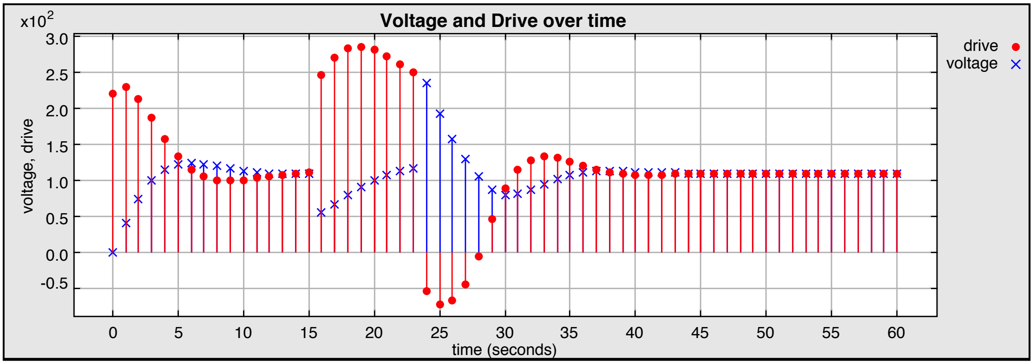


Figure 1.11: The plot produced by the model in Figure 1.10. [online]

In this test scenario, the load admittance is quite high (1.0) compared to the output impedance (also 1.0), so when the load is connected at time 15, the voltage abruptly drops to half its target value of 110 volts. The Controller compensates for this by substantially increasing the *drive*, but this causes the voltage to overshoot the target, and at time 24, to exceed the *OVT* threshold. The Supervisor reacts to this over-voltage condition by disconnecting the load, which causes the voltage to spike quite high, since the generator now has a substantial *drive* input. The Controller eventually brings the voltage back to the target level.

Notice further that when the load is disconnected, the Controller takes the *drive* signal negative. If this is a gas-powered generator, the Controller is trying to give the generator a negative flow of gas. Fortunately, our generator model includes a Limiter actor that prevents the model from actually providing that negative flow of gas.

The model in Figure 1.10 includes two very different kinds of controllers, a supervisory controller called Supervisor, and a low-level controller called simply Controller. These two controllers are specified using two additional MoCs, as explained next.

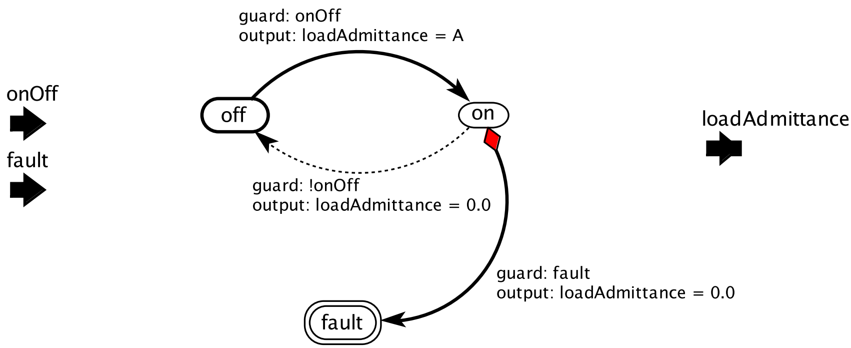


Figure 1.12: The Supervisor of Figure 1.10. [online]

Example 1.7: The Supervisor model of Figure 1.10 is a finite state machine, shown in Figure 1.12. The notation here is explained in Chapter 6, but we can easily grasp the general behavior.

This FSM has two inputs, *onOff* (a boolean that requests to connect or disconnect the load) and *fault* (a boolean that indicates that an over-voltage condition has occurred). It has one output, *loadAdmittance*, which will be the actually load admittance provided to the generator.

The initial state of the FSM is *off*. When an *onOff* input arrives that has value true, the FSM will transition from the *off* state to the *on* state and produce a *loadAdmittance* output with value given by *A*, a parameter of the model. This connects the load.

When the FSM is in state *on*, if a *fault* event arrives with value true, then it will transition to the final state *fault* and set the *loadAdmittance* to 0.0, disconnecting the load. If instead an *onOff* event arrives with value false, then it will transition to the state *off* and also disconnect the load. The difference between these two transitions is that once the FSM has entered the *fault* state, it cannot reconnect the load without a system reset (which will bring the FSM back to the initial state).

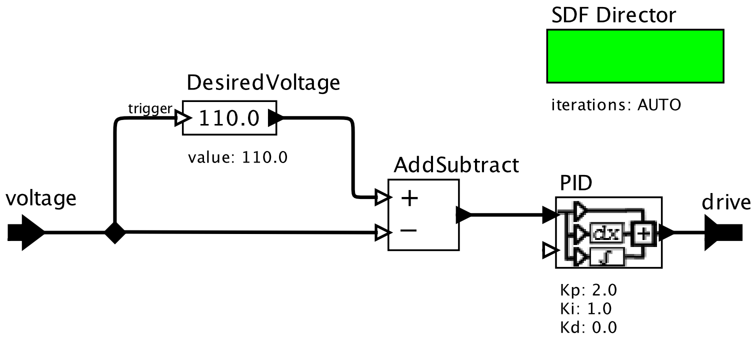


Figure 1.13: The Controller of Figure 1.10. [online]

Example 1.8: The Controller model of Figure 1.10 is the dataflow model shown in Figure 1.13. This model uses the SDF director (see Chapter 3), which is suitable for sampled-data signal processing. In this case, the controller compares the input *voltage* against a desired voltage (110 volts), and feeds the resulting error signal into a PID controller. A PID controller is a commonly used linear time-invariant system. A control engineer would know how to set the parameters of this controller, but in this case, we have simply chosen some parameters experimentally to yield an interesting test case.

Notice that the pieces of the model in Figure 1.10 are distinctly heterogeneous, touching on several disciplines within engineering and computer science. Typically, models of this type are the result of teams of engineers working together, and a framework that enables these teams to compose their models can become extremely valuable.

Many elaborations of this model are easy to envision. For example:

- The Generator could be defined as an actor-oriented class, so that it can be instantiated multiple times, and yet developed and maintained in a single centralized definition (see Section 2.6).
- The Generator model could be elaborated to reflect more sophisticated linear and non-linear dynamics using the techniques discussed in Chapter 9.

- The Generator model could be elaborated to include frequency and phase effects, for example by using complex-valued impedances and admittances together with a phasor representation.
- Models with a variable size (e.g., n generators and m loads, where n and m are parameters) could be created using the higher-order components considered in Section 2.7.
- The effects of network timing, clock synchronization, and contention for shared resources could be modeled using the techniques in Chapter 10.
- Signal processing techniques such as machine learning and spectral analysis, (see Chapter 3), could be integrated into the control algorithms.
- A units system could be included to make the model precise about the units used to measure time, voltage, frequency, etc.
- An ontology could be included to make the model precise about which signals and parameters represent voltages, admitances, impedances, etc., or even to make distinctions between domain-specific concepts such as the internal voltage (effort) of a generator vs. the voltage exhibited at its output, which is affected by its output impedance and load.

1.10 Summary

Ptolemy II focuses on actor-oriented modeling of complex systems, providing a disciplined approach to heterogeneity and concurrency. The central notion in hierarchical model decomposition is that of a domain, which implements a particular model of computation. Technically, a domain serves to separate the flow of control and data between components from the actual functionality of individual components. Besides facilitating hierarchical models, this separation can dramatically increase the reusability of components and models. The remainder of this book shows how to build Ptolemy II models and how to leverage the properties of each of the models of computation.

2

Building Graphical Models

Christopher Brooks, Edward A. Lee, Stephen Neuendorffer, and John Reekie

Contents

This chapter provides a tutorial on constructing Ptolemy II simulation models using **Vergil**, the Ptolemy graphical user interface (**GUI**). Figure 2.1 shows a simple Ptolemy II model in Vergil. This model is shown in the graph editor, one of several possible entry mechanisms available in Ptolemy II. It is also possible, for example, to define models in Java or XML.

2.1 Getting Started

Executing the examples in this chapter requires installation of Ptolemy II[1]. Once it is installed, you will need to invoke Vergil, which will display the initial welcome window shown in Figure 2.2. The "Tour of Ptolemy II" link takes you to the page shown in Figure 2.3.

2.1.1 Executing a Pre-Built Signal Processing Example

On the "Tour of Ptolemy II" page, the first example listed under "Basic Modeling Capabilities" (Spectrum), is the model shown in Figure 2.1. This model creates a sinusoidal signal, multiplies it by a sinusoidal carrier, adds noise, and then estimates the power spectrum. This model can be executed using the run button in the toolbar (the blue triangle

[1]See `http://ptolemy.org/ptolemyII/ptIIlatest` for the latest release, and `http://chess.eecs.berkeley.edu/ptexternal/` for access to the ongoing development version. Alternatively, most of the figures in this book have an online version of the model that you can browse using any web browser. More interestingly, if you are reading this book on a Java-capable machine, you can also follow a link that uses Java **Web Start** to launch Vergil without any explicit installation step. You can browse, edit, and execute models, and also save them to local disk. For information on Web Start, see `http://en.wikipedia.org/wiki/Java_Web_Start`.

pointing to the right). Two signal plots will then be displayed in their own windows, as shown in Figure 2.1. The plot on the right shows the power spectrum and the plot on the left shows the time-domain signal. Note the four peaks, which indicate the modulated sinusoid. You can adjust the frequencies of the signal and the carrier as well as the amount

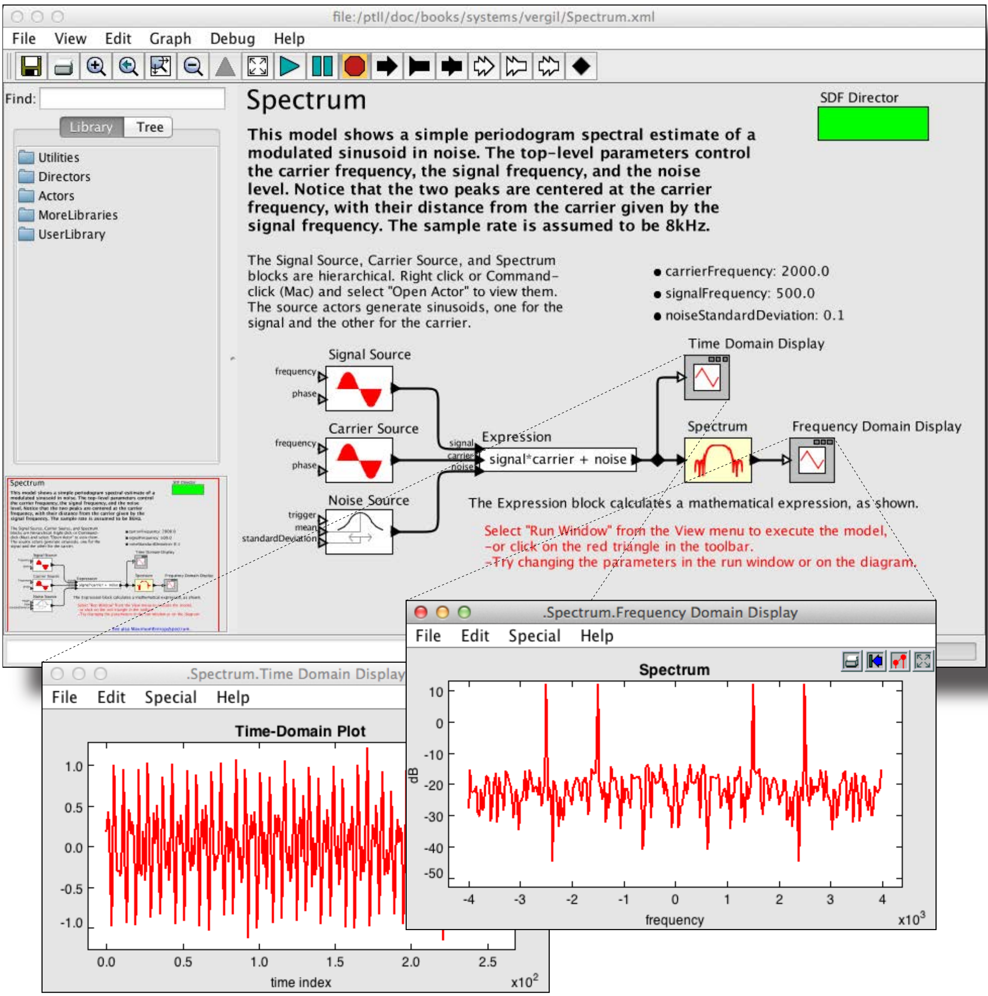


Figure 2.1: Example of a Vergil window and the windows that result from running the model. [online]

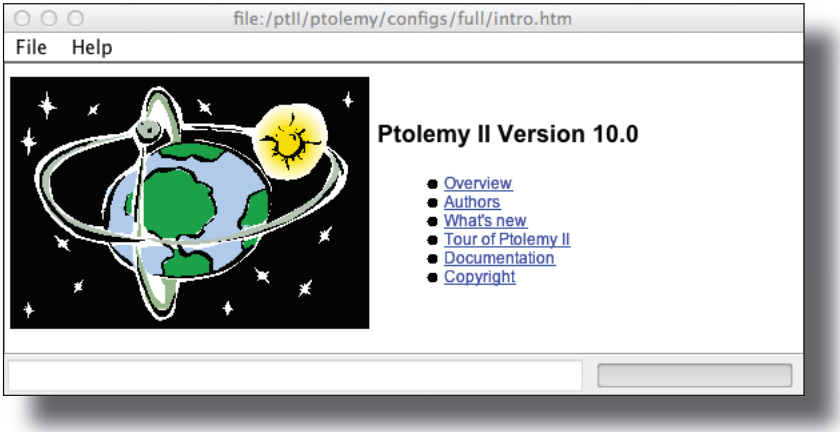


Figure 2.2: Initial welcome window.

of noise by double clicking on those parameters in the block diagram in Figure 2.1 near the upper right of the window.

The icon (the graphical block) for the Expression actor displays the expression that will be calculated:

```
signal*carrier + noise
```

The identifiers in this expression, `signal`, `carrier`, and `noise`, refer to the input ports. The Expression actor is flexible; it can have any number of input ports (which can have arbitrary names) and it uses a rich expression language to specify the value of the output as a function of the inputs. It can also specify parameters for the model in which the expression is contained. (The expression language is described in Chapter 13.)

Right clicking and selecting [`Documentation→Get Documentation`] displays documentation for that actor. Figure 2.4 shows the documentation for the Expression actor.

Three of the actors in Figure 2.1 are composite actors, meaning that their implementation is itself a Ptolemy II model. You can invoke the `Open Actor` context menu[2] to reveal the

[2]A **context menu** is a menu that is specific to the object under the cursor. It is obtained by right clicking the mouse (or control clicking, if the mouse does not have a right button) while the cursor is over the icon.

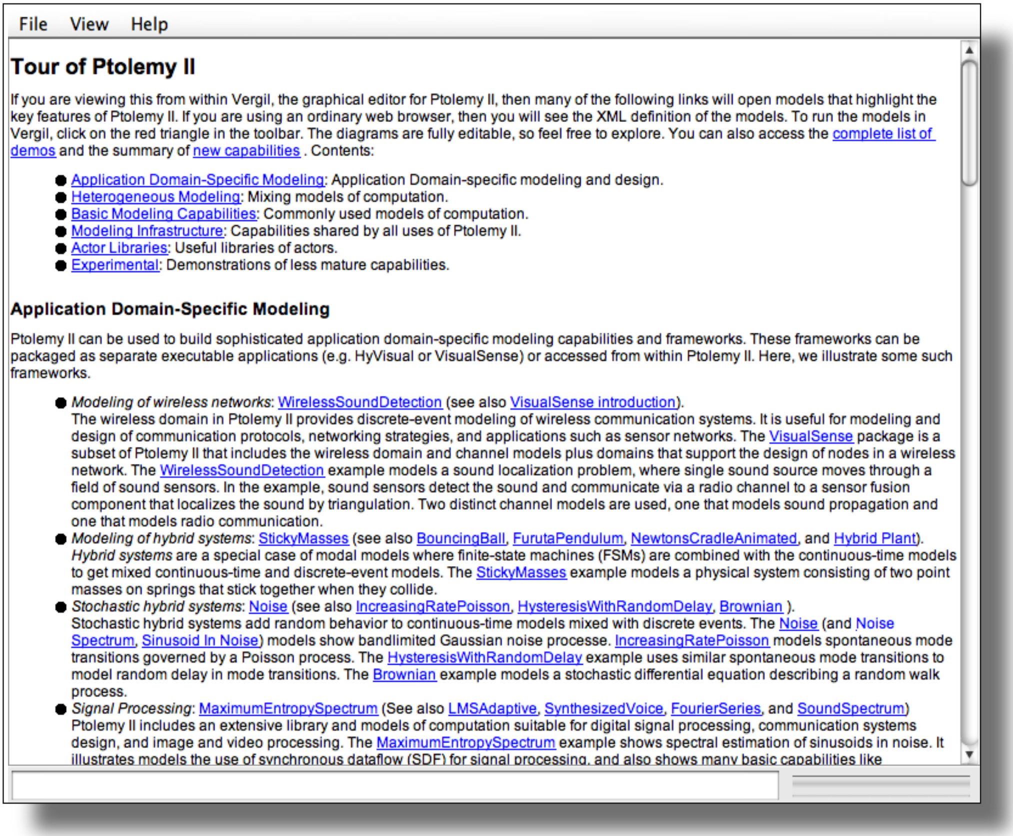


Figure 2.3: The tour of Ptolemy II page.

Signal Source implementation, as shown in Figure 2.5. This block diagram shows how the sinusoidal signal is generated.

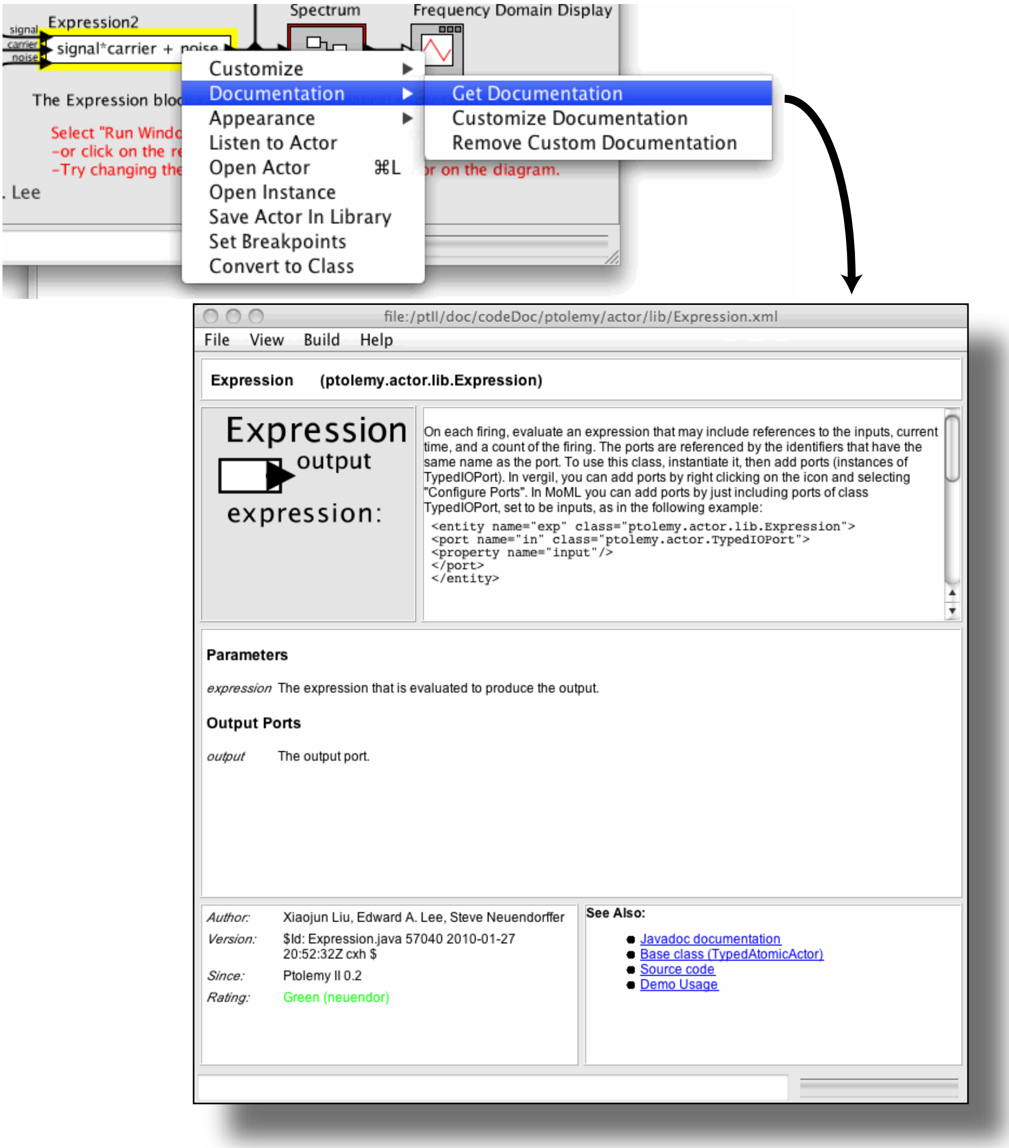


Figure 2.4: Viewing documentation for actors.

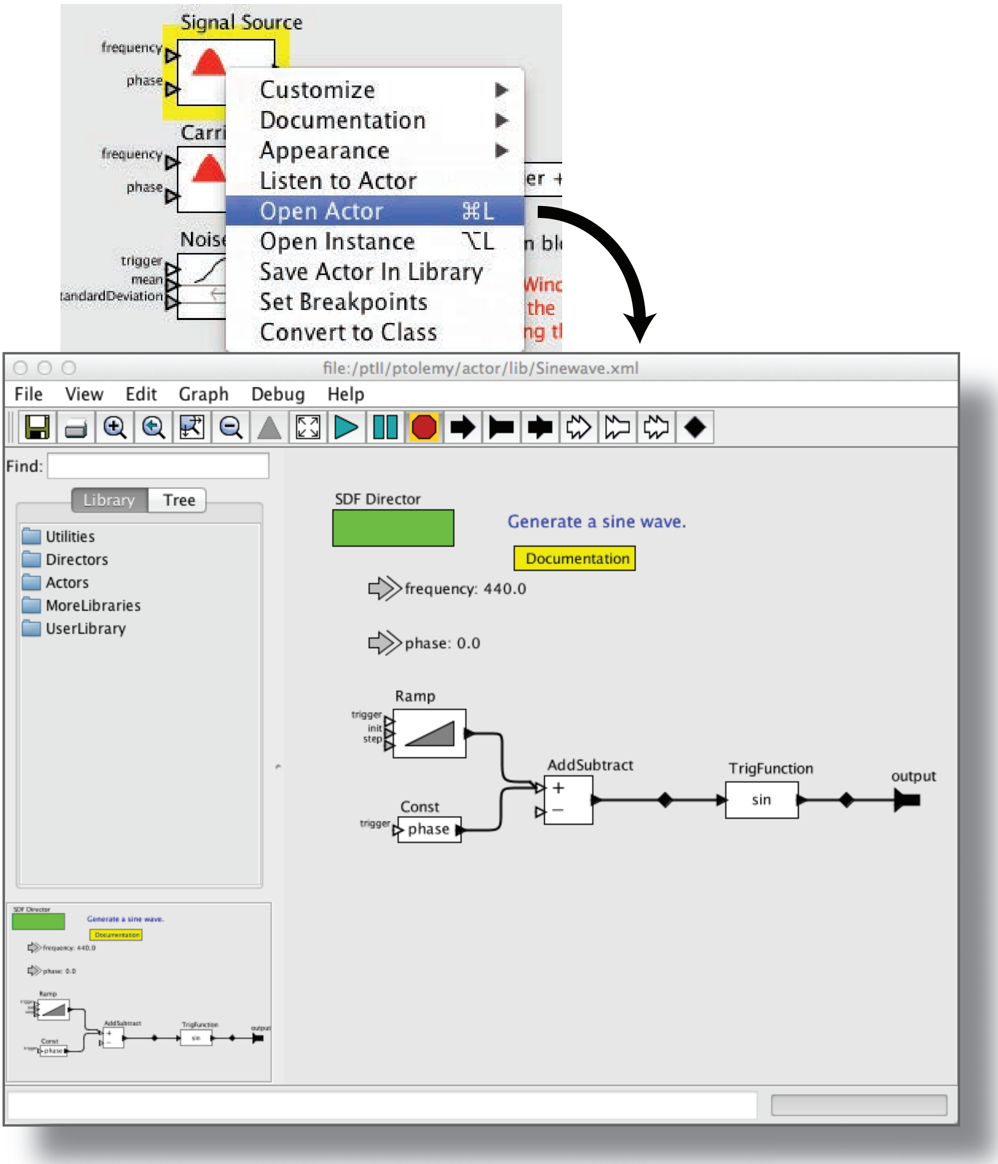


Figure 2.5: Invoke `Open Actor` on composite actors to reveal their implementation.

2.1.2 Creating and Running a Model

Create a new model by selecting [File→New→Graph Editor] in the menu bar. You should see something like the window shown in Figure 2.6. The left-hand side of the page shows a library of components that can be dragged into the model-building area. Perform the steps outlined below to create a simple model.

- Open the Actors library and then Sources. Find the Const actor under Sources→GenericSources and drag an instance into the model-building area.
- Open Sinks→GenericSinks and drag a Display actor onto the page (see boxes on pages 48 and 49).

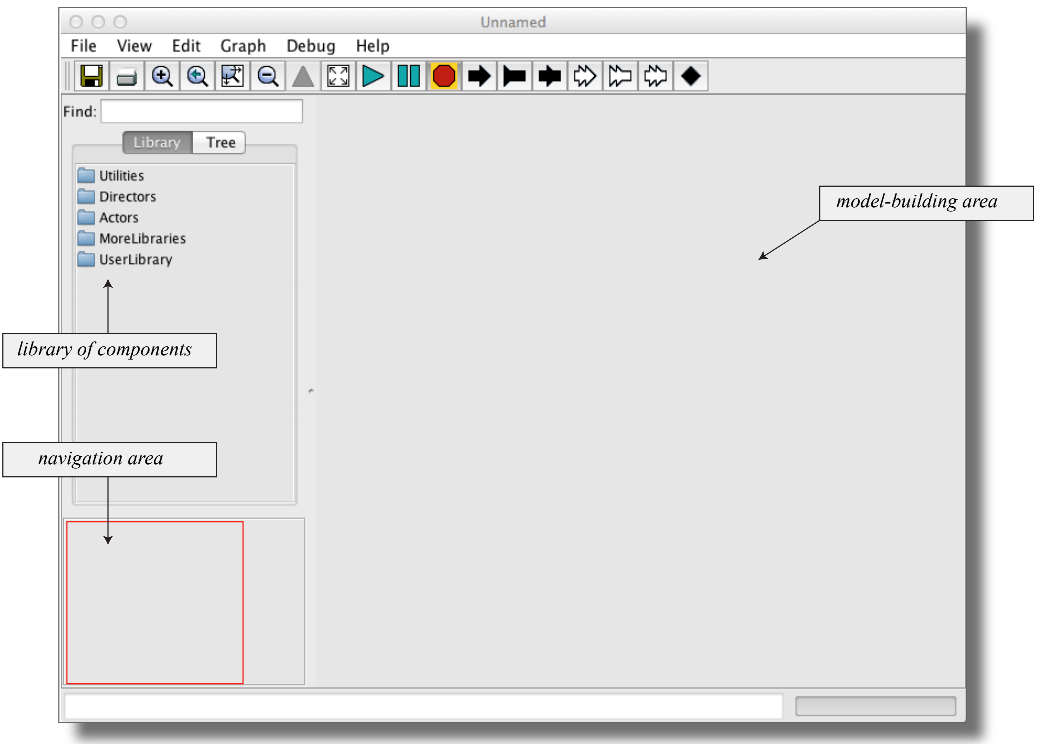


Figure 2.6: An empty Vergil Graph Editor.

- Drag a connection from the output port on the right of the Const actor to the input port of the Display actor.
- Open the `Directors` library and drag the SDF Director onto the page. The director controls various aspects of the model's functionality, such as (for example) how many iterations the model will execute (which is, by default, just one iteration).

Now you should have something that looks like Figure 2.7. In this model, the Const actor will create a string and the Display actor will display that string. Set the string variable to "Hello World" by double or right clicking on the Const actor icon and selecting [`Customize`→`Configure`], which will display the dialog box shown in Figure 2.8. En-

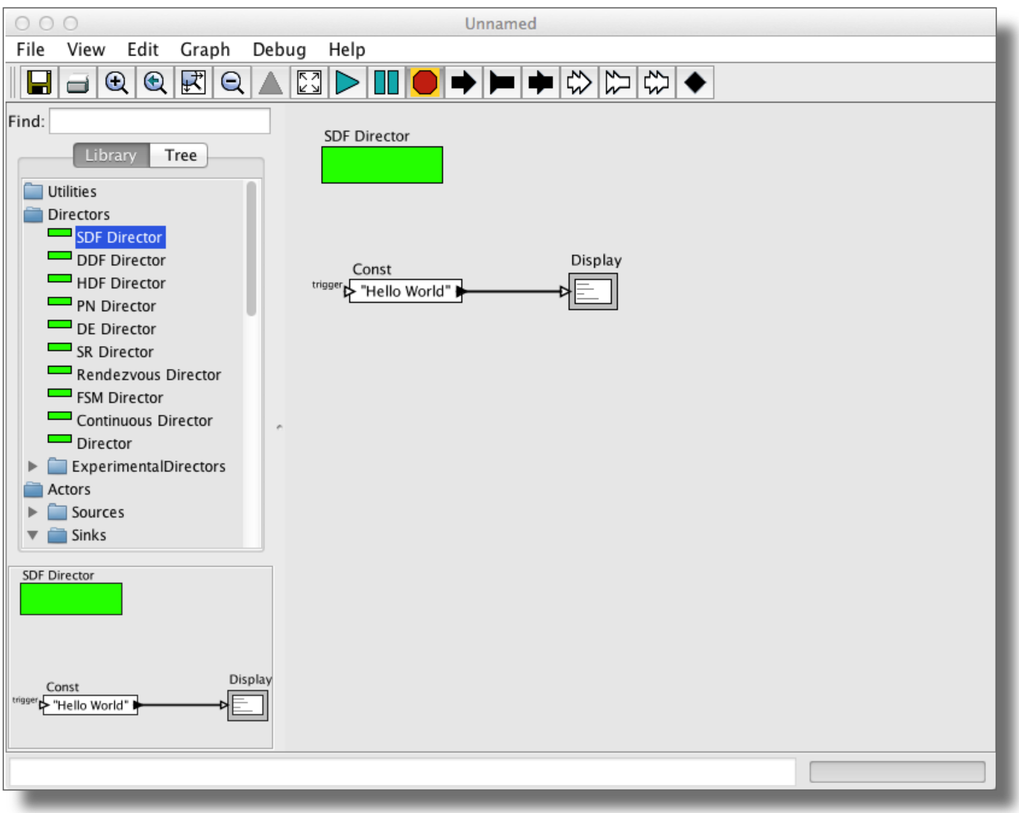


Figure 2.7: The Hello World example. [online]

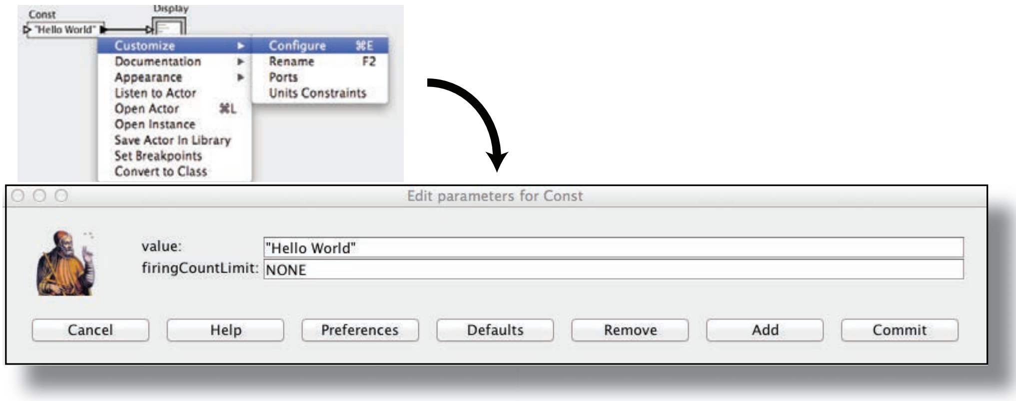


Figure 2.8: The Const parameter editor.

ter the string `"Hello World"` (with the quotation marks) for the *value* parameter and click the Commit button. (The quotation marks ensure that the expression will be interpreted as a string.) If you run this model, a display window will show the text "Hello World."

You may wish to save your model using the File menu. File names for Ptolemy II models should end in ".xml" or ".moml" so that Vergil will properly process the file the next time it is opened.

2.1.3 Making Connections

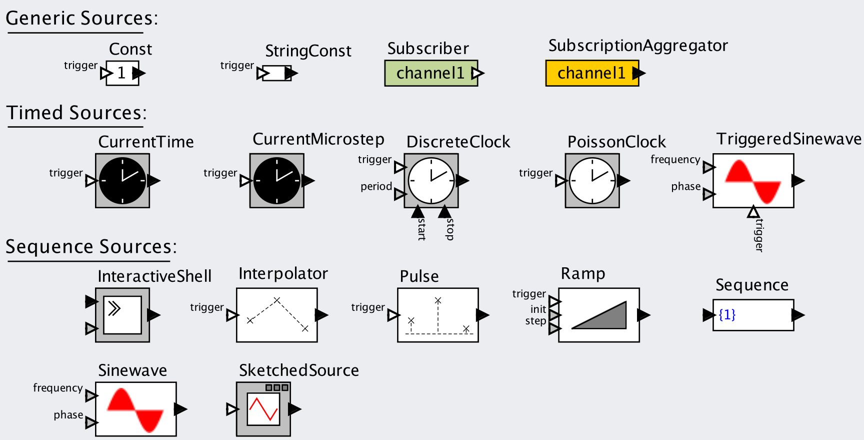
The model constructed above contains two actors and one connection between them. If you move either actor (by clicking and dragging), the connection will be re-routed automatically. We can now explore how to create and manipulate more complicated connections.

First, create a model in a new graph editor that includes an SDF Director, a Ramp actor (found in the `Sources→SequenceSources` library), a Display actor, and a SequencePlotter actor (found in the `Sinks→SequenceSinks` library), as shown in Figure 2.9.

Suppose we wish to route the output of the Ramp actor to both the Display and the SequencePlotter. In this case, we need an explicit relation in the diagram. A **relation** is a

Sidebar: Sources Library

The three `Actors→Sources` libraries contain sources of signals.



- **Const** produces a value (or expression) set by a parameter (see Chapter 13).
- **StringConst** produces strings and may reference other parameters (see Section 13.2.3).
- **Subscriber** outputs data received from a Publisher (described in the Sinks Library sidebar on page 49).
- **SubscriptionAggregator** combines data from multiple Publishers using a specified operation (currently addition or multiplication).
- **CurrentTime**, **CurrentMicrostep**, **DiscreteClock**, and **PoissonClock** are timed sources described in Chapter 7.
- **TriggeredSinewave** and **Sinewave** produce sinusoidal outputs based on either the current time or a specific sample rate, respectively.
- **InteractiveShell** outputs the value entered in a shell window opened during execution.
- **Interpolator** and **Pulse** both generate waveforms, one by interpolating between values, and the other by zero-filling between values.
- **Ramp** produces a steadily increasing or decreasing output sequence.
- **Sequence** produces an arbitrary sequence of values, possibly periodic.
- **SketchedSource** produces a sequence specified interactively with the mouse.

Sidebar: Sinks Library

Actors in the Actors→Sinks library serve as destinations for signals. Most are plotters, described in Chapter 17. The few that are not are contained in the Sinks→GenericSinks library, and shown below. All of these sinks accept any data type at their inputs.

- **Discard** discards all inputs. In most domains, leaving an output disconnected has the same effect (the Rendezvous domain is an exception).
- **Display** displays the values of inputs in a text window that opens when the model is executed.
- **MonitorValue** displays its inputs in the Vergil icon that displays the model.
- **Publisher** establishes named connections to instances of Subscriber and SubscriptionAggregator (described in the Sources sidebar on page 48).
- **Recorder** stores the values of inputs internally; custom Java code is then needed to access those values.
- **SetVariable** sets the value of a variable or parameter defined in the model. Since this variable may be read by another actor that has no connection to the SetVariable actor, SetVariable may introduce nondeterminism into a model. That is, the results of executing the model may depend on arbitrary scheduling decisions that determine whether SetVariable executes before actors that read the affected variable. To mitigate this risk, SetVariable has a parameter called *delayed* that by default is set to true. When it is true, the affected variable will only be set at the end of the current iteration of the director. For most directors (but notably, not for PN or Rendezvous), this setting will ensure deterministic behavior. The SetVariable actor has an output port that can be used to ensure that other specified actors are executed only after SetVariable executes. This approach can also eliminate nondeterminacy, even if *delayed* is set to false.

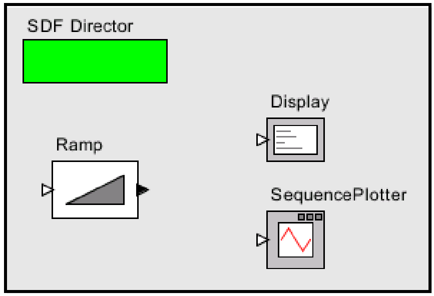


Figure 2.9: Three unconnected actors in a model.

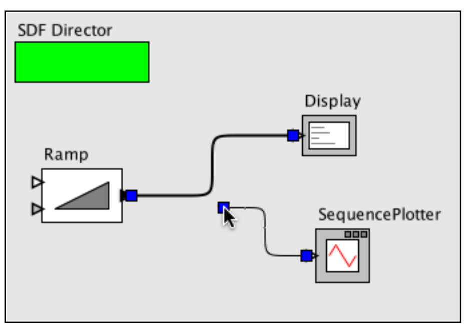


Figure 2.10: Dragging a connection to an existing one will create a three-way connection by creating an explicit relation.

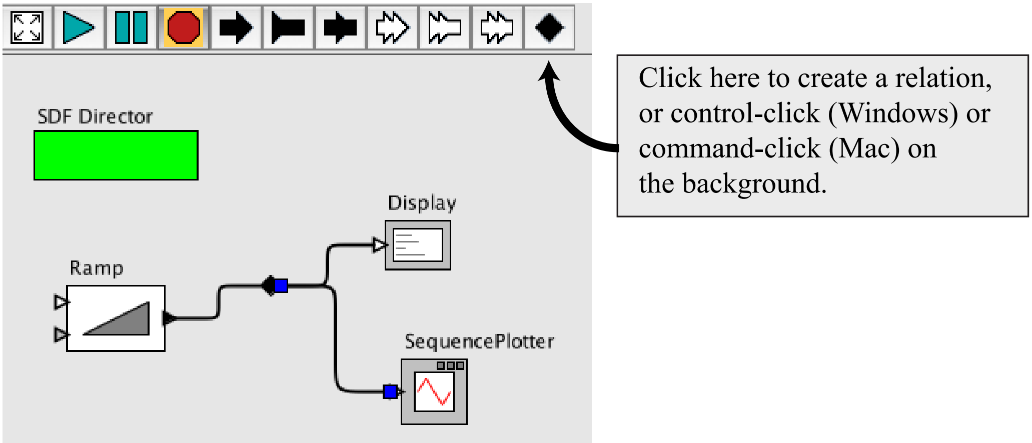


Figure 2.11: A relation can also be created by clicking on the black diamond in the toolbar.

splitter that enables connecting more than two ports; it is represented in the diagram by a black diamond, as shown in Figure 2.11. It can be created in several ways:

- Drag the endpoint of a link into the middle of an existing line, as shown in Figure 2.10.
- Control-click[3] on the background. A relation will appear.
- Click on the button in the toolbar with the black diamond on it, as shown in Figure 2.11.

Note that if you simply click and drag on the relation, the relation is selected and moved, which does not result in a connection. To make a connection, hold the control key while clicking and dragging on the relation.

In the model shown in Figure 2.11, the relation is used to broadcast the output from a single port to a number of places. The single port still has only one connection — a connection to the relation.

Even with the simple diagrams we have created so far, it can become tedious to arrange the icons and control the routing of wires between ports and relations. Fortunately, Ptolemy II includes a sophisticated automated layout tool, which you can find in the [Graph→

[3]On a Macintosh, use command-click. (This substitution applies throughout the book.)

`Automatic Layout`] menu command (or invoke using Control-T). This tool was contributed by Spönemann et al. (2009). Occasionally, however, you may still need to manually adjust the layout. To specifically control the routing of connections, you can use multiple relations along a connection and independently control the position of each one. Multiple relations used in a single connection are called a **relation group**. In a relation group, there is no significance to the order in which relations are linked.

Ptolemy II supports two types of ports, which are indicated by filled and unfilled triangles. Filled triangles designate single-input (or single-output) ports, while unfilled triangles allow multiple inputs (or outputs). For example, the output port of the Ramp actor is a **single port**, while the input port of the Display and SequencePlotter actors are **multiports**. In multiports, each connection is treated as a separate **channel**.

Choose another signal source from the library and add it to the model. Connect this additional source to the SequencePlotter or to the Display, thus implementing a multiport input to those blocks. Not all data type inputs will be accepted, however; the SequencePlotter, for example, can only accept inputs of type *double* or types that can be losslessly converted to *double*, such as *int*. In the next section, we discuss data types and their use in Ptolemy II models.

2.2 Tokens and Data Types

In the example of Figure 2.7, the Const actor creates a sequence of values on its output port. Each value in the sequence is called a **token**. A token is created by one actor, sent through an output port, and received by one or more destination actors, each of which retrieves the token through an input port. In this case, the Display actor will receive the tokens produced by the Const actor and display them in a window. A token is simply a unit of data, such as a string or numerical value, that is communicated between two actors via ports.

The tokens produced by the Const actor can have any value that can be expressed in the Ptolemy II expression language (see Chapter 13). Try setting the value to `1` (the integer with value one), or `1.0` (the floating-point number with value one), or `{1.0}` (a single-entry array containing 1.0), or `{value=1,name="one"}` (a record with two elements: an integer named "value" and a string named "name"), or even `[1,0;0,1]` (the two-by-two identity matrix). These are all valid expressions that can be used in the Const actor.

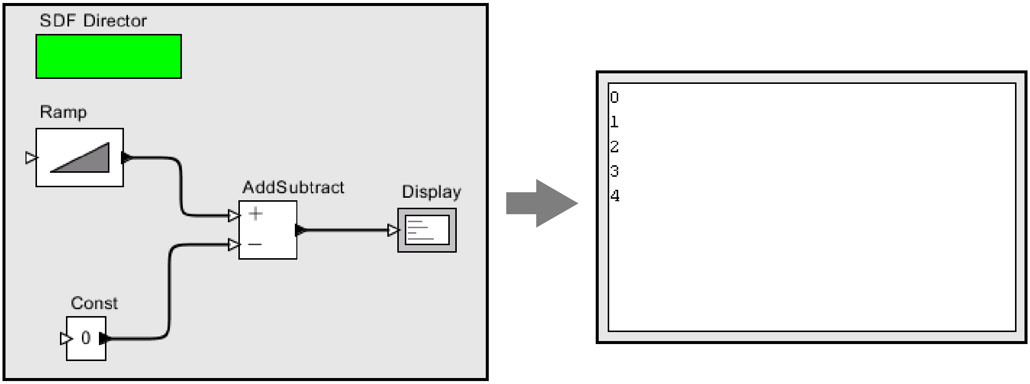


Figure 2.12: Another example, used to explore data types. [online]

The Const actor is able to produce data with different **types**, and the Display actor is able to display data with different types. Most actors in the actor library are **polymorphic actors**, meaning that they can operate on or produce data with multiple types, though their specific processing behavior may be different for different types. Multiplying matrices, for example, is not the same operation as multiplying integers, but both can be accomplished using the MultiplyDivide actor in the `Math` library. Ptolemy II includes a sophisticated type system that allows actors to process different data types efficiently and safely. The type system was created by Xiong (2002) and is described in Chapter 14.

To explore data types further, create the model in Figure 2.12, using the SDF Director. The Ramp actor is listed under the `Sources→SequenceSources` sublibrary, and the AddSubtract actor is listed under the `Math` library (see box on page 57). Set the *value* parameter of the Const to be 0 and the *iterations* parameter of the director to 5. Running the model will result in the display of five consecutive numbers starting at 0 and ending at 4, as shown in the figure. These values are produced by subtracting the constant output of the Const actor from the current value of the Ramp. Experiment with changing the value of the Const actor and see how it changes the five numbers at the output.

Now change the value of the Const actor back to `"Hello World"`. When you execute the model, you should see an **exception** window, as shown in Figure 2.13. This error is caused by attempting to subtract a string value from an integer value. This error is one kind of **type error**.

The actor that caused the exception is highlighted, and the name of the actor is shown in the exception. In Ptolemy II models, all objects have a dotted name. The dots separate elements in the hierarchy. Thus, ".helloWorldSubtractError.AddSubtract" is an object named "AddSubtract" contained by a object named ".helloWorldAddSubtractError". (The model was saved as a file called `helloWorldAddSubtract.xml`.)

Exceptions can be a very useful debugging tool, particularly when developing components in Java. To illustrate how to use them, click on the Display Stack Trace button in the exception window of Figure 2.13. You should see a stack trace like that shown in Figure 2.13. This window displays the execution sequence that resulted in the exception. For example, if you scroll towards the bottom, text similar to the following will be displayed:

```
at ptolemy.data.StringToken._subtract(StringToken.java:359)
```

This line indicates that the exception occurred within the `subtract` method of the class ptolemy.data.StringToken, at line 359 of the source file StringToken.java. Since Ptolemy II is distributed with source code (and most installation mechanisms support installation of the source), this can be very useful information. For type errors, you probably do not need to see the stack trace, but if you have extended the system with your own Java code or you encounter a subtle error that you do not understand, then looking at the stack trace can be very illuminating.

To find the file StringToken.java referred to above, find the Ptolemy II installation directory. If that directory is `$PTII`, then the location of this file is given by the full class name, but with the periods replaced by slashes; in this case, it is at

```
$PTII/ptolemy/data/StringToken.java
```

The slashes will be backslashes under Windows.

Let's try a small change to the model to eliminate the exception. Disconnect the Const actor from the lower port of the AddSubtract actor and connect it instead to the upper port, as shown in Figure 2.14. You can do this by selecting the connection, deleting it (using the delete key), and adding a new connection–or by selecting it and dragging one of its endpoints to the new location. Notice that the upper port is an unfilled triangle; this indicates that you can make more than one connection to it. Now when you run the model you should see a sequence of string values starting with "`0Hello World`", as shown in

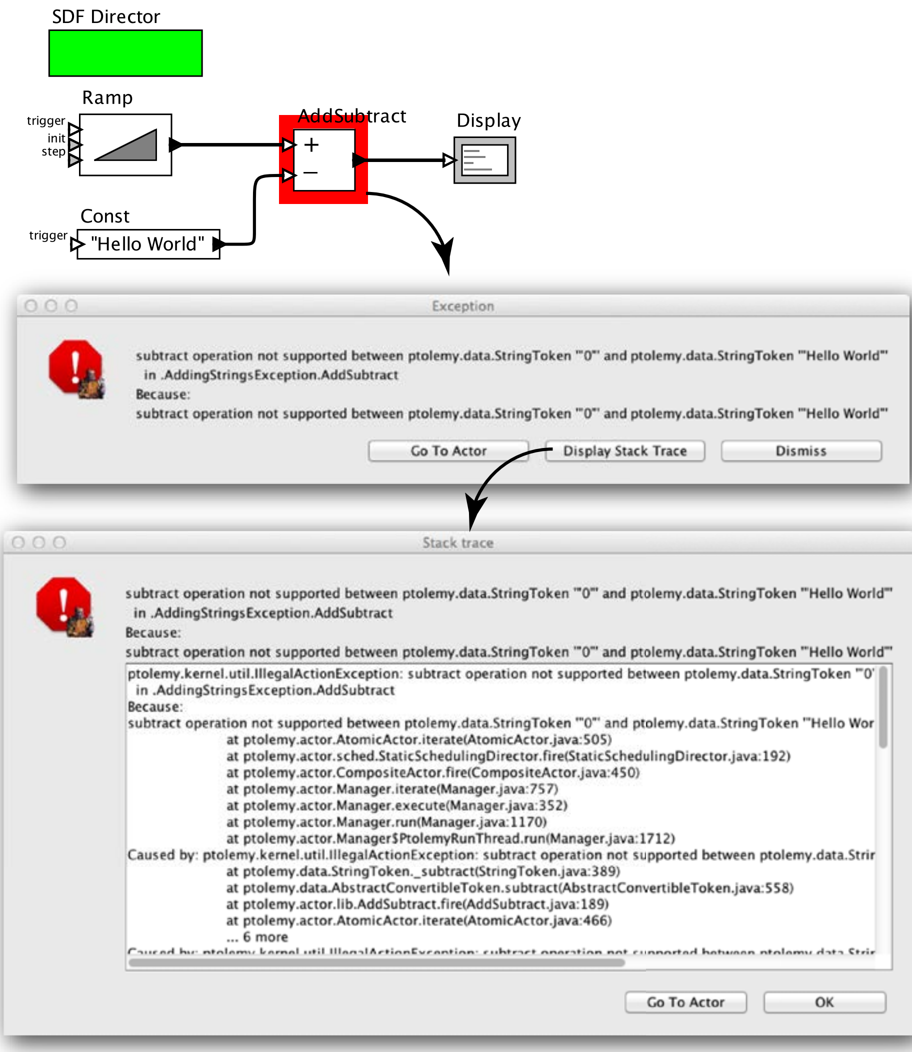


Figure 2.13: An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from integers. Examining the stack trace for such exceptions can sometimes be useful for debugging your model, particularly if it includes custom actors. [online]

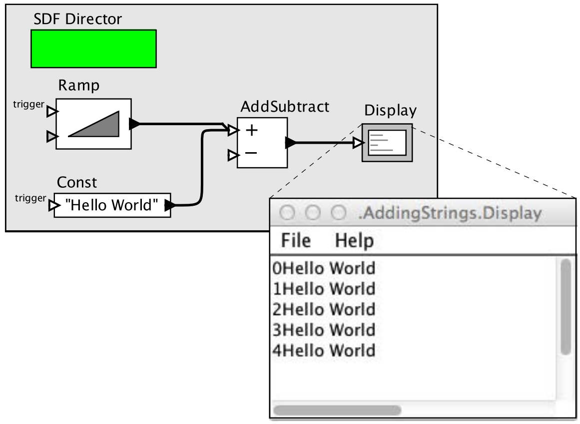


Figure 2.14: Addition of a string to an integer. [online]

the figure. Following Java conventions, strings can be added (using concatenation), but they cannot be subtracted.

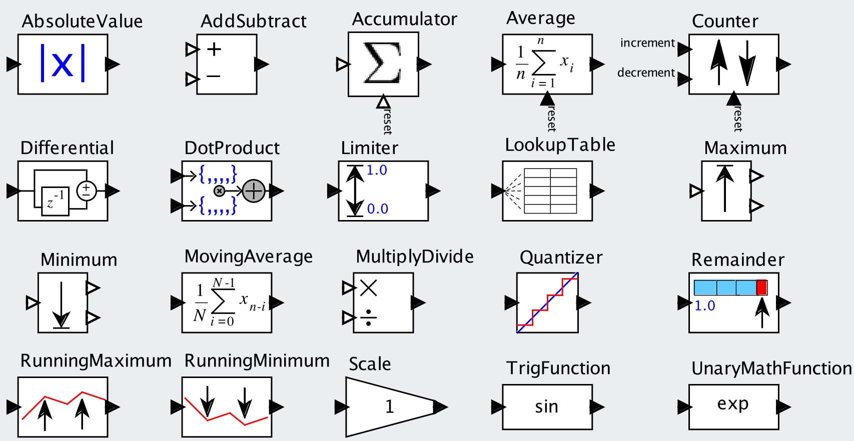
All the connections to a multiport must have the same type. In this case, the multiport has a sequence of integers coming in (from the Ramp) and a sequence of strings (from the Const). The Ramp integers are automatically converted to strings and concatenated with "Hello World" to generate the output sequence.

As illustrated by the above example, Ptolemy II automatically performs type conversions when required by the actor, if possible. As a rough guideline, Ptolemy II will perform automatic type conversions when there is no loss of information. An *int* can be converted to a *string*, for example, but not vice versa. An *int* can be converted to a *double*, but not vice versa. An *int* can be converted to a *long*, but not vice versa. The details are explained in Chapter 14, but it is usually not necessary to remember the conversion rules. Typically, the data type conversions and resulting computations will perform as would be expected.

To further explore data types, try modifying the Ramp so that its parameters have different types. For example, try making *init* and *step* strings.

Sidebar: Math Library

Actors in the `Actors→Math` library perform mathematical operations, and are shown below (except the most versatile, Expression, explained in Chapter 13).



These are mostly self explanatory:

- **AbsoluteValue** computes the absolute value of the input.
- **AddSubtract** adds tokens at the *plus* inputs, and subtracts tokens at the *minus* inputs.
- **Accumulator** outputs the sum of all tokens that have arrived.
- **Average** outputs the average of all tokens that have arrived.
- **Counter** counts up or down when inputs are received on the two input ports.
- **Differential** outputs the difference between the current input and the previous one.
- **DotProduct** computes the inner product of two array or matrix inputs.
- **Limiter** limits the value of the input to a specified range of values.
- **LookupTable** performs a lookup into a table provided as an array.

(Continued on page 58.)

Sidebar: Math Library (Continued)

- **Maximum** outputs the maximum of the currently available input tokens.
- **Minimum** outputs the minimum of the currently available input tokens.
- **MovingAverage** outputs the average of some number of the most recent inputs.
- **MultiplyDivide** multiplies the tokens on the *multiply* inputs, and divides by the tokens on the *divide* inputs.
- **Quantizer** outputs the nearest value to the input from a specified list of values.
- **Remainder** outputs the remainder after dividing the input by the *divisor* parameter.
- **RunningMaximum** outputs the maximum value seen so far at the input.
- **RunningMinimum** outputs the minimum value seen so far at the input.
- **Scale** multiplies the input by a constant given as a parameter.
- **TrigFunction** performs trigonometric functions, including cosine, sine, tangent, arccosine, arcsine, and arctangent.
- **UnaryMathFunction** performs various math functions with a single argument, including exponentiation, logarithm, sign, squaring, and square root.

There are subtleties. Some actors with multiple input ports (AddSubtract, Counter, and MultiplyDivide) or with multiport inputs (Maximum and Minimum), do not require input tokens on all input channels. When these actors fire, they operate on whatever input tokens are available. For example, if AddSubtract has no tokens on any *plus* input channel, and only one token on a *minus* input channel, then the output will be the negative of that one token. Whether inputs are available when the actor fires depends on the director and the model. With the SDF director, exactly one input token is provided on every input channel for every firing.

These actors are **polymorphic**; they can operate on a variety of data types. For example, the AbsoluteValue actor accepts inputs of any scalar type (see Chapter 14). If the input type is *complex*, it computes the magnitude. For other scalar types, it computes the absolute value.

Accumulator and Average both have *reset* input ports (at the bottom of the icon). They calculate the sum or average of sequences of inputs, and they can be reset.

2.3 Hierarchy and Composite Actors

Ptolemy II supports (and encourages) hierarchical models. These are models that contain components that are themselves models. These components are called composite actors.

Consider the typical signal processing problem of recovering a signal from a noisy channel. Using Ptolemy II, we will create a composite actor modeling a communication channel that adds noise, and then use that actor within a larger model.

To create the composite actor, drag a **CompositeActor** from the `Utilities` library. In the context menu (obtained by right clicking over the composite actor), select [`Customize` →`Rename`], and give the composite an appropriate name, like **Channel**, as shown in Figure 2.15. (Note that you can also supply a *Display name*, which is arbitrary text that will be displayed instead of the name of the actor.) Then, using the context menu again, select

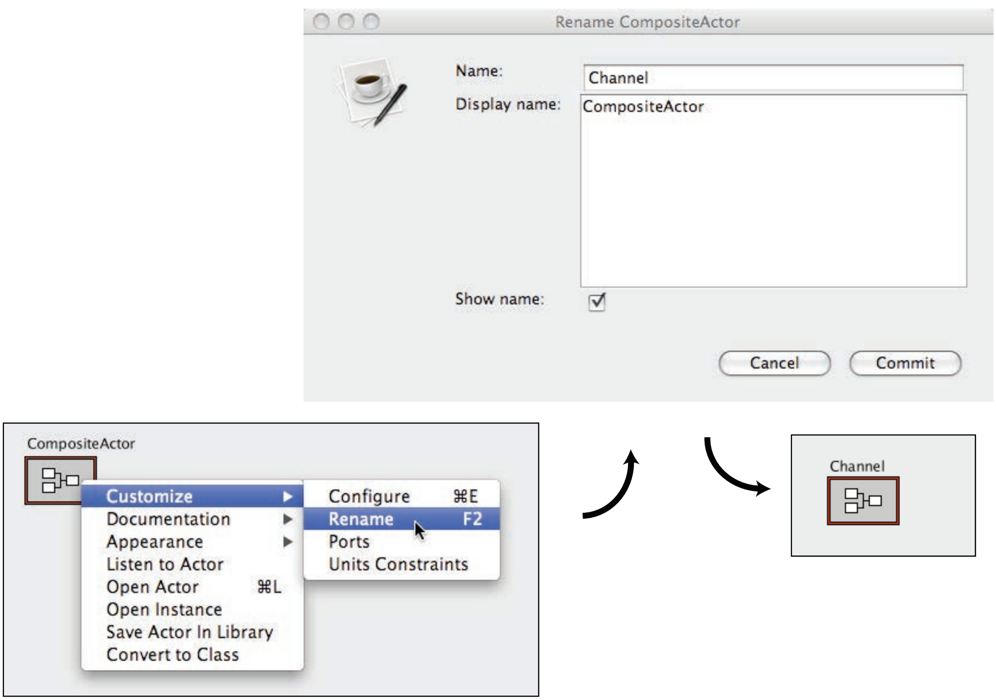


Figure 2.15: Changing the name of an actor.

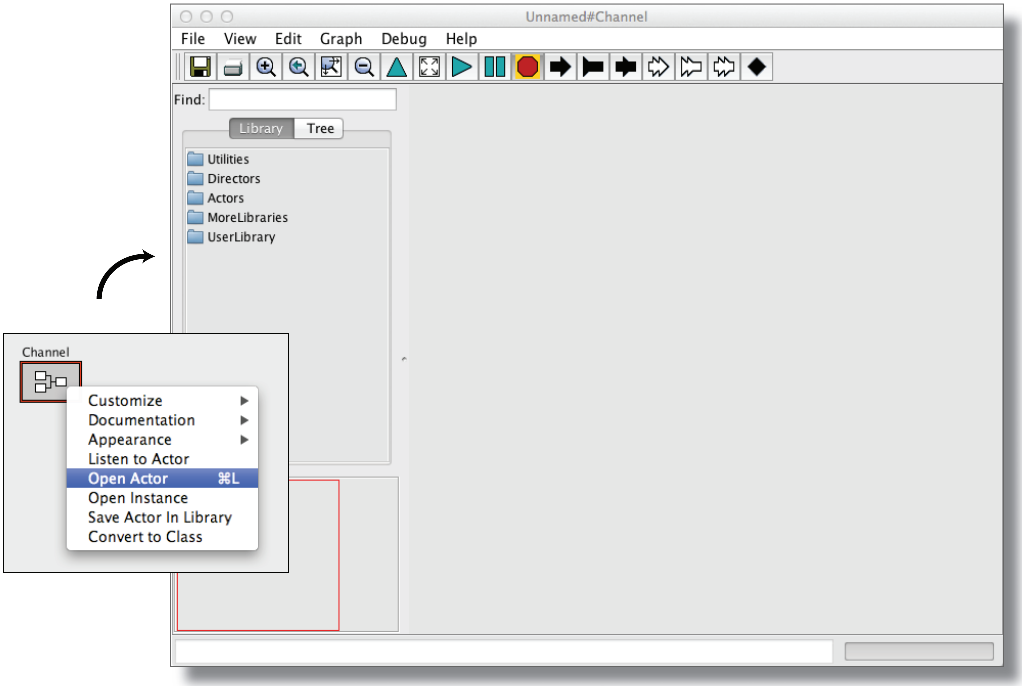


Figure 2.16: Opening a new composite actor, which shows the blank inner actor.

`Open Actor`. This will open a new window with an empty graph editor, as shown in Figure 2.16. Note that the original graph editor is still open; to see it, move the new graph editor window aside by dragging the title bar of the window. Alternatively, you can click on the upward pointing triangle in the toolbar to navigate back to the container.

2.3.1 Adding Ports to a Composite Actor

The newly created composite actor needs input and output ports. There are several ways to add them, the easiest of which is to click on the port buttons in the toolbar. The port buttons are shown as white and black arrowheads on the toolbar, as described in Figure 2.17. You can explore the ports in the toolbar by hovering the mouse over each button; a tool tip will pop up that describes the button.

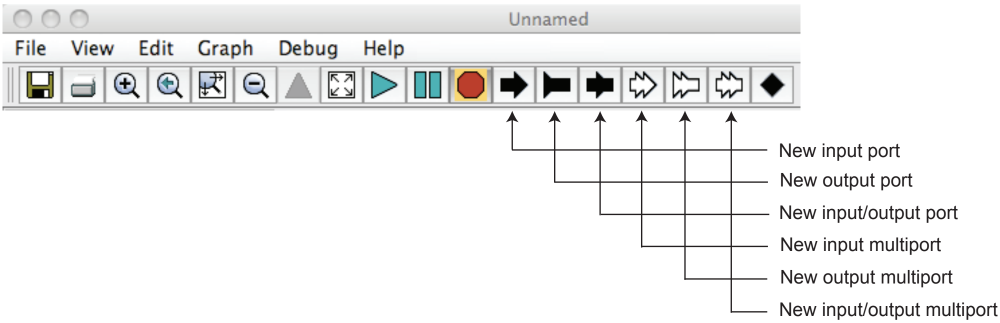


Figure 2.17: Summary of toolbar buttons for creating new ports.

Create an input port and an output port and rename them *input* and *output* by right clicking on the ports and selecting [Customize→Rename]. Note that, as shown in Figure 2.18, you can also right click on the background of the composite actor and select [Customize →Ports] to add ports, remove ports, or change whether a port is an input, an output, or a multiport. The resulting dialog box also allows you to set the type of the port, though you will typically not need to set the port type since Ptolemy II determines the type from the

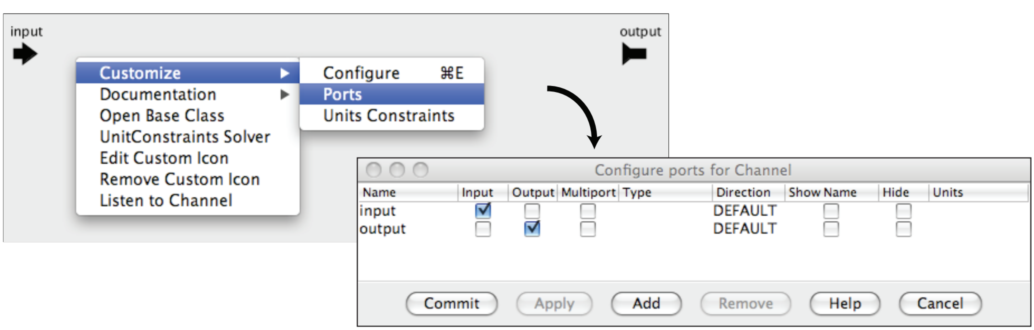


Figure 2.18: Right clicking on the background brings up a dialog that can be used to configure ports.

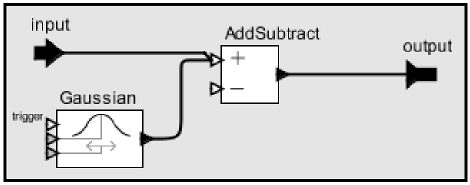


Figure 2.19: A simple channel model defined as a composite actor.

port's connections. You can also specify the direction of a port[4] and whether the name of the port is shown outside the icon (by default it is not), or even whether the port is shown at all.

Using these ports, create the diagram shown in Figure 2.19[5]. The **Gaussian** actor, which is in the `Random` library, creates values from a Gaussian distributed random variable. If you then navigate back to the container, you should be able to easily create the model shown in Figure 2.20. The Sinewave actor is listed under `Sources→SequenceSources`, and the SequencePlotter actor under `Sinks→SequenceSinks`. The Sinewave actor is also a composite actor (try opening the actor). If you execute this model (you will probably want to set the *iterations* parameter of the director to something reasonable, like 100), you should see a plot similar to the one in Figure 2.20.

2.3.2 Setting the Types of Ports

In the above example, it was not necessary to define the port types. Their types were inferred from the connections and constants in the model, as explained in Chapter 14. Occasionally, however, you will need to set the types of the ports. Notice in Figure 2.18 that there is a column in the dialog box that enables the port type to be specified. To specify that a port has type boolean, for example, you could enter *boolean*. This will only have an effect, however, if the port is contained by an opaque composite actor (one that has its own director). In the model you have built, the Channel composite actor is transparent, and setting the types of its ports will have no effect. Transparent composite

[4]The direction of a port is defined by where it appears on the icon for the actor. By default, input ports appear on the left, output ports on the right, and ports that are both inputs and outputs appear on the bottom of the icon.

[5]Hint: to create a connection starting on one of the external ports, hold down the control key when dragging, or on a Macintosh, the command key.

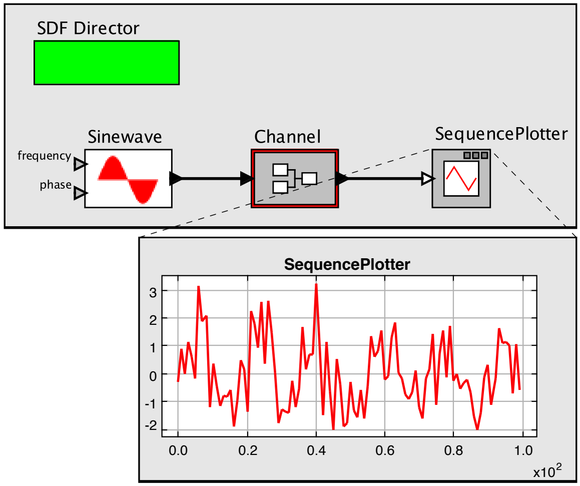


Figure 2.20: A simple signal processing example that adds noise to a sinusoidal signal. [online]

actors are merely a notational convenience, and the ports of the composite play no role in the execution of the model.

Commonly used types include *complex*, *double*, *fixedpoint*, *float*, *general*, *int*, *long*, *matrix*, *object*, *scalar*, *short*, *string*, *unknown*, *unsignedByte*, *xmlToken*, *arrayType(int)*, *arrayType(int, 5)*, *[double]* and {*x=double, y=double*}. A detailed description of types appears in Chapter 14.

In the Ptolemy II expression language, square braces are used for matrices, and curly braces are used for arrays. An array is an ordered list of tokens of arbitrary type. A Ptolemy II matrix is a one- or two-dimensional structure containing numeric types. To specify a double matrix port type, for example, you would use the following expression:

```
[double]
```

This expression creates a 1 x 1 matrix containing a *double* (the value of which is irrelevant here). It serves as a prototype to specify a double matrix type. Similarly, we can specify an array of complex numbers as

```
{complex}
```

A record has an arbitrary number of data elements, each of which has a name and a value, where the value can be of any type. To specify a record containing a string called "name" and an integer called "address" you would use the following expression to specify the type:

```
{name=string, address=int}
```

Details about arrays, matrices, and records are given in Chapter 13.

2.3.3 Multiports, Busses, and Hierarchy

As explained above in Section 2.1.3, a multiport handles multiple independent channels. In a similar manner, a relation can handle multiple independent channels. A relation has a *width* parameter, which by default is set to Auto, meaning that the width is inferred from the context. If the width is not unity, it will be displayed as shown in Figure 2.21

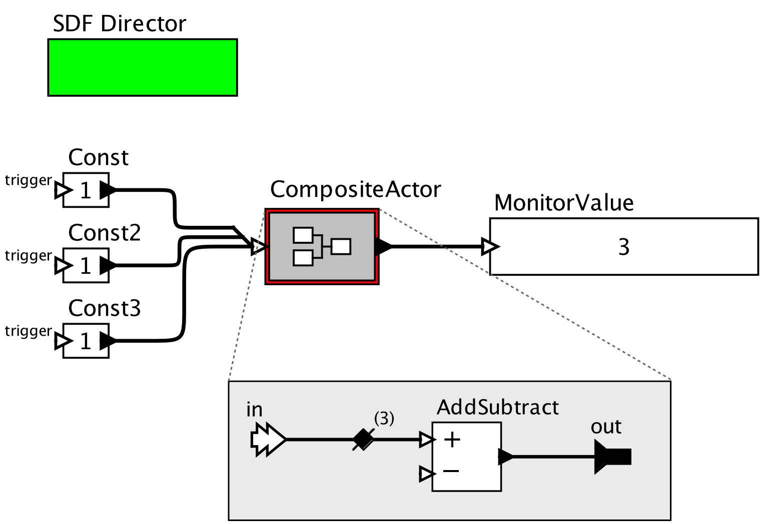


Figure 2.21: Relations can have a width greater than one, meaning that they carry more than one channel. The width of the relation inside the composite in this figure is inferred to be three. [online]

as a number adjacent to a slash through a relation. Such a connection is called a **bus**, because it carries multiple signals. If in the example in Figure 2.21 you set the width of the relation inside the composite to 2, then only two of the three channels will be used inside the composite.

In most circumstances, the width of a relation is inferred from the usage (Rodiers and Lickly, 2010), but occasionally it is useful to set it explicitly. You can also explicitly construct a bus using the **BusAssembler** actor, or split one apart using a **BusDisassembler** actor, both found in the `FlowControl→Aggregators` library.

2.4 Annotations and Parameterization

In this section, we will enhance the model in Figure 2.20 in several ways. We will add parameters, insert decorative and documentary annotations, and customize the actor icons.

2.4.1 Parameters in Hierarchical Models

First, notice from Figure 2.20 that the noise overwhelms the sinusoid, making it barely visible. A useful modification to this channel model would be to add a **parameter** that sets the level of the noise. To make this change, open the channel model by right clicking on it and selecting `Open Actor`. In the channel model, add a parameter by dragging one in from the sublibrary `Utilities→Parameters`, as shown in Figure 2.22. Right click on the parameter to rename it to `noisePower`. (To use a parameter in expressions, its name cannot have spaces.) Right click (or double click) on the parameter to change its default value to 0.1.

This parameter can now be used to set the amount of noise. The Gaussian actor has a parameter called *standardDeviation*. Since the noise power is equal to the variance, not the standard deviation, change the *standardDeviation* parameter so that its value is `sqrt(noisePower)`, as shown in Figure 2.23. (See Chapter 13 for details on the expression language.)

To observe the effect of the parameter, return to the top-level model and edit the parameters of the Channel actor (by either double clicking or right clicking and selecting [`Customize→Configure`]). Change the *noisePower* from the default 0.1 to 0.01. Run the model. You should now get a relatively clean sinusoid like that shown in Figure 2.24.

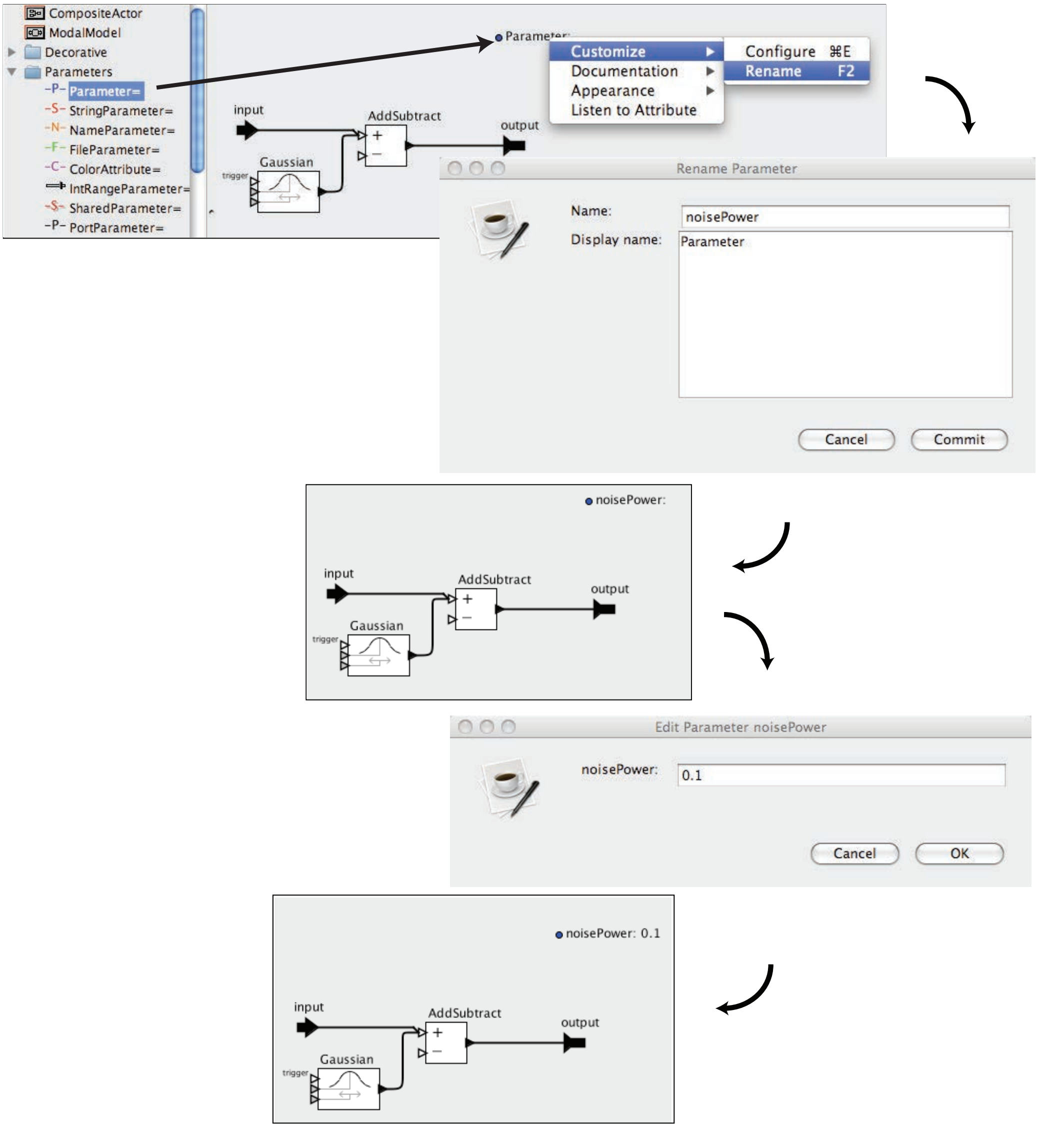


Figure 2.22: Adding a parameter to the Channel model. [online]

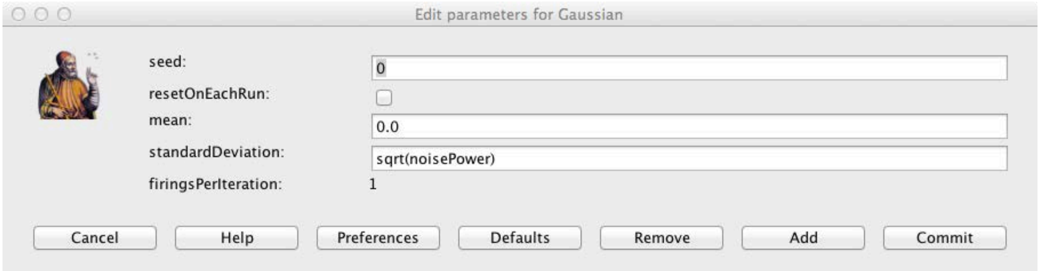


Figure 2.23: The standard deviation of the Gaussian actor is set to the square root of the noise power.

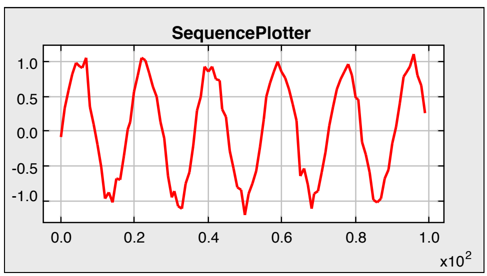


Figure 2.24: The output of the simple signal processing model in Figure 2.20 with noise power = 0.01. [online]

You can also add parameters to a composite actor by clicking on the "Add" button in the edit parameters dialog for the Channel composite. This dialog is accessed by double clicking on the Channel icon, by right clicking and selecting [`Customize→Configure`], or by right clicking on the background inside the composite and selecting [`Customize→Configure`]. A key point to note here is that parameters that are added this way will not be visible in the diagram, so this mechanism should be used sparingly and only when there is a good reason to hide parameters from someone browsing the model.

Finally, note that it is possible to create an object called a **port parameter** or **parameter port** that is both a parameter and a port. The *frequency* and *phase* objects in Figure 2.5 are port parameters. They can be accessed in an expression like any other parameter, but when an input arrives at the parameter port during execution, the value of the parameter gets updated. To create a port parameter object in a model, simply drag one in from the `Utilities→Parameters` library and assign it a name.

2.4.2 Decorative Elements

The model can also be enhanced with a variety of decorative elements — that is, elements that affect its appearance but not its functionality. Such elements can improve the readability and aesthetics of a model. For example, try dragging an *Annotation* from the `Utilities→Decorative` sublibrary, and creating a title for the diagram. Such annotations are highly recommended; they correspond to comments in programs and can greatly improve readability. Other decorative elements (such as geometric shapes) can be dragged into the diagram from the same library.

2.4.3 Creating Custom Icons

Vergil provides an icon editor to enable users to create custom actor icons. To create a custom icon, right click on the standard icon and select [`Appearance→Edit Custom Icon`], as shown in Figure 2.25. The box in the middle of the icon editor displays the size of the default icon, for reference. Try creating an icon like the one shown in Figure 2.26. Hint: The fill color of the rectangle is set to `none` and the fill color of the trapezoid is first set using the color selector, then modified to have an *alpha* (transparency) of 0.5. Finally, since the icon itself has the actor name in it, the [`Customize→Rename`] dialog is used to deselect *show name*.

2.5 Navigating Larger Models

Some models are too large to view on one screen. There are four toolbar buttons, shown in Figure 2.27, that permit zooming in and out. The "Zoom reset" button restores the zoom factor to its original value, and "Zoom fit" calculates the zoom factor so that the entire model is visible in the editor window.

It is also possible to pan over a model. Consider the window shown in Figure 2.28. Here, we have zoomed in so that icons are larger than the default. The **pan window** at the lower left shows the entire model, with a red box indicating the visible portion of the model. By clicking and dragging in the pan window, it is easy to navigate the entire model.

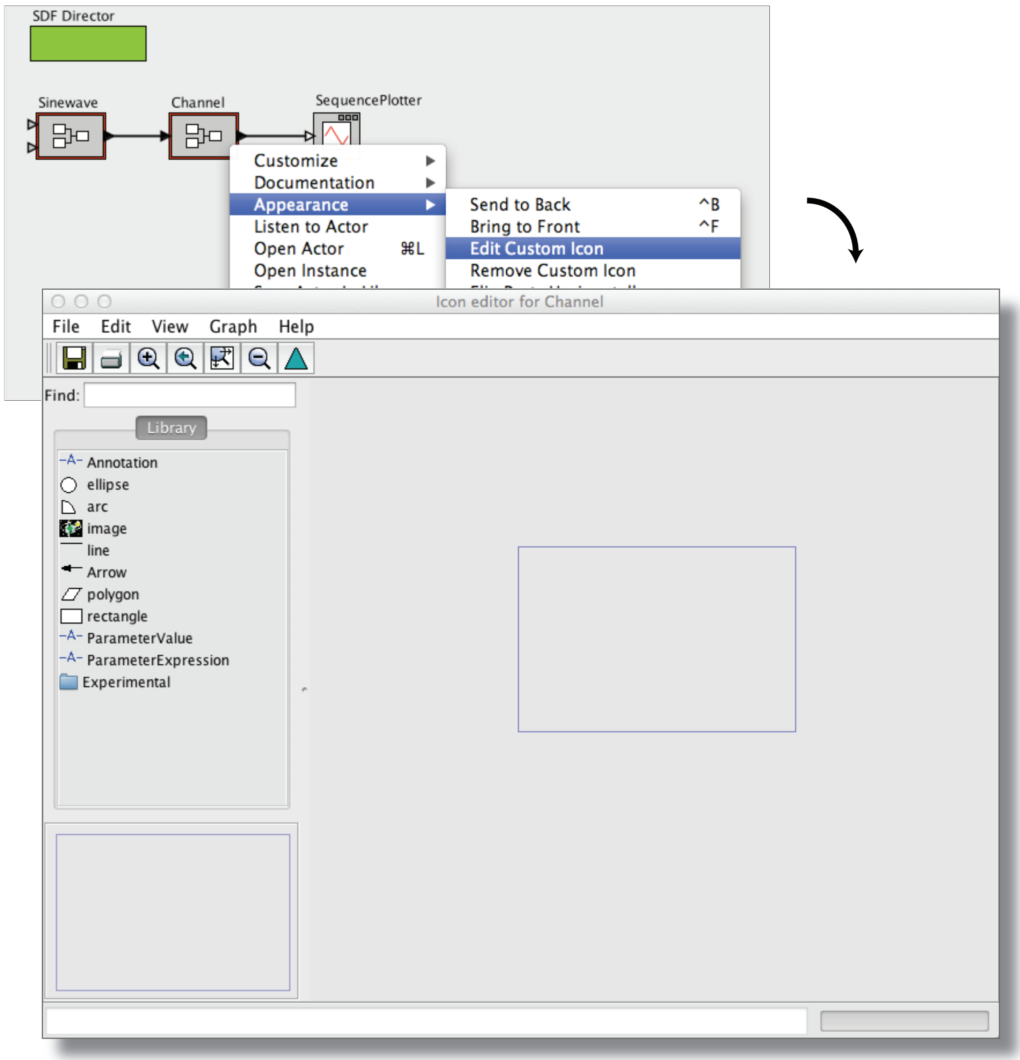


Figure 2.25: Custom icon editor for the Channel actor.

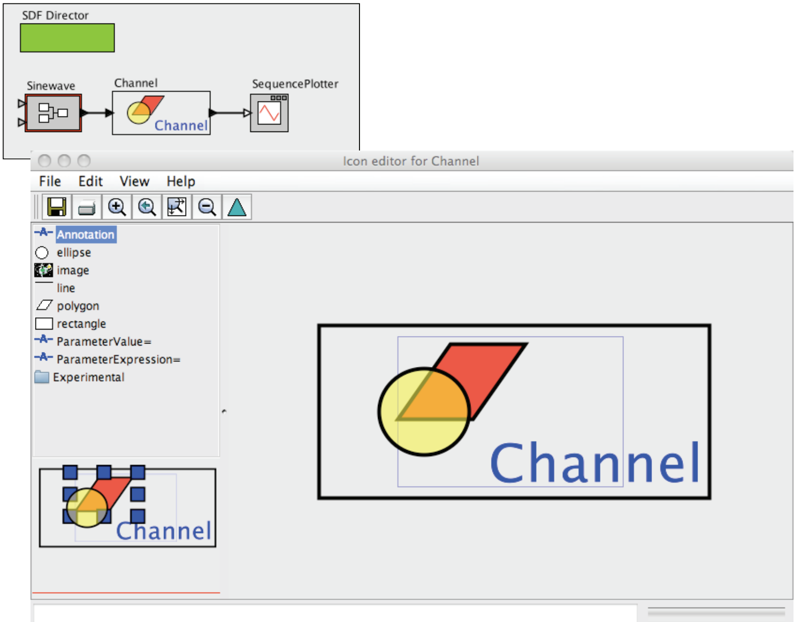


Figure 2.26: Custom icon for the Channel actor. [online]

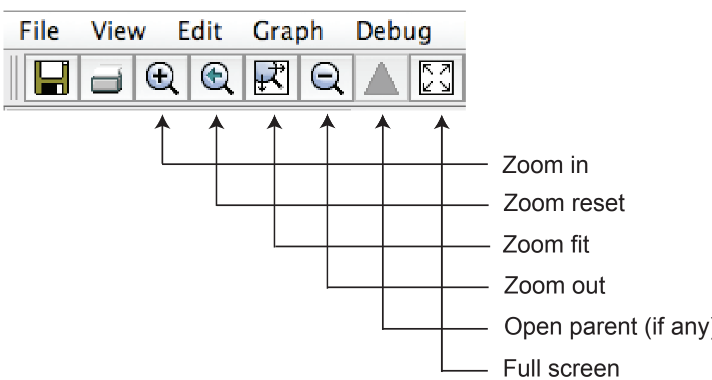


Figure 2.27: Summary of toolbar buttons for zooming and fitting.

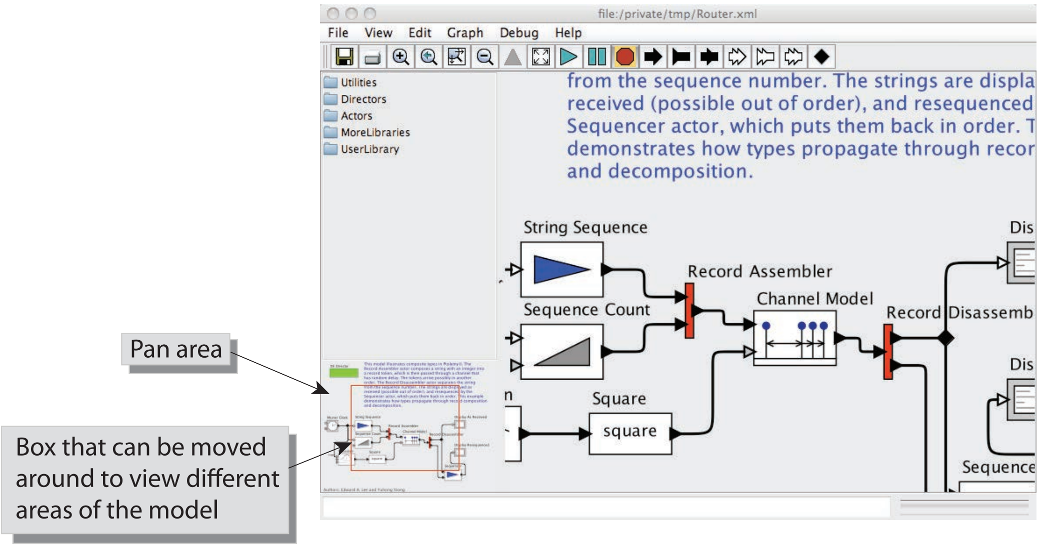


Figure 2.28: The pan window at the lower left has a red box representing the visible area of the model in the main editor window. This red box can be moved around to view different parts of the model.

2.6 Classes and Inheritance

Ptolemy II includes the ability to define **actor-oriented classes** (Lee et al., 2009a). These classes can then be used to create **instances** and **subclasses**, both of which use the concept of class **inheritance**. A class provides a general definition (or template) for an actor. An instance is a single, specific occurrence of that class, and a subclass is derived from the class — it includes the same structure, but may have some modifications. This approach improves design modularity, as illustrated in the example below.

Example 2.1: Consider the model that was developed in Section 2.3, shown for reference in Figure 2.29. Suppose that we wish to create multiple instances of the channel, as shown in Figure 2.30. In that figure, the sinewave signal passes through five distinct channels (note the use of a black-diamond relation between the sine wave and the channels to broadcast the same signal to each of the five channels). The outputs of the channels are added together and plotted. The result is a significantly cleaner sine wave than the one that results from one channel alone. (In communication systems, this technique is known as a diversity system, where multiple copies of a channel, each with independent noise, are used to achieve more reliable communications.)

Although it is functional, this is a poor design, for two reasons. First, the number of channels is hardwired into the diagram. (We will address that problem in the next section.) Second, each of the channels is a *copy* of the composite actor in Figure 2.29. Therefore, if the channel design needs to be changed, all five copies must be changed. This approach results in models that are difficult to maintain or scale.

A more subtle advantage to using classes and instances is that the XML file representation of the model will be smaller, since the design of the class is given only once rather than multiple times.

A better solution would be to define a Channel class, and use instances of that class to implement the diversity system. To implement this change, begin with the design in Figure 2.29, and remove the connections to the channel, as shown in Figure 2.31. Then right click and select `Convert to Class`. (Note that if you fail to first remove the connections, you will get an error message when you try to

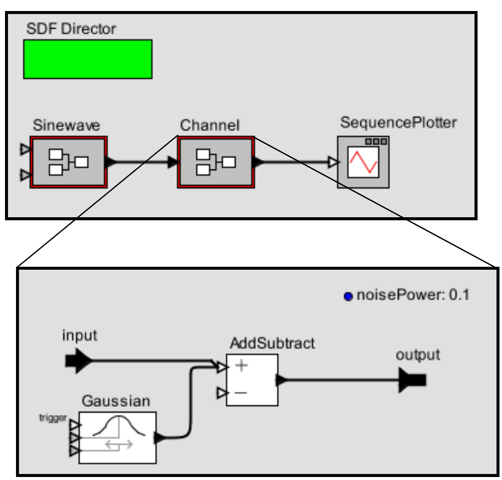


Figure 2.29: Hierarchical model that we will modify to use classes.

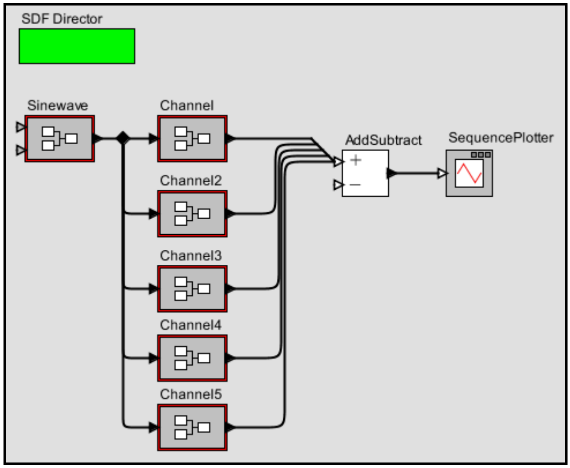


Figure 2.30: A poor design of a diversity communication system, which has multiple copies of the channel as defined in Figure 2.29. [online]

convert to class, as a class is not permitted to have connections.) The actor icon acquires a blue halo, which serves as a visual indication that it is a class rather than an ordinary actor (which is an instance).

Classes play no role in the execution of the model, and merely serve as definitions of components that must then be instantiated. By convention, we put classes at the top of the model, near the director, since they function as declarations.

Note also that instead of using `Convert to Class`, you can drag in an instance of **CompositeClassDefinition** from the `Utilities` library, and then select `Open Actor` and populate the class definition. You will want to give this class definition a more meaningful name, such as Channel.

Once you have defined a class, you can create an instance by right clicking and selecting `Create instance` or typing Control-N. Do this five times to create five instances of the class, as shown in Figure 2.31. Although this looks similar to the design in Figure 2.30, it is, in fact, a much better design, for the reasons described earlier. Try making a change to

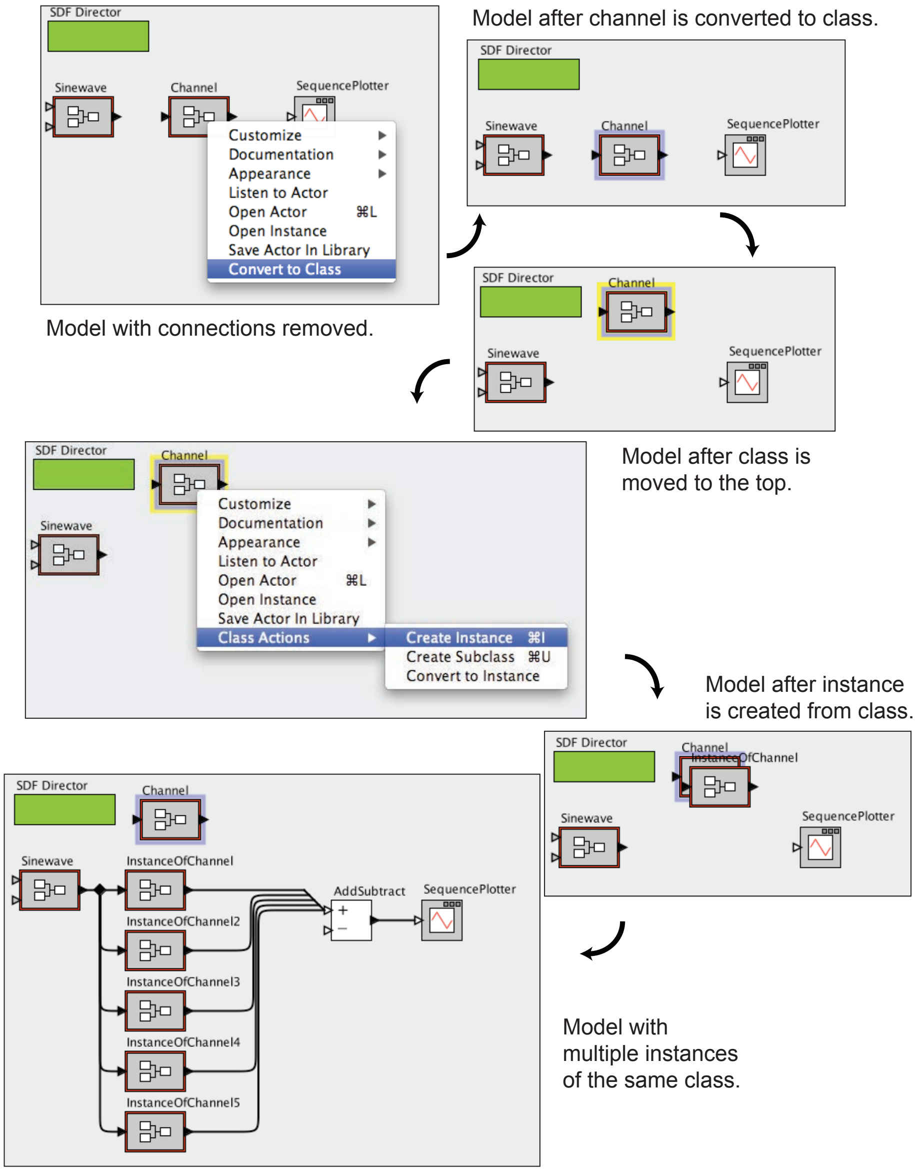


Figure 2.31: Creating and using a Channel class. [online]

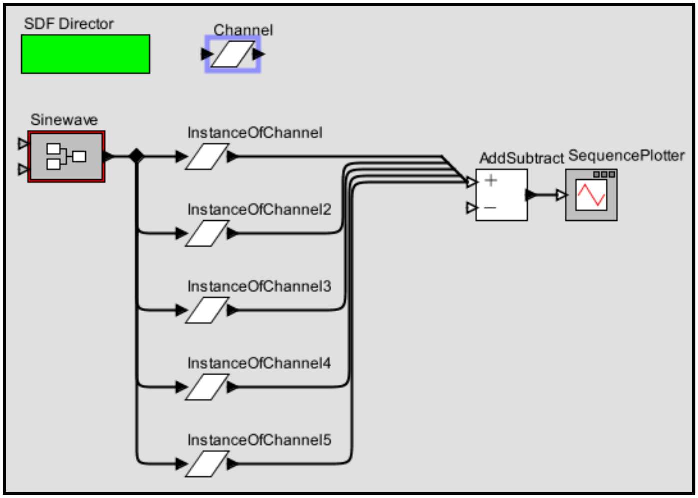


Figure 2.32: The model from Figure 2.31 with the icon changed for the class. Changes to the base class propagate to the instances. [online]

the class — for example, by creating a custom icon for it, as shown in Figure 2.32. Note that the changes propagate to each of the instances of the class.

If you invoke `Open Actor` on any of the instances (or the class) in Figure 2.32, you will see the same channel model. In fact, you will see the class definition. Any change you make inside this hierarchical model will be automatically propagated to all the instances. Try changing the value of the *noisePower* parameter, for example, and observe the result.

If you wish to view the instance rather than the class definition, you can select `Open Instance` on one of the instances. The window that opens shows only that instance. Each subcomponent that is inherited from the class definition is highlighted with a pink halo.

2.6.1 Overriding Parameter Values in Instances

By default, all instances of Channel in Figure 2.32 have the same icon and the same parameter values. However, each instance can be customized by overriding these values. In Figure 2.33, for example, we have modified the custom icons so that each has a different

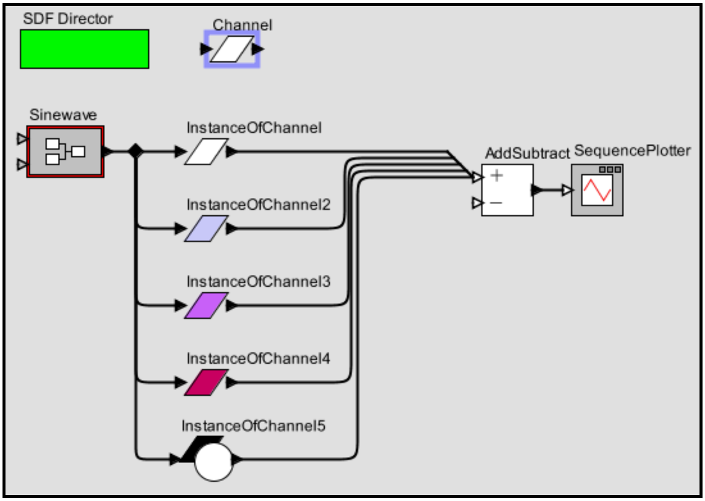


Figure 2.33: The model from Figure 2.32 with the icons of the instances changed to override parameter values in the class. [online]

color, and the fifth one has an extra graphical element. These changes are made by right clicking on the icon of the instance and selecting `Edit Custom Icon`. If you update class parameters that are overridden in an instance, then an update to the class will have no effect on the instance. It only has an effect on instances that have not been overridden.

2.6.2 Subclasses and Inheritance

Suppose now that we wish to modify some of the channels to add interference, in the form of another sine wave. A good way to do this is to create a subclass of the Channel class, as shown in Figure 2.34. A subclass is created by right clicking on the class icon and selecting `Create Subclass`. The resulting icon for the subclass appears on top of the icon for the class, and can be moved aside.

The subclass contains all of the elements of the class, but with the icons now surrounded by a pink halo. These elements are inherited and cannot be removed from the subclass (attempting to do so will generate an error message). You can, however, change their parameter values and add additional elements. Consider the design shown in Figure 2.35,

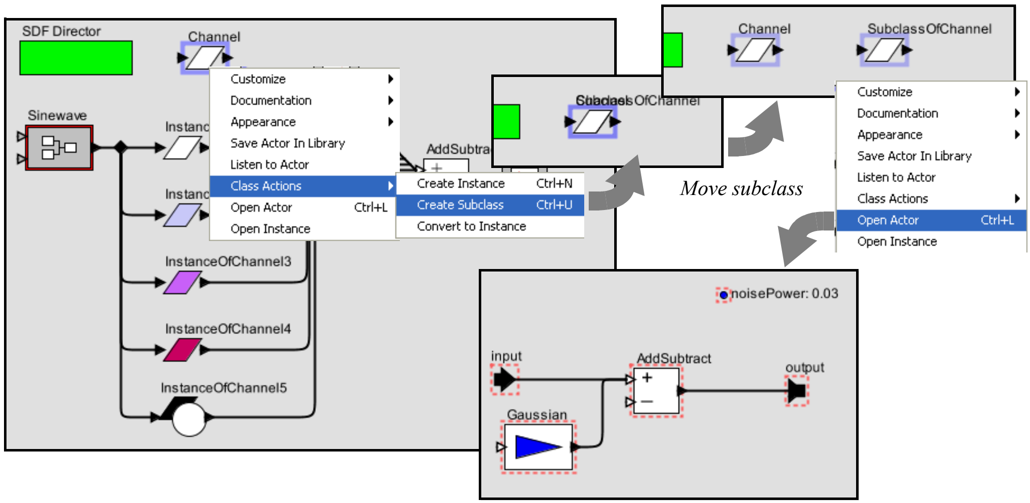


Figure 2.34: The model from figure 2.33 with a subclass of the Channel with no overrides (yet). [online]

which adds an additional pair of parameters named *interferenceAmplitude* and *interferenceFrequency* and an additional pair of actors implementing the interference.

A model that replaces the last channel with an instance of this subclass is shown in Figure 2.36, along with a plot showing the sinusoidal interference.

An instance of a class must be located in the same composite actor as the class itself, or in a composite actor that is itself contained in the model with the class itself (that is, in a submodel). To add an instance to a submodel, simply copy (or cut) an instance from the composite model that contains the class, and paste the instance into the submodel.

2.6.3 Sharing Classes Across Models

A class may be shared across multiple models by saving the class definition in its own file. We will illustrate this technique with the Channel class. First, right click and invoke `Open Actor` on the Channel class, and then select `Save As` from the `File` menu. The dialog that appears is shown in Figure 2.37. The check box labeled `Save submodel only` is

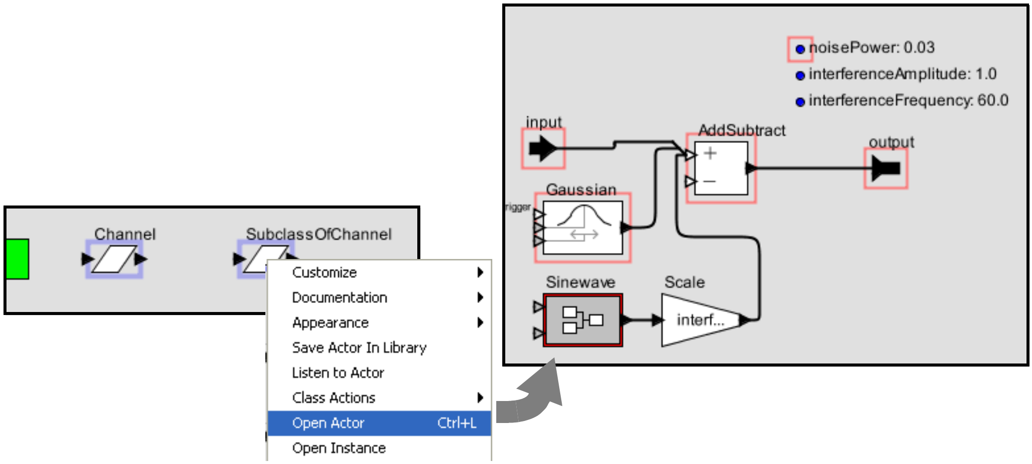


Figure 2.35: The subclass from Figure 2.34 with overrides that add sinusoidal interference. [online]

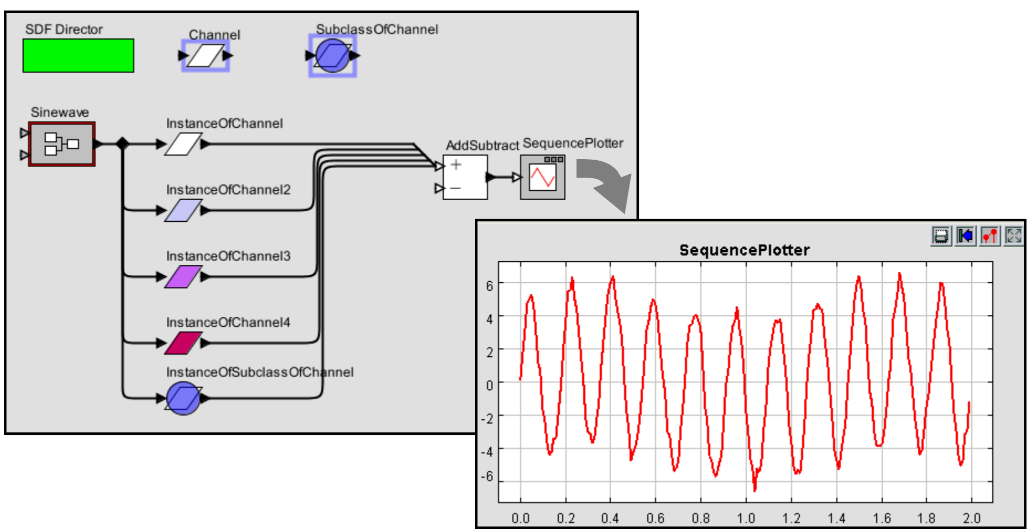


Figure 2.36: A model using the subclass from Figure 2.35 and a plot of an execution. [online]

by default unchecked. If left unchecked, the entire model will be saved.[6] In our case, we wish to save the Channel submodel only, so we must check the box.

It is important to save the class definition in a location that will be found by the model. In general, Ptolemy II searches for class definitions relative to the **classpath**, which is given by an environment variable called CLASSPATH. In principle, you can set this environment variable to include any directory that you would like to have searched. In practice, however, changing the CLASSPATH variable may cause problems with other programs, so we recommend that you store the file in a directory within the Ptolemy II installation directory or within a directory called ".ptolemyII" in your home directory.[7] In either case, Ptolemy will find class files stored in those directories.

Let's assume you save the Channel class to a file called `Channel.xml` in the directory `$PTII/myActors`, where `$PTII` is the location of the Ptolemy II installation. This class definition can now be used in any model, as follows. Open the model and select `Instantiate Entity` in the `Graph` menu, as shown in Figure 2.38. Enter the fully qualified class name relative to the `$PTII` entry in the classpath, which in this case is "`myActors.Channel`".

Once you have an instance of the Channel class that is defined in its own file, you can add it to the **UserLibrary** that appears in the library browser on the left side of Vergil windows, as shown in Figure 2.39. Right click on the instance and select `Save Actor in Library`. As shown in the figure, this causes another window to open and display the user library. The user library is itself a Ptolemy II model stored in an XML file. When you save the library model the class instance becomes available in the UserLibrary for any Vergil window (for the same user). Note that saving the class definition itself (vs. an instance of the class) in the user library does not have the same result. In that case, the user library would provide a new class definition rather than an instance of the class.

2.7 Higher-Order Components

Ptolemy II includes a number of **higher-order components**. These are actors that operate on the structure of the model rather than on input data. Consider the several examples

[6] On some platforms, a separate dialog will appear with the `Save submodel only` check box.

[7] If you don't know where Ptolemy II is installed on your system, you can invoke [`File→New→Expression Evaluator`] and type PTII followed by Enter. Or, in a Graph editor, select [`View→JVM Properties`] and look for the **ptolemy.ptII.dir** property.

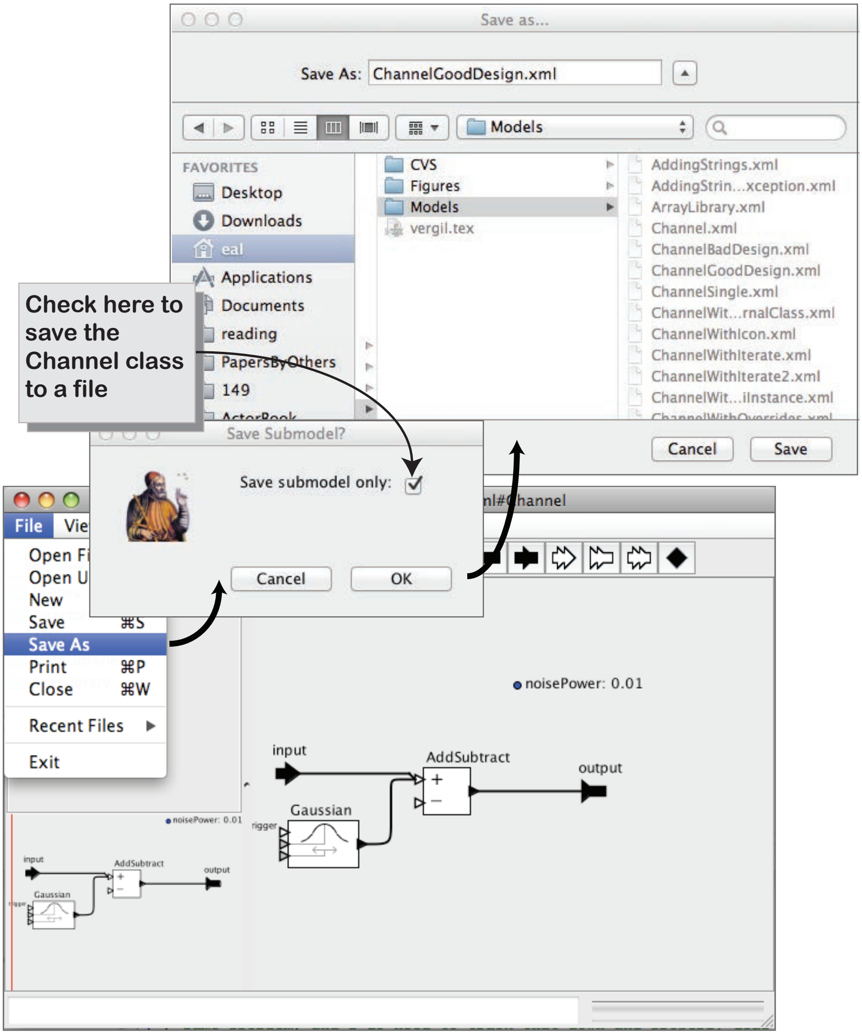


Figure 2.37: A class can be saved in a separate file to then be shared among multiple models. On some platforms, a separate dialog will appear with the `Save submodel only` check box, as shown. On others, the check box will be integrated into a single dialog. [online]

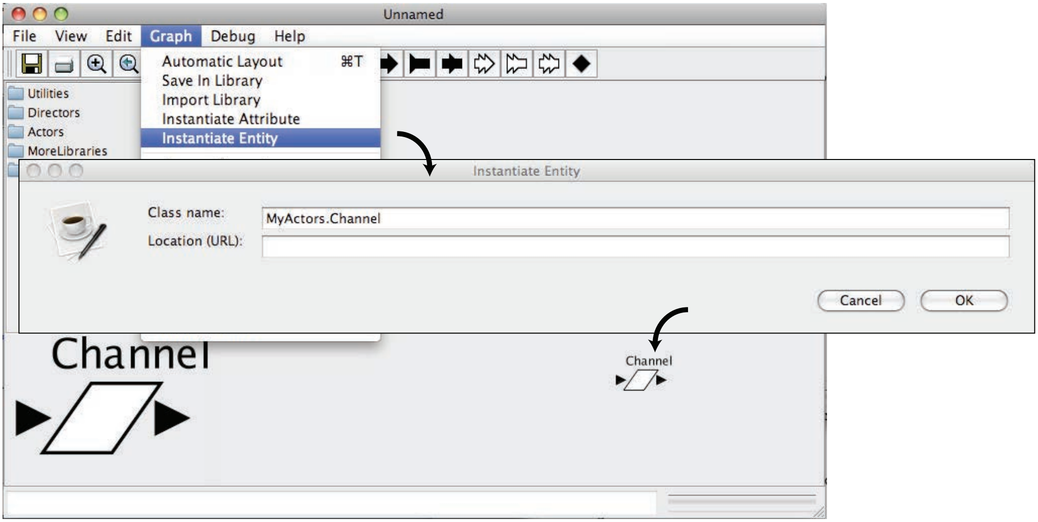


Figure 2.38: An instance of a class defined in a file can be created using `Instantiate Entity` in the `Graph` menu. [online]

above where five instances of the Channel actor are put into a model. Why five? Perhaps it would be better to have a single component that represents n instances of Channel, where n is a variable. This is exactly the sort of capability provided by higher-order actors. Higher-order actors, which were introduced by Lee and Parks (1995), make it easier to build large designs when the model structure does not depend on the scale of the problem. In this section, we describe a few of these actors, all of which are found in the `HigherOrderActors` library.

2.7.1 MultiInstance Composite

Consider the earlier model shown in Figure 2.32, which has five instances of the Channel class connected in parallel. The number of instances is hardwired into the diagram, and it is awkward to change this number, particularly if it needs to be increased. This problem can be solved by using the **MultiInstanceComposite** actor,[8] as shown in Figure 2.40. The MultiInstanceComposite is a composite actor into which we have inserted a single

[8]Contributed by Zoltan Kemenczy and Sean Simmons, of Research In Motion Limited.

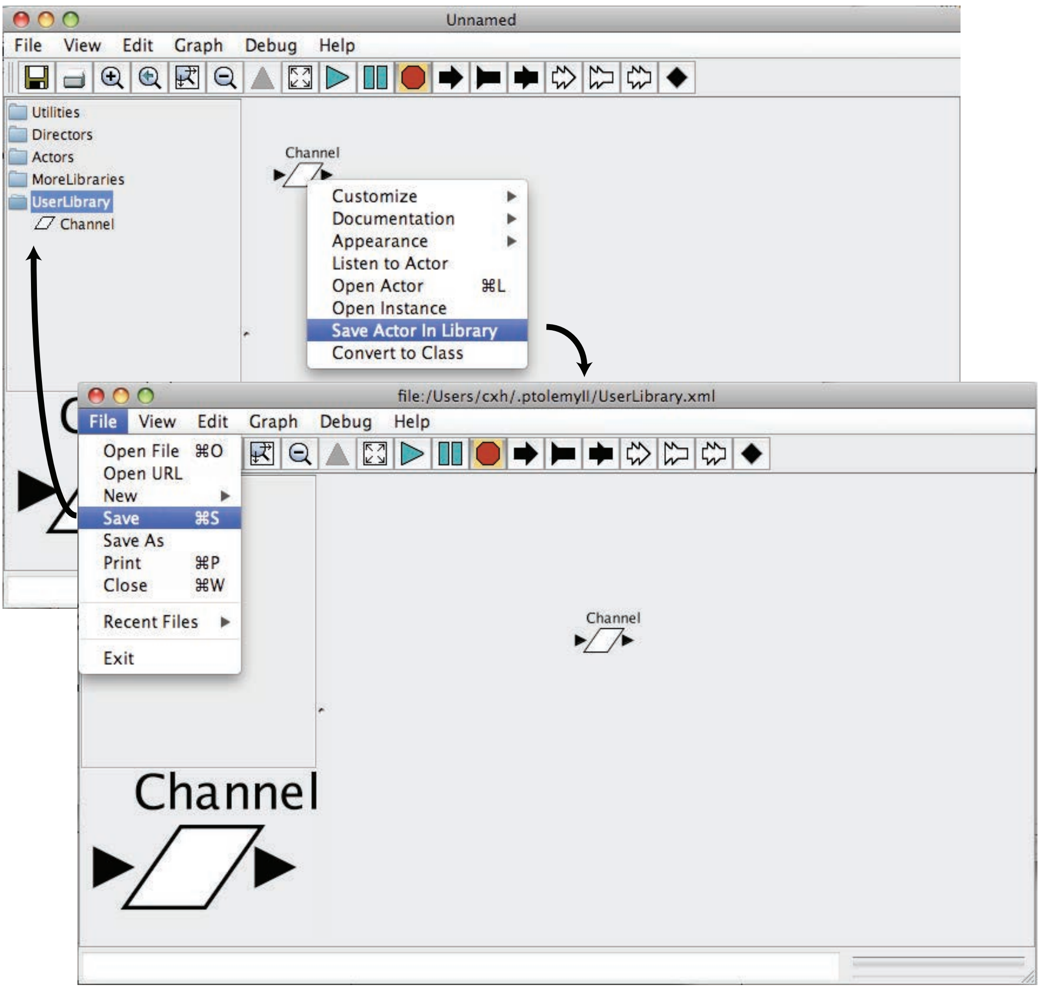


Figure 2.39: Instances of a class that are defined in their own files can be made available in the UserLibrary.

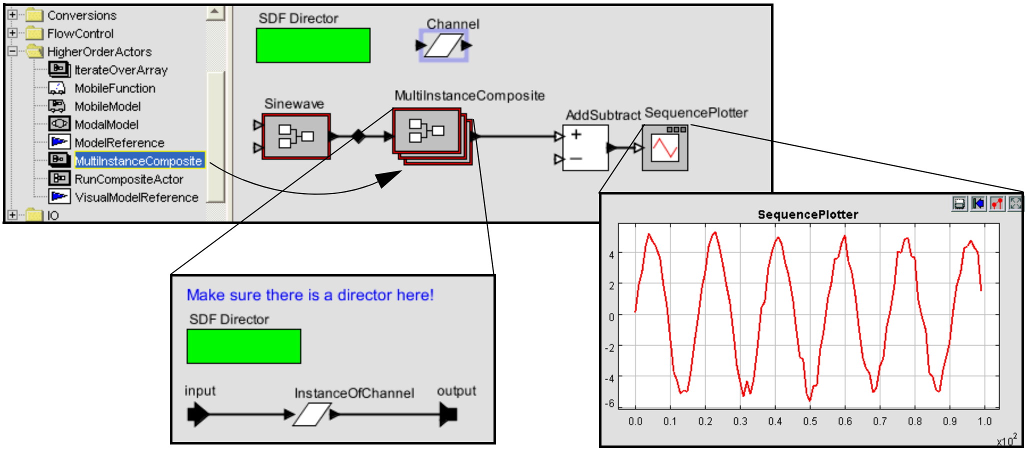


Figure 2.40: A model that is equivalent to that of Figure 2.32, but using a MultiInstanceComposite, which permits the number of instances of the channel to change by simply changing one parameter value. [online]

instance of the Channel (by creating an instance of the Channel, then copying and pasting it into the composite). MultiInstanceComposite is required to be opaque (meaning that it contains a director). It functions within the model as a single actor, but internally it is realized as a multiplicity of actors operating in parallel.

The MultiInstanceComposite actor has three parameters, *nInstances*, *instance*, and *showClones*, as shown in Figure 2.41. The first of these specifies the number of instances to create. At run time, this actor replicates itself this number of times, connecting the inputs and outputs to the same sources and destinations as the first (prototype) instance. In Figure 2.40, notice that the input of the MultiInstanceComposite is connected to a relation (the black diamond), and the output is connected directly to a multiport input of the AddSubtract actor. As a consequence, the multiple instances will be connected in a manner similar to Figure 2.32, where the same input value is broadcast to all instances, but distinct output values are supplied to the AddSubtract actor.

The model created using multi-instances is better than the original version because the number of instances can be changed with a single parameter. Each instance can be customized as needed by expressing its parameter values in terms of the *instance* parameter of the MultiInstanceComposite. Try, for example, making the *noisePower* parameter of

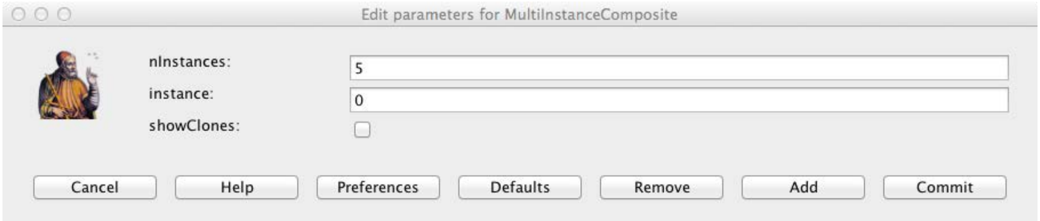


Figure 2.41: The first parameter of the MultiInstanceComposite specifies the number of instances. The second parameter is available to the model builder to identify individual instances. The third parameter controls whether the instances are rendered on the screen (when the model is run).

the InstanceOfChannel actor in Figure 2.40 depend on *instance*. For example, set it to `instance * 0.1` and then set *nInstances* to 1. You will see a clean sine wave when you run the model, because the one instance has number zero, so there will be no noise for that instance.

2.7.2 IterateOverArray

The implementation of the Channel class, which is shown in Figure 2.37, does not have any state, meaning that an invocation of the Channel model does not depend on data calculated in a previous invocation. As a consequence, it is not necessary to use *n* distinct instances of the Channel class to realize a diversity communication system; a single instance could be invoked n times on n copies of the data. This approach can be accomplished by using the **IterateOverArray** higher-order actor.

The IterateOverArray actor can be used in a manner similar to the MultiInstanceComposite in the previous section. That is, we can populate it with an instance of the Channel class, similar to Figure 2.40. The IterateOverArray actor also requires a director inside the model.

Example 2.2: Consider the example in Figure 2.42. In this case, the top-level model uses an array with multiple copies of the channel input rather than using a relation to broadcast the input to multiple instances of the channel.

This is accomplished using a combination of the Repeat actor (found in the `FlowControl→SequenceControl` sublibrary) and the SequenceToArray actor (see box on page 86). The Repeat actor has a parameter, *numberOfTimes*, which in Figure 2.42 is set equal to the *diversity* parameter. The SequenceToArray actor has a parameter *arrayLength* that has also been set to be equal to *diversity* (this parameter can also be set via the *arrayLength* port, which is filled in gray to indicate that it is both a parameter and a port). The output is sent to an ArrayAverage actor (see box on page 88).

The execution of the model in Figure 2.42 is similar to that of the earlier version, except that the scale of the output is different, reflecting the fact that the output is an average rather than a sum.

The IterateOverArray actor simply executes whatever actor it contains repeatedly on each element of an input array. The actor that it contains can be, as shown in Figure 2.42, an opaque composite actor. Interestingly, however, it can also be an atomic actor. To

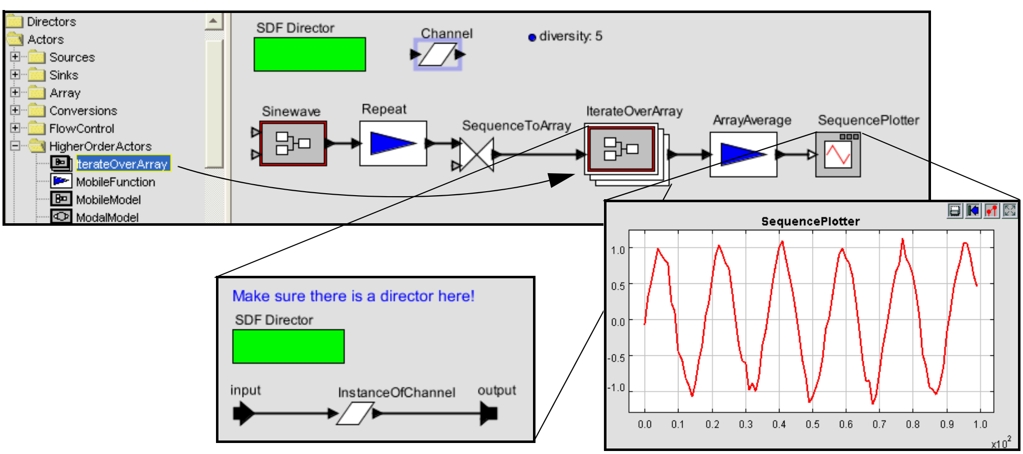


Figure 2.42: The IterateOverArray actor can be used to accomplish the same diversity channel model as in Figure 2.40, but without creating multiple instances of the channel model. This approach is possible because the channel model has no state. [online]

use an atomic actor with IterateOverArray, simply drag and drop the atomic actor onto an instance of IterateOverArray. It will then execute that atomic actor on each element of the input array, and produce as output the array of the results. This mechanism is illustrated in Figure 2.43. When an actor is dragged from the library and moved over the IterateOverArray actor, the icon acquires a white halo. The halo indicates that if the actor is dropped, it will be dropped into the actor under the cursor, rather than onto the model containing that actor. The actor you drop onto the IterateOverArray will become the actor that is executed for each element of the input array. In order for this to work with the Channel actor we defined above, however, we need to convert the Channel actor into an opaque actor by inserting a director, because IterateOverArray can only apply opaque actors to array elements.

2.7.3 Lifecycle Management Actors

A few actors in `HigherOrderActors` invoke the execution of a full Ptolemy II model. These actors generally associate ports (created by the user or actor) with parameters of the model. They can be used, for example, to create models that repeatedly run other mod-

Sidebar: Array Construction and Deconstruction Actors

The following actors construct and take apart arrays:

- **ArrayToElements** outputs the elements of an array on channels of the output port.
- **ElementsToArray** constructs an array from elements on channels of the input port.
- **ArrayToSequence** outputs the elements of an array sequentially on the output port.
- **SequenceToArray** constructs an array from a sequence of elements on the input port.
- **StringToUnsignedByteArray** constructs an array from a string.
- **UnsignedByteArrayToString** constructs a string from an array.
- **ArrayToMatrix** constructs a matrix from an array.
- **MatrixToArray** constructs an array from a matrix.

In addition, many polymorphic actors, such as AddSubtract, also operate on arrays.

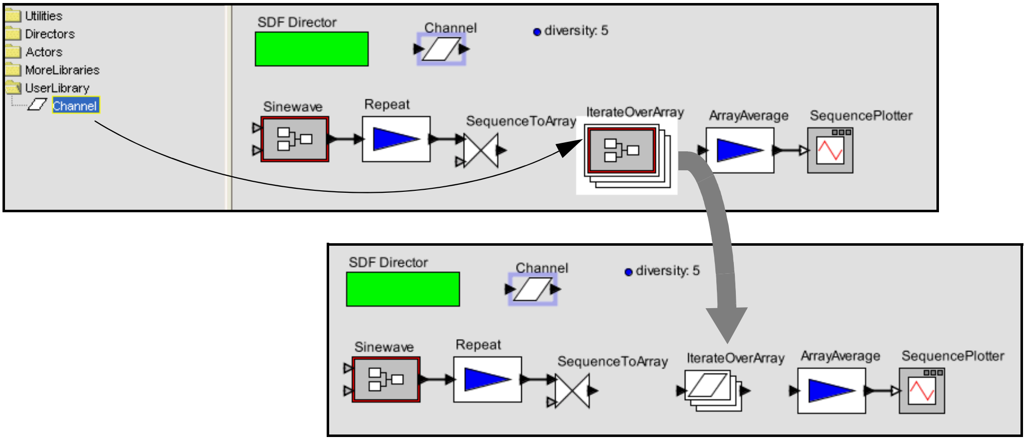


Figure 2.43: The IterateOverArray actor supports dropping an actor onto it. It transforms to mimic the icon of the actor you dropped onto it, as shown. Here we are using the Channel class, saved to the UserLibrary as shown in Figure 2.39.

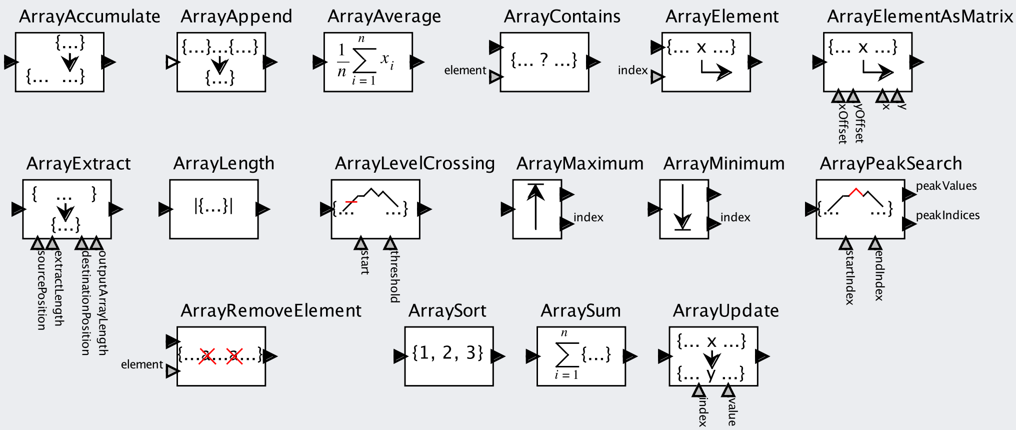
els with varying parameter values. These include **RunCompositeActor**, which executes the model that it contains. The **ModelReference** actor executes a model that is defined elsewhere in its own file or URL. The **VisualModelReference** actor opens a Vergil view of a model when it executes that model. Additional details can be found in the actor documentation and in the Vergil tour demonstrations.

2.8 Summary

This chapter has introduced Vergil, the visual interface to Ptolemy II, which supports graphical construction of models. Along the way, this chapter has introduced a number of capabilities of the underlying Ptolemy II system. Subsequent chapters will focus on properties of the various directors that are available. The appendices then focus on generic architecture and capabilities that span models of computation.

Sidebar: Array Manipulation Actors

The following actors operate on arrays:



- **ArrayAccumulate** appends input arrays, growing the output array.
- **ArrayAppend** appends input arrays provided on channels of a multiport.
- **ArrayAverage** averages the elements of an array.
- **ArrayContains** determines whether an array contains a specified element.
- **ArrayElement** extracts an element from an array.
- **ArrayElementAsMatrix** extracts an element using matrix-like indexing.
- **ArrayExtract** extracts a subarray.
- **ArrayLength** outputs the length of the input array.
- **ArrayLevelCrossing** finds an element that crosses a threshold.
- **ArrayMaximum** finds the largest element of an array.
- **ArrayMinimum** finds the smallest element of an array.
- **ArrayPeakSearch** finds peak values in an array.
- **ArrayRemoveElement** removes instances of a specified element.
- **ArraySort** sorts an array.
- **ArraySum** sums the elements of an array.
- **ArrayUpdate** outputs a new array like the input array, but with an element replaced.

In addition, many polymorphic actors, such as AddSubtract, also operate on arrays.

Sidebar: Mobile Code

A pair of actors in Ptolemy II support mobile models, where the data sent from one actor to another is a model to be executed rather than data on which a model operates. The **ApplyFunction** actor accepts a function in the expression language (see Chapter 13) at one input port and applies that function to data that arrives at other input ports (which you must create). The **MobileModel** actor accepts a MoML description of a Ptolemy II model at an input port and then executes that model, streaming data from the other input port through it.

A use of the ApplyFunction actor is shown in Figure 2.44. In that model, two functions are provided to the ApplyFunction in an alternating fashion, one that computes x^2 and the other that computes 2^x. These two functions are provided by two instances of the Const actor, found in the `Sources→GenericSources` sublibrary. The functions are interleaved by the Commutator actor, from the `FlowControl→Aggregators` sublibrary.

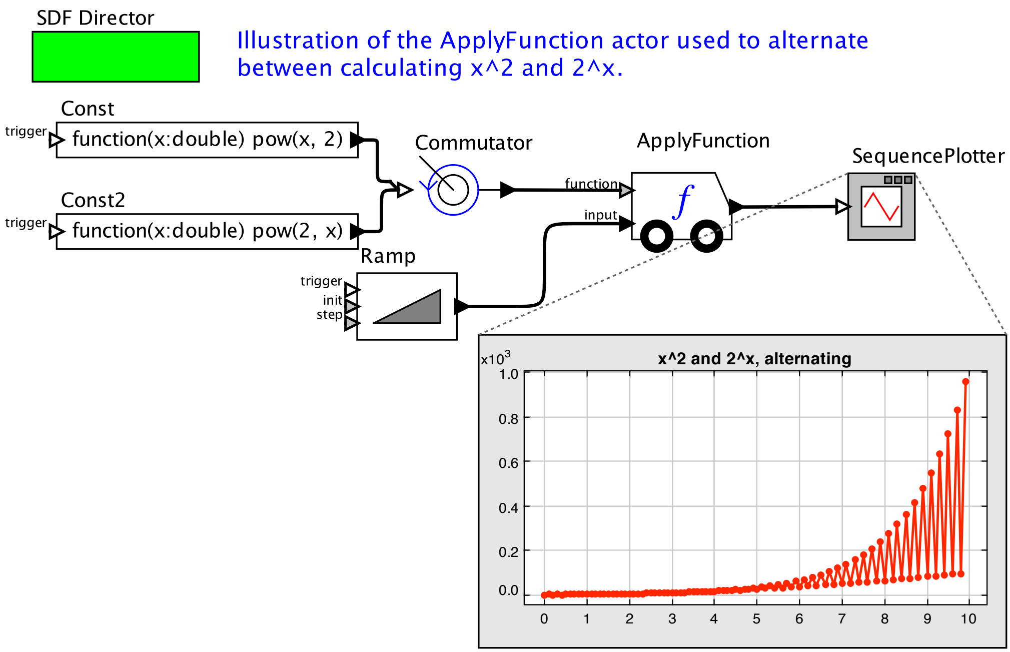


Figure 2.44: The ApplyFunction actor accepts a function definition at one port and applies it to data that arrives at the other port. [online]

Part II

Models of Computation

This part of this book introduces a few of the models of computation used in system design, modeling, and simulation. This is not a comprehensive set, but rather a representative set. The particular MoCs described here are relatively mature, well understood, and fully implemented. Most of these are available in other tools besides Ptolemy II, though no other tool provides all of them.

3 Dataflow

Edward A. Lee, Stephen Neuendorffer, Gang Zhou

Contents

Ptolemy II was created to enable heterogeneous models to be developed and simulated together as part of an overall system model. As discussed in previous chapters, a key innovation in Ptolemy II is that, unlike other design and modeling environments, Ptolemy II supports multiple models of computation that are tailored to specific types of modeling problems. These models of computation define how the model will behave and are determined by the director that is used within that model. In Ptolemy II terminology, the director realizes a domain, which is an implementation of a model of computation. Thus, the director, domain, and model of computation are all tied together; when you construct a model that contains an SDFDirector (a synchronous dataflow director), for example, you have constructed a model "in the SDF domain," using the SDF model of computation.

This chapter describes the **dataflow domains** that are currently available in Ptolemy II, which include synchronous (static) and dynamic dataflow models. Dataflow domains are appropriate for applications that involve processing streams of data values. These streams can flow through sequences of actors that transform them in some way. Such models are often called **pipe and filter** models, because the connections between actor are analogous to pipes that carry flows, and the actors are analogous to filters the change the flows in some way. Dataflow domains mostly ignore time, although SDF is capable of modeling streams with uniformly spaced time between iterations. Subsequent chapters discuss other domains and their selection and use.

3.1 Synchronous Dataflow

The **Synchronous dataflow** (**SDF**) domain, also called **static dataflow**,[1] was introduced by Lee and Messerschmitt (1987b), and is one of the first domains (or models of computation) developed for Ptolemy II. It is a specific type of **dataflow** model. In dataflow models, actors begin execution (they are fired) when their required data inputs become available. SDF is a relatively simple case of dataflow; the order in which actors are exe-

[1]The term "synchronous dataflow" can cause confusion because it is not synchronous in the sense of SR, considered in Chapter 5. There is no global clock in SDF models, and actors are fired asynchronously. For this reason, some authors prefer the term "static dataflow." This does not avoid all confusion, however, because Dennis (1974) had previously coined the term "static dataflow" to refer to dataflow graphs where buffers could hold at most one token. Since there is no way to avoid a collision of terminology, we stick with the original "synchronous dataflow" terminology used in the literature. The term SDF arose from a signal processing concept, where two signals with sample rates that are related by a rational multiple are deemed to be synchronous.

cuted is static, and does not depend on the data that is processed (the values of the tokens that are passed between actors).

In a **homogeneous SDF** model, an actor fires when there is a token on each of its input ports and produces a token on each output port. In this case, the director simply has to ensure that each actor fires after the actors that supply it with data, and an iteration of the model consists of one firing of each actor. Most of the examples in Chapter 2 were homogeneous SDF models.

Not all actors produce and consume a single token each time they are fired, however; some require multiple input tokens before they can be fired and produce multiple output tokens. The SDF scheduler, which is responsible for determining the order in which actors are executed, supports more complex models than homogeneous SDF. It is capable of scheduling the execution of actors with arbitrary data rates, as long as these rates are given by specifying the number of tokens consumed and produced by the firing of each actor on each port.

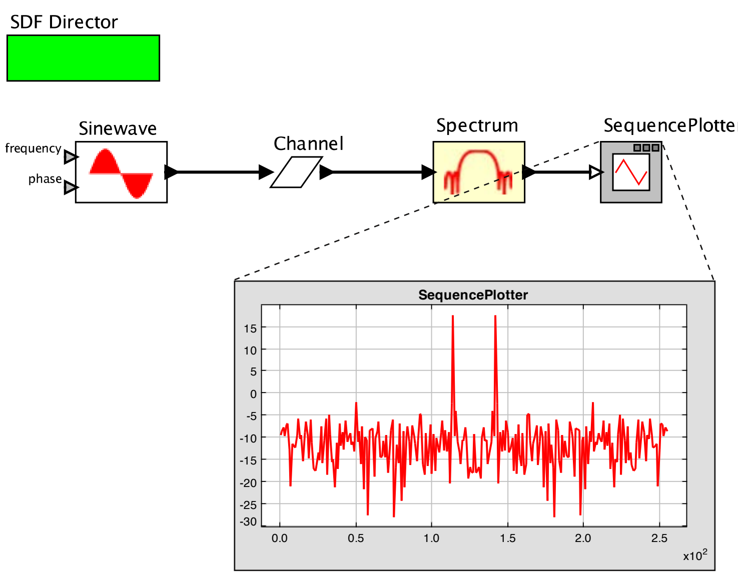


Figure 3.1: A multirate SDF model. The Spectrum actor requires 256 tokens to fire, so one iteration of this model requires 256 firings of Sinewave, Channel, and SequencePlotter, and one firing of Spectrum. [online]

Example 3.1: One example of an actor that requires multiple input tokens to fire is the Spectrum actor (see box on page 101). Figure 3.1 shows a system that computes the spectrum of the same noisy sine wave that we constructed in Figure 2.20. The Spectrum actor has a single parameter that specifies the order of the fast Fourier transform (**FFT**) used to calculate the spectrum. Figure 3.1 shows the output of the model with *order* set to 8 and the number of iterations set to 1. (See Chapter 17, Section 17.2 to improve the labeling of the plot.)

When the *order* parameter is set to 8, the Spectrum actor requires 2^8 (256) input samples to fire, and produces 2^8 output samples. In order for the Spectrum actor to fire once, the actors that supply its input data, Sinewave and Channel, must each fire 256 times. The SDF director extracts this relationship and defines one iteration of the model to consist of 256 firings of Sinewave, Channel, and SequencePlotter, and one firing of Spectrum.

This example implements a **multirate** model; that is, the firing rates of the actors are not identical. In particular, the Spectrum actor executes at a different rate than the other actors. It is common for the execution of a multirate model to consist of exactly one iteration. The director determines how many times to fire each actor in an iteration using balance equations, as described in the next section.

3.1.1 Balance Equations

Consider a single connection between two actors, A and B, as shown in Figure 3.2. The notation here means that when A fires, it produces M tokens on its output port, and when B fires, it consumes N tokens on its input port. M and N are nonnegative integers. Suppose that A fires q_A times and B fires q_B times. All tokens that A produces are

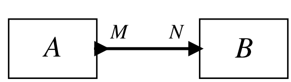


Figure 3.2: SDF actor A produces M tokens when it fires, and actor B consumes N tokens when it fires.

consumed by B if and only if the following **balance equation** is satisfied,

$$q_A M = q_B N. \tag{3.1}$$

Given values q_A and q_B satisfying (3.1), the system remains in balance; A produces exactly as many tokens as B consumes.

Suppose we wish to process an arbitrarily large number of tokens, a situation that is typical of streaming applications. A naive strategy is to fire actor A an arbitrarily large number q_A times, and then fire actor B q_B times, where q_A and q_B satisfy (3.1). This strategy is naive, however, because it requires storing an arbitrarily large number of unconsumed tokens in a buffer. A better strategy is to find the smallest positive q_A and q_B that satisfy (3.1). Then we can construct a schedule that fires actor A q_A times and actor B q_B times, and we can repeat this schedule as many times as we like without requiring any more memory to store unconsumed tokens. That is, we can achieve an **unbounded execution** (an execution processes an arbitrarily large number of tokens) with **bounded buffers** (buffers with a bound on the number of unconsumed tokens). In each round of the schedule, called an iteration, actor B consumes exactly as many tokens as actor A produces.

Example 3.2: Suppose that in Figure 3.2, $M = 2$ and $N = 3$. There are many possible solutions to the corresponding balance equation, one of which is $q_A = 3$ and $q_B = 2$. With these values, the following schedule can be repeated forever:

$$A, A, A, B, B.$$

An alternative schedule could also be used:

$$A, A, B, A, B.$$

In fact, the latter schedule has an advantage in that it requires less memory for storing intermediate tokens; B fires as soon as there are enough tokens, rather than waiting for A to complete its entire cycle.

Another solution to (3.1) is $q_A = 6$ and $q_B = 4$. This solution includes more firings in the schedule than are strictly needed to keep the system in balance.

The equation is also satisfied by $q_A = 0$ and $q_B = 0$, but if the number of firings of actors is zero, then no useful work is done. Clearly, this is not a solution we want. Negative solutions are also not meaningful.

The SDF director, by default, finds the least positive integer solution to the balance equations, and constructs a schedule that fires the actors in the model the requisite number of times, given by this solution. An execution sequence that fires the actors exactly as many times as specified by this solution is called a **complete iteration**.

In a more complicated SDF model, every connection between actors results in a balance equation. Hence, the model defines a system of equations, and finding the least positive integer solution is not entirely trivial.

Example 3.3: Figure 3.3 shows a network with three SDF actors. The connections result in the following system of balance equations:

$$\begin{aligned} q_A &= q_B \\ 2q_B &= q_C \\ 2q_A &= q_C. \end{aligned}$$

The least positive integer solution to these equations is $q_A = q_B = 1$, and $q_C = 2$, so the following schedule can be repeated forever to get an unbounded execution with bounded buffers,

$$A, B, C, C.$$

The balance equations do not always have a non-trivial solution, as illustrated in the following example.

Example 3.4: Figure 3.4 shows a network with three SDF actors where the only solution to the balance equations is the trivial one, $q_A = q_B = q_C = 0$. A conse-

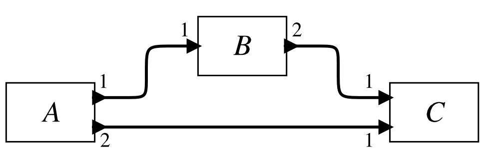


Figure 3.3: A consistent SDF model.

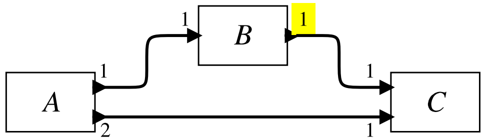


Figure 3.4: An inconsistent SDF model.

quence is that there is no unbounded execution with bounded buffers for this model. It cannot be kept in balance.

An SDF model that has a non-zero solution to the balance equations is said to be **consistent**. If the only solution is zero, then it is **inconsistent**. An inconsistent model has no unbounded execution with bounded buffers.

Lee and Messerschmitt (1987b) showed that if the balance equations have a non-zero solution, then they also have a solution where q_i is a nonnegative integer for all actors i. Moreover, for connected models (where there is a communication path between any two actors), they give a procedure for finding the least positive integer solution. Such a procedure forms the foundation for a scheduler for SDF models.

Example 3.5: Figure 3.5 shows an SDF model that makes extensive use of the multirate capabilities of SDF. The **AudioCapture** actor captures sound from the microphone on the machine on which the models run, producing a sequence of samples at a default rate of 8,000 samples per second. The Chop actor extracts chunks of 128 samples from each input block of 500 samples (see box on page 106). The Spectrum actor computes the power spectrum, which measures the power as a function of frequency (see box on page 101). The two SequenceToArray actors (box on page 106) construct arrays that are then plotted using ArrayPlotter actors (see Chapter 17). The particular plots that are shown are the response to a whistle. Notice the peaks in the spectrum at about 1,700 Hz and -1,700 Hz.

The SDF director in this model figures out that the AudioCapture actor needs to fire 500 times for each firing of Chop, Spectrum, and SequenceToArray, and the plotters.

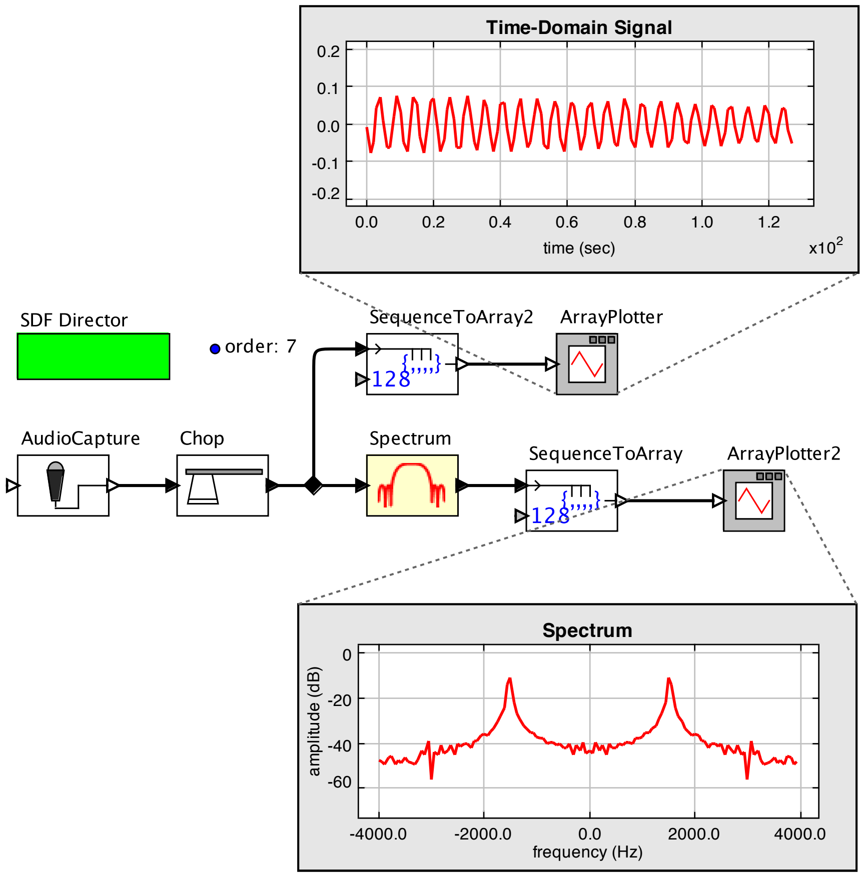


Figure 3.5: Model that computes the power spectrum of the audio signal captured from the microphone. The plots here show a whistle at about 1,700 Hz. [online]

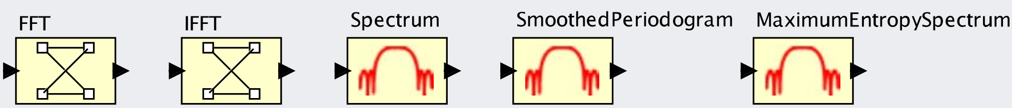
Sidebar: SDF Schedulers

A key advantage of using SDF is that there may be many possible schedules for a given model, including some that execute actors in parallel. In this case, actors in the dataflow graph can be mapped onto different processors in a multicore or distributed architecture for improved performance. Lee and Messerschmitt (1987a) adapt classical job-shop scheduling algorithms (Coffman, 1976), particularly those introduced by Hu (1961), to SDF by converting the SDF graph into an acyclic precedence graph (**APG**). Lee and Ha (1989) classify scheduling strategies into **fully dynamic scheduling** (all scheduling decisions are made at run time), **static assignment scheduling** (all decisions except the assignment to processors are made at run time), **self-timed scheduling** (only the timing of an actor firing is determined at run time), and **fully-static scheduling** (every aspect of the schedule is determined before run time). Sih and Lee (1993a) extend the job-shop scheduling techniques to account for interprocessor communication costs (see also Sih and Lee (1993b)). Pino et al. (1994) show how to construct schedules for heterogeneous multiprocessors. Falk et al. (2008) give a parallel scheduling strategy based on clustering and demonstrate significant performance gains for multimedia applications.

In addition to parallel scheduling strategies, other scheduling optimizations are also useful (see Bhattacharyya et al. (1996b) for a collection of these). Ha and Lee (1991) relax the constraints of SDF to allow data-dependent iterative firing of actors (a technique called **quasi-static scheduling**). Bhattacharyya and Lee (1993) develop optimized schedules for iterated invocations of actors (see also Bhattacharyya et al. (1996a)). Bhattacharyya et al. (1993) optimize schedules to minimize memory usage and later apply these optimizations to code generation for embedded processors (Bhattacharyya et al., 1995). Murthy and Bhattacharyya (2006) collect algorithms that minimize the use of memory through scheduling and buffer sharing. Geilen et al. (2005) show that model checking techniques can be used to optimize memory. Stuijk et al. (2008) explore the tradeoff between throughput and buffering (see also Moreira et al. (2010)). Sriram and Bhattacharyya (2009) develop scheduling optimizations that minimize the number of synchronization operations in parallel SDF. Synchronization ensures that an actor does not fire before it receives the data it needs to fire. However, synchronization is not required if a previous synchronization already provides assurance that the data are present. By manipulating the schedule, one can minimize the number of required synchronization points.

Sidebar: Frequency Analysis

The SDF domain is particularly useful for signal processing. One of the basic operations in signal processing is to convert a time domain signal into a frequency domain signal and vice versa (see Lee and Varaiya (2011)). Actors that support this operation are found in the `Actors→SignalProcessing→Spectrum` library, and shown below:



- **FFT** and **IFFT** calculate the discrete Fourier transform (DFT) and its inverse, respectively, of a signal using the fast Fourier transform algorithm. The *order* parameter specifies the number of input tokens that are used for each FFT calculation. It is a "radix two" algorithm, which implies that the number of tokens is required to be a power of two, and the *order* parameter specifies the exponent. For example, if *order*=10, then the number of input tokens used for each firing is $2^{10} = 1024$. The remaining actors implement various spectral estimation algorithms, and are all composite actors that use FFT as a component. These algorithms output signal power in decibels (dB) as a function of frequency. The output frequency ranges from $-f_N$ to f_N, where f_N is the Nyquist frequency (half the sampling frequency). That is, the first half of the output represents negative frequencies and the second half represents positive frequencies.
- **Spectrum** is the simplest of the spectral estimators. It calculates the FFT of the input signal and converts the result to a power measurement in dB.
- **SmoothedPeriodogram** calculates a power spectrum by first estimating the autocorrelation of the input. This approach averages the inputs and is less sensitive to noise.
- **MaximumEntropySpectrum** is a parametric spectral estimator; it uses the Levinson-Durbin algorithm to construct the parameters of autoregressive (AR) models that could plausibly have generated the input signal. It then selects the model that maximizes the entropy (see Kay (1988)). It is the most sophisticated of the spectral estimators and typically produces the smoothest estimates.

Outputs from the three spectral estimators are compared in Figure 3.6, where the input consists of three sinusoids in noise. Choosing the right spectral estimator for an application is a sophisticated topic, beyond the scope of this book.

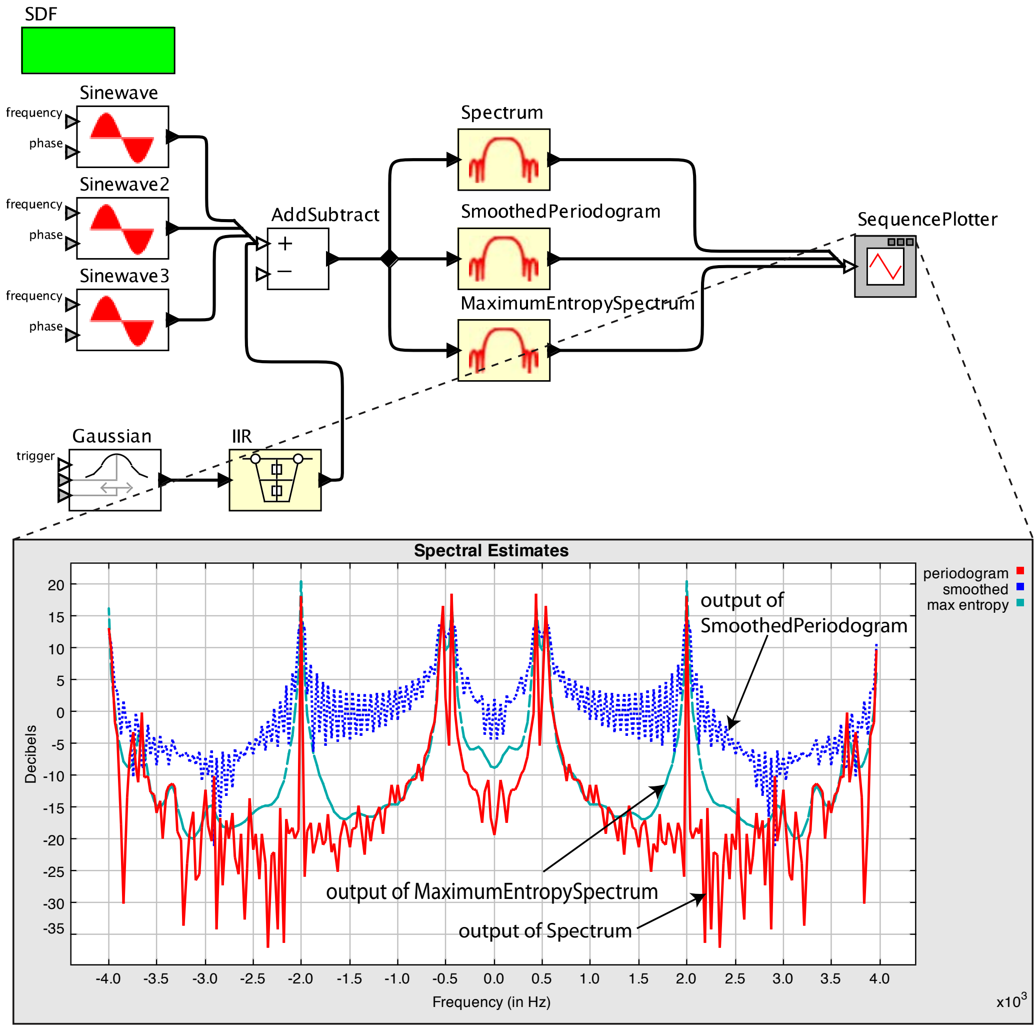


Figure 3.6: Comparison of three spectral estimation techniques described in the box on page 101. [online]

3.1.2 Feedback Loops

A feedback loop in SDF must include at least one instance of the **SampleDelay** actor (found in the `FlowControl→SequenceControl` sublibrary). Without this actor, the loop would **deadlock**; actors in the feedback loop would be unable to fire because they depend on each other for tokens. The SampleDelay actor resolves this problem by producing initial tokens on its output before the model begins firing. The initial tokens are specified by the *initialOutputs* parameter, which defines an array of tokens. These initial tokens enable downstream actors to fire and break the circular dependencies that would otherwise result from a feedback loop.

Example 3.6: Consider the model in Figure 3.7. This homogeneous SDF model generates a counting sequence using a feedback loop. The SampleDelay actor begins the process by producing a token with value of 0 on its output. This token, together with a token from the Const actor, enables the AddSubtract actor to fire. The output of that actor enables the next firing of SampleDelay. After the initial firing, the SampleDelay copies its input to its output unchanged.

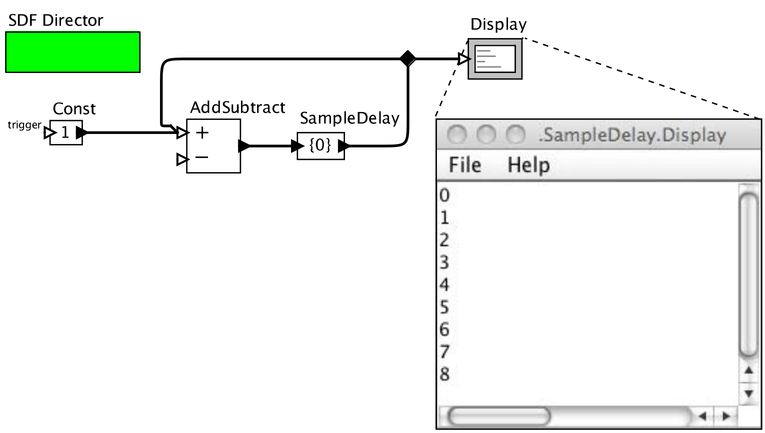


Figure 3.7: An SDF model with a feedback loop must have at least one instance of the SampleDelay actor in it. [online]

Consistency is sufficient to ensure bounded buffers, but it is not sufficient to ensure that an unbounded execution exists. The model may deadlock even if it is consistent. The SDF director analyzes a model for both consistency and deadlock. To allow feedback, it treats **delay actors** differently than other actors. A delay actor is able to produce initial output tokens before it receives any input tokens. It subsequently behaves like a normal SDF actor, consuming and producing a fixed number of tokens on each firing. In the SDF domain, the initial tokens are understood to be initial conditions for execution rather than part of the execution itself. Thus, the scheduler will ensure that all initial tokens are produced before the SDF execution begins. Conceptually, the SampleDelay actor could be replaced by initial tokens placed on a feedback connection.

Example 3.7: Figure 3.8 shows an SDF model with initial tokens on a feedback loop. The balance equations are

$$\begin{aligned} 3q_A &= 2q_B \\ 2q_B &= 3q_A. \end{aligned}$$

The least positive integer solution exists and is $q_A = 2$, and $q_B = 3$, so the model is consistent. With four initial tokens on the feedback connection, as shown, the following schedule can be repeated forever,

$$A, B, A, B, B.$$

This schedule starts with actor A, because at the start of execution, only actor A can fire, because actor B does not have sufficient tokens. When A fires, it consumes three tokens from the four initial tokens, leaving one behind. It sends three tokens to B. At this point, only B can fire, consuming two of the three tokens sent by A,

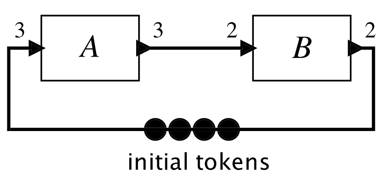


Figure 3.8: An SDF model with initial tokens on a feedback loop. In Ptolemy II, these initial tokens would be provided by a SampleDelay actor.

and producing two more tokens on its output. At this point, actor A can fire again, because there are exactly three tokens on its input. It will consume all of these and produce three tokens. At this point, B has four tokens on its input, enabling two firings. After those two firings, both actors have been fired the requisite number of times, and the buffer on the feedback arc again has four tokens. The schedule has therefore returned the dataflow graph to its initial condition.

Were there any fewer than four initial tokens on the feedback path, however, the model would deadlock. If there were only three tokens, for example, then A could fire, followed by B, but neither would have enough input tokens to fire again.

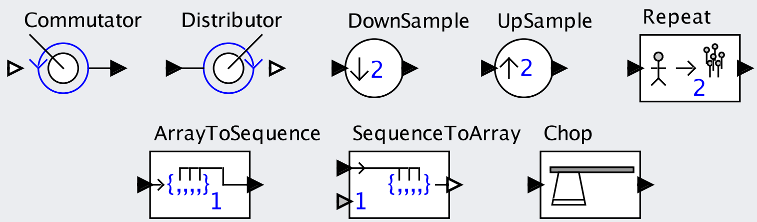
Lee and Messerschmitt (1987b) discuss the procedure for solving the balance equations, along with a procedure that will either provide a schedule for an unbounded execution or prove that no such schedule exists. Using these procedures, both bounded buffers and deadlock are **decidable** for SDF models (meaning that it is possible for Ptolemy to determine whether deadlock or unbounded buffers occur in any SDF model).

3.1.3 Time in Dataflow Models

In the SDF examples we have considered thus far, we have used the SequencePlotter actor but not the TimedPlotter actor (see Chapter 17). This is because the SDF domain does not generally use the notion of time in its models. By default, time does not advance as an SDF model executes (though the SDFDirector does contain a parameter, called *period*, that can be used to advance time by a fixed amount on each iteration of the model). Therefore, in most SDF models, the TimedPlotter actor would show the time axis as always being equal to zero. The SequencePlotter actor, in contrast, plots a sequence of values that are not time-based, and is therefore frequently used in SDF models. The discrete event (DE) and Continuous domains, discussed in Chapters 7 and 9, include a much stronger notion of time, and often use the TimedPlotter.

Sidebar: Multirate Dataflow Actors

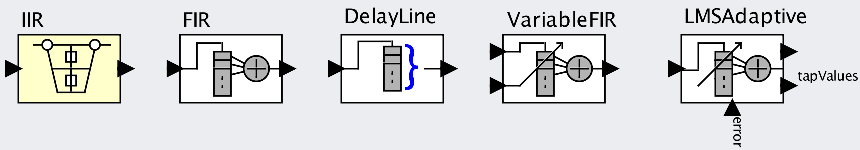
The Ptolemy II library offers a few actors that produce and/or consume multiple tokens per firing on a port. The most basic ones are shown below:



- **Commutator** and **Distributor**, in the `FlowControl→Aggregators` sublibrary, convert tokens arriving from multiple signals into a sequence of tokens and vice versa. Commutator has a multiport input, and on each firing, it reads a fixed number of tokens (given by its *blockSize* parameter) from each input channel, and outputs all the tokens from all the input channels as a sequence. Distributor reverses this process.
- **DownSample** and **UpSample**, in `SignalProcessing→Filtering`, discard or insert tokens. Downsample reads a fixed number of tokens (given by its *factor* parameter), and outputs one of those tokens (selected by its *phase* parameter). UpSample inserts a fixed number of zero-valued tokens between input tokens.
- **Repeat**, found under `FlowControl→SequenceControl`, is similar to UpSample except that instead of inserting zero-valued tokens, it repeats the input token.
- ArrayToSequence and SequenceToArray, found in the `Array` library, convert array tokens into sequences of tokens and vice versa. Both actors have an *arrayLength* parameter that specifies the length of the incoming (or outgoing) array. ArrayToSequence also has an *enforceArrayLength* parameter, which, if set to `true`, causes the actor to generate an error message if it receives an array of the wrong length. In SequenceToArray, *arrayLength* is a PortParameter, and hence the number of input tokens that are read can vary. These actors are SDF actors only when the array length is constant.
- **Chop**, in `FlowControl→SequenceControl`, reads a specified number of input tokens and produces a specified subset of those inputs, possibly padded with zero-valued tokens or previously consumed tokens.

Sidebar: Signal Processing Actors

In addition to the spectral analysis actors described on page 101, Ptolemy II includes several other key signal processing actors, as shown below.



- **IIR** implements an infinite impulse response filter, also called a recursive filter (see Lee and Varaiya (2011)). Filter coefficients are provided in two arrays, one for the numerator and one for the denominator polynomial of the transfer function.
- **FIR** implements a finite impulse response filter, also called a tapped delay line, with coefficients specified by the *taps* parameter. Whereas IIR is a homogeneous SDF (single-rate) actor, FIR is potentially a multirate actor. When the *decimation* (*interpolation*) parameters are not equal to 1, the filter behaves as if it were followed (preceded) by a DownSample (UpSample) actor (see sidebar on page 106). However, the implementation is much more efficient than it would be using UpSample or DownSample actors; a polyphase structure is used internally, avoiding unnecessary use of memory and unnecessary multiplication by zero. Arbitrary sample-rate conversions by rational factors can be accomplished in this manner.
- **DelayLine** produces an array rather than the scalar produced by FIR. Instead of a weighted average of the contents of the delay line (which is what FIR produces), DelayLine simply outputs the contents of the delay line as an array.
- **VariableFIR** is identical to FIR except that the coefficients are provided as an array on an input port (and thus can vary) rather than being defined as actor parameters.
- **LMSAdaptive** is similar to FIR, except that the coefficients are adjusted on each firing using a gradient descent adaptive filter algorithm that attempts to minimize the power of the signal at the *error* input port.

In addition to the actors described here, the signal processing library includes fixed and adaptive lattice filters, statistical analysis actors, actors for communications systems (such as source and channel coders and decoders), audio capture and playback, and image and video processing actors. See the actor index on page 632.

Sidebar: Dynamically Varying Rates

A variant of SDF that is called **parameterized SDF** (**PSDF**), introduced by Bhattacharya and Bhattacharyya (2000), allows the production and consumption rates of ports to be given by a parameter rather than being a constant. The value of the parameter is permitted to change, but only between complete iterations. When the value of such a parameter changes, a new schedule must be used for the next complete iteration.

The example in Figure 3.9 illustrates how PSDF can be achieved with the SDF director in Ptolemy II. First, notice that the director's *allowRateChanges* parameter has been set to true. This indicates to the director that it may need to compute more than one schedule, since rate parameters may change during the execution of the model.

Second, notice that the Repeat actor's *numberOfTimes* parameter is set equal to the model parameter *rate*, which initially has value zero. Hence, when this model executes its first iteration, the Repeat actor will produce zero tokens, so the Display actor will not fire. The initial output from the Ramp actor, which has value 1, will not be displayed.

During this first iteration, the Expression and SetVariable actor both fire once. The Expression actor sets its output equal to input, unless the input is equal to the value of the *iterations* parameter (which it doesn't in this first iteration). The SetVariable actor sets the value of the *rate* parameter to 1. By default, SetVariable has a *delayed* parameter with value true, which means that the *rate* parameter changes only after the current iteration is complete.

In the second iteration, the value of the *rate* parameter is 1, so the Repeat actor copies its input (which has value 2) once to its output. The Expression and SetVariable actors set the *rate* parameter now to 2, so in the third iteration, the Repeat actor copies its input (which has value 3) twice to its output. The sequence of displayed outputs is therefore $2, 3, 3, 4, 4, 4, \cdots$.

To stop the model, the *iterations* parameter of the director is set to 5. In the last iteration of the execution, the Expression actor ensures that the *rate* parameter gets reset to 0. Hence, the next time the model executes, it will start again with the *rate* parameter set to 0.

In this example, each time the *rate* parameter changes, the SDF director recomputes the schedule. In a better implementation of PSDF, it would probably precompute schedules and/or cache previously computed schedules, but this implementation does not do that. It just recomputes the schedule between iterations.

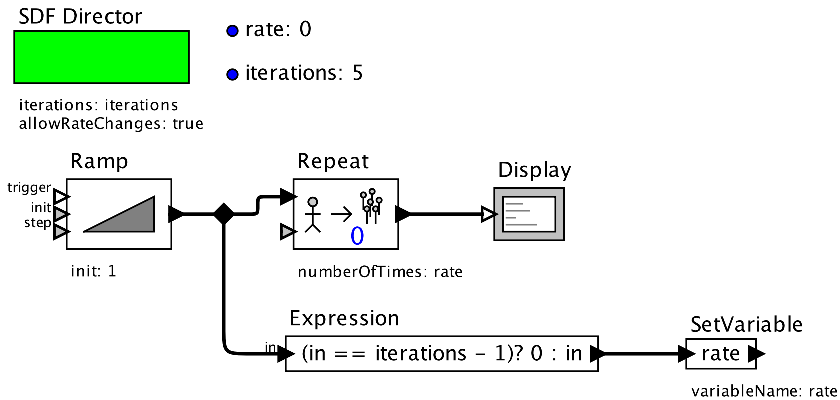


Figure 3.9: An SDF model with dynamically varying rates. [online]

Sidebar: StreamIt

Thies et al. (2002) give a textual programming language, **StreamIt**, based on SDF and intended for use with streaming data applications such as multimedia. Software components (called **filters** rather than actors) produce and consume fixed amounts of data. The language provides compact structured constructs for common patterns of actor composition, such as chains of filters, parallel chains of filters, or feedback loops.

A key innovation in StreamIt is the notion of a **teleport message** (Thies et al., 2005). Teleport messages improve the expressiveness of SDF by allowing one actor to sporadically send a message to another; that is, rather than sending a message on every firing, only some firings send messages. The teleport message mechanism nonetheless ensures determinism by ensuring that the message is received by the receiving actor in exactly the same firing that it would have if the sending actor had sent messages on every firing. But it avoids the overhead of sending messages on every firing. This approach models a communication channel where tokens are sometimes, but not always, produced and consumed. But it preserves the determinism of SDF models, where the results of execution are the same for any valid schedule.

Sidebar: Other Variants of Dataflow

A disadvantage of SDF is that every actor must produce and consume a fixed amount of data; the production and consumption rates cannot depend on the data. DDF (Section 3.2) relaxes this constraint at the cost of being able to statically precompute the firing schedule. In addition, as discussed earlier in the chapter, it is no longer possible to analyze all models for deadlock or bounded buffers (these questions are undecidable). Researchers have developed a number of variants of dataflow, however, that are more expressive than SDF but still amenable to some forms of static analysis.

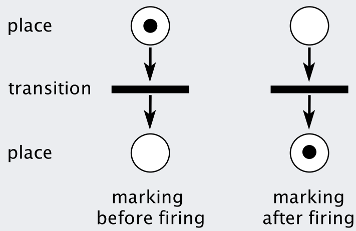
Cyclo-static dataflow (**CSDF**) allows production and consumption rates to vary in a periodic manner (Bilsen et al., 1996). An example is the **SingleTokenCommutator** actor (in `FlowControl→Aggregators`). This actor is similar to the Commutator actor (see sidebar on page 106), but instead of consuming all inputs in a single firing, it consumes inputs from only one channel in each firing, and rotates through the input channels on successive firings. For each input channel, the consumption rate alternates between zero and one. This actor is useful in feedback systems where the input to the second channel depends on the input to the first channel.

SDF can also be combined hierarchically with finite state machines (FSMs) to create modal models, described in Chapter 8. Each state of the FSM is associated with a submodel (a mode refinement, where each refinement can have different production and consumption rates). If the state transitions of the FSM are constrained to occur only at certain times, the model remains decidable. This combination was introduced by Girault et al. (1999), who called it **heterochronous dataflow** (**HDF**). **SDF Scenarios** (Geilen and Stuijk, 2010) are similar to HDF in that they also use an FSM, but rather than having mode refinements, in SDF Scenarios each state of the FSM is associated with a set of production and consumption rates for a single SDF model. Bhattacharya and Bhattacharyya (2000) introduced parameterized SDF (PSDF), where production and consumption rates can depend on input data as long as the same dependence can be represented in parameterized schedule.

Murthy and Lee (2002) introduced **multidimensional SDF** (**MDSDF**). Whereas a channel in SDF carries a sequence of tokens, a channel in MDSDF carries a multidimensional array of tokens. That is, the history of tokens can grow along multiple dimensions. This model is effective for expressing certain kinds of signal processing applications, particularly image processing, video processing, radar and sonar.

Sidebar: Petri Nets

Petri nets, named after Carl Adam Petri, are a popular modeling formalism related to dataflow (Murata, 1989). They have two types of elements, **places** and **transitions**, depicted as white circles and rectangles, respectively. A place can contain any number of tokens, depicted as black circles. A transition is **enabled** if all places connected to it as inputs contain at least one token.



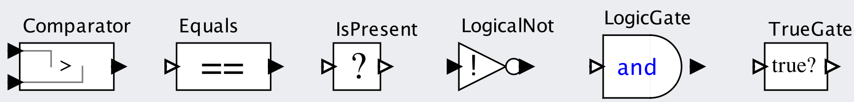
Once a transition is enabled, it can **fire**, consuming one token from each input place and depositing one token on each output place. The state of a network, called its **marking**, is the number of tokens on each place in the network. The figure above shows the marking of a simple network before and after a firing. If a place provides inputs to more than one transition, then a token on that place may trigger a firing of either destination transition (one or the other fires, nondeterministically).

Petri net transitions are like dataflow actors; they fire when sufficient inputs are available. In basic Petri nets, tokens have no value, and firing of a transition does not involve any computation on tokens. A firing is just the act of moving tokens from one place to another. Also, places do not preserve token ordering, unlike dataflow buffers. Like homogeneous SDF, transitions are enabled by a single token on each input place. Unlike SDF, a place may feed more than one transition, resulting in nondeterminism.

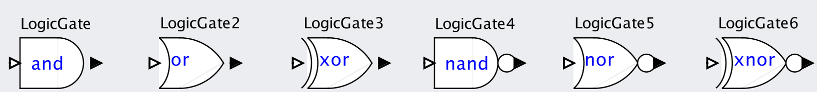
There are many variants of Petri nets, at least one of which is equivalent to SDF. In particular, tokens can have values (such tokens are called **colored tokens** in the literature, where the color of a token represents its value). Transitions can manipulate the color of tokens (analogous to the firing function of a dataflow actors). Arcs connecting places to transitions can be weighted to indicate that more than one token is required to fire a transition, or that a transition produces more than one token. This is analogous to SDF production and consumption rates. And finally, the Petri net graph structure can be constrained so that for each place, there is exactly one source transition and exactly one destination transition. With order-preserving places, such Petri nets are SDF graphs.

Sidebar: Logic Actors

The actors found in the Logic library are useful for building control logic:



- The **Comparator** actor compares two values of type *double* (or of any type that can be losslessly converted to *double*, as explained in Chapter 14). The available comparisons are $>$, $>=$, $<$, $<=$, and $==$. The output is a boolean.
- The **Equals** actor compares any number of input tokens of any type for equality and outputs a boolean true if they are equal and a false otherwise.
- The IsPresent actor outputs a boolean true if the input is present when it fires and false otherwise. In dataflow domains, the input is always present, so the output will always be true. This actor is more useful in the SR and DE domains (Chapters 5 and 7).
- **LogicalNot** accepts an input boolean and outputs the converse boolean.
- **LogicGate** implements the following six logic functions (the icon changes when you select the logic function):



- The TrueGate actor produces a boolean true output when the input is a boolean true. Otherwise, it produces no token at all. This is clearly not an SDF actor, but it can be used with DDF. It is also useful in SR (Chapter 5).

3.2 Dynamic Dataflow

Although the ability to guarantee bounded buffers and rule out deadlock is valuable, it comes at a price: SDF is not very expressive. It cannot directly express conditional firing, for example, such as when an actor fires only if a token has a particular value.

A number of dataflow variants have been developed that loosen the constraints of SDF; several of these are discussed in the sidebar on page 110. In this section, we describe a variant known as **dynamic dataflow** (**DDF**). DDF is much more flexible than SDF, because actors can produce and consume a varying number of tokens on each firing.

3.2.1 Firing Rules

As in other dataflow MoCs (such as SDF) DDF actors begin execution when they have sufficient input data. For a given actor to fire, its **firing rule** (that is, the condition that must be met before an actor can fire) must be satisfied. In SDF, the actors' firing rule is constant. It simply specifies the fixed number of tokens that are required on each input port before the actor can fire. In the DDF domain, however, firing rules can be more complicated, and may specify a different number of tokens for each firing.

Example 3.8: The SampleDelay actor of Example 3.6 is directly supported by the DDF MoC, without any need for special treatment of initial tokens. Specifically, the firing rule for SampleDelay states that on the first firing, it requires no input tokens. On subsequent firings, it requires one token.

Another difference is that, in SDF, actors produce a fixed number of tokens on each output port. In DDF, the number of tokens produced can vary.

Example 3.9: On its first firing, the SampleDelay actor produces the number of tokens specified in its *initialOutputs* parameter. On subsequent firings it produces a single token that is equal to the token it consumed.

The firing rules themselves need not be constant. Upon firing, a DDF actor may change the firing rules for the next firing.

A key DDF actor is the BooleanSelect, which merges two input streams into one stream according to a stream of boolean-valued control tokens (see sidebar on page 119). This actor has three firing rules. Initially, it requires one token on the *control* (bottom) port, and no tokens on the other two ports. When it fires, it records the value of the control token and changes its firing rule to require a token on one of the *trueInput* port (labeled *T*) or the *falseInput* port (labeled *F*), depending on the value of the control token. When the actor next fires, it consumes the token on the corresponding port and sends it to the output. Thus, it fires twice to produce one output. After producing an output, its firing rule reverts to requiring a single token on the *control* port.

A more general version of the BooleanSelect is the Select actor, which merges an arbitrary number of input streams into one stream according to a stream of integer-valued control tokens, rather than just two streams (see sidebar on page 119).

Whereas BooleanSelect and Select merge multiple input streams into one, BooleanSwitch and Switch do the converse; they split a single stream into multiple streams. Again, a stream of control tokens determines, for each input token, to which output stream that token should be sent. These Switch and Select actors accomplish conditional routing of tokens, as illustrated in the following examples.

Example 3.10: Figure 3.10 uses BooleanSwitch and BooleanSelect to accomplish conditional firing, the equivalent of if-then-else in an imperative programming language. In this figure, the **Bernoulli** actor produces a random stream of Boolean-valued tokens. This control stream controls the routing of tokens produced by the Ramp actor. When Bernoulli produces `true`, the output of the Ramp actor is multiplied by -1 using the Scale actor. When Bernoulli produces `false`, Scale2 is used; it passes its input through unchanged. The BooleanSelect uses the same control stream to select the appropriate Scale output.

Example 3.11: Figure 3.11 shows a DDF model that uses BooleanSwitch and BooleanSelect to realize data-dependent iteration using a feedback loop. The Ramp

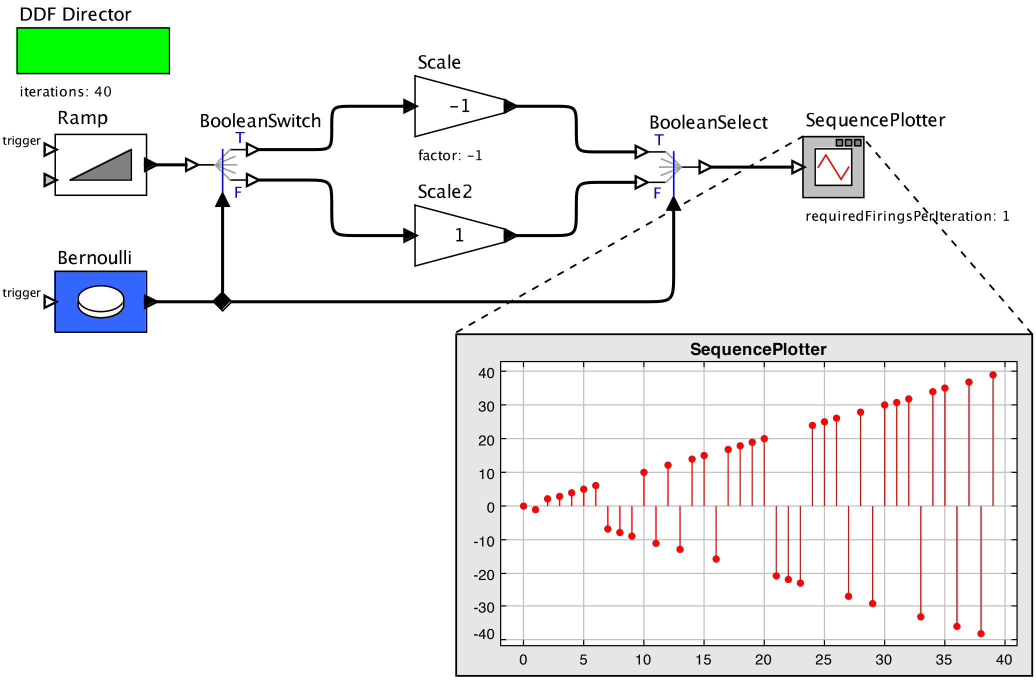


Figure 3.10: A DDF model that accomplishes conditional firing. [online]

actor feeds the loop with a sequence of increasing integers, 0, 1, 2, 3, etc. The SampleDelay initiates the loop by providing a `false` token to the *control* port of the BooleanSelect. In the full cycle, each input integer is repeatedly multiplied by 0.5 until the resulting value is less than 0.5. The Comparator actor (found in the `Logic` library) controls whether the token is routed back around the loop for another iteration or routed out of the loop to the Discard actor (the one at the right with the icon that looks like a ground symbol on an electrical circuit diagram, found in `Sinks→GenericSinks`). The Discard actor receives and discards its input, but in this case, it is also used to control what an iteration means. The parameter *requiredFiringsPerIteration* has been added to the actor and assigned a value of 1 (see Section 3.2.2 below). Hence, one iteration of the model consists of as many iterations of the loop as needed to produce one firing of Discard. This structure is analogous to a do-while loop in an imperative programming language.

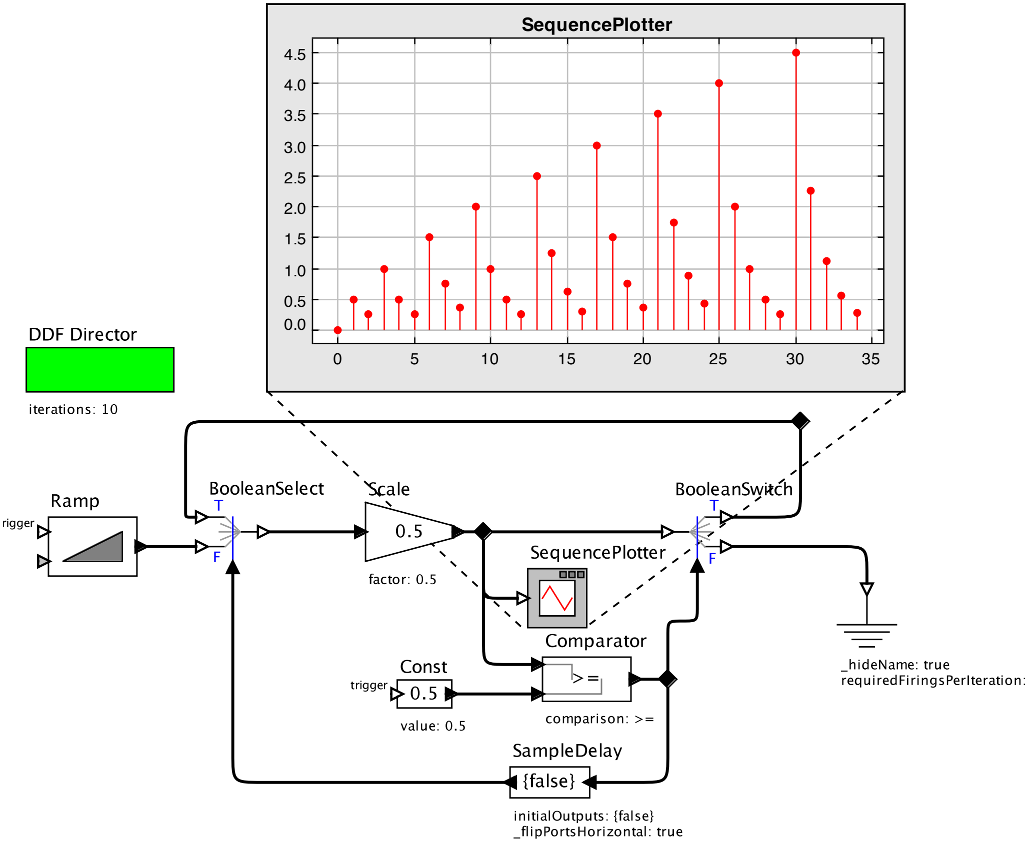


Figure 3.11: A DDF model that accomplishes data-dependent iteration. [online]

The pattern shown in Figure 3.11 is sufficiently useful that it might be used repeatedly. Fortunately, Ptolemy II includes a mechanism for storing and re-using a **design pattern**, created by Feng (2009). The pattern shown in Figure 3.12, for example, is available as a unit in the `MoreLibraries→DesignPatterns`. In fact, any Ptolemy II model can be exported to a library as a design pattern and reimported into another model as a unit by simply dragging it into the model.

The Switch and Select actors (and their boolean versions) that are part of the DDF domain provide increased flexibility and expressiveness relative to the SDF domain, but their use means that it may not be possible to determine a schedule with bounded buffers, nor is

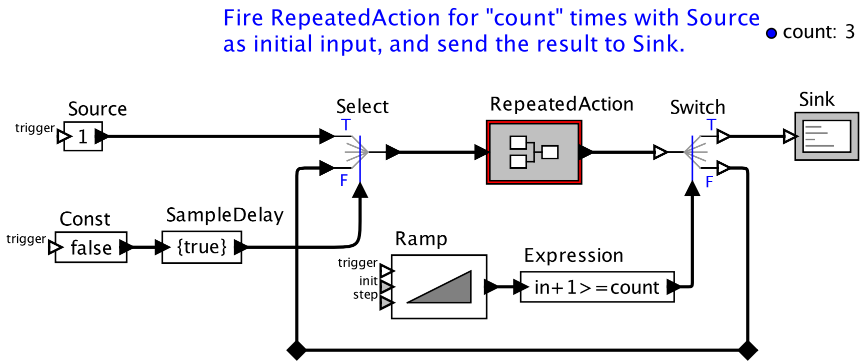


Figure 3.12: A design pattern stored as a unit in a library.

possible to ensure that the model will not deadlock. In fact, Buck (1993) showed that bounded buffers and deadlock are undecidable for DDF models. For this reason, DDF models are not as readily analyzed.

Example 3.12: A variant of the if-then-else model shown in Figure 3.10 is shown in Figure 3.13. In this case, the inputs to the BooleanSelect have been reversed. Unlike the earlier model, this model has no schedule that assures bounded buffers. The Bernoulli actor is capable of producing an arbitrarily long sequence of `true`-valued tokens, during which an arbitrarily long sequence of tokens may build up on input buffer for the *false* port of the BooleanSelect, thus potentially overflowing the buffer.

Switch and Select and their boolean cousins are dataflow analogs of the **goto** statement in imperative languages. They provide low-level control over execution by conditionally routing tokens. Like goto statements, their use can result in models that are difficult to understand. This problem is addressed using structured dataflow, described in the sidebar on page 120, and implemented in Ptolemy II using higher-order actors, described in Section 2.7.

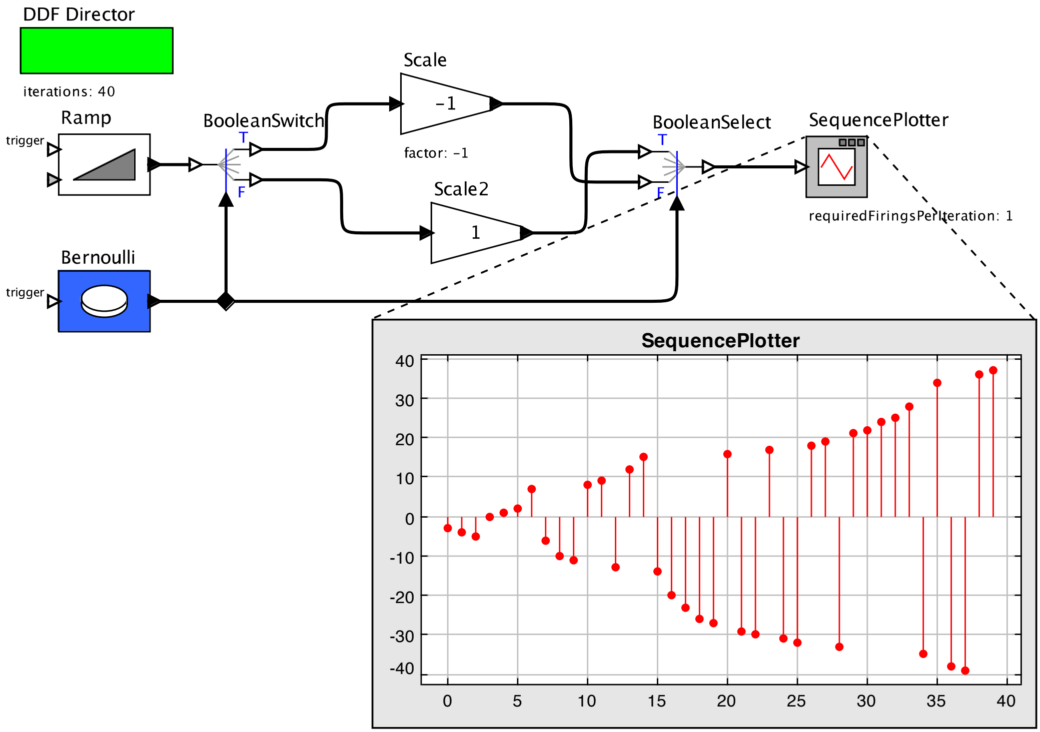


Figure 3.13: A DDF model that has no bounded buffer schedule. [online]

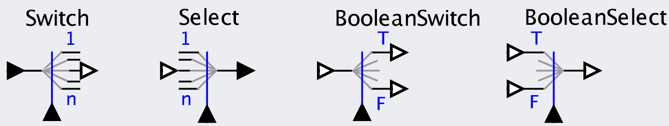
3.2.2 Iterations in DDF

One of the advantages of SDF is that a complete iteration is uniquely defined. It consists of a fixed number of firings of each of the actors in the model. It is therefore easy to control the duration of an overall execution of the model by setting the *iterations* parameter of the SDF director, which controls the number of times each actor will be executed.

The DDF director also has an *iterations* parameter, but defining an iteration is more difficult. An iteration can be defined by adding a parameter to one or more actors named *requiredFiringsPerIteration* and giving that parameter an integer value, as illustrated in the following example.

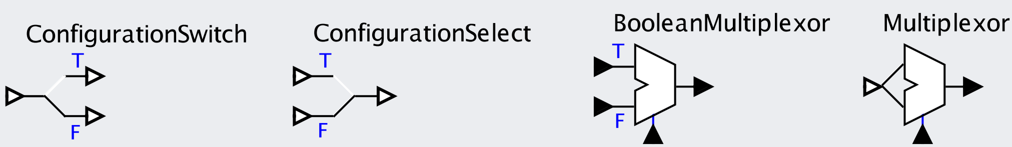
Sidebar: Token Flow Control Actors

Ptolemy II provides a number of actors that can route tokens in a model. The most basic of these are **Switch** and **Select** (and their boolean variants), shown here:



On each firing, Switch consumes one token from the input and an integer-valued token from the *control* port (on the bottom) and routes the input token to the output channel specified by the control token. All other output channels produce no tokens on that firing. Select does the converse, consuming a single token from the channel specified by the control token and sending that token to the output. The other input channels consume no tokens. **BooleanSwitch** (**BooleanSelect**) are variants where the number of outputs (or inputs) is constrained to be two, and the control token is boolean rather than integer valued.

Switch and Select can be compared to the following actors with related functionality:



ConfigurationSwitch is similar to BooleanSwitch except that instead of a *control* input port it has a *parameter* that determines which output to send data to. If the value of the parameter does not change during execution of the model (normally this is the case with parameters), then this actor is an SDF actor that always produces zero tokens on one output and one token on the other. **ConfigurationSelect** is likewise similar to BooleanSelect.

BooleanMultiplexor and **Multiplexor** are similar to BooleanSelect and Select except that they consume one token from *all* input channels. These actors discard all but one of those input tokens, and send that one token to the output. Since these two actors consume and produce exactly one token on every channel, they are homogeneous SDF actors.

Sidebar: Structured Dataflow

In an imperative language, structured programming replaces goto statements (which can be problematic, as described in Dijkstra (1968)) with nested `for` loops, `if-then-else`, `do-while`, and recursion. In structured dataflow, these concepts are adapted to the dataflow modeling environment.

Figure 3.14 shows an alternative way to accomplish the conditional firing of Figure 3.10. The result is an SDF model that has many advantages over the DDF model in Figure 3.10. The **Case** actor is an example of a higher-order actor, like those discussed in Section 2.7. Inside, it contains two sub-models (refinements), one named true that contains a Scale actor with a parameter of -1, and one named default that contains a Scale actor with a parameter of 1. When the control input to the Case actor is true, the true refinement executes one iteration. For any other control input, the default refinement executes.

This style of conditional firing is called **structured dataflow**, because, much like structured programming, control constructs are nested hierarchically. The approach avoids arbitrary data-dependent token routing (which is analogous to avoiding arbitrary branches using goto instructions). Moreover, the use of the Case actors enables the overall model to be an SDF model. In the example in Figure 3.14, every actor consumes and produces exactly one token on every port. Hence, the model is analyzable for deadlock and bounded buffers.

This style of structured dataflow was introduced in LabVIEW, a design tool developed by National Instruments (Kodosky et al., 1991). In addition to providing a conditional operation similar to that of Figure 3.14, LabVIEW provides structured dataflow constructs for iterations (analogous to `for` and `do-while` loops in an imperative language), and for sequences (which cycle through a finite set of submodels). Iterations can be achieved in Ptolemy II using the higher-order actors of Section 2.7. Sequences (and more complicated control constructs) can be implemented using modal models, discussed in Chapter 8. Ptolemy II supports structured recursion using the **ActorRecursion** actor, found in `DomainSpecific→DynamicDataflow` (see Exercise 3). However, without careful constraints, boundedness again becomes undecidable with recursion (Lee and Parks, 1995).

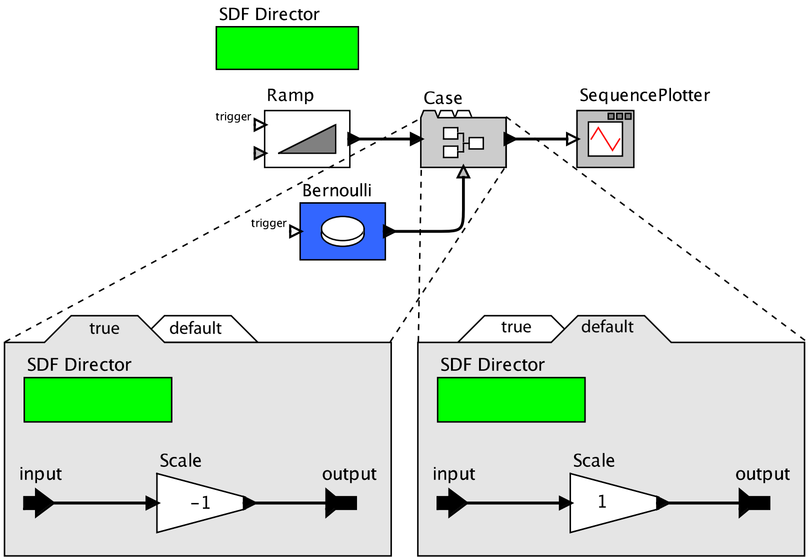


Figure 3.14: Structured dataflow approach to conditional firing. [online]

Example 3.13: Consider the if-then-else example in Figure 3.10, discussed in Example 3.10. The *iterations* parameter of the director is set to 40, and indeed the plot has 40 points. This is because a parameter named *requiredFiringsPerIteration* has been added to the SequencePlotter actor and assigned the value 1. As a consequence, each iteration must include at least one firing of the SequencePlotter. In this case, no other actor in the model has a parameter named *requiredFiringsPerIteration*, so this parameter ends up determining what constitutes an iteration.

When multiple actors within a model have parameters named *requiredFiringsPerIteration*, or when there are no such parameters, the situation is more subtle. In these cases, DDF still has a well-defined iteration, but the definition is complex, and can surprise the designer (see sidebar on page 124).

Example 3.14: Consider again the if-then-else example in Figure 3.10. If we remove the *requiredFiringsPerIteration* from the SequencePlotter, then 40 iterations of the model will produce only nine plotted points. Why? Recall from Section 3.2.1 that BooleanSelect fires twice for each output it produces. Absent any constraints in the model, the DDF director will not fire any actor more than once in an iteration.

Example 3.15: Figure 3.15 shows a DDF model that replaces all instances of the SequencePlotter actor with instances of the Test actor for all Ptolemy models in a directory (see box on page 126 for why you might want to do this). This model uses the DirectoryListing actor (see box on page 128) to construct an array of file names for actors in a specified directory. Ptolemy models are identified by files whose names match the regular expression `.*.xml`, which matches any file name that ends with `.xml`. The *firingCountLimit* parameter of the DirectoryListing ensures that this actor fires only once. It will produce one array token on its output, and then will refuse to fire again. Once the data in that array have been processed, there are no more tokens to process, so the model deadlocks, and stops execution.

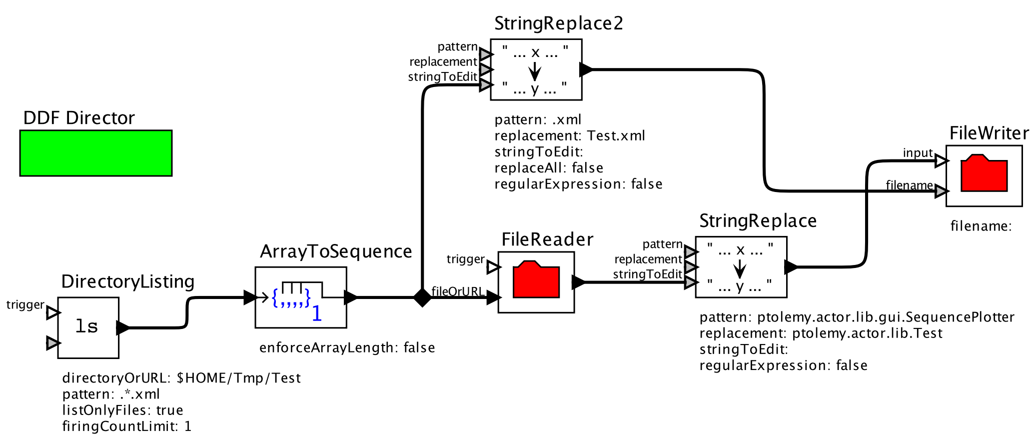


Figure 3.15: A DDF model that replaces all instances of the SequencePlotter actor with instances of the Test actor for all Ptolemy models in a directory.

The ArrayToSequence actor (see box on page 106) converts the array of file names into a sequence of tokens, one string-valued token for each file name. Notice that the *enforceArrayLength* parameter is set to false for this actor. If we were to know exactly how many XML files were in the directory in question, would could have left this parameter with its default value true, set the *arrayLength* parameter to the number of files, and used the SDF director instead of the DDF director. The ArrayToSequence actor would consume one token and produce a fixed, known number of output tokens, and hence would be an SDF actor. But since we do not know in general how many matching files there will be in the directory, the DDF director is more useful.

The FileReader actor (see box on page 128) reads the XML file and outputs its contents as a single string. The StringReplace actor (see box on page 125) replaces all instances of the full classname for the SequencePlotter actor with the full classname for the Test actor.

The second StringReplace actor, named StringReplace2, is used to create a new filename from the original file name. For example, the filename `Foo.xml` will become `FooTest.xml`. The FileWriter actor then writes the modified filename to a file with the new file name.

Note that we could have used the IterateOverArray actor and the SDF director instead (see Section 2.7.2). We leave this as an exercise (see Problem 2 at the end of this chapter).

3.2.3 Combining DDF with Other Domains

Although a system may be best modeled as DDF overall, it may contain some subsystems that can be modeled as SDF. Thus, a DDF model may contain an opaque composite actor that has an SDF director. This approach can improve efficiency and provide better control over the amount of computation done in an iteration.

Conversely, a DDF model may be placed within an SDF model if it behaves like SDF at its input/output boundaries. That is, to be used within an SDF model, a DDF opaque composite actor should consume and produce a fixed number of tokens. It is not generally possible for the DDF director to determine from the model how many tokens are produced and consumed at the boundary (this question is undecidable in general) so it is

Sidebar: Defining a DDF Iteration

An **iteration** in DDF consists of the minimum number of **basic iterations**,(defined below) that satisfy all constraints imposed by *requiredFiringsPerIteration* parameters.

In one **basic iteration**, the DDF director fires all **enabled** and **non-deferrable** actors once. An enabled actor is one that has sufficient data at its input ports, or has no input ports. A **deferrable actor** is one whose execution can be deferred because its execution is not currently required by a downstream actor. This is the case when the downstream actor either already has enough tokens on the channel connecting it to the deferrable actor, or the downstream actor is waiting for tokens on another channel or port. If there are no enabled and non-deferrable actors, then the director fires those enabled and deferrable actors that have the smallest maximum number of tokens on their output channels that will satisfy the demands of destination actors. If there are no enabled actors, then a deadlock has occurred. The above strategy was shown by Parks (1995) to guarantee that buffers remain bounded in an unbounded execution if there exists an unbounded execution with bounded buffers.

The algorithm that implements one basic iteration is as follows. Let E denote the set of enabled actors, and let D denote the set of deferrable enabled actors. One basic (default) iteration then consists of the following, where the notation $E \setminus D$ means "the set of elements in E that are not in D":

if $E \setminus D \neq \emptyset$ **then**
  fire actors in $(E \setminus D)$
else if $D \neq \emptyset$ **then**
  fire actors in minimax(D)
else
  declare deadlock
end if

The function "minimax(D)" returns a subset of D with the smallest maximum number of tokens on their output channels that satisfy the demand of destination actors. This will always include **sink actors** (actors with no output ports).

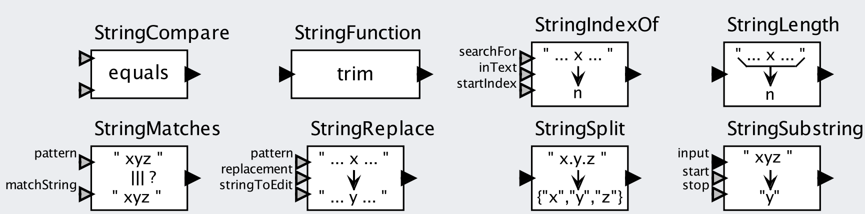
up to the model designer to assert the production and consumption rates. If they are not equal to a value of 1 (which need not be explicitly asserted), then the model designer can assert consumption and production rates by creating a parameter in each input port called *tokenConsumptionRate* and assigning it an integer value. Similarly, output ports should be given a parameter called *tokenProductionRate*.

Once the rates at the boundary are set, it is up to the model designer to ensure that they are respected at run time. This can be accomplished using the *requiredFiringsPerIteration* parameter, as explained above in Section 3.2.2. In addition, the DDF director has a parameter *runUntilDeadlockInOneIteration* that, when set to true, defines an iteration to be repeated invocations of a basic iteration (see sidebar on page 124). until deadlock is reached. If this parameter is used, it overrides any *requiredFiringsPerIteration* that may be present in the model.

DDF conforms with the loose actor semantics, meaning that if a DDF director is used in opaque composite actor, its state changes when its `fire` method is invoked. In particular,

Sidebar: String Manipulation Actors

The `String` library provides actor for manipulating strings:

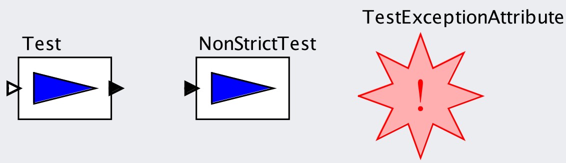


The **StringCompare** actor compares two strings, determine whether they are equal, or if one string starts with, ends with, or contains another string. The **StringMatches** actor checks whether a string matches a pattern given as a regular expression. The **StringFunction** actor can trim white space around a string or convert it to lower case or upper case. The **StringIndexOf** actor searches for a substring within a string a returns the index of that substring. The **StringLength** actor outputs the length of a string. **StringReplace** replaces a substring that matches a specified pattern with a specified replacement string. **StringSplit** divides a string at specified separators. **StringSubstring** extracts a substring, given a start and stop index.

dataflow actors **consume** input tokens in their `fire` method. Once the tokens have been consumed, they are no longer available in the input buffers. Thus, a second firing will see new data, regardless of whether `postfire` has been invoked. For this reason, DDF and SDF composite actors should not be used within domains that require strict actor semantics, such as SR and Continuous, unless the model builder can ensure that these

Sidebar: Building Regression Tests

When developing nontrivial models and when extending Ptolemy II, good engineering practice requires creating **regression tests**. Regression tests guard against future changes that may change behavior in ways that can invalidate applications that were created earlier. Fortunately, in Ptolemy II, it is extremely easy to create regression tests. Key components are found in `MoreLibraries→RegressionTest`:



The **Test** actor compares the inputs against the value specified by the *correctValues* parameter. The actor has a *trainingMode* parameter, which when set to true, simply records the inputs it receives. Therefore, a typical use is to put the actor in training mode, run the model, take the actor out training mode, and then save the model in some directory where all models are executed as part of daily testing. (This is how the vast majority of the rather extensive regression tests for the Ptolemy II itself are created.) The model will throw an exception if the Test actor receives any input that differs from the ones it recorded. Note that one of the key advantages of determinate models is the ability to construct such regression tests.

The **NonStrictTest** is similar, except that it tolerates (and ignores) absent inputs, and it tests the inputs in the postfire phase of execution rather than the fire phase. It is useful for domains such as SR and Continuous, which iterate to a fixed point.

Sometimes, a model is expected to throw an exception. A regression test for such a model should include an instance of **TestExceptionAttribute**, which also has a training mode. The presence of this attribute in a model causes the model to throw an exception if the execution of the model does *not* throw an exception, or if the exception it throws does not match the expected exception.

composite actors will not be fired more than once in an iteration of the SR of Continuous container.

Note that any SDF model can be run with a DDF Director. However, the notion of iteration may be different. Sometimes, a DDF model may be run with an SDF director even when there is data-dependent iteration. Figure 3.14 shows one example, where the Case actor facilitates this combination. However, it is sometimes possible to use this combination even when a Switch is used. The SDF scheduler will assume the Switch produces one token on every output channel, and will construct a schedule accordingly. While executing this schedule, the director may encounter actors that it expects to be ready to fire but which do not actually have sufficient input data to fire. Many actors can be safely iterated even if they have no input data. Their `prefire` method returns false, indicating to the director that they are not ready to fire. The SDF director will respect this, and will simply skip over that actor in a schedule. However, this technique is rather tricky and is not recommended. It can result in unintended sequences of actor execution.

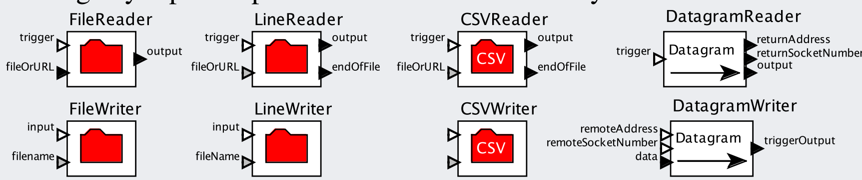
3.3 Summary

Dataflow is a simple and versatile model of computation in which the execution of actors is driven by the availability of input data. It is particularly useful for expressing streaming applications, where long sequences of data values are routed through computations, such as is common in signal processing and multimedia applications.

SDF is a simple (though restrictive) form of dataflow that enables extensive static analysis and efficient execution. DDF is more flexible, but also more challenging to control and more costly to execute, because scheduling decisions are made during run time. The two can be mixed within a single model, so the extra costs of DDF may be incurred only where they are absolutely required by the application. Both SDF and DDF are useful in modal models, as explained in Chapter 8. Using SDF and DDF within modal models provides a versatile concurrent programming model.

Sidebar: IO Actors

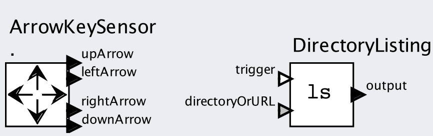
The following key input/output actors are in the `IO` library:



FileReader and **FileWriter** read and write files from the local disk or from a remote location specified via a URL or URI. For FileReader, the entire contents of the file is produced on a single output string token. For FileWriter, each input string token is written to a file, overwriting the previous contents of the file. In both cases, a new file name can be given for each firing. To read from standard input, specify System.in as the file name. To write to standard output, specify System.out as the file name. **LineReader** and **LineWriter** are similar, except that they read and write a line at a time.

CSVReader and **CSVWriter** read and write files or URLs that are in CSV format, or comma-separated values (actually, the separator can be anything; it need not be commas). CSV files are converted into record tokens, and record tokens are converted into CSV files. The first line of the file defines the field names of the records. To use CSVReader, you need to help the type system to determine the output type. The simplest way to do this is to enable backward type inference (see Section 14.1.4). This sets the data type of the output port of the CSVReader actor to be the most general type that is acceptable to the actors downstream. Thus, for example, if the actors downstream extract fields from the record, then the type constraints will automatically require those fields to be present and to have compatible types. You can also coerce the output type using the [`Customize→Ports`] context menu command.

The following actors are also in the `IO` library:



ArrowKeySensor produces outputs that respond to the arrow keys on the keyboard. **DirectoryListing** produces on its output an array of file names in a specified directory that match an (optional) pattern.

Exercises

1. The multirate actors described in the box on page 106 and the array actors described in the boxes on page 88 and 86 are useful with SDF to construct **collective operations**, which are operations on arrays of data. This exercise explores the implementation of what is called an **all-to-all scatter/gather** using SDF. Specifically, construct a model that generates four arrays with values:

```
{"a1", "a2", "a3", "a4"}
{"b1", "b2", "b3", "b4"}
{"c1", "c2", "c3", "c4"}
{"d1", "d2", "d3", "d4"}
```

   and converts them into arrays with values

```
{"a1", "b1", "c1", "d1"}
{"a2", "b2", "c2", "d2"}
{"a3", "b3", "c3", "d3"}
{"a4", "b4", "c4", "d4"}
```

   Experiment with the use of ArrayToElements and ElementsToArray, as well as ArrayToSequence and SequenceToArray (for the latter, you will also likely need Commutator and Distributor). Comment about the relative merits of your approaches. **Hint:** You may have to explicitly set the channel widths of the connections to 1. Double click on the wires and set the value. You may also experiment with MultiInstanceComposite.

2. Consider the model in Figure 3.15, discussed in Example 3.15. Implement this same model using the IterateOverArray actor and only the SDF director instead of the DDF director (see Section 2.7.2).

3. The DDF director in Ptolemy II supports an actor called ActorRecursion that is a recursive reference to a composite actor that contains it. For example, the model shown in Figure 3.16 implements the sieve of Eratosthenes, which finds prime numbers, as described by Kahn and MacQueen (1977).

   Use this actor to implement a composite actor that computes Fibonacci numbers. That is, a firing of your composite actor should implement the firing function

$f : \mathbb{N} \to \mathbb{N}$ defined by, for all $n \in \mathbb{N}$,

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

When ActorRecursion fires, it clones the composite actor above it in the hierarchy (i.e., its container, or its container's container, etc.) whose name matches the value of its *recursionActor* parameter. The instance of ActorRecursion is populated with ports that match those of that container. This actor should be viewed as a highly experimental realization of a particular kind of higher-order actor. It is a higher-order actor because it is parameterized by an actor that contains it. Its implementation, however, is very inefficient. The cloning of the actor it references on each firing is expensive in terms of both memory and time. A better implementation would use an approach similar to the stack frame approach used in procedural programming languages. Instead, the approach it uses is more like copying the source code at run time and then interpreting it. In an attempt to make execution more efficient, this actor avoids creating the clone if it has previously created it. Also, the visual representation of the recursive reference is inadequate. There is no way, looking only at the image in Figure 3.16, to tell what composite actor the ActorRecursion instance references. Thus, you cannot really read the program from its visual representation.

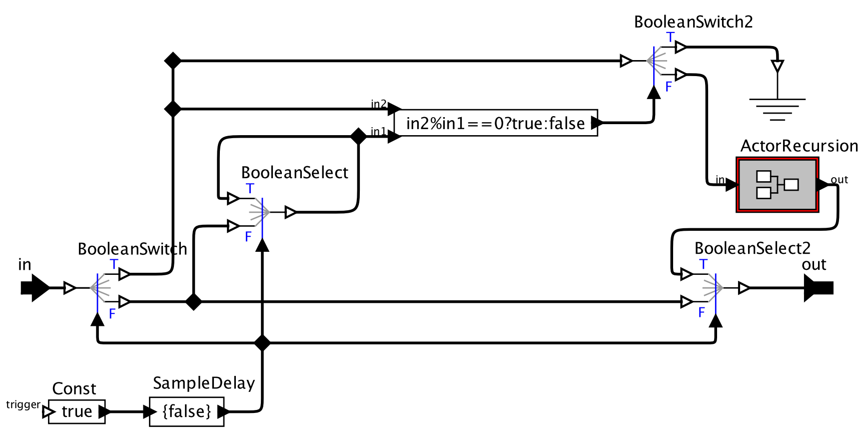


Figure 3.16: The sieve of Eratosthenes, using ActorRecursion in DDF. [online]

4

Process Networks and Rendezvous

Neil Smyth, John S. Davis II, Thomas Huining Feng, Mudit Goel, Edward A. Lee, Thomas M. Parks, Yang Zhao

Contents

Dataflow models of computation are concurrent. The order of the firings is constrained only by data precedence, so the director has quite a bit of freedom to choose an ordering. There is no reason that two actors that do not depend on each other's data could not be fired simultaneously. Indeed, there is a long history of schedulers that do exactly that, as described in the sidebar on page 100. Nevertheless, the directors described in Chapter 3 fire actors one at a time. This chapter introduces two directors that fire actors concurrently. Both are similar to dataflow, semantically, but they are used for very different purposes.

The directors in this chapter execute each actor in its own thread. A **thread** is a sequential program that can potentially share variables with other threads that execute concurrently. On a multicore machine, two threads may execute in parallel on separate cores. On a single core, the instructions of each thread are arbitrarily interleaved. Understanding the interactions between threads is notoriously difficult (Lee, 2006) because of this arbitrary interleaving of instructions. The directors described in this chapter provide a way for threads to interact in more understandable and predictable ways.

One possible motivation for using the directors in this chapter is to exploit the parallelism offered by multiple cores. Every model described in the previous chapter, for example, could also be executed with a PN director, described below, and actors will fire simultaneously on multiple cores. One could expect models to execute faster than they would with an SDF director. However, this is not our experience. Our experience is that PN models almost always run slower than SDF models, regardless of the number of cores. The reason for this appears to be the cost of thread interactions associated with the mutual exclusion locks that enable Ptolemy II models to be edited while they run. In principle, this capability could be disabled, and one would then expect performance improvements. But as of this writing, no mechanism has been built in Ptolemy II to do this. Hence, improving performance is probably not an adequate reason to use the threaded directors of this chapter over SDF or DDF when SDF or DDF is suitable.

Instead, we focus in this chapter on models where concurrent firing of actors is required by the goals of the model. These examples show that these directors do in fact provide a fundamental increment in expressiveness, and not just a performance improvement.

4.1 Kahn Process Networks

A model of computation that is closely related to dataflow models is **Kahn process networks** (variously called **KPN**, **process networks** or **PN**), named after Gilles Kahn, who introduced them (Kahn, 1974). The relationship between dataflow and PN is studied in detail by Lee and Parks (1995) and Lee and Matsikoudis (2009), but the short story is quite simple. In PN, each actor executes concurrently in its own thread. That is, instead of being defined by its firing rules and firing functions, a PN actor is defined by a (typically non-terminating) program that reads data tokens from input ports and writes data tokens to output ports. All actors execute simultaneously (conceptually; whether they actually execute simultaneously or are interleaved is irrelevant to the semantics).

In the original paper, Kahn (1974) gave very elegant mathematical conditions on the actors that ensure that a network of such actors is determinate. In this, case "determinate" means that the sequence of tokens on every connection between actors is uniquely defined, and specifically is independent of how the processes are scheduled. Every legal thread scheduling yields exactly the same data streams. Thus, Kahn showed that concurrent execution was possible without nondeterminism.

Three years later, Kahn and MacQueen (1977) gave a simple, easily implemented mechanism for programs that guarantees that the mathematical conditions are met to ensure determinism. A key part of the mechanism is to perform **blocking reads** on input ports whenever a process is to read input data. Specifically, blocking reads mean that if the process chooses to access data through an input port, it issues a read request and blocks until the data become available. It cannot test the input port for the availability of data and then perform a conditional branch based on whether data are available, because such a branch would introduce schedule-dependent behavior.

Blocking reads are closely related to firing rules. Firing rules specify the tokens required to continue computing (with a new firing function). Similarly, a blocking read specifies a single token required to continue computing (by continuing execution of the process). Kahn and MacQueen showed that if every actor implements a mathematical *function* from input sequences to output sequences (meaning that for each input sequence, the output sequence is uniquely defined), then blocking reads are sufficient to ensure determinism.

When a process writes to an output port, it performs a **nonblocking write**, meaning that the write succeeds immediately and returns. The process does not block to wait for the receiving process to be ready to receive data.[1] This is exactly how writes to output ports work in dataflow MoCs as well. Thus, the only material difference between dataflow and

[1]The Rendezvous director, discussed later in this chapter, differs at exactly this point, in that with that director, a write to an output port does not succeed until the actor with the corresponding input port is ready to read.

PN is that with PN, the actor is not broken down into firing functions. It is designed as a continuously executing program. The firing of an actor does not need to be finite.

Historical Notes: Process Networks

Gilles Kahn (1946 – 2006)

The notion of concurrent processes interacting by sending messages is rooted in Conway's **coroutines** (Conway, 1963). Conway described software modules that interacted with one another as if they were performing I/O operations. In Conway's words, "When coroutines A and B are connected so that A sends items to B, B runs for a while until it encounters a read command, which means it needs something from A. The control is then transferred to A until it wants to write, whereupon control is returned to B *at the point where it left off*."

The least fixed-point semantics is due to Kahn (1974), who developed the model of processes as continuous functions on a **CPO** (complete partial order). Kahn and MacQueen (1977) defined process interactions using non-blocking writes and blocking reads as a special case of continuous functions, and developed a programming language for defining interacting processes. Their language included recursive constructs, an optional functional notation, and dynamic instantiation of processes. They gave a demand-driven execution semantics, related to the lazy evaluators of Lisp (Friedman and Wise, 1976; Morris and Henderson, 1976). Berry (1976) generalized these processes with stable functions.

The notion of unbounded lists as data structures first appeared in Landin (1965). This underlies the communication mechanism between processes in a process network. The UNIX operating system, due originally to Ritchie and Thompson (1974) includes the notion of **pipes**, which implement a limited form of process networks (pipelines only). Later, named pipes provided a more general form.

Kahn (1974) stated but did not prove what has come to be known as the **Kahn principle**, that a maximal and fair execution of process network yields the least fixed point. This was later proved by Faustini (1982) and Stark (1995).

Sidebar: Process Networks and Dataflow

Three major variants of dataflow have emerged in the literature: Dennis dataflow (Dennis, 1974), Kahn process networks (KPN) (Kahn, 1974), and dataflow synchronous languages (Benveniste et al., 1994). The first two are closely related, while the third is quite different. This chapter deals with Kahn process networks, Dennis dataflow is addressed in Chapter 3, and dataflow synchronous languages are addressed in Chapter 5.

In **Dennis dataflow**, the behavior of an actor is given by a sequence of atomic firings that are enabled by the availability of input data. KPN, by contrast, has no notion of an atomic firing. An actor is a process that executes asynchronously and concurrently with others. Dennis dataflow can be viewed as a special case of KPN (Lee and Parks, 1995) by defining a firing to be the computation that occurs between accesses to inputs. But the style in which actors and models are designed is quite different. Dennis' approach is based on an operational notion of atomic firings driven by the satisfaction of firing rules. Kahn's approach is based on a denotational notion of processes as continuous functions on infinite streams.

Dennis' approach influenced computer architecture (Arvind et al., 1991; Srini, 1986), compiler design, and concurrent programming languages (Johnston et al., 2004). Kahn's approach influenced process algebras (e.g. Broy and Stefanescu (2001)) and concurrency semantics (e.g. Brock and Ackerman (1981); Matthews (1995)). It has had practical realizations in stream languages (Stephens, 1997) and operating systems (e.g. Unix pipes). Interest in these MoCs has grown with the resurgence of parallel computing, driven by multicore architectures (Creeger, 2005). Dataflow MoCs are being explored for programming parallel machines (Thies et al., 2002), distributed systems (Lzaro Cuadrado et al., 2007; Olson and Evans, 2005; Parks and Roberts, 2003), and embedded systems (Lin et al., 2006; Jantsch and Sander, 2005). There are improved execution policies (Thies et al., 2005; Geilen and Basten, 2003; Turjan et al., 2003; Lee and Parks, 1995) and standards (Object Management Group (OMG), 2007; Hsu et al., 2004).

Lee and Matsikoudis (2009) bridge the gap between Dennis and Kahn, showing that Kahn's methods extend naturally to Dennis dataflow, embracing the notion of firing. This is done by establishing the relationship between a firing function and the Kahn process implemented as a sequence of firings of that function. A consequence of this analysis is a formal characterization of firing rules and firing functions that preserve determinacy.

Kahn and MacQueen (1977) called the processes in a PN network coroutines for an interesting reason. A **routine** or **subroutine** is a program fragment that is "called" by another program. The subroutine executes to completion before the calling fragment can continue executing. The interactions between processes in a PN model are more symmetric, in that there is no caller and callee. When a process performs a blocking read, it is in a sense invoking a routine in the upstream process that provides the data. Similarly, when it performs a write, it is in a sense invoking a routine in the downstream process to process the data. But the relationship between the producer and consumer of the data is much more symmetric than with subroutines.

When using a conventional Ptolemy II actor with the PN director (vs. a custom-written actor), the actor is automatically wrapped in an infinite loop that fires it until either the model halts (see Section 4.1.2 below) or the actor terminates (by returning false from its postfire method). When the actor accesses an input, it blocks until input is available. When it sends an output, the output token goes into a **FIFO queue** (first-in, first-out) that is (conceptually) unbounded. The tokens will be eventually delivered in order to the destination actors.

An interesting subtlety arises when using actors that behave differently depending on whether input tokens are available on the inputs. For example, the AddSubtract actor will add all available tokens on its *plus* port, and subtract all available tokens on its *minus* port. When executing under the PN director, this actor will not complete its operation until it has received one token from every input channel. This is because accesses to the inputs are blocking. When the actor asks whether a token is available on the input (using the `hasToken` method), the answer is always yes! When it then goes to read that token (using the `get` method of the input port), it will block until there actually is a token available.

Just like dataflow, PN poses challenging questions about boundedness of buffers and about deadlock. PN is expressive enough that these questions are undecidable. An elegant solution to the boundedness question is given by Parks (1995) and elaborated by Geilen and Basten (2003). The solution given by Parks is the one implemented in Ptolemy II.

4.1.1 Concurrent Firings

PN models are particularly useful when there are actors that do not return immediately when fired. The **InteractiveShell** actor, for example, found in `Sources→SequenceSources`, opens a window (like the top and bottom windows in Figure 4.1) into which a user can enter information. When the actor fires, it displays in the window whatever input it has received, followed by a prompt (which defaults to ">>"). It then waits for the user to enter a new value. When the user types a value and hits return, the actor outputs an event containing that value.

When this actor fires in response to an input, the firing will not complete until a user enters a value in the window that is opened. If a dataflow director is used (or any other director that fires actors one at a time), then the entire model will block waiting for user

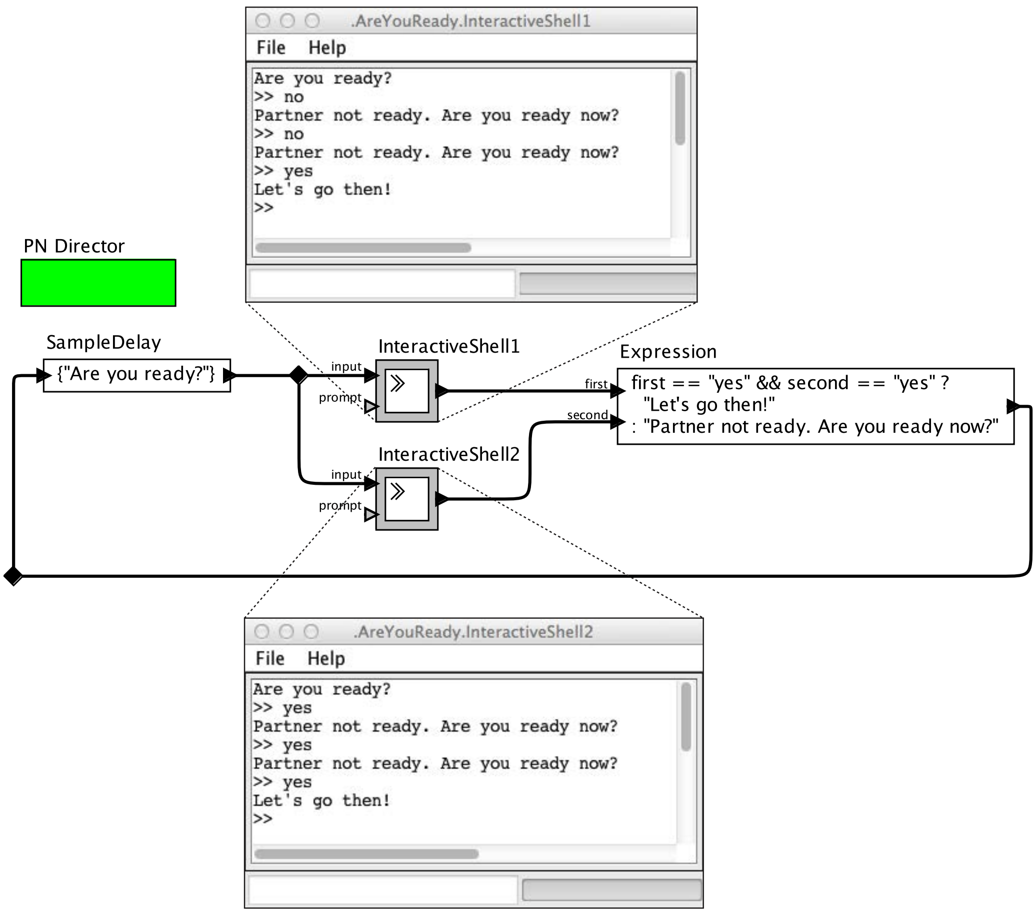


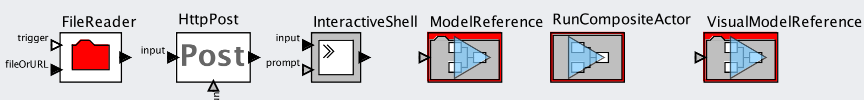
Figure 4.1: A model with two instances of InteractiveShell, each of which blocks waiting for user input. [online]

input from just one InteractiveShell. No other actor can fire. For some models, this will be a problem.

Example 4.1: Consider the model in Figure 4.1. This model emulates a dialog with two users where it is seeking concurrence from both users before proceeding with some other action. In this case, it simply asks each user "Are you ready?" It waits for one response from each user, and if both users have respond "yes," then it responds "Let's go then!" In this PN model, each instance of InteractiveShell executes in its own thread. If this model were executed instead using the SDF director, then the SDF director would fire one of these first, and would not even ask the second user the question until the first one has responded.

Sidebar: Actors With Unbounded Firings

Below are a few actors that benefit particularly from use of the PN director because they do not (necessarily) return immediately when fired:



- FileReader reads a file on the local computer or a URL (uniform resource locator from the Internet. In the latter case, the actor may block for an indeterminate amount of time while the server responds.
- HttpPost posts a record to a URL on the Internet, emulating what a browser typically does when a user fills in an online form. The actor may block for an indeterminate amount of time while the server responds.
- InteractiveShell opens a window and waits for user input.
- ModelReference executes a referenced model to completion (or indefinitely, if the model does not terminate).
- RunCompositeActor executes a contained model to completion (or indefinitely, if the model does not terminate).
- VisualModelReference opens a Vergil window to display a referenced model and executes the model to completion (or indefinitely, if the model does not terminate).

In the previous example, the InteractiveShell actor interacts with the world outside the PN model. It performs I/O, and in particular, it blocks while performing the I/O, waiting for actions from the outside world. Another example that has this property is considered in Exercise 5, which considers actors that collect data from sensors.

As explained above, a key property of Kahn process networks is that models are deterministic. It may seem odd to assert that the model in Figure 4.1 is deterministic, however. Clearly, the sequence of values generated by the Expression actor, for example, differs from run to run, because it depends on what users type into the dialog windows that open. Indeed, determinism requires that every actor implement a mathematical *function* from input sequences to output sequences. Strictly speaking, InteractiveShell does not implement such a function. In two different runs, the same sequence of inputs will not generate the same sequence of outputs, because the outputs depend on what a user types. Nevertheless, the Kahn property is valuable, because it asserts that if you fix the user behavior, then the behavior of the model is unique and well defined. If on two successive runs, two users type exactly the same sequence of responses, then the model will behave exactly the same way. The behavior of the model depends *only* on the behavior of the users, and not on the behavior of the thread scheduler.

In Ptolemy II, it is possible to build process networks that are nondeterminate using a special actor called a NondeterministicMerge (see box on page 143). Such models can be quite useful.

Example 4.2: Consider the example in Figure 4.2. This model emulates a chat interaction between two users, where users can provide input in any order. In this model, users type their inputs into the windows opened by the InteractiveShell actors, and their inputs are merged and displayed by the Display actor.

In this model, it is *not* true that if on two successive runs, users type the same thing, that the results displayed will be the same. Indeed, if two users type things nearly simultaneously, then the order in which their text appears at the output of the NondeterministicMerge is undefined. Either order is correct. The order that results will depend on the thread scheduler, not on what the users have typed. This contrasts notably with the model in Figure 4.1.

The nondeterminism introduced in the previous model can be quite useful. But it comes at a cost. Models like this are much harder to test, for example. There is no single correct result of execution. For this reason, NondeterministicMerge should be used with caution, and only when it is really needed. If a model can be built using deterministic mechanisms, then it should be built using deterministic mechanisms.

Although the previous example emulates a chat session between two clients, it is not a realistic implementation of a chat application. The two interactive shell windows are opened on the same screen and run in the same process, so two distinct users cannot prac-

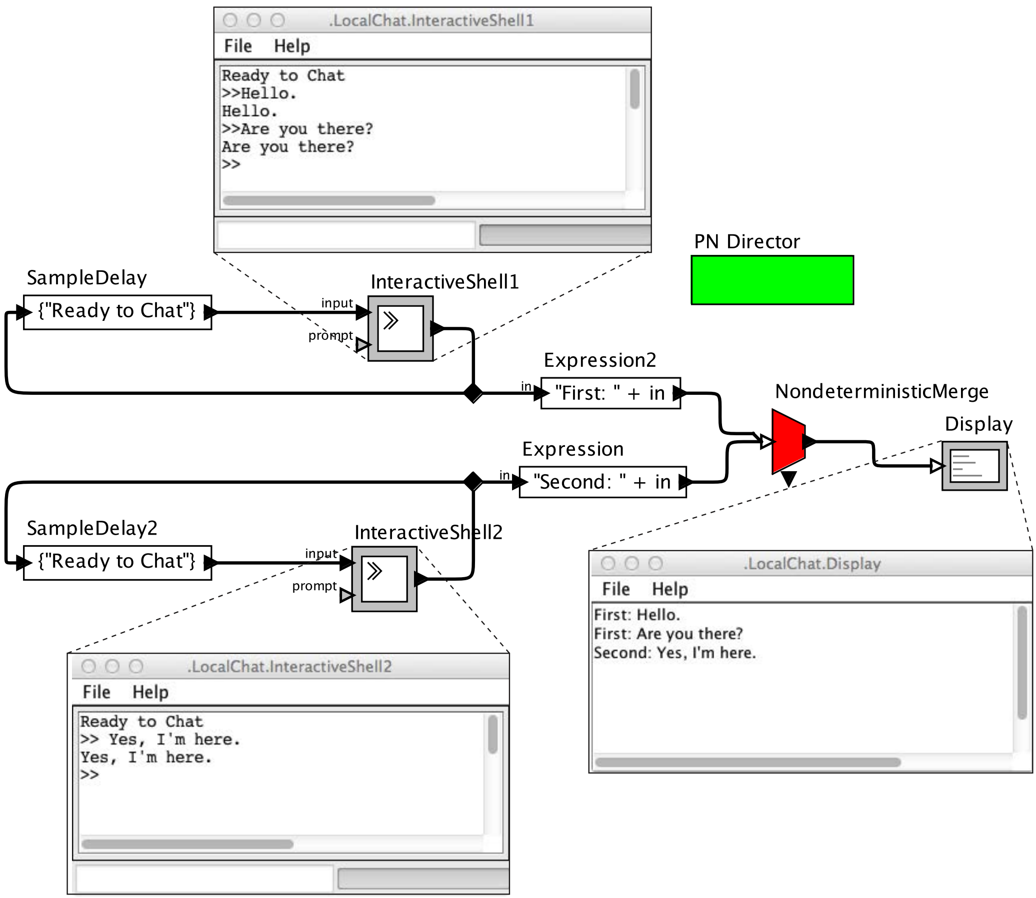


Figure 4.2: A model using a NondeterministicMerge to create a nondeterministic process network. [online]

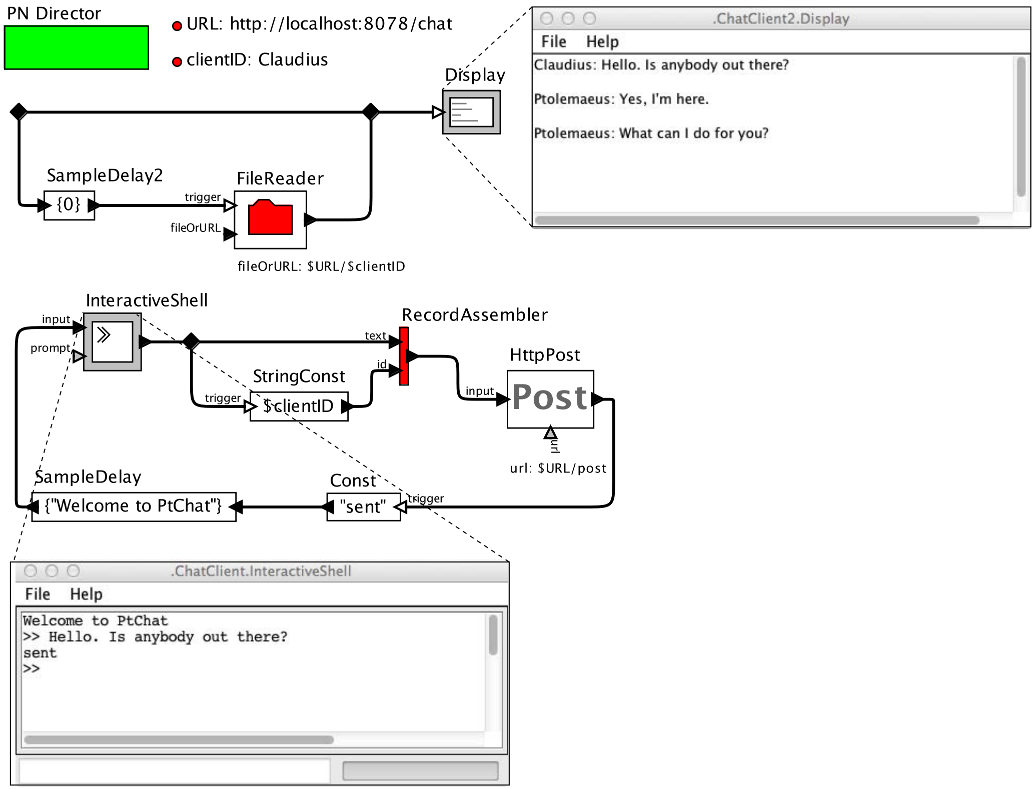


Figure 4.3: A model that implements a chat client. [online]

tically chat. PN can be used to build to build an actual chat client using only deterministic mechanisms.

Example 4.3: The model in Figure 4.3 implements a chat client. This model assumes that a server has been deployed at the URL specified by the *URL* parameter of the model. An implementation of such a server is developed in Exercise 1 of Chapter 16.

This client assumes that the server provides two interfaces. An HTTP Get request is used to retrieve chat text that will include text entered by the user of this model interleaved with text provided by any other participant in the chat session. The upper feedback loop in Figure 4.3 issues this HTTP Get using a FileReader actor. Normally, the server will not respond immediately, but rather will wait until it has chat data to provide. Hence, the FileReader actor will block, waiting for a response. Upon receiving a response, the model will display the response using the Display actor, and then issue another HTTP Get request.

This technique, where Get requests block until there are data to provide, is called **long polling**. In effect, it provides a simple form of push technology, where the server pushes data to a client when there are data to provide. Of course, it only works if the client has issued a Get request, and if the client can patiently wait for a response, rather than, say, timing out.

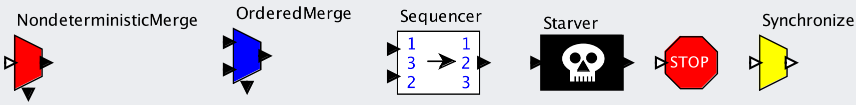
The lower feedback loop in Figure 4.3 is used for this client to supply chat text to the chat session. It uses an InteractiveShell to provide a window into which a user can enter text. When the user provides text, it assembles a record containing the text and user ID and posts the record to the server using an HttpPost actor. When the server replies, the model provides a confirmation to the user, "sent," and waits for the user to enter new text.

In the meantime, the server will normally respond to the HTTP Post by issuing a reply to all pending instances of HTTP Get. In the example execution shown in Figure 4.3, the local user, Claudius, first types "Hello. Is anybody there?" Another user, Ptolemaeus, somewhere else on the Internet, replies "Yes, I'm here." The remote user then further types "What can I do for you?" That text is "pushed" to this chat client using the long polling technique, and hence appears on the display spontaneously.

The previous example shows a use of PN to create two simultaneously executing tasks, each of which can block. This model is deterministic, in that for any sequence of outputs produced by the FileReader and InteractiveShell, all signals in the model are uniquely defined. The behavior of the model, therefore, depends *only* on user input, and not on thread scheduling, unlike the example in Figure 4.2. This form of determinism is very valuable. Among the many benefits is that models can be tested by defining input sequences and checking the responses.

Sidebar: Useful Actors for PN Models

A few actors that are particularly useful in PN models are shown below:



All of these can be found in `DomainSpecific→ProcessNetworks`, but some of them also appear in `Actors→FlowControl`.

- **NondeterministicMerge**. Merge sequences of tokens by arbitrarily interleaving their tokens onto a single stream.
- **OrderedMerge**. Merge two monotonically increasing sequences of tokens into one monotonically increasing sequence of tokens.
- **Sequencer**. Take a stream of input tokens and stream of sequence numbers and re-order the input tokens according to the sequence numbers. On each iteration, this actor reads an input token and a sequence number. The sequence numbers are integers starting with zero. If the sequence number is the next one in the sequence, then the token read from the *input* port is produced on the *output* port. Otherwise, it is saved until its sequence number is the next one in the sequence.
- **Starver**. Pass input tokens unchanged to the output until a specified number of tokens have been passed. At that point, consume and discard all further input tokens. This can be used in a feedback loop to limit the execution to a finite data set.
- **Stop**. Stop execution of a model when a true token is received on any input channel.
- **Synchronizer**. Synchronize multiple streams so that they produce tokens at the same rate. That is, when at least one new token exists on every input channel, exactly one token is consumed from each input channel, and the tokens are output on the corresponding output channels.

4.1.2 Stopping Execution of a PN Model

A PN model in Ptolemy II executes until one of the following occurs:

- All actors have terminated. An actor terminates when its postfire method returns false. Many actors in the Ptolemy II library have a *firingCountLimit* parameter (e.g., the Const actor). Setting this parameter to a positive integer will cause the actor's process to terminate after the actor has fired the specified number of times.
- All processes are blocked on reads of input ports. This is a deadlock, similar to the ones that can occur in dataflow models. A deadlock may occur due to an error in the model, or it may be deliberate. For example, the *firingCountLimit* parameter mentioned above and the Starver actor (see box on page 143) can be used to create a **starvation** condition, where actors that have not terminated are blocked on reads that will never be satisfied. Despite the pejorative tone of the words "deadlock" and "starvation," these are perfectly reasonable and useful techniques for terminating the execution of a PN model.
- The Stop actor reads a true-valued input token on its one input port. Upon reading this input, the Stop actor will coordinate with the director to stop all executing threads. Specifically, any thread that is blocked on a read or write of a port will terminate immediately, and any thread that is not blocked will terminate on its next attempt to perform a read or write. The Stop actor can be found in the `Actors→FlowControl→ExecutionControl` library.
- A buffer overflows. This occurs when the number of unconsumed tokens on a communication channel exceeds the value of the *maximumQueueCapacity* parameter of the director. Note that if you set *maximumQueueCapacity* to 0 (zero), then this will not occur until the operating system denies the Ptolemy system additional memory, which typically occurs when you have run out system memory.
- An exception occurs in some actor process. Other threads are terminated in a manner similar to what the Stop actor does.
- A user presses the stop button in Vergil. All threads are terminated in a manner similar to what the Stop actor does.

These are the only mechanisms for stopping an execution. How to use them is explored in Exercise 6.

Even so, there are some limitations. The FileReader actor, for example, used in Figure 4.3, sadly, cannot be interrupted if it is blocked on a read. Stopping the model in Figure 4.3, therefore, is difficult. If you hit the stop button, it will eventually time out, but the wait may be considerable. This appears to be a limitation in the underlying java.net.URLConnection class that is used by this actor.

4.2 Rendezvous

In the **Rendezvous** domain in Ptolemy II, like PN, each actor executes in its own thread. Unlike PN, communication between actors is by **rendezvous** rather than message passing with unbounded FIFO queues. Specifically, when an actor is ready to send a message via an output port, it blocks until the receiving actor is ready to receive it. Similarly if an actor is ready to receive a message via an input port, it blocks until the sending actor is ready to send it. Thus, this domain realizes both **blocking writes** and blocking reads.

This domain supports both conditional and multiway rendezvous. In **conditional rendezvous**, an actor is willing to rendezvous with any one of several other actors.[2] In **multiway rendezvous**, an actor requires rendezvous with multiple other actors at the same time.[3] When using conditional rendezvous, the choice of which rendezvous occurs is nondeterministic, in general, since which rendezvous occurs will generally depend on the thread scheduler.

The Rendezvous domain is based on the communicating sequential processes (**CSP**) model first proposed by Hoare (1978) and the calculus of communicating systems (**CCS**), given by Milner (1980). It also forms the foundation for the **Occam** programming language (Galletly, 1996), which enjoyed some success for a period of time in the 1980s and 1990s for programming parallel computers.

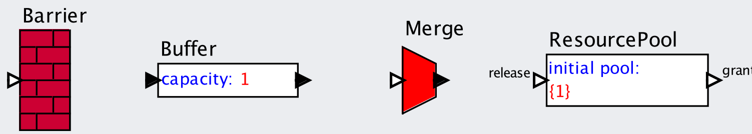
Rendezvous-based communication is also known as **synchronous message passing**, but we avoid this term to avoid confusion with the SR (synchronous-reactive) domain, described in Chapter 5, and the SDF domain, described in Chapter 3.

[2]For those fameliar with the Ada language, this is similar to the select statement.

[3]Multiway rendezvous is also called **barrier synchronization**, since it blocks each participating thread until all participating threads have reached a barrier point, indicated by a send or a get via a port.

Sidebar: Useful Actors for Rendezvous Models

A few particularly useful actors in Rendezvous models are shown below:



- **Barrier**. A barrier synchronization actor. This actor will accept inputs only when the sending actors on *all* input channels are ready to send.
- **Buffer**. A FIFO buffer. This actor buffers data provided at the input, sending it to the output when needed. The actor is willing to rendezvous with the output whenever the buffer is not empty. It is willing to rendezvous with the input as long as the buffer is not full. Inputs are delivered to the output in FIFO (first-in, first-out) order. You can specify a finite or infinite capacity for the buffer.
- **Merge**. A conditional rendezvous. This actor merges any number of input sequences onto one output sequence. It begins with a rendezvous with any input. When it receives an input, it will then rendezvous with the output. After successfully delivering the input token to the output, it returns again to being willing to rendezvous with any input.
- **ResourcePool**. A resource contention manager. This actor manages a pool of resources, where each resource is represented by a token with an arbitrary value. Resources are granted on the *grant* output port and released on the *release* input port. These ports are both multiports, so resources can be granted to multiple users of the resources, and released by multiple actors. The initial pool of resources is provided by the *initialPool* parameter, which is an array of arbitrary type. At all times during execution, this actor is ready to rendezvous with any other actor connected to its *release* input port. When such a rendezvous occurs, the token provided at that input is added to the resource pool. In addition, whenever the resource pool is non-empty, this actor is ready to rendezvous with any actor connected to its *grant* output port. If there are multiple such actors, this will be a conditional rendezvous, and hence may introduce nondeterminism into the model. When such an output rendezvous occurs, the actor sends the first token in the resource pool to that output port and removes that token from the resource pool.

4.2.1 Multiway Rendezvous

In multiway rendezvous, an actor performs a rendezvous with multiple other actors at the same time. Two mechanisms are provided by the Rendezvous director for this. First, if an actor sends an output to multiple other actors, the communication forms a multiway rendezvous. All destination actors must be ready to receive the token before any destination actor will receive it. And the sender will block until all destination actors are ready to receive the token.

Example 4.4: The model in Figure 4.4 illustrates multiway rendezvous. In this example, the Ramp is sending an increasing sequence of integers to the Display. However, the transfer is constrained to occur only when the Sleep actor reads inputs, because in the Rendezvous domain, sending data to multiple recipients via a relation is accomplished via a multiway rendezvous. The **Sleep** actor reads data, then sleeps an amount of time given by its bottom input before reading the next input data token. In this case, it will wait a random amount of time (given by the **Uniform** random-number generator) between input readings. This has the side effect of constraining the transfers from the Ramp to the Display to occur with the same random intervals.

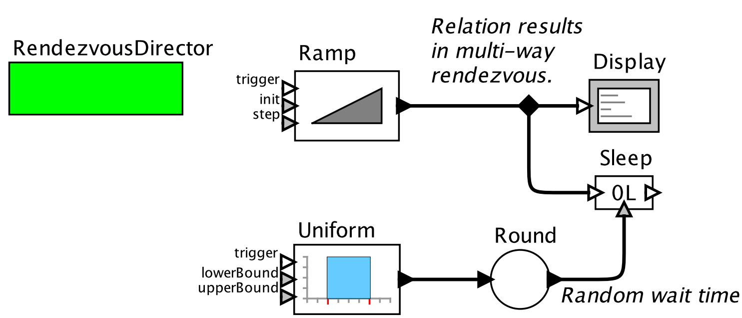


Figure 4.4: An illustration of multiway rendezvous, where the timing of the communication between the Ramp and the Display is controlled by the Sleep actor. [online]

The Barrier actor (see box on page 146) also performs multiway rendezvous.

Example 4.5: The model in Figure 4.5 illustrates multiway rendezvous using the Barrier actor. In this example, the two Ramp actors are sending increasing sequences of integers to the Display actors. Again, the transfer is constrained to occur only when both the Barrier actor and the Sleep actor read inputs. Thus, a multiway rendezvous between the two Ramp actors, the two Display actors, the Barrier actor, and the Sleep actor constrains the two transfers to the Display actors to occur simultaneously.

4.2.2 Conditional Rendezvous

In conditional rendezvous, an actor is willing to rendezvous with any one of several other actors. Usually, this results in nondeterministic models.

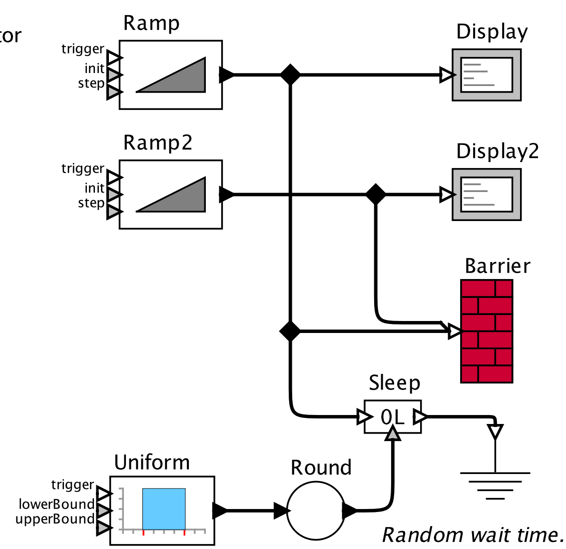


Figure 4.5: An illustration of multiway rendezvous using the Barrier actor. [online]

Example 4.6: The model in Figure 4.6 illustrates conditional rendezvous. This model uses the Merge actor (see box on page 146). The top Ramp actor will produce the sequence 0, 1, 2, 3, 4, 5, 6, 7 on its output. The bottom Ramp will produce the sequence -1, -2, -3, -4, -5, -6, -7, -8. The Display actor will display a nondeterministic merging of these two sequences.

The example in Figure 4.6 includes a parameter *SuppressDeadlockReporting* with value true. The Ramp actors starve the model by specifying a finite *firingCountLimit*, thus providing a stopping condition similar to the ones we would use with PN (see Section 4.1.2). By default, the director will report the deadlock, but with the *SuppressDeadlockReporting* parameter, it silently stops the execution of the model. This parameter indicates that the deadlock is a normal termination and not an error condition.

Suppose that, unlike the previous example, we wish to deterministically interleave the outputs of the two Ramp actors, alternating their outputs to get the sequence 0, -1, 1, -2, 2, ... One way to accomplish that, due to Arbab (2006), is shown in Figure 4.7. This model relies on the fact that the input to the Buffer actor (see box on page 146) participates in a multiway rendezvous with both instances of Ramp and the top channel of the Merge actor. Since it has capacity one, it forces this rendezvous to occur before it provides an input to

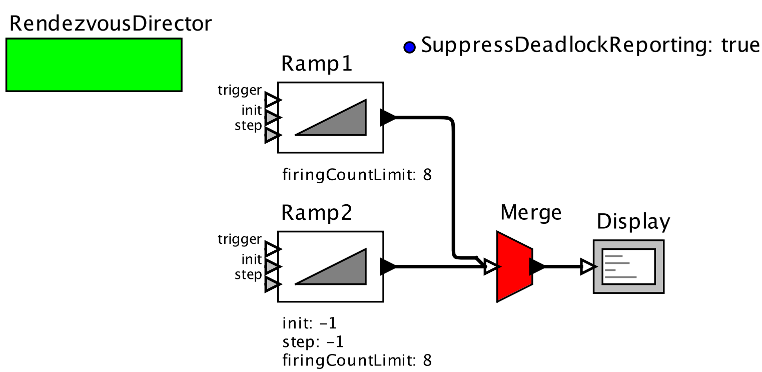


Figure 4.6: An illustration of conditional rendezvous for nondeterministic merge. [online]

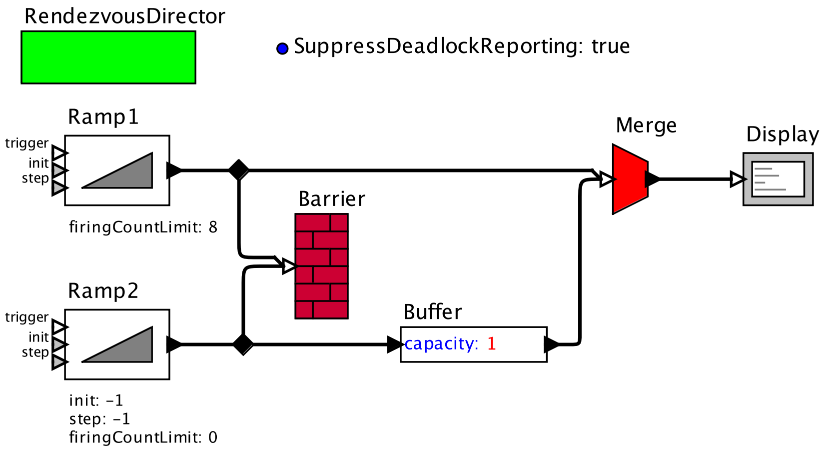


Figure 4.7: An illustration of conditional rendezvous used to create a deterministic merge. [online]

the bottom channel of the merge, and it blocks subsequent instances of this rendezvous until after it has provided the bottom input to the Merge.

Although this model is extremely clever, it is using nondeterministic mechanisms to accomplish deterministic aims. In fact, it is easy to construct a much simpler model that accomplishes the same goal without any nondeterministic mechanisms (see Exercise 7).

4.2.3 Resource Management

The conditional rendezvous mechanism provided by the Rendezvous director is ideally suited to resource management problems, where actors compete for shared resources. Generally, nondeterminism is acceptable and expected for such models. The ResourcePool actor (see box on page 146) is ideal for such applications.

Example 4.7: The model in Figure 4.8 illustrates resource management where a pool (in this case containing only one resource) provides that resource nondeterministically to one of two Sleep actors. The resource is represented by an integer that initially has value 0. This value will be incremented each time the resource is

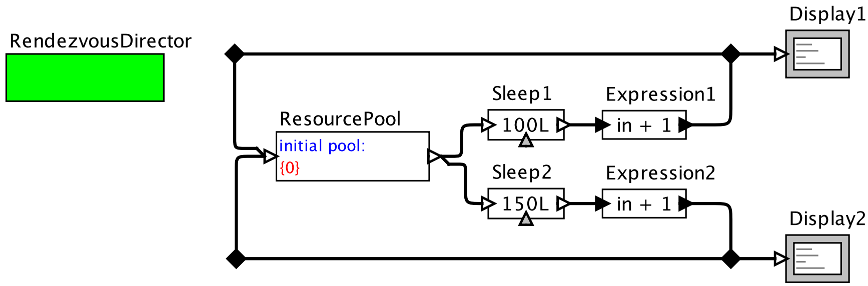


Figure 4.8: An illustration of conditional rendezvous for resource management. [online]

used. The Sleep actor that gets the resource holds it for a fixed amount of time (100 and 150 ms, respectively). After this time, it releases the resource, sending it to an Expression actor, which increments the value of the resource and then returns it to the resource pool.

The input and output ports of the ResourcePool actor both realize a conditional rendezvous. Hence, it is nondeterministic which Sleep actor will acquire the resource when both are ready for the resource. Note that there is no assurance of fairness in this system, and in fact it is possible for only one of the two Sleep actors to get access to the resource.

4.3 Summary

The two domains described in this chapter, PN and Rendezvous, both execute each actor in a model in its own thread. A PN model is deterministic as long as it does not include an instance of NondeterministicMerge. The Rendezvous domain is not generally deterministic. PN is particularly useful when models include actors that may block for indeterminate amounts of time, and where we don't wish to block the entire the model when this occurs. Rendezvous is particularly useful for resource management problems, where there is contention for limited resources, and for models where the timing of concurrent actions needs to be synchronized.

Exercises

The purpose of these exercises is to develop some intuition about the process networks model of computation and how programming with it differs from programming with an imperative model.[4] For all of the following exercises, you should use the PN Director and "simple" actors to accomplish the task. In particular, the following actors are sufficient:

- Ramp and Const (in `Sources`)
- Display and Discard (in `Sinks`)
- BooleanSwitch and BooleanSelect (in `FlowControl→BooleanFlowControl`)
- SampleDelay (in `FlowControl→SequenceControl`)
- Comparator, Equals, LogicalNot, or LogicGate (in `Logic`)

Feel free to use any other actor that you believe to be "simple." Also, feel free to use any other actors, simple or not, for testing your composite actors, but stick to simple ones for the implementation of the composite actors.

1. The SampleDelay actor produces initial tokens. In this exercise, you will create a composite actor that consumes initial tokens, and hence be thought of as a negative delay.

   (a) Create a PN model containing a composite actor with one input port and one output port, where the output sequence is the same as the input sequence except that the first token is missing. That is, the composite actor should discard the first token and then act like an identity function. Demonstrate by some suitable means that your model works as required.

   (b) Augment your model so that the number of initial tokens that are discarded is given by a parameter of the composite actor. **Hint:** It may be useful to know that the expression language[5] (see Chapter 13) includes a built-in function `repeat`, where, for example,

[4] You may want to run vergil with the -pn option, which gives you a subset of Ptolemy II that is more than adequate to do these exercises. To do this on the command line, simply type "`vergil -pn`". If you are running Ptolemy II from Eclipse, then in the toolbar of the Java perspective, select Run Configurations. In the Arguments tab, enter -pn.

[5] Note that you can easily explore the expression language by opening an ExpressionEvaluator window, available in the [`File→New`] menu. Also, clicking on Help in any parameter editor window will provide documentation for the expression language.

```
repeat(5, 1) = {1, 1, 1, 1, 1}
```

2. This problem explores operations on a stream of data that depend on the data in the stream.
   (a) Create a PN model containing a composite actor with one input port and one output port, where the output sequence is the same as the input sequence except that any consecutive sequence of identical tokens is replaced by just one token with the same value. That is, redundant tokens are removed. Demonstrate by some suitable means that your model works as required.
   (b) Can your implementation run forever with bounded buffers? Give an argument that it can, or explain why there is no implementation that can run with bounded buffers.

3. Create an implementation of an OrderedMerge actor using only "simple" PN actors. (Note that there is a Java implementation of OrderedMerge, which you should not use, see box on page 143.) Your implementation should be a composite actor with two input ports and one output port. Given any two numerically increasing sequences of tokens on the input ports, your actor should merge these sequences into one numerically increasing sequence without losing any tokens. If the two sequences contain tokens that are identical, then the order in which they come out does not matter.

4. In Figure 4.9 is a model that generates a sequence of numbers known as the **Hamming numbers**. These have the form $2^n 3^m 5^k$, and they are generated in numerically increasing order, with no redundancies. This model can be found in the PN demos (labeled as OrderedMerge). Can this model run forever with bounded buffers? Why or why not?

   For this problem, assume that the data type being used is unbounded integers, rather than what is actually used, which is 32 bit integers. With 32 bit integers, the numbers will quickly overflow the representable range of the numbers, and wrap around to negative numbers.

5. A common scenario in embedded systems is that multiple sensors provide data at different rates, and the data must be combined to form a coherent view of the physical world. In general, this problem is called **sensor fusion**. The signal processing involved in forming a coherent view from noisy sensor data can be quite sophisticated, but in this exercise we will focus not on the signal processing, but rather on the concurrency and logical control flow. At a low level, sensors are connected to

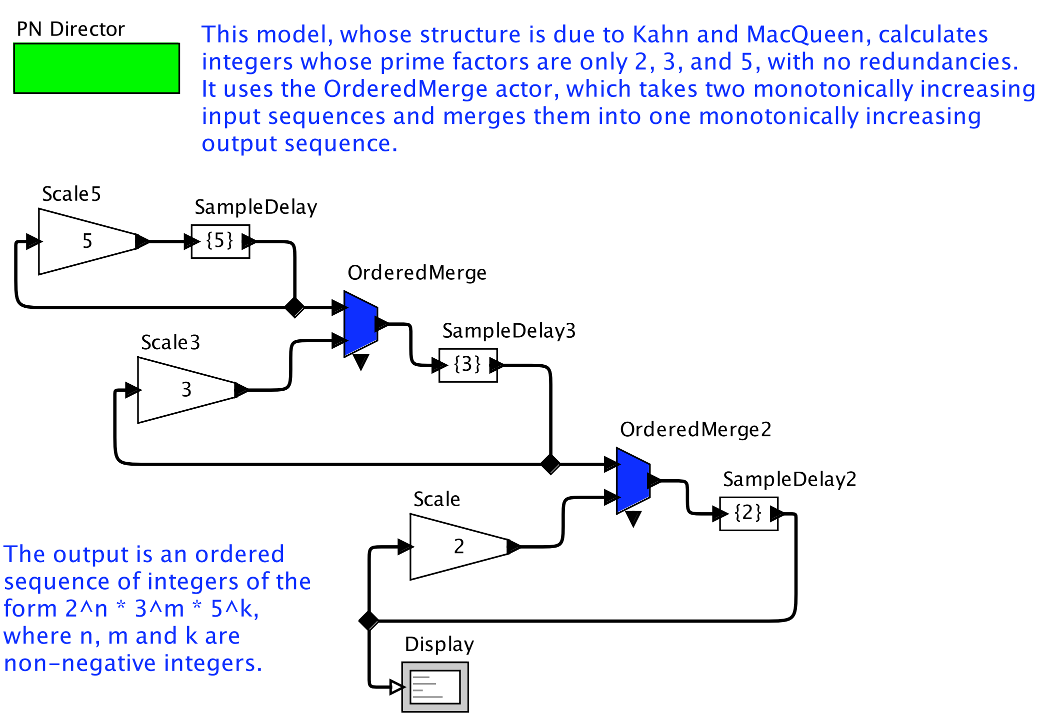


Figure 4.9: Model that generates a sequence of Hamming numbers. [online]

embedded processors by hardware that will typically trigger processor interrupts, and interrupt service routines will read the sensor data and store it in buffers in memory. The difficulties arise when the rates at which the data are provided are different (they may not even be related by a rational multiple, or may vary over time, or may even be highly irregular).

Assume we have two sensors, SensorA and SensorB, both making measurements of the same physical phenomenon that happens to be a sinusoidal function of time, as follows:

$$\forall\, t \in \mathbb{R}, \quad x(t) = sin(2\pi t/10)$$

Assuming time t is in seconds, this has a frequency of 0.1 Hertz. Assume further that the two sensors sample the signal with distinct sampling intervals to yield the

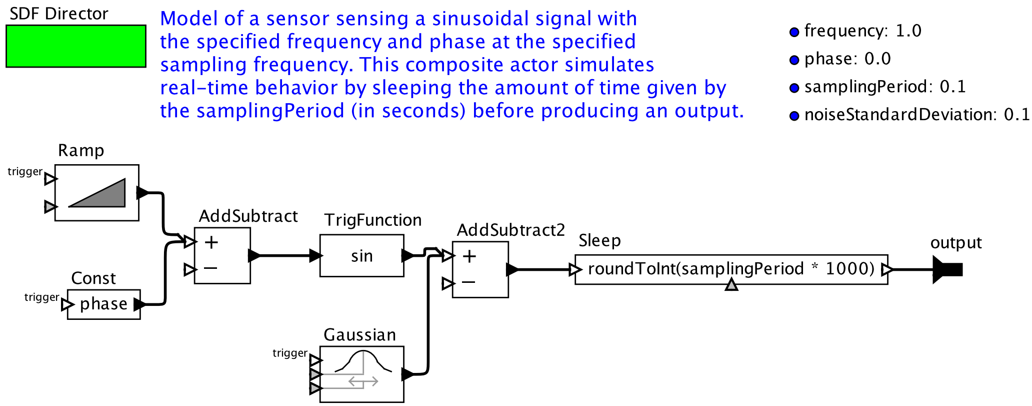


Figure 4.10: Model of a real-time sensor. [online]

following measurements:

$$\forall\, n \in \mathbb{Z}, \quad x_A(n) = x(nT_A) = sin(2\pi n T_A/10),$$

where T_A is sampling interval of SensorA. A similar set of measurements is taken by SensorB, which samples with period T_B.

A model of such a sensor for use with the PN director of Ptolemy II is shown in figure 4.10. You can create an instance of that sensor in Vergil by invoking the [`Graph→Instantiate Entity`] menu command, and filling in the boxes as follows:

```
class: SensorModel
location (URL): http://embedded.eecs.berkeley.edu/
                concurrency/models/SensorModel.xml
```

Create two instances of the sensor in a Ptolemy II model with a PN director.

The sensor has some parameters. The *frequency* you should set to 0.1 to match the equations above. The *samplingPeriod* you should set to 0.5 seconds for one of the sensor instances, and 0.75 seconds for the other. You are to perform the following experiments.[6]

[6] You may find the actors described in the sidebars on pages 106, 107, and 119 useful.

(a) Connect each sensor instance to its own instance of the SequencePlotter. Execute the model. You will likely want to change the parameters of the SequencePlotter so that *fillOnWrapup* is `false`, and you will want to set the *X Range* of the plot to, say, "0.0, 50.0" (do this by clicking on the second button from the right at the upper right of each plot). Describe what you see. Do the two sensors accurately reflect the sinusoidal signal? Why do they appear to have different frequencies?

(b) A simple technique for sensor fusion is to simply average sensor data. Construct a model that averages the data from the two sensors by simply adding the samples together and multiplying by 0.5. Plot the resulting signal. Is this signal an improved measurement of the signal? Why or why not? Will this model be able to run forever with bounded memory? Why or why not?

(c) The sensor fusion strategy of averaging the samples can be improved by normalizing the sample rates. For the sample periods given, 0.5 and 0.75, find a way to do this in PN. Comment about whether this technique would work effectively if the sample periods did not bear such a simple relationship to one another. For example, suppose that instead of 0.5 seconds, the period on the first sensor was 0.500001.

(d) When sensor data is collected at different rates without a simple relationship, one technique that can prove useful is to create time stamps for the data and to use those time stamps to improve the measurements. Construct a model that does this, with the objective simply of creating a plot that combines the data from the two sensors in a sensible way.

6. In this problem, we explore how to use the mechanisms of Section 4.1.2 to deterministically halt the execution of a PN model. Specifically, in each case, we consider a Source actor feeding a potentially infinite sequence of data tokens to a Display actor. We wish to make this sequence finite with a specific length, and we wish to ensure that the Display actor displays every element of the sequence.

   (a) Suppose that you have a Source actor with one output port and no parameters whose process iterates forever producing outputs. Suppose that its outputs are read by a Display actor, which has one input port and no output ports. Find a way to use the Stop actor to deterministically stop the execution, or argue that there is no way to do so. Specifically, the Source actor should produce a specified number of outputs, and every one of these outputs should be consumed and displayed by the Display actor before execution halts.

(b) Most Source actors in Ptolemy II have a *firingCountLimit* parameter that limits the number of outputs they produce. Show that this can be used to deterministically halt the execution without the help of a Stop actor.

(c) Many Source actors in Ptolemy II have *trigger* input ports. If these inputs are connected, then the actor process will read a value from that input port before producing each output. Show how to use this mechanism, with or without the Stop actor, to achieve our goal of deterministically halting execution, or argue that it is not possible to do so. Again, the Source should produce a prespecified amount of data, and the Display should consume and display all of that data. You may use Switch, Select, or any other reasonably simple actor. Be sure to explain each actor you use, unless you are sure it is exactly the actor provided in the Vergil library.

7. Figure 4.7 shows a model that deterministically interleaves the outputs of two Ramp actors. That model uses a nondeterministic mechanism (the conditional rendezvous of the Merge actor), and then carefully regulates the nondeterminism using a multiway rendezvous and a Buffer actor. The end result is deterministic. However, the same objective (deterministically interleaving two streams in an alternating, round-robin fashion) can be accomplished with purely deterministic mechanisms. Construct a Rendezvous model that does this. **Hint**: Your model will probably work unchanged with using the PN or SDF directors instead of Rendezvous.

5

Synchronous-Reactive Models

Stephen A. Edwards, Edward A. Lee, Stavros Tripakis, Paul Whitaker
In memory of Paul Caspi

Contents

The **synchronous-reactive** (**SR**) model of computation is designed for modeling systems that involve synchrony, a fundamental concept in concurrent systems (see sidebar on page 165). It is an appropriate choice for modeling applications with complicated control logic where many things are happening at once (concurrently) and yet determinism and precise control are important. Such applications include embedded control systems, where safety must be preserved. SR systems are good at orchestrating concurrent actions, managing shared resources, and detecting and adapting to faults in a system. Whereas dataflow models are good for managing streams of data, SR systems are good at managing sporadic data, where events may be present or absent, and where the absence of events has meaning (more than just transport delay). For example, detecting the absence of an event may be an essential part of a **fault management** system. SR is also a good model of computation for coordinating finite state machines, described in Chapters 6 and 8, which can be used to express the control logic of the individual actors that are concurrently executed.

The Ptolemy II SR domain has been influenced by the family of so-called **synchronous languages** (see sidebar on page 166) and in particular **dataflow synchronous languages** such as **Lustre** (Halbwachs et al., 1991) and **Signal** (Benveniste and Le Guernic, 1990). SR primarily realizes the model of **synchronous block diagrams** as described by Edwards and Lee (2003b). The model of computation is closely related to synchronous digital circuits. In fact, this chapter will illustrate some of the ideas using circuit analogies, although the SR domain is intended more for modeling embedded software than circuits.

SR can be viewed as describing **logically timed** systems. In such systems, time proceeds as a sequence of discrete steps, called **reactions** or **ticks**. Although the steps are ordered, there is not necessarily a notion of "time delay" between steps like there is in discrete time systems; and there is no *a priori* notion of real time. Thus, we refer to time in this domain as **logical time** rather than discrete time.

The similarities and differences with dataflow models are:

1. Like homogeneous SDF, an iteration of an SR model consists of one iteration of each actor in the model. Each iteration of the model corresponds to one tick of the logical clock. Indeed, most of the SDF models considered in Chapter 2 could just as easily have been SR models. For example, the behavior of the channel model in Figure 2.29 and all of its variants would behave identically under the SR director.
2. Unlike dataflow and process networks, there is no buffering on the communication between actors. In SR, and output produced by one actor is observed by the destination

actors in the same tick. Unlike rendezvous, which also does not have buffered communication, SR is determinate.

3. Unlike dataflow, an input or output may be **absent** at a tick. In dataflow, the absence of an input means simply that the input hasn't arrived yet. In SR, however, an absent signal has more meaning. Its absence is not a consequence of accidents of scheduling or of the time that computation or communication may take. Instead, the absence of a signal at a tick is *defined* deterministically by the model. As a consequence, in SR, actors may react to the *absence* of a signal. This is quite different from dataflow, where actors react only the *presence* of a signal.
4. As we will explain below, in SR, an actor may be fired multiple times between invocations of postfire. That is, one iteration of an actor may consist of more than invocation of the fire method. For simple models, particularly those without feedback, you will never notice this. Sometimes, however, significant subtleties arise. We focus on such models in this chapter.

5.1 Fixed-Point Semantics

Consider a model with three actors with the structure shown in Figure 5.1(a). Let n denote the tick number. The first tick of the local clock corresponds to $n = 0$, the second to $n = 1$, etc. At each tick, each actor implements an input-output function (which typically changes from tick to tick, possibly in ways that depend on previous inputs). For example, actor 1, in tick 0, implements the function $f_1(0)$. That is, given an input value $s_1(0)$ on port *p1*, it will produce output value $s_2(0) = (f_1(0))(s_1(0))$ on output port *p5*.

At any tick, an input may be absent; in this view, "absent" is treated like any other value. The actor can respond to an absent input, and it may assert an absent output or assign the output some value compatible with the data type of the output port.

Each actor thus produces a sequence of values (or absent values), one at each tick. Actor 1 produces values $s_2(0), s_2(1), \cdots$, while actor 2 produces $s_1(0), s_2(1), \cdots$, and actor 3 produces $s_3(0), s_3(1), \cdots$, where any of these can be absent. The job of the SR director is to find these values (and absences). This is what it means to execute the model.

As illustrated in Figures 5.1(b) through (d), any SR model may be rearranged so that it becomes a single actor with function $f(n)$ at tick n. The domain of this function is a tuple of values (or absences) $s(n) = (s_1(n), s_2(n), s_3(n))$. So is the codomain. Therefore, at

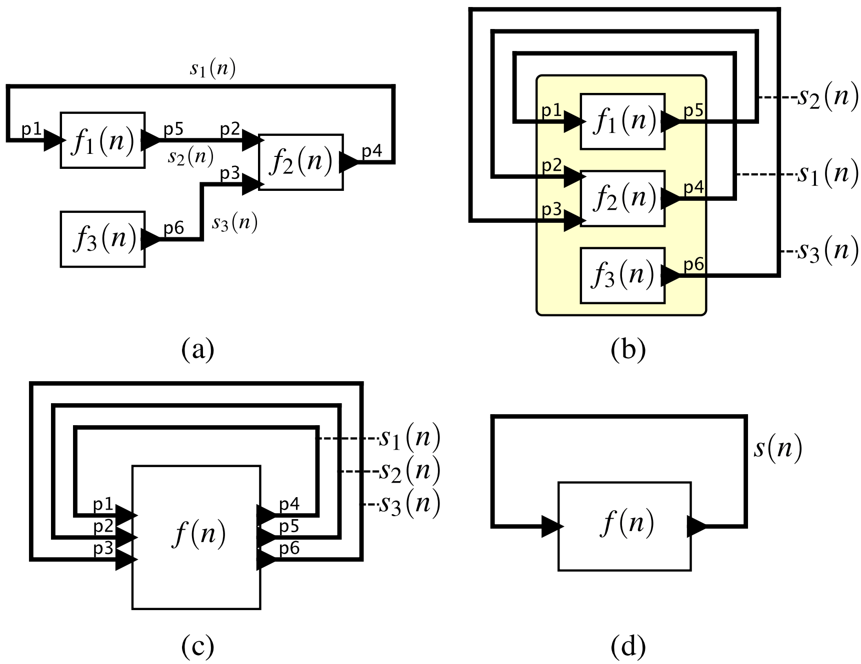


Figure 5.1: An SR model is reducible to a fixed point problem at each tick of he logical clock.

tick n, the job of the director is to find the tuple $s(n)$ such that

$$s(n) = (f(n))(s(n)).$$

At each tick of the logical clock, the SR director finds a fixed point $s(n)$ of the function $f(n)$. The subtleties around SR models concern whether such a fixed point exists and whether it is unique. As we will see, in a well constructed SR model, there will be a unique fixed point that can be found in a finite number of steps.

Logically, as SR model can be conceptualized as a **simultaneous and instantaneous** reaction of all actors at each tick of the clock. The "simultaneous" part of this asserts that the actors are reacting all the same time. The "instantaneous" part means that the outputs of each actor are simultaneous with its inputs. The inputs and outputs are all part of the same fixed point solution. This mental model is called the **synchrony hypothesis**, where one thinks of actors as executing in zero time. But it's a bit more subtle than just that,

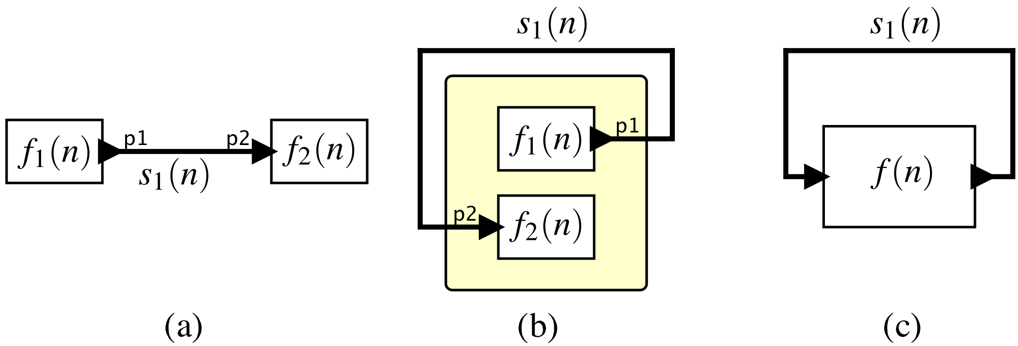


Figure 5.2: Even an SR model without feedback is reducible to a fixed point problem at each tick of he logical clock.

because when there is feedback, an actor may be reacting to an input *that is a function of its own output*. Clearly, one can get trapped in **causality** problems, where the input is not known until the output is known, and the output can't be known until the input is known. Indeed, such causality problems are the major source of subtlety in SR models.

A simple case of SR is a model without feedback, as shown in Figure 5.2. Even such a model is reducible to a fixed-point problem, but in this case it becomes a rather simple problem. The function $f_1(n)$ at tick n only needs to be evaluated once at each tick, and it immediately finds the fixed point. The function $f_2(n)$ never needs to be evaluated (from the perspective of the SR director), but the SR director fires and postfixes actor 2 anyway because of the side effects it may have (e.g. updating a display). But actor 2 plays no role in finding the fixed point.

Once the director has found the fixed point, it can then allow each actor to update its function to $f(n+1)$ in preparation for the next tick. Indeed, this is what an actor does in its postfire phase of execution. An iteration of the model, therefore, consists of some number of firings of the actors, until a fixed point is found, followed by one invocation of postfire, allowing the actor update its state in reaction to the inputs provided by the fixed point that was found. The details of how this execution is carried out are described below in Section 5.3, but first, we consider some examples.

5.2 SR Examples

5.2.1 Acyclic Models

SR models without feedback are much like homogeneous SDF models without feedback, except that signals may be absent. The ability to have absent signals can be convenient for controlling the execution of actors.

Example 5.1: Recall the if-then-else of Figure 3.10, which uses dynamic dataflow to conditionally route tokens to the computations to be done. A similar effect can be achieved in SR using When and Default (see sidebar on page 167), as shown in Figure 5.3. This model operates on a stream produced by the Ramp actor in one of

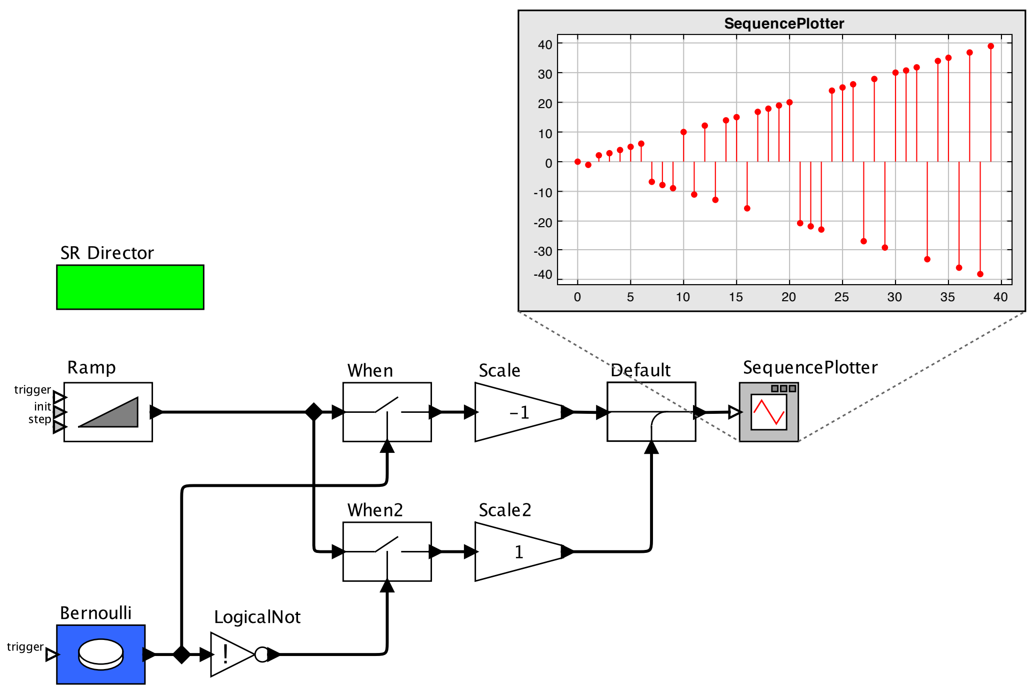


Figure 5.3: A model accomplishing conditional execution using SR. [online]

two (rather trivial) ways. Along the top path, it multiplies the stream by -1. Along the bottom path, it multiplies by 1. Such a pattern might be used, for example, to model intermittent failures in a system.

The Bernoulli actor generates a random boolean that is used to control two instances of When. The top When actor will convey the output from the Ramp to its output when the boolean is true. The bottom When actor will convey the output from the Ramp to its output when the boolean is false. When the output of a When actor is absent, then the downstream Scale actor will also have an absent output. Hence, the Default actor will have only one present input in each tick, and it will convey that input to its output. Finally, the SequencePlotter plots the result.

Whereas with dataflow models, it is possible to make wiring errors that will result in unbounded buffers, as for example in Figure 3.13, in SR, execution is always bounded. Every connection between actors stores at most one token on each tick of the clock. Hence, there is no mechanism for memory usage to become unbounded (unless, of course, an actor does so internally).

5.2.2 Feedback

More interesting SR models involve feedback (directed cycles in the graph), as in Figure 5.1. With such feedback systems, causality becomes a concern. Consider in particular the relationship between actors 1 and 2 in Figure 5.1(a). At a tick n of the logical clock, it would seem that we need to know $s_1(n)$ in order to evaluate function $f_1(n)$. But to know $s_1(n)$, it seems we need to evaluate $f_2(n)$. But to evaluate $f_2(n)$, it seems we need to know $s_2(n)$, which requires evaluating $f_1(n)$. We appear to have gotten stuck in a **causality loop**.

Causality loops must be broken by **non-strict actors**. An actor is said to be **strict** if it requires knowledge of all its inputs in order to provide outputs. If it can provide outputs without full knowledge of the inputs, then it is non-strict. The simplest non-strict actor is the NonStrictDelay (see box on page 167). It can be used to break causality loops, as illustrated in the following example.

Sidebar: About Synchrony

The general definitions of the term **synchronous** are (1) occurring or existing at the same time or (2) moving or operating at the same rate. In engineering and computer science, the term has a number of meanings that are mostly consistent with these definitions, but oddly inconsistent with one another. In referring to concurrent software using threads or processes, synchronous communication refers to a rendezvous style of communication, where the sender of a message must wait for the receiver to be ready to receive, and the receiver must wait for the sender. Conceptually, the communication occurs at the same time from the perspective of each of the two threads, consistent with definition (1). In Java, however, the keyword `synchronized` defines blocks of code that are *not* permitted to execute simultaneously, which is inconsistent with both definitions.

There is yet a third meaning of the word synchronous, which is the definition we use in this chapter. This third meaning underlies synchronous languages (see box on page 166). Two key ideas govern these languages. First, the outputs of components in a program are (conceptually) simultaneous with their inputs (this is called the synchrony hypothesis). Second, components in a program execute (conceptually) simultaneously and instantaneously. Even though this cannot occur in reality, a correct execution must behave as though it did. This interpretation is consistent with *both* definitions (1) and (2) above, since executions of components occur at the same time and operate at the same rate.

In circuit design, the word synchronous refers to a style where a clock signal that is distributed throughout a circuit causes circuit components called "latches" to record their inputs on the rising or falling edges of the clock. The time between clock edges needs to be sufficient for circuit gates between latches to settle. Conceptually, this model is very similar to the model in synchronous languages. Assuming that the gates between latches have zero delay is equivalent to the synchrony hypothesis, and global clock distribution gives simultaneous and instantaneous execution of those gates. Hence, the SR domain is often useful for modeling digital circuits.

In power systems engineering, synchronous means that electrical waveforms have the same frequency and phase. In signal processing, synchronous means that signals have the same sample rate, or that their sample rates are fixed multiples of one another. The term synchronous dataflow, described in Chapter 3.1, is based on this latter meaning of the word synchronous. This usage is consistent with definition (2).

Sidebar: Synchronous-Reactive Languages

The synchronous-reactive model of computation dates back to at least the mid-1980s, when a number of programming languages were developed. The term "reactive" comes from a distinction in computational systems between **transformational systems**, which accept input data, perform a computation, and produce output data, and **reactive systems**, which engage in an ongoing dialog with their environment (Harel and Pnueli, 1985). Manna and Pnueli (1992) state

> "The role of a reactive program ... is not to produce a final result but to maintain some ongoing interaction with its environment."

The distinctions between transformational and reactive systems led to the development of a number of innovative programming languages. The **synchronous languages** (Benveniste and Berry, 1991) take a particular approach to the design of reactive systems, in which pieces of the program react simultaneously and instantaneously at each tick of a global clock. Primary among these languages are Lustre (Halbwachs et al., 1991), Esterel (Berry and Gonthier, 1992), and Signal (Le Guernic et al., 1991). Statecharts (Harel, 1987) and its implementation in Statemate (Harel et al., 1990) also have a strongly synchronous flavor.

SCADE (Berry, 2003) (Safety Critical Application Development Environment), a commercial product of Esterel Technologies, builds on Lustre, borrows concepts from Esterel, and provides a graphical syntax in which state machines similar to those in Chapter 6 are drawn and actor models are composed synchronously. One of the main attractions of synchronous languages is their strong formal properties that facilitate formal analysis and verification techniques. For this reason, SCADE models are used in the design of safety-critical flight control software systems for commercial aircraft made by Airbus.

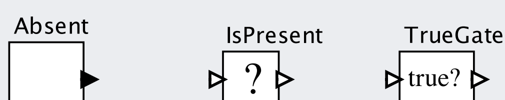
In Ptolemy II, SR is a form of **coordination language** rather than a programming language, (see also ForSyDe (Sander and Jantsch, 2004), which also uses synchrony in a coordination language). This allows for "primitives" in a system to be complex components rather than built-in language primitives. This, in turn, enables heterogeneous combinations of MoCs, since the complex components may themselves include components developed under another model of computation.

Sidebar: Domain-Specific SR Actors

The SR actors in `DomainSpecific→SynchronousReactive` below are inspired by the corresponding operators of the synchronous languages Lustre and Signal.

- **Current** outputs the most recently received non-absent input. If no input has been received, then the output is absent.
- **Default** merges two signals with a priority. If the preferred input (on the left) is present, then the output is equal to that input. If the preferred input is absent, then the output is equal to the alternate input (on the bottom, whether it is absent or not).
- **NonStrictDelay** provides a one-tick delay. On each firing, it produces on the output port whatever value it read on the input port in the previous tick. If the input was absent on the previous tick of the clock, then the output will be absent. On the first tick, the value may be given by the *initialValue* parameter. If no value is given, the first output will be absent.
- **Pre** outputs the previously received (non-absent) input. When the input is absent, the output is absent. The first time the input is present, the output is given by the *initialValue* parameter of the actor (which by default is absent). It is worth noting that, contrary to NonStrictDelay, Pre is strict, meaning that the input must be known before the output can be determined. Thus, it will not break a causality loop. To break a causality loop, use NonStrictDelay.
- **When** filters a signal based on another. If the control input (on the bottom) is present and true, then the data input (on the left) is copied to the output. If control is absent, false, or true with the data input being absent, then the output is absent.

The Ptolemy II library also offers several actors to manipulate absent values:



- **Absent**. Output is always absent.
- **IsPresent** outputs true if its input is present and false otherwise.
- **TrueGate** outputs true if its input is present and true; otherwise, absent.

Example 5.2: A simple model of a digital circuit is shown in Figure 5.4. It is a model of a 2-bit, modulo-4 counter that produces the integer sequence $0, 1, 2, 3, 0, 1, \ldots$. The feedback loops use NonStrictDelay actors, each of which models a latch (a latch is a circuit element that captures a value and holds it for some period of time). It also includes two actors that model logic gates, the LogicalNot and LogicGate (see box on page 112).

The upper loop, containing the LogicalNot, models the low-order bit (**LOB**) of the counter. It starts with value false, the initial output of the NonStrictDelay, and the alternates between true and false in each subsequent tick.

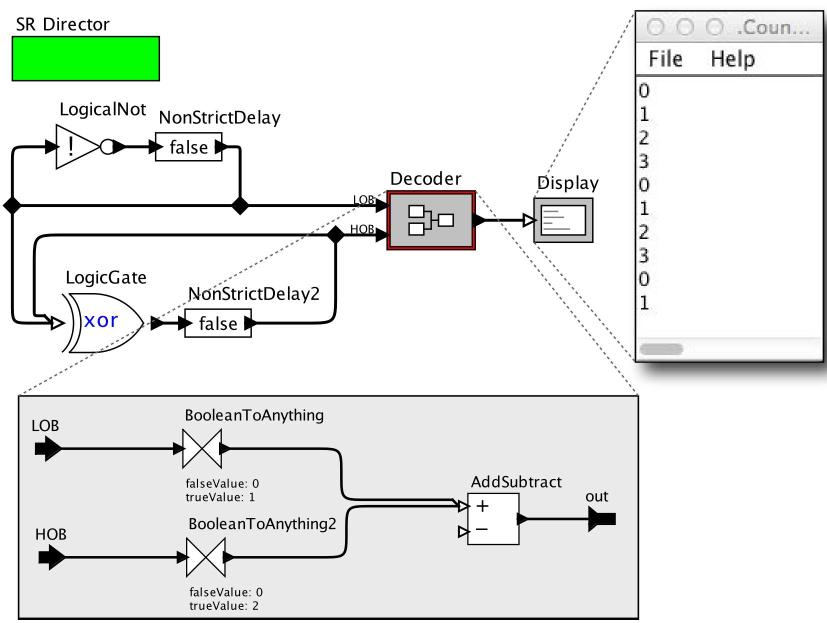


Figure 5.4: A model of a 2-bit counter in SR. The top-level model includes a Decoder composite actor that translates the boolean data into integers. [online]

The lower loop, containing the LogicGate, models a carry circuit, implementing the high-order bit (**HOB**) of the counter. It also starts with false, and toggles between true and false in each tick where the the LOB is true.

The Decoder is a composite actor provided just to generate a more readable display. It converts the two Boolean values into a numerical value from 0 to 3 by assigning values to the LOB It contains two **BooleanToAnything** actors that convert the Boolean values to the values of the LOB and HOB, which are then added together.

The NonStrictDelay actors in Figure 5.4 are non-strict. They are able to produce outputs without knowing the inputs. On the first tick, the values of the outputs are given by the *initialValue* parameters of the actors. In subsequent ticks, the values of the outputs are given by the input *from the previous tick*, which has been found by identifying the fixed point. Thus, these actors break the potential causality loops.

Notice that it would not work to use Pre instead of NonStrictDelay (see box on page 5.2.1). The Pre actor is strict, because it has to know whether the input is present or not in order to determine whether the output is present or not.*

The model of Figure 5.4 is rather simple and does not illustrate the full power of SR. In fact, the same model would work with an SDF director, provided that NonStrictDelay actors are replaced by SampleDelay actors.†

Every directed cycle in SR is required to contain at least one non-strict actor. But NonStrictDelay is not the only non-strict actor. Another example of a non-strict actor is the **NonStrictLogicGate** actor, which can be parameterized to implement functions such as non-strict logical AND, also called a **parallel AND**. The truth table of the non-strict AND with two inputs is shown below (the actor can in fact accept an arbitrary number of in-

*The Lustre synchronous language (Halbwachs et al., 1991) is able to make Pre non-strict by using a **clock calculus**, which analyzes the model to determine in which ticks the inputs will be present. Thus, in Lustre, Pre *does not execute* in ticks where its input is absent. As a consequence, when it does execute, it knows that the input is present, and even though it does not know the value of the input, it is able to produce an output. The SR director in Ptolemy II does not implement a clock calculus, adopting instead the simpler clocking scheme of Esterel (Berry and Gonthier, 1992).

†SampleDelay produces initial outputs during the initialize phase of execution. In dataflow domains, those initial outputs are buffered and made available during the execution phase. In SR, however, there is no buffering of data, and any outputs produced during initialize are lost. Hence, SampleDelay is not useful in SR.

puts):

inputs	$\bot$	$true$	$false$
$\bot$	$\bot$	$\bot$	$false$
$true$	$\bot$	$true$	$false$
$false$	$false$	$false$	$false$

Here, the symbol $\bot$ means **unknown**. Observe that when one input is known to be false, the output is false, even if the other input is unknown.

Example 5.3: The model shown in Figure 5.5 results in non-ambiguous semantics despite its feedback loop. The NonStrictLogicGate implements the AND logic function, and outputs a Boolean "false" value at every tick because one of the inputs is always false.

A practical example that also has cycles without NonStrictDelay is next.

Example 5.4: Figure 5.6 shows an SR realization of **token-ring media access control** (**MAC**) protocol given by Edwards and Lee (2003b). The top-level model has three instances of an **Arbiter** class connected in a cycle. It also has a ComposeDisplay composite actor used to construct a human-readable display of the results of execution, shown at the bottom.

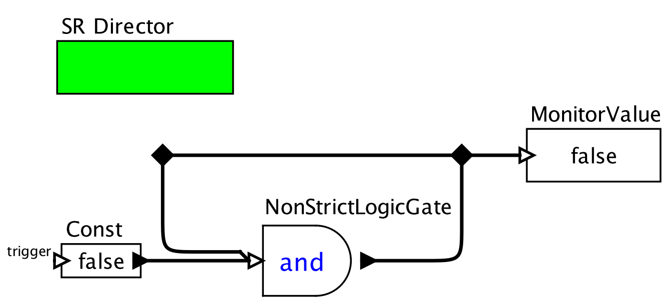


Figure 5.5: A non-ambiguous model which uses a non-strict logical AND. [online]

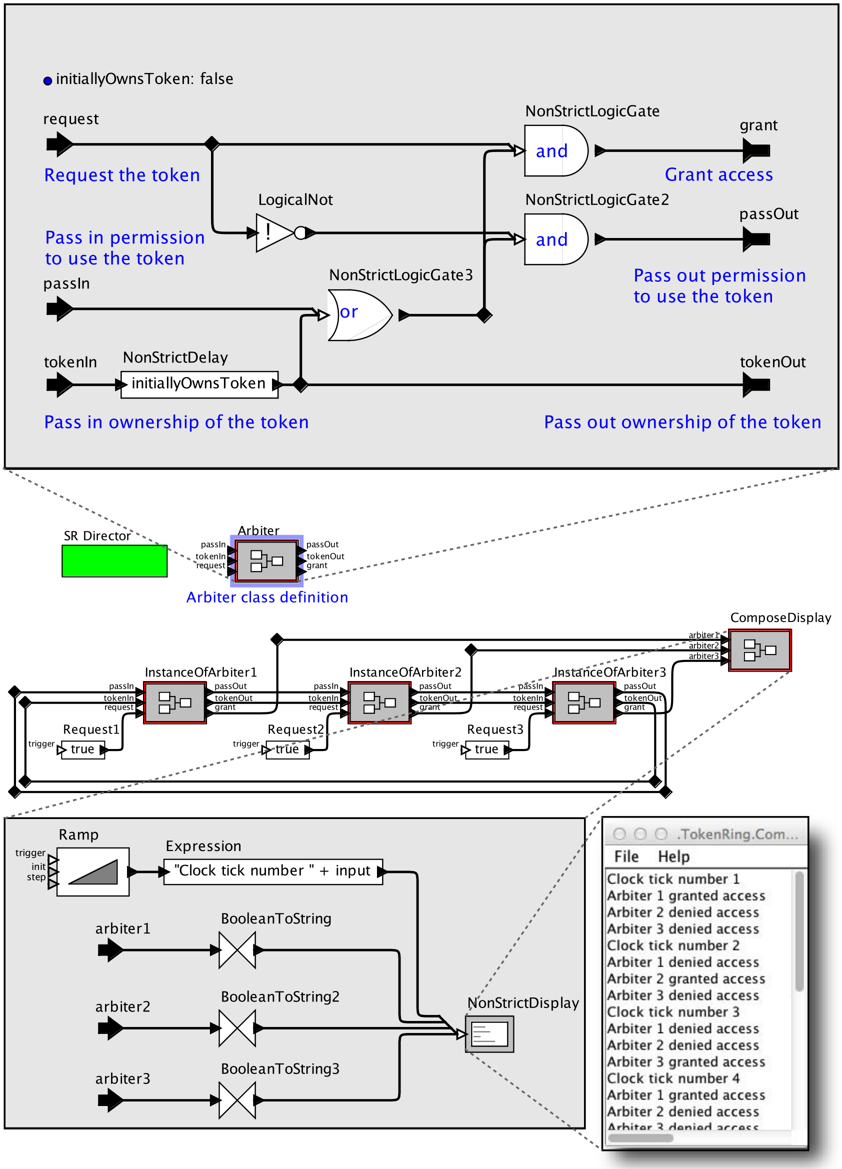


Figure 5.6: A token-ring media access protocol implemented using SR. From Edwards and Lee (2003b). [online]

The goal of this system is to arbitrate fairly among requests for exclusive access to a shared resource by marching a token around a ring. At each tick of the logical clock, the arbiter grants access to requestor holding the token, if it requests access. If it does not request access, then the model grants access to the first requestor downstream of the block with the token that requests access. In the figure, all three requestors are always requesting access, and in the display at the bottom, you can see that access is granted fairly in a round-robin fashion. In this model, InstanceOfArbiter1 starts with the token (see the parameter of the instance).

The three arbiters are instances of the actor-oriented class shown at the top of the figure. This class has three inputs and three outputs. It has an instance of NonStrict-Delay that outputs true for the arbiter that currently holds the token. Exactly one of the three is initialized with value true. At each tick of the clock, the arbiter passes the token down to the next arbiter. This forms a cycle that include three instances of NonStrictDelay.

However, there are another cycles that have no instances of NonStrictDelay, for example the cycle passing through each *request* input and *grant* output. This cycle has three instances of NonStrictLogicGate, configured to implement the parallel AND. This logic gate will grant access to the requestor if it has a request and it either holds the token or its *passIn* input is true (meaning that the upstream arbiter has the token but does not have a request). Although it is far from trivial to see at glance, every cycle of logical gates can be resolved without full knowledge of the inputs, so the model does not suffer from a causality loop.

Another example of a non-strict actor is the Multiplexor or BooleanMultiplexor (see box on page 119). These require their control input (at the bottom of the icon) to be known; the value of this input then determines which of the data inputs are to be forwarded to the output. Only that one data input needs to be known for the actor to able to produce an output.

Example 5.5: An interesting example, shown in Figure 5.7, calculates either $\sin(\exp(x))$ or $\exp(\sin(x))$, depending on a coin toss from the Bernoulli actor. Malik (1994) called examples like this **cyclic combinational circuit**s, because, although there is feedback, there is actually no state stored in the system. The output (each value plotted) depends only on the current inputs (the data from the Ramp

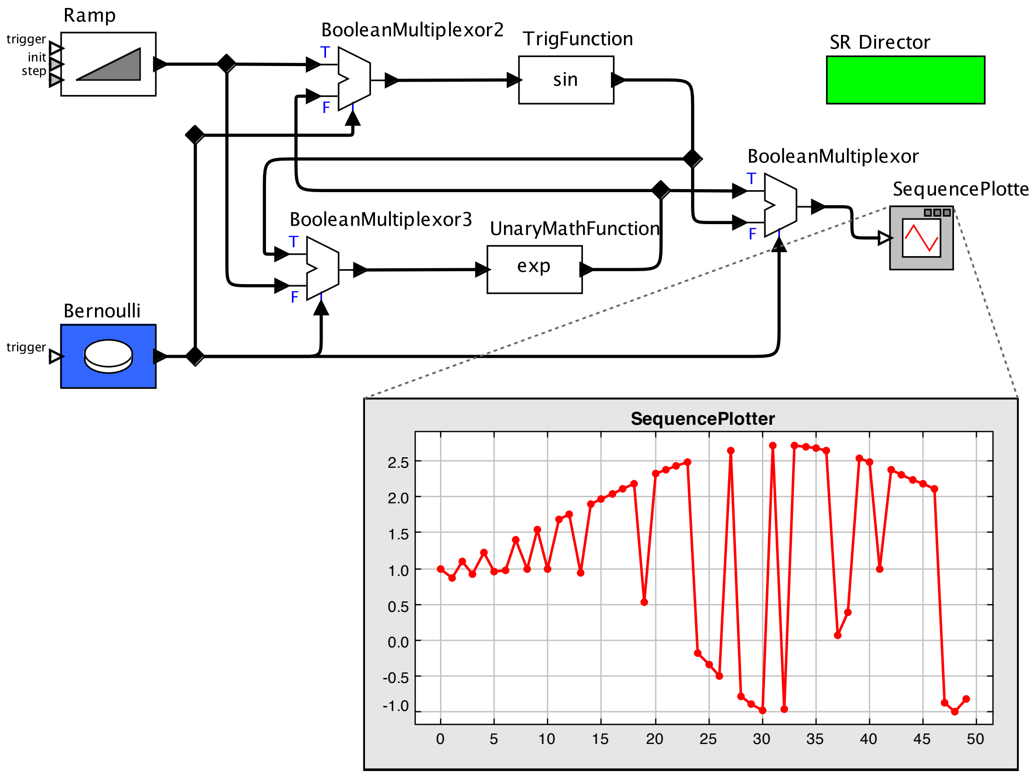


Figure 5.7: A model of a cyclic combinational circuit in SR. From Malik (1994). [online]

and Bernoulli actors). A circuit whose output depends only on the current inputs and not on the past history of inputs is called a **combinational** circuit. Most circuits with feedback are not combinational. The output depends not only on the current inputs, but also on the current state, and the current state changes over time.

In this case, feedback is being used to avoid having to have two copies of the actors that do the actual computation, the TrigFunction and UnaryMathFunction (see box on page 58). An equivalent model that uses two copies of these actors is shown in Figure 5.8. If these models are literally implemented in circuits, with a separate cir-

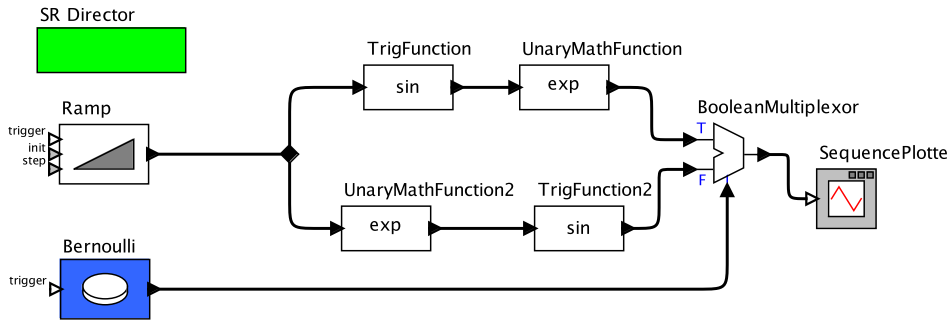


Figure 5.8: Acyclic version of the model of Figure 5.7 that uses two copies of each of the math actors. [online]

cuit for each actor, then the model in Figure 5.7 may be considerably less expensive than the one in Figure 5.8.

The model has three BooleanMultiplexor actors. These actors send either their "T" or "F" input value to the output port depending depending on whether the control input (at the bottom of the actor) is set to true or false. At each tick, one of the two BooleanMultiplexor actors on the left will be able to provide an output (once it is provided with an input from the Ramp). That one BooleanMultiplexor, therefore, breaks the causality loop and enables finding a fixed point.

5.2.3 Causality Loops

Not all SR models are executable. In particular, it is possible to construct feedback models that exhibit a causality loop, as illustrated by the following examples.

Example 5.6: Two examples of loops with unresolvable cyclic dependencies are shown in Figure 5.9. Both the Scale and the LogicalNot actors are strict, and hence their inputs must be know for the outputs to be determined. But the outputs are equal to the inputs in these models, so the inputs cannot be known. The SR director will reject these models, reporting an exception

```
IllegalActionException:  Unknown inputs remain.  Possible
causality loop:
  in Display.input
```

5.2.4 Multiclock Models

The logical clock in the SR domain is a single, global clock. Every actor under the control of an SR director will be fired on every tick of this clock. But what we want some actors to be fired more or less frequently? Fortunately, the hierarchy mechanism in Ptolemy II makes it relatively easy to construct models with multiple clocks proceeding at different rates. The **EnabledComposite** actor is particularly useful for building such **multiclock** models.

Example 5.7: Consider the **guarded count** model of Figure 5.10, which counts down to zero from some initial value and then restarts the count from some new value. At the top level, the model has two composite actors and two Display actors. The CountDown composite actor uses SR primitive actors to implement the following count-down behavior: whenever it receives a non-absent value n (an integer) at its *start* input port, it (re)starts a count-down from n; that is, it outputs the sequence of values $n, n-1, ..., 0$ at its *count* output port. When the count reaches 0, the *ready* port outputs a value true, signaling that the actor is ready for a new count down.

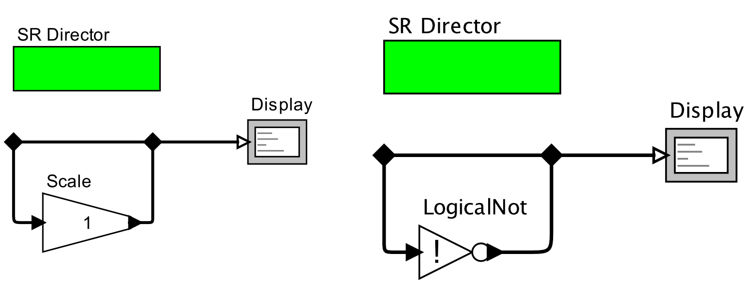


Figure 5.9: Two SR models with invalid loops.

The ready signal controls the firing of the EnabledComposite actor. Within this composite, a reaction only occurs when a true value is provided on the *enable* input port (the port at the bottom of the actor). Note that the ready signal is initially true, due to the NonStrictDelay actor used inside CountDown.

The clock of the SR director inside EnabledComposite progresses at a slower rate than the clock of the top-level SR director. In fact, the relationship between these rates is determined dynamically by the data provided by the Sequence actor.

5.3 Finding the Fixed-Point

For acyclic models (such as the one shown Figure 5.8) or cyclic models where every cycle is "broken" by a NonStrictDelay actor (such as the model shown in Figure 5.4), executing the model efficiently is easy. The actors of the model can be ordered according to their dependencies (e.g., using a topological sorting algorithm) and then fired according to that order. In this case, each actor only needs to be fired once at each tick of the logical clock.

However, this strategy will not work with models like those in Figures 5.6 or 5.7, because the order in which the actors have to be fired depends on data computed by some of the actors. Fortunately, there is a simple execution strategy that works. The key is to start each tick of the logical clock by assigning a special value called unknown, denoted $\perp$, to all signals. The director can then simply evaluate actors in arbitrary order until no more progress is made. For strict actors, if there are any unknown inputs, then the outputs will remain unknown. For non-strict actors, even when some inputs are unknown, some outputs may become known. This procedure is said to have converged when no firing of any actor changes the state of any signal. If the actors all follow the strict actor semantics (see box on page 433), then it can be proven that this procedure converges in a finite number of steps (see, for example, Edwards and Lee (2003b)).

Upon convergence, either all signals will be known, or some signals will remain unknown. If every iteration results in all signals being known for all possible inputs, then the model is said to be **constructive** (that is, a solution can be "constructed" in a finite number of steps). Otherwise, the model is declared to be **non-constructive**, and it is rejected.

Note that in the Ptolemy II SR domain, the causality analysis is performed dynamically, at run-time. This is in contrast to languages such as Esterel, where the compiler attempts

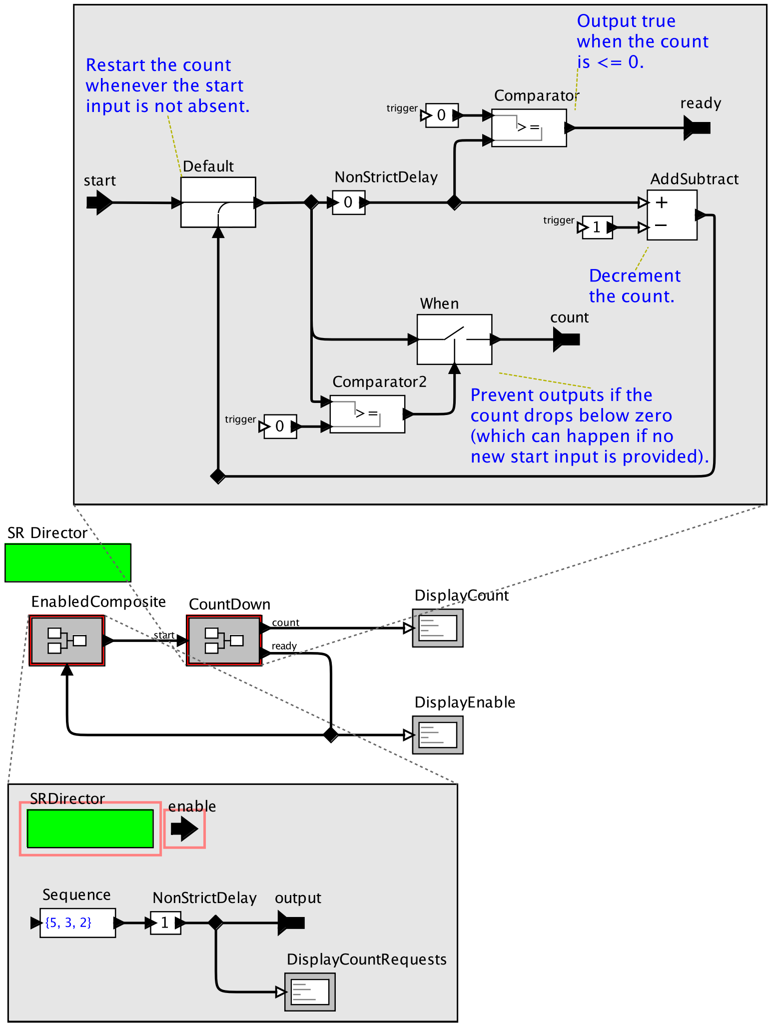


Figure 5.10: Multiclock model in SR. [online]

to prove statically (i.e., at compile-time) that the program is constructive (see the sidebar on page 179).

SR can only work correctly with actors that follow the strict actor semantics. To understand this, we can model an actor as a state machine. Let $\vec{x}$, $\vec{y}$ and $\vec{s}$ denote the vectors of inputs, outputs, and states, respectively. Then the behavior of the state machine can be described as

$$\vec{y}(n) = f(\vec{x}(n), \vec{s}(n)) \tag{5.1}$$

$$\vec{s}(n+1) = g(\vec{x}(n), \vec{s}(n)), \tag{5.2}$$

where n indexes the ticks of the logical clock, f models the fire method that computes outputs from current inputs and current state, and g models the postfire method that computes the next state from current inputs and current state. The key here is that the fire method does not change the state of the actor. Hence, the fire method can be invoked repeatedly, and each time, given the same inputs, it will produce the same outputs.

An additional condition on actors is that they be monotonic (see box on page 182). Although the mathematical underpinnings of this constraint are quite sophisticated, the practical manifestation of the constraint is simple. An actor is monotonic if it does not change its mind about outputs given more information about inputs. Specifically, if the fire method is invoked with some inputs unknown, then if the actor is non-strict, it may be able to produce outputs. Suppose that it does. Then the actor is monotonic if given more information about the inputs (fewer inputs are unknown) does not cause it to produce a *different* output than the one it produced with less information.

Most Ptolemy II actors conform to the strict actor semantics and are monotonic and therefore can be used in SR.

5.4 The Logic of Fixed Points

Recall the two models of Figure 5.9, both of which exhibit causality loops. These models, however, are different from one another in an interesting way. They exhibit the difference between a deterministic and a constructive semantics of synchronous models. The constructive semantics is based on ideas from **intuitionistic logic**, and although it is also deterministic, it rejects some models that would be accepted by a broader deterministic semantics based on classic logic.

Sidebar: Causality in Synchronous Languages

The problem of how to resolve cyclic dependencies, the **causality problem**, is one of the major challenges in synchronous languages. We briefly summarize several solutions here, and refer the reader to research literature and survey articles such as Caspi et al. (2007) for more details.

The most straightforward solution to the causality problem is to forbid cyclic dependencies altogether. This is the solution adopted by the Lustre language, which requires that every dataflow loop must contain at least one **pre** operator. The same effect could be achieved by the Ptolemy II SR director by requiring that every loop contain at least one NonStrictDelay actor. This actor breaks the instantaneous cyclic dependency. The Lustre compiler statically checks this condition and rejects those programs that violate it. The same policy is followed in SCADE.

Another approach is to accept a broader set of constructive programs, as is the case with the Ptolemy II SR domain. This approach was pioneered by Berry (1999) for Esterel. A key difference between Esterel and SR is that in SR a fixed-point is computed at run-time (at each tick of the logical clock), while the Esterel compiler attempts to prove that a program is constructive at compile-time. The latter is generally more difficult since the inputs to the program are generally unknown at compile-time. On the other hand, statically proving that a program is constructive has two key benefits. First, it is essential for safety-critical systems, where run-time exceptions are to be avoided. Second, it allows generation of implementations that minimize the run-time overhead of fixed-point iteration.

Yet another approach is to accept only deterministic programs, or conversely, reject programs that, when interpreted as a set of constraints, do not yield unique solutions. This approach is followed in Signal (Benveniste and Le Guernic, 1990) and Argos (Maraninchi and Rémond, 2001). One drawback with this approach is that it sometimes accepts dubious programs. For instance, consider a program representing the system of equations

$$Y = X \wedge \neg Y.$$

Although this system admits a unique solution in classic two-valued logic, namely, $X = Y =$ false, it is unclear whether the corresponding implementation is meaningful. In fact, a straightforward combinational circuit implementation is unstable; it oscillates.

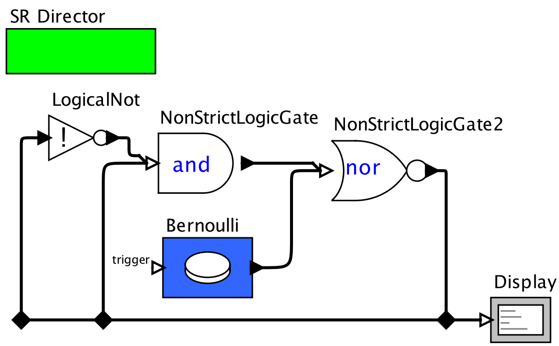


Figure 5.11: Non-constructive example with a unique fixed point. [online]

In particular, we could have interpreted the left model of Figure 5.9 as defining an equation between the input and output of the Scale actor, say x, as follows:

$$x = 1 \cdot x$$

In the classic logic interpretation, the above equation has multiple solutions, e.g., $x = 0$, $x = 1$, and so on. A non-deterministic semantics based on classic logic would accept any of these solutions as a valid behavior of the system. A deterministic semantics would declare the model ambiguous, and thus invalid. In the SR semantics, the above equation has a unique least fixed-point solution, namely, $x = \bot$, unknown. Hence, SR also rejects this model.

The right model of Figure 5.9 can be seen as defining the equation

$$x = \neg x$$

where $\neg$ denotes logical negation. In this case, in the classic logical interpretation, there is no solution at all, quite a different situation. A deterministic semantics may again reject this model. In the case of SR, the solution is again $x = \bot$, unknown, resulting in rejection of the model.

A third situation, due to Malik (1994) and shown in Figure 5.11, however, could be accepted by a deterministic semantics, but is rejected by a constructive semantics. Logically, the output of the AND gate should always be false, and hence the output sent to the Display actor should be equal to the negation of the input value produced by the Bernoulli actor. Hence, there is a single unique behavior for all possible inputs. The model, however, is rejected by the Ptolemy II SR director as non-constructive whenever the Bernoulli actor produces a false. In that case, all signals in the loop remain unknown. In the constructive SR semantics, this solution with unknowns is the *least* fixed point, and hence is the behavior selected, even though there is a unique fixed point with no unknowns.

Even though the circuit in Figure 5.11 seems to have a logically consistent behavior for every input, there are good reasons for rejecting it. If this were actually implemented as a circuit, then time delays in the logic gates would cause the circuit to oscillate. It would not, in fact, realize the logic specified by the model. To realize such circuits in software, the only known technique for finding the unique fixed point and verifying that it is unique, in general, is to exhaustively search over all possible signal assignments. In a small model like this, such an exhaustive search is possible, but it becomes intractable for larger models, and it becomes impossible if the data types have an infinite number of possible values. Thus, the fact that the model is non-constructive reveals very real practical problems with the model.

We now give a brief introduction to the theoretical foundation of the SR semantics. This is a rather deep topic, and our coverage here is meant only to whet the appetite of the reader to learn more. The SR semantics is based on the theory of continuous functions over complete partially ordered sets (CPOs) (see box on page 182). In the case of SR, the key CPO is a so-called flat CPO shown in Figure 5.12. This CPO consists of the minimal element $\perp$ and all "legal" values of Ptolemy models, such as booleans, integer and real numbers, but also tuples, records, lists, and so on (see Chapter 14). Any legal value is considered to be greater than $\perp$ in the CPO order, but the legal values are incomparable among themselves, leading to the term "flat".

Now, consider an SR model. The output of every actor in the model can be seen as a variable taking values in the above flat CPO. The vector of all output variables can be seen as taking values in the product CPO obtained by forming the cartesian product of all individual CPOs, with element-wise ordering. For simplicity, let us suppose that

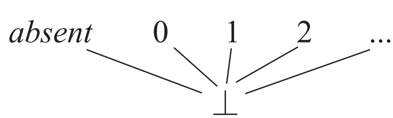


Figure 5.12: The flat CPO ensuring existence of a unique least fixed-point in SR.

Sidebar: CPOs, Continuous Functions and Fixed Points

The SR semantics is based on order theory, which we summarize here; see Davey and Priestly (2002) for a more thorough explanation

Consider a set S. A **binary relation** on S is a subset $\sim \subseteq S \times S$. We often write $x \sim y$ instead of $(x, y) \in \sim$. A **partial order** on S is a binary relation $\sqsubseteq$ which is **reflexive** (i.e., $\forall x \in S : x \sqsubseteq x$), **antisymmetric** (i.e., $\forall x, y \in S : x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$), and **transitive** (i.e., $\forall x, y, z \in S : x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$). A **partially ordered set** or **poset** is a set equipped with a partial order.

Let $X \subseteq S$. An **upper bound** of X is an element $u \in S$ such that $\forall x \in X : x \sqsubseteq u$. A **least upper bound** of X, denoted $\sqcup X$, is an element $\ell \in S$ such that $\ell \sqsubseteq u$ for all upper bounds u of X. A **chain** of S is a subset $C \subseteq S$ which is **totally ordered**: $\forall x, y \in C : x \sqsubseteq y$ or $y \sqsubseteq x$. A **complete partial order** or CPO is a poset S such that every chain of S has a least upper bound in S. This condition also guarantees that every CPO S has a **bottom element** $\bot$, such that $\forall x \in S : \bot \sqsubseteq x$. (Indeed, the empty chain must have a least upper bound in S, and the set of upper bounds of the empty subset of S is the entire S.)

To illustrate the above concepts, consider the set of natural numbers $\mathbb{N} = \{0, 1, 2, ...\}$. $\mathbb{N}$ is a poset with the usual (total, and therefore also partial) order $\leq$. Because $\leq$ is a total order, $\mathbb{N}$ is a chain. The least upper bound of $\mathbb{N}$ can be defined to be a new number ω such that $n < \omega$ for all $n \in \mathbb{N}$. ω is not a natural number, therefore, $\mathbb{N}$ is not a CPO. On the other hand, the set $\mathbb{N}^\omega = \mathbb{N} \cup \{\omega\}$ is a CPO. The bottom element of $\mathbb{N}^\omega$ is 0.

Every poset whose chains are all finite is a CPO. This is because the greatest element in a chain is also the least upper bound of the chain. This is why the "flat" poset of Figure 5.12 is a CPO.

Consider two CPOs X and Y. A function $f : X \to Y$ is **Scott-continuous** or simply **continuous** if for all chains $C \subseteq X$, $f(\sqcup C) = \sqcup\{f(c) \mid c \in C\}$. It can be shown that every continuous function is also **monotonic**, i.e. it satisfies: $\forall x, y \in X : x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. However, not all monotonic functions are continuous. For example, consider the function $f : \mathbb{N}^\omega \to \mathbb{N}^\omega$ such that $f(n) = 0$ for all $n \in \mathbb{N}$ and $f(\omega) = \omega$. Then $f(\sqcup\mathbb{N}) = f(\omega) = \omega$, whereas $\sqcup\{f(n) \mid n \in \mathbb{N}\} = \sqcup\{0\} = 0$. The following **fixed-point theorems** are well-known results of order theory: (A) Every monotonic function $f : X \to X$ on a CPO X has a least fixed-point x^*. (B) If f is also continuous then $x^* = \bigsqcup_{i \geq 0} f^i(\bot)$, where $f^0(\bot) = \bot$ and $f^{i+1}(\bot) = f(f^i(\bot))$. (B) is used to obtain an effective procedure for computing the semantics of an SR model.

the model is *closed*, in the sense that every input port of every actor in the model is connected to some output port (the theory also works for open models, but is slightly more complicated; we refer the reader to Edwards and Lee (2003b) for a more detailed explanation). The SR model then defines a function F which has both as domain and co-domain this product CPO: this is because the model is closed, so every input is also an output. Thus, F takes as input a vector $\vec{x}$ and returns as output another vector $\vec{y}$. The latter is obtained by firing all actors in the model once. Given this interpretation, a closed SR model defines the equation

$$\vec{x} = F(\vec{x})$$

This equation has a unique least solution $\vec{x}^*$, provided that F is monotonic; that is, provided that $\vec{x} \leq \vec{y}$ implies $F(\vec{x}) \leq F(\vec{y})$. (The precise condition is for the function to be continuous, but in the case of flat CPOs, monotonicity is equivalent to continuity.) The solution $\vec{x}^*$ is called a fixed-point because it satisfies $\vec{x}^* = F(\vec{x}^*)$. It is 'least' in the sense that it is smaller in the CPO ordering than every other solution of the above equation. That is, for any $\vec{y}$ such that $\vec{y} = F(\vec{y})$, it must be $\vec{x}^* \leq \vec{y}$.

Moreover, the least fixed-point can be computed effectively in a finite number of iterations, in fact, at most N iterations, where N is the total number of outputs in the model. Indeed, starting with all outputs set to $\perp$, every iteration that fires all actors without reaching the fixed-point is guaranteed to update at least one output. The first time an output is updated, it changes from $\perp$ to some legal value v. Because F is monotonic, the same output can no longer change from v to $\perp$ or any other v', since $v > \perp$ and v is incomparable with any $v' \neq v$. Therefore, each output can be updated at most once. As a result, the fixed-point must be reached after at most n iterations.

The monotonicity of F is ensured by ensuring that every individual actor is monotonic; that is, that its fire method is monotonic. Monotonicity of F then follows from the fact that composition of monotonic functions results in a monotonic function. Monotonicity of atomic actors is ensured in Ptolemy by construction. The key is to ensure that if an actor outputs a known value, say v, in the presence of unknown inputs, then if those inputs become known, the actor will not "change its mind" and output a different value v'. A straightforward way to ensure this property is by making an actor strict, in the sense that it requires all inputs to be known, otherwise, it produces unknown outputs. Most actors in

Ptolemy are strict, but a few key ones that we have discussed are non-strict. Every cycle in an SR model requires some non-strict actors.

5.5 Summary

This chapter has introduced the SR domain in Ptolemy II. In SR, execution is governed by a logical clock, and at each tick of the clock, actors execute, conceptually, simultaneously and instantaneously. We have explained how this results in a fixed-point semantics, and have given examples of both cyclic and acyclic models. We have shown that SR admits multiple clock domains, where clocks progress at different rates. Finally, we have given a brief introduction to the (rather deep) mathematical foundations behind the semantics of SR models.

Exercises

1. This exercise studies the use of absent events in SR.

   (a) As a warmup, use Sequence and When to construct an SR model that generates a sequence of values *true* interspersed with *absent*. For example, produce the sequence

   $$(\mathit{true}, \mathit{absent}, \mathit{absent}, \mathit{true}, \mathit{absent}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{absent})\ .$$

   Make sure your model adequately displays the output. In particular, *absent* should be visible in the display.[‡]

   (b) Use Default and When to create a composite actor **IsAbsent** that given any input sequence, produces an output *true* at every tick when the input is *absent*, and otherwise produces the output *absent*.[§]

   (c) Create a composite actor that can recognize the difference between single and double mouse clicks. Your actor should have an input port named *click*, and two output ports, *singleClick* and *doubleClick*. When a `true` input at *click* is followed by N *absent*s, your actor should produce output `true` on *singleClick*, where N is a parameter of your actor. If instead a second `true` input occurs within N ticks of the first, then your actor should output a `true` on *doubleClick*.

   How does your model behave if given three values *true* within N ticks on input port *click*?

   (d) **Extra credit**: Redo (a)-(c) by writing a custom a Java actor for each of the three functions above. How does this design compare with the design implemented using primitive SR actors? Is it more or less understandable? Complex?

2. The token-ring model of Figure 5.6 is constructive under the assumption that exactly one of the instances of the Arbiter initially owns the token (it has its *initiallyOwnsToken* parameter set to true). If no instance of Arbiter initially owns the token, then is the model still constructive? If so, explain why. If not, given a set of values of the Request actors that exhibits a causality loop.

[‡]Note that you could use TrueGate to implement this more simply, but part of the goal of this exercise is to fully understand When.

[§]Again, a simpler implementation is available using IsPresent, but the goal of this exercise is to fully under Default.

6

Finite State Machines

Thomas Huining Feng, Edward A. Lee, Xiaojun Liu, Christian Motika, Reinhard von Hanxleden, and Haiyang Zheng

Contents

Finite state machines are used to model system behavior in many types of engineering and scientific applications. The **state** of a system is defined as its condition at a particular point in time; a **state machine** is a system whose outputs depend not only on the current inputs, but also on the current state of the system. The state of a system is a summary of everything the system needs to know about previous inputs in order to produce outputs. It is represented by a state variable $s \in \Sigma$, where Σ is the set of all possible states for the system. A **finite state machine** (**FSM**) is a state machine where Σ is a finite set. In a finite state machine, a system's behavior is modeled as a set of states and the rules that govern transitions between them.

A number of Ptolemy II actors include state and behave as simple state machines. For example, the Ramp actor (which produces a counting sequence) has state, which is the current position in the sequence. This actor uses a local variable, called a **state variable**, to keep track of its current value. The Ramp actor's reaction to a *trigger* input depends on how many times it has previously fired, which is captured by the state variable. The number of possible states for a Ramp actor depends on the data type of the counting sequence. If it is `int`, then there are 2^{32} possible states. If it is `double`, then there are 2^{64}. If the data type is `String`, then the number of possible states is infinite (and thus the Ramp cannot be described as a finite state machine).

Although the number of Ramp actor states is potentially very large, the logic for changing from one state to the next is simple, which makes it easy to characterize the behavior of the actor. In contrast, it is common to have actors that have a small number of possible states, but use relatively complex logic for moving from one state to the next. This chapter focuses on such actors.

This chapter discusses approaches for designing, visualizing, and analyzing finite state machines in Ptolemy II. In Chapter 8, we extend these approaches to construct modal models, in which the states themselves are Ptolemy II models.

6.1 Creating FSMs in Ptolemy

A Ptolemy II finite state machine is created in a similar manner to the previously described actor-oriented models, but it is built using states and transitions rather than actors and connections/relations. A **transition** represents the act of moving from one state to another; it can be triggered by a guard, which specifies the conditions under which the transition is taken. It is also possible to specify output actions (actions that produce outputs when the transition is taken) and set actions (actions that set parameters when the transition is taken).

The main actor used to implement FSM models in Ptolemy II is **ModalModel**, found in the `Utilities` library.* A ModalModel contains an FSM, which is a collection of states and transitions depicted using visual notation shown in Figure 6.1. In this figure, the ModalModel has two input and two output ports, though in general it could have any number of input and output ports. It has three states. One of these states is an initial state (labeled *initialState* in the figure), which is the state of the actor when the model begins execution. The initial state is indicated visually by a bold outline. Some of the states may also be final states, indicated visually with a double outline (more about final states later). The process for creating an FSM model in Vergil is shown in Figure 6.2.

To begin creating an FSM, drag the ModalModel into your model from the library. Populate the actor with input and output ports by right clicking (or control-clicking on a Mac) and selecting [`Customize→Ports`], clicking `Add`, and specifying port names and whether they are inputs or outputs. Then right click on the ModalModel and select `Open Actor`. The resulting window is shown in Figure 6.3. It is similar to other Vergil win-

*You can also use **FSMActor**, found in `MoreLibraries→Automata`, which is simpler in that it does not support mode refinements, used in Section 6.3 and Chapter 8.

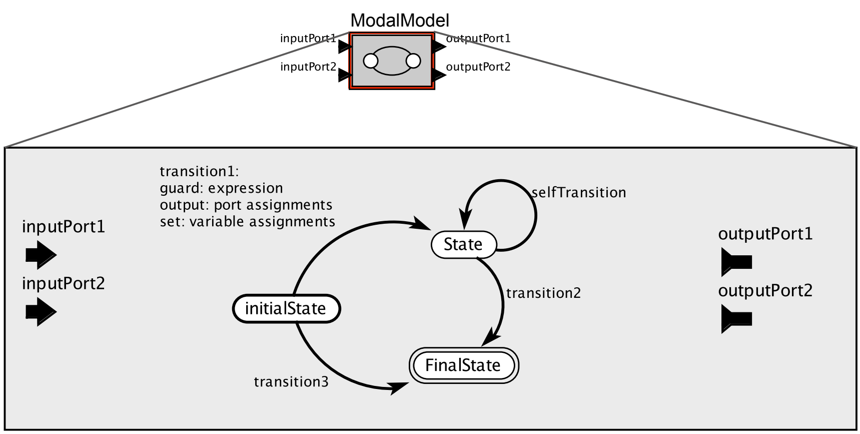


Figure 6.1: Visual notation for state machines in Ptolemy II.

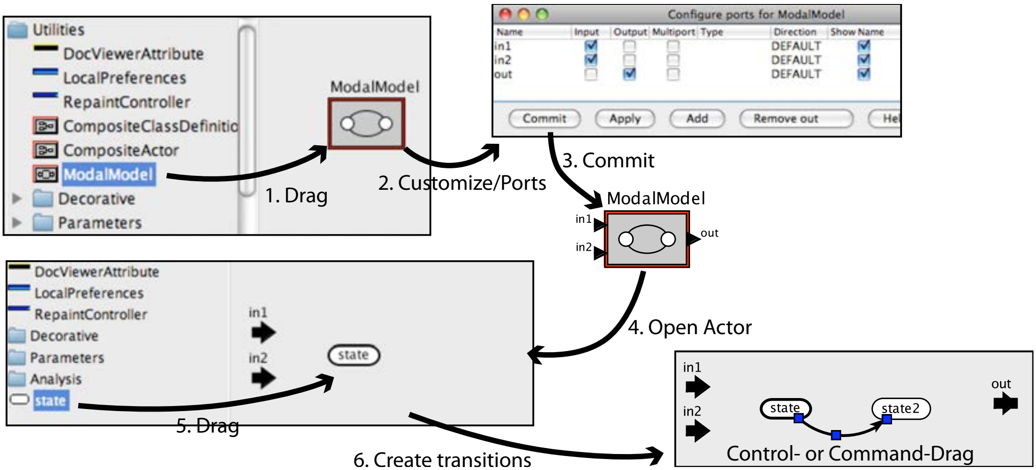


Figure 6.2: Creating FSMs in Vergil, using the ModalModel actor (a similar procedure applies to using the FSMActor).

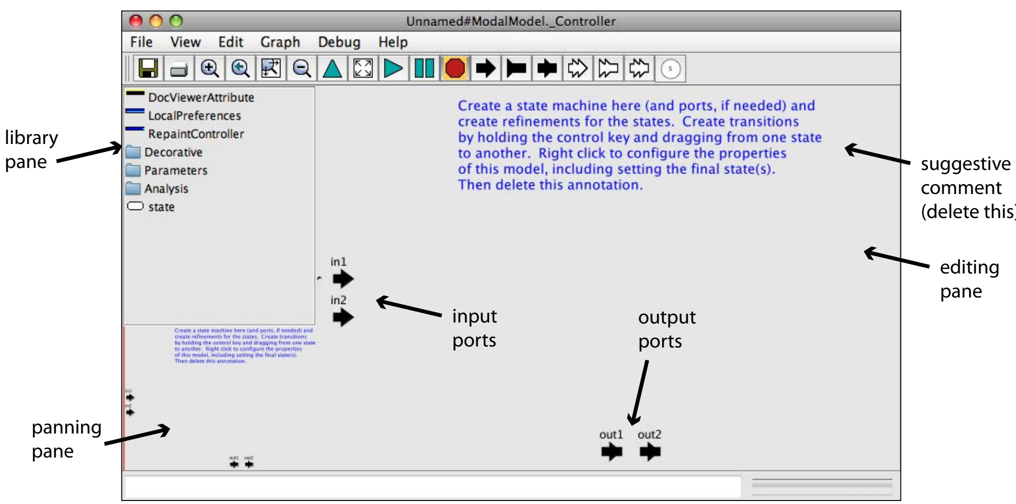


Figure 6.3: Editor for FSMs in Vergil, showing two input and two output ports, before being populated with an FSM.

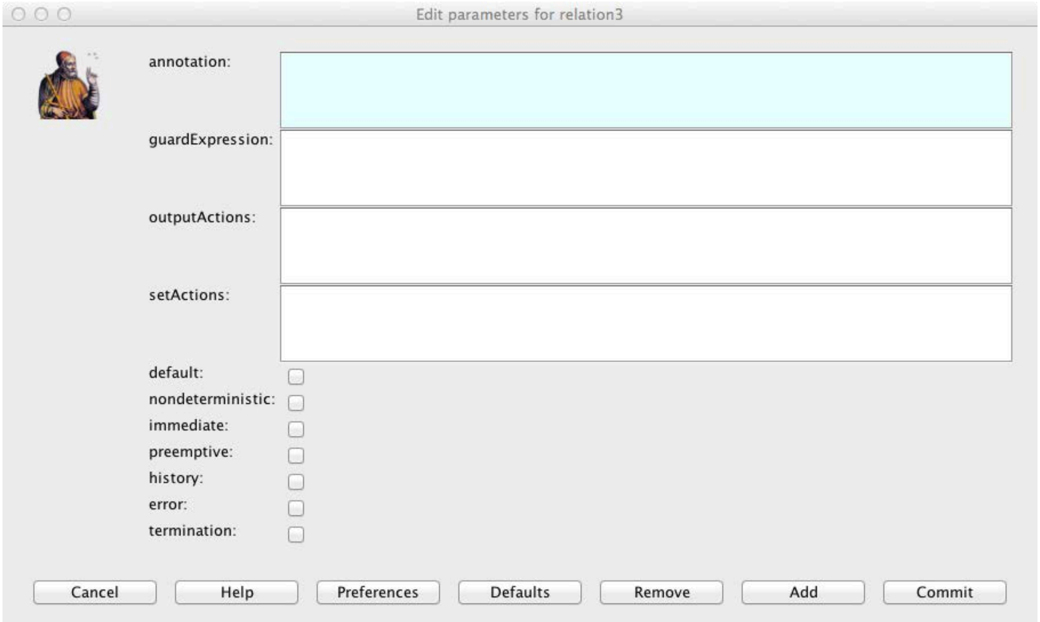


Figure 6.4: Dialog box for configuring a transition in an FSM.

dows, but has a customized library consisting primarily of a *State*, a library of parameters, and a library of decorative elements for annotating your design.

Drag in one or more states. To create transitions between states, **hold the control key** (or the Command key on a Mac) and click and drag from one state to the other. The "grab handles" on the transitions can be used to control the curvature and positioning of the transitions.

Double click (or right click and select `Configure`) on the transition to set the guard, output actions, and set actions by entering text into the dialog box shown in Figure 6.4. For readability, you can also specify an annotation associated with the transition.

The ModalModel implementing the finite state machine can be placed within a larger model, to be executed using another director. The choice of director will depend on the application. All directors are compatible with it.

We illustrate the use of this process with a simple FSM application example, described below.

Example 6.1: Consider a thermostat that controls a heater. The thermostat is modeled as a state machine with states $\Sigma = \{heating, cooling\}$. If the state $s = heating$, then the heater is on. If $s = cooling$, then the heater is off. Suppose the target temperature is 20 degrees Celsius. It would be undesirable for the heater to cycle on and off whenever the temperature is slightly above or below the target temperature; thus, the state machine should include hysteresis around the setpoint. If the heater is on, then the thermostat allows the temperature to rise slightly above the target, to an upper limit specified as 22 degrees. If the heater is off, then it allows the temperature to drop below the target to 18 degrees. Note that the behavior of the system at temperatures between 18 and 22 degrees depends not only on the input temperature but also on the state. This strategy avoids **chattering**, where the heater would turn on and off rapidly when the temperature is close to the target temperature.

This FSM is constructed as shown in 6.5. The FSM has a *temperature* input and a *heat* output; its output specifies the rate at which the air is being heated (or cooled). This system has two states, $\Sigma = \{heating, cooling\}$. There are four transitions, each of which has a guard that specifies the conditions under which the transition

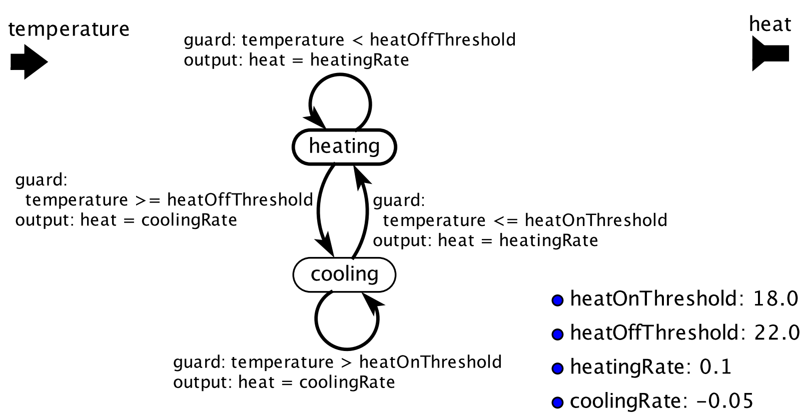


Figure 6.5: FSM model of a thermostat.

is taken. The transitions show the values produced on the output ports when the transition is taken. When the system is in the *heating* state, if the *temperature* input is less than *heatOffThreshold* (22.0), then the output value is *heatingRate* (0.1). When the *temperature* input becomes greater than or equal to *heatOffThreshold*, then the FSM changes to the *cooling* state and produces output value given by *coolingRate* (-0.05). Notice that the guards are mutually exclusive, in that in each state, it is not possible for guards on two of the outgoing transitions to evaluate to true. This makes the machine deterministic.

The FSM of Figure 6.5 is embedded in an SDF model as shown in Figure 6.6. The Temperature Model actor, whose definition is shown in Figure 6.7, models changes in the ambient temperature value based on the the output of the FSMActor. The system's output is plotted in Figure 6.8.

6.2 Structure and Execution of an FSM

As shown in the previous example, an FSM contains a set of states and transitions. One of the states is an **initial state**, and any number of states may be final states. Each transition has a guard **expression**, any number of output actions, and any number of set actions. At the start of execution, the state of the actor is set to the initial state. Subsequently, each firing of the actor executes a sequence of steps as follows. In the fire phase of execution, the actor

1. reads inputs;
2. evaluates guards on outgoing transitions of the current state;
3. chooses a transition whose guard evaluates to true; and
4. executes the output actions on the chosen transition, if any.

In the postfire phase, the actor

5. executes the set actions of the chosen transition, which sets parameter values; and
6. changes the current state to the destination of the chosen transition.

Each of these steps is explained in more detail below. Table 6.1 summarizes the Ptolemy II notations for FSM transitions (without hierarchy), which follow those in Kieler (Fuhrmann and Hanxleden, 2010) and Klepto (Motika et al., 2010), that are explained in this chapter.

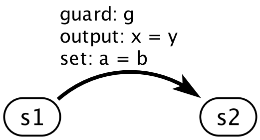
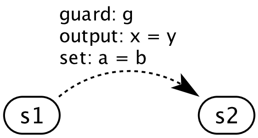
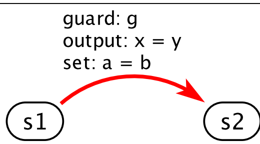
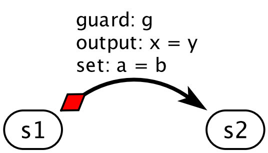
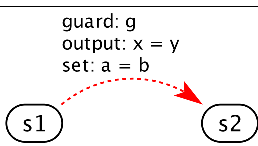
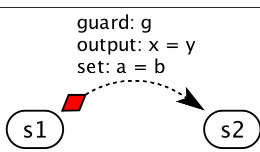
notation	description
	An **ordinary transition**. Upon firing, if the guard g is `true` (or if no guard is specified), then the FSM will choose the transition and produce the value y on output x. Upon transitioning, the actor will set the variable a to have value b.
	A **default transition**. Upon firing, if no other non-default transition is enabled and the guard g is `true`, then the FSM actor will choose this transition, produce outputs, and set variables in the same manner as above.
	A **nondeterministic transition** This transition allows another nondeterministic transition to be enabled in the same iteration. One of the enabled transitions will be chosen nondeterministically.
	An **immediate transition**. If state *s1* is the current state, then this is like an ordinary transition. However, if state *s1* is the destination state of some transition that will be taken and the guard g is `true`, then the FSM will also immediately transition to *s2*. In this case, there will be two transitions in a single iteration. The output x will be set to value y upon firing, and the variable a will be set to b upon transitioning. If more than one transition in a chain of immediate transitions sets an output or variable, then the last transition will prevail.
	A **nondeterministic default transition**. A nondeterministic transition with the (lower) priority of a default transition.
	An **immediate default transition**. An immediate transition with the (lower) priority of a default transition, compared with other immediate transitions.

Table 6.1: Summary of FSM transitions and their notations, which may be combined to indicate combinations of transition types. For example, a nondeterministic immediate default transition will be colored red, have the initial diamond, and be rendered with dotted lines.

6.2.1 Defining Transition Guards

Defining appropriate guards on state transitions is a critical part of creating a finite state machine. As we discuss below, however, the behavior of some guard expressions may cause unexpected results, depending on the director chosen to run the model. In particular, different directors handle absent input values in different ways, which can cause guard expressions to be evaluated in a manner that may seem counterintuitive.

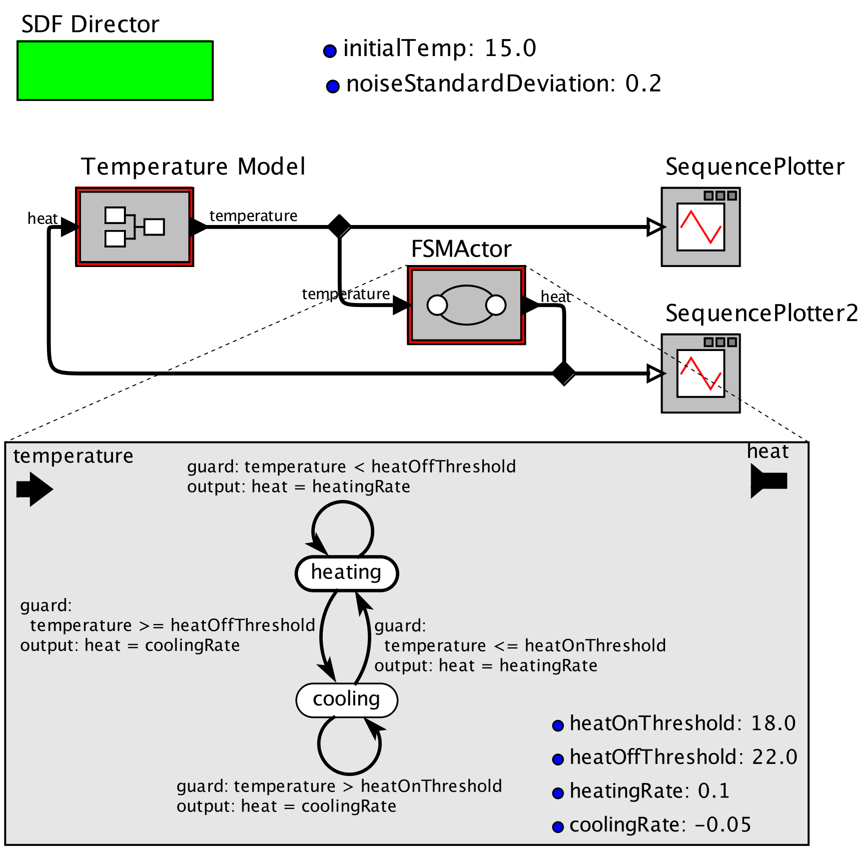


Figure 6.6: The FSM model of a thermostat of Figure 6.5 embedded in an SDF model. The Temperature Model actor is shown in Figure 6.7. [online]

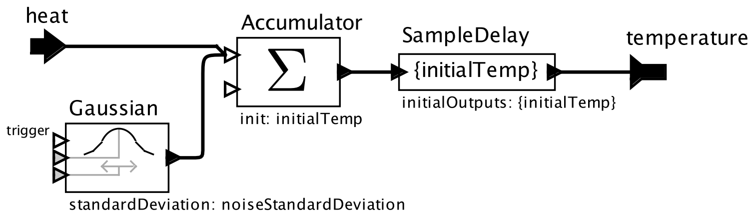


Figure 6.7: The Temperature Model composite actor of Figure 6.6.

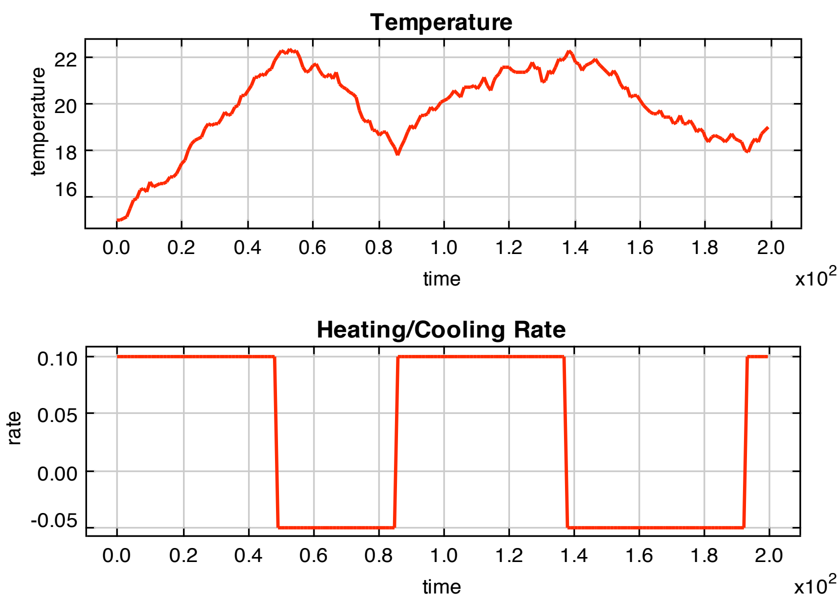


Figure 6.8: Two plots generated by Figure 6.6, showing the temperature (above) and the heating rate (below), which reflects whether the heater is on or off. Both are shown as a function of time.

Each transition has a **guard**, which is a predicate (a boolean-valued expression) that can depend on inputs to the state machine, parameters and variables, and outputs of mode refinements (which are explained in Chapter 8).

> **Example 6.2:** In Figure 6.5, in the guard expression
>
> `temperature < heatOffThreshold`,
>
> the variable `temperature` refers to the current value in the port named *temperature*, and `heatOffThreshold` refers to the parameter named *heatOffThreshold*.

<table>
<tr><th></th><th>guard</th><th>description</th></tr>
<tr><td>1</td><td></td><td>A blank guard always evaluates to true.</td></tr>
<tr><td>2</td><td><code>p_isPresent</code></td><td>True if there is a token at port p.</td></tr>
<tr><td>3</td><td><code>p</code></td><td>True if there is a token at port p and it has value <code>true</code>.</td></tr>
<tr><td>4</td><td><code>!p</code></td><td>True if there is a token at port p and it has value <code>false</code>.</td></tr>
<tr><td>5</td><td><code>p > 0</code></td><td>True if there is a token at port p and it has value greater than zero.</td></tr>
<tr><td>6</td><td><code>p > a</code></td><td>True if there is a token at port p and it has value greater than the value of the parameter a.</td></tr>
<tr><td>7</td><td><code>a > 0</code></td><td>True if parameter a has value greater than 0.</td></tr>
<tr><td>8</td><td><code>p && q</code></td><td>True if ports p and q both have tokens with value <code>true</code>.</td></tr>
<tr><td>9</td><td><code>p || q</code></td><td>True if port p is present and true or if p is present and false and q is present and true.</td></tr>
<tr><td>10</td><td><code>p_0 > p_1</code></td><td>True if port p has a token on both channel 0 and channel 1 and the token on channel 0 is larger than the one on channel 1.</td></tr>
<tr><td>11</td><td><code>p_1_isPresent && (p_0 || p_1)</code></td><td>True if port p has a token on both channel 0 and channel 1 and one of the two tokens is <code>true</code>.</td></tr>
<tr><td>12</td><td><code>timeout(t)</code></td><td>True when time <code>t</code> has elapsed since entering the source state.</td></tr>
</table>

Table 6.2: Examples of guard expressions, where *p* and *q* are ports, and *a* is a parameter.

A few examples of valid guards are given in Table 6.2 for an FSM with input ports *p* and *q* and parameter *a*.

As shown in line 2 of the table, for any port *p*, the symbol `p_isPresent` may be used in guard expressions. This is a boolean that is true if an input token is present on port *p*. Conversely, the expression `!p_isPresent` evaluates to true when *p* is absent. Note that in domains where *p* is never absent, such as PN, this expression will never evaluate to true.

If port *p* has no input tokens (it is *absent* on all channels), then *all* the guards in the table except number 1 are false. In particular, if *p* has type boolean, and it has no input tokens, then it is possible for both `p` and `!p` to be false. Similarly, it is possible for `p > 0`, `!(p > 0)`, and `p <= 0` to simultaneously evaluate to false. Of course, this can only happen if the FSM is used with a director that can fire it with absent inputs, such as SR and DE.

Note that because of the way absent inputs are treated, guard 9 in the table has a particularly subtle effect. It cannot evaluate to true unless *p* has an input token, but it does not require that *q* have a token. If the intent is that both ports have a token for the transition to become enabled, then the guard should be written

```
q_isPresent && (p || q)
```

It would be clearer, though not strictly necessary, to write

```
(p_isPresent && q_isPresent) && (p || q)
```

In short, any mention of an input port *p* in a guard expression can cause the entire guard expression to evaluate to false if the port *p* is absent. But it may not evaluate to false if the subexpression involving *p* is not evaluated. In particular, the logical OR notated as `||` will not evaluate its right argument if the left argument is true. This is why *q* in guard 9 in the table is not required to be present for the guard to evaluate to true.

A consequence of this evaluation strategy is that erroneous guard expressions may not be detected. For example, if the guard expression is specified as `p.foo()`, but `foo()` is not a defined method on the data type of *p*, then this error will not be detected if *p* is known to be absent. The fact that *p* appears in the guard expression causes it to evaluate to false. Moreover, the expression "`true || p < 10`" always evaluates to true, whether *p* has a token or not.

For multiports with multiple channels, the guard expression can specify a channel using the symbol `p_`i, where i is an integer between 0 and $n - 1$ and n is the number of channels connected to the port. For example, line 10 in Table 6.2 compares input tokens on two channels of the same input port. Similarly, a guard expression may refer to `p_`i`_isPresent`, as shown in line 11.

Line 12 shows a guard expression that can be used to trigger a transition after some time has elapsed. The expression `timeout(t)`, where `t` is a double, becomes true when the FSM has spent `t` time units in the source state. In domains with partial support for time, such as SDF and SR, the transition will be taken at the next firing time of the FSM greater than or equal to `t` (and hence, of course, will only be taken if the *period* parameter of the director is not zero); see Section 3.1.3. In domains with full support for time, such as such as DE and Continuous, covered in later chapters, the transition will be taken exactly `t` time units after entering the source state, unless some other transition becomes enabled sooner.

In all cases, the type of an input port or parameter must match the usage in an expression. For example, the expression `p || q` will trigger an exception if port *p* has type *int*.

6.2.2 Output Actions

Once a transition is chosen, its **output actions** are executed. The output action are specified by the *outputActions* parameter of the transition (see Figure 6.4). The format of an output action is typically *portName* = *expression*, where the expression may refer to input values (as in guard expressions) or to a parameter. For example, in Figure 6.5, the line

```
output: heat = coolingRate
```

specifies that the output port named *heat* should produce the value given by the parameter *coolingRate*.

As explained in the sidebar on page 199, the two classes of state machines are Mealy machines and Moore machines. The above-described behavior constitutes a Mealy machine; a Moore machine can be implemented using state refinements that produce outputs, as explained in Chapter 8.

Multiple output actions may be given by separating them with semicolons, as in `port1 = expression1; port2 = expression2`.

6.2.3 Set Actions and Extended Finite State Machines

The **set actions** for a transition can be used to set the values of parameters of the state machine. One practical use for this feature is to create an **extended state machine**, which is a finite state machine extended with a numerical state variable. It is called "extended" because the number of states depends on the number of distinct values that the variable can take. It can even be infinite.

Sidebar: Models of State Machines

State machines are often described in the literature as a five-tuple $(\Sigma, I, O, T, \sigma)$. Σ is the set of states, and σ is the initial state. Nondeterminate state machines may have more than one initial state, in which case $\sigma \subset \Sigma$ is itself a set, although this particular capability is not supported in Ptolemy II FSMs. I is a set of possible valuations of the inputs. In Ptolemy II FSMs, I is a set of functions of the form $i\colon P_i \to D \cup \{absent\}$, where P_i is the set of input ports (or input port names), D is the set of values that may be present on the input ports at a particular firing, and *absent* represents "absent" inputs (i.e., $i(p) = absent$ when `p_isPresent` evaluates to false). O is similarly the set of all possible valuations for the output ports at a particular firing.

For a deterministic state machine, T is a function of the form $T\colon \Sigma \times I \to \Sigma \times O$, representing the transition relations in the FSM. The guards and output actions are, in fact, just encodings of this function. For a nondeterministic state machine (which is supported by Ptolemy II), the codomain of T is the powerset of $\Sigma \times O$, allowing there to be more than one destination state and output valuation.

The classical theory of state machines (Hopcroft and Ullman, 1979) makes a distinction between a **Mealy machine** and a **Moore machine**. A Mealy machine associates outputs with transitions. A Moore machine associates outputs with states. Ptolemy II supports both, using output actions for Mealy machines and state refinements in modal models for Moore machines.

Ptolemy II state machines are actually extended state machines, which require a richer model than that given above. Extended state machines add a set V of variable valuations, which are functions of the form $v\colon N \to D$, where N is a set of variable names and D is the set of values that variables can take on. An extended state machine is a six-tuple $(\Sigma, I, O, T, \sigma, V)$ where the transition function now has the form $T\colon \Sigma \times I \times V \to \Sigma \times O \times V$ (for deterministic state machines). This function is encoded by the transitions, guards, output actions, and set of actions of the FSM.

Example 6.3: A simple example of an extended state machine is shown in Figure 6.9. In this example, the FSM has a parameter called *count*. The transition from the initial state *init* to the *counting* state initializes *count* to 0 in its set action. The *counting* state has two outgoing transitions, one that is a self transition, and the other that goes to the state called *final*. The self transition is taken as long as *count* is less than 5. That transition increments the value of *count* by one in its set actions. In the firing after the value of *count* reaches 5, the transition to *final* is taken. At that firing, the output is set equal to 5. In subsequent firings, the output will always be 5, as specified by the self loop on the *final* state. This model, therefore, outputs the sequence 0, 1, 2, 3, 4, 5, 5, 5, $\cdots$.

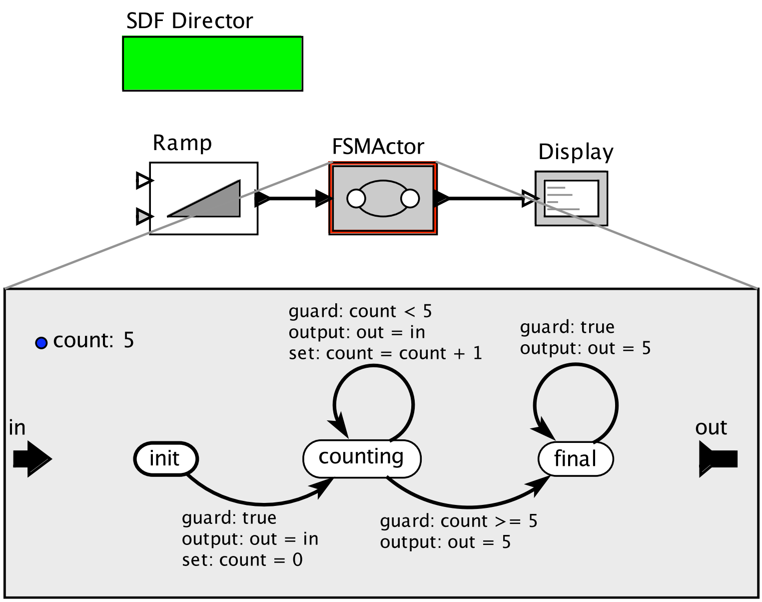


Figure 6.9: An extended state machine, where the *count* variable is part of the state of the system. [online]

6.2.4 Final States

An FSM may have **final states**, which are states that, when entered, indicate the end of execution of the state machine.

Example 6.4: A variant of Example 6.3 is shown in Figure 6.10. This variant has the *isFinalState* parameter of the *final* state set to `true`, as indicated by the double outline around the state. Upon entering that state, the FSM indicates to the enclosing director that it does not wish to execute any more (it does this by returning `false` from its postfire method). As a result, the output sent to the Display actor is the finite sequence 0, 1, 2, 3, 4, 5, 5. Notice the two 5's at the end. This underscores the fact that guards are evaluated *before* set actions are executed. Thus, at the start of the sixth firing, the input to the FSM is 5 and the value of *count* is 4. The self-loop on the *counting* state will be taken, producing output 5. At the start of the next firing, *count* is 5, so the transition to the *final* state is taken, producing another 5.

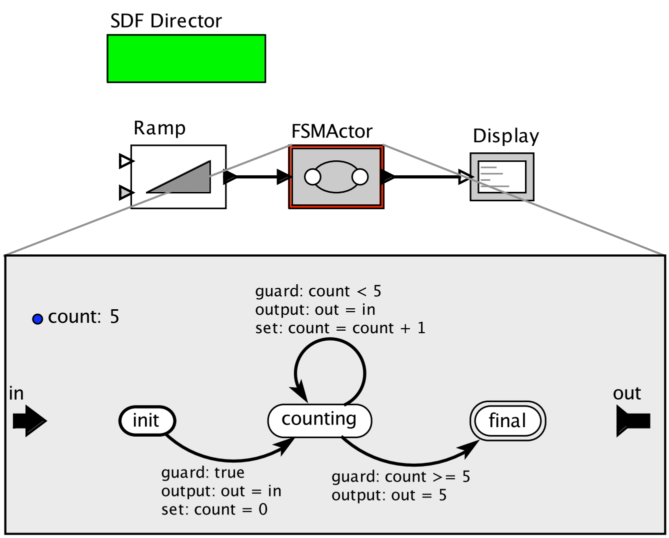


Figure 6.10: A state machine with a final state, which indicates the end of execution of the state machine. [online]

In the iteration in which an FSM enters a state that is marked final, the `postfire` method of the ModalModel or FSMActor returns false. This indicates to the enclosing director that the FSM does not wish to be fired again. Most directors will simply avoid firing the FSM again, but will continue executing the rest of the model. The SDF director, however, is different. Since it assumes the same consumption and production rates for all actors, and since it constructs its schedule statically, it cannot accommodate non-firing actors. Hence, the SDF director will stop execution of the model altogether if *any* actor returns `false` from `postfire`. In contrast, the SR director will continue executing, but all outputs of the now terminated FSM will be absent in subsequent ticks.

Example 6.5: Figure 6.11 shows an SR model that produces a finite count, but unlike Example 6.4, the model does not stop executing when the state machine reaches its final state. The display output is shown for 10 iterations. Notice that after the FSM reaches the final state, the output of the FSM is *absent*. Notice also that the first output of the FSM is *absent*. This is because the transition from

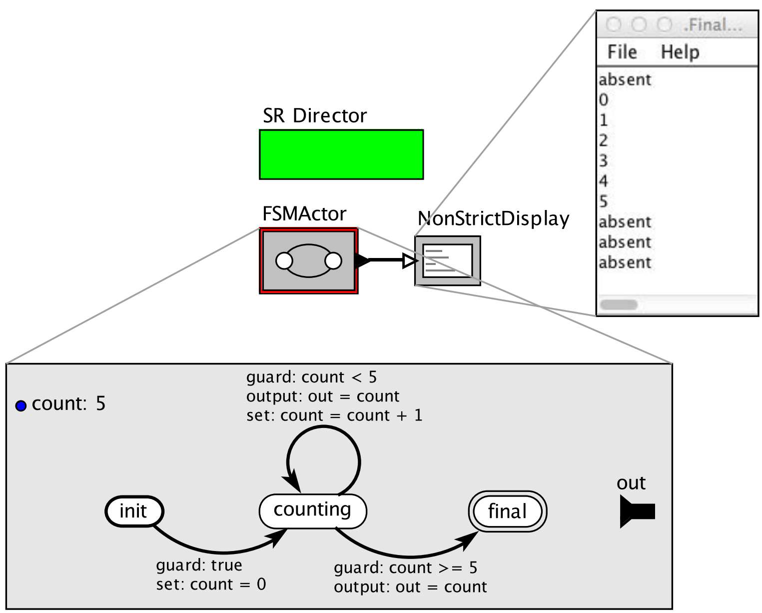


Figure 6.11: A state machine with a final state in an SR model. [online]

init to *counting* does not include any output action. Such a transition would not be compatible with SDF, because actors in SDF are required to produce a fixed number of outputs on each firing.

Notice the use of **NonStrictDisplay**. This actor is similar to Display except that it displays "absent" when the input is absent, whereas Display does not display anything when the input is absent.

As illustrated by the above example, SR supports a notion of absent values. Dataflow domains and PN have no such notion. Failure to produce outputs will starve downstream actors, preventing them from executing. An FSM with a final state in PN will simply stop producing outputs when it reaches the final state. This can result in termination of the entire model if it causes starvation (i.e., if other actors require inputs from the FSM in order to continue).

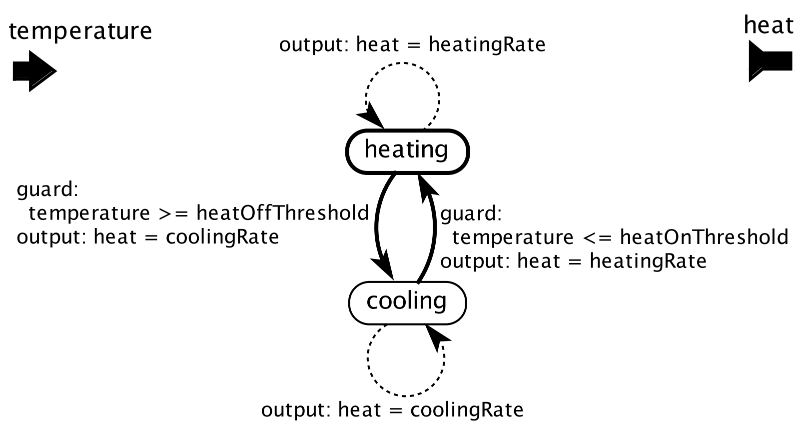


Figure 6.12: An FSM equivalent to that shown in Figure 6.5, but using default self-transitions (indicated with dashed lines). These are taken if the other outgoing transition is not enabled. [online]

6.2.5 Default Transitions

An FSM may have **default transition**s, which are transitions that have the *default* parameter set to true (see Figure 6.4). These transitions become enabled if no other outgoing (non-default) transition of the current state is enabled. Default transitions are shown as dashed arcs rather than solid arcs.

> **Example 6.6:** The thermostat FSM of Figure 6.5 can be equivalently implemented using default transitions as shown in Figure 6.12. Here, the default transitions simply specify that if the outgoing transition to the other state is not enabled, then the FSM should remain in the same state and produce an output.

If a default transition also has a guard expression, then that transition is enabled only if the guard evaluates to true *and* there are no other non-default transitions enabled. Default transitions, therefore, provide a rudimentary form of **priority**; non-default transitions have priority over default transitions. Unlike some state-machine languages, such as SyncCharts, Ptolemy II FSMs offer only two levels of priority, although it is always possible to encode arbitrary priorities using guards. Note that using default transitions with timed models of computation can be somewhat tricky; see Section 8.5 in Chapter 8.

Default transitions can often be used to simplify guard expressions, as illustrated by the following example.

> **Example 6.7:** Consider the counting state machine of Example 6.5, shown in Figure 6.11. We can add a *reset* input, as shown in Figure 6.13, to enable the count to be reset. If *reset* is present and true, then the state machine returns to the *init* state. However, the implementation in Figure 6.13 must then be modified; the two existing transitions out of the *counting* state must include an additional clause
>
> ```
> && (!reset_isPresent || !reset)
> ```
>
> This clause ensures that the self loop on the *counting* state is only taken if the *reset* input is either absent or false. Without this clause, the state machine would have become nondeterministic, since two of the transitions out of the *counting* state could

have become simultaneously enabled. This clause, however, increases the visual complexity of the guard expression, which is functionally quite simple. Figure 6.14 shows a version where default transitions are used instead. These indicate that the machine should count only if the *reset* input is not present and `true`.

For this machine, if *reset* is present in the fourth firing, for example, then the first few outputs will be `absent, 0, 1, 2, absent, 0, 1`. In the iteration when *reset* is present and true, the output "2" is produced, and then the machine starts over.

6.2.6 Nondeterministic State Machines

If more than one guard evaluates to true at any time, then the FSM is a **nondeterministic FSM** (unless one of the guards is on a default transition and the other is not). The transitions that are simultaneously enabled are called **nondeterministic transitions**. By default, transitions are not allowed to be nondeterministic, so if more than one guard evaluates to true, Ptolemy will issue an exception similar to the below:

Nondeterministic FSM error: Multiple enabled transitions found but not all of them are marked nondeterministic.

in ... name of a transition not so marked ...

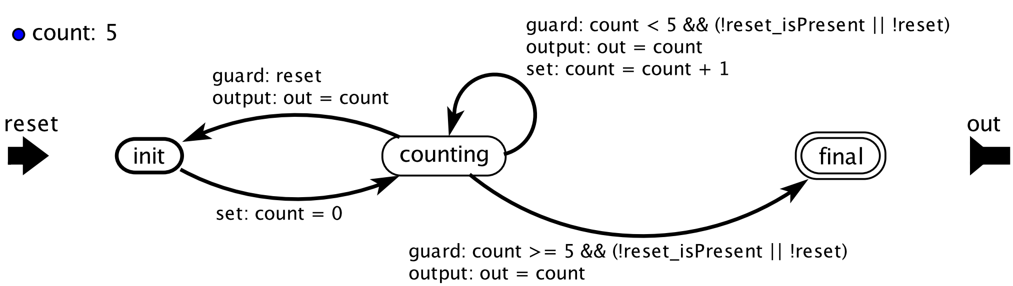


Figure 6.13: A state machine like that in Figure 6.11, but with an additional *reset* input port. [online]

There are cases, however, where it is desirable to allow nondeterministic transitions. In particular, nondeterministic transitions provide good models of systems that can exhibit multiple behaviors for the same inputs. They can also be useful for modeling the possibility of fault conditions where there is no information about the likelihood of a fault occurring. Nondeterministic transitions are allowed by setting the *nondeterministic* parameter to `true` on every transition that can be enabled while another another transition is enabled (see Figure 6.4).

Example 6.8: A model of a faulty thermostat is shown in Figure 6.15. When the FSM is in the *heating* state, both outgoing transitions are enabled (their guards are both `true`), so either one can be taken. Both transitions are marked nondeterministic, indicated by the red arc color. A plot of the model's execution is shown in Figure 6.16. Note that the heater is on for relatively short periods of time, causing the temperature to hover around 18 degrees, the threshold at which the heater is turned on.

In a nondeterministic FSM, if more than one transition is enabled and they are all marked nondeterministic, then one is chosen at random in the `fire` method of the ModalModel or FSMActor. If the `fire` method is invoked more than once in an iteration (see Section

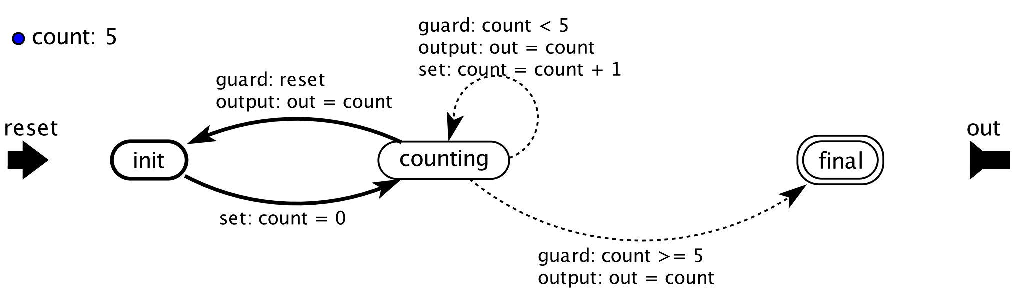


Figure 6.14: A state machine like that in Figure 6.13, but using default transitions to simplify the guard expressions. [online]

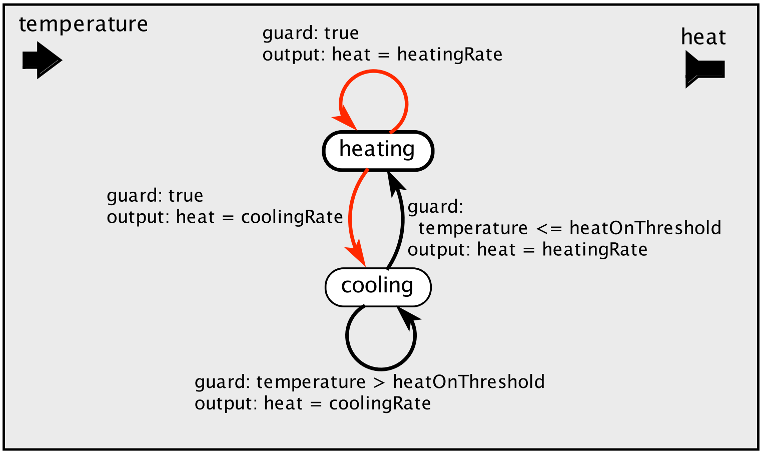


Figure 6.15: A model of a faulty thermostat that nondeterministically switches from heating to cooling. [online]

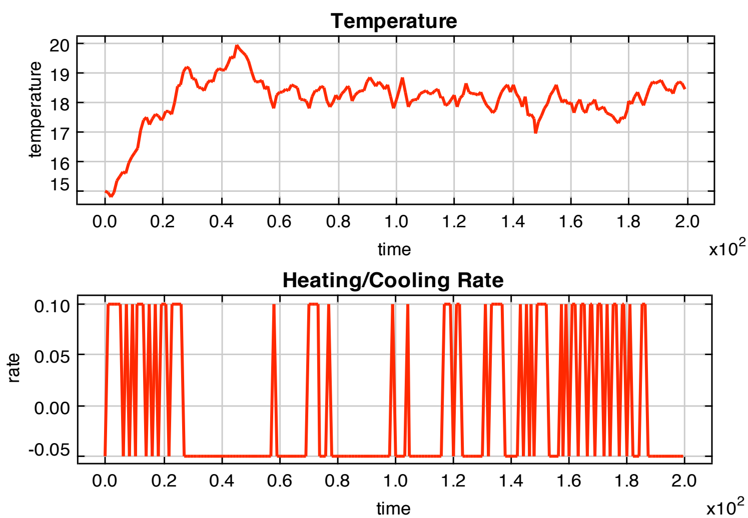


Figure 6.16: Plot of the thermostat FSM of Figure 6.15, a variant of Figure 6.5.

6.4 below), then subsequent invocations in the same iteration will always choose the same transition.

6.2.7 Immediate Transitions

Thus far we have only considered the case where each firing of an FSM results in a single transition. It is possible, however, to take more than one transition in a single firing, by using an **immediate transition**. If a state A has an immediate transition to another state B, then that transition will be taken in the same firing as a transition into state A if the guard on the immediate transition is true. The transition into and out of A will occur in the same firing. In this case, A is called a **transient state**.

Example 6.9: In Example 6.7, the output of the thermostat is absent in the first iteration and in the iteration immediately following a *reset*. These absent outputs can be avoided by marking the transition from *init* to *counting* immediate, as shown in Figure 6.17. This change has two effects. First, when the model is initialized, the transition from *init* to *counting* is taken immediately (during initialization), which sets the `count` variable to 0. Thus, in the first iteration of the state machine, it will be in state *counting*. This prevents the initial *absent* from appearing at the output. Instead, the output in the first iteration will be 0.

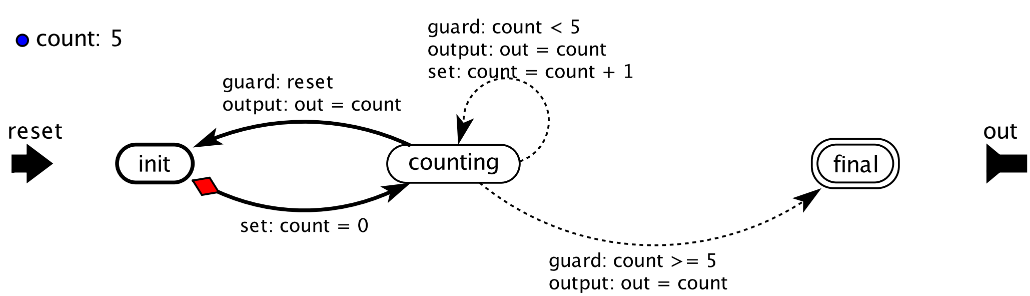


Figure 6.17: A state machine like that in Figure 6.14, but using an immediate transition to prevent absent outputs before counting. The immediate transition is indicated by a red diamond. [online]

The second effect is that in the *counting* state, if the *reset* input is present and true, then the machine will transition from *counting* to *init*, and back to *counting*, *in the same iteration*, reseting the `count` variable to 0.

For this machine, if *reset* is present in the fourth firing (for example), then the first few outputs displayed will be as follows: `0, 1, 2, 3, 0, 1`. In the iteration when *reset* is present and true, the output "3" is produced by the transition back to *init*, and then the machine starts over.

Note that a transient state is not quite the same thing as a state in which the state machine spends zero time; because of superdense time in Ptolemy II, a state machine may spend zero time in a state, but the transition into the state and out of the state occur in different firings, at different microstep indexes (see sidebar on page 210).

When a state machine reacts, the state at the start of the reaction is called the **current state**. The current state may have immediate transitions coming out of it. For this to be the case, it is necessary that in the previous reaction the guards on these transitions evaluated to false; otherwise, the state would have been transient and would not have become the current state. When the current state has both immediate and non-immediate transitions out of it, those two classes of transitions are treated identically. There is no distinction between them, and no priority order between them. If an immediate and non-immediate transition out of the current state both have guards that evaluate to true, then either one of them needs to a default transition, or both of them need to marked nondeterministic.

If there are immediate transitions out of the initial state, then their guards are evaluated when the FSM is initialized, and if the guard is true, then the transition is taken before the FSM starts executing. Notice that in some domains, such as SR, outputs produced prior to the start of execution will never be observed by the destination, so in those domains, an immediate transition out of an initial state should not produce outputs.

Immediate, default, and nondeterministic transitions can be used in combination to sometimes dramatically simplify a state machine diagram, as illustrated in the following example.

Example 6.10: An **ABRO** state machine is a class of FSM that waits for a signal *A* and a signal *B* to arrive. Once both have arrived, it produces an output *O*, unless a reset signal *R* arrives, in which case it starts all over, waiting for *A* and *B*.

This pattern is used to model a variety of applications. For example, *A* may represent a buyer for a widget, *B* a seller, and *O* the occurrence of a transaction. *R* may represent the widget becoming unavailable.

Specifically, the system has boolean-valued inputs *A*, *B*, and *R*, and a boolean-valued output *O*. Output *O* will be present and true as soon as both inputs *A* and *B* have been present and true. In any iteration where *R* is present and true, the behavior is restarted from the beginning.

An implementation of this state machine is shown in Figure 6.18. The initial state, *nAnB* (short for "not *A* and not *B*") represents the situation where neither *A* nor *B* has arrived. The state *nAB* represents the situation where *A* has not arrived but *B* has – and so on.

Probing Further: Weakly Transient States

A state machine may spend zero time in a state without the use of immediate transitions. Such a state is called a **weakly transient state**. It is not quite like the transient states of Section 6.2.7, which have immediate transitions that move out of the state *within a single reaction*. A weakly transient state is the final state of one reaction, and the current state of the next reaction, but no model time elapses between reactions. Note that any state that has default transitions (without guards or with guards that evaluate to true immediately) is a transient state, since exiting the state is always immediately enabled after entering the state.

When a transition is taken in an FSM, the FSMActor or ModalModel calls the `fireAtCurrentTime` method of its enclosing director. This method requests a new firing in the next microstep regardless of whether any additional inputs become available. If the director honors this request (as timed directors typically do), then the actor will be fired again at the current time, one microstep later. This ensures that if the destination state has a transition that is immediately enabled (in the next microstep), then that transition will be taken before model time has advanced. Note also that in a modal model (see Section 6.3 and Chapter 8), if the destination state has a refinement, then that refinement will be fired at the current time in the next microstep. This is particularly useful for continuous-time models (see Chapter 9), since the transition may represent a discontinuity in otherwise continuous signals. The discontinuity translates into two distinct events with the same time stamp.

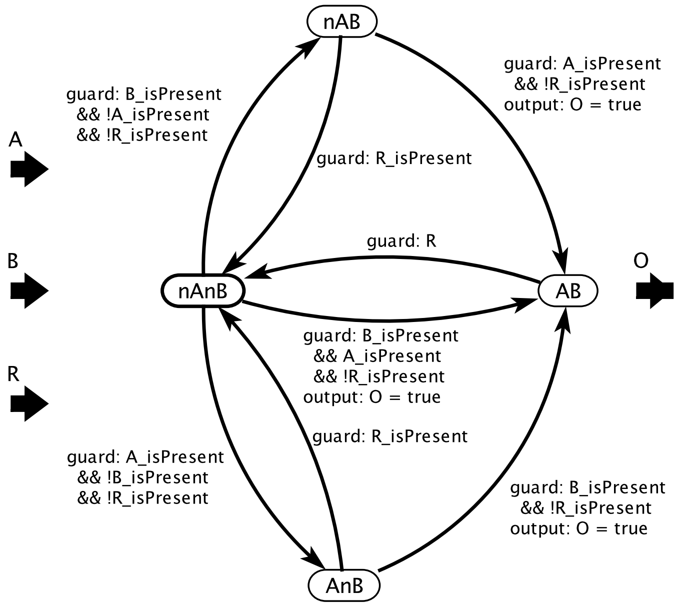


Figure 6.18: A brute-force implementation of the classic ABRO state machine. [online]

The guard expressions in Figure 6.18 can be difficult to read (although it can get much worse — see Exercise 4). An alternative implementation that is easier to read (once you are familiar with the transition notation, which are summarized in Table 6.1) is shown in Figure 6.19. This example uses nondeterminate, default, and immediate transitions to simplify guard expressions.

Immediate transitions may write to the same output ports that are written to by previous transitions taken in the same iteration. FSMs are **imperative**, with a well-defined sequence of execution, so the output of the FSM will be the *last* value written to an out-

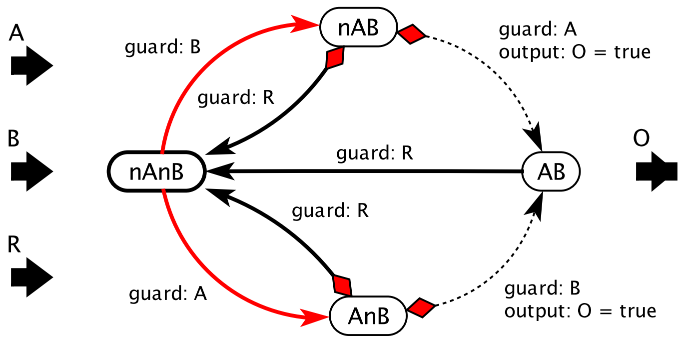


Figure 6.19: An implementation of the ABRO state machine that leverages default transitions, immediate transitions, and nondeterministic transitions to simplify the guard expressions. [online]

put port in a chain of transitions. Similarly, immediate transitions may write to the same parameter in their set actions, overwriting values written in previously taken transitions.

6.3 Hierarchical FSMs

It is always possible (and encouraged) to construct an FSM by using the ModalModel actor rather than the FSMActor. The ModalModel actor allows states to be defined with one or more **refinements**, or submodels. In Chapter 8 we discuss the general form of this approach, called modal models, where the submodel can be an arbitrary Ptolemy II model. Here, we consider only the special case where the submodel is itself an FSM. The approach yields a **hierarchical FSM** or **hierarchical state machine**.

To create a hierarchical FSM, select `Add Refinement` in the context menu for a state, and choose `State Machine Refinement`, as shown in Figure 6.20. This creates a state machine refinement (submodel) that can reference the higher-level state machine's input ports and write to the output ports. The refinement's states can themselves have refinements (either `Default Refinements` or `State Machine Refinements`).

In addition to the transition types of Table 6.1, hierarchical state machines offer additional transition types, summarized in Table 6.3. These will be explained below.

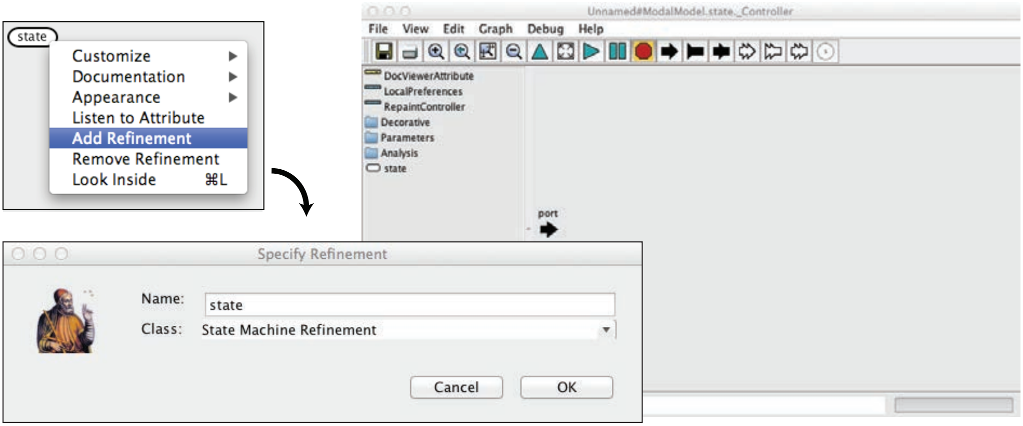


Figure 6.20: How to add a refinement to the state of a ModalModel.

6.3.1 State Refinements

The execution of a modal model follows a simple pattern. When the modal model fires, first, its refinements fire. Then its guards are evaluated and a transition may be chosen. A refinement may produce outputs, and so may a transition, but since the refinement is fired first, if both produce output values on the same port, the transition will overwrite the value produced by the refinement. Execution is strictly sequential.

Postfire is similar. When the modal model postfires, it first postfires its refinements, and then commits the transition, executing any set actions on the transition. Again, if the refinement and the transition write to the same variable, the transition prevails.

Note that a state can have more than one refinement. To create a second refinement, invoke `Add Refinement` again. Refinements are executed in order, so if two refinements of the same state produce a value on the same output or update the same variable, then the second one will prevail.[†] The last output value produced becomes the output of the modal model for the firing. It overwrites the actions of the first, as with chains of immediate transitions. To change the order in which refinements execute, simply double click on the state and edit the *refinementName* parameter, which is a comma-separated list of refinements.

[†] If you wish to have refinements that execute concurrently, see Chapter 8.

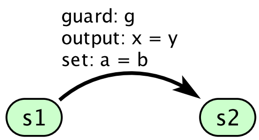
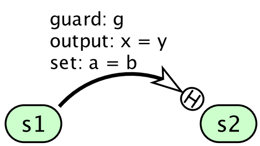
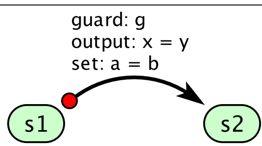
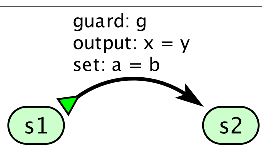
notation	description
	An **ordinary transition**. Upon firing, the refinement of the source state is fired first, and then if the guard g is `true` (or if no guard is specified), then the FSM will choose the transition. It will produce the value y on output port x, overwriting any value that the source state refinement might have produced on the same port. Upon transitioning (in postfire), the actor will set the variable a to have value b, again overwriting any value that the refinement may have assigned to a. Finally, the refinements of state *s2* are reset to their initial states. For this reason, these transitions are sometimes called **reset transitions**.
	A **history transition**. This is similar to an ordinary transition, except that when entering state *s2*, the refinements of that state are *not* reset to their initial states, but rather resume from whatever state they were in when the refinement was last active. On first entry to *s2*, of course, the refinements will start from their initial states.
	A **preemptive transition**. If the current state is *s1* and the guard is true, then the state refinement (the FSM sub-model) for *s1* will not be invoked prior to the transition.
	A **termination transition**. If all refinements of state *s1* reach a final state and the guard is true, then the transition is taken.

Table 6.3: Summary of FSM transitions and their notations for hierarchical state machines. Here, we assume all refinements are themselves FSMs, although in Chapter 8 we will see that refinements can be arbitrary Ptolemy II models.

It is also possible for a refinement to be the refinement of more than one state. To add a refinement to a state that is already the refinement of another state, double click on the state and insert the name of the refinement into the *refinementName* parameter.

6.3.2 Benefits of Hierarchical FSMs

Hierarchical FSMs can be easier to understand and more modular than flat FSMs, as illustrated in the following example.

Example 6.11: A hierarchical FSM that combines the normal and faulty thermostats of Examples 6.1 and 6.8 is shown in Figure 6.21.

In this model, a Bernoulli actor is used to generate a *fault* signal (which will be `true` with some fixed probability, shown as 0.01 in the figure). When the *fault* signal is `true`, the modal model will transition to the faulty state and remain there for ten iterations before returning to the *normal* mode. The state refinements are the same as those in Figures 6.12 and 6.15, modeling the normal and faulty behavior of the thermostat.

The transitions from *normal* to *faulty* and back in top-level FSM are preemptive transitions, indicated by the red circles on their stem, which means that when the guards on those transitions become true, the refinement of the current state is not executed, and the refinement of the destination state is reset to its initial state. In contrast, the self-loop transition from *faulty* back to itself is a history transition, which, as we will explain below, means that when the transition is taken, the destination state refinement is not initialized. It resumes where it left off.

An equivalent flat FSM is shown in Figure 6.22. Arguably, the hierarchical diagram is easier to read and more clearly expresses the separate normal and faulty mechanisms and how transitions between these occur. See Exercise 7 of this chapter for a more dramatic illustration of the potential benefits of using a hierarchical approach.

Notice that the model in Figure 6.21 combines a **stochastic state machine** with a nondeterministic FSM. The stochastic state machine has random behavior, but an explicit probability model is provided in the form of the Bernoulli actor. The nondeterministic FSM also has random behavior, but no probability model is provided.

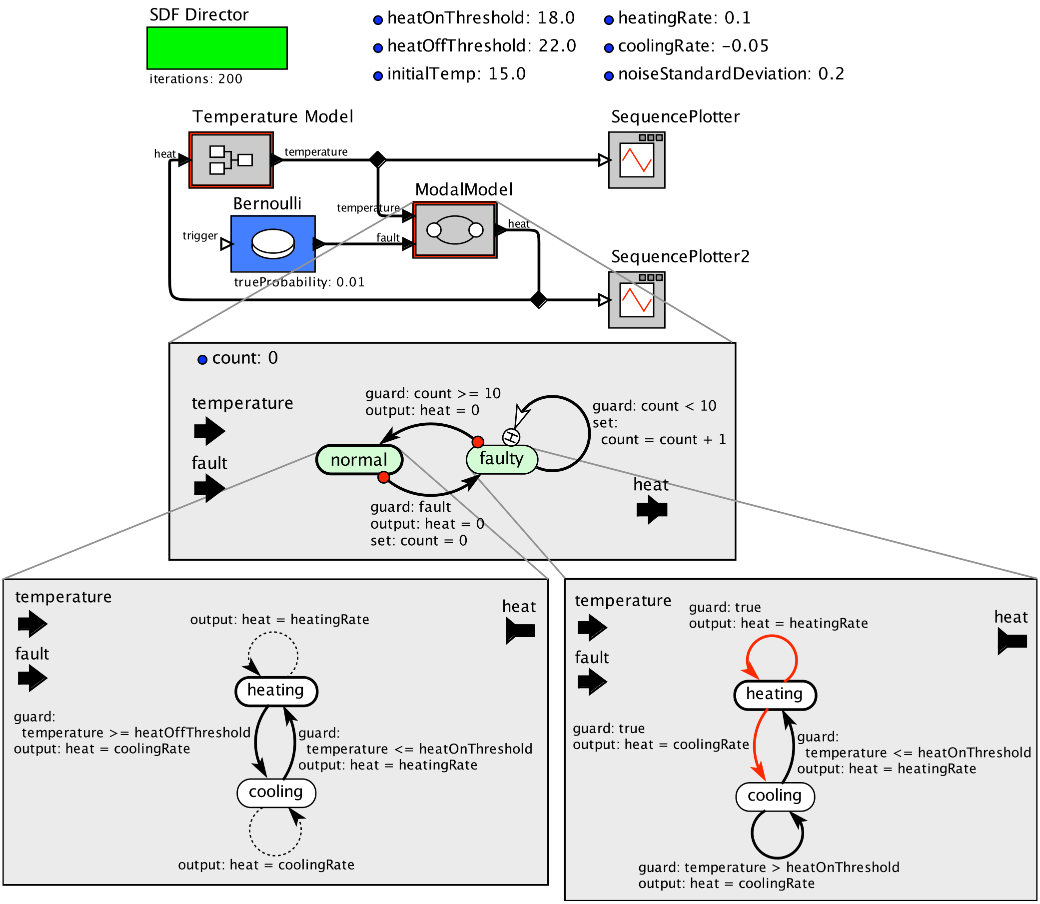


Figure 6.21: A hierarchical FSM that combines the normal and faulty thermostats of Examples 6.1 and 6.8. [online]

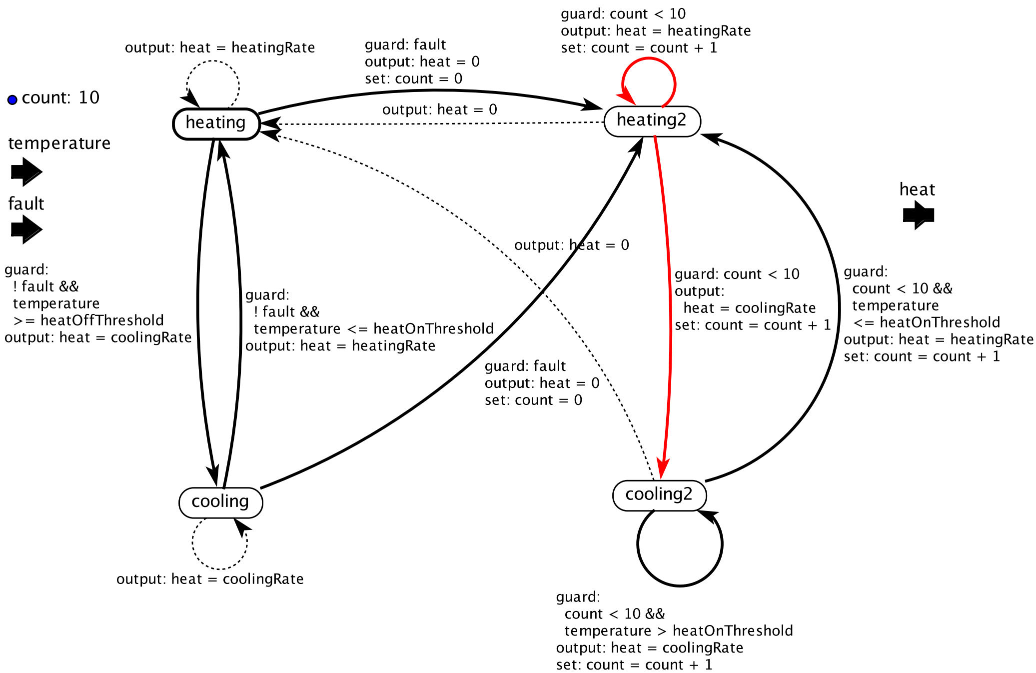


Figure 6.22: A flat FSM version of the hierarchical FSM of Figure 6.21. [online]

6.3.3 Preemptive and History Transitions

A state that has a refinement is shaded in light green in Vergil, as shown in Figure 6.21. The top-level FSM in that figure also uses two new specialized transitions, which we now explain (see Table 6.3).

The first is a **preemptive transition**, indicated by red circle at the start of the transition. In a firing where the current state has a preemptive transition leading to another state, the refinement does not fire if the guard on the transition is true. It is preempted by the transition.‡ If the transition out of the *normal* state was not preemptive in this example, then in an iteration where the *fault* input is true and present, the refinement FSM of the *normal* state would nonetheless produce a normal output. The preemptive transition prevents this

‡In the literature, this is sometimes called **strong preemption**, where **weak preemption** refers to a normal transition out of a state that allows the refinement to execute.

from occurring. In iterations where a fault occurs, the preemptive transition generates outputs that are not the normal outputs produced by the *normal* or *faulty* submodels. The model shown in the figure assigns the outputs the value 0 in the iteration when either a transition occurs from *normal* to *faulty*, or vice versa.

A current state may have preemptive, default preemptive, non-preemptive, and default non-preemptive transitions. The guards on these transitions are checked in that order, giving four priority levels. Similarly, immediate transitions may also be preemptive and/or default transitions, so they again have four possible priority levels (see Exercise 9).

The second of the two specialized transitions is a **history transition**, indicated by an outlined arrowhead and a circle with an "H." When such a transition is taken, the refinement of the destination state is *not* initialized, in contrast to an ordinary transition. Instead, it resumes from the state it was last in when the refinement was previously active. In Figure 6.21, the self transition from *faulty* back to itself is a history transition because its purpose is to just count iterations, not to interfere with the execution of the refinement.

Transitions that are not history transitions are often called **reset transitions**, because they reset the destination refinements.

6.3.4 Termination Transitions

A **termination transition** is a transition that is enabled only when the refinements of the current state reach a final state. The following example uses such a transition to significantly simplify the ABRO example.

Example 6.12: A hierarchical version of the ABRO model of Figure 6.19 is shown in Figure 6.23. At the top level is a single state and a preemptive reset transition that is triggered by an input *R*. Below that is a two-state machine that waits in *waitAB* until the two refinements of *waitAB* transition reach a final state. Its transition is a termination transition, indicated by the green diamond at its stem. When that the termination transition is taken, it will transition to the final state called *done* and produce the output *O*. Each refinement of *waitAB* waits for one of *A* or *B*, and once it receives it, transitions to a final state.

In each firing of the modal model, while in *waitAB*, both of the lowest level refinements execute. In this case, it does not matter in which order they execute.

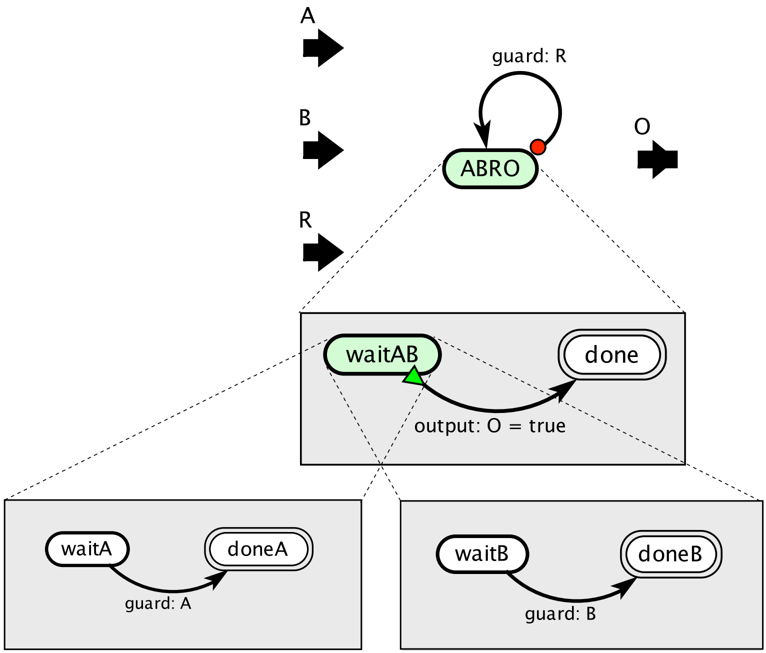


Figure 6.23: A hierarchical version of the ABRO model of Figure 6.19. [online]

Hierarchical machines can be much more compact than their flat counterparts. Exercise 5 (at the end of the chapter), for example, illustrates that if you increase the number of signals that ABRO waits for (making, for example **ABCRO**, with three inputs), then the flat machine gets very large very quickly, whereas the hierarchical machine scales linearly.

This use of transitions triggered by entering final states in the refinements is sometimes referred to as **normal termination**. The submodel stops executing when it enters a final state and can be restarted by a reset. André (1996) points out that specialized termination transitions are not really necessary, as local signals can be used instead (see Exercise 6). But they can be convenient for making diagrams simpler.

6.3.5 Execution Pattern for Modal Models

Execution of a ModalModel proceeds in two phases, fire and postfire. In `fire`, it:

1. reads inputs, makes inputs available to current state refinements, if any;
2. evaluates the guards of preemptive transitions out of the current state;
3. if a preemptive transition is enabled, the actor choses that transition and executes its output actions.
4. if no preemptive transition is enabled, then it:
   a. fires the refinements of the current state (if any), evaluating guards on transitions of the lower-level FSM and producing any required outputs;
   b. evaluates guards on the non-preemptive transitions of the upper-level FSM (which may refer to outputs produced by the refinement); and
   c. executes the output actions of the chosen transition of the upper-level FSM.

In `postfire`, the ModalModel actor

1. postfires the refinements of the current state if they were fired, which includes executing set actions on any chosen transitions in the lower-level FSM and committing its state change;
2. executes the set actions of the chosen transition of the upper-level FSM;
3. changes the current state to the destination of the chosen transition; and
4. initializes the refinements of the destination state if the transition is a reset transition.

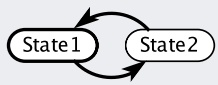
The transitions out of a state are checked in the following order:

1. preemptive transitions,
2. preemptive default transitions,
3. non-preemptive transitions, and
4. non-preemptive default transitions.

For transitions emerging from the current state (the state at the start of a reaction), no distinction is made between immediate and non-immediate transitions. The distinction only matters upon entering a state, when immediate transitions are also checked in the same order as above (preemptive, preemptive default, non-preemptive, and non-preemptive default immediate transitions).

Probing Further: Internal Structure of an FSM

FSMActor is a subclass of CompositeEntity, just like CompositeActor. Internally, it contains some number of instances of State and Transition, which are subclasses of Entity and Relation respectively. The simple structure shown below:



is represented in MoML as follows:

```
<entity name="FSMActor" class="...FSMActor">
  <entity name="State1" class="...State">
    <property name="isInitialState" class="...Parameter"
      value="true"/>
  </entity>
  <entity name="State2" class="...State"/>
  <relation name="relation" class="...Transition"/>
  <relation name="relation2" class="...Transition"/>
  <link port="State1.incomingPort" relation="relation2"/>
  <link port="State1.outgoingPort" relation="relation"/>
  <link port="State2.incomingPort" relation="relation"/>
  <link port="State2.outgoingPort" relation="relation2"/>
</entity>
```

The same structure can be specified in Java as follows:

```
  import ptolemy.domains.modal.kernel.FSMActor;
  import ptolemy.domains.modal.kernel.State;
  import ptolemy.domains.modal.kernel.Transition;
  FSMActor actor = new FSMActor();
  State state1 = new State(actor, "State1");
  State state2 = new State(actor, "State2");
  Transition relation = new Transition(actor, "relation");
  Transition relation2 = new Transition(actor, "relation2");
  state1.incomingPort.link(relation2);
  state1.outgoingPort.link(relation);
  state2.incomingPort.link(relation);
  state2.outgoingPort.link(relation2);
```

Thus, above, we see three distinct concrete syntaxes for the same structure. A ModalModel contains an FSMActor, the controller, plus each of the refinements.

Probing Further: Hierarchical State Machines

State machines have a long history in the theory of computation (Hopcroft and Ullman, 1979). An early model for hierarchical FSMs is **Statecharts**, developed by Harel (1987). As with Ptolemy II FSMs, states in Statecharts can have multiple refinements, but unlike ours, in Statecharts the refinements are not executed sequentially. Instead, they execute concurrently, roughly under the synchronous-reactive model of computation. We achieve the same effect with modal models, as shown in Chapter 8. Another feature of Statecharts, not provided in Ptolemy II, is the ability for a transition to cross levels of the hierarchy.

The **Esterel** synchronous language also has the semantics of hierarchical state machines, although it is given a textual syntax rather than a graphical one (Berry and Gonthier, 1992). Esterel has a rigorous SR semantics for concurrent composition of state machines (Berry, 1999). **SyncCharts**, which came later, provides a visual syntax (André, 1996).

PRET-C (Andalam et al., 2010), **Reactive C** (**RC**) (Boussinot, 1991), and **Synchronous C** (**SC**) (von Hanxleden, 2009) are C-based languages inspired by Esterel that support hierarchical state machines. In both RC and PRET-C, state refinements (which are called "threads") are executed sequentially in a fixed, static order. The PRET-C model is more restricted than ours, however, in that distinct states cannot share the same refinements. A consequence is that refinements will always be executed in the same order. The model in Ptolemy II, hence, is closer to that of SC, which uses "priorities" that may be dynamically varied to determine the order of execution of the refinements.

Both RC and PRET-C, like our model, allow repeated writing to outputs, where the last write prevails. In RC, however, if such an overwrite occurs after a read has occurred, a runtime exception is thrown. Our model is closer to that of PRET-C, where during an iteration, outputs function like variables in an ordinary imperative language. Like RC and PRET-C, only the last value written in an iteration is visible to the environment on the output port of the FSM. In contrast, Esterel provides **combine operators**, which merge multiple writes into a single value (for example by adding numerical values).

6.4 Concurrent Composition of State Machines§

Since FSMs can be used in any Ptolemy II domain, and most domains have a concurrent semantics, a Ptolemy user has many ways to construct concurrent state machines. In most domains, an FSM behaves just like any other actor. In some domains, however, there are some subtleties. In this section we particularly focus on issues that arise when constructing feedback loops in domains that perform fixed-point iteration, such as the SR and Continuous domains.

As described earlier, when an FSM executes, it performs a sequence of steps in the `fire` method, and additional steps in the `postfire` method. This separation is important in constructing a fixed point, because the `fire` method may be invoked more than once per iteration while the director searches for a solution, and it cannot include any persistent state changes. Steps 1-4 in the fire method of the FSM read inputs, evaluate guards, choose a transition, and produce outputs – but they do not commit to a state transition or change the value of any local variables.

Example 6.13: Consider the example in Figure 6.24, which requires that the `fire` method be invoked multiple times. As explained in Chapter 5, execution of an SR model requires the director to find a value for each signal at each tick of a global clock. On the first tick, each of the NonStrictDelay actors places the value shown in its icon on its output port (the values are 1 and 2, respectively). This defines the *in1* value for FSMActor1 and the *in2* value for FSMActor2. But the other input ports remain undefined. The value of *in2* of FSMActor1 is specified by FSMActor2, and the value of *in1* of FSMActor2 is specified by FSMActor1. This may appear to create a causality loop, but as discussed below, it does not.

In Figure 6.24, note that for all states of the FSMActors, each input port has a guard that depends on the port's value. Thus, it would seem that both inputs need to be known before any output value can be asserted, which suggests a causality loop. However, looking closely at the left FSM, we see that the transition from *state1* to *state2* will be enabled at the first tick of the clock because *in1* has value 1, given by NonStrictDelay1. If the state machine is determinate, then this must be the only enabled transition. Since there are no nondeterministic transitions in the

§This section may be safely skipped on a first reading unless you are particularly focusing on fixed-point domains such as SR and Continuous.

state machine, we can assume this will be the chosen transition. Once we make that assumption, we can assert both output values as shown on the transition (*out1* is 2 and *out2* is 1).

Once we assert those output values, then both inputs of FSMActor2 become known, and it can fire. Its inputs are *in1* = 2 and *in2* = 2, so in the right state machine the transition from *state1* to *state2* is enabled. This transition asserts that *out2* of FSMActor2 has value 1, so now both inputs to FSMActor1 are known to have value 1. This reaffirms that FSMActor1 has exactly one enabled transition, the one from *state1* to *state2*.

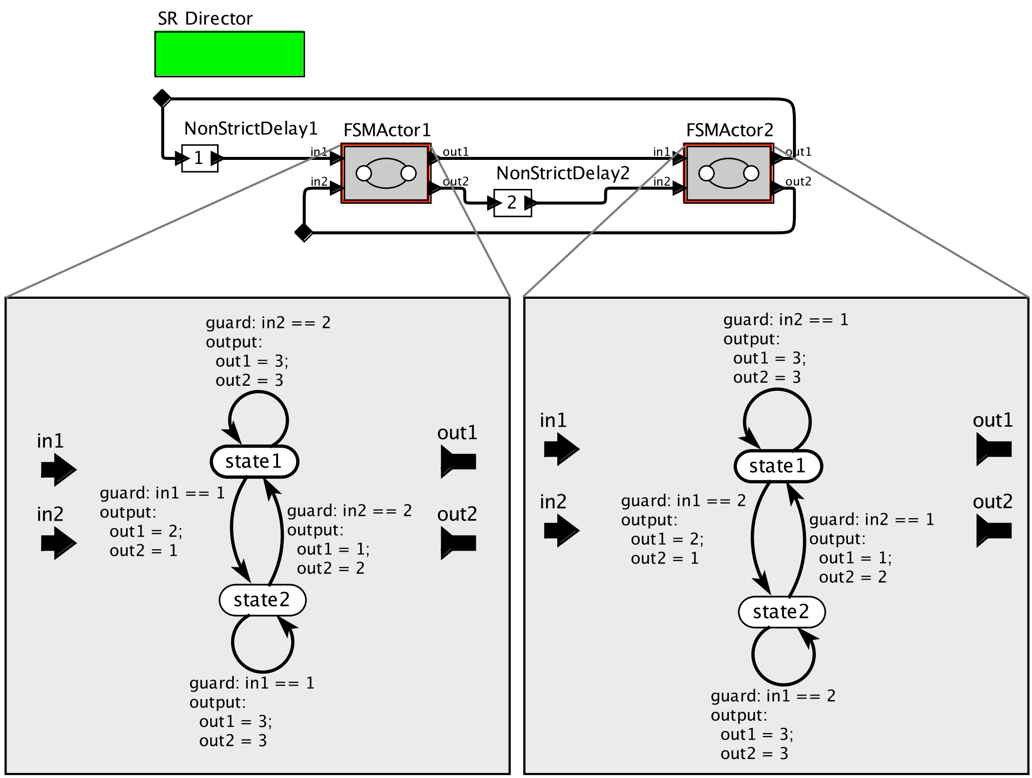


Figure 6.24: A model that requires separation of actions between the `fire` method and the `postfire` method in order to be able to converge to a fixed point. [online]

It is easy to verify that at each tick of the clock, both inputs of each state machine have the same value, so no state ever has more than one enabled outgoing transition. Determinism is preserved. Moreover, the values of these inputs alternate between 1 and 2 in subsequent ticks. For FSMActor1, the inputs are 1, 2, 1, $\cdots$ in ticks 1, 2, 3, $\cdots$. For FSMActor2, the inputs are 2, 1, 2, $\cdots$ in ticks 1, 2, 3, $\cdots$.

To understand a fixed-point iteration, it is helpful to examine more closely the four steps of execution of the `fire` method explained in Section 6.2 above.

1. *reads inputs*: Some inputs may not be known. Unknown inputs cannot be read, so the actor simply doesn't read them.
2. *evaluates guards on outgoing transitions of the current state*: Some of these guards may depend on unknown inputs. These guards may or may not be able to be evaluated. For example, if the guard expression is "`true || in1`" then it can be evaluated whether the input *in1* is known or not. If a guard cannot be evaluated, then it is not evaluated.
3. *chooses a transition whose guard evaluates to true*: If exactly one transition has a guard that evaluates to true, then that transition is chosen. If a transition has already been chosen in a previous invocation of the `fire` method in the same iteration, then the actor checks that the *same* transition is chosen this time. If not, it issues an exception and execution is halted. The FSM is not permitted to change its mind about which transition to take partway through an iteration. If more than one transition has a guard that evaluates to true, then the actor checks that every such transition is identified as a nondeterministic transition. If any such transition is not marked as nondeterministic, then the actor issues an exception. If all such transitions are marked nondeterministic, then it chooses one of the transitions. Subsequent invocations of the `fire` method in the same iteration will choose the same transition.
4. *executes the output actions on the chosen transition, if any*: If a transition is chosen, then the output values can all be defined. Some of these may be specified on the transition itself. If they are not specified, then they are asserted to be `absent` at this tick. If all transitions are disabled (all guards evaluate to false), then all outputs are set to `absent`. If no transition is chosen but at least one transition remains whose guard cannot be evaluated, then the outputs remain unknown.

In all of the above, choosing a transition may actually amount to choosing a chain of transitions, if there are immediate transitions enabled.

As described earlier, in the postfire() method, the actor executes the set actions of the chosen transition and changes the current state to the destination of the chosen transition. These actions are performed exactly once after the fixed-point iteration has determined all signal values. If any signal values remain undefined at the end of the iteration, the model is considered defective, and an error message will be issued.

Nondeterministic FSMs that are executed in a domain that performs fixed-point iteration involve additional subtleties. It is possible to construct a model for which there is a fixed point that has two enabled transitions but where the selection between transitions is not actually random. It could be that only one of the transitions is ever chosen. This occurs when there are multiple invocations of the `fire` method in the fixed-point iteration, and in the first of these invocations, one of the guards cannot be evaluated because it has a dependence on an input that is not known. If the other guard can be evaluated in the first invocation of `fire`, then the other transition will always be chosen. As a consequence, for nondeterministic state machines, the behavior may depend on the order of firings in a fixed-point iteration.

Note that default transitions may also be marked nondeterministic. However, a default transition will not be chosen unless all non-default transitions have guards that evaluate to false. In particular, it will not be chosen if any non-default transition has a guard that cannot yet be evaluated because of unknown inputs. If all non-default transitions have guards that evaluate to false and there are multiple nondeterministic default transitions, then one is chosen at random.

6.5 Summary

This chapter has introduced the use of finite-state machines in Ptolemy II to define actor behavior. Finite-state machines can be constructed using the FSMActor or ModalModel actors, where the latter supports hierarchical refinement of states in the FSM and the former does not. A number of syntactic devices are provided to make FSM descriptions more compact. These include the ability to manipulate variables (extended state machines), default transitions, immediate transitions, preemptive transitions, and hierarchical state machines, to name a few. Transitions have output actions, which are executed in the `fire` method when a transition is chosen, and set actions, which are executed in the `postfire` method and are used to change the value of variables. This chapter also briefly introduces concurrent composition of state machines, but that subject is studied in much more depth in Chapter 8, which shows how state refinements can themselves be concurrent Ptolemy II models in another domain.

Exercises

1. Consider a variant of the thermostat of example 6.1. In this variant, there is only one temperature threshold, and to avoid chattering the thermostat simply leaves the heat on or off for at least a fixed amount of time. In the initial state, if the temperature is less than or equal to 20 degrees Celsius, it turns the heater on, and leaves it on for at least 30 seconds. If the temperature is greater than 20 degrees, it turns the heater off and leaves it off for at least 30 seconds. In both cases, once the 30 seconds have elapsed, it returns to the initial state.

   (a) Create a Ptolemy II model of this thermostat. You may use an SDF director and assume that it runs at a rate of one iteration per second.

   (b) How many possible states does your thermostat have? (**Careful!** The number of states should include the number of possible valuations of any local variables.)

   (c) The thermostat in example 6.1 exhibits a particular form of state-dependent behavior called **hysteresis**. A system with hysteresis has the property that the absolute time scale is irrelevant. Suppose the input is a function of time, $x\colon \mathbb{R} \to \mathbb{R}$ (for the thermostat, $x(t)$ is the temperature at time t). Suppose that input x causes output $y\colon \mathbb{R} \to \mathbb{R}$, also a function of time. E.g., in Figure 6.8, x is upper signal and y is the lower one. For this system, if instead of x is the input is x' given by
   $$x'(t) = x(\alpha \cdot t)$$
   for a non-negative constant α, then the output is y' given by
   $$y'(t) = y(\alpha \cdot t) \ .$$
   Scaling the time axis at the input results in scaling the time axis at the output, so the absolute time scale is irrelevant. Does your new thermostat model have this property?

2. Exercise 1 of Chapter 5 asks for a model that recognizes the difference between single and double mouse clicks. Specifically, the actor should have an input port named *click*, and two output ports, *singleClick* and *doubleClick*. When a `true` input at *click* is followed by N *absent*s, the actor should produce output `true` on *singleClick*, where N is a parameter of the actor. If instead a second `true` input occurs within N ticks of the first, then the actor should output a `true` on *doubleClick*.

(a) Create an implementation of this actor using an extended state machine.

(b) How does your model behave if given three values *true* within N ticks on input port *click*, followed by at least N *absent* ticks.

(c) Discuss the feasibility and attractiveness of implementing this as a simple FSM, with no use the arithmetic variables of extended state machines.

3. A common scenario in embedded systems is where a component A in the system monitors the health of another component B and raises an alarm. Assume B provides sensor data as timed events. Component A will use a local clock to provide a regular stream of local timed events. If component B fails to send sensor data to component A at least once in each clock interval, then something may be wrong.

   (a) Design an FSM called MissDetector with two input ports, *sensor* and *clock*, and two output ports *missed* and *ok*. Your FSM should produce an event on *missed* when two *clock* events arrive without an intervening *sensor* event. It should produce an *ok* event when the first *sensor* event after (or at the same time that) a *clock* event arrives.

   (b) Design a second FSM called StatusClassifier that takes inputs from your first FSM and decides whether component B is operating normally. Specifically, it should enter a *warning* state if it receives *warningThreshold missed* events without an intervening *ok* event, where *warningThreshold* is a parameter. Moreover, once it enters a *warning* state, it should remain in that state until at least *normalThreshold ok* events arrive without another intervening *ok*, where *normalThreshold* is another parameter.

   (c) Comment about the precision and clarity of the English-language specification of the behavior in this problem, compared to your state machine implementation. In particular, find at least one ambiguity in the above specification and explain how your model interprets it.

4. Figures 6.18, 6.19, and 6.23 show the ABRO example implemented as a finite state machine, discussed in Example 6.10. In these realizations, in an iteration where the reset input *R* arrives, the output *O* will not be produced, even if in the same iteration *A* and *B* arrive.

   Make a variant of each of these that performs weak preemption upon arrival of *R*. That is, *R* prevents the output *O* from occurring only if it arrives strictly before both *A* and *B* have arrived. Specifically:

(a) Create a weak preemption ABRO that like Figure 6.18, uses only ordinary transitions and has no hierarchy.

(b) Create a weak preemption ABRO that like Figure 6.19, uses any type of transition, but has no hierarchy.

(c) Create a weak preemption ABRO that like Figure 6.23, uses any type of transition and hierarchy.

5. Figures 6.19 and 6.23 show the ABRO example implemented as a flat and a hierarchical state machine, respectively. Construct corresponding flat and hierarchical ABCRO models, which wait for three inputs, *A*, *B*, and *C*. If you had to wait for, say, 10 inputs, would you prefer to construct the flat or the hierarchical model? Why?

6. André (1996) points out that termination transitions are not necessary, as local signals can be used instead. Construct a hierarchical version of ABRO like that in Example 6.12 but without termination transitions.

7. The hierarchical FSM of Example 6.11 uses reset transitions, which initialize each destination state refinement when it is entered. It also uses preemptive transitions, which prevent firing of the refinement when taken. If these transitions were not reset or preemptive transitions, then the flattened equivalent machine of Figure 6.22 would be much more complex.

   (a) Construct a flat FSM equivalent to the hierarchical one in Figure 6.21, except that the transitions from *normal* to *faulty* and back are not preemptive.

   (b) Construct a flat FSM equivalent to the hierarchical one in Figure 6.21, except that the transitions from *normal* to *faulty* and back are preemptive, as in Figure 6.21, but are also history transitions instead of reset transitions.

   (c) Construct a flat FSM equivalent to the hierarchical one in Figure 6.21, except that the transitions from *normal* to *error* and back are nonpreemptive history transitions.

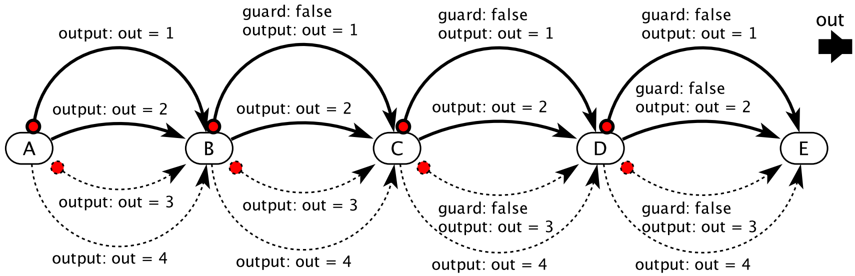
8. Consider the compact implementation of the ABRO state in Figure 6.19.

   (a) Is it possible to do a similarly compact model that does not use nondeterminism?

   (b) Can a similarly compact variant of ABCRO be achieved without nondeterminism?

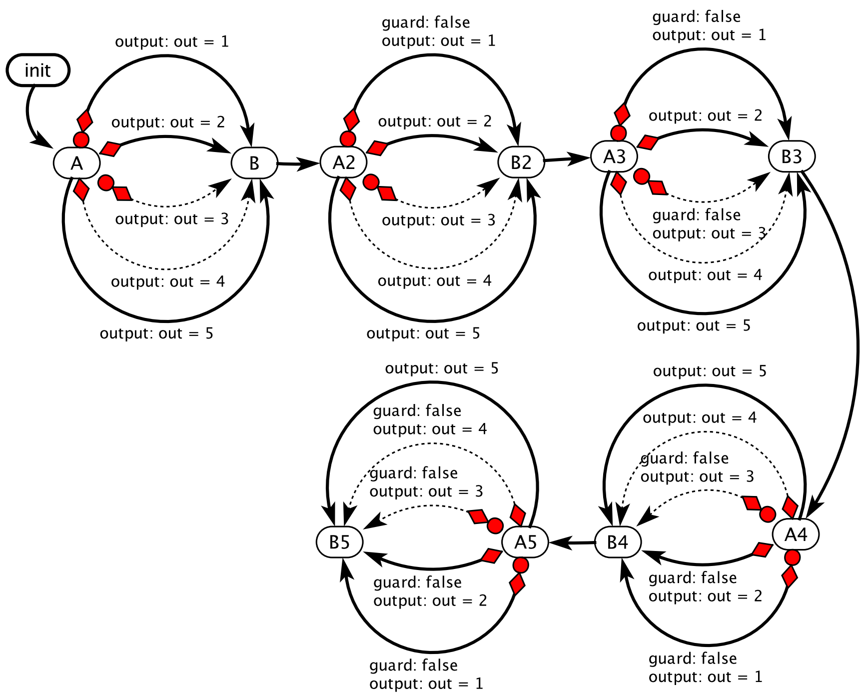
9. This exercise studies relative priorities of transitions.

   (a) Consider the following state machine:



   Determine the output from the first six reactions.

   (b) Consider the following state machine:



   Determine the output from the first six reactions.

7

Discrete-Event Models

Edward A. Lee, Jie Liu, Lukito Muliadi, and Haiyang Zheng

Contents

The **discrete-event** (**DE**) domain is used to model timed, discrete interactions between concurrent actors. Each interaction is called an **event**, and is conceptually understood to be an instantaneous message sent from one actor to another. An event in Ptolemy II is a token (encapsulating the message) arriving at a port at a particular model time. The key idea in DE is that each actor reacts to input events in temporal order. That is, every time an actor fires, it will react to input events that occur later than events from previous firings. Because this domain relies on temporal sequencing, its model of time (discussed later in this chapter) is essential to its operation.

Example 7.1: An example of a discrete-event model is shown in Figure 7.1. This example illustrates a common application of DE, modeling faults or error conditions that occur at random times. This example models what is called a **stuck at fault**, where at a random time, a signal become stuck at a fixed value and no longer varies with the input. This example illustrates why it is important that events be processed in time-stamp order.

Specifically, the StuckAtFault actor is a state machine (see Chapter 6) with two states, *normal* and *faulty*. When it is in the *normal* state, then when an input arrives on the *in* port, the value of the input is copied to the *out* port and stored in the *previousIn* parameter. When an *error* event arrives, the state machine switches to the *faulty* state, and henceforth produces a constant output.

In the plot in Figure 7.2, we see that the model switches to the faulty state between times 7 and 8. The events at the *in* port are triggered in this model by the DiscreteClock actor, which produces events that are regularly spaced in time, while the PoissonClock, which triggers the error condition, produces events that are irregularly spaced in time (representing the occurrence of an error). (See the sidebar on page 241 for a description of these clock actors.) These actors are common in discrete-event models, where typically the only importance of their output is the time at which the output is produced. The value of the output does not matter much.

The *meanTime* parameter of the PoissonClock actor, which specifies the expected time between events, is set to 10.0 in the model, so the actual time of the error in

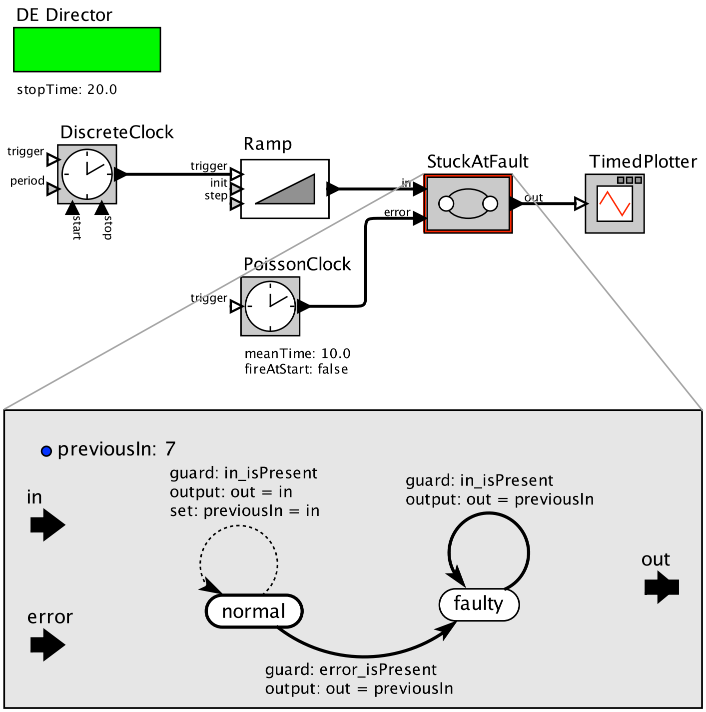


Figure 7.1: Simple example of a DE model. [online]

this run, approximately 7.48, is reasonably close to the expected time. Note that as with all random actors in Ptolemy II (see sidebar on page 252 for more such actors), you can control the *seed* of the random number generator in order to get reproducible simulations.

In general, for any actor that changes state in reaction to input events, as the StuckAtFault actor does, it is important to react to events in temporal order. The behavior of the actor depends on its state. In this case, once it has made the transition to the *faulty* state, its input-output behavior is very different. All subsequent input values will be ignored.

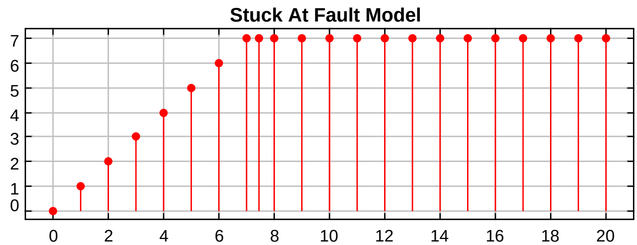


Figure 7.2: Sample output from the model in Figure 7.1.

In this chapter, we examine the mechanics and subtleties of DE models. We begin with a discussion of the model of time, and then show how DE provides a determinate MoC. We discuss the subtleties that arise when events occur simultaneously and when models include feedback loops.

7.1 Model of Time in DE Domain

In DE models, connections between actors carry signals that consist of events placed on a time line. Each event has both a value and a time stamp, where the time stamp defines a global ordering between events. This is different from dataflow, where a signal consists of a sequence of tokens, and there is no time significance in the signal. A time stamp is a superdense time value, consisting of a model time and a microstep.*

7.1.1 Model Time vs. Real Time

A DE model executes chronologically, processing the oldest events (those with earlier time stamps) first. Time advances as events are processed, both in the model and in the outside world. There is potential confusion, therefore, between model time, which

*Do not confuse "model time" with "model of time." In Ptolemy nomenclature, "model time" is a time value (e.g. 10:15 AM), whereas the "model of time" encompasses the overall approach to handling time sequencing.

is the time that evolves in the model, and **real time**, which is the time that elapses in the real world while the model executes (also called **wall-clock time**). Model time may advance more rapidly or more slowly than real time. The DE director has a parameter, *synchronizeToRealTime*, that, when set to `true`, synchronizes the two notions of time, to the extent possible. It does this by delaying execution of the model (when necessary) to allow real time to "catch up" with model time. Of course, this only works if the computer executing the model is fast enough that model time is advancing more rapidly than real time. When this parameter is set to `true`, model-time values are interpreted as being in units of seconds, but otherwise the units are arbitrary.

Example 7.2: Consider the DE model shown in Figure 7.3. This model includes a PoissonClock actor, a CurrentTime actor, and a WallClockTime actor (see sidebars on pages 241 and 242). The plot shows that wall-clock time and model time are indeed quite different; in this case, wall-clock time barely advances during execution, whereas model time advances quite far (to about 9 seconds). Since the horizontal axis in this plot is model time, the model time plot increases linearly. If you set the

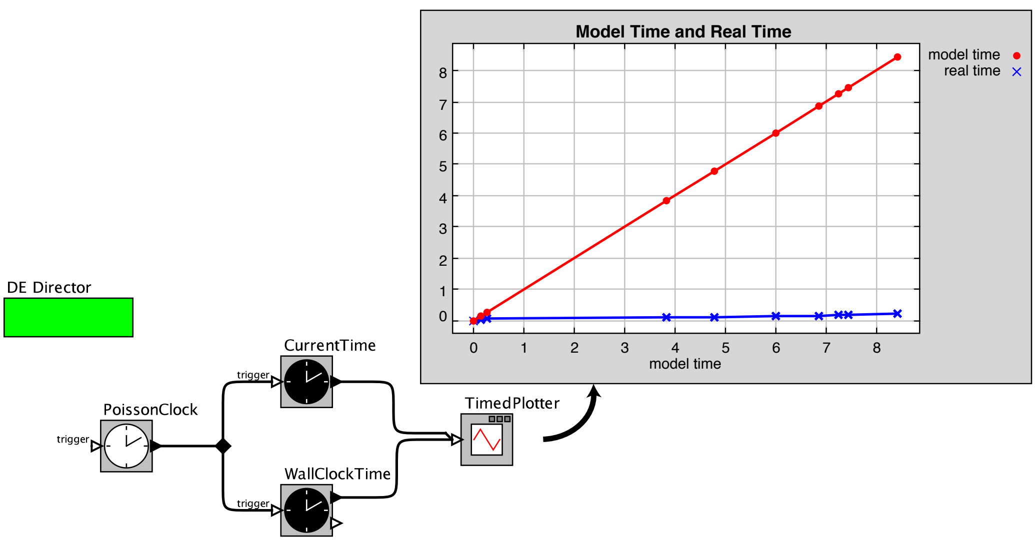


Figure 7.3: Model time vs. real time (wall clock time). [online]

synchronizeToRealTime parameter of the director to `true`, then you will find that the two plots coincide almost perfectly.

The ability to synchronize model time and real time is useful when you want a model to accurately display to a user the timing of events in the model. For example, a model that generates sounds according to some rhythm will not be very satisfying if it just executes as fast as possible.

7.1.2 Simultaneous Events

A question that arises in the DE domain is how simultaneous events are handled. As previously described, strongly simultaneous events have the same time stamp (model time and microstep), whereas weakly simultaneous events have the same model time, but not necessarily the same microstep. We have stated that events are processed in chronological order, but if two events have the same time stamp, which one should be processed first?

Example 7.3: Consider the model shown in Figure 7.4, which produces a histogram of the interarrival times of events from a PoissonClock actor. This model calculates the difference between the current event time and the previous event time, resulting in the plot that is shown in the figure. The Previous actor is a **zero-delay actor**, meaning that it produces an output with the same time stamp as the input (except on the first firing, where in this case it produces no output–see sidebar on 243). Thus, when the PoissonClock actor produces an output, there will be two (strongly) simultaneous events, one at the input to the *plus* port of the AddSubtract actor, and one at the input of the Previous actor.

At this point, should the director fire the AddSubtract actor or the Previous actor first? Either approach appears to respect the chronological order of the events, but intuitively we might expect that the Previous actor should be fired first. And indeed, in this example, the director will fire the Previous actor first, for reasons described below.

To ensure determinism, the order in which actors are fired must be well defined. The order is governed by a **topological sort** of the actors in the model, which is a list of the actors in

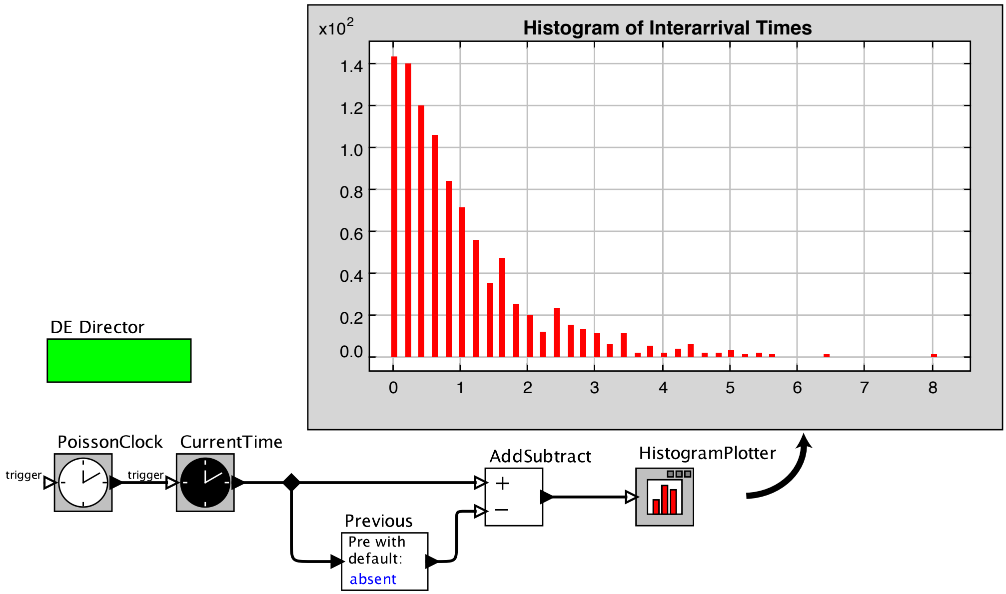


Figure 7.4: Histogram of interarrival times, illustrating handling of simultaneous events. There is a subtle bug in this model, corrected in Figures 7.5 and 7.6. [online]

data-precedence order. For a given model, there may be multiple valid topological sorts, but all adhere to the rule that any actor A that sends events to another actor B will always occur earlier in the topological sort. Thus, the DE director, by analyzing the structure of the model, delivers determinate behavior, where actors that produce data are fired before actors that consume their results, even in the presence of simultaneous events. All valid sorts yield the same events.

In the example above, it is helpful to know how the AddSubtract actor works. When it fires, it will add all (strongly simultaneous) available tokens on the *plus* port, and subtract all (strongly simultaneous) available tokens on the *minus* port. If there is a token at the *plus* port, but none at the *minus* port, then the output will equal the token at the *plus* port. Conversely, if there is a token at the *minus* port, but none at the *plus* port, then the output will equal the *negative* of the token at the *minus* port.

Based on this behavior, there is only one valid topological sort here: PoissonClock, CurrentTime, Previous, AddSubtract, and HistogramPlotter. In this list, AddSubtract appears after Previous, because Previous sends its event to AddSubtract. Therefore, given strongly simultaneous events at the inputs of AddSubtract and Previous as described in the example above, the director will always fire the Previous actor first.

7.1.3 Synchronizing Events

Although the example given in Figure 7.4 provides a good overview of how actor firings are sequenced, it has a subtle problem. Each output of the AddSubtract actor is supposed to be the time between arrivals of two successive events from the PoissonClock actor (the **interarrival time**). However, the very first event produced by the CurrentTime actor has a value equal to the time stamp of the first event produced by the PoissonClock (which in this case is 0.0 because the PoissonClock by default produces an initial event when it begins execution). The Previous actor, however, will not produce any output at this time, because (by default) its output is absent upon arrival of its first input (see sidebar on page 243). Hence, the first output of the AddSubtract will have value 0.0, which is not, in fact, an interarrival time! Thus, the plotted histogram includes a spurious value 0.0.

To fix this problem, we would like to ensure that the AddSubtract actor receives exactly one event on each input port, and only receives events when there is one available for each port. We can accomplish this using the Sampler actor, as shown in Figure 7.5 (see sidebar on page 244). This actor produces an output event only if there is an input event

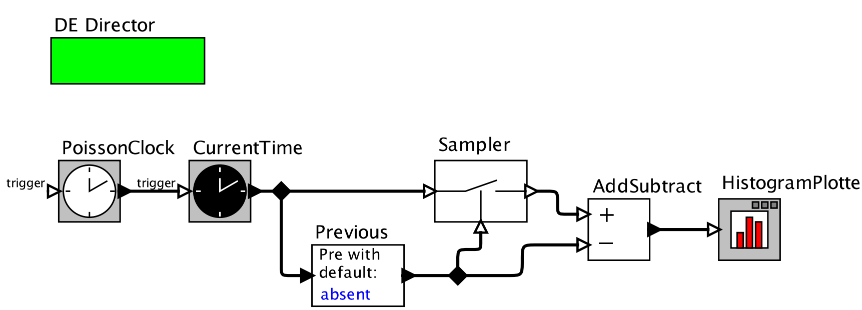


Figure 7.5: Histogram of interarrival times, correcting the subtle bug in Figure 7.4 using the Sampler actor. [online]

on its *trigger* input (the bottom port). Upon receiving a trigger input, it passes to its output whatever (strongly simultaneous) events are available on its input port. So in this example, the first output from the CurrentTime actor will be discarded, because, at that time, there is no event on the *trigger* input.

There are a number of other ways to accomplish the same goal. For example, the Synchronizer actor described in the sidebar on page 244 can also be used in place of the Sampler. More directly, the TimeGap actor can also be used to solve this problem (see sidebar on page 242), as shown in Figure 7.6. It combines the functionality of the Previous, Sampler, and AddSubtract actors.

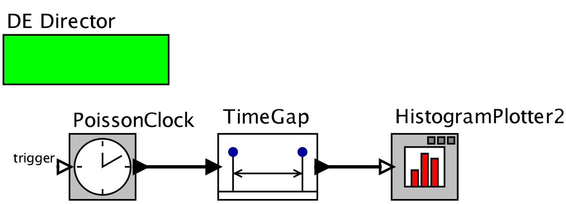
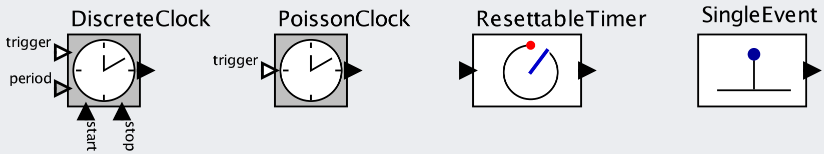


Figure 7.6: Histogram of interarrival times using the TimeGap actor. [online]

Sidebar: Clock Actors

Clock actors generate timed events. Four such actors are shown below:

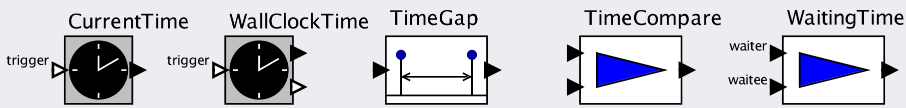


These actors are found in the `Sources→TimedSources` library, except for Resettable-Timer, which is in `DomainSpecific→DiscreteEvent`.

- **DiscreteClock** is a versatile event generator. Its simplest use is to generate a sequence of events that are regularly spaced in time. Its default parameters cause the actor to produce tokens with value 1 (an *int*) spaced one time unit apart, starting at execution start time (typically time zero). This default setting is used to produce the plot in Figure 7.2. But DiscreteClock can generate much more complex event patterns that cycle through a finite sequence of values and have periodically repeating time offsets from the start of each period. It can also generate a one-time, finite sequence of events (instead of a periodic pattern) by setting the *period* to `Infinity`. These events can be arbitrarily placed in time. See the actor documentation for details.
- **PoissonClock**, in contrast, produces events at random times. The time between events is given by independent and identically distributed (**IID**) exponential random variables. An output signal with these characteristics is called a **Poisson process**. Like the DiscreteClock actor, the PoissonClock actor can cycle through a finite sequence of values, or it can produce events with the same value each time.
- **ResettableTimer** produces an output event after the time specified by the input event has elapsed (in model time, not real time). That is, an output event is produced at the model time of the input plus the value of the input. If the input value is zero, then the output will be produced at the next microstep. This actor allows a pending event to be canceled, and also allows a new input event to preempt a previously scheduled output. See the actor documentation for details.
- **SingleEvent** is not really needed, since DiscreteClock is capable of producing single events. Nonetheless, this actor is sometimes useful because it visually emphasizes in a model that only a single event is produced.

Sidebar: Time Measurement

The following actors provide access to the model time of events:



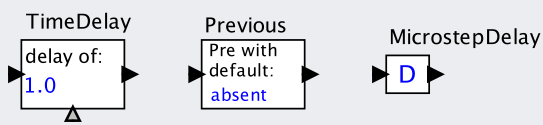
The **CurrentTime** actor, which is found in `Sources→TimedSources`, observes the model time of events. It produces an output event whenever an input event arrives, where the value of the output event is the model time of the input event. (The time value on the output of the CurrentTime actor is represented as a *double*, so it may not be an exact representation of the internal model time, which, as explained in Section 1.7.3, has a fixed resolution that does not change as the value of time increases. The resolution of a *double*, by contrast, decreases as the magnitude increases.) The output event has the same time stamp as the input event, so the actor's reaction is conceptually instantaneous. (**CurrentMicrostep**, found in same library and not shown above, is useful primarily for debugging.)

The **WallClockTime** actor, which is found in `RealTime`, observes the real time of events. When it fires in reaction to a trigger event, it outputs a *double* representing the amount of real time that has passed (in seconds) since the actor was initialized. Since this value depends on arbitrary scheduling decisions that the DE director makes, it is nondeterminate. Nonetheless, it can be useful for measuring performance, for example by firing it at the beginning and ending of a part of the model's execution. The inputs that arrive on the *trigger* input also pass through to an output port, a feature that makes it somewhat easier to control the scheduling of downstream actors in some domains.

The `DomainSpecific→DiscreteEvent` library has the other time-related actors, which are used to measure model-time differences between events. **TimeGap** measures the difference between successive events in a signal. **TimeCompare** measures the time gap between events in two signals. It is triggered by the arrival of an event at either input, and outputs the difference between the model time of this event and that of the last event on the other port. **WaitingTime** also measures the gap between two signals, but in a somewhat different manner. When an event arrives on the *waiter* port, the actor begins waiting for an event to arrive on the *waitee* port. When the first such event arrives, the actor outputs the model time difference.

Sidebar: Time Delays

The following actors provide mechanisms for delaying events by manipulating their time stamps:



The most useful of these is **TimeDelay**, which increments the model time of the input event by a specified amount. The amount of the increment is displayed in the icon and defaults to 1.0. This actor delays an event (in model time, not in real time). Only the model time is incremented; the microstep of the output is the same as the microstep of the input.

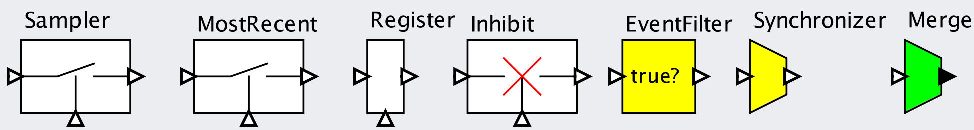
The amount of the delay can optionally be given on the bottom input port. If an event is provided on the bottom input port, then the value of that event specifies the delay for input events that arrive at the same time or later (i.e., that have a time stamp greater than or equal to that of the event arriving on the bottom input port). The TimeDelay actor can be found in `DomainSpecific→DiscreteEvent`.

Occasionally, it is useful to set a model-time delay to zero. In this case, the model time of the output event is the same as that of the input, but its microstep is incremented. This setting can be useful for feedback loops, as discussed in the main text. A TimeDelay with delay equal to 0.0 is equivalent to the **MicrostepDelay** actor.

The **Previous** actor, upon receiving an input event, produces an output event with the same time stamp as the input event, but with the value of the previously received input event. When it receives the first event (i.e., there has been no previous event) it outputs a default value if one is given, or it outputs nothing (absent output). This actor, therefore, delays each input event until the arrival of the next input event.

Sidebar: Samplers and Synchronizers

The following actors provide mechanisms for synchronizing events:



The Sampler and Synchronizer are in `FlowControl→Aggregators`, whereas the rest are in `DomainSpecific→DiscreteEvent`.

- **Sampler** copies selected events from its input port (a multiport) to its output port when it receives an event on the *trigger* port (at the bottom of the icon). Only those input events that are strongly simultaneous with the trigger are copied; these events are sent to the corresponding output channel.
- **MostRecent** is similar to Sampler, except that, upon receiving a trigger, it copies the *most recent* input event (which may or may not be simultaneous with the trigger event) to the corresponding output channel. It provides an optional *initialValue* parameter, which specifies the output value if no input has arrived upon triggering.
- **Register** is similar to MostRecent, except that upon receiving a trigger, it copies the most recent *strictly earlier* input event to the corresponding output channel. This actor, unlike Sampler or MostRecent, always introduces delay, and hence is also similar to the delay actors described in the sidebar on page 243. The delay can be as small as one microstep, in which case the input and output will be weakly simultaneous. Because it introduces delay, this actor is useful in feedback loops (see Section 7.3.2).
- **Inhibit** is the converse of the Sampler. It copies all inputs to the output *unless* it receives a *trigger* input.
- **EventFilter** accepts only *boolean* inputs, and copies only true-valued inputs to the output.
- **Synchronizer** copies inputs to outputs only if every input channel has a strongly simultaneous event.
- **Merge** merges events on any number of input channels into a single signal in timestamp order. If it receives strongly simultaneous events, then it either discards all but the first one (if the *discard* parameter is `true`), or it outputs the events with increasing microsteps (if the *discard* parameter is `false`).

Probing Further: DE Semantics

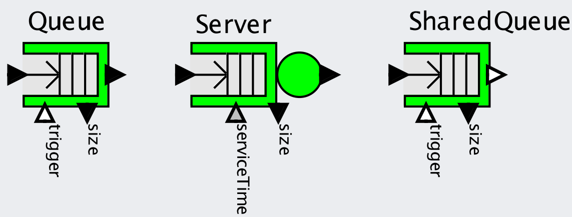
Discrete-event modeling based on time stamps has been around for a long time. One of the earliest complete discrete-event formalisms is called DEVS, for Discrete Event System Specification (Zeigler, 1976). Many subtly different variants have appeared over time (see for example Ramadge and Wonham (1989), Cassandras (1993), Baccelli et al. (1992), and Zeigler et al. (2000)). Variants of DE also form the foundation for the widely used hardware description languages VHDL, Verilog, and SystemC, and a number of network simulation tools such as OPNET Modeler (from OPNET Technologies, Inc.), ns-2 (a collaborative open-source effort led by the Virtual Internetwork Testbed Project VINT), and ns-3 (`http://www.nsnam.org/`). The SysML implementation in IBM Rational's **Rhapsody** tool is also a variant of a DE model. The variant of DE described in this chapter, particularly its use of superdense time, appears to be unique.

The formal semantics of DE is a fascinating and deep topic. The oldest approaches describe execution of DE models in terms of state machines (Zeigler, 1976). More recent approaches define a metric space (Bryant, 1985) in which actors become contraction maps and the meaning of a model becomes a fixed point of these maps (Reed and Roscoe, 1988; Yates, 1993; Lee, 1999; Liu et al., 2006). DE semantics have also been given as a generalization of the semantics of synchronous-reactive languages (Lee and Zheng, 2007), and as fixed points of monotonic functions over a complete partial order (CPO) (Broy, 1983; Liu and Lee, 2008). These latter approaches are denotational (Baier and Majster-Cederbaum, 1994), whereas the state machine approaches have a more operational flavor.

DE models can be large and complex, so execution performance is important. There has also been extensive work on simulation strategies for DE models. A particularly interesting challenge is exploiting parallel hardware. The strong ordering imposed by time stamps makes parallel execution difficult (Chandy and Misra, 1979; Misra, 1986; Jefferson, 1985; Fujimoto, 2000). A recently proposed strategy called **PTIDES** (for programming temporally integrated distributed embedded systems), leverages network time synchronization to provide efficient distributed execution (Zhao et al., 2007; Lee et al., 2009b; Eidson et al., 2012). In PTIDES, DE is used not only as a simulation technology, but also as an implementation technology. That is, the DE event queue and execution engine become part of the deployed embedded software.

Sidebar: Queue and Server Actors

The following actors provide queueing and are particularly useful for modeling behavior in communication networks, manufacturing systems, service systems, and many other queueing systems. These actors are found in the `DomainSpecific→DiscreteEvent` library.



- **Queue** takes a token received on its input port and stores it in the queue. When the *trigger* port receives an event, the oldest element in the queue is produced on the output. If there is no element in the queue when a token is received on the trigger port, then no output is produced. In this case, if the *persistentTrigger* parameter is true, then the next input that is received will be sent immediately to the output. A *capacity* parameter limits the capacity of the queue; inputs received when the queue is full are discarded. The *size* output produces the size of queue after each input is handled.
- **Server** models a server with a fixed or variable service time. A server is either busy (serving a customer) or not busy at any given time. If an input arrives when the server is not busy, then the input token is produced on the output with a delay given by the *serviceTime* parameter. If an input arrives while the server is busy, then that input is queued until the server becomes free, at which point it is produced on the output with an additional delay given by the *serviceTime* parameter. If several inputs arrive while the server is busy, then they are served on a first-come, first-served basis. The *serviceTime* may be provided on an input port rather than in the parameter. A *serviceTime* received on the input port applies to all events that arrive at the same or later times, until another *serviceTime* value is received. The *size* output produces the size of queue after each input is handled.
- **SharedQueue** is similar to Queue but supports multiple outputs, each of which draws tokens from the same queue.

7.2 Queueing Systems

A common use of DE is to model **queueing systems**, which are networks of queues and servers. These models are typically paired with models of random arrivals and service times. One of the most basic queueing systems is the **M/M/1 queue**, where events (such as customers) arrive according to a Poisson process and enter a queue. When they reach the head of the queue, they are served by a single server with a random service time. In an M/M/1 queue, the service time has an exponential distribution, as illustrated by the next example.

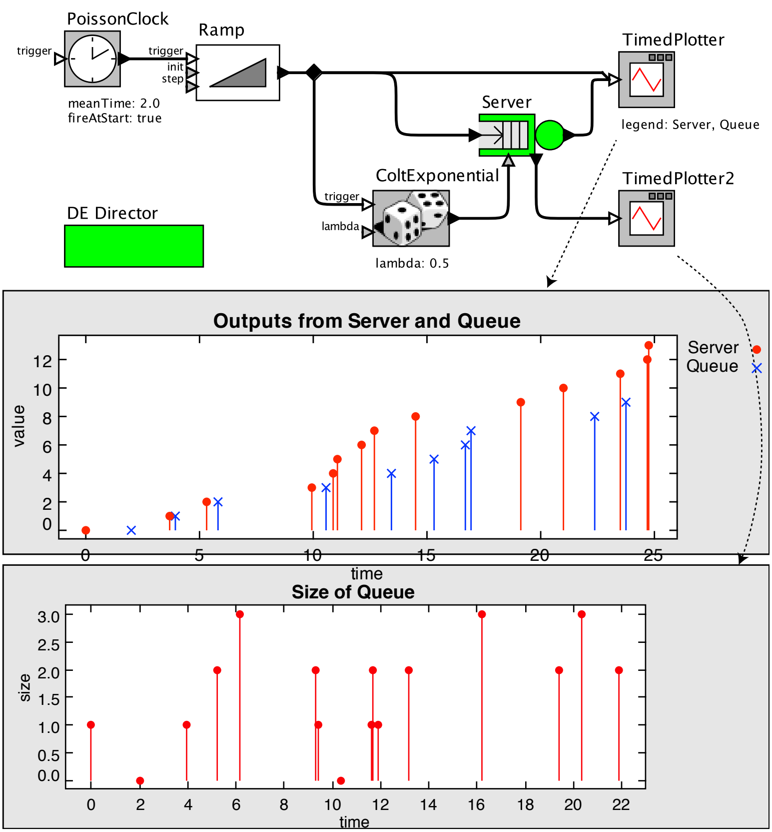


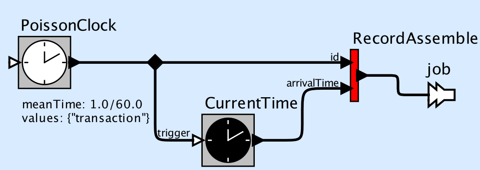
Figure 7.7: A model of an M/M/1 queue. [online]

Example 7.4: A Ptolemy II model of an M/M/1 queue is shown in Figure 7.7. The PoissonClock simulates the arrival of customers. In this case, the average interarrival time is set to 2.0. The Ramp actor is used to label the customers with distinct integers for identification. The **ColtExponential** shown in the figure is one of many random number generators in Ptolemy II (see box on page 252). For each customer arrival, it generates a new random number according to an exponential distribution. The *lambda* parameter of the ColtExponential actor is set to 0.5, which results in a mean value of 1/0.5 or 2.0. The Server actor is a queue with a server (see box on page 246). In this case, a new service time is specified for each customer that arrives. The Server outputs both the customer number, shown in the upper plot, and the size of the queue when a customer arrives or departs, shown in the lower plot. Note that three customers arrive in a burst around time 4, resulting in a buildup of the queue size at that time.

More interesting examples use networks of queues and servers.

Example 7.5: Figure 7.8 shows a model of a storage system where jobs arrive at random and are processed by a CPU (central processing unit). The CPU then writes to disk 1, performs additional processing, writes to disk 2, and finally performs a third round of processing. This particular model is given by Simitci (2003), who analyzes the model using queueing theory and predicts an average latency through the network of 0.057 seconds. The experimental result, shown in the **MonitorValue** actor, is very close to the predicted value for this particular Monte Carlo run.

To allow easy measurement of job latencies, incoming jobs are time stamped with the time of arrival. Specifically, the EventGenerator composite actor is implemented as follows:



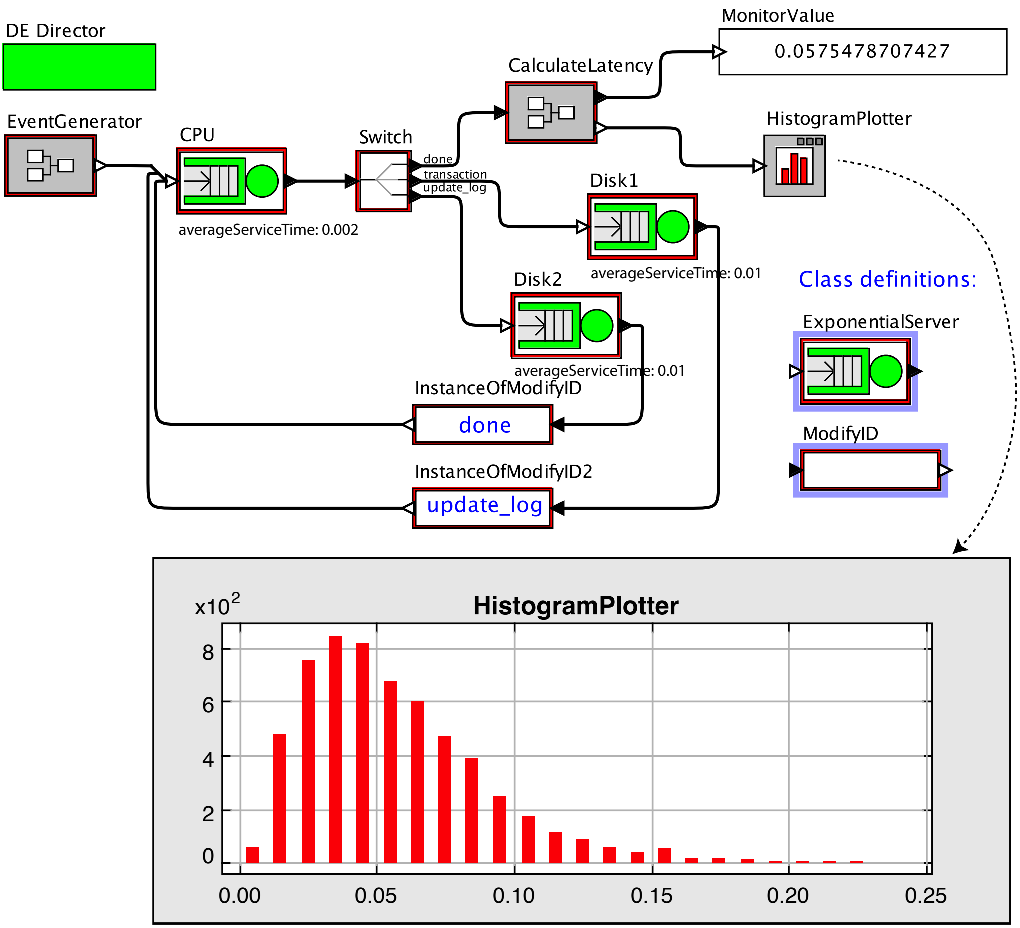
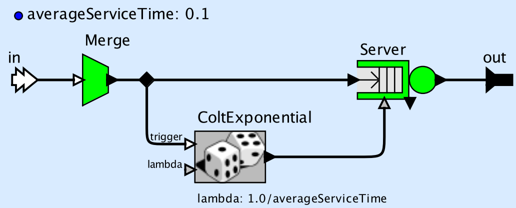


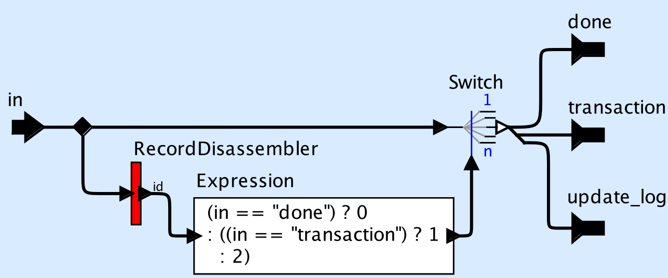
Figure 7.8: A queueing model of a transaction processor that writes to two disks. [online]

This composite actor produces events according to a Poisson process with average interarrival time of 1/60-th of a second. Each event is a record with two fields: an *id* and an *arrivalTime* (see box on page 253). The *arrivalTime* carries the time stamp of the event, which is provided by the CurrentTime actor. The *id* is a constant string “transaction” that will be used to route the event to the appropriate disk. Referring again to Figure 7.8, the CPU, Disk1, and Disk2 actors are also composite actors, each of which is an instance of the actor-oriented class **ExponentialServer**, defined as follows:



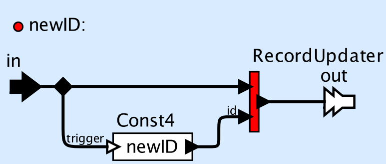
This composite actor has a single parameter, *averageServiceTime*. It implements a server with a random service time. Upon arrival of an event on any input channel, the ColtExponential actor generates a new random number from an exponential distribution. This random number specifies the service time that the newly arrived event will experience when it is served. If the queue is empty, it will be served immediately. Otherwise, it will be serviced when the server has served all events that precede it in the queue.

Referring again to Figure 7.8, the Switch is defined as follows:

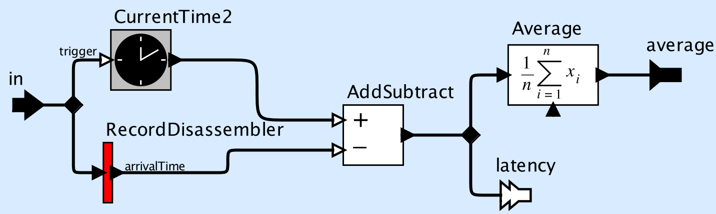


This actor extracts the *id* field from the incoming record, and, using an Expression and a Switch actor, routes the input event to one of three output ports depending on the *id*.

The two feedback paths contain instances of the actor-oriented class **ModifyID**, which is defined as follows:



This actor constructs a new record from the input record by replacing the *id* field of the record using a RecordUpdater (see box on page 253). The event with the new *id* is processed by the CPU again. When the *id* becomes "done," the job event is routed to the CalculateLatency composite actor, implemented as follows:



This submodel measures the overall latency of the job. It extracts the *arrivalTime* from the incoming record and subtracts it from the time stamp of the completed job. It then outputs both the calculated latency and a running average of the latency calculated with the Average actor. A HistogramPlotter actor is used to plot a histogram of the latencies, and the MonitorValue is used to display the running average in the model itself.

7.3 Scheduling

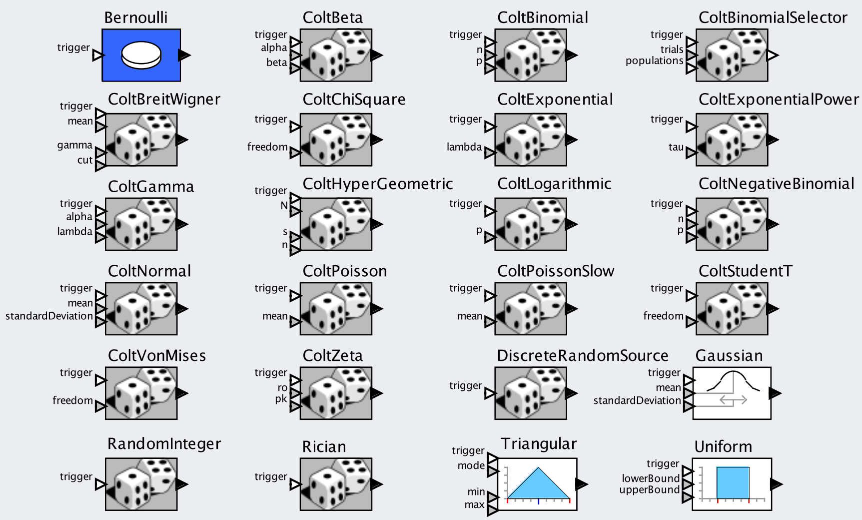
The main task of the DE director is to fire actors in time-stamp order. The DE director maintains an event queue that is sorted by time stamp, first by model time and then by microstep. The DE director controls the execution order by selecting the earliest event in the event queue and making its time stamp the **current model time**. It then determines the actor the event is destined for, and finds all other events in the event queue with the same time stamp that are destined for the same actor. It sends those events to the actor's input ports and then fires the actor. If after firing the actor it has not consumed all these events, then the director will fire the actor again until all input events have been consumed.[†]

[†]Note that if an actor is written incorrectly and does not consume input events, then this strategy results in an infinite loop, where the actor will be repeatedly fired forever. This reflects a bug in the actor itself. Actors are expected to read inputs, and in the DE domain, reading an input consumes it.

The DE director uses a specific rule to ensure determinism. Specifically, it guarantees that it has provided *all* events that have a time stamp equal to the current time stamp before it fires the actor. It cannot subsequently provide an additional event with the same time stamp. Such an additional event may possibly have the same model time, but in that case it does not have the same microstep.

Sidebar: Random Number Generators

Ptolemy II includes a number of **random number generators**, shown below:



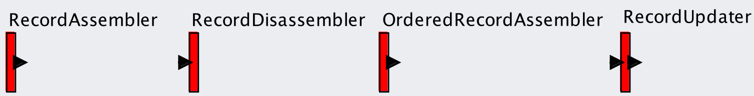
The actors all produce a random number on their output port on each firing. Most of them have parameters that can also be set on input ports, so that parameters can vary on each firing. See the documentation of each actor for details.

All of these actors provide a *seed* parameter, which can be used to ensure repeatable random sequences. When you set the seed parameter of one random actor, then you set it for all random actors (it is a **shared parameter**, meaning that that its value is shared by all actors in a model that have this parameter). All random actors also share a single underlying random number generator, so reproducible experiments are easy to control.

The actors named with the prefix **Colt** were created by David Bauer and Kostas Oikonomou, with contributions from others, using an open source library called Colt, originally developed by CERN (the European Organization for Nuclear Research).

Sidebar: Record Actors

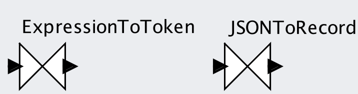
The record type in Ptolemy II is similar to a `struct` in C. It can contain any number of named fields, each of which can have an arbitrary data type (it can even be a record). There are several actors that manipulate records, found in [`Actors→FlowControl→Aggregators`] and shown below.



All of these actors require the addition of new ports using the [`Configure→Ports`] menu item on the actor. We recommend making the field name visible using the Show Name column; see Figure 2.18.

- **RecordAssembler** outputs a record that contains one field for each input port, where the name of the field is the same as the name of the input port.
- **RecordDisassembler** extracts fields from a record. The name of the output port must match the name of the field; the type system (see Chapter 14) will report a type error if the input record does not have a field that matches the output port name.
- **OrderedRecordAssembler** constructs a record token in which the order of the fields in the record matches the order of the input ports (top to bottom). This actor is probably not useful unless you are writing Java code that iterates over the fields of the record and depends on the order.
- **RecordUpdater** adds or modifies fields of a record. The icon's built-in input port provides the original record. Additional input ports must be added and named with the field name you wish to add or modify. The output record will be a modified record.

There are two other actors that are useful for constructing records; these are found in [`Actors→Conversions`] and shown below:



Both of these actors accept string inputs. **ExpressionToToken** parses the input string, which can be any string accepted by the expression language described in Chapter 13, including records. If the input string specifies a record, then the output token will be a record token. **JSONToRecord** accepts any string in the widely used Internet JSON format and produces a record.

As discussed earlier, the DE director performs a topological sort of the actors. Once this sort has been performed, each actor can be assigned a **level**. The level is largest number of upstream actors along a path from either a source actor (which has no upstream actors) or a delay actor.

Example 7.6: For example, the actors in Figure 7.5 have the following levels:

- PoissonClock: 0
- CurrentTime: 1
- Previous: 2
- Sampler: 3
- AddSubtract: 4
- HistogramPlotter: 5

When two events with the same time stamp are inserted into the event queue, the event that is destined for the actor with the lower level will appear earlier in the queue. This ensures, for example, that in Figure 7.5, Previous will fire before Sampler, and Sampler will fire before AddSubtract. Thus, when AddSubtract fires, it is assured of seeing all input events at the current model time.

7.3.1 Priorities

The DE director sorts events by model time, then by microstep, and then by level. But it is still possible to have events that have the same time stamp and level.

Example 7.7: Consider the model shown in Figure 7.9. Here, the two Ramp actors have the same level (1) and the two FileWriter actors have the same level (2). When the clock produces an event, there will be two events in the event queue with the same time stamp and level, one destined for Ramp and one destined for Ramp2. Since these two actors do not communicate, the order in which they are fired does not matter. However, the two FileWriter actors might be writing to the same file (or to standard out). In this case, the firing order does matter, but the order cannot be determined by the time stamp and level alone. We may not want the director to arbitrarily choose an order. In this case, the designer may choose to use priority parameters, as described below.

The previous example illustrates an unusual scenario: two actors can affect each other even though there is no direct communication between them in the model. They are interacting **under the table**, in a manner that is invisible to the DE director. When there is such interaction, the model builder may wish to exercise some control over the order of execution.

The `Utilities→Parameters` library contains a *Priority* parameter that can be dragged and dropped onto actors. The value can be set for each actor independently by double clicking on the actor. A lower value is interpreted as a higher priority. When events have the same time stamp and the same level, the DE director consults the priority of the destination actors, and places events destined to actors whose priorities are higher (*Priority* has a lower value) earlier in the event queue.

Example 7.8: For the example in Figure 7.9, Ramp2 and FileWriter2 have *Priority* zero, so they will fire before Ramp and FileWriter, which both have *Priority* one. Without the use of priorities, the order would be nondeterminate.

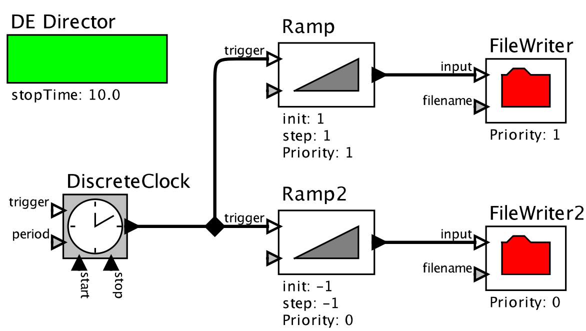


Figure 7.9: Although it is rarely necessary, it may sometimes be useful to set the priorities of actors in DE. This only has an effect when the firing order is otherwise not determined by time stamps or by data precedences (levels). [online]

In DE modeling it is rare to need priorities; because their value has global meaning (i.e., the priorities are honored throughout the model), this mechanism is not very modular.

7.3.2 Feedback Loops

If the model has a directed loop (called a **feedback** loop), then a topological sort is not possible. In the DE domain, every feedback loop is required to have at least one actor that introduces a time delay, such as TimeDelay, Register, or queues or servers (see sidebars on pages 243, 244, and 246).

Example 7.9: Consider the model shown in Figure 7.10. That model has a DiscreteClock actor that produces events every 1.0 time units. Those events trigger the Ramp actor, which produces outputs that start at 0 and increase by 1 on each firing. In this model, the output of the Ramp goes into an AddSubtract actor, which subtracts from the Ramp output its own prior output delayed by one time unit. The result is shown in the plot in the figure.

To ensure that models with feedback are determinate, the DE director assigns delay actors a level of zero, but delays the time at which they read their inputs until the postfire stage, which occurs after firing any other actors that are scheduled to run at the same time.

Example 7.10: In the example in Figure 7.10, TimeDelay has level 0, whereas AddSubtract has level 1, so TimeDelay will fire before AddSubtract at any time stamp when it is scheduled to produce an output. However, it does not read its input until after the AddSubtract actor has fired in response to its own output event. TimeDelay reads its input in the postfire phase, at which time it simply records the input and requests a new firing at the current time plus its time delay (1.0, in this case).

Occasionally, it is necessary to put a TimeDelay actor in a feedback loop with a delay of 0.0. This has the effect of incrementing the microstep without incrementing the model time, which allows iteration without time advancing.

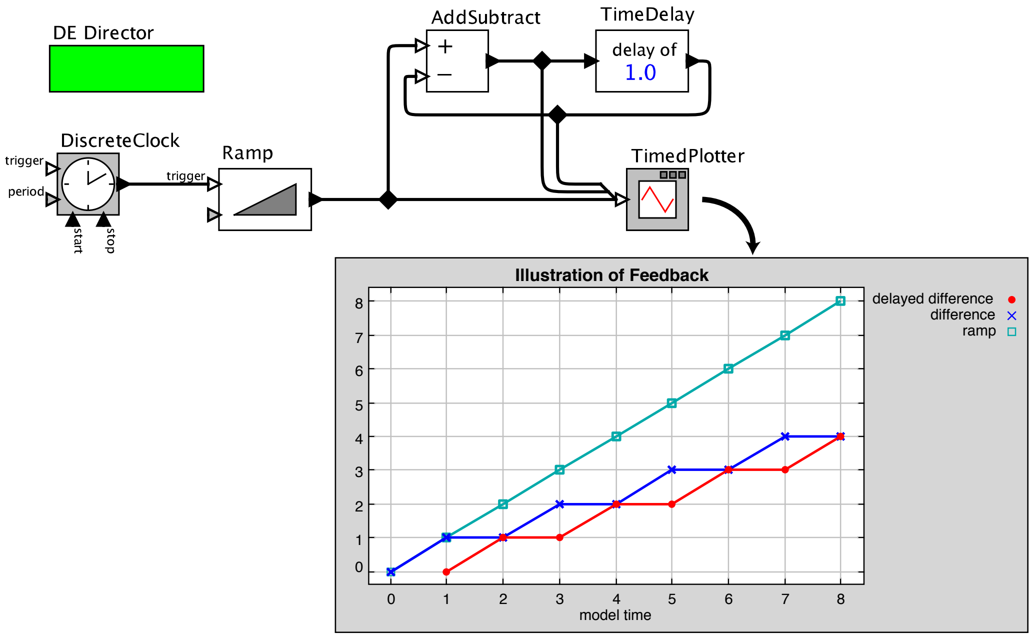


Figure 7.10: Discrete-event model with feedback, which requires a delay actor such as TimeDelay. [online]

Example 7.11: Consider the model in Figure 7.11, which produces the plot in Figure 7.12. This model produces a variable number of events at each integer model time using a feedback loop that has a TimeDelay actor with the delay set to 0.0. This causes the events that are fed back to use incremented microsteps at the same model time. This model uses a BooleanSwitch (see sidebar on page 119) to feed back a token only if its value is non-negative.

The previous example illustrates iteration in DE using a construct similar to the feedback iteration in dataflow illustrated in Example 3.11. In DE, we can use a Merge actor rather than a BooleanSelect because the use of time stamps makes the Merge determinate.

7.3.3 Multithreaded Execution

The DE director fires one actor at a time and does not fire the next actor until the previous one finishes firing. This approach creates two potential problems. First, if an actor does not return from its `fire` method, then the entire model will be blocked. This issue can arise if the actor is attempting to perform I/O. Second, the execution of the model is unable to exploit multicore architectures. Both of these problems can be solved using the **ThreadedComposite** actor, found in the `HigherOrderActors` library. The following example illustrates the first problem.

Example 7.12: Consider the example in Figure 7.13, which uses the InteractiveShell actor, previously considered in Section 4.1.1. In this model, the Expression actor is used to format a string for display (see Section 13.2.4 in Chapter 13).

Note that the time stamps in this model are not particularly meaningful. They do not accurately reflect the time at which the user types a value, but they do represent the order of what the user typed. That is, the time stamp is bigger for values that are entered later. In addition, this model cannot do anything while the InteractiveShell actor is waiting for the user to type something.

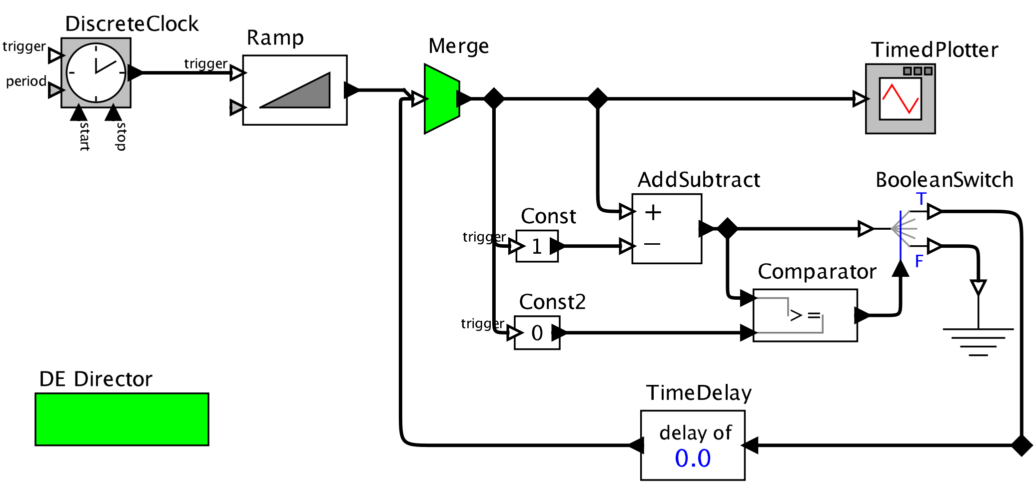


Figure 7.11: Illustration of TimeDelay with delay value of zero. [online]

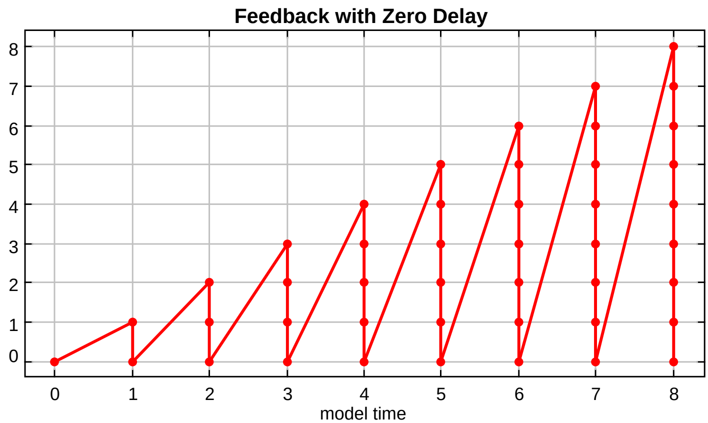


Figure 7.12: Result of executing the model in Figure 7.11.

The ThreadedComposite actor is an example of a higher-order component, which is an actor that has another actor as a parameter or an input. ThreadedComposite is parameterized by another actor, and when it fires, instead of performing any functionality itself, it begins the execution of the other actor in another thread, and returns immediately from the `fire` method. Since the actor's functionality executes in another thread, the model is not blocked. Most interestingly, the ThreadedComposite is able to perform such concurrent execution while ensuring determinate results. This capability can be used to create a much more useful interactive model, as shown in the next example.

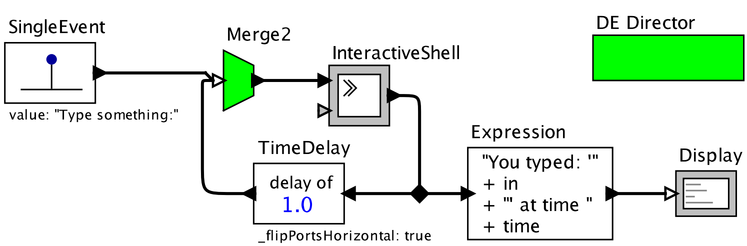


Figure 7.13: A DE model using the InteractiveShell actor, whose execution is stalled until a user types something. [online]

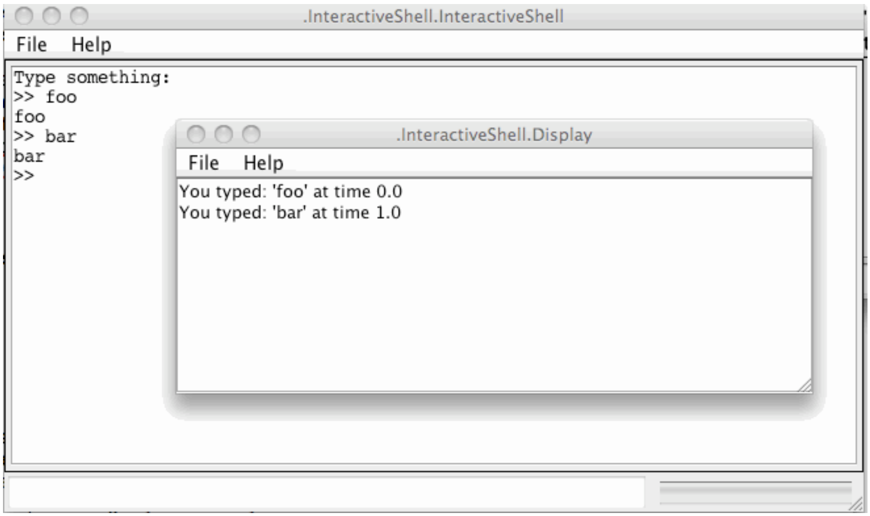


Figure 7.14: An execution of the model in Figure 7.13.

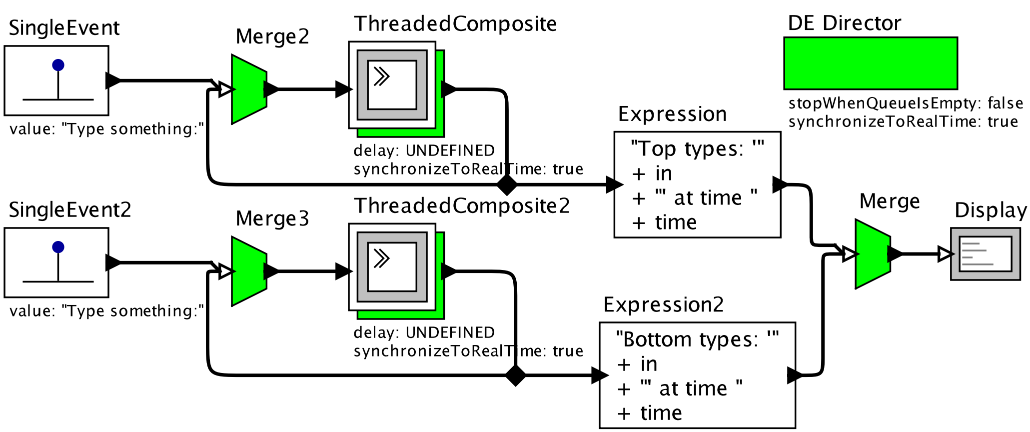


Figure 7.15: A model where two instances of InteractiveShell are executed in separate threads by the ThreadedComposite actor. [online]

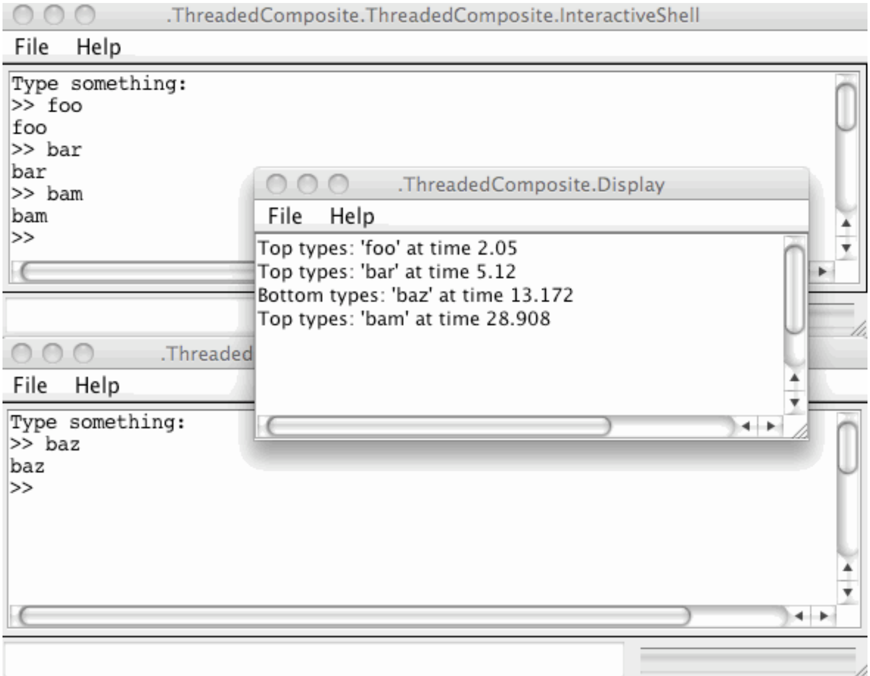


Figure 7.16: An execution of the model in Figure 7.15.

Example 7.13: Consider the model in Figure 7.15. This model opens two interactive shell windows into which users can type, as shown in Figure 7.16. The model will not block if the user does not type something, as it would without the ThreadedComposite.

This model is created by dragging two instances of ThreadedComposite, and then dropping instances of InteractiveShell onto the ThreadedComposite instances. The icons of the ThreadedComposite actors become like those of InteractiveShell, decorated with small green shadow.

In this model, the two instances of InteractiveShell execute asynchronously in threads that are separate from that of the DE director; when the model is running, there are three threads executing. The threads running the InteractiveShell actors block while waiting for user input. When the user types something and hits return, the InteractiveShell actor produces an output, causing its containing ThreadedComposite to produce an output.

There are a number of subtle points about this model. The *delay* parameter of the ThreadedComposite actors is set to `UNDEFINED`, which instructs the ThreadedComposite actors to assign the current model time, whatever that time may be, to their output event time stamps. The time stamps of the outputs are therefore nondeterminate.

If instead we had set this parameter to some number τ, then the output events would be assigned model time $t + \tau$, where t is the model of time of the triggering input event. This makes the output time stamps determinate. However, it also limits concurrency. When *delay* is set to a number τ, the model will block when current model time reaches $t + \tau$. Otherwise, the ThreadedComposite would attempt to produce output events with time stamps in the past.

Another subtle effect is that the *synchronizeToRealTime* parameter of the director is set to `true`. This ensures that "current model time" advances no faster than real time. Thus, in the output traces shown in Figure 7.16, the reported times can be interpreted as a measure of the elapsed time in seconds from the start of execution until the user typed something. This gives the time stamps a physical meaning. It also justifies the nondeterminacy in the model, since the time at which a user types something is certainly nondeterminate (it is not specified by the model).

A third subtlety is that the *stopWhenQueueIsEmpty* parameter of the director is set to `false`. By default, the DE director will stop executing when there are no more events to process. But in this model, events can still appear later on, because the user types something. Thus, we do not want the model to stop when the queue becomes empty.

The ThreadedComposite actor offers a mechanism for executing models concurrently in multiple threads, but the mechanism is much more determinate and controllable than using threads directly, which is fraught with difficulties (Lee, 2006). The actor contained by a ThreadedComposite need not be an atomic actor; it can be an arbitrarily complex composite actor. The fact that the contained actor executes in a separate thread enables the use of actors that may get stalled waiting for I/O, and it also enables parallel execution on multicore machines for improved performance. The subtleties of this actor and further details on its usage are described by Lee (2008b).

7.3.4 Scheduling Limitations

As of this writing, the DE director in Ptolemy II implements an approximation of the semantics described by Lee and Zheng (2007). In particular, the current implementation is not able to execute all models that can, in theory, be executed by an exact implementation of the semantics.

Example 7.14: Consider the model shown in Figure 7.17. This model has an opaque composite actor in a feedback loop. The *clock* output of the composite actor does not depend on the input. Hence, at any given time stamp, the composite actor should be able to produce an event on the *clock* output without knowing whether an event is present on the input port. However, attempting to execute this model results in an exception:

```
IllegalActionException:  Found a zero delay loop containing
OpaqueComposite
  in FixedPointLimitation
```

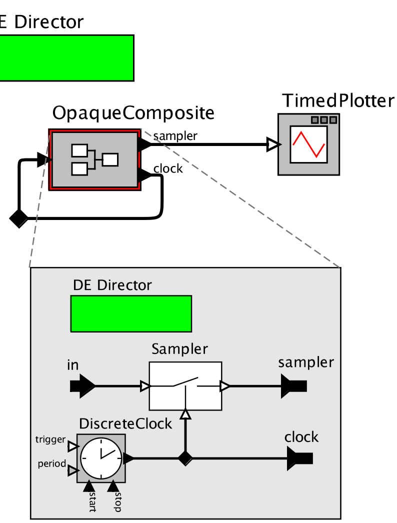


Figure 7.17: A model that should, in theory, be executable, but is not executable in the current implementation of DE.

In the current implementation, at each time stamp (model time and microstep), the DE director will fire an actor at most once, and when it fires that actor, it guarantees that all input events at that time stamp are available. As a consequence, the composite actor in Figure 7.17 can never be fired, because the director cannot ensure that an input event is present or absent at the current time stamp until it has already fired the composite actor once at that time stamp. Note that this problem cannot be corrected by putting a zero-delay TimeDelay actor in the feedback loop (see Exercise 1). It *can* be corrected by using a director with a fixed point semantics, as shown in Example refexample:FixedPointNoLimitation.

7.4 Zeno Models

It is possible to create DE models where time fails to advance, as illustrated by the example below.

Example 7.15: Suppose that if in Figure 7.11 we were to omit the BooleanSwitch and unconditionally feed back the tokens. Then time would never advance; only the microsteps would advance.

A model where model time stops advancing and only the microsteps advance is called a **chattering Zeno** model. Microsteps are implemented by the DE director as a Java *int*, so the increment of the microstep will eventually overflow, causing the director to report an exception.

It is also possible to create models, called **Zeno** models, where the time advances, but will not advance past some finite value.

Example 7.16: An example of a Zeno model is shown in Figure 7.18. This model triggers a feedback loop using a SingleEvent actor (see sidebar on page 241). The feedback loop contains a TimeDelay actor (see sidebar on page 243) whose delay value is given by $1/n^2$, where n begins at $n = 1$ and is incremented each time a token cycles around the loop. The delay, therefore, approaches zero, and this model can produce an infinite number of events before time 2.0.

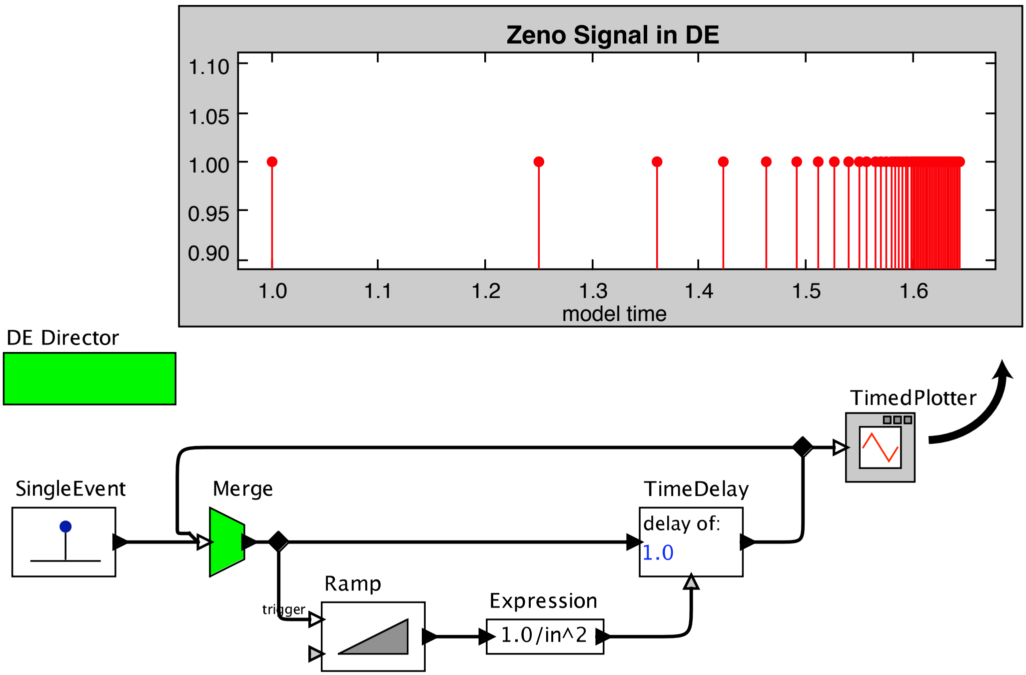


Figure 7.18: Example of a Zeno model, where the model time fails to advance. [online]

In DE, because time has finite precision (see Section 1.7.3), eventually a Zeno model becomes a chattering Zeno model. When the time increment falls below the time resolution, then time stops advancing. The particular model in the previous example, however, fails before that happens because when n becomes sufficiently large, floating point errors in calculating $1/n^2$ (oddly) yield a negative number, causing the TimeDelay actor to report an exception.

7.5 Using DE with other Models of Computation

DE models can be usefully combined with other models of computation. Here we highlight some of the most useful combinations.

7.5.1 State Machines and DE

As shown in Figure 7.1, an actor in a DE model may be defined by a state machine. Such a state machine may be capable of initiating a feedback loop without requiring an external stimulus event, as illustrated by the next example.

Example 7.17: Figure 7.19 shows a simple DE model containing a single FSMActor. In this example, the initial state has an enabled transition (with guard `true`), which causes the FSMActor to fire at the execution start time and produce an output. This output, in turn, initiates the feedback loop. This actor does not need an input event to trigger the first firing and initiate feedback.

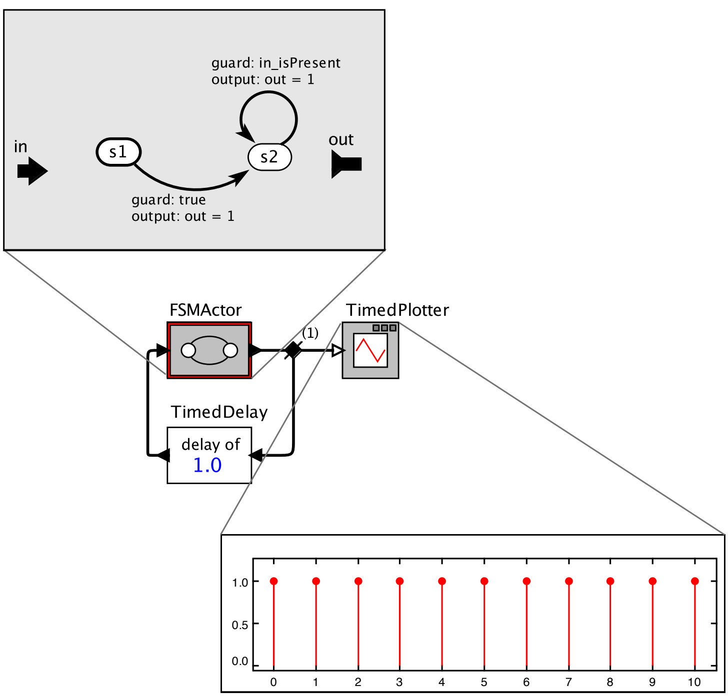


Figure 7.19: A simple DE model containing an FSM. [online]

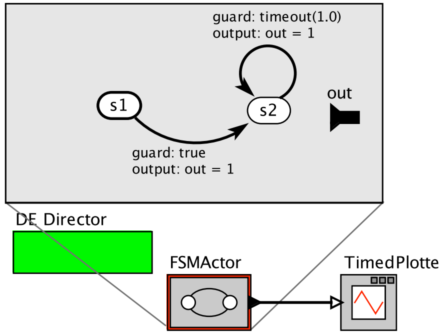


Figure 7.20: A DE model with identical behavior to that in Figure 7.19, but containing an FSM that controls timing with the timeout function. [online]

After the first firing, the actor is in state *s2*, where the outgoing transition has guard `in_isPresent`. Hence, all subsequent firings require an input event. The net result is an unbounded sequence of events one time unit apart, as shown in the plot.

The same effect can be achieved by an FSM that directly controls timing using the `timeout` function (see Table 6.2), as shown in Figure 7.20.

7.5.2 Using DE with Dataflow

It is possible to use a dataflow director within a DE model. The SDF director, for example, can be quite useful within DE models. If a composite actor in a DE model contains an SDF director, then the internal dataflow model will execute one complete iteration each time an event is received on an input port. This approach can be useful when the overall DE system model includes complex computations that do not involve timed events and are conveniently described by a dataflow model.

The DDF director can also be used within DE models, although as discussed in Section 3.2, controlling the definition of an iteration in DDF is more difficult than with SDF. Using the dataflow PN (Process Network) director within DE rarely makes sense, because, as discussed in Section 4.1, it is quite difficult to deterministically bound an iteration.

The dataflow directors described in Chapter 3 are generally untimed, except that (as described earlier) the SDF director has a *period* parameter. This parameter can be useful within a DE model. If the parameter is set to something other than zero, then the SDF submodel will execute periodically within the DE model, regardless of whether any input events are provided. This approach can be used to design clock actors with much more complicated output patterns than are easily specified using DiscreteClock. Note, however, that the submodel will execute *only* at multiples of the *period*, and not whenever an input event is provided. If at some multiple of the period there are insufficient inputs to execute a complete iteration, then the SDF model will not fire at that time. Moreover, if inputs are provided faster than the SDF submodel consumes them, then they will queue up, possibly eventually exhausting available memory. See Exercise 2 for an example.

It is rarely useful to put a DE model inside an SDF model (or any other dataflow model). The DE submodel will expect to be fired at model times determined by its internal actors, but the SDF model will only fire at multiples of the *period*. Hence, this combination will typically trigger an exception similar to the following:

```
IllegalActionException:  SDF Director is unable to fire CompositeActor
at the requested time:  ...  .  It responds it will fire it at:  ...
  in .DEwithinSDF.CompositeActor.DE Director
```

It is possible, however, to put a DE submodel within an SDF model if the SDF model is itself nested within a DE model, and the *period* parameter of the SDF model is set to zero. In this case, the SDF director will delegate firing requests to the higher-level DE director, ignoring time advancements itself.

7.6 Wireless and Sensor Network Systems

The **wireless domain** builds on the DE domain to support modeling of wireless networks. In the wireless domain, channel models mediate communication between actors, and the visual syntax (i.e., the graphical representation of the model) does not require wiring between components. The visual representation of models in the wireless domain is more important than in other Ptolemy II domains because the location of icons forms a two-dimensional map of the wireless system being modeled. The positions of icons on the screen, the distance between them and the objects in between them, affect their communication.

Example 7.18: The top level of the model in Figure 7.21 contains a WirelessDirector, two instances of **WirelessComposite**, and a **DelayChannel**. WirelessComposite1 has an output port, and WirelessComposite2 has an input port. Although these ports are not directly connected, they do communicate with each other. Each port has a parameter *outsideChannel* that names the wireless channel through which it communicates, which in this case is DelayChannel.

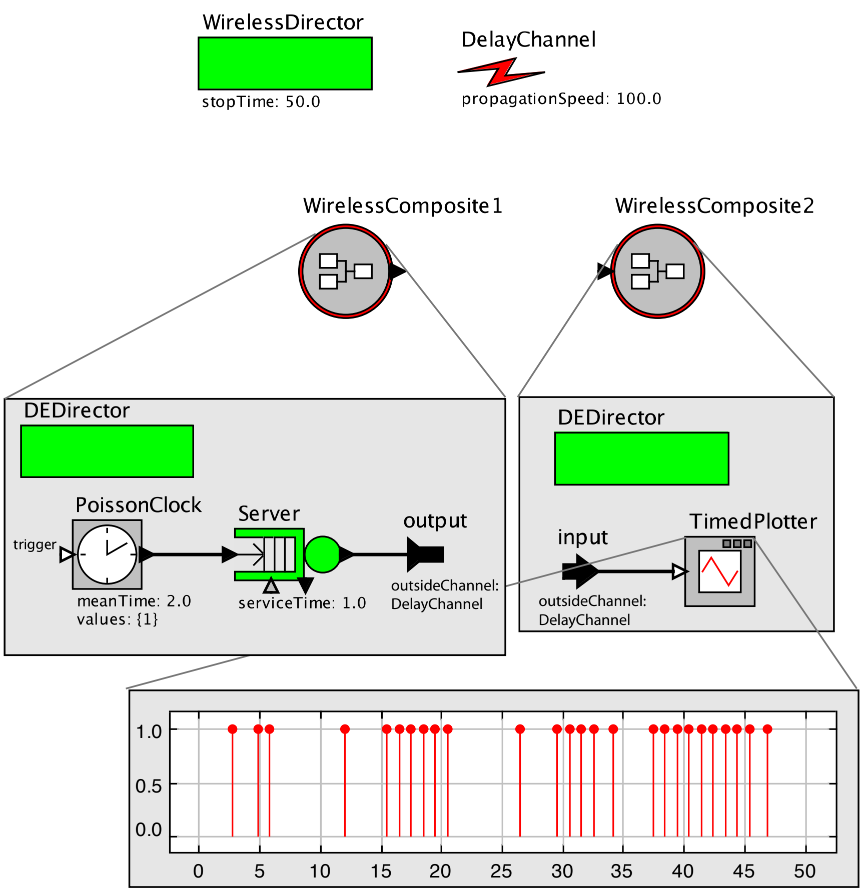


Figure 7.21: A model of a wireless system. [online]

The DelayChannel component models a wireless communication channel that, in this simple example, delays the message by an amount of time that is proportional to the distance between the two WirelessComposite icons in the model. The proportionality constant is given by the *propagationSpeed* parameter (in arbitrary units of distance/time). In this case, the icons are about 175 units apart, so a propagation speed of 100 translates into a delay of approximately 1.75 time units.

In this model, WirelessComposite1 is a **sporadic source** of events. A sporadic source is a random source where there is a fixed lower bound on the time between events. In this case, the lower bound is enforced by the Server actor (see sidebar on page 246) and the randomness is provided by the PoissonClock actor.

WirelessComposite2 in this model simply plots the received events as a function of time. The first event is sent by WirelessComposite1 at time 1.0 and received by WirelessComposite2 at approximately 2.75. The plot shows that no two events are more closely spaced than one time unit apart.

In addition to modeling channel delays, the wireless domain can model power loss, interference, noise, and occlusion. It can also model directional antenna gain and mobile transmitters and receivers. See Baldwin et al. (2004) and Baldwin et al. (2005) for details, and see the demos included in the Ptolemy II package for examples.

7.7 Summary

The DE domain provides a solid foundation for modeling discrete timed behavior. Mastering its use requires an understanding of the model of time, simultaneity, and feedback, each of which is covered in this chapter. A key characteristic of this domain is that execution is determinate even when events are simultaneous.

Exercises

1. Consider the model in Figure 7.22. Unlike the model Figure 7.17, the DE director can execute this model because of the TimeDelay in the feedback loop.

   (a) Explain why the model in Figure 7.22 produces no output events.

   (b) Explain why the model in Figure 7.22 is not equivalent to the model in Figure 7.17, were the DE director able to execute it.

2. Consider the model in Figure 7.23. It has an SDF submodel within a DE model.

   (a) Suppose the *period* parameter of the SDF director is 1.5 and the *period* of the DiscreteClock is 1.0. What output do you expect from the CompositeActor? Does this model execute with bounded memory?

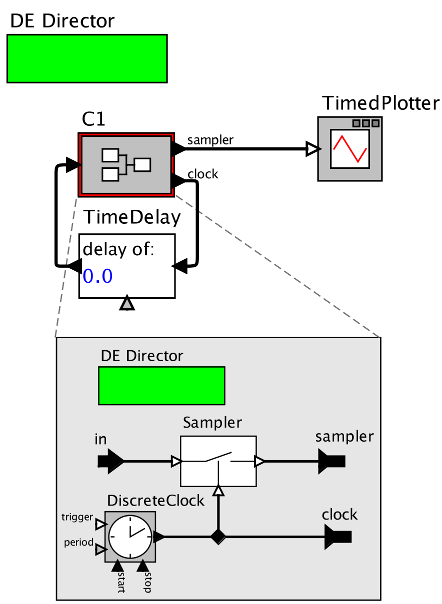


Figure 7.22: A model that is executable but is not equivalent to the model in Figure 7.17. [online]

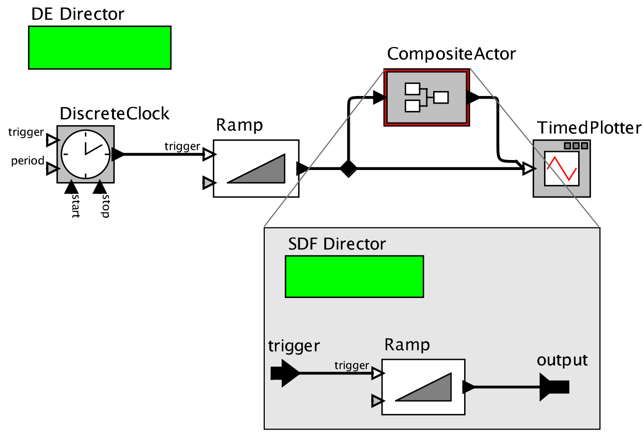


Figure 7.23: Simple example of an SDF submodel within a DE model. [online]

(b) Find values of the *period* parameter of the SDF director and of the DiscreteClock actor that will generate the plot in Figure 7.24. Explain why this output makes sense.

3. This problem explores some subtleties of combining FSMs with DE models. Construct a DE model consisting of a PoissonClock that triggers a Ramp that provides

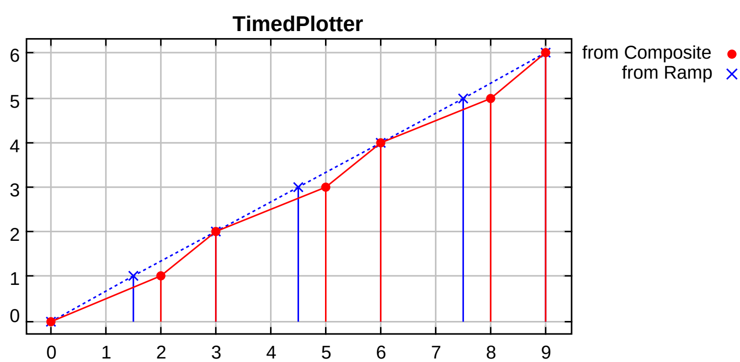


Figure 7.24: A plot that can be generated by the model in Figure 7.23.

input to an FSM (see Chapter 6). Set the *fireAtStart* parameter of the PoissonClock actor to `false`) so that it does not produce an output at time zero.

(a) Construct an FSM that will produce an output at time zero even with no input event at time zero.

(b) Modify your FSM so that, in addition, when it does receive an input event, it produces two outputs at the model time of the input event. The first such output should have the same value as the input event. The second such output should occur one microstep later and should have a value that is twice the value of the input event.

8

Modal Models

Thomas Huining Feng, Edward A. Lee, Xiaojun Liu,
Stavros Tripakis, Haiyang Zheng, and Ye Zhou

Contents

Most interesting systems have multiple modes of operation. Changes in modes may be triggered by external or internal events, such as user inputs, hardware failures, or sensor data. For example, an engine controller in a car may have different behavior when the car is in Park than when it is in Drive.

A **modal model** is an explicit representation of a finite set of behaviors (or modes) and the rules that govern transitions between them. The rules are captured by a finite state machine (FSM).

In Ptolemy II, the ModalModel actor is used to implement modal models. ModalModel is a hierarchical actor, like a composite actor, but with multiple refinements instead of just one. Each refinement specifies a single mode of behavior, and a state machine determines which refinement is active at any given time. The ModalModel actor is a more general form of the FSMActor described in Chapter 6; the FSMActor does not support state refinements. Modal models use the same transitions and guards described in Chapter 6, plus some additional ones.

Example 8.1: The model shown in Figure 8.1 represents a communication channel with two modes of operation: clean and noisy. It includes a ModelModel actor (labeled "Modal Model") with two states, *clean* and *noisy*. In the *clean* mode, the model passes inputs to the output unchanged. In the *noisy* mode, it adds a Gaussian random number to each input token. The top-level model provides an *event* signal generated by a PoissonClock actor, which generates events at random times according to a Poisson process. (In a Poisson process, the time between events is given by independent and identically distributed random variables with an exponential distribution.) A sample execution of this model, in which the Signal Source actor provides an input sine wave, results in the plot shown in Figure 8.2.

This example combines three distinct models of computation (MoCs). At the top level, the timed behavior of randomly occurring events is captured using the DE domain. The next level down in the hierarchy, an FSM is used to capture mode changes. The third level uses SDF to capture untimed processing of sample data.

The process of creating a modal model is illustrated in Figure 8.3. To create a modal model in Vergil, drag in a ModalModel actor from the `Utilities` library and populate

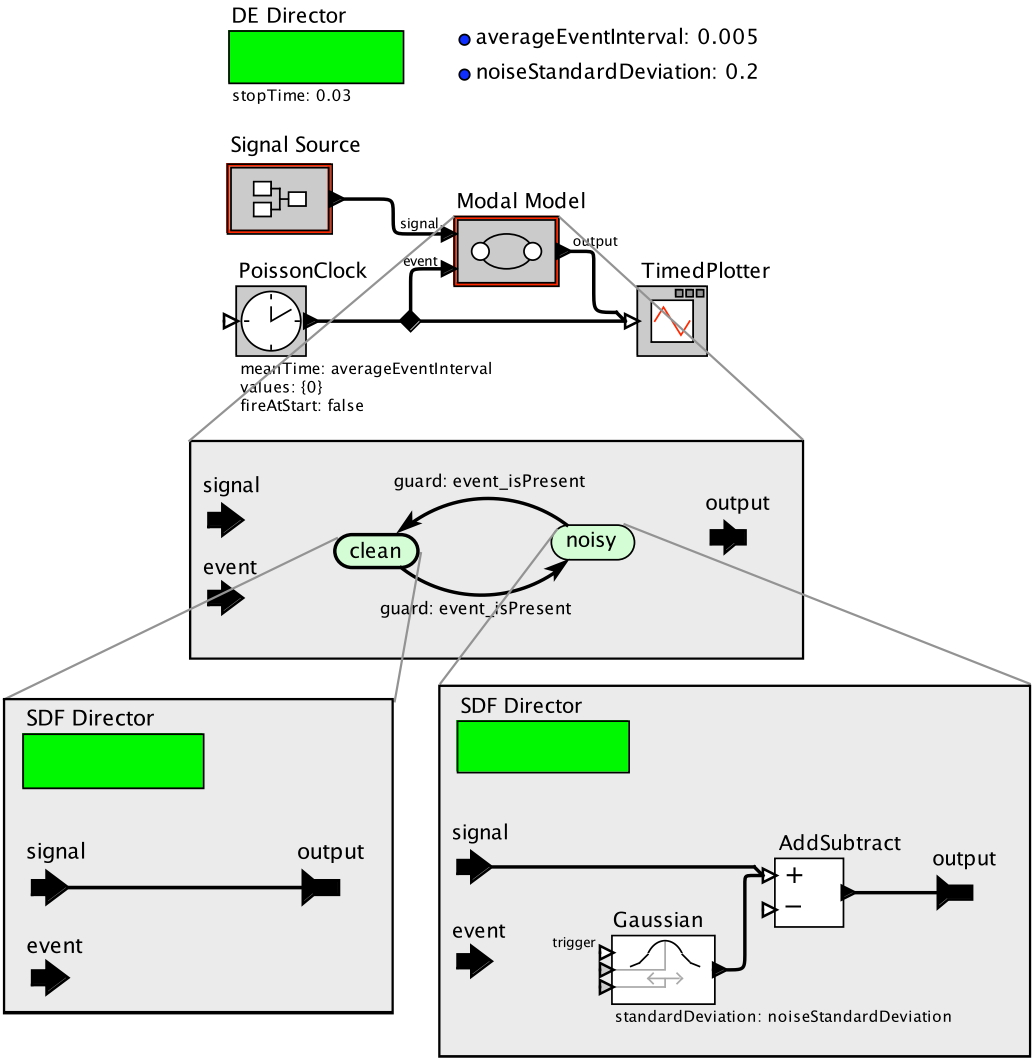


Figure 8.1: Simple modal model that has a normal (clean) operating mode, in which it passes inputs to the output unchanged, and a faulty mode, in which it adds Gaussian noise. It switches between these modes at random times determined by the PoissonClock actor. [online]

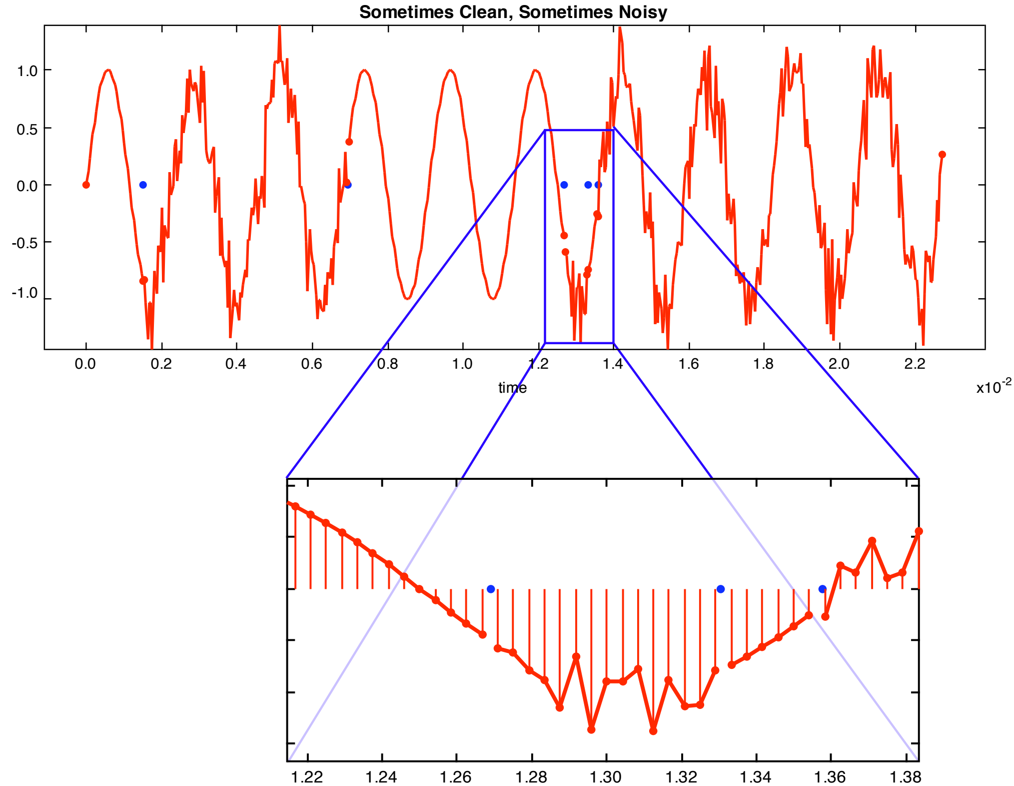


Figure 8.2: Plot generated by the model in Figure 8.1.

it with ports. Open the modal model actor and add one or more states and transitions. To create the transitions, hold the Control key (or the Command key on a Mac) and click and drag from one state to the other. To add a refinement, right click on a state and select `Add Refinement`. You can choose a `Default Refinement` or a `State Machine Refinement`. The former is used in the above example; it will require in each refinement a director and actors that process input data to produce outputs. The latter will enable creation of a hierarchical FSM, as described in Chapter 6.

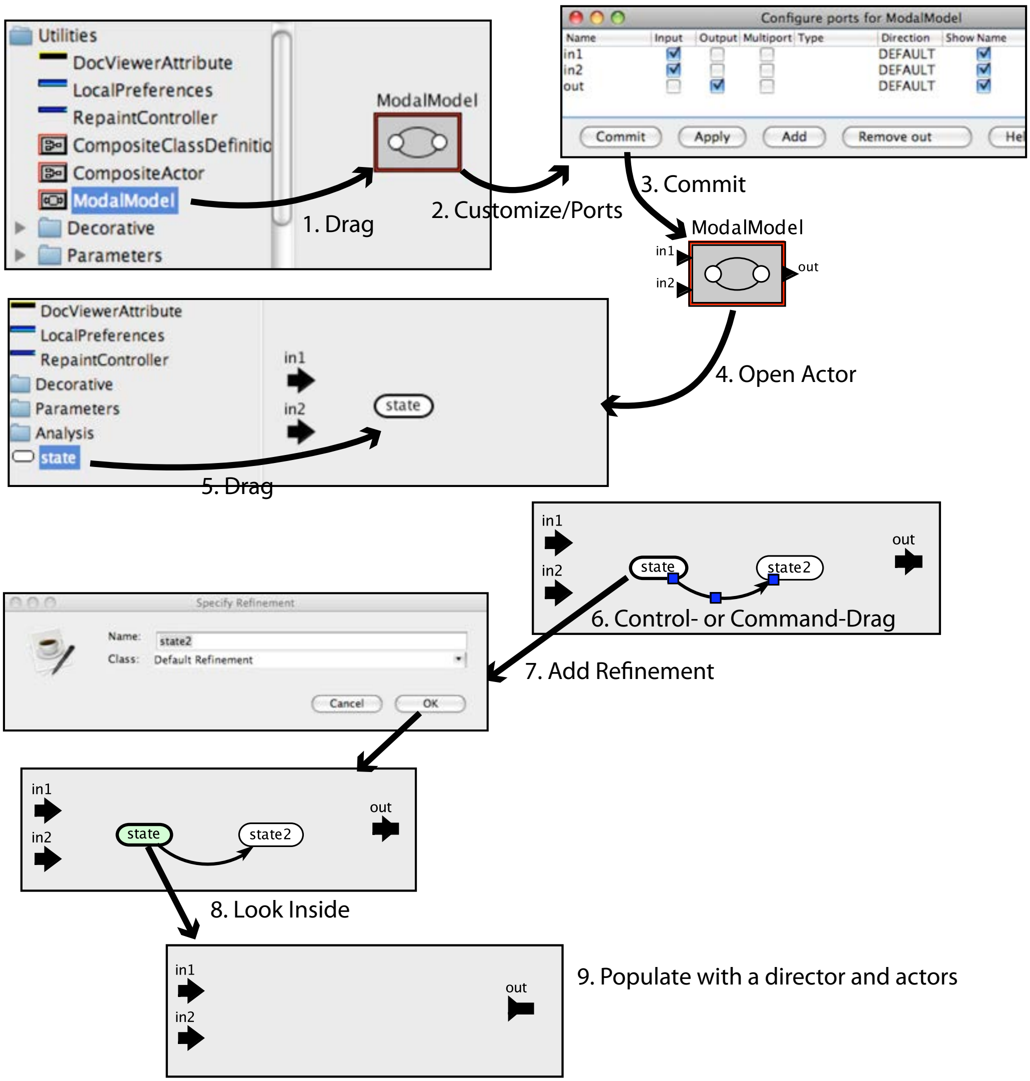


Figure 8.3: How to create modal models.

8.1 The Structure of Modal Models

The general structure of a modal model is shown in Figure 8.4. The behavior of a modal model is governed by a state machine, where each state is called a **mode**. In Figure 8.4, each mode is represented by a bubble (like a state in a state machine) but it is colored to indicate that it is a mode rather than an ordinary state. A mode, unlike an ordinary state, has a **mode refinement**, which is an opaque composite actor that defines the mode's behavior. The example in Figure 8.1 shows two refinements, each of which is an SDF model that processes input tokens to produce output tokens.

The mode refinement must contain a director, and this director must be compatible with the director that governs the execution of the modal model actor. The example in Figure 8.1 has an SDF director inside each of the modes and a DE director outside the modal model. SDF can generally be used inside DE, so this combination is valid.

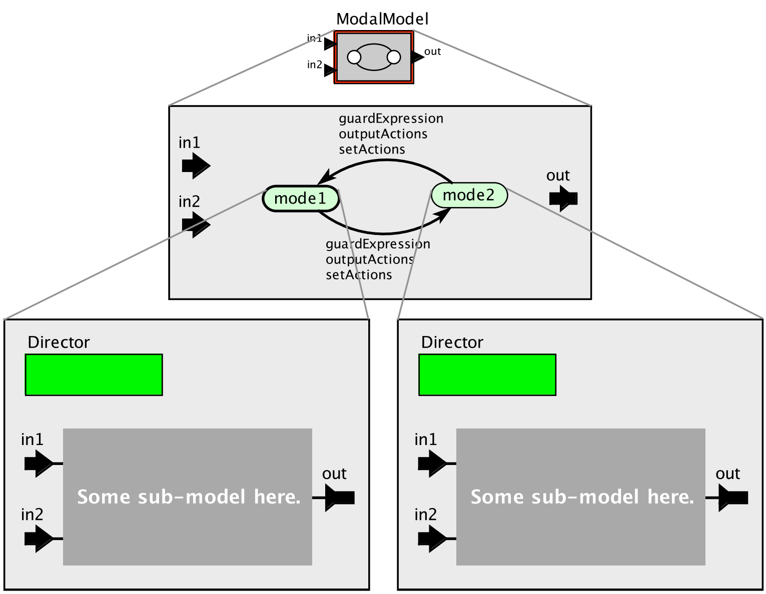


Figure 8.4: General pattern of a modal model with two modes, each with its own refinement.

Like in a finite state machine, modes are connected by arcs representing transitions with guards that specify when the transition should be taken.

Example 8.2: In Figure 8.1, the transitions are guarded by the expression `event_isPresent`, which evaluates to true when the *event* input port has an event. Since that input port is connected to the PoissonClock actor, the transitions will be taken at random times, with an exponential random variable governing the time between transitions.

A variant of the structure in Figure 8.4 is shown in Figure 8.5, where two modes share the same refinement. This is useful when the behavior in different modes differs only by parameter values. For example, Exercise 2 constructs a variant of the example in Figure

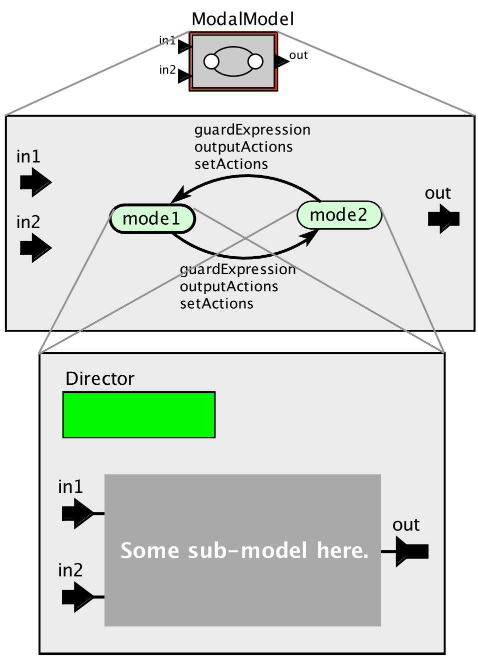


Figure 8.5: Variant of the pattern in Figure 8.4 where two modes share the same refinement.

8.1 where the *clean* refinement differs from the *noisy* refinement only by having a different parameter value for the Gaussian actor. To construct a model where multiple modes have the same refinement, add a refinement to one of the states, giving the refinement a name (by default, the suggested name for the refinement is the same as the name of the state, but the user can choose any name for the refinement). Then, for another state, instead of choosing `Add Refinement`, choose `Configure` (or simply double click on the state) and specify the refinement name as the value for the *refinementName* parameter. Both modes will now have the same refinement.

Another variant is when a mode has multiple refinements. This effect can be accomplished by executing `Add Refinement` multiple times or by specifying a comma-separated list of refinement names for the *refinementName* parameter. These refinements will execute in the order that they are added. This order can be changed by invoking `Configure` on the state (or double clicking on it) and editing the comma-separated list of refinements.

Probing Further: Internal Structure of a Modal Model

In Ptolemy II, every object (actor, state, transition, port, parameter, etc.) can have at most one container. Yet in a modal model, two states can share the same refinement, which may seem to violate that general rule.

The key difference is that a ModalModel actor is actually a specialized composite actor that contains an instance of FSMDirector, an FSMActor, and any number of composite actors. Each composite actor can be a refinement for any state of the FSMActor. The FSMActor is the controller, in the sense that it determines which mode is active at any time. The FSMDirector ensures that input data is delivered to the FSMActor and all active modes. This same structure is used for the hierarchical FSMs explained in Section 6.3.

The Vergil user interface, however, hides this structure. When you execute an `Open Actor` command on a ModalModel, the user interface skips a level of the hierarchy and takes you directly the FSMActor controller. It does not show the layer of the hierarchy that contains the FSMActor, the FSMDirector, and the refinements. Moreover, when you `Look Inside` a state, the user interface goes up one level of the hierarchy and opens *all* refinements of the selected state. This architecture balances expressiveness with user convenience.

8.2 Transitions

All the transition types of Table 6.1 can be used with modal models. They have exactly the same meaning given in that table. The transition types shown in Table 6.3, which are explained for hierarchical FSMs, however, have slightly different meanings for refinements that are not FSMs. Refinements of a state in an FSM can be arbitrary opaque composite actors (composites that contain a director). They can even be mixed, where some refinements are FSMs and some are other kinds of models. The more general meanings for such transitions are explained in this section, and then summarized in Table 8.1.

8.2.1 Reset Transitions

By default, a transition is a reset transition, which means that the refinements of the destination state are initialized when the transition is taken. If the refinement is an FSM, as explained in Section 6.3, this simply means that the state of the FSM is set to its initial state. If that initial state itself has refinement state machines, then those too are set to their initial states. In fact, the mechanism of a reset transition is simply that the initialize method of the refinement is invoked. This causes all components within the refinement to be initialized.

Example 8.3: For the example in Figure 8.1, it does not matter whether the transitions are history transitions or not because the refinements of the two states themselves have no state. The actors in the model (Gaussian and AddSubtract) have no state, so initializing them does not change their behavior.

In the example in Figure 8.6, however, the Ramp actors have state. The example shows the transitions being history transitions, which produces the plot in Figure 8.7(a). In this case, the Ramp actors will resume counting from where they last left off when a state is re-entered.

If on the other hand we were to change the transitions to reset transitions, the result would be the plot in Figure 8.7(b). Each time a transition is taken, the Ramp actors are initialized (along with the rest of the refinement), so they begin again counting from zero.

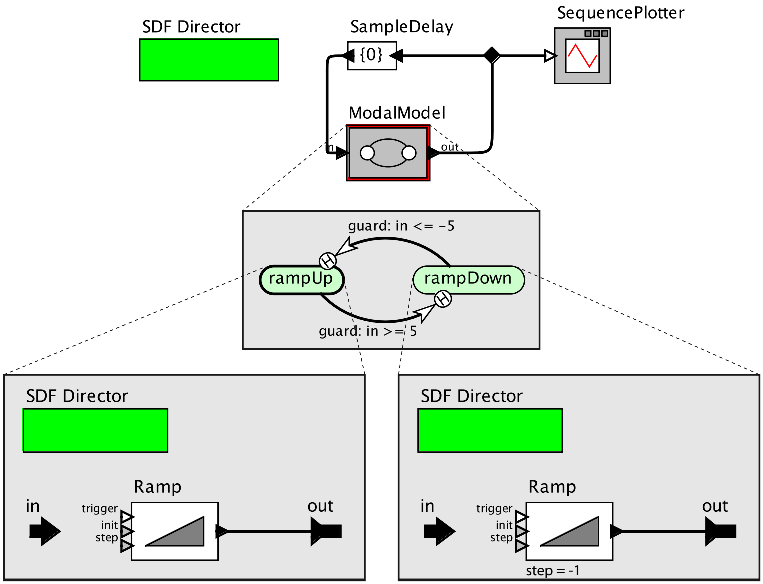


Figure 8.6: A modal model whose behavior depends on whether transitions are reset transitions or history transitions. [online]

8.2.2 Preemptive Transitions

For general modal models, preemptive transitions work the same way as for hierarchical FSMs. If the guard is enabled, then the refinement does not execute. A consequence is that the refinement does not produce output.

Example 8.4: In Figure 8.8, we have modified Figure 8.6 so that the transitions are both preemptive. This means that when a guard evaluates to true, the refinement of the current state does not produce output. In this particular model, no output at all is produced in that iteration, violating the contract with SDF, which expects every firing to produce a fixed, pre-determined number of tokens. An error therefore arises, as shown in the figure. This error can be corrected by producing an output on the transitions or by using a different director.

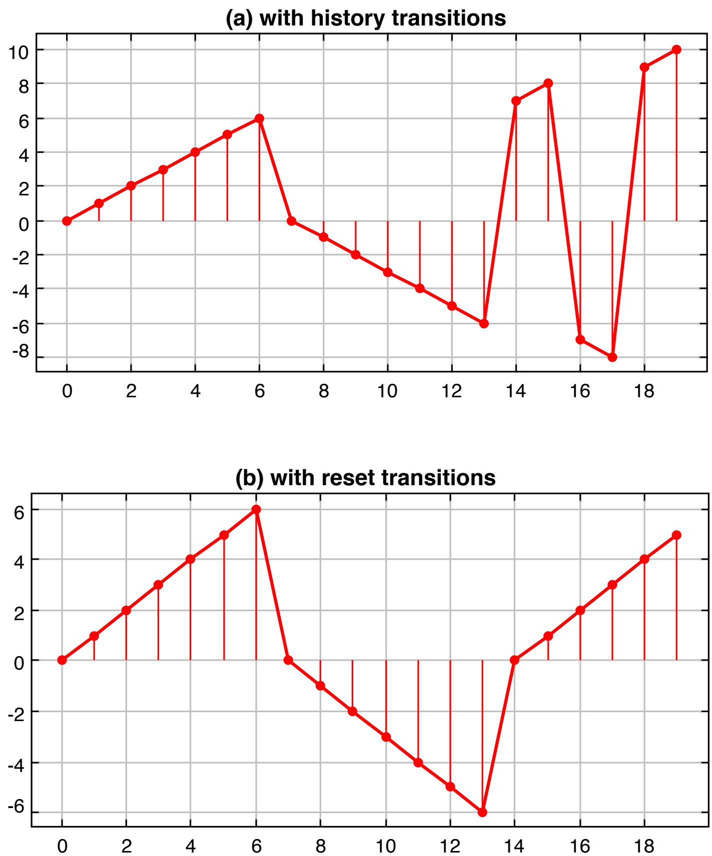


Figure 8.7: (a) The plot resulting from executing the model in Figure 8.6, which has history transitions. (b) The plot that would result from from changing the transitions to reset transitions.

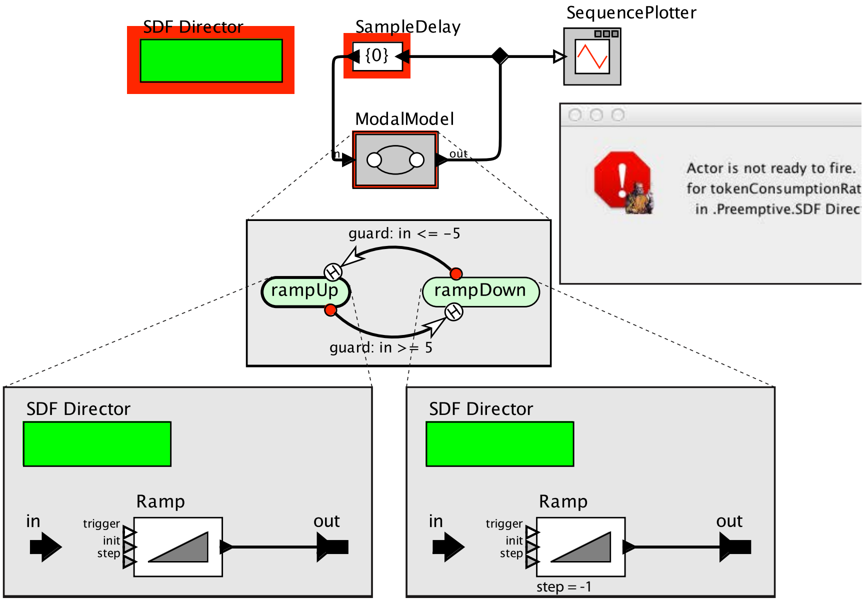


Figure 8.8: A modal model where preemptive transitions prevent the refinements from producing outputs that are expected by the SDF director. [online]

8.2.3 Error Transitions

When executing a refinement, an error may occur that causes an exception to be thrown. By default, an exception will cause the entire execution of the model to halt. This is not always desirable. It might be possible to gracefully recover from an error. To allow for this, Ptolemy II state machines include an **error transition**, which is enabled when an error occurs while executing a refinement of the current state. An error transition may also have a guard, output actions, and set actions. Some caution is necessary when using output actions, however, because if the error occurs in the postfire phase of execution of the refinement, then it may be too late to produce outputs. Most errors, however, will occur in the fire phase, so most of the time this will not be a problem.

Example 8.5: A model with an error transition is shown in Figure 8.9. Like Example 8.6, this model includes an InteractiveShell actor, which allows the user to

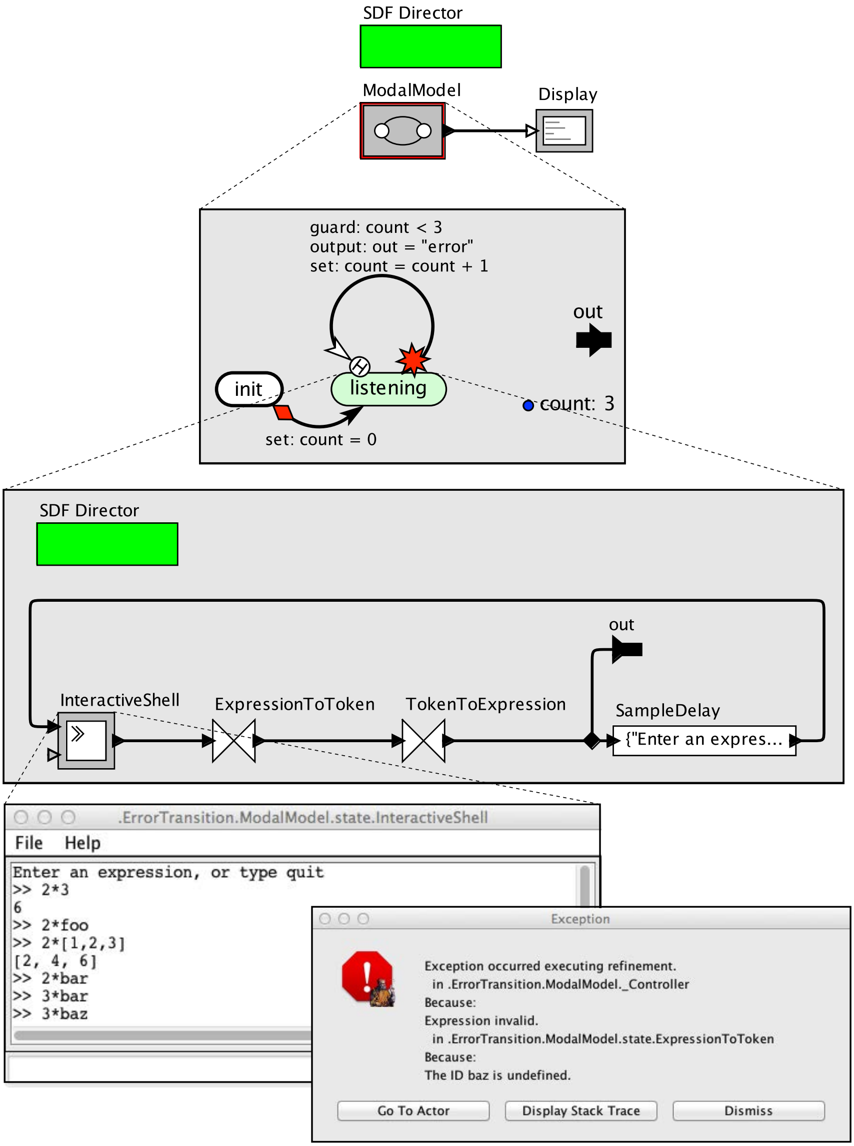


Figure 8.9: A modal model with an error transition. [online]

type in arbitrary text. In this model, what the user types is then sent to an ExpressionToToken actor, which parses what the user types, interpreting the text as an expression in the Ptolemy II expression language (see Chapter 13). Of course, the user may type an invalid expression, which will cause ExpressionToToken to throw an exception.

In the FSM, the *listening* state has an error transition self loop. The error transition is indicated by the red star at its stem. It is enabled when the refinement of the *listening* state has thrown an exception and its guard (if there is a guard) is true. In this case, the guard ensures that this transition is taken no more than three times. After it has been taken three times, it will no longer be enabled.

An example of an execution of this model is shown at the bottom of the figure. Here, the user first types in a valid expression, "`2*3`," which produces the result 6. Then the user types an invalid expression, "`2*foo`." This is invalid because there is no variable named "`foo`" in scope. This triggers an exception, which will be caught by the error transition.

In this simple example, the error transition simply returns to the same state. In fact, this transition is also a history transition, so the refinement is not reinitialized. This could be dangerous with error transitions because an exception may leave the refinement in some inconsistent state. But in this case, it is OK. Were this a reset transition, then the InteractiveShell would be initialized after the error is caught. This would cause the shell window to be cleared, erasing the history of the interaction with the user.

On the fourth invalid expression, "`3*baz`," the error transition guard is no longer true, so the exception is not caught. This causes the model to stop executing and an exception window to appear, as shown at the bottom of the figure.

Error transitions provide quite a powerful mechanism for recovering from errors in a model. When an error transition is taken, two variables are set that may be used in the guard or the output or set actions of this transition:

- *errorMessage*: The error message (a string).
- *errorClass*: The name of the class of the exception thrown (also a string).

In addition, for some exceptions, a third variable is set:

- *errorCause*: The Ptolemy object that caused the exception.

For the above example, the *errorCause* variable will be a reference to the ExpressionTo-Token actor. This is an ObjectToken on which you can invoke methods such as `getName` in the guard or output or set actions of this transition (see Chapter 14).

8.2.4 Termination Transitions*

A termination transition behaves rather differently when the state refinements are general Ptolemy models rather than hierarchical FSMs. Such a transition is enabled when all refinements of the current state have terminated, but for general Ptolemy models, it is not possible to know whether the model has terminated prior to the postfire phase of execution. As a consequence, if at least one of the refinements of the current state is a default refinement (vs. a state machine refinement), then:

- the termination transition is not permitted to produce outputs, and
- the termination transition has lower priority than any other transition, including default transitions.

The reason for these constraints is a bit subtle. Specifically, in many domains (SR and Continuous, for example), the postfire phase is simply too late to be producing outputs. The outputs will not be seen by downstream actors. Second, the guards on all other transitions (non-termination transitions) will be evaluated in fire phase of execution, and a transition may be chosen before it is even known whether the termination transition will become enabled.

As a consequence of these constraints, termination transitions are not as useful for general refinements as they are for hierarchical FSMs. Nevertheless, they do occasionally prove useful.

Example 8.6: Figure 8.10 shows a model that uses a termination transition. The key actor here is the InteractiveShell, which opens a dialog window into which the user can type, as shown in Figure 8.11. The InteractiveShell asks the user to type something, or to type "quit" to stop. When the user types "quit," the postfire

*Termination transitions are rather specialized. The reader may want to skip this subsection on a first reading.

method of the InteractiveShell returns false, which causes the bottom SDFDirector to terminate the model (SDF terminates a model when any actor terminates because the SDF contract to produce a fixed number of tokens can no longer be honored).

In the FSM, the transition from *listening* to *check* is a termination transition, so it triggers when the user types quit. This transition has a set action of the form:

```
response = yesNoQuestion("Do you want to continue?")
```

which invokes the yesNoQuestion function to pop up a dialog asking the user a question, as shown in Figure 8.11 (see Table 13.16 in Chapter 13 for information

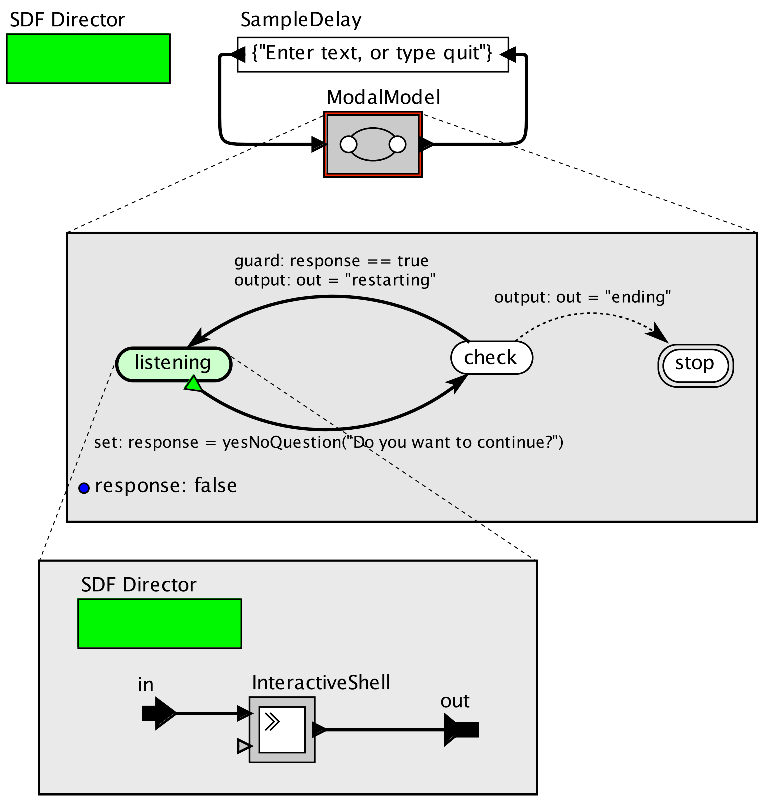


Figure 8.10: A modal model with a termination transition. [online]

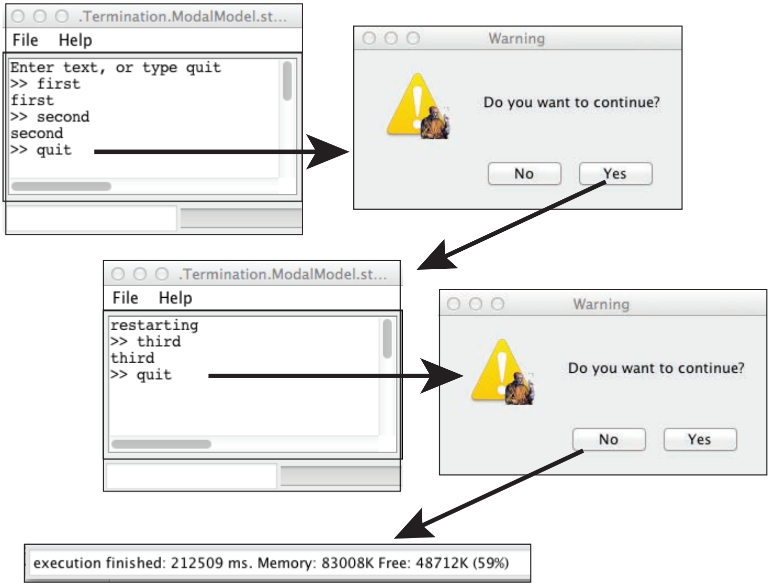


Figure 8.11: An execution of the model in Figure 8.10.

about the yesNoQuestion function). If the user responds "yes" to the question, then the transition sets the *response* parameter to true, and otherwise it sets it to false. Hence, in the next iteration, the FSM will either take a reset transition back to the initial listening state, opening another dialog, or it will transition to *stop*, a final state. Transitioning to a final state will cause the top-level SDF director to terminate.

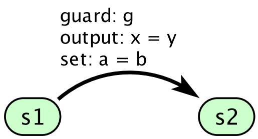
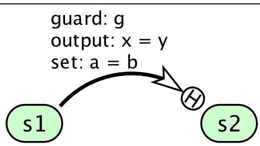
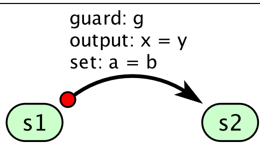
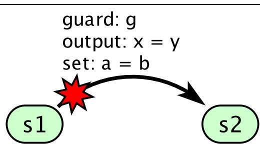
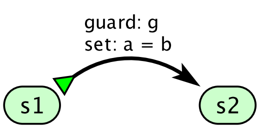
notation	description
	An **ordinary transition**. Upon firing, the refinement of the source state is fired first, and then if the guard g is `true` (or if no guard is specified), then the FSM will choose the transition. It will produce the value y on output port x, overwriting any value that the source state refinement might have produced on the same port. Upon transitioning (in postfire), the actor will set the variable a to have value b, again overwriting any value that the refinement may have assigned to a. Finally, the refinements of state *s2* are initialized. For this reason, these transitions are sometimes called **reset transitions**.
	A **history transition**. This is similar to an ordinary transition, except that when entering state *s2*, the refinements of that state are *not* initialized. On first entry to *s2*, of course, the refinements will have been initialized.
	A **preemptive transition**. If the current state is *s1* and the guard is true, then the state refinement for *s1* will not be iterated prior to the transition.
	An **error transition**. If any refinement of state *s1* throws an exception or a model error, and the guard is true, then this transition will be taken. The output and set actions of the transition can refer to special variables *errorMessage*, *errorClass*, and *errorCause*, as explained in Section 8.2.3.
	A **termination transition**. If all refinements of state *s1* have returned false on postfire, and the guard is true, then the transition is taken. Notice that since it cannot be known until the postfire phase that this transition will be taken, the transition cannot produce outputs. For most domains, postfire is too late to produce outputs. Moreover, this transition has lower priority than all other transitions, including default transitions, because it cannot become enabled until postfire.

Table 8.1: Summary of modal model transitions and their notations. We assume the state refinements are arbitrary Ptolemy II models, each with a director.

8.3 Execution of Modal Models

Execution of a ModalModel is similar to the execution of an FSMActor. In the `fire` method, the ModalModel actor

1. reads inputs;
2. evaluates the guards of preemptive transitions out of the current state;
3. if no preemptive transition is enabled, the actor
   1. fires the refinements of the current state (if any); and
   2. evaluates guards on non-preemptive transitions out of the current state;
3. chooses a transition whose guard evaluates to true, giving preference to preemptive transitions; and
4. executes the output actions of the chosen transition;

In `postfire`, the ModalModel actor

1. postfires the refinements of the current state if they were fired;
2. executes the set actions of the chosen transition;
3. changes the current state to the destination of the chosen transition; and
4. initializes the refinements of the destination state if the transition is a reset transition.

The ModalModel actor makes no persistent state changes in its `fire` method, so as long as the same is true of the refinement directors and actors, a modal model may be used in any domain. Its behavior in each domain may have subtle differences, however, particularly in domains that use fixed-point iteration or when nondeterministic transitions are used. In the next section (Section 8.4), we discuss the use of modal models in various domains.

Note that state refinements are fired before guards on non-preemptive transitions are evaluated. One consequence of this ordering is that the guards can refer to the outputs of the refinements. Thus, whether a transition is taken can depend on how the current refinement reacts to the inputs. The astute reader may have already noticed in the figures here that output ports shown in the FSM do not look like normal output ports (notice the *output* ports in Figures 8.1 and 8.4). In the FSM, these output ports are actually both an output and an input. It serves both of these roles in the FSM. An output of the current state refinement is also an input to the FSM, and guards can refer to this input.

Example 8.7: Figure 8.12 shows a variant of the model in Figure 8.10 that includes a guard that references an output from a refinement. This guard customizes the response when the user types "hello," as shown at the bottom of the figure.

The above example shows that the current state refinement and a transition's output action can both produce outputs on the same output port. Since execution of FSMs is strictly sequential, there is no ambiguity about the result produced on the output of the ModalModel. It is always the last of the values written to the output in the firing. There could even be

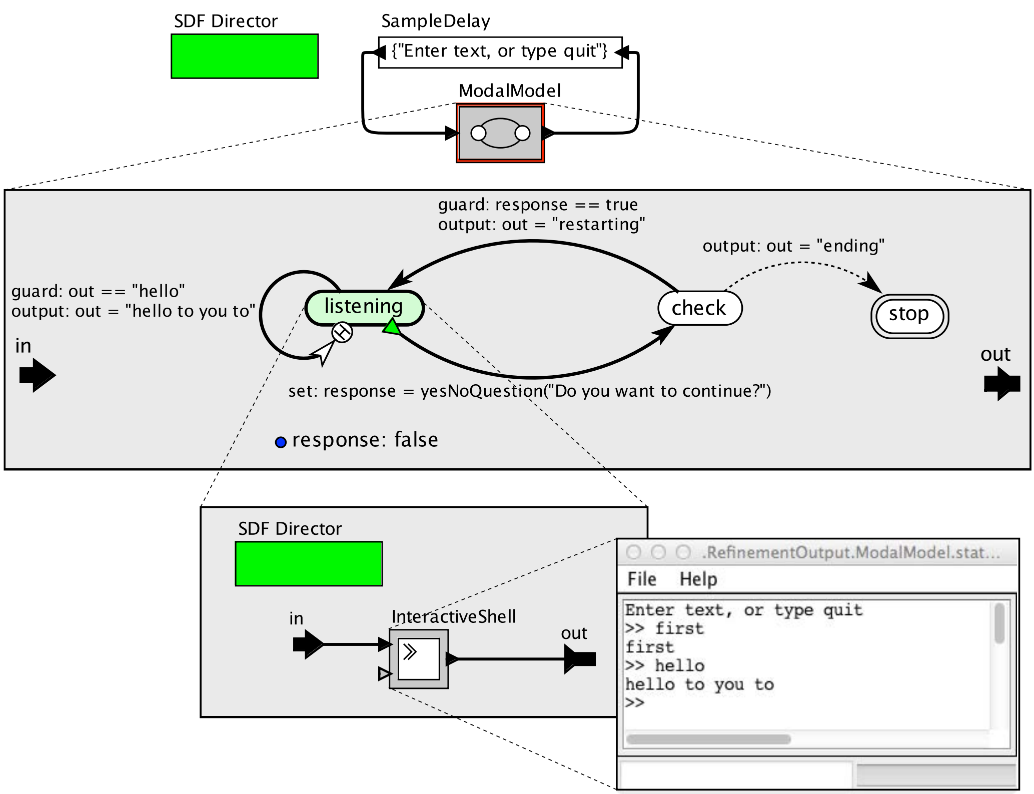


Figure 8.12: A variant of the model in Figure 8.10 that includes a guard that references an output from a refinement. [online]

a chain of immediate transitions, each passing through states that have refinements that write to the same output port, and each transition also writing to the same output port. These writes always occur in a well-defined order, and only the last of these writes will be visible outside the modal model.

8.4 Modal Models and Domains

Our modal model examples so far have mostly used the SDF and DE domains in simple ways. For the DE examples, such as that in Figure 8.1, the modal model fires when there is an event on at least one of the input ports. Some inputs may be absent, and transitions may be triggered by the presence (or absence) of an input. The modal model may or may not produce an output on each output port; if it does not, then the output will be absent. The only real subtlety with using modal models in DE concerns that passage of time, which will be considered below in Section 8.5.

However, the role of modal models in some other domains is not so simple. In this section, we discuss some of the subtleties.

8.4.1 Dataflow and Modal Models

Our SDF examples so far have all been homogeneous SDF, where every actor consumes and produces a single token. When the modal model in these examples fires, all inputs to the modal are present and contain exactly one token. And the firing of the modal model results in one token produced on each output port, with the exception on of Figure 8.9, where an error prevents production of the output token.

With some care, modal models can be used with multirate SDF models, as illustrated by the following example.

Example 8.8: In the example shown in Figure 8.13, the refinements of each of the states require 10 samples in order to fire, because of the SequenceToArray actor. This model alternates between averaging 10 input samples and computing the maximum of 10 input samples. Each firing of the ModalModel executes the current refinement for one iteration, which in this case processes 10 samples. As you can see from the resulting plot, when the input is a sine wave, averaging sequences of 10 samples yields another sine wave, whereas taking the maximum does not.

Probing Further: Concurrent and Hierarchical Machines

An early model for concurrent and hierarchical FSMs is Statecharts, developed by Harel (1987). With Statecharts, Harel introduced the notion of **and states**, where a state machine can be in both states A and B at the same time. On careful examination, the Statecharts model is a concurrent composition of hierarchical FSMs under an SR model of computation. Statecharts are therefore (roughly) equivalent to modal models combining hierarchical FSMs and the SR director in Ptolemy II. Specifically, use of the SR director in a mode refinement to govern concurrent actors, each of which is a state machine, provides a variant of Statecharts. Statecharts were realized in a software tool called **Statemate** (Harel et al., 1990).

Harel's work triggered a flurry of activity, resulting in many variants of the model (von der Beeck, 1994). One variant was adopted to become part of UML (Booch et al., 1998). A particularly elegant version is SyncCharts (André, 1996), which provides a visual syntax to the Esterel synchronous language (Berry and Gonthier, 1992).

One of the key properties of synchronous composition of state machines is that it becomes possible to model a composition of components as a state machine. A straightforward mechanism for doing this results in a state machine whose state space is the cross product of the individual state spaces. More sophisticated mechanisms have been developed, such as interface automata (de Alfaro and Henzinger, 2001).

Hybrid systems (Chapter 9) can also be viewed as modal models, where the concurrency model is a continuous time model (Maler et al., 1992; Henzinger, 2000; Lynch et al., 1996). In the usual formulation, hybrid systems couple FSMs with ordinary differential equations (ODEs), where each state of the FSMs is associated with a particular configuration of ODEs. A variety of software tools have been developed for specifying, simulating, and analyzing hybrid systems (Carloni et al., 2006).

Girault et al. (1999) first showed that FSMs can be combined hierarchically with a variety of concurrent models of computation. They called such compositions ***charts** or **starCharts**, where the star represents a wildcard. Several active research projects continue to explore expressive variants of concurrent state machines. **BIP** (Basu et al., 2006), for example, composes state machines using rendezvous interactions. Alur et al. (1999) give a very nice study of semantic questions around concurrent FSMs, including various complexity questions. Prochnow and von Hanxleden (2007) describe sophisticated techniques for visual editing of concurrent state machines.

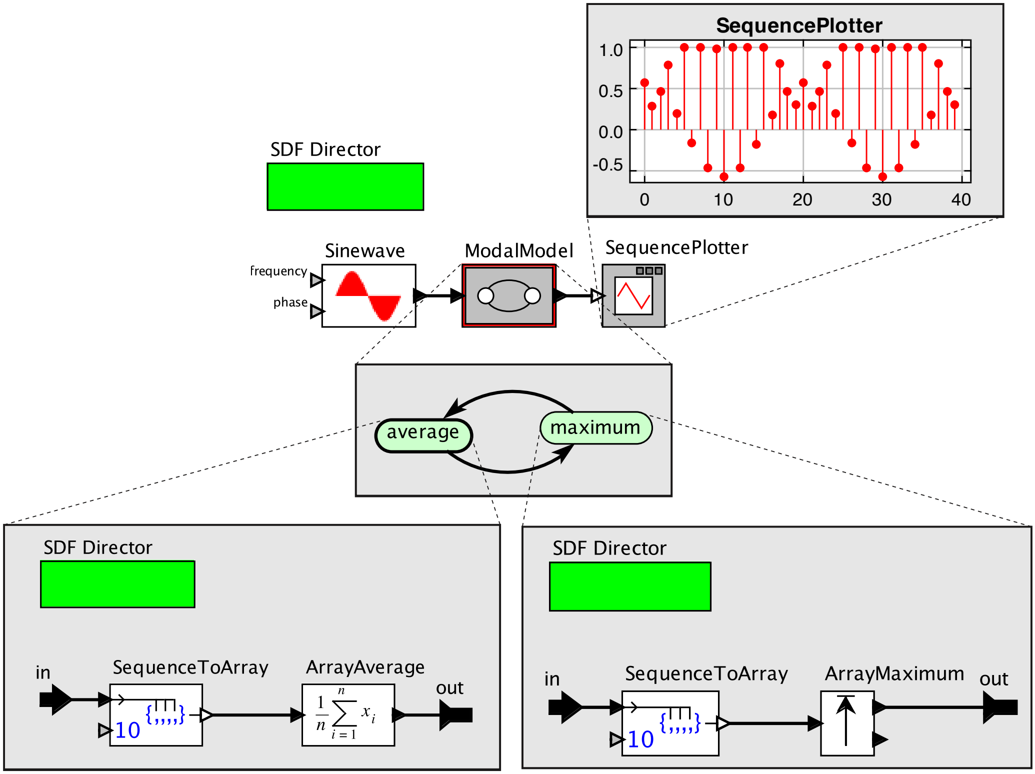


Figure 8.13: An SDF model where the ModalModel requires more than one token on its input in order to fire. [online]

In order for multirate modal models to work, it is necessary to propagate the production and consumption information from the refinements to the top-level SDF director. To do this, you must change the *directorClass* parameter of the ModalModel actor, as shown in Figure 8.14. The default director for a ModalModel makes no assertion about tokens produced or consumed, because it is designed to work with any Ptolemy II director, not specifically to work with SDF. The MultirateFSMDirector, by contrast, is designed to cooperate with SDF and convey production and consumption information across levels of the hierarchy.

In certain circumstances, it is even allowed for the consumption and production profiles of the refinements to differ in different modes. This has to be done very carefully, however. If

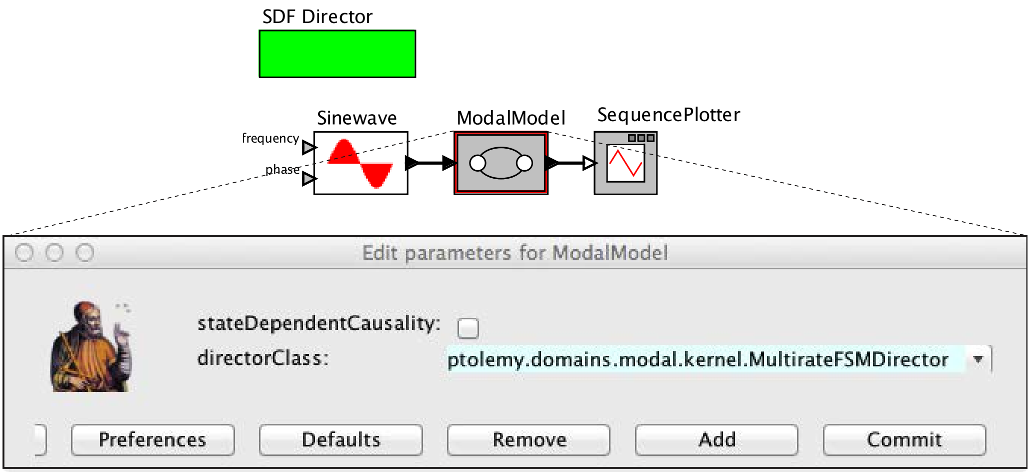


Figure 8.14: In order for the ModalModel to become an SDF actor with non-unit consumption and production on its inputs and outputs, it has to use the specialized MultirateFSMDirector.

different modes have different production and consumption profiles, then the ModalModel actor is actually not an SDF actor. Nevertheless, the SDF director will sometimes tolerate it.

Example 8.9: In Figure 8.13, for example, you can get away with changing the parameters of the SequenceToArray actor so that they differ in the two refinements. Behind the scenes, each time a transition is taken, the SDF director at the top level notices the change in the production consumption profile and compute a new schedule.

This is a major subtlety, however, with relying on the SDF director to recompute the schedule when an actor's production and consumption profile changes. Specifically, the SDF director will only recompute the schedule after a complete iteration has executed (see Section 3.1.1). If the production and consumption profile changes *in the middle of a complete iteration*, then the SDF director may not be able to finish the complete iteration.

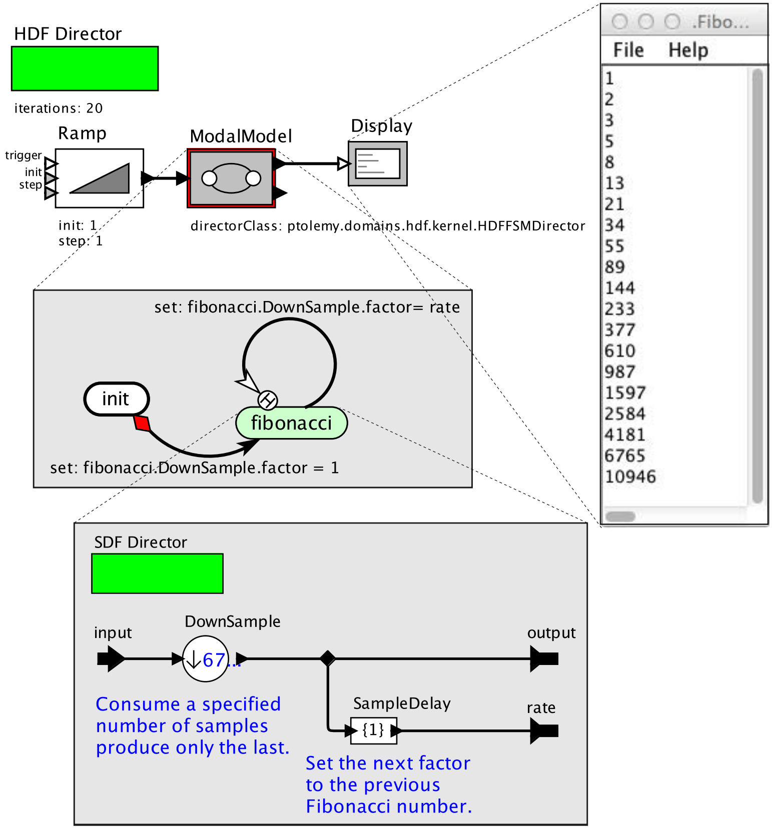


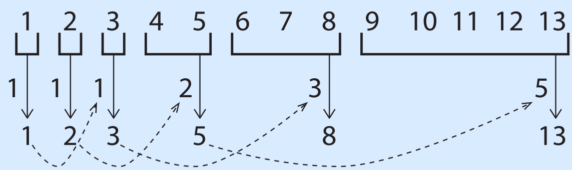
Figure 8.15: A model that calculates the Fibonacci sequence using a heterochronous dataflow model. This model is due to Ye Zhou. [online]

You may see errors about actors being unable to fire due to insufficient input tokens or errors about buffer sizes being inadequate.

The heterochronous dataflow (HDF) director provides a proper way to use multirate modal models with SDF. With this director, it is necessary to select the **HDFFSMDirector** for the *directorClass* of the ModalModel. These two directors cooperate to ensure that transitions of the FSM are taken *only after each complete iteration*. This combination is very expressive, as illustrated by the following examples.

Example 8.10: The model in Figure 8.15 uses HDF to calculate the **Fibonacci** sequence. In the Fibonacci sequence, each number is the sum of the previous two numbers. One way to generate such a sequence is to extract the Fibonacci numbers from a counting sequence (the natural numbers) by sampling each number that is a Fibonacci number. This can be done by a DownSample actor where the n-th Fibonacci number is generated by downsampling with a factor given by the $(n-2)$th Fibonacci number.

The calculation is illustrated in the following figure:



The top row shows the counting sequence from which we select the Fibonacci numbers. The downward arrows show the amount of downsampling required at each stage to get the next Fibonacci number. A downsampling operation simply consumes a fixed number of tokens and outputs only the last one. The first two downsampling factors are fixed at 1, but after that, the downsampling factor is itself a previously selected Fibonacci number.

In the model, the FSM changes the *factor* parameter of a DownSample actor each time it fires. The HDF director calculates a new schedule each time the downsampling rate is changed, and the new schedule outputs the next Fibonacci number.

Example 8.11: Another interesting example is shown in Figure 8.16. In this example, two increasing sequences of numbers are merged into one increasing sequence. In the initial state, the ModalModel consumes one token from each input and outputs the smaller of the two on its upper output port, and the larger of the two on its lower output port. The smaller, of course, is the first token of the merged sequence. The larger of the two is fed back to the input port named *previous* of the ModalModel.

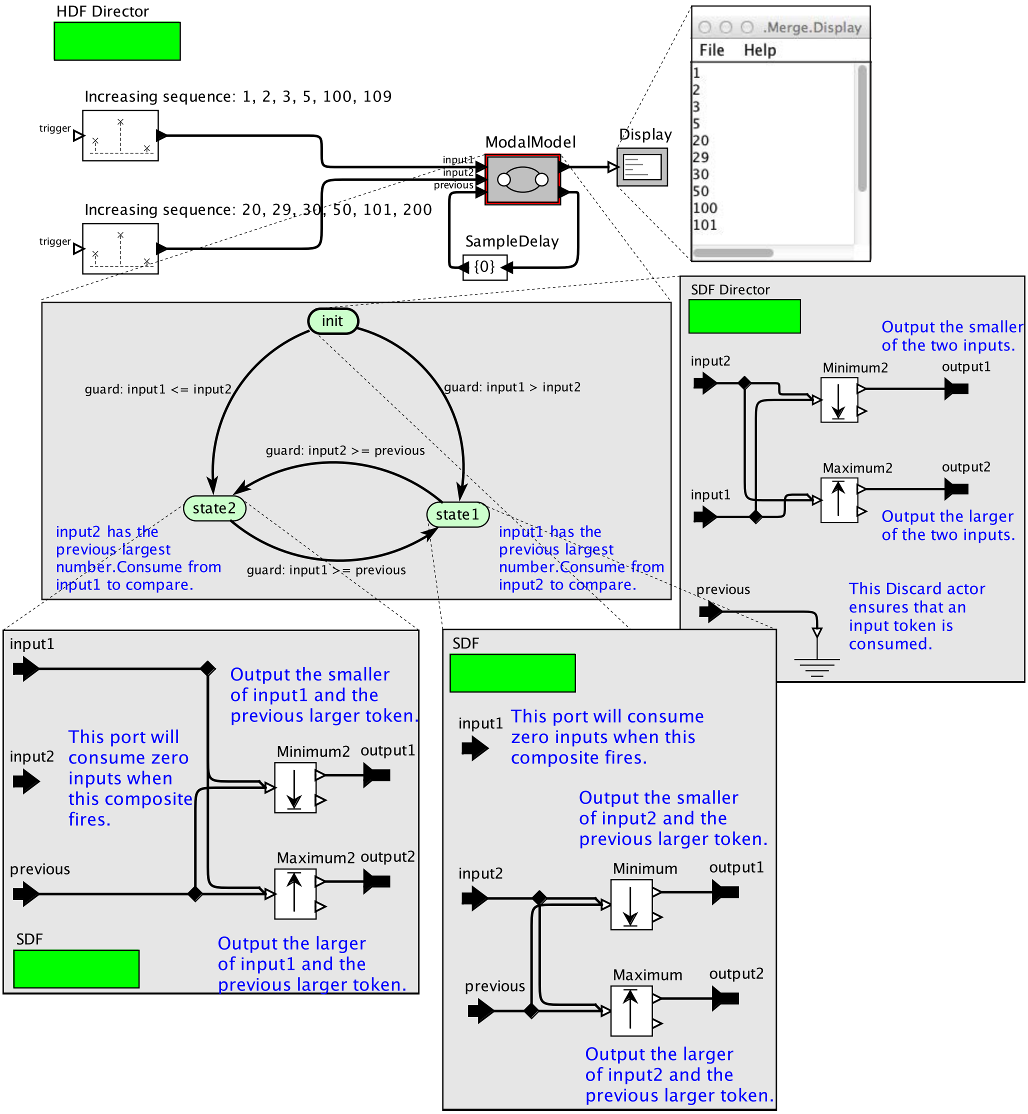


Figure 8.16: A heterochronous dataflow model that merges two numerically increasing streams into one numerically increasing stream (due to Ye Zhou and Brian Vogel). [online]

If the larger input came from *input1*, then the FSM transitions to *state1*. The refinement of this state does not read a token at all from *input1* (its consumption parameter will be zero). Instead, it reads one from *input2* and compares it against the value that was fed back.

If instead the initial larger input came from *input2*, then the FSM transitions to *state2*, which reads from *input1* and compares that input against the value that was fed back.

These examples demonstrate that HDF allows consumption and production rates to vary dynamically. In each case, the production and consumption profiles of the modal models are determined by the model inside the current state refinement.

The HDF model of computation was introduced by Girault et al. (1999), and the primary author of the HDF director is Ye Zhou. It has many interesting properties. Like SDF, HDF models can be statically analyzed for deadlock and bounded buffers. But the MoC is much more flexible than SDF because data-dependent production and consumption rates are allowed. In order to use it, however, the model builder has to fully understand the notion of a complete iteration, because this notion will constrain when transitions are taken in the FSM.

8.4.2 Synchronous-Reactive and Modal Models

The SR domain, explained in Chapter 5, can use modal models in very interesting ways. The key subtlety, compared with DE or dataflow, is that SR models may have feedback loops that require iterative convergence to a fixed point. An example of such a feedback loop using FSMs is given in Section 6.4.

The key issue, then, is that when a ModalModel actor fires in the SR domain, some of its inputs may be unknown. This not the same as being absent. When an input is unknown, we don't know whether it is absent or present.

In order for modal models to be useful in feedback loops, it is important that the modal model be able to assert outputs even if some inputs are unknown. Asserting an output means specifying that it is either absent or present, and if it is present, optionally giving it a value. But the modal model has to be very careful to make sure that it does not make assertions about outputs that become incorrect when the inputs become known. This constraint ensures that the actor is monotonic.

If a ModalModel actor fires with some inputs unknown, then it must make a distinction between a transition that is known to not be enabled and one where it is not known whether it is enabled. If the guard refers to unknown inputs, then it cannot be known whether a transition is enabled. This makes it challenging, in particular, to assert that outputs are absent. It is not enough, for example, that no transition be enabled in the current state. Instead, the modal model has to determine that every transition that could potentially make the output present is *known to be not enabled*.

This constraint becomes subtle with chains of immediate transitions, because all chains emanating from the current state have to be considered. If in any transition in such a chain has a guard that is not known to be true or false, then the possible outputs of all subsequent transitions have to be considered. If there are state refinements in chains of immediate transitions, then it becomes extremely difficult to assert that an output is absent when not all inputs are known.

Because of these subtleties, we recommend avoiding using modal models in feedback loops that rely on the modal model being able to assert outputs without knowing inputs. The resulting models can be extremely difficult to understand, so that even recognizing correct behavior becomes challenging.

8.4.3 Process Networks and Rendezvous

Modal models can be used with PN and Rendezvous, but only in a rather simple way. When a ModalModel actor fires, it will read from each input port (in top-to-bottom order), which in each of these domains will cause the actor to block until an input is available. Thus, in both cases, a modal model always consumes exactly one token from each input. Whether it produces a token on the output, however, will depend on the FSM.

8.5 Time in Modal Models

Many Ptolemy II directors implement a timed model of computation. The ModalModel actor and FSMActor are themselves untimed, but they include features to support their use in timed domains.

The FSMs we have described so far are **reactive**, meaning that they only produce outputs in reaction to inputs. In a timed domain, the inputs have time stamps. For a reactive FSM, the time stamps of the outputs are the same as the time stamps of the inputs. The FSM appears to be reacting instantaneously.

In a timed domain, it is also possible to define spontaneous FSM and modal models. A **spontaneous FSM** or **spontaneous modal model** is one that produces outputs even when inputs are absent.

Example 8.12: The model shown in Figure 8.17 uses the `timeout` function, described in Section 6.2.1, in the guard expression to trigger a transition every 1.0 time units. This is a spontaneous FSM with no input ports at all.

Example 8.13: The model in Figure 8.18 switches between two modes every 2.5 time units. In the *regular* mode it generates a regularly spaced clock signal with period 1.0 (and with value 1, the default output value for DiscreteClock). In the

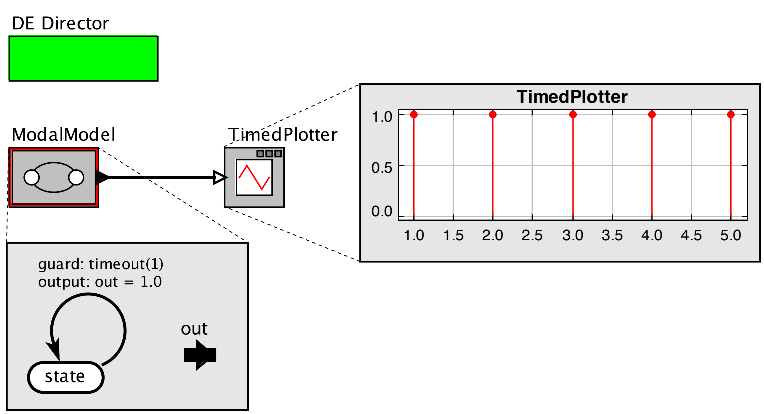


Figure 8.17: A spontaneous FSM, which produces output events that are not triggered by input events. [online]

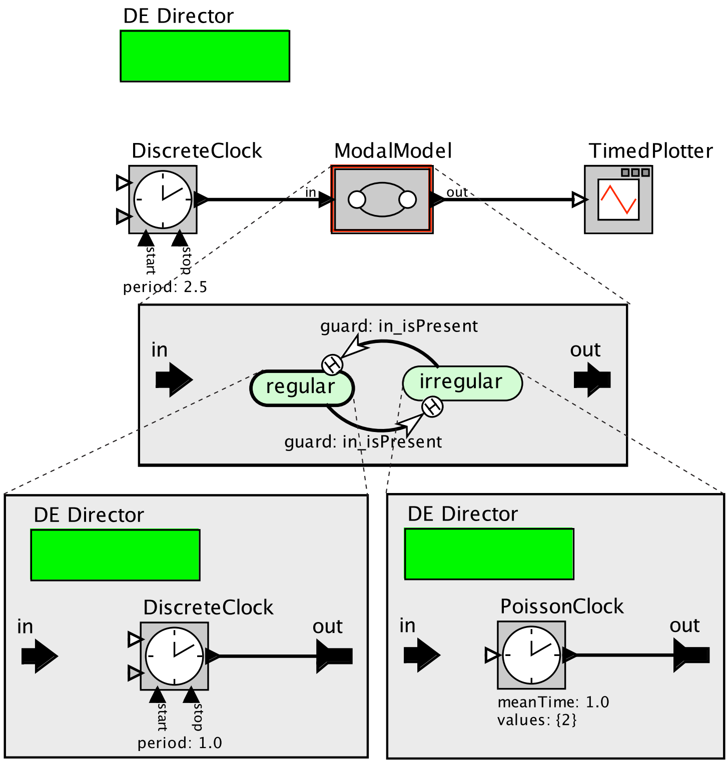


Figure 8.18: Another spontaneous modal model, which produces output events that are not triggered by input events. [online]

irregular mode, it generates randomly spaced events using a PoissonClock actor with a mean time between events set to 1.0 and value set to 2. The result of a typical run is plotted in Figure 8.19, with a shaded background showing the times during which it is in the two modes. The output events from the ModalModel are spontaneous; they are not necessarily produced in reaction to input events.

This example illustrates a number of subtle points about the use of time in modal models. In Figure 8.19, we see that two events are produced at time zero: one with a value of 1, and one with a value of 2. Why? The initial state is *regular*, and the execution policy

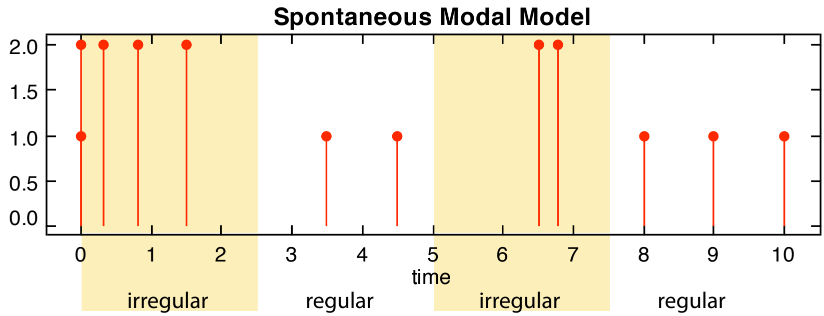


Figure 8.19: A plot of the output from one run of the model in Figure 8.18.

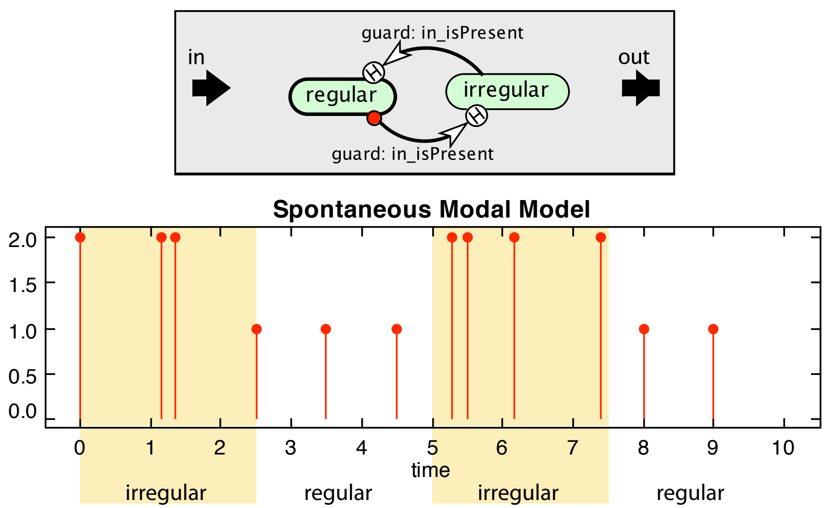


Figure 8.20: A variant of Figure 8.18 where a preemptive transition prevents the initial firing of the DiscreteClock.

described in section 8.3 explains that the refinement of that initial state is fired before guards are evaluated. That firing produces the first output of the DiscreteClock, with value 1. If we had instead used a preemptive transition, as shown in Figure 8.20, then that first output event would not appear.

The second event in Figure 8.19 (with value 2) at time zero is produced because the PoissonClock, by default, produces an event at the time when execution starts. This event is produced in the second iteration of the ModalModel, after entering the *irregular* state. Although the event has the same time stamp as the first event (both occur at time zero), they have a well-defined ordering. The event with value 1 appears before the event with value 2.

As previously described, in Ptolemy II, the value of time is represented by a pair of numbers, $(t, n) \in \mathbb{R} \times \mathbb{N}$, rather than a single number (see Section 1.7). The first of these numbers, t, is called the time stamp. It approximates a real number (it is a quantized real number with a specified precision). We interpret the time stamp t to represent the number of seconds (or any other time unit) that have elapsed since the start of execution of the model. The second of these numbers, n, is called the microstep, and it represents a sequence number for events that occur at the same time stamp. In our example, the first event (with value 1) has tag $(0, 0)$, and the second event (with value 2) has tag $(0, 1)$. If we had set the *fireAtStart* parameter of the PoissonClock actor to `false`, then the second event would not occur.

Notice further that the DiscreteClock actor in the *regular* mode refinement has period 1.0, but produces events at times 0.0, 3.5, and 4.5, 8.0, 9.0, etc.. These are not multiples of 1.0 from the start time of the execution. Why?

The modal begins in the *regular* mode, but spends zero time there. It immediately transitions to the *irregular* mode. Hence, at time 0.0, the *regular* mode becomes inactive. While it is inactive, its local time does not advance. It becomes active again at global time 2.5, but its local time is still 0.0. Therefore, it has to wait one more time unit, until time 3.5, to produce the next output.

This notion of **local time** is important to understanding timed modal models. Very simply, local time stands still while a mode is inactive. Actors that refer to time, such as TimedPlotter and CurrentTime, can base their responses on either local time or global time, as specified in the parameter *useLocalTime* (which defaults to `false`). If no actor accesses global time, however, then a mode refinement will be completely unaware that it was ever suspended. It does not appear as if time has elapsed.

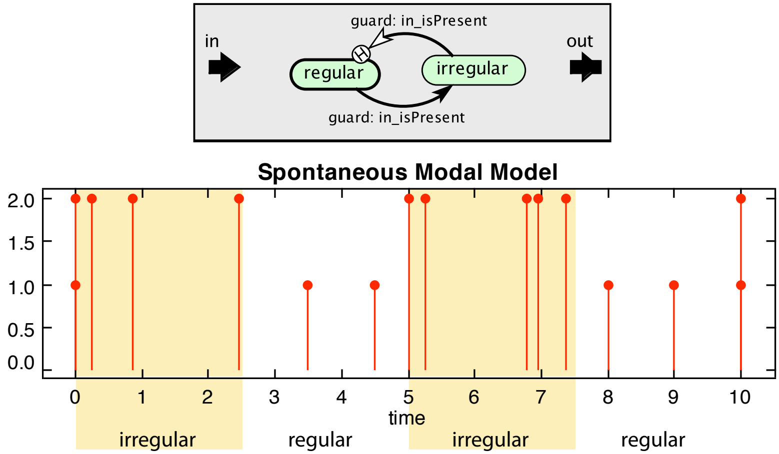


Figure 8.21: A variant of Figure 8.18 in which a reset transition causes the PoissonClock to produce events when the *irregular* mode is reactivated.

Another interesting property of the output of this model is that no event is produced at time 5.0, when the *irregular* mode becomes active again. This behavior follows from the same principle described above. The *irregular* mode became inactive at time 2.5, and hence, from time 2.5 to 5.0, its local notion of time has not advanced. When it becomes active again at time 5.0, it resumes waiting for the right time (local time) to produce the next output from the PoissonClock actor.†

If an event is desired at time 5.0 (when the irregular mode becomes active) then a reset transition can be used, as shown in Figure 8.21. The `initialize` method of the PoissonClock causes an output event to be produced *at the time of the initialization*. A reset transition causes local time to match the environment time (where environment time is the time of the model in which the modal model resides; these distinct time values are discussed further in the next section). The time lag between local time and environment time goes to zero.

†Interestingly, because of the memoryless property of a Poisson process, the time to the next event after becoming active is statistically identical to the time between events of the Poisson process. But this fact has little to do with the semantics of modal models.

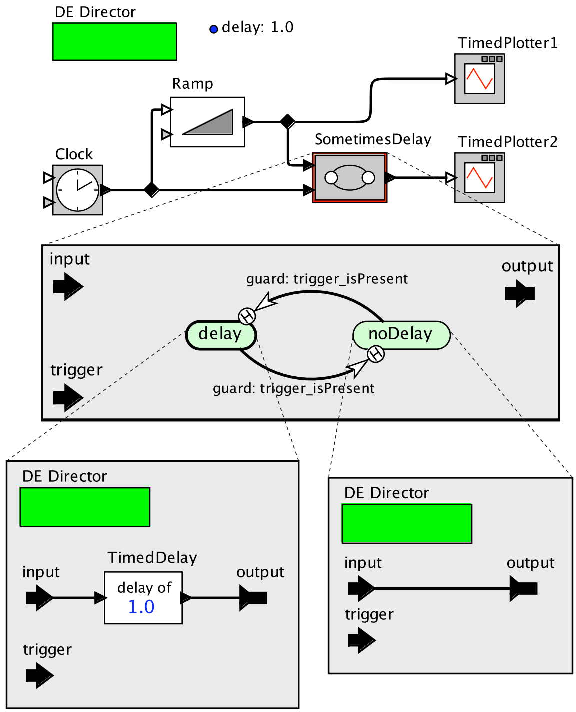


Figure 8.22: A modal model that switches between delaying the input by one time unit and not delaying it. [online]

8.5.1 Time Delays in Modal Models

The use of time delays in a modal model can produce several interesting effects, as shown in the example below.

Example 8.14: Figure 8.22 shows a model that produces a counting sequence of events spaced one time unit apart. The model uses two modes, delay and non-Delay, to delay every other event by one time unit. In the *delay* mode, a TimeDelay actor imposes a delay of one time unit. In the *noDelay* mode, the input is sent directly to the output without delay. The result of executing this model is shown in Figure 8.23. Notice that the value 0 is produced at time 2. Why?

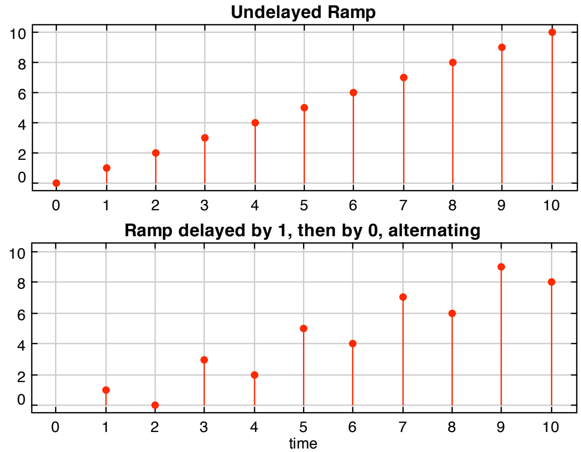


Figure 8.23: The result of executing the model in Figure 8.22.

The model begins in the *delay* mode, which receives the first input. This input has value 0. However, the modal model transitions immediately out of that mode to the *noDelay* mode, with zero elapsed time. The *delay* mode becomes active again at time 1, but its local time is still time 0. Therefore, it must delay the input with value 0 by one time unit, to time 2. Its output is produced at time 2, just before transitioning to the *noDelay* mode again.

8.5.2 Local Time and Environment Time

As shown in the previous examples, modal models may have complex behaviors, particularly when used in timed domains. It is useful to step back and ponder the principles that govern the design choices in the Ptolemy II implementation of modal models. The key idea behind a mode is that it specifies a portion of the system that is active only part of the time. When it is inactive, does it cease to exist? Does time pass? Can its state evolve? These are not easy questions to answer because the desired behavior depends on the application.

In a modal model, there are potentially four distinct times that can affect the behavior of the model: local time, environment time, global time, and real time. **Local time** is the time within the mode (or other local actor). **Environment time** is the time within the model that contains the modal model. **Global time** is the model time at the top level of a hierarchical model. **Real time** is the wall-clock time outside the computer executing the model.

In Ptolemy II, the guiding principle is that when a mode is inactive, local time stands still, while environment time (and global time) passes. An inactive mode is therefore in a state of suspended animation. Local time within a mode will lag the time in its environment by an **accumulated suspend time** or **lag** that is non-decreasing.

The time lag in a mode refinement is initially the difference between the start time of the environment of the modal model and the start time of the mode refinement (normally this difference is zero, but it can be non-zero, as explained below in Section 8.5.3). The lag increases each time the mode becomes inactive, but within the mode, time seems uninterrupted.

When an event crosses a hierarchical boundary into or out of the mode, its time stamp is adjusted by the amount of the lag. That is, when a mode refinement produces an output event, if the local time of that event is t, then the time of event that appears at the output of the modal model is $t + \tau$, where τ is the accumulated suspend time.

A key observation is that when a submodel is inactive, it does not behave in the same manner as a submodel the receives inputs and then ignores them. This point is illustrated by the model of Figure 8.24. This model shows two instances of DiscreteClock, labeled DiscreteClock1 and DiscreteClock2, which have the same parameter values. DiscreteClock2 is inside a modal model labeled ModalClock, and DiscreteClock1 is not inside a modal model. The output of DiscreteClock1 is filtered by a modal model labeled ModalFilter that selectively passes the input to the output. The two modal models are controlled by the same ControlClock, which determines when they switch between the *active* and *inactive* states. Three plots are shown. The top plot is the output of DiscreteClock1. The middle plot is the result of switching between observing and not observing the output of DiscreteClock1. The bottom plot is the result of activating and deactivating DiscreteClock2, which is otherwise identical to DiscreteClock1.

The DiscreteClock actors in this example are set to produce a sequence of values, 1, 2, 3, 4, cyclically. Consequently, in addition to being timed, these actors have state, since they need to remember the last output value in order to produce the next output value. When

DiscreteClock2 is inactive, its state does not change, and time does not advance. Thus, when it becomes active again, it simply resumes where it left off.

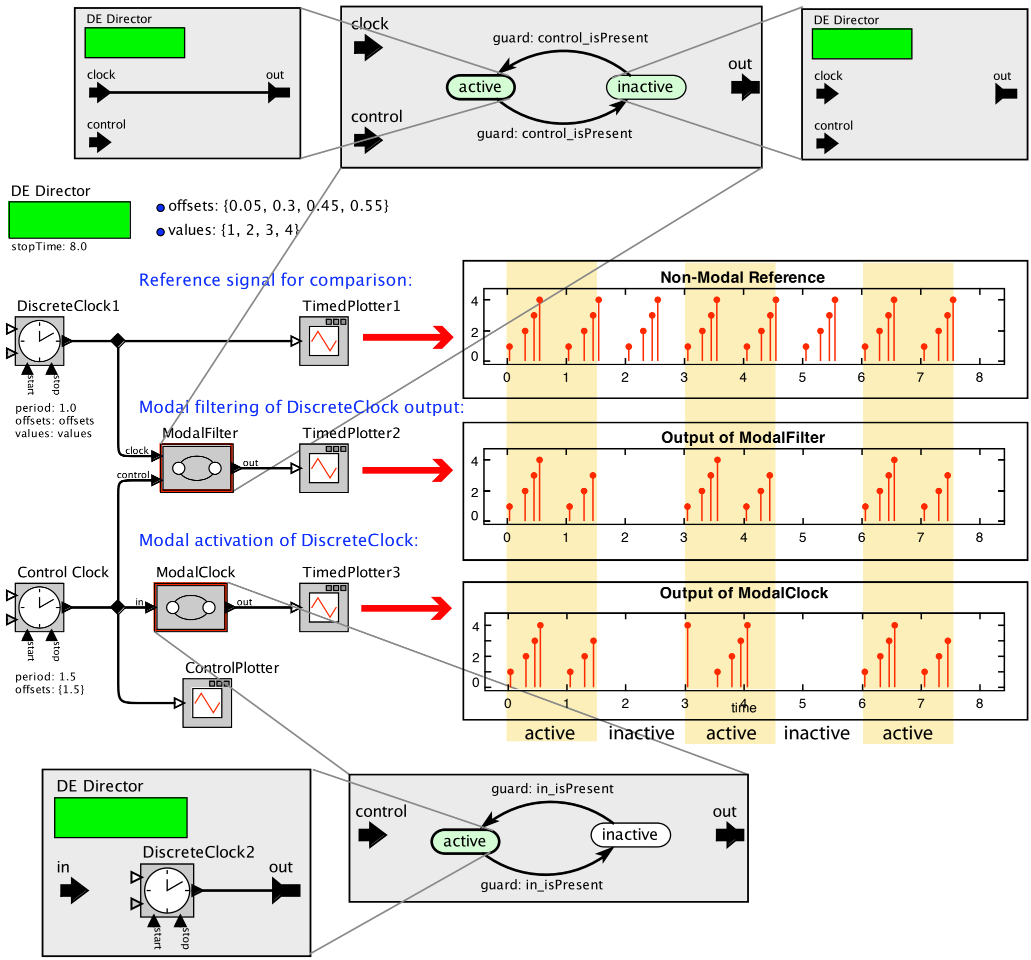


Figure 8.24: A model that illustrates that putting a timed actor such as DiscreteClock inside a modal model is not the same as switching between observing and not observing its output. [online]

8.5.3 Start Time in Mode Refinements

Usually, when we execute a timed model, we want it to execute over a specified time interval, from a start time to a stop time. By default, execution starts at time zero, but the *startTime* parameter of the director can specify a different start time.

When a DE model is inside a mode refinement, however, by default, the start time in the submodel is the time at which is initialized. Normally, this is the same as the start time of the enclosing model, but when a reset transition is used, then the submodel may be reinitialized at an arbitrary time.

When a submodel is reinitialized by a reset transition, occasionally it is useful to restart execution at a particular time in the past. This can be accomplished by changing the *startTime* parameter of the inside DEDirector to something other than the default (which is blank, interpreted as the time of initialization).

Example 8.15: This use of the *startTime* parameter is illustrated in Figure 8.25, which implements a **resettable timer**. This example has a modal model with a single mode and a single reset transition. The *startTime* of the inside DEDirector is set to 0.0, so that each time the reset transition is taken, the execution of the submodel begins again at time 0.0.

In this example, a PoissonClock generates random reset events that cause the reset transition to be taken. The refinement of the mode has a SingleEvent actor that is set to produce an event at time 0.5 with value 2.0. As shown in the plot, this modal model produces an output event 0.5 time units after receiving an input event, unless it receives a second input event during the 0.5 time unit interval. The second event resets the timer to start over. Thus, the event at time 1.1 does not result in any output because the event at time 1.4 resets the timer.

When a reset transition is taken and the destination mode refinement has a specified *startTime*, the accumulated suspend time increases by t, where t is the current time of the enclosing model. After the reset transition is taken, the lag between local time and global time is larger by t than it was before the transition was taken.

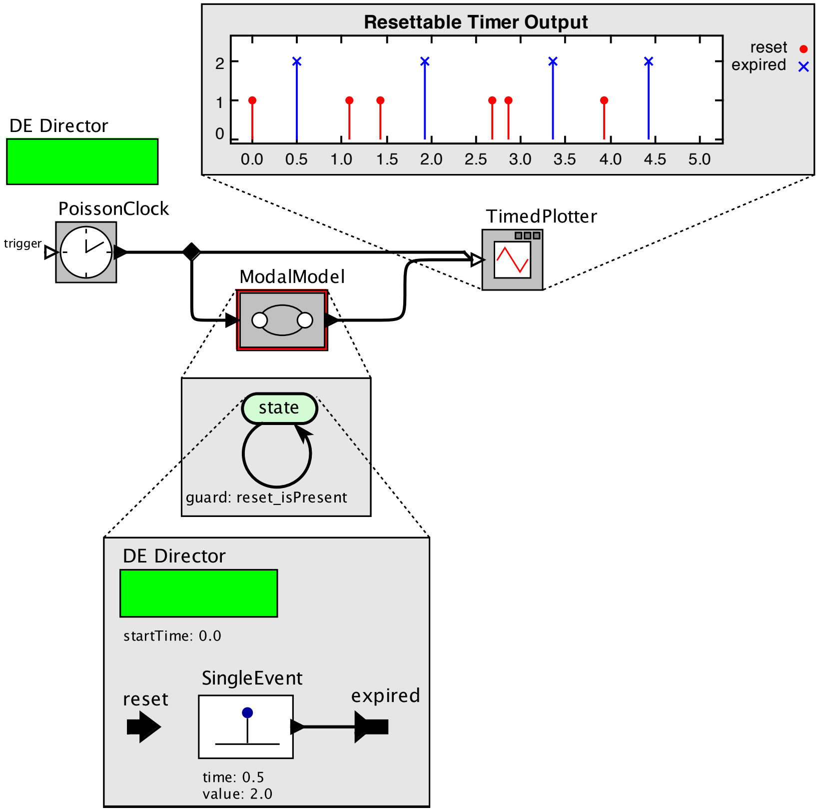


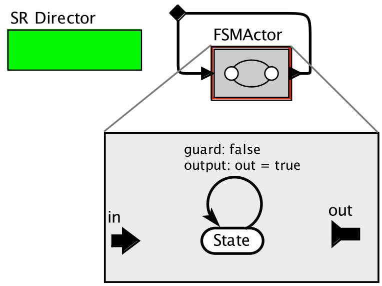
Figure 8.25: A resettable timer implemented by using a reset transition to restart a submodel at time zero. [online]

8.6 Summary

FSMs and modal models in Ptolemy II provide a very expressive way to build up complex modal behaviors. As a consequence of this expressiveness, it takes some practice to learn to use them well. This chapter is intended to provide a reasonable starting point. Readers who wish to probe further are encouraged to examine the documentation for the Java classes that implement these mechanisms. Many of these documents are accessible when running Vergil by right clicking and selecting `Documentation`.

Exercises

1. In the following model, the only signal (going from the output of FSMActor back to its input) has value `absent` at all ticks.



   Explain why this is correct.

2. Construct a variant of the example in Figure 8.1 where the *clean* and *noisy* states share the same refinement, yet the behavior is the same.

3. This problem explores the use of the SDF model of computation together with modal models to improve expressiveness. In particular, you are to implement a simple run-length coder using no director other than SDF, leveraging modal models with state refinements. Specifically, given an input sequence, such as

$$(1, 1, 2, 3, 3, 3, 3, 4, 4, 4)$$

   you are to display a sequence of pairs (i, n), where i is a number from the input sequence and n is the number of times that number is repeated consecutively. For the above sequence, your output should be

$$((1, 2), (2, 1), (3, 4), (4, 3)).$$

   Make sure your solution conforms with SDF semantics. Do not use the non-SDF techniques of section 3.2.3. Note that this pattern arises commonly in many coding applications, including image and video coding.

9 Continuous-Time Models

Janette Cardoso, Edward A. Lee, Jie Liu, and Haiyang Zheng

Contents

Continuous-time models are realized using the Ptolemy II **continuous-time** (**CT**) domain (also called the **Continuous** domain), which models physical processes. This domain is particularly useful for cyber-physical systems, which are characterized by their mixture of computational and physical processes.

The CT domain conceptually models time as a continuum. It exploits the superdense time model in Ptolemy II to process signals with discontinuities, signals that mix discrete and continuous portions, and purely discrete signals. The resulting models can be combined hierarchically with discrete event models, and modal models can be used to develop hybrid systems.

9.1 Ordinary Differential Equations

The continuous dynamics of physical processes are represented using **ordinary differential equations** (**ODE**s), which are differential equations over a time variable. The Ptolemy II models of continuous-time systems are similar to those used in Simulink (from The MathWorks), but Ptolemy's use of superdense time provides cleaner modeling of mixed signal and hybrid systems (Lee and Zheng, 2007). This section focuses on how continuous dynamics are specified in a Ptolemy II model and how the Continuous director executes the resulting models.

9.1.1 Integrator

In Ptolemy II, differential equations are represented using **Integrator** actors in feedback loops. At time t, the output of an Integrator actor is given by

$$x(t) = x_0 + \int_{t_0}^{t} \dot{x}(\tau) d\tau, \tag{9.1}$$

where x_0 is the *initialState* of the Integrator, t_0 is the *startTime* of the director, and $\dot{x}$ is the input signal to the Integrator. Note that since the output x of the Integrator is the integral of its input $\dot{x}$, then at any given time, the input $\dot{x}$ is the derivative of the output x,

$$\dot{x}(t) = \frac{d}{dt} x(t). \tag{9.2}$$

Thus, the system describes either an integral equation or a differential equation, depending on which of these two forms you use. ODEs can be represented using Integrator actors, as illustrated by the following example.

Example 9.1: The well-known **Lorenz attractor** is a non-linear feedback system that exhibits a style of chaotic behavior known as a **strange attractor**. The model in Figure 9.1 is a block diagram representation of the set of nonlinear ODEs that govern the behavior of this system. Let the output of the top integrator be x_1, the output of the middle integrator be x_2, and the output of the bottom integrator be x_3.

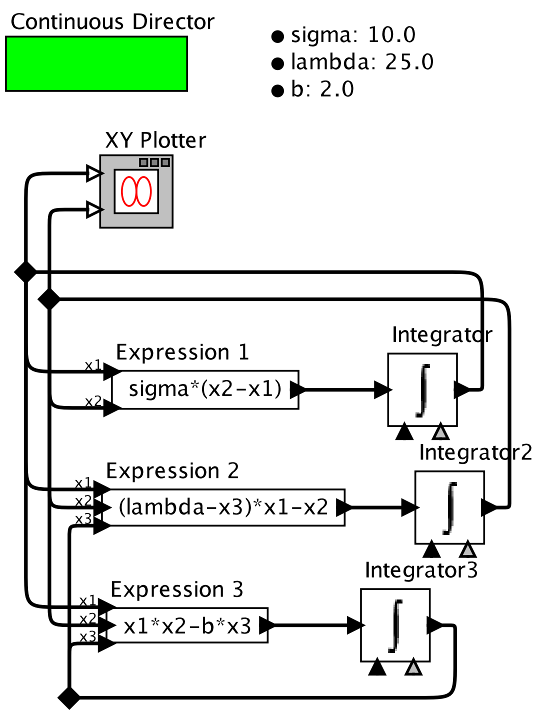


Figure 9.1: A model describing a set of nonlinear ordinary differential equations. [online]

Then the equations described by Figure 9.1 are

$$\begin{aligned}\dot{x}_1(t) &= \sigma(x_2(t) - x_1(t)) \\ \dot{x}_2(t) &= (\lambda - x_3(t))x_1(t) - x_2(t) \\ \dot{x}_3(t) &= x_1(t)x_2(t) - bx_3(t)\end{aligned} \tag{9.3}$$

where σ, λ, and b are real-valued constants. For each equation, the expression on the right side of the equals sign is implemented by an Expression actor, whose icon shows the expression. Each expression refers to parameters (such as *lambda* for λ and *sigma* for σ) and input ports of the actor (such as x1 for x_1 and x2 for x_2). The expression in each Expression actor can be edited by double clicking on the actor, and the parameter values can be edited by double clicking on the parameters, which are shown next to bullets at the top.

The three integrators specify initial values for x_1, x_2, and x_3; these values can be changed by double-clicking on the corresponding Integrator icon. In this example, all three initial values are set to 1.0 (not shown in the figure).

The Continuous Director, shown at the upper left, manages the simulation of the model. It contains a sophisticated ODE solver with several key parameters. These parameters can be accessed by double clicking on the director, which results in the dialog box shown in Figure 9.2.

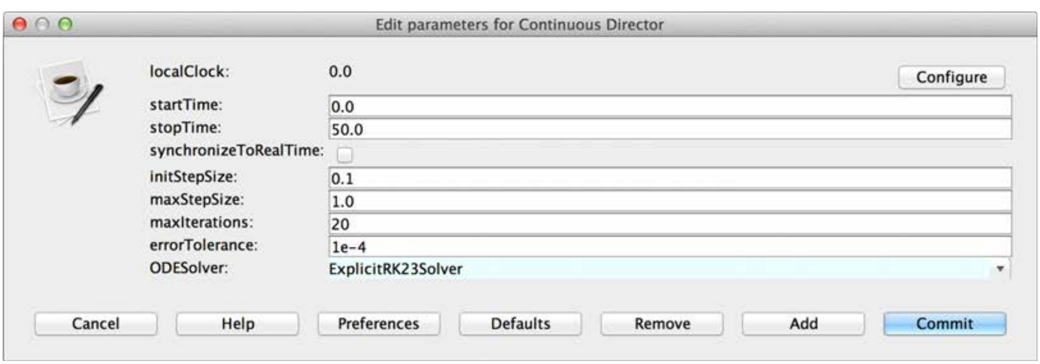


Figure 9.2: Dialog box showing director parameters for the model in Figure 9.1.

The simplest parameters are *startTime* and *stopTime*, which define the region of the time line over which the simulation will execute. The effects of the other parameters are explored in Exercise 1.

The output of the Lorenz model is shown in Figure 9.3. The XY Plotter displays $x_1(t)$ vs. $x_2(t)$ for values of t in between *startTime* and *stopTime*.

Like the Lorenz model, many continuous-time models contain integrators in feedback loops. Instead of using Integrator actors, however, it is possible to use more elaborate blocks that implement linear and non-linear dynamics, as described below.

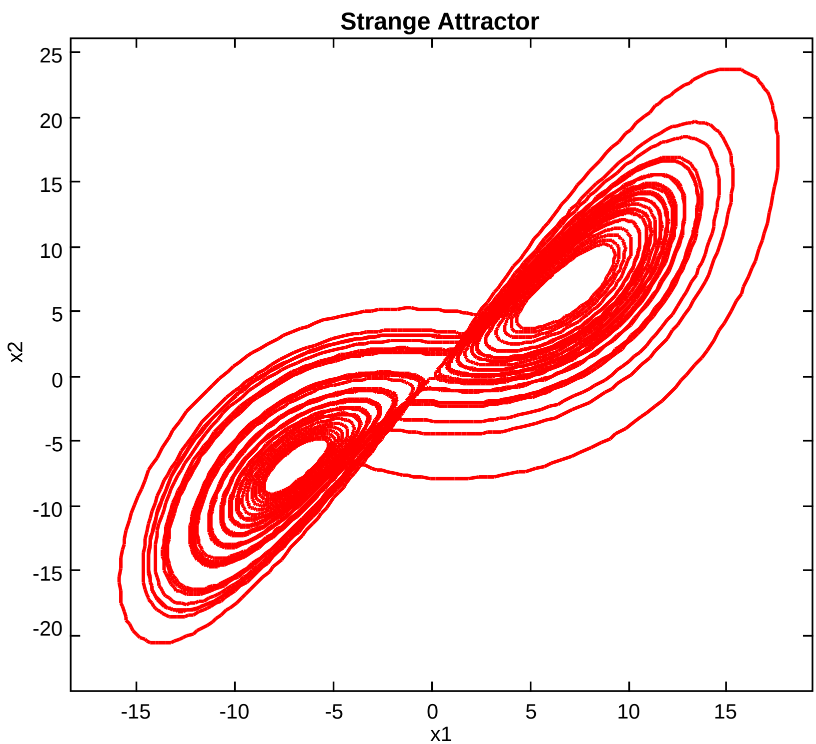


Figure 9.3: Result of running the Lorenz model.

9.1.2 Transfer Functions

When representing continuous-time systems, it is often more convenient to use a higher-level description than individual integrators. For example, for linear time invariant (**LTI**) systems, it is common to characterize their input output behavior in terms of a **transfer function**, which is the Laplace transform of the impulse response. Specifically, for an input x and output y, the transfer function may be given as a function of a complex variable s:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{b_1 s^{m-1} + b_2 s^{m-2} + \cdots b_m}{a_1 s^{n-1} + a_2 s^{n-2} + \cdots a_n} \tag{9.4}$$

where Y and X are the Laplace transforms of y and x, respectively. The number n of denominator coefficients is strictly greater than the number m of numerator coefficients. A system that is described by a transfer function can be constructed using individual integrators, but is more conveniently implemented using the **ContinuousTransferFunction** actor, as illustrated by the following example.

Example 9.2: Consider the model in Figure 9.4, which produces the plot in Figure 9.5. This model generates a square wave using a ContinuousClock actor (see sidebar on page 326) and feeds that square wave into a ContinuousTransferFunction actor. The transfer function implemented by the ContinuousTransferFunction actor is given by the following:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{1}{0.001s^2 + 0.01s + 1}.$$

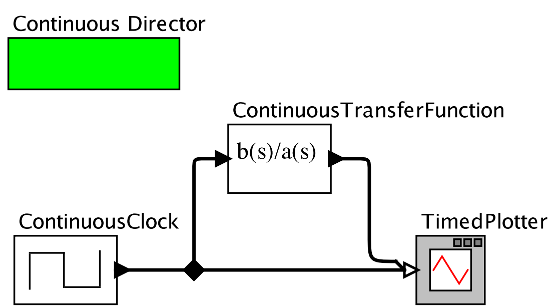


Figure 9.4: Model illustrating the use of ContinuousTransferFunction. [online]

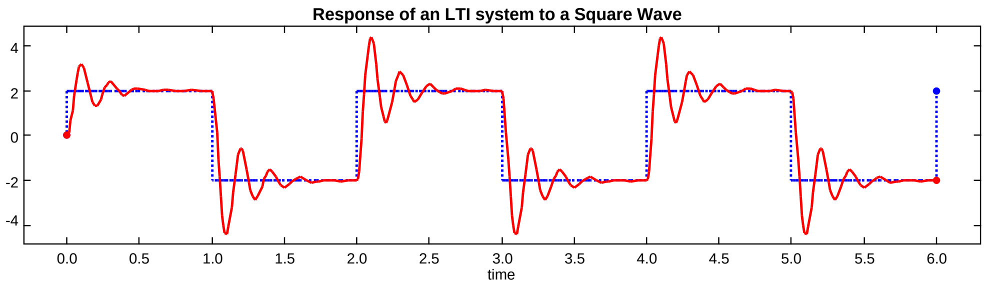


Figure 9.5: Result of running the model in Figure 9.4.

Comparing the equation above to Equation (9.4), we see that $m = 1$ and $n = 3$, with additional parameters as follows:

$$\begin{aligned} b_1 &= 1 \\ a_1 &= 0.001, \quad a_2 = 0.01, \quad a_3 = 1.0 \end{aligned}$$

The parameters of the actor are therefore set to

$$\begin{aligned} \textit{numerator} &= \{\texttt{1.0}\} \\ \textit{denominator} &= \{\texttt{0.001, 0.01, 1.0}\} \end{aligned}$$

An equivalent model constructed with individual integrators is shown in Figure 9.6 (see Exercise 2 to explore why these are equivalent).

The previous example shows that a complex network of integrators, gains, and adders can be represented compactly using the ContinuousTransferFunction actor. In fact, this actor uses the specified parameter values to construct a hierarchical model similar to the one shown in Figure 9.6. It is possible to view this hierarchical model by right clicking on the ContinuousTransferFunction actor and selecting `Open Actor`. (Select [`Graph`→ `Automatic Layout`] so that the actors are shown in a more readable layout.) ContinuousTransferFunction is an example of a higher-order actor, where the parameters specify an actor network that implements the functionality of the actor.

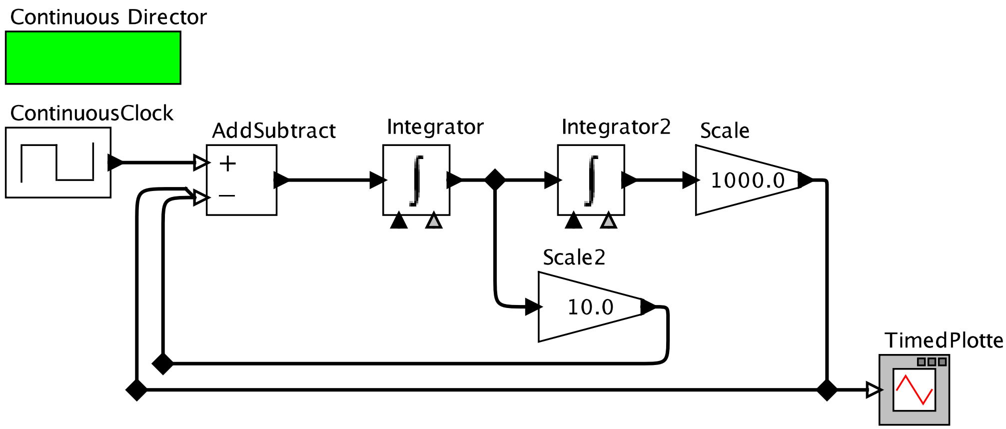


Figure 9.6: A model equivalent to the one in Figure 9.4 assuming the parameters of Example 9.2. [online]

The ContinuousTransferFunction actor and other actors that support higher-level descriptions of dynamics are summarized in the sidebar on page 327.

9.1.3 Solvers

Numerical integration is an old, complex, and deep topic (Press et al., 1992). A complete treatment of this topic is beyond the scope of this text, but it is useful to understand the basic concepts in order to make effective use of the Ptolemy solver functions (which use numerical integration to find solutions to equations). In this section, we will give a brief overview of the solver mechanisms that are implemented in the Ptolemy II Continuous director.

Suppose that w is a continuous-time signal. For the moment, let us ignore the superdense time model used in Ptolemy II, and assume that w is an integrable function of the form $w\colon \mathbb{R} \to \mathbb{R}$. Assume further that for any $t \in \mathbb{R}$, we have a procedure to evaluate $w(t)$. Suppose further that x is a continuous-time signal given by

$$x(t) = x_0 + \int_0^t w(\tau) d\tau,$$

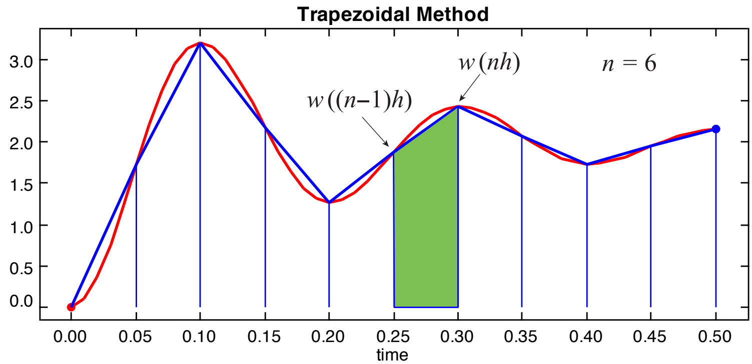


Figure 9.7: Illustration of the trapezoidal method. The area under the curve is approximated by the sum of the areas of the trapezoids. One of the trapezoids is shaded.

where x_0 is a constant. Equivalently, $x(t)$ is the area under the curve formed by $w(\tau)$ from $\tau = 0$ to $\tau = t$, plus an initial value x_0. Note that, consequently, w is the derivative of x, or $w(t) = \dot{x}(t)$. Given w, we can construct x by providing w as the input to an Integrator actor with *initialState* set to x_0; x will then be the output.

Numerical integration is the process of evaluating x at enough points $t \in \mathbb{R}$ to accurately deduce the shape of the function. Of course, the meaning of "accurate" may depend on the application, but one of the key criteria is that the value of x is sufficiently accurate at sufficiently many points that those values of x can be used to calculate values of x at additional points $t \in \mathbb{R}$ in time. A **solver** is a realization of a numerical integration algorithm. The simplest solvers are **fixed step size solvers**. They define a **step size** h, and calculate x at intervals of h, specifically $x(h)$, $x(2h)$, $x(3h)$, etc.

A reasonably accurate fixed step size solver uses the **trapezoidal method**, where it approximates x as follows:

$$x(nh) = \begin{cases} x_0, & \text{if } n = 0 \\ x((n-1)h) + h(w((n-1)h) + w(nh))/2, & \text{if } n \geq 1 \end{cases}$$

This approach is illustrated in Figure 9.7. As defined by the equations above, the area under the curve w from 0 to nh is approximated by the sum of the areas of trapezoids of width h, where the heights of the sides of the trapezoids are given by $w(mh)$ for integers

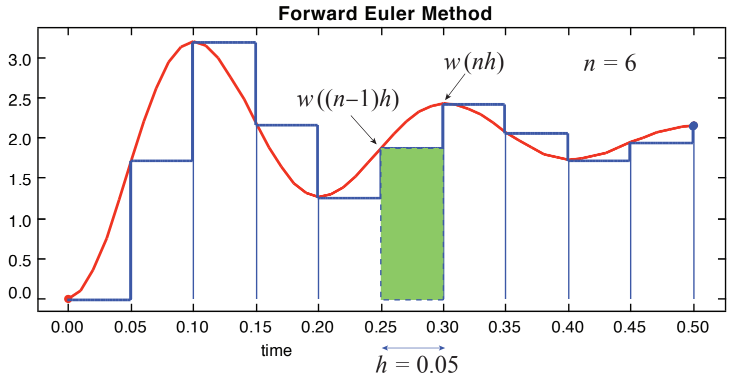


Figure 9.8: Illustration of the forward Euler method. The area under the curve is approximated by the sum of the areas of the rectangles, like the shaded one.

m. The shaded trapezoid in the figure approximates the area under the curve from time $(n-1)h$ to time nh, where $n = 6$ and $h = 0.05$.

A trapezoidal method solver is difficult to use within feedback systems like the one shown in Figure 9.1, however, because the solver needs to compute the outputs of the Integrator actors based on the inputs. To compute the output of an Integrator at time $t_n = nh$, the solver needs to know the value of the input at both $t_{n-1} = (n-1)h$ and $t_n = nh$. But in Figure 9.1, the input to the integrators at any time t_n depends on the output of the same integrators at that same time t_n; there is a circular dependency. Solvers that exhibit a circular dependency are called **implicit method** solvers. One way to use them within feedback systems is to "guess" the feedback value and iteratively refine the guess until some desired accuracy is achieved, but in general there is no assurance that such strategies yield unique answers.

In contrast to implicit method solvers, the **forward Euler method** is an **explicit method** solver. It is similar to the trapezoidal method but is easier to apply to feedback systems. It approximates x by

$$x(nh) = x((n-1)h) + hw((n-1)h).$$

This approach is illustrated in Figure 9.8. The area under each step is approximated as a rectangle rather than as a trapezoid. This method is less accurate, usually, and errors accumulate faster, but it does not require the solver to know the input at time nh.

In general, using a smaller step size h increases the accuracy of the solution, but increases the amount of computation required. The step size required to meet a target level of accuracy depends on how rapidly the signal is varying. Both the trapezoidal method and the forward Euler method can be generalized to become **variable step size solvers** that dynamically adjust their step size based on the variability of the signal. Such solvers evaluate the integral at time instants t_1, t_2, etc., using an algorithm to determine the increment to use between time instants. This algorithm first chooses a step size, then performs the numerical integration, then estimates the error. If the estimate of the error is above some threshold (controlled by the *errorTolerance* parameter of the director), then the director redoes the numerical integration with a smaller step size.

A variable-step-size forward Euler solver will first determine a time increment h_n to define $t_n = t_{n-1} + h_n$ and then calculate

$$x(t_n) = x(t_{n-1}) + h_n w(t_{n-1}).$$

The variable-step-size forward Euler method is a special case of the widely used **Runge-Kutta** (**RK**) methods. The Continuous director offers two variants of RK solvers, `ExplicitRK23Solver` and `ExplicitRK45Solver`, selected using the *ODESolver* parameter of the director. These variants are described in more detail in the sidebars on pages 328 and 329. The plot in Figure 9.5 is generated using the `ExplicitRK23Solver`. A closeup with stems that indicate where the solver chose to calculate signal values is shown in Figure 9.9. This figure shows that the solver uses smaller step sizes in regions where the signal is varying more rapidly.

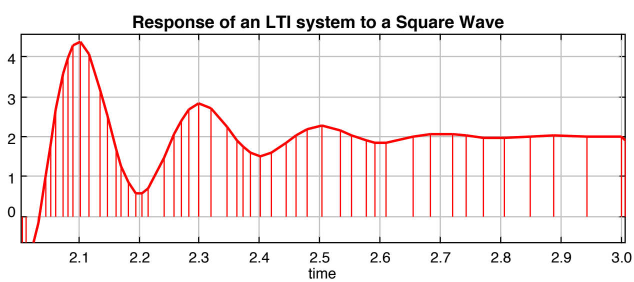
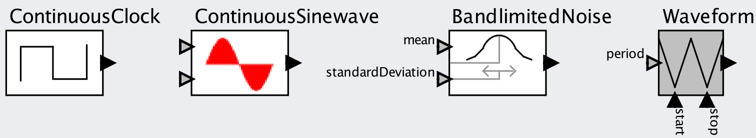


Figure 9.9: A closeup of the plot in Figure 9.5 (with stems showing) reveals that the solver uses smaller step sizes in regions where the signal varies more rapidly.

Sidebar: Continuous-Time Signal Generators

The continuous domain provides several actors that generate continuous-time signals.

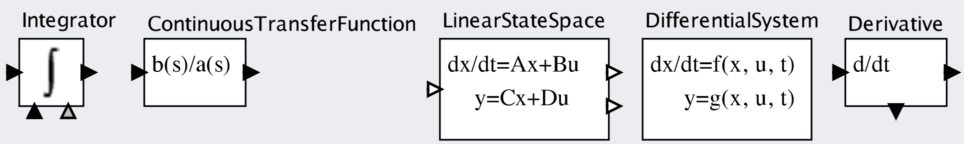


These actors are located in `DomainSpecific→Continuous→SignalGenerators`, except BandlimitedNoise, which is in `DomainSpecific→Continuous→Random`.

- **ContinuousClock** has parameters similar to DiscreteClock, but produces a **piecewise constant** signal. A square wave, like that shown in Figure 9.5, is a simple example of such a signal; the actor is also capable of producing complex repeating or non-repeating patterns.
- **ContinuousSinewave**, as the name implies, produces a sine wave. The frequency, phase, and amplitude of the sine wave are set via parameters. This actor constrains the step size of the solver to ensure that its output is reasonably smooth; the step size is set to be no greater than one tenth of a period of the sine wave.
- **BandlimitedNoise** is the most sophisticated of these signal generators. This actor generates continuous-time noise with a Gaussian distribution and controlled bandwidth. Although a full discussion of the topic is beyond the scope of this text, we note that it is not theoretically possible for a causal system to produce perfectly bandlimited noise (see Lee and Varaiya (2011)). This actor implements a reasonable approximation. Like ContinuousSinewave, this actor affects the step size chosen by the solver; it ensures that the solver samples the signal at least as frequently as twice the specified bandwidth. This is nominally the Nyquist frequency of an ideally bandlimited noise signal.
- **Waveform** produces a periodic waveform from a finite set of samples. It provides two interpolation methods, linear and Hermite, where the latter uses a third-order interpolation technique based on the Hermite curves in Chapter 11 of Foley et al. (1996). Hermite interpolation is useful for generating smooth curves of arbitrary shape. The interpolation assumes that the waveform is periodic. Note that this actor also affects the step sizes taken by the solver. In particular, it ensures that the solver includes the specified samples, though it does not require the solver to include any samples between them.

Sidebar: Actors for Modeling Dynamics

Ptolemy II provides several actors that can be used to model continuous-time systems with complicated dynamics (behavior over time). These actors are shown below, and can be found in `DomainSpecific→Continuous→Dynamics`:



The fundamental actor is the Integrator, described in Sections 9.1.1 and 9.2.4. The others are higher-order actors that construct submodels using instances of Integrator.

- ContinuousTransferFunction, as explained in Section 9.1.2, can realize a continuous-time system based on a transfer function specified as a ratio of two polynomials. The ContinuousTransferFunction actor does not support non-zero initial conditions for the Integrators.
- **LinearStateSpace** specifies the input-output relationship of a system with a set of matrices and vectors that describe a linear constant-coefficient difference equation (**LCCDE**). Unlike ContinuousTransferFunction, this actor supports non-zero initial conditions, but it is similarly constrained to model systems that can be characterized by linear functions.
- **DifferentialSystem** can be used to model complicated nonlinear dynamics. For example, it can be used to specify the Lorenz attractor of Example 9.1, as shown in Exercise 3. See the actor documentation for details.
- **Derivative** provides a crude estimate of the derivative of its input. Use of this actor is discouraged, however, because its output can be very noisy, even if the input is continuous and differentiable. The output is simply the difference between the current input and the previous input divided by the step size. If the input is not differentiable, however, the output is not piecewise continuous, which may force the solver to use the smallest allowed step size. Note that this actor has *two* outputs. The bottom output produces a discrete event when the input has a discontinuity. This event represents a Dirac delta function, explained in Section 9.2.4.

Sidebar: Runge-Kutta Methods

In general, an ODE can be represented by a system of differential equations on a vector-valued state

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

where $x : \mathbb{R} \to \mathbb{R}^n$, $y : \mathbb{R} \to \mathbb{R}^m$, and $u : \mathbb{R} \to \mathbb{R}^l$ are state, output, and input signals. The functions $g : \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R} \to \mathbb{R}^n$ and $f : \mathbb{R}^n \times \mathbb{R}^l \times \mathbb{R} \to \mathbb{R}^m$ are state functions and output functions respectively. The state function g is represented by the Expression actors in the feedback path in Figure 9.1. This function gives the inputs $\dot{x}(t)$ of the Integrator actors as a function of their outputs $x(t)$, external inputs $u(t)$ (of which there are none in Figure 9.1), and the current time t (which is also not used in Figure 9.1).

Given this formulation, an explicit k-stage RK method has the form

$$x(t_n) = x(t_{n-1}) + \sum_{i=0}^{k-1} c_i K_i, \tag{9.5}$$

where

$$\begin{aligned} K_0 &= h_n g(x(t_{n-1}), u(t_{n-1}), t_{n-1}), \\ K_i &= h_n g(x(t_{n-1}) + \sum_{j=0}^{i-1} A_{i,j} K_j, u(t_{n-1} + hb_i), \\ &\qquad t_{n-1} + hb_i), \quad i \in \{1, \cdots, k-1\} \end{aligned}$$

and $A_{i,j}$, b_i and c_i are algorithm parameters calculated by comparing the form of a Taylor series expansion of x with (9.5).

The first-order RK method, also called the forward Euler method, has the (much simpler) form

$$x(t_n) = x(t_{n-1}) + h_n \dot{x}(t_{n-1}).$$

This method is conceptually important but not as accurate as other available methods.

Continued on page 329.

Sidebar: Runge-Kutta Methods - Continued

Continued from page 328.

More accurate Runge-Kutta methods have three or four stages, and also control the step size for each integration step. The `ExplicitRK23Solver` implemented by the Continuous director is a $k = 3$ (three-stage) method and is given by

$$x(t_n) = x(t_{n-1}) + \frac{2}{9}K_0 + \frac{3}{9}K_1 + \frac{4}{9}K_2, \tag{9.6}$$

where

$$K_0 = h_n g(x(t_{n-1}), t_{n-1}), \tag{9.7}$$

$$K_1 = h_n g(x(t_{n-1}) + 0.5K_0, u(t_{n-1} + 0.5h_n), t_{n-1} + 0.5h_n), \tag{9.8}$$

$$K_2 = h_n g(x(t_{n-1}) + 0.75K_1, u(t_{n-1} + 0.75h_n), t_{n-1} + 0.75h_n). \tag{9.9}$$

Notice that in order to complete one integration step, this method requires evaluation of the function g at intermediate times $t_{n-1} + 0.5h_n$ and $t_{n-1} + 0.75h_n$, in addition to the times t_{n-1}, where h_n is the step size. This fact significantly complicates the design of actors, because they have to tolerate multiple evaluations (firings) that are speculative and may have to be redone with a smaller step size if the required accuracy is not achieved. The validity of a step size h_n is not known until the full integration step has been completed. In fact, any method that requires intermediate evaluations of the state function g, such as the classical fourth-order RK method, linear multi-step methods (LMS), and BulirschStoer methods, will encounter the same issue.

In the Continuous domain, the RK solvers speculatively execute the model at intermediate points, invoking the `fire` method of actors but not their `postfire` method. As a consequence, actors used in the Continuous domain must all conform to the strict actor semantics; they must not change state in their `fire` method.

9.2 Mixed Discrete and Continuous Systems

The continuous domain supports mixtures of discrete and continuous behaviors. The simplest such mixtures produce piecewise continuous signals, which vary smoothly over time except at particular points in time, where they vary abruptly. Piecewise continuous signals are explained in Section 9.2.1.

In addition, signals can be genuinely discrete. In particular, as with the DE domain, a signal in the Continuous domain can be absent at a time stamp. A signal that is *never* absent is a true **continuous-time signal**. A signal that is always absent except at a discrete set of time stamps is a **discrete-event signal**, explained in Section 9.2.2. A model that mixes both types of signals is a **mixed signal** model. It is possible to have signals that are continuous-time over a range of time stamps, and discrete-event over another range. These are called **mixed signals**.

9.2.1 Piecewise Continuous Signals

As shown in Figure 9.9, variable step-size solvers produce more samples per unit time when a signal is varying rapidly. These solvers do not, however, directly support discontinuous signals, such as the square wave shown in Figure 9.5. The Continuous director in Ptolemy II augments the standard ODE solvers with techniques that handle such discontinuities, but the signals must be piecewise continuous. Meeting this prerequisite requires some care, as we will discuss later in the chapter.

Recall that Ptolemy II uses a superdense time model. This means that a continuous-time signal is a function of the form

$$x\colon T \times \mathbb{N} \to V, \tag{9.10}$$

where T is the set of model time values (see Section 1.7.3), $\mathbb{N}$ is the non-negative integers representing the microstep, and V is some set of values (the set V is the data type of the signal). This function specifies that, at each model time $t \in T$, the signal x can have several values, and these values occur in a defined order. For the square wave in Figure 9.5, at time $t = 1.0$, for example, the value of the square wave is first $x(t, 0) = 2$ and then $x(t, 1) = -2$.

In order for time to progress past a model time $t \in T$, we need to ensure that every signal in the model has a finite number of values at t. Thus, we require that for all $t \in T$, there exist an $m \in \mathbb{N}$ such that

$$\forall n > m, \quad x(t, n) = x(t, m). \tag{9.11}$$

This constraint prevents chattering Zeno conditions, where a signal takes on infinitely many values at a particular time. Such conditions prevent an execution from progressing beyond that point in model time, assuming the execution is constrained to produce values in chronological order.

Assuming x has no chattering Zeno condition, then there is a least value of m satisfying (9.11). We call this value of m the **final microstep** and $x(t, m)$ the **final value** of x at t. We call $x(t, 0)$ the **initial value** at time t. If $m = 0$, then we say that x has only one value at time t.

Define the **initial value function** $x_i \colon T \to V$ by

$$\forall\, t \in T, \quad x_i(t) = x(t, 0).$$

Define the **final value function** $x_f \colon T \to V$ by

$$\forall\, t \in T, \quad x_f(t) = x(t, m_t),$$

where m_t is the final microstep at time t. Note that x_i and x_f are conventional continuous-time functions if we abstract model time as the real numbers $T = \mathbb{R}$.

A **piecewise continuous** signal is defined to be a function x of the form $x \colon T \times \mathbb{N} \to V$ with no chattering Zeno conditions that satisfies three requirements:

1. the initial value function x_i is continuous on the left;
2. the final value function x_f is continuous on the right; and
3. x has only one value at all $t \in T \backslash D$, where D is a discrete subset of T.

The last requirement is a subtle one that deserves further discussion. First, the notation $T \backslash D$ refers to a set that contains all elements of the set T except those in the set D. D is constrained to be a discrete set, described in the sidebar on page 334. Intuitively, D is a set of time values that can be counted in temporal order. It is easy to see that if $D = \emptyset$ (the empty set), then $x_i = x_f$, and both x_i and x_f are continuous functions. Otherwise each of these functions is piecewise continuous.

A key constraint of the Continuous domain in Ptolemy II is that all signals are piecewise continuous in the above sense. Based on this definition, the square wave shown in Figure 9.5 is piecewise continuous. At each discontinuity, it has two values: an initial value that matches the values before the discontinuity, and a final value that matches the value after the discontinuity. It can be easy to create signals that are not piecewise continuous, however, as illustrated by the following example.

Example 9.3: The model in Figure 9.10 contains an Expression actor whose input is a continuous-time signal. The expression is shown below:

```
(in > 1.0) ? in + 1 : 0
```

If the input is greater than 1.0, then the output will be the input plus one; otherwise the output will be zero. This output signal is not piecewise continuous in the sense described above. Before or at time 1.0, the output value is zero. But at any time after 1.0, the value is not zero. The signal is not continuous from the right.

Figure 9.11 shows the resulting plot, where the output of the Expression actor is labeled "second." The transition from zero to non-zero is not instantaneous, as

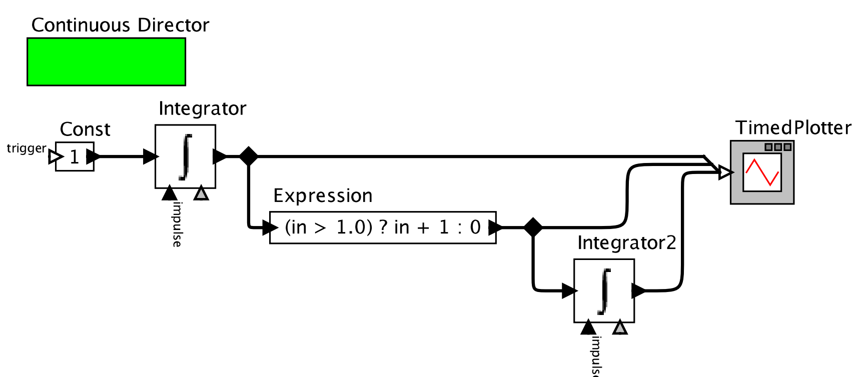


Figure 9.10: A model that produces a signal that is not piecewise continuous, and therefore can exhibit solver-dependent behavior. This problem is eliminated in the model in Figure 9.23. [online]

shown by the slanted dashed line in the middle plot. Worse, the width of the transition depends on seemingly irrelevant details of the model. The model shows the signal connected to a second integrator. If that second integrator is deleted from the

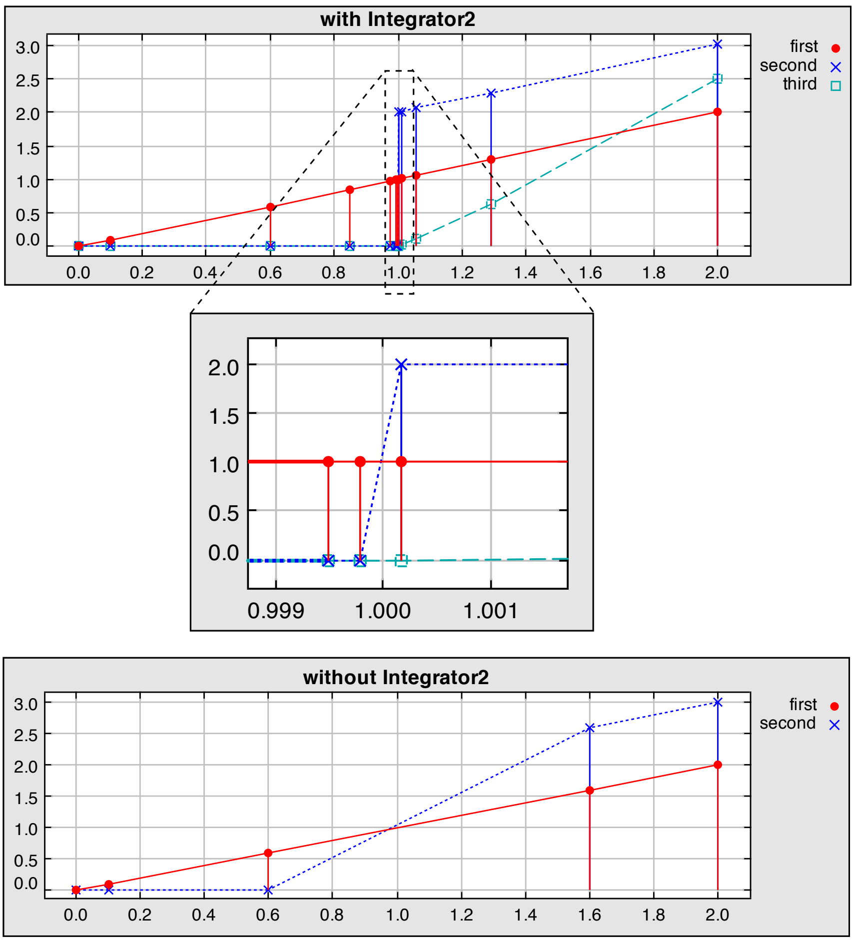


Figure 9.11: A plot of the signal produced by the model in Figure 9.10 with and without the second Integrator. Note that the output of the Expression actor seems to depend on whether the second Integrator is present. The signals labeled “first,” “second,” and “third” are the top-to-bottom inputs of the plotter, respectively.

model, then the width of the transition changes, as shown in the lower plot. This is because the second Integrator affects the step size taken by the solver. In order to achieve adequate integration accuracy, the second Integrator forces a smaller step size in the vicinity of time 1.0.

The problem is not solved by changing the expression to

```
(in >= 1.0) ? in + 1 : 0
```

Here, the transition from zero to non-zero occurs when the input is greater than *or equal to* 1.0. In this case, the resulting signal is not continuous from the left. The resulting plots are identical. The Expression actor simply calculates a specified function of its inputs when it fires. It has no mechanism for generating distinct values at distinct microsteps unless its input already has distinct values at distinct microsteps.

The previous example shows that using an actor whose output is a discontinuous function of the input can create problems if the input is a continuous-time signal. The next few sections describe various mechanisms for properly constructing discontinuous signals. The particular problem with Figure 9.10 is solved using modal models in Section 9.3.1.

Sidebar: Probing Further: Discrete Sets

A set D is a **discrete set** if it is a totally ordered set (for any two elements d_1 and d_2, either $d_1 \leq d_2$ or $d_1 > d_2$) where there exists a one-to-one function $f\colon D \to \mathbb{N}$ that is **order preserving**. Order preserving simply means that for all $d_1, d_2 \in D$ where $d_1 \leq d_2$, we have that $f(d_1) \leq f(d_2)$. The existence of such a one-to-one function ensures that we can arrange the elements of D *in temporal order*. Notice that D is a countable set, but not all countable sets are discrete. For example, the set $\mathbb{Q}$ of rational numbers is countable but not discrete. There is no such one-to-one function.

9.2.2 Discrete-Event Signals in the Continuous Domain

As described earlier, the Continuous domain supports genuinely discrete signals, which are signals that are present only at particular instants. As a consequence, the clock actors that are used in the DE domain (see sidebar on page 241) can be used in the continuous domain.

Example 9.4: The ContinuousClock actor in Figure 9.4 is a composite actor that uses a DiscreteClock and a ZeroOrderHold (see box on page 338), as shown in Figure 9.12. The DiscreteClock produces a discrete-event signal and the ZeroOrderHold converts that signal to a continuous-time signal (see sidebar on page 338).

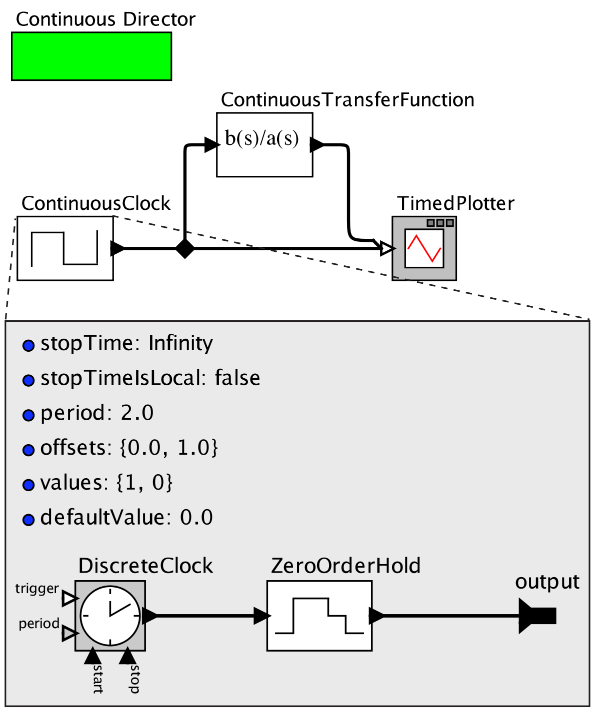


Figure 9.12: The ContinuousClock actor in this model is a composite actor that uses a DiscreteClock and a ZeroOrderHold.

Both signals are piecewise continuous. The output signal from DiscreteClock at the model time of each event is characterized as follows: it is absent at microstep zero (which matches its value at times just before the event); present at microstep one (which is the discrete event); and absent again at microsteps two and higher (which matches its value at larger times until the next discrete event). Therefore, the output of DiscreteClock is piecewise continuous, as required by the solver.

The clock actors described in the sidebar on page 241 all behave in a manner that is similar to DiscreteClock and hence they can all be used in the continuous domain.

Note that although many actors that operate on or produce discrete events have a *trigger* input port, there is rarely a reason to connect that port in the Continuous domain. In the DE domain, a *trigger* port is used to trigger execution of an actor at the time of the input event. But in the Continuous domain, every actor executes every time that there is an execution. Nevertheless, it is sometimes useful to use the *trigger* port, as illustrated by the example in Figure 9.13.

9.2.3 Resetting Integrators at Discrete Times

In addition to its signal input and output, the Integrator actor has two extra ports at the bottom of the icon. The one at the lower right is a PortParameter called *initialState*. When an input token is provided on that port, the state of the Integrator will be reset to the value of the token. The output of the Integrator will change instantaneously to the specified value.

Example 9.5: The model in Figure 9.14 uses a DiscreteClock actor to periodically reset an Integrator.

The input events on the *initialState* port are required to be purely discrete. This means that at all model times, the input signal must be absent at microstep 0. Any attempt to feed a continuous signal into this port results in an exception similar to the one below:

```
IllegalActionException: Signal at the initialState port is not purely
discrete.
```

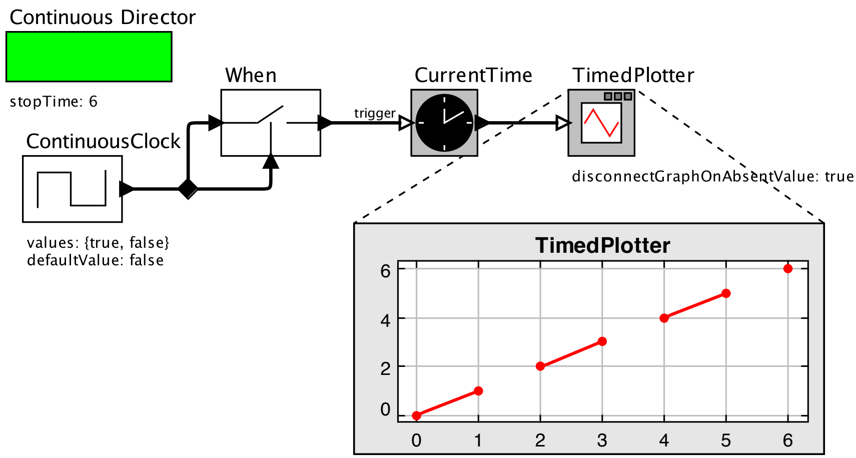


Figure 9.13: The *trigger* port of the CurrentTime actor in this model is used to turn on and off its output. During the time intervals where the output of the DiscreteClock actor is `false`, the CurrentTime actor is disabled, and hence its output will be absent. [online]

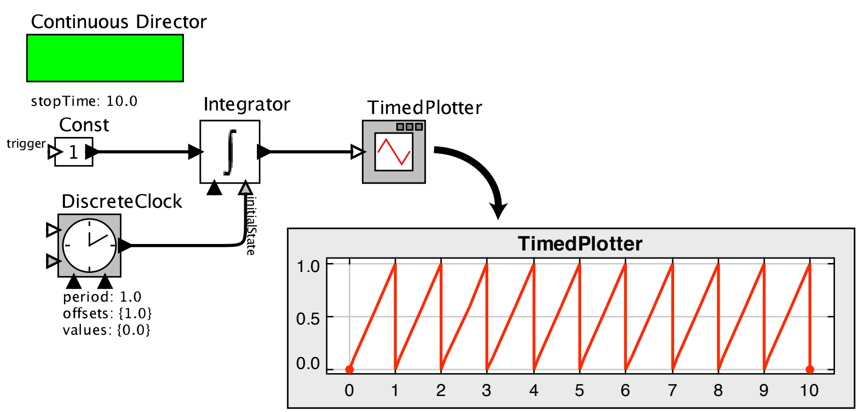


Figure 9.14: Illustration of the use of the *initialState* port of the Integrator actor. [online]

```
    in Integrator
```

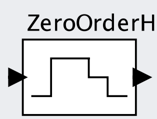
This check ensures that the output of the Integrator is piecewise continuous. In Figure 9.14, for example, at the points of discontinuity the signal first takes the value of the Integrator state prior to applying the reset. In the subsequent microstep, it takes the value after the reset. In contrast, if the output of the Integrator had changed abruptly at microstep zero, then the output could have been a different value at a time infinitesimally earlier — thus violating the requirement for piecewise continuity.

9.2.4 Dirac Delta Functions

The other Integrator input port is called *impulse*. Like the *initialState* port discussed in the previous section, the signal at this port is required to be purely discrete. When an impulse event arrives, it causes an instantaneous increment or decrement of the state (and output)

Sidebar: Continuous Signals from Discrete Events

The **ZeroOrderHold** actor, shown below, takes a discrete-event signal in and produces a continuous-time signal on its output:



This actor is in `DomainSpecific→Continuous→Discrete to Continuous`.

At times between input events, the value of the output is the value of the most recent event, so the output is piecewise constant. At the time of each input event, the output at microstep zero is the value of the previous event, and at microstep one, it takes the value of the current event. Hence, the output signal is piecewise continuous.

It may seem desirable to define an actor that interpolates between the values of the input events, as is done by the Waveform actor (see sidebar on page 326). However, in order to interpolate, the actor would have to know the value of a *future* event. Actors in the continuous domain are required to be **causal**, meaning that their outputs depend only on current and past inputs. The outputs cannot depend on future inputs. Hence, no such interpolation is possible. The Waveform actor is able to perform interpolation because the values that it is interpolating are specified as parameters, not as input events.

of the Integrator. That is, rather than resetting the state to a specified value, it adds to (or subtracts from) the current state.

Mathematically, such functionality is often represented as a **Dirac delta function** in signals and systems. A Dirac delta function is a function $\delta\colon \mathbb{R} \to \mathbb{R}^+$ given by

$$\forall\, t \in \mathbb{R},\ t \neq 0, \quad \delta(t) = 0$$

and

$$\int_{-\infty}^{\infty} \delta(\tau) d\tau = 1.$$

That is, the signal value is zero everywhere except at $t = 0$, but its integral is unity. At $t = 0$, therefore, its value cannot be finite. Any finite value would yield an integral of zero. This is indicated by $\mathbb{R}^+$ in the form of the function, $\delta\colon \mathbb{R} \to \mathbb{R}^+$, where $\mathbb{R}^+$ represents the **extended reals**, which includes infinity. Dirac delta functions are widely used in modeling continuous-time systems (see Lee and Varaiya (2011), for example), so it is important to be able to include them in simulations.

Suppose that a signal y has a Dirac delta function occurring at time t_1 as follows,

$$y(t) = y_1(t) + K\delta(t - t_1),$$

where y_1 is an ordinary continuous-time signal, and K is a scaling constant. Then

$$\int_{-\infty}^{t} y(\tau)d\tau = \begin{cases} \int_{-\infty}^{t} y_1(\tau)d\tau & t < t_1 \\ K + \int_{-\infty}^{t} y_1(\tau)d\tau & t \geq t_1 \end{cases}$$

The component $K\delta(t - t_1)$ is a Dirac delta function at time t_1 with weight K, and it causes an instantaneous increment in the integral by K at time $t = t_1$.

Example 9.6: LTI systems can be characterized by their impulse response, which is their response to a Dirac delta function. The model in Figure 9.15 is an LTI system with transfer function

$$H(s) = \frac{1}{1 + as^{-1} + bs^{-2}}.$$

The model provides a Dirac delta function at time 0.2, producing the impulse response shown in the plot.

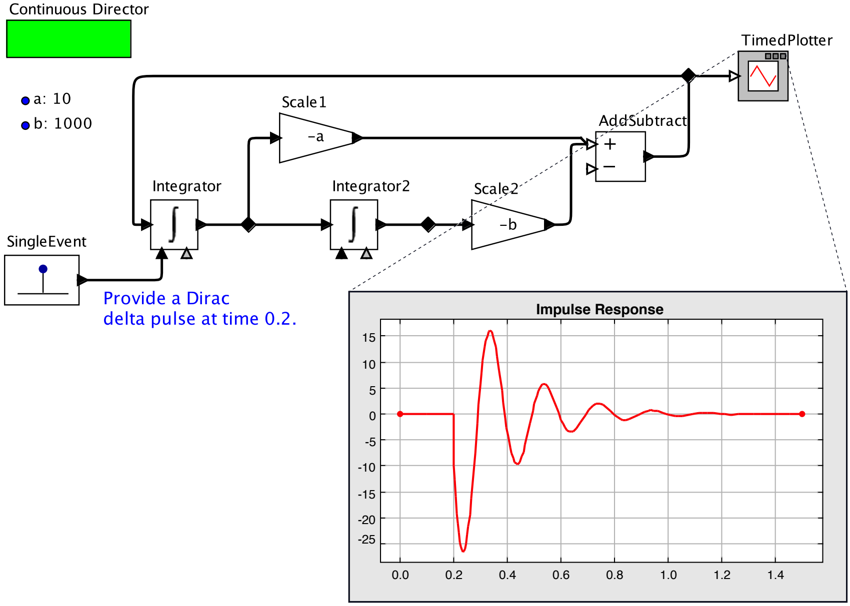
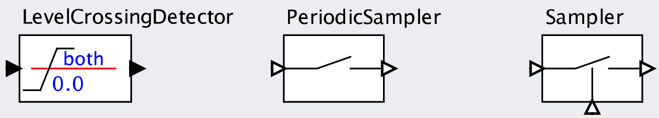


Figure 9.15: Response of an LTI system to a Dirac delta function. [online]

It is difficult to model Dirac delta functions in computing systems because of their instantaneous and infinite nature. The superdense time model of Ptolemy II coupled with the semantics of the Continuous domain provide a rigorous, unambiguous model that can support Dirac delta functions.

Sidebar: Generating Discrete Events

Several actors convert continuous-time signals to discrete-event signals (these actors are located in `DomainSpecific→Continuous→Continuous to Discrete`):



- **LevelCrossingDetector** converts continuous signals to discrete events when the input signal crosses a threshold specified by the *level* parameter. A *direction* parameter constrains the actor to detect only rising or falling transitions. This actor introduces a one-microstep delay before it produces an output. That is, when a level crossing is detected, the actor requests a refiring in the next microstep at the current time, and produces the output during that refiring. This ensures that the output satisfies the piecewise continuity constraint; it is always absent at microstep 0. The one-microstep delay enables the actor to be used in a feedback loop. An example is shown in Figure 9.16, where the feedback loop resets the Integrator each time it reaches a threshold (1.0 in the example).
- **PeriodicSampler** generates discrete events by periodically sampling an input signal. The sampling rate is given by a parameter. By default, the actor reads the initial value of the input signal (the input value at microstep 0), but sends it to the output port one microstep later (at microstep 1). This ensures that the output at microstep 0 is always absent, thus ensuring that the output signal is piecewise continuous. (The input is absent prior to the sample time, so piecewise continuity requires that it be absent at microstep 0 at the sample time.) Because of the one-step delay, the PeriodicSampler can also be used in a feedback loop. For example, it can be used to periodically reset an Integrator, as shown in the example in Figure 9.17.
- **Sampler** is a simple actor. Whenever the *trigger* signal (at the bottom port on the icon) is present, it copies the input from the left port to the output. There is no microstep delay. If the signal at the *trigger* port is a piecewise continuous discrete-event signal, then the output will also be a piecewise continuous discrete-event signal. Sampler will normally read its inputs at microstep 1 because the *trigger* input is discrete. (PeriodicSampler will behave in the same way if its *microstep* parameter is set to 1.)

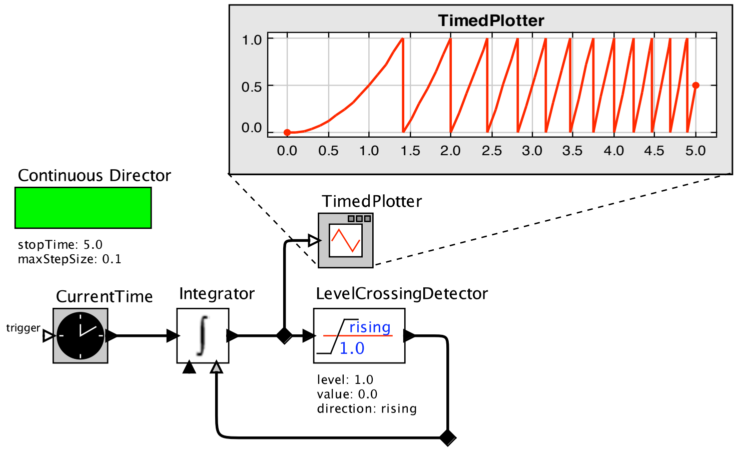


Figure 9.16: Illustration of the LevelCrossingDetector, which can be put in a feedback loop. In this case, whenever the output of the Integrator reaches 1.0, it is reset to zero. [online]

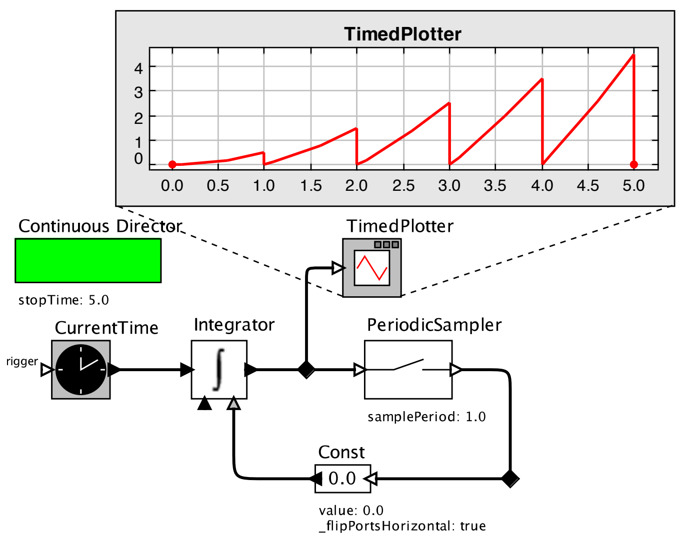


Figure 9.17: The PeriodicSampler actor, placed in a feedback loop. In this case, the Integrator will be reset to zero at intervals of one time unit regardless of its state. [online]

9.2.5 Interoperating with DE

The Continuous and DE domains both support discrete-event signals. There is a subtle but important difference between these domains, however. In the Continuous domain, at a time stamp selected by the solver, *all* actors are fired. In the DE domain, an actor is only fired if either it has an event at an input port or it has previously asked to be fired. As a consequence, DE models can be much more efficient, particularly when events are sparse.

It can be useful to build models that combine the two domains. Such combinations are suitable for many cyber-physical systems, for example, which combine continuous dynamics with software-based controllers. Constructing models with a mixture of Continuous and DE domains is easy, as illustrated by the following example.

Example 9.7: Consider the model in Figure 9.18. The top level of the model is implemented in the DE domain, and includes an opaque composite actor that is a Continuous model. This example models a "job shop," where job arrivals are discrete events, the processing rate is given by an exponential random variable, and the job processing is modeled in continuous time.

The model assigns an integer number to each job. It then approaches that number with a slope given by the (random) rate. The higher the rate, the faster the job is completed. The job is complete when the blue (dashed) line in the plot reaches the red (solid) line in the lower plot. The upper plot shows the times at which each job is generated and completed. Note that this model has a feedback loop such that each time a job is finished, a new one is started with a new service time.

This example is somewhat contrived, however, in the sense that it does not actually require the use of the Continuous domain (see Exercise 4). In fact, models where continuous-time signals linearly increase or decrease can usually be realized within the DE domain alone, without the need for a solver. That said, there is still value in constructing the mixed-domain model because it can easily evolve to support more complex dynamics in the Continuous portion.

Continuous models can be placed within DE models, as shown in the previous example. Conversely, DE models can be placed within Continuous models. The choice of top level domain is often determined by emphasis. If the emphasis is on a discrete controller, then

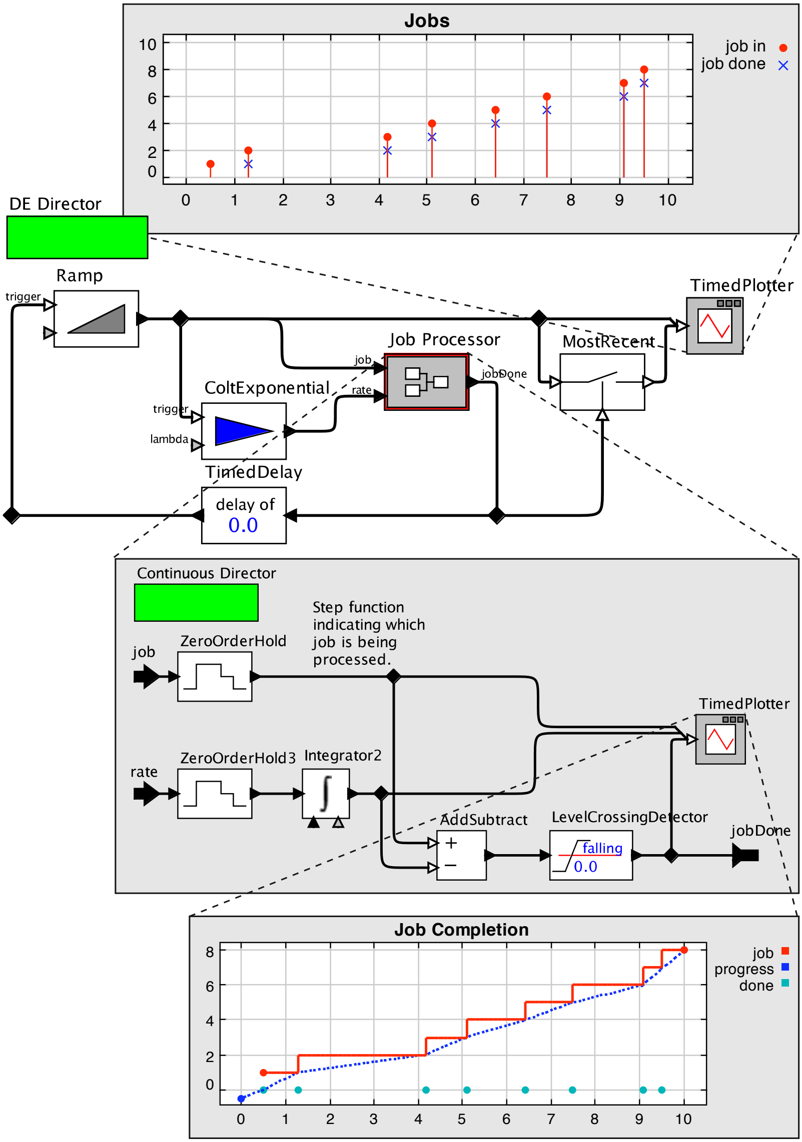


Figure 9.18: Illustration of a hierarchical combination of DE and Continuous models, as considered in Example 9.7. [online]

using DE at the top level often makes sense. If the emphasis is on the physical plant, then using Continuous at the top level may be better.

If multiple Continuous submodels are placed within a DE model, then the solvers in the submodels are decoupled. This can be useful for modeling systems with widely disparate time scales; one solver can use small step sizes without forcing the other submodel to use small step sizes.

The ability to combine DE and Continuous domains relies on a key property of DE, which is that events in DE normally occur at microstep one, not at microstep zero. When these events cross the boundary into a Continuous model, they preserve this microstep. Hence, a signal that passes from the DE domain to the Continuous domain will normally be absent at microstep zero, thus ensuring piecewise continuity. When a signal goes from the Continuous domain to the DE domain, however, it is important that the signal be discrete, as would be produced by a Sampler or LevelCrossingDetector (see page 341).

9.2.6 Fixed-Point Semantics

Recall from Section 7.3.4 that, as of this writing, the DE director in Ptolemy II implements an approximation of the fixed-point semantics described by Lee and Zheng (2007). In contrast, the Continuous director implements an exact fixed-point semantics, and can therefore execute some models that DE cannot.

Example 9.8: Consider the model shown in Figure 9.19. This model is identical to the model considered in Example 7.14, except that the Continuous director is used instead of the DE director. The Continuous director, unlike the DE director, is able to fire actors multiple times at a given time stamp. As a consequence, it does not need to know whether an event is present or absent at the input of the composite actor before it is fired. The director can fire the composite actor, obtain an event from the DiscreteClock, and then later fire the composite actor again once that event has been fed back.

9.3 Hybrid Systems and Modal Models

A **hybrid system** is a model that combines continuous dynamics with discrete mode changes. Such models are created in Ptolemy II using ModalModel actors, found in the `Utilities` library and explained in Chapter 8. This section starts by examining a prebuilt hybrid system, and concludes by explaining the principles that make hybrid models work. Chapter 8 explains how to construct such models, and explains how time is handled in mode refinements.

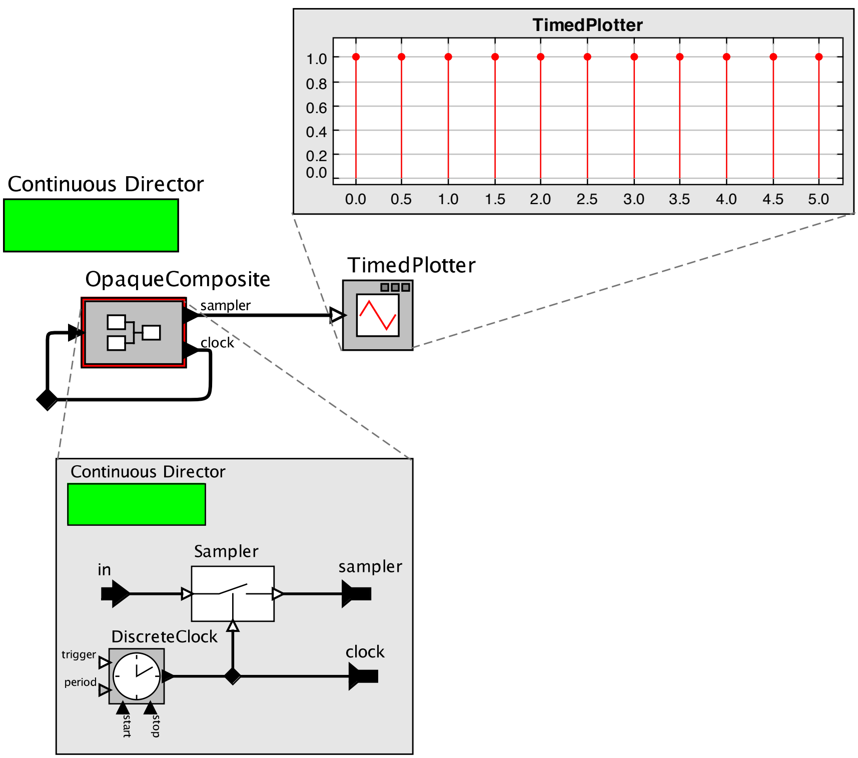


Figure 9.19: A discrete-event model that is executable using the Continuous director, but not using the DE director, as shown in Example 7.14. [online]

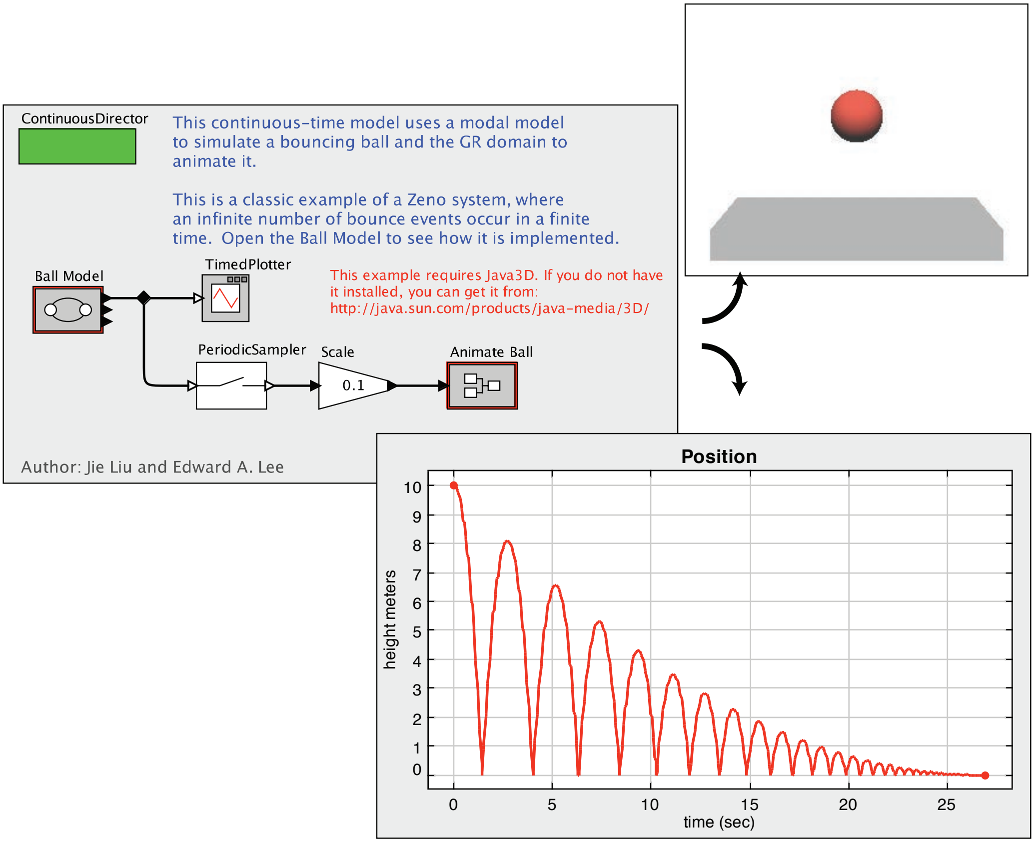


Figure 9.20: Top level of the bouncing ball hybrid system example. [online]

Example 9.9: A bouncing ball model is shown in Figure 9.20. It can be found under "Bouncing Ball" in the Tour of Ptolemy II (Figure 2.3, in the "Hybrid Systems" entry). The bouncing ball model uses a ModalModel component named Ball Model. Executing the model yields a plot like that in the figure (along with 3-D animation that is constructed using the GR (graphics) domain, which is not covered here). This model has continuous dynamics during times when the ball is in the air, and discrete events when the ball hits the surface and bounces.

Figure 9.21 shows the contents of Ball Model, which is a modal model with three states: init, free, and stop. During the time a modal model is in a state, its behavior is specified by the mode refinement. In this case, only the free state has a refinement,

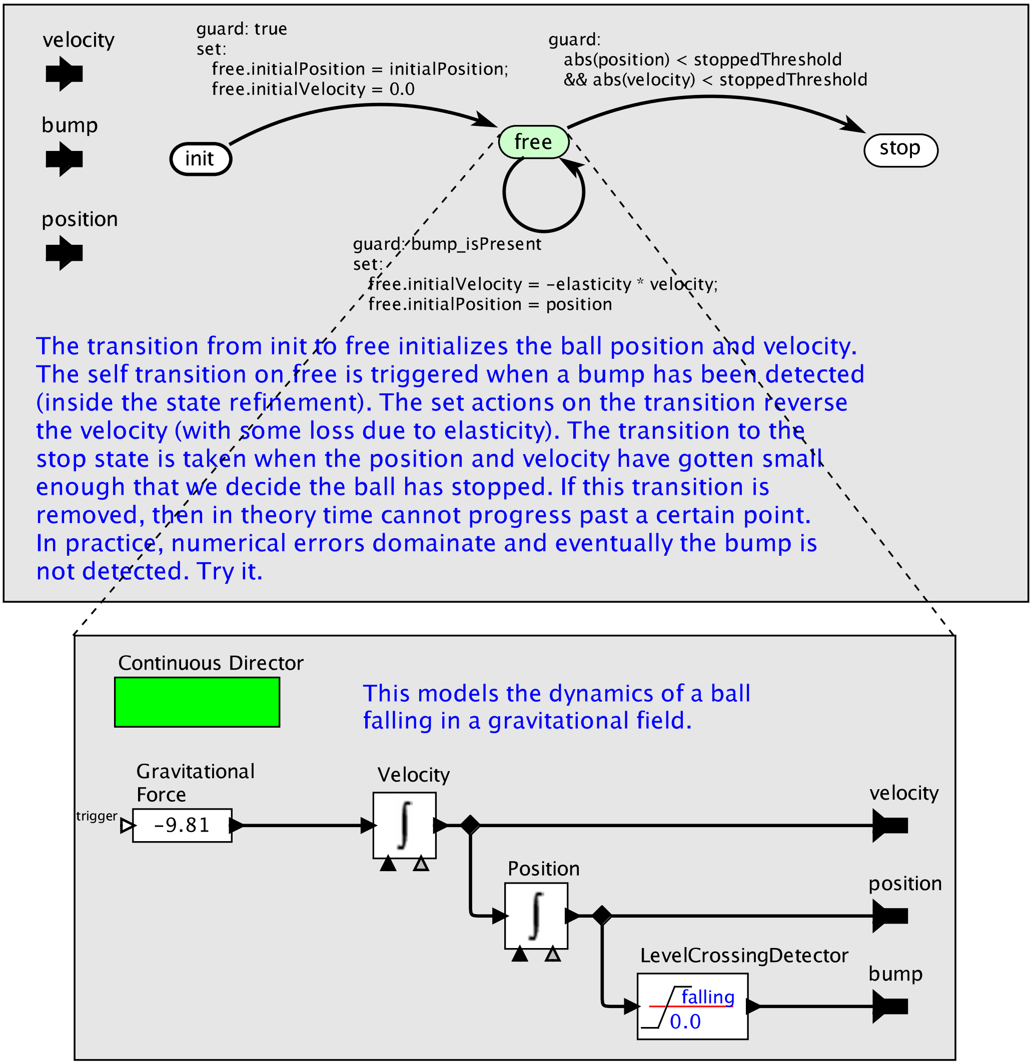


Figure 9.21: Inside the Ball Model of Figure 9.20.

shown at the bottom of Figure 9.21. The *init* state is the initial state, which is used only for its outgoing transition, and has set actions to initialize the ball model. Specifically, the transition is labeled as follows:

```
guard: true
set:
 free.initialPosition = initialPosition;
 free.initialVelocity = 0.0
```

The first line is a guard, which is a predicate that determines when the transition is enabled. In this case, the transition is always enabled, since the predicate has value `true`. Thus, the model immediately transitions to mode *free*. This transition occurs in microstep zero at the start of the execution. The "set:" line indicates that the successive lines define set actions (see Section 6.2). The third and fourth lines set the parameters of the destination mode *free*. The *free* state represents the mode of operation when the ball is in free fall, and the *stop* state represents the mode where the ball has stopped bouncing.

When the model begins executing, it is in the *init* state. Since the *init* state has no refinement, the outputs of the Ball Model will be absent while the modal model is in that state. The outgoing transition has a guard that is always enabled, so the Ball Model will be in that state for only one microstep.

Inside the *free* state, the refinement represents the law of gravity, which states that an object of any mass will have an acceleration of about $9.81 \text{meters}/\text{second}^2$. The acceleration is integrated to find the velocity, which is, in turn, integrated to find the vertical position. In the refinement, a LevelCrossingDetector actor is used to detect when the vertical position of the ball is zero. Its output produces events on the (discrete) output port *bump*. Figure 9.21 shows that this event triggers a state transition back to the same *free* state, but now the *initialVelocity* parameter is changed to reverse the sign and attenuate its value by the *elasticity*. The ball loses energy when it bounces, as shown by the plot in Figure 9.20.

Figure 9.21 shows that when the position and velocity of the ball drop below a specified threshold, the state machine transitions to the state *stop*, which has no refinement. At this point, the model produces no further outputs.

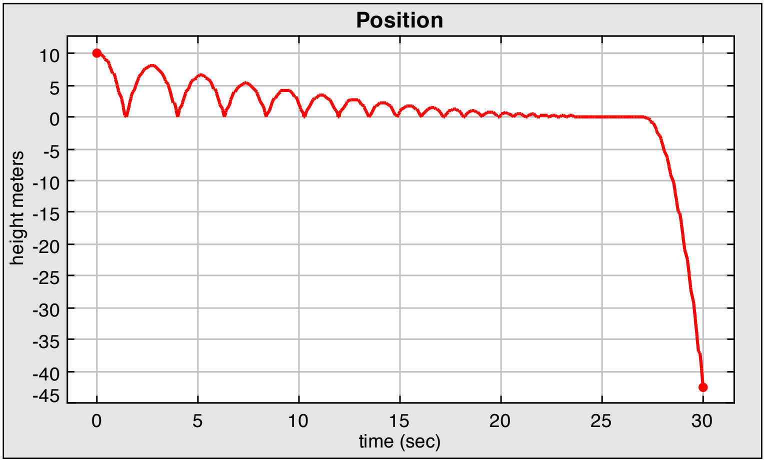


Figure 9.22: Result of running the bouncing ball model without the stop state.

The bouncing ball model illustrates an interesting property of hybrid system modeling. The *stop* state, it turns out, is essential. Without it, the time between bounces keeps decreasing, as does the magnitude of each bounce. At some point, these numbers get smaller than the representable precision, and large errors start to occur. Removing the *stop* state from the FSM and re-running the model yields the result shown in Figure 9.22. In effect, the ball falls through the surface on which it is bouncing and then goes into a free-fall in the space below.

The error that occurs here illustrates a fundamental pitfall that can occur with hybrid system modeling. In this case, the event detected by the LevelCrossingDetector actor can be missed by the simulator. This actor works with the solver to attempt to identify the precise point in time when the event occurs. It ensures that the simulation includes a sample at that time. However, when the numbers become sufficiently small they are dominated by numerical errors, and the event is missed.

The bouncing ball is an example of a Zeno model (see Section 7.4). The time between bounces gets smaller as the simulation progresses, and it gets smaller fast enough that, with infinite precision, an infinite number of bounce events would occur in a finite amount of time.

9.3.1 Hybrid Systems and Discontinuous Signals

Recall from Example 9.3 that actors whose outputs are a discontinuous function of the input can create signals that are not piecewise continuous. This can result in solver-dependent behavior, in which arbitrary step-size decisions made by the solver strongly affect the execution of the model. These problems can be solved using modal models, as illustrated in the following example.

Example 9.10: Figure 9.23 shows a variant of the model in Figure 9.10 that correctly produces a piecewise continuous signal. This variant uses a ModalModel, which specifies a transition at the discontinuity of the signal. The transitions of a modal model are instantaneous, in that model time does not advance. The microstep, however, does advance. In this model, the transition occurs within the *errorTolerance* (a director parameter) after time 1.0. At the time of the transition, the refinement of the *zero* state fires first, producing output 0 at microstep 0, and then the refinement of the *increment* state fires at microstep 1, producing output 2.0 (or within the *errorTolerance* of 2.0). Hence, the output signal is piecewise continuous.

The operation of ModalModel actors is explained in Chapter 8. When combined with the Continuous director, such operation translates naturally into an effective and useful semantics for hybrid systems. To fully understand the interoperation of modal models and the Continuous domain, it is useful to review the execution semantics of modal models, described in Section 6.2. Specifically, a firing of the modal model consists of firing of the refinement of the current state (if there is one), evaluating the guards, and taking a transition if a guard is true. It is also important to understand that while a mode is inactive, time does not advance in the refinement. Thus, the local notion of time within a refinement lags the global notion of time in its environment.

In modal models, transitions are allowed to have output actions (see Section 6.2). Such actions should be used with care because the transition may be taken in microstep 0, and the resulting output will not be piecewise continuous. If output actions are used to produce discrete events, the transition must be triggered by a discrete event from a piecewise continuous signal.

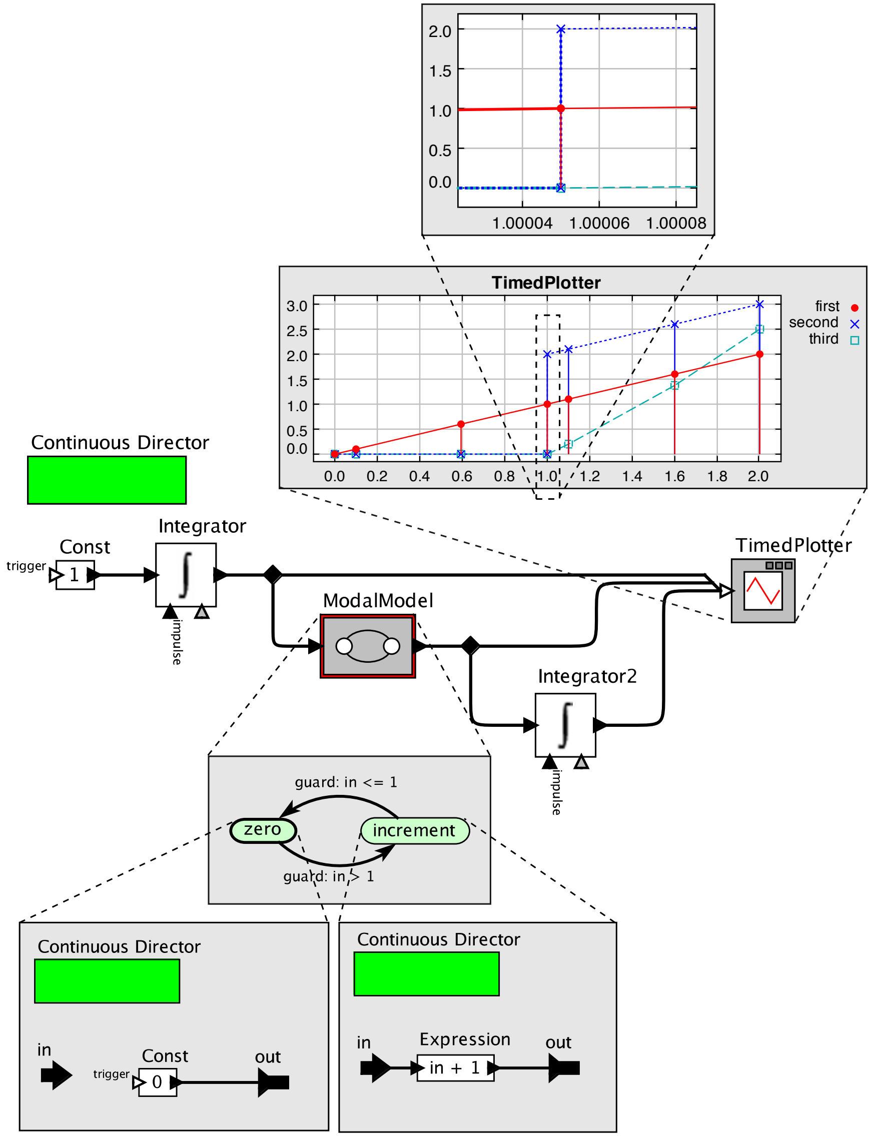


Figure 9.23: A variant of the model in Figure 9.10 that correctly produces a piece-wise continuous signal. [online]

9.4 Summary

Modeling continuous-time systems and approximating their behavior on digital computers can be tricky. The superdense time model of Ptolemy II makes it easier to accurately model a large class of systems, and is particularly useful for systems that mix continuous and discrete behaviors. The Continuous domain, described in this chapter, exploits this model of time to deliver sophisticated modeling and simulation capabilities.

Exercises

1. Let x be a continuous-time signal where $x(0) = 1$ and $\ddot{x}(t) = -x(t)$, where $\ddot{x}$ is the second derivative of x with respect to time t. It is easy to verify that a solution to this equation is $x(t) = \cos(t)$.

   (a) Use Integrator actors to construct this signal x without using any actors or expressions involving trigonometric functions. Plot the execution over some reasonable time to verify that the solution matches what theory predicts.

   (b) Change the solver that the director uses from ExplicitRK23Solver to ExplicitRK45Solver. Describe qualitatively the difference in the results. Which solver gives a better solution? What criteria are you using for "better"? Give an explanation for the differences.

   (c) All numerical ODE solvers introduce errors. Although the theory predicts that the amplitude of the solution $x(t) = \cos(t)$ remains constant for all time, a numerical solver will be unable to sustain this. Describe qualitatively how the ExplicitRK23Solver and ExplicitRK45Solver perform over the long run, leaving other parameters of the director at their default values. Which solver is better? By what criteria?

   (d) Experiment with some of the other director parameters. How does the *errorTolerance* parameter affect the solution? How about *maxStepSize*?

2. Example 9.2 shows the use of ContinuousTransferFunction to specify a transfer function for a continuous-time system. Show that with the parameters given in the example, that the models in Figures 9.4 and 9.6 are equivalent. **Hint:** This problem is easy if you have taken a typical electrical engineering signals and systems class, but it is doable without that if you recognize the following fact: If a signal w has Laplace transform W, then the integral of that signal has Laplace transform W' where for all complex numbers s, $W'(s) = W(s)/s$. That is, dividing by s in the Laplace domain is equivalent to integrating in the time domain.

3. Consider the Lorenz attractor in Example 9.1. Implement the same system using the DifferentialSystem higher-order actor. Give the parameter names and values for your DifferentialSystem.

4. The model in Example 9.7 does not actually require the Continuous domain to achieve the same functionality. Construct an equivalent model that is purely a DE model.

10 Modeling Timed Systems

Janette Cardoso, Patricia Derler, John C. Eidson, Edward A. Lee, Slobodan Matic, Yang Zhao, Jia Zou

Contents

This chapter is devoted to modeling timing in complex systems. We begin with a discussion of clocks, with particular emphasis on multiform time. We then illustrate how to use multiform time in three particular modeling problems. First, we consider clock synchronization, where network protocols are used to correct clocks in distributed systems to ensure that the clocks progress at approximately the same rates. Second, we consider the problem of assessing the effect of communication delays on the behavior of systems. And third, we consider the problem of assessing the effect of execution time on the behavior of systems. We then conclude the chapter with an introduction to a programming model called Ptides that makes possible systems whose behavior is unaffected by variations in the timing of computation and networking, up to a point of failure. The Ptides model of computation enables much more deterministic cyber-physical systems.

As a preface to this chapter, we issue a warning to the reader. Discussing the modeling of timing in cyber-physical systems can be very confusing, because in such models, time is intrinsically multiform. Several distinct views and measurements of time may simultaneously coexist, making the use of words like "when" and phrases such as "at the same time" treacherous.

The most obvious source of temporal diversity is in the distinction between real time and model time. By "real time" we mean here the time that elapses while a model executes, or while the system that the model is supposed to model executes. If the execution of the model is a simulation of some physical system, then "real time" may refer to the time elapsing in the world where the simulation is executing (e.g. the time that your wristwatch measures while you watch a simulation run on your laptop). Model time, by contrast, exists within the simulation and advances at a rate that bears little relationship with real time.

But even this can be confusing, because the physical system being simulated may be a real-time system, in which case, model time is a simulation of real time. But not the same real time that your wristwatch is measuring. Worse, within a simulation of a cyber-physical system, there may be a multiplicity of time measuring devices. There is no single wristwatch. Instead, there are clocks on microcontroller boards and in networking infrastructure. These may or may not be synchronized, but even if they are synchronized, the synchronization is inevitably imperfect, and modeling the imperfections may be an important part of the model. As a consequence, a single model may have several distinct timelines against which the components of the system are making progress. Moreover, as discussed in Chapter 8, modal models lead to some timelines becoming frozen, while others progress. Keeping these multiple timelines straight can be a challenge. This is the primary topic of this chapter.

10.1 Clocks

As explained in Section 1.7, Ptolemy II provides a coherent notion of time across domains. Ptolemy II supports multiform time. Every director contains a **local clock** that keeps track of the local time. The local time is initialized with the *startTime* of the director and evolves at a given *clockRate*. The *clockRate* can change.

The parameter dialogue of a simple director is shown in figure 10.1 (most directors have more parameters than these, but every director has at least these). The *startTime* parameter, if given, specifies the time of the local clock when the model is initialized. If it is not given, then the time at initialization will be set to the time of the environment (the **enclosing director**, or the next director above in the hierarchy), or will be set to zero if the director is at the top level of the model. When the local time of the director reaches the value described by *stopTime*, the director will request to not be fired anymore (by returning *false* from its postfire method).

The parameter dialog of a director also contains a Configure button for configuring the local clock, as shown in Figure 10.1. This can be used to set the time resolution, which is explained in Section 1.7.3, and the **clock rate**. The *clockRate* parameter specifies how rapidly the local clock progresses relative to the clock of the enclosing director. A director refers to the time of the enclosing director as the **environment time**. If there is no enclosing director, then advancement of the clock is entirely controlled by the director.

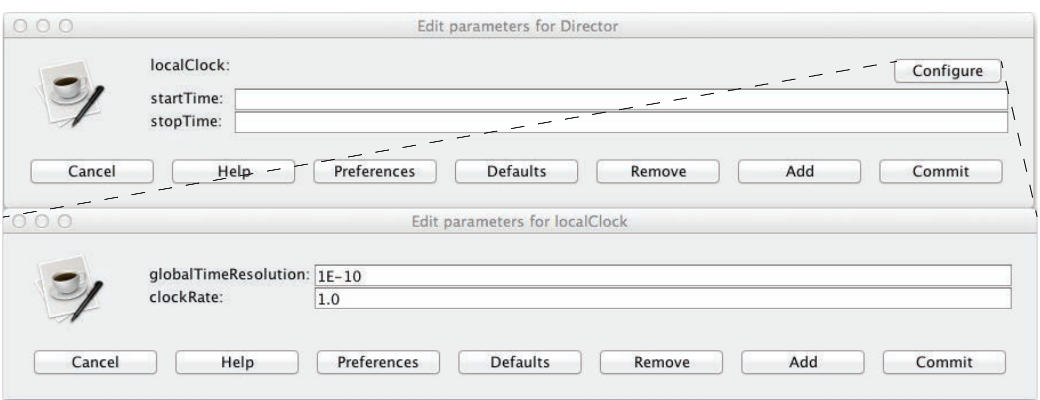


Figure 10.1: The director parameters for the local clock.

For example, if a DE director has an enclosing director, then the *clockRate* is used to translate the time stamps of input events from the environment into local time stamps. If it has no enclosing director, then all events are generated locally, and the director will always advance time to the least time stamp of unprocessed events.

Every director in Ptolemy has a local clock. If an untimed director such as SDF or SR has no enclosing director, then the clock value never changes (unless its *period* parameter is set to a non-zero value).

Example 10.1: Figure 10.2 shows four different time lines of clocks $c1$ through $c4$. The clock $c1$ (solid red line) represents a clock that evolves uniformly with the environment time. Clocks $c2$ through $c4$ have varying clock rates, clock values and offsets. During the first 5 time units, all clocks evolve with the same rate as the environment time. Clock $c3$ starts with an offset of -5.0, i.e. it is 5 time units behind environment time. At environment time 5, the clock rates of $c2$ and $c3$ are modified; the clock rate of $c2$ is increased and the clock rate of $c3$ is decreased. Clock $c4$ is suspended, so that its value does not change during the next 3 time units. At time 8, $c4$ is resumed. At time 10, the value of $c3$ is set to 10 to match the environment time. Because the clock rate of $c3$ is still less than 1.0, the clock immediately starts lagging.

These different clock behaviors can be modeled in Ptolemy. We can perform the following actions on clocks: define an **offset**, change the **clock value**, **suspend** and **resume** the clock, and change the **clock rate**.

Example 10.2: The model that generates the plot shown in Figure 10.2 is presented in Figure 10.3. The clock rate is modified by changing the parameter *clockRate* of the parameter *localClock* in a director. In the Fast composite actor at the upper right, that parameter is set equal to the port parameter *rate*, so that each time a new rate is provided on that input port, the rate of the local clock changes. RegularToFast is a DiscreteClock actor that starts the clock with rate 1.0 at time 0.0, then changes the rate to 1.5 at time 5.0.

The clock value is modified by changing the value of the parameter *startTime* of a director. Modifying the parameter *startTime* any time during the simulation will set the current value of the clock to the value in the *startTime* parameter. The SlowWithOffset composite actor at the middle right has a port parameter called *clockValue*, and its director's *startTime* is set to the expression `clockValue` to reference the port parameter. Whenever a new value arrives at that port parameter, the clock value gets set. The Offset actor at the left sets the clock to -5.0 at environment time 0.0, to 2.5 at environment time 10.0, and to 10.0, again at environment time 10.0. The latter update leverages the superdense time model in Ptolemy II to instantaneously change the clock value from one value to another, as indicated by the vertical dashed line segment in Figure 10.2.

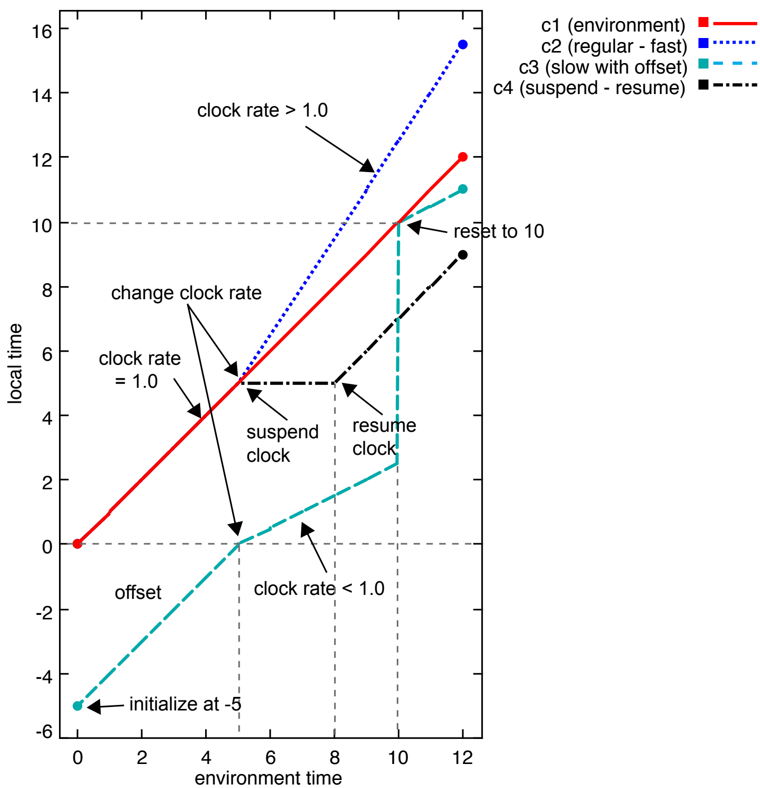


Figure 10.2: Clocks progressing at different rates relative to environment time.

The SupendAndResume actor at the lower right is a modal model where the clock of the inside Continuous director is suspended when the state machine is not in the *active* state, resulting in the horizontal segment of the dash-dot line in Figure 10.2. Notice that the transition entering the *active* state is a history transition to prevent the local clock from being reset to its start time (if given) or the environment time (if a start time is not given).

The local time of a director can be plotted by using a CurrentTime actor with the *useLocalTime* parameter set to true (which is the default). If the *useLocalTime* parameter is set to false, then the output produced will be the environment time of the top level of the model. The time lines in Figure 10.2 are obtained by periodically triggering these CurrentTime actors.

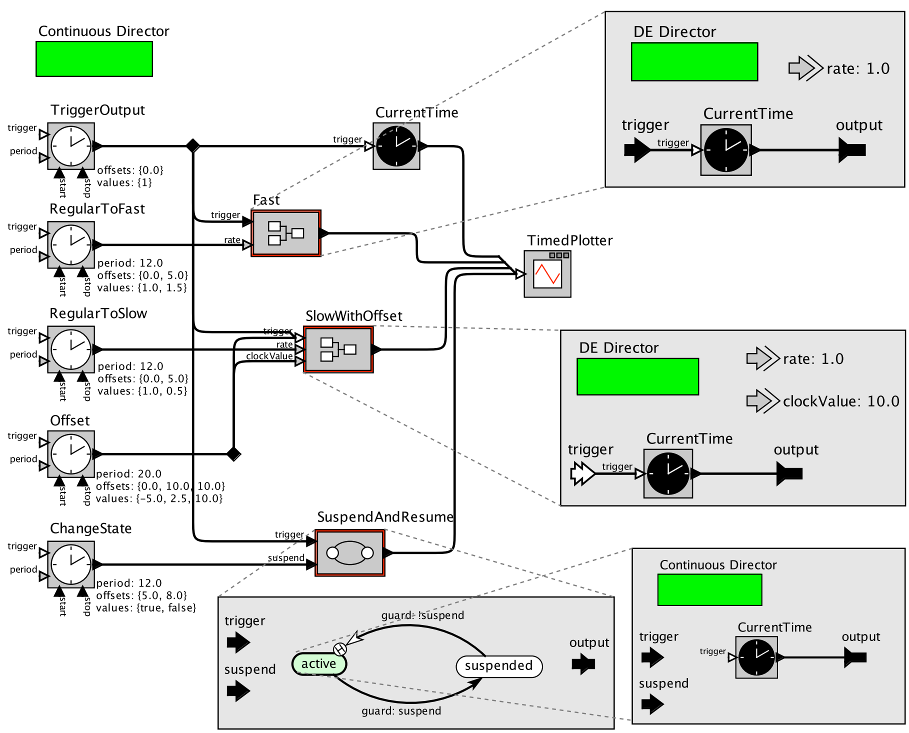


Figure 10.3: The model that generates the plot of Figure 10.2. [online]

10.2 Clock Synchronization

Many distributed systems rely on a common notion of time. A brute-force technique for providing a common notion of time is to broadcast a clock over the communication network. Whenever any component needs to know the time, it consults this broadcast clock. A well-implemented example of this is the **global positioning system** (**GPS**), which with some care can be used to synchronize widely distributed clocks to within about 100 nanoseconds. This system relies on atomic clocks deployed on a network of satellites, and careful calculations that even take into account relativistic effects. GPS, however, is not always available to systems (particularly indoor systems), and it is vulnerable to spoofing and jamming. More direct and self-contained realizations of broadcast clocks may be expensive and difficult to implement, since lack of control over communication delays can render the resulting clocks quite inaccurate. In addition, avoiding brittleness in such systems, where the source of the clock becomes a single point of failure that can bring down the entire system, may be expensive and a significant engineering challenge (Kopetz, 1997; Kopetz and Bauer, 2003).

A more modern technique that improves robustness and precision is to use **precision time protocols** (**PTP**) to provide **clock synchronization**. Such protocols keep a network of loosely coupled clocks synchronized by exchanging time-stamped messages that each clock uses to make small corrections in its own rate of progress. This technique is more robust, because sporadic failures in communication have little effect, and even with permanent failures in communication, clocks can remain synchronized for a period of time that depends on the stability of the clock technology.

This technique is also usually more precise than what is achieved by a broadcast clock; for most such protocols, the achievable precision does not depend on the communication delays, but rather instead depends on the *asymmetry* of the communication delays. That is, if the latency of communication from point A to point B is exactly the same as the latency of the communication from point B to point A, then perfect clock synchronization is theoretically possible. In practice, such protocols can come quite close to this theoretical limit over practical networks. The White Rabbit project at CERN, for example, claims to be able synchronize clocks on a network spanning several kilometers to under 100 picoseconds (Gaderer et al., 2009). This means that if you simultaneously ask two clocks separated by, say, 10 kilometers of networking cable, what time it is, their response will differ by less than 100 picoseconds. Over standard Ethernet-based local area networks, it is routine today to achieve precisions well under tens of nanoseconds using a PTP known as IEEE 1588 (Eidson, 2006). Over the open Internet, it is common to use a PTP known as NTP (Mills, 2003) to achieve precisions on the order of tens of milliseconds.

Typically, one or several master clocks are elected (and reelected in the event of failure), and slaves synchronize their clocks to the master by messages sent over the network. This guarantees a common notion of time across all platforms, with a well-defined error margin. The following example simulates the effects of imperfect clock synchronization.

Example 10.3: In electric power systems, a transmission line may span many kilometers. When a fault occurs, for example due to a lightning strike, finding the location of the fault may be very expensive. Hence, it is common to estimate the location of the fault based on the time that the fault is observed at each end of the transmission line. Assume a transmission line of length 60 kilometers between substation A and substation B. When a fault occurs, both substations will experience an observable event. Assuming that electricity travels through the transmission line

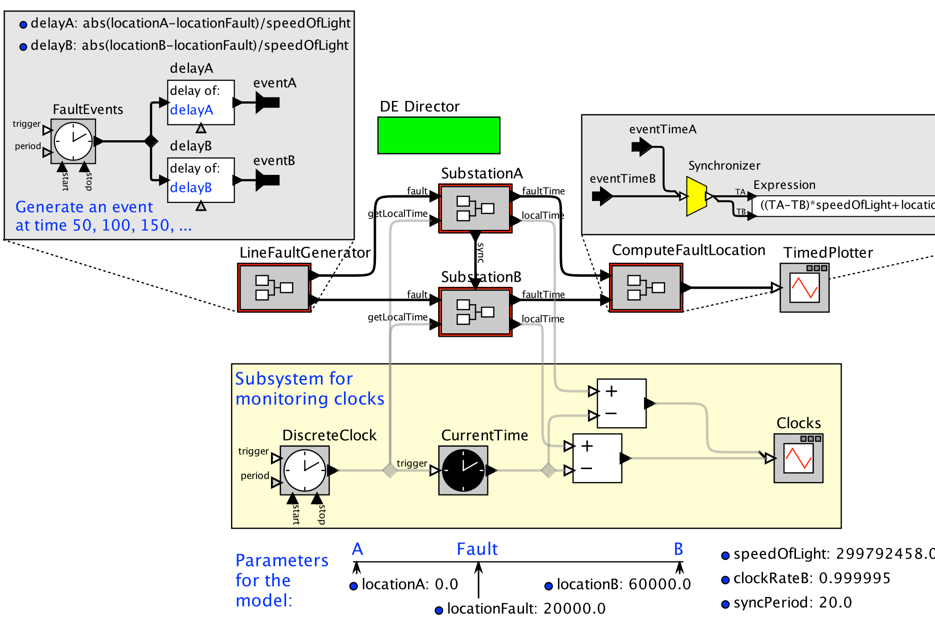


Figure 10.4: Line fault detection model. [online]

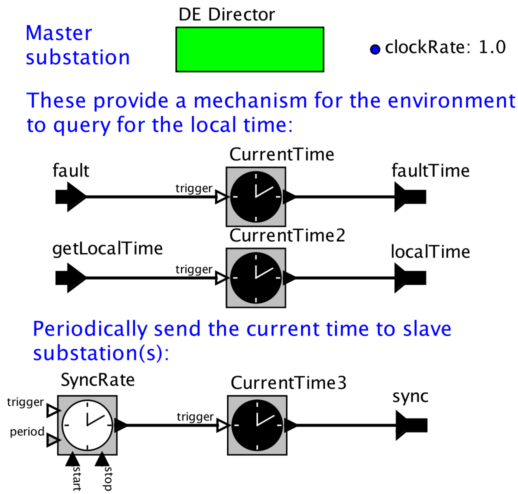


Figure 10.5: Line fault detection — Substation A, the clock master.

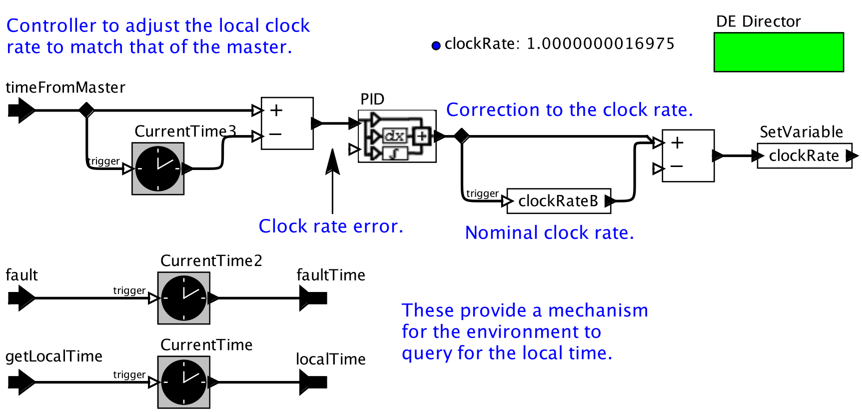


Figure 10.6: Line fault detection — Substation B, the clock slave.

at a known speed, then the time difference between when substation A observes the event and substation B observes the event can be used to calculate the location of the event.

Let X denote the location of the fault event along the transmission line (the distance from substation A). Then X satisfies the following equations,

$$\begin{aligned} s \times (T_A - T_0) &= X \\ s \times (T_B - T_0) &= D - X \end{aligned}$$

where T_0 is time of fault (which is unknown), T_A is the time of fault detection at substation A, T_B is the time of fault detection at substation B, s is the speed of propagation along the transmission line (the speed of light), D is the distance from A to B, and $D - X$ is the distance from B to the fault. Subtracting the above equations and solving for X yields

$$X = ((T_A - T_B) \times s + D)/2.$$

Of course, this calculation is only correct if the clocks at the two substations are perfectly synchronized. Suppose that the substations use a PTP to synchronize their clocks, and that substation A is the master. Then periodically, A and B will exchange messages that can be used to compute the discrepancy between their clocks. To see how this is done, see the sidebar on page 367 or Eidson (2006). Here, we will assume that this done perfectly (something that is only possible if the communication latency between A and B is perfectly symmetric). We focus in this model only on the effects of the control strategy that uses this information to adjust the clock of substation B. The model shown in Figure 10.4 shows SubstationA periodically sending its local time to SubstationB, where the period is given by the *syncPeriod* parameter, set to 20.0 seconds.

In that model, the LineFaultGenerator produces faults at times 50, 100, 150, etc., and the fault is assumed to occur 20 kilometers from substation A. The substation actors send the local times at which they observe the faults to a ComputeFaultLocation composite actor, whose task it is to determine the fault location using the above formulas. Since the measured fault times arrive at the ComputeFaultLocation actor at different times, a Synchronizer is used to wait until one data value from each substation has been received before it will do a calculation. Note that inputs will be misaligned if one of the substations fails to detect the event and provide an input, so a more realistic model needs to be more sophisticated.

The substation models are depicted in Figures 10.5 and 10.6. SubstationA is modeled very simply as, from top to bottom, responding to a *fault* input by sending the time of the fault, responding to a *getLocalTime* input with the local time, and periodically sending the local time to the *sync* output.

Substation B is a bit more complicated. When it receives a *sync* signal from the master, it calculates the discrepancy with its local clock; this calculation is not realistic, since there is an unknown time delay in receiving the *sync* signal, but PTP protocols take care of making this calculation, so this detail is not modeled here. Instead, the model focuses on what is done with the information, which is to use a PID controller to generate a correction to the local clock. The correction is added to the local clock rate and then stored in the *clockRate* parameter using a SetVariable actor. The director's clock uses this same parameter for the rate of its clock, so each time a correction is made, the rate of the local clock will change.

Figure 10.7 shows the simulation result. The upper plot shows the errors in the clocks. Since substation A is the master, it has no error, so its error is a constant zero. At the start of simulation, the clock of B is drifting linearly with respect to A. At 20 seconds, B receives the first *sync* input, and the PID controller provides a correction that reduces the rate of drift. At 40 seconds, another *sync* signal further reduces the rate of drift. The lower plot shows the estimated locations of the faults occurring at times 50, 100, 150, etc. The correct fault location is 20 km, so we can see that as the clocks get synchronized, the estimate converges to 20 km.

Figure 10.8 shows what happens if there is no clock synchronization (the *sync* signal never arrives at B). In this case, the clock of B drifts linearly with respect to A, and the error in the estimated fault location grows without bound.

10.3 Modeling Communication Delays

In design-space exploration, designers evaluate whether their designs work well on a given architecture. Part of the architecture is the communication network, which introduces delays that can affect the behavior of a system. A communication network introduces delay. It is straightforward to model constant communication delays that are independent of one another, but it is much more interesting (and realistic) to take into account shared resources, which result result in correlated and variable delays. We begin with the simpler models, and then progress to the more interesting ones.

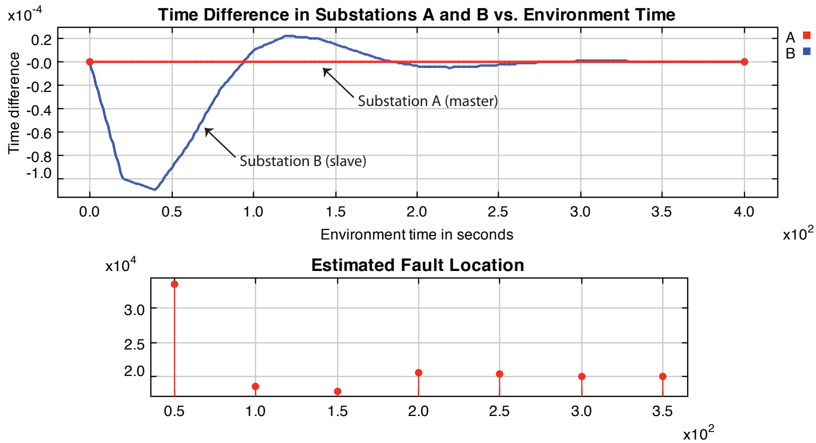


Figure 10.7: Line fault detection with clock synchronization.

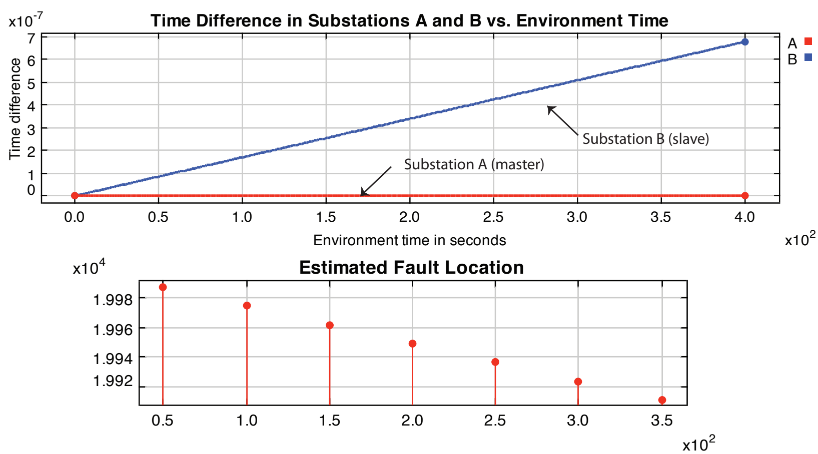
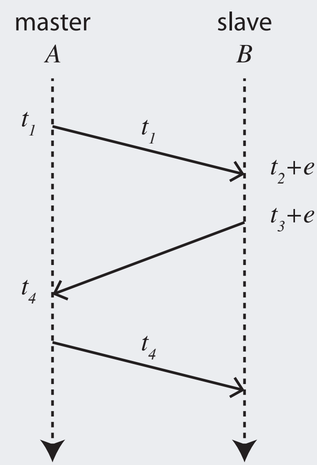


Figure 10.8: Line fault detection without clock synchronization.

Sidebar: Precision Time Protocols



The figure at the right shows how a typical PTP works. The clock master A initiates an exchange of messages. The first message is sent at time t_1 (by the master clock) and contains the value of t_1. That message is received by the slave B at time t_2 (by the master clock), but since the slave does not have access to the master clock, the slave records the time t_2' that it receives the message according to its own clock. If its clock is off by e vs. the master clock, then

$$t_2' = t_2 + e.$$

The slave responds by sending a message back to the master at time t_3' according to its clock, or t_3 according to the master's clock, so

$$t_3' = t_3 + e.$$

The master receives this second message at t_4, and replies with a third message containing the value of t_4. The slave now has t_1, t_2', t_3', and t_4. Now notice that the round trip communication latency (the time that a message from A to B and a reply message spend in transport) is

$$r = (t_2 - t_1) + (t_4 - t_3) = (t_4 - t_1) - (t_3 - t_2).$$

At B, this value can be calculated even though t_2 and t_3 are not known, because $(t_3 - t_2) = (t_3' - t_2')$, and slave B has t_2', and t_3'. If the communication latencies are symmetric, then the one way latency is $r/2$. To correct the clock of B, we simply need to estimate e. This will tell us whether the clock is ahead or behind, so we can slow it down or speed it up, respectively. If the communication channel has symmetric delays (i.e. $t_2 - t_1 = t_4 - t_3$), then a very good estimate is given by

$$\tilde{e} = t_2' - t_1 - r/2.$$

In fact, if the communication latency is exactly symmetric, then $\tilde{e} = e$, the exact clock error. B can now adjust its local clock by $\tilde{e}$.

10.3.1 Constant and Independent Communication Delays

In the DE domain, network delays that are independent of one another can be easily modeled using the TimeDelay actor.

Example 10.4: The line fault detector of Figure 10.4 idealizes the calculation of the clock error in substation B. In practice, calculating clock discrepancies is not trivial. A typical technique implemented in a PTP is described in the sidebar on page 367 and implemented in the model in Figure 10.9.

In Figure 10.9, substation A periodically initiates a sequence of messages that are used to calculate the clock discrepancy. First, at master time t_1, it sends the value of this time to substation B. Substation B responds. Substation A responds to the response with the time t_4 that it receives the response. When substation B has received this final message, it has enough information to estimate the discrepancy between its clock and that of the master. The Synchronizer actor ensures that this estimate is only calculated after all the requisite information has been received.

Figure 10.9 has three TimeDelay actors that can model network latency in the communication of synchronization messages. Interestingly, if all three delays are set to the same value, even a rather large value such as 1.0 seconds, then the performance of the model in identifying the location of the fault is essentially identical to that of the idealized model. However, if, as shown, one of the delays is changed only slightly, to 1.0001, then the performance degrades considerably, as shown in Figure 10.10. The slave clock settles into a substantial steady-state error, and the estimated fault location converges to approximately 13 kilometers, quite different from the actual fault location at 20 kilometers. Clearly, if the communication channel is expected to be asymmetric, then the designer has work to do to improve the control algorithm. A different choice of parameters for the PID controller would probably help, but perhaps at the expense of lengthening the convergence time.

10.3.2 Modeling Contention for Shared Resources

In the model in Figure 10.9, each connection between actors has a fixed communication delay. This is not very realistic for practical communication channels, where the delay

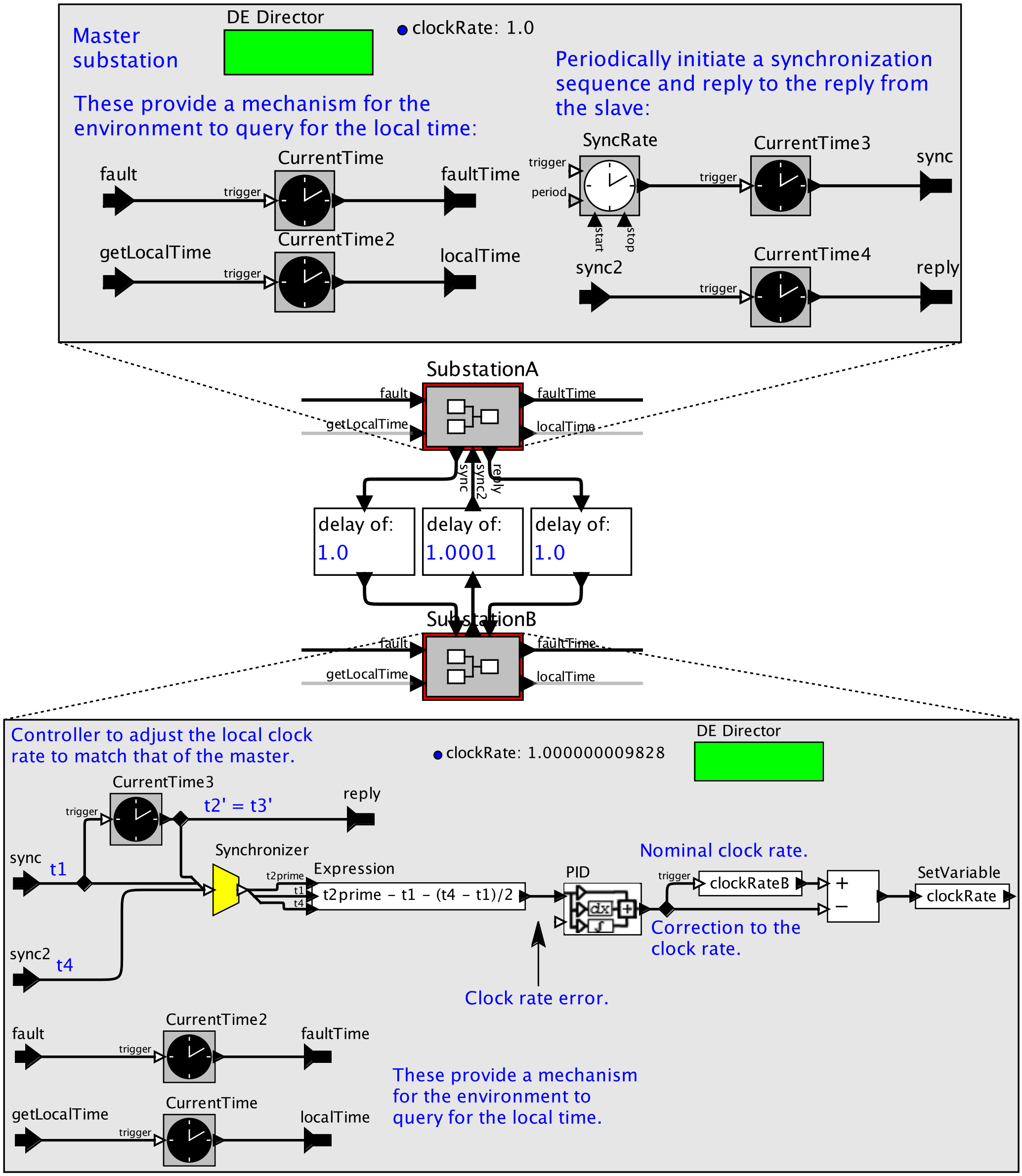


Figure 10.9: Line fault detection with communication delays in the PTP implementation. [online]

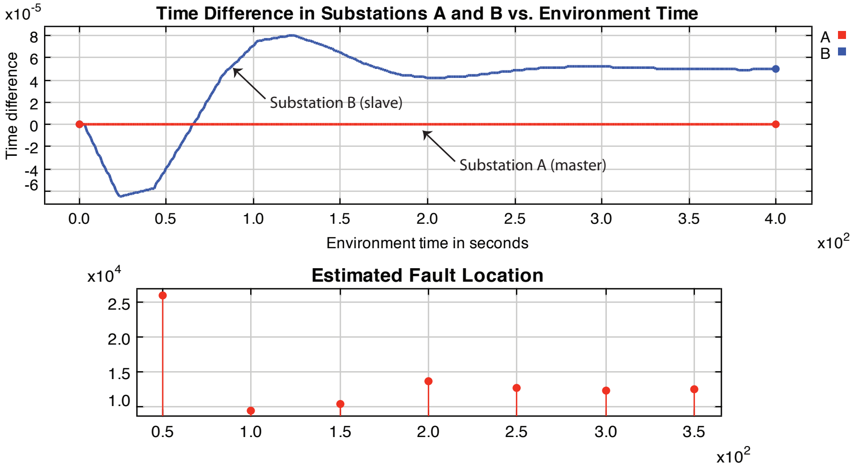


Figure 10.10: Line fault detection performs poorly with PTP when network latencies are asymmetric.

will depend on other uses of the channel. Most communication channels have shared resources (radio bandwidth, wires, buffer space in routers, etc.), and the latency through the network can vary significantly depending on other uses of these resources.

The model in Figure 10.9 could be modified to route all the messages through a single, more elaborate network model. However, this leads to considerable modeling complexity. Suppose for example that we wish to model the network using a single Server actor, perhaps the most basic model for a shared resource. Then all messages that traverse the channel would have to be merged into a single stream to feed to the Server actor. These streams would then have to be separated after emerging from the Server actor, so destination addresses would have to be encoded in the messages before the streams are merged. The model suddenly becomes very complicated.

Fortunately, Ptolemy II has a much cleaner mechanism for handling shared resources. We use **aspect-oriented modeling** (**AOM**), which is based on aspect-oriented programming (Kiczales et al., 1997), to map functionality to implementation. This way of associating functional models with implementation models and schedulers was introduced in Metropolis (Balarin et al., 2003), where the mechanism was called a **quantity manager**. In Ptolemy II, an **aspect** is an actor that manages a resource; it is associated with the actors and ports that share the resource. In a simulation run, the aspect actor schedules the use of the resource. The association between the resource and the users of the resource is done via parameters, not by direct connections through ports. As a consequence, aspects are added to an existing model without changing the interconnection topology of the existing model. The next example shows how **communication aspects** can be used to cleanly model shared communication resources.

Example 10.5: Figure 10.11 shows a variant of the line fault detector model where we have dragged into the model a communication aspect called **Bus**. In this example, the PTP communications between SubstationA and SubstationB use the shared bus. This is indicated in the figure by the annotation "Aspects: Bus" on the input ports, and by red fill in the port icon.

The Bus has a *serviceTime* parameter that specifies the amount of time that it takes a token to traverse the channel. During that time, the Bus is busy, so any further attempts to use the Bus will be delayed. The Bus therefore acts like a Server actor with an unbounded buffer, but since it is an aspect, there is no need for the model to explicitly show all the communication paths passing through a single instance of a Server.

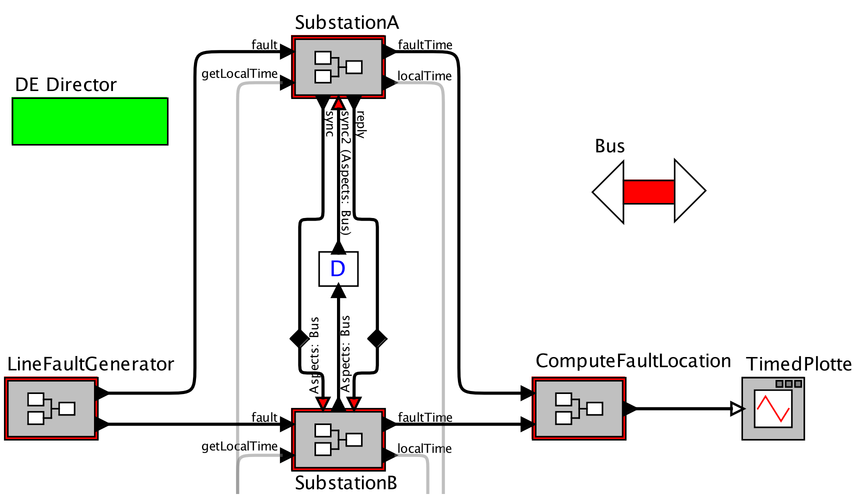


Figure 10.11: Line fault detection where communication uses a shared bus, modeled using a communication aspect. [online]

In this example, the communications latencies are symmetric, because there is no contention for the bus. Hence, the line fault detection algorithm performs well, behaving similarly to Figure 10.7. If, on the other hand, you enable use of the bus for other communication paths in the model, such as at the input ports of ComputeFaultLocation, then the performance will degrade considerably, because contention for the bus will introduce asymmetries in the communication latencies.

To use an aspect for modeling communication, simply drag one into the model from the library and assign it a meaningful name. The Bus, along with several others that are (as of this writing) still rather experimental, can be found in the `MoreLibraries→Aspects` library.

An aspect is a decorator, which means that it endows elements of the model with parameters (see sidebar on page 373). In the case of the Bus, it decorates ports with an *enable* and *messageLength* parameter, as shown in Figure 10.12. When an input port has the Bus enabled, then messages sent to that input port will be delayed by at least the product of the *messageLength* parameter of the port and the *serviceTimeMultiplicationFactor* parameter of the Bus. The delay is *at least* this, because if the bus is busy when the message is sent, then the message has to wait until the bus becomes free.

You can add any number of aspects to a model. Each is a decorator, and each can be independently enabled. If an input port enables multiple communication aspects, then

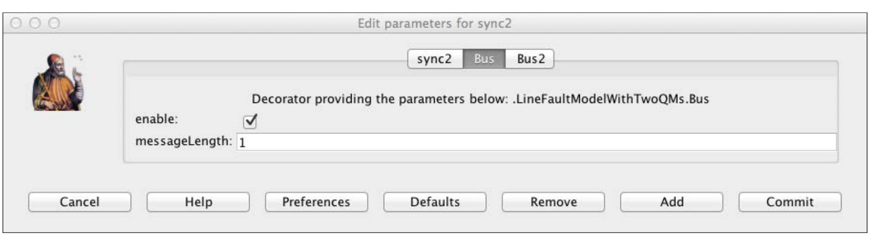


Figure 10.12: The Bus aspect decorates ports with an *enable* and *messageLength* parameter. This figure shows a parameter editor for a port in model in Figure 10.14, which has two busses.

those aspects mediate the communication in the order in which the aspects are enabled. Hence, aspects may be composed.

Example 10.6: In Figure 10.14, a second bus has been added to the model, and the communication from SubstationB to SubstationA traverses Bus and Bus2, in that order, as you can see from the annotation on the *sync2* input port to SubstationA. As a consequence, the communication latencies become asymmetric, and the line fault detection algorithm performs poorly, yielding results similar to those in Figure

Sidebar: Decorators

A **decorator** in Ptolemy II is an object that adds to other objects in the model parameters, and then uses those parameter values to provide some service. The simplest decorator provided in the standard library is the **ConstraintMonitor**, which can be found in the `Utilities→Analysis` library. The ConstraintMonitor is an attribute that, when inserted in model, adds a parameter called *value* to actors in the model. The ConstraintMonitor keeps track of the sum of all the values that are set for actors in the model, displays that sum in its icon, and compares that sum against a *threshold*.

An example use of ConstraintMonitor is shown in Figure 10.13, where a ConstraintMonitor has been dragged into a model with three actors and renamed "Cost." Once that ConstraintMonitor is in the model, then the parameter editing window for each actor acquires a new tab, as shown at the top of the figure, where the label on the tab matches the name of the ConstraintMonitor. The user can enter a cost for each actor in the model, and the ConstraintMonitor will display the total cost in its icon.

The ConstraintMonitor has a parameter *threshold*, which specifies a limit on the sum of the values. When the total approaches the limit, the color of the ConstraintMonitor icon changes to yellow. When the total hits or exceeds the limit, if the *warningEnabled* parameter is true, then the user is warned. The default value for *threshold* is `Infinity`, which means no limit.

The ConstraintMonitor has two other parameters, as shown at the bottom of Figure 10.13. If *includeOpaqueContents* is true, then actors inside opaque composite actors will also be decorated. Otherwise, they will not be decorated. If *includeTransparents* is true, then transparent composite actors will be decorated. Otherwise, they will not be.

There are many other uses for decorators. A director can be a decorator. The aspects described in this chapter are decorators.

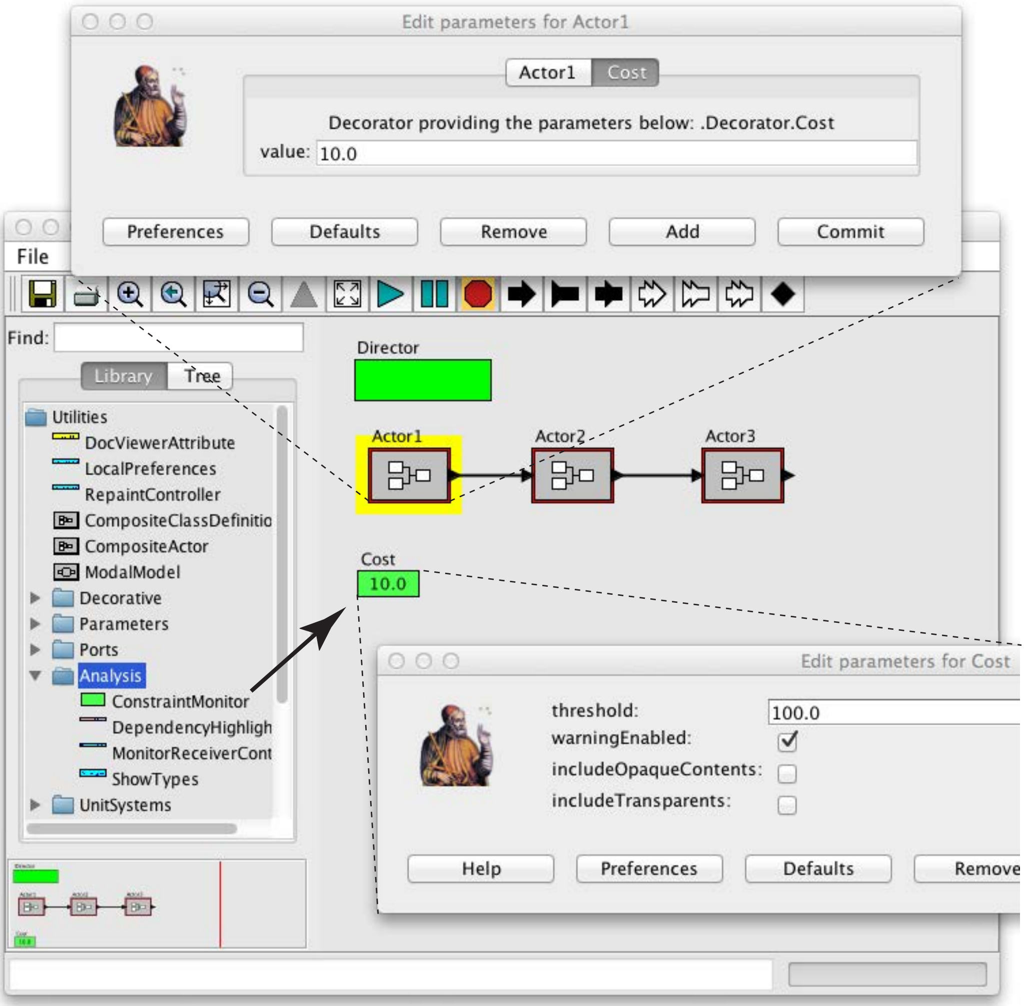


Figure 10.13: A decorator in Ptolemy II is an object that adds to other objects in the model parameters, and then uses those parameter values to provide some service. In this example, a ConstraintMonitor (which has been renamed Cost) is monitoring the total cost of components in the model, checking them against a threshold of 100.0.

10.10. The *decoratorHighlightColor* parameter of Bus2 has been changed from red to green, resulting in green highlighting of both the port and the bus icon.

10.3.3 Composite Aspects

The aspects discussed in the previous section are like atomic actors; their logic is defined in a Java class. For a more flexible way of describing communication aspects, we use the **CompositeCommunicationAspect**. This actor can be found in Ptolemy under `MoreLibraries→Aspects`.

Example 10.7: Figure 10.15 shows the bus example implemented using a CompositeQuantityManager. In this case, the bus behavior is modeled using a discrete-event subsystem with a Server actor. Requests to use the bus queue up at the input

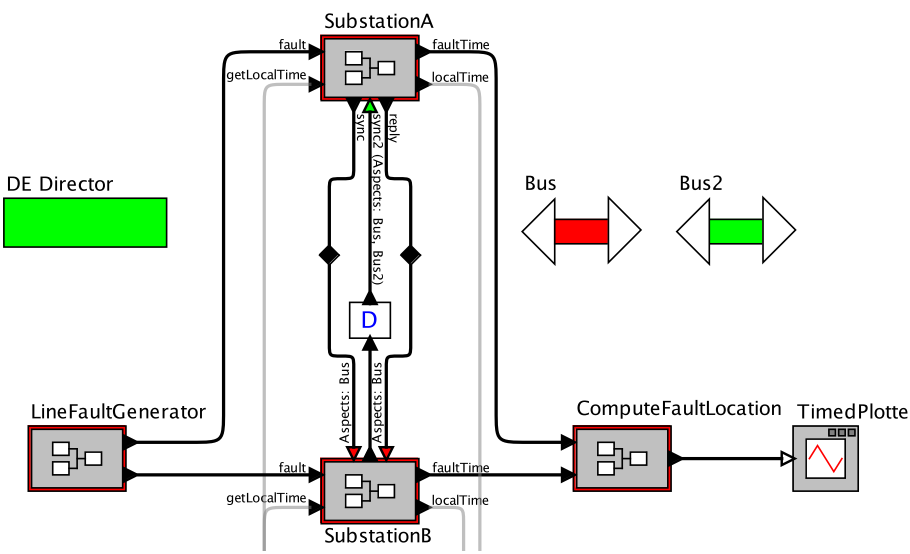


Figure 10.14: Line fault detection where one of the communications traverses two busses, yielding asymmetric communication. [online]

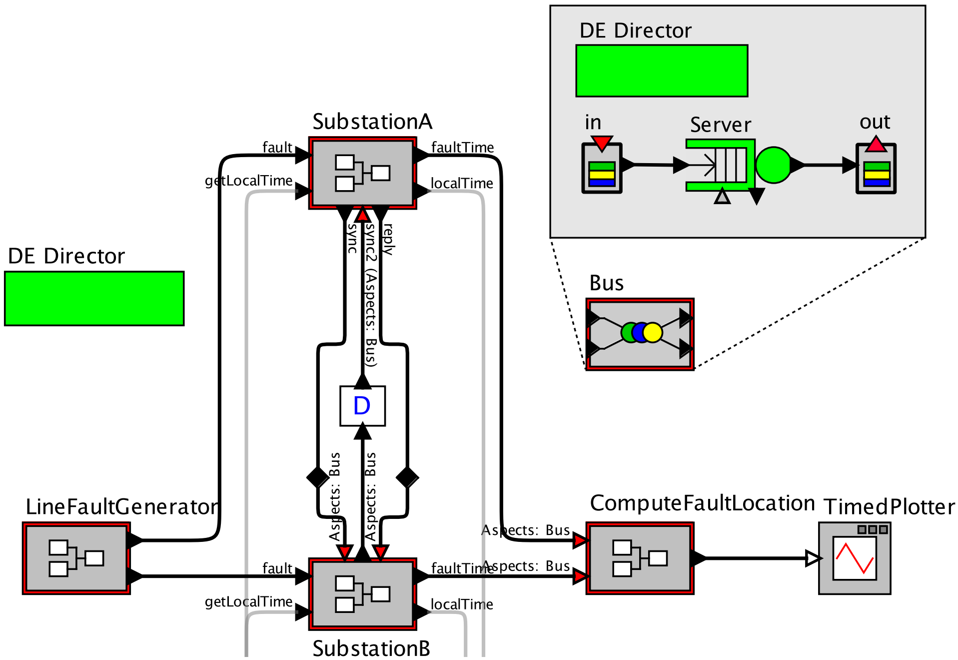


Figure 10.15: A bus implemented as a composite communication aspect. [online]

to the server. When the server becomes free, the first queued input is delayed by the *serviceTime*. The behavior is identical to that of the atomic Bus aspect.

Since a composite communication aspect is simply a Ptolemy II model, we have a great deal of freedom in its design.

Example 10.8: In the example of Figure 10.15, the bus is being used not only for the communication between SubstationA and SubstationB, but also in the communication to the ComputeFaultLocation actor. Contention for the bus makes the communication latencies asymmetric, degrading the performance of the clock syn-

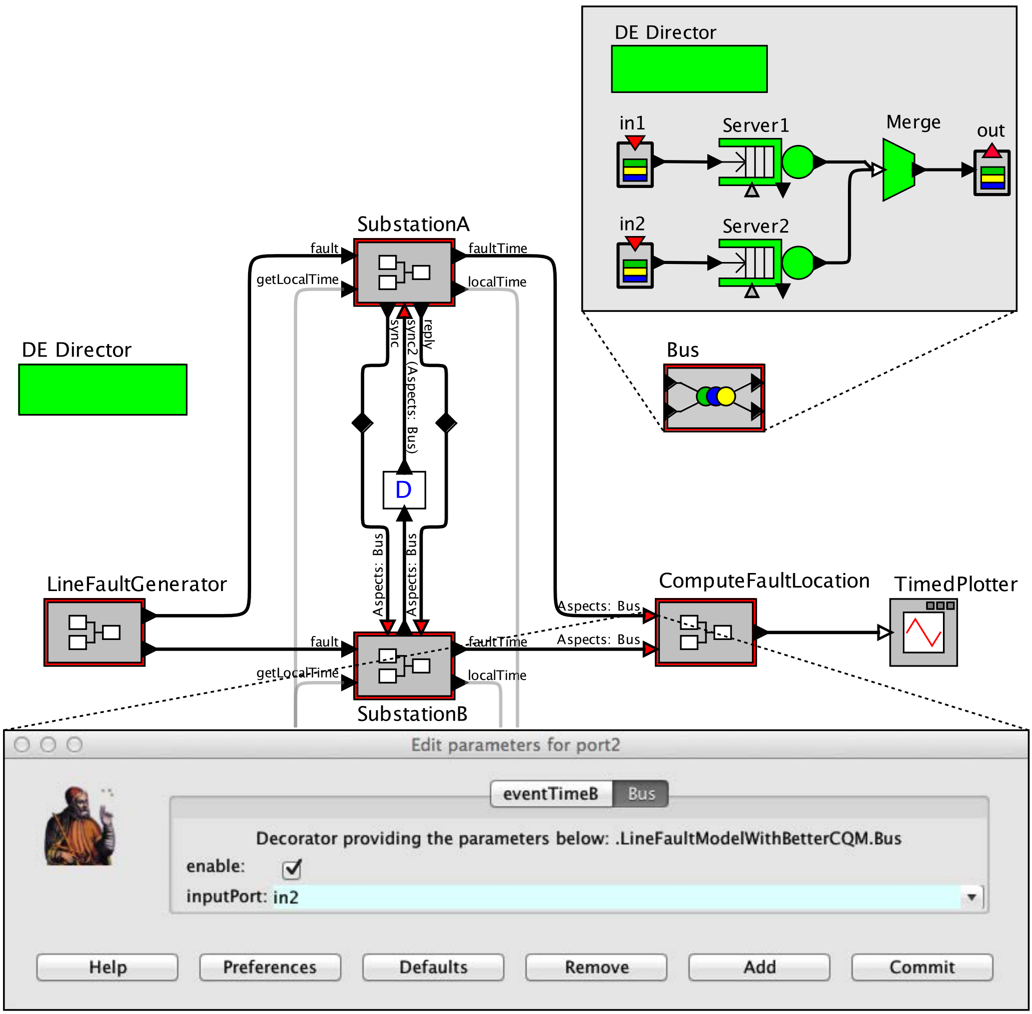


Figure 10.16: A network that reduces contention implemented as a composite aspect. At the bottom is shown how the decorator parameters of a port are used to select the port of the aspect that handles the communication. [online]

chronization, and resulting in very poor performance in computing the fault location.

We can improve the performance with a better network, as shown in Figure 10.16. In that figure, we have modified the Bus so that it is now a more sophisticated network with two input ports and two distinct servers. By routing communication to the two servers, contention can be reduced. Each input port involved in a communication specifies which input port, *in1* or *in2*, of the aspect should handle the communication. At the bottom of the figure is shown how the decorator parameters of a port are used to select the port of the aspect that handles the communication. In the figure, the top input port of ComputeFaultLocation is using *in2*. If the bottom port also uses *in2*, and the ports handling the communication between SubstationA and SubstationB use *in1*, then contention is reduced enough to deliver excellent performance, similar to that in Figure 10.7.

10.4 Modeling Execution Time

In addition to modeling network characteristics such as communication delays, one might also want to model **execution time**, the time it takes it takes an actor to perform its function on a particular implementation platform. The joint modeling of an application's functionality and its performance on a model of the implementation platform is a very powerful tool for **design-space exploration**. It makes it much easier to understand the impact of choices in networking infrastructure and processor architecture.

In a discrete-event model, execution times can be simulated using a Server actor for each execution resource (such as a processor), where the service time is the execution time. Example 7.5 and Figure 7.8 illustrate this for a simple storage system. However, such models are difficult to combine with models of complex functionality.

Fortunately, Ptolemy provides **execution aspects**, which, like communication aspects, provide a form of aspect-oriented modeling. Execution aspects can be used to model contention for resources that are required to execute an application model. The mechanisms are similar to those of the communication aspects, as illustrated in the next example.

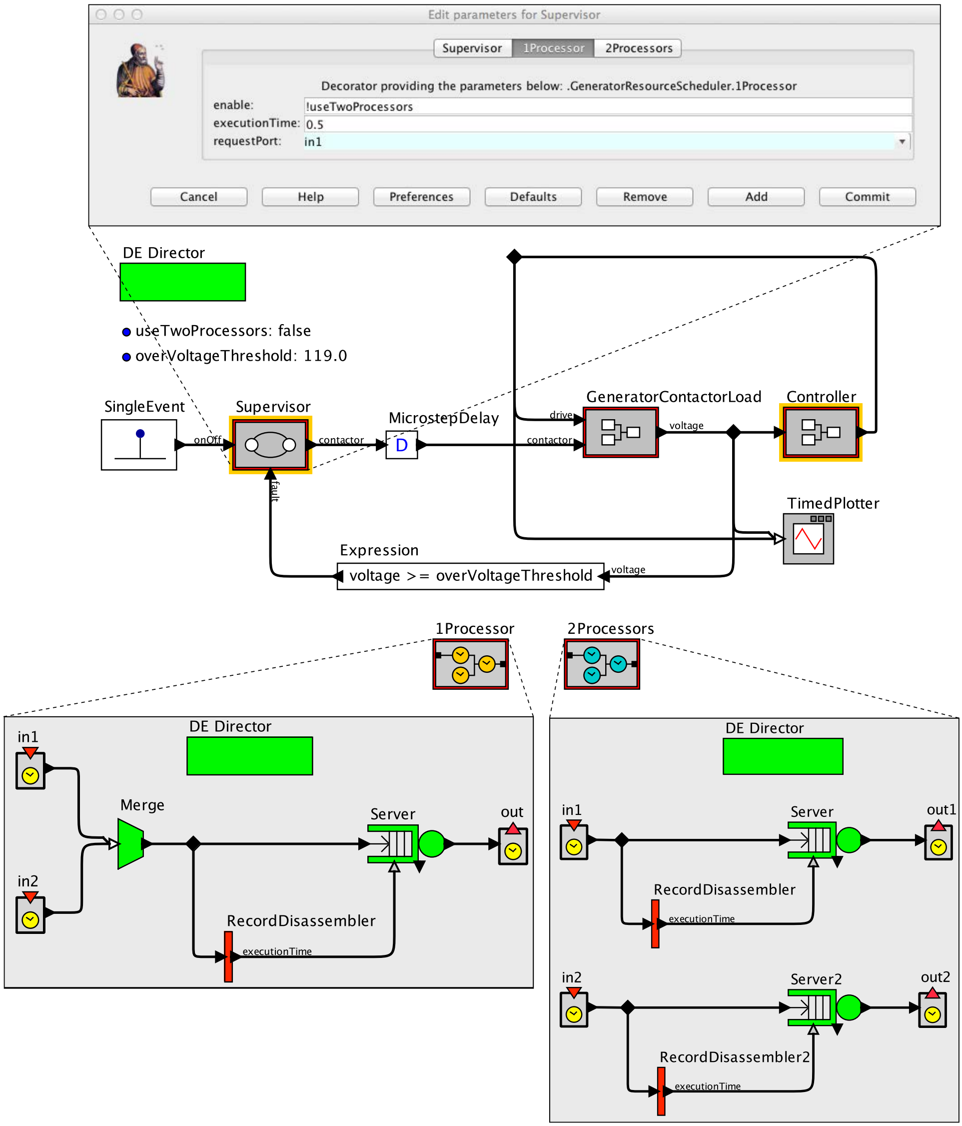


Figure 10.17: A model with two alternative execution aspects, one that models a one-processor execution platform and one that models a two-processor execution platform. [online]

Example 10.9: A variant of the generator model considered in Section 1.9 is shown in Figure 10.17. This model includes two possible implementation platforms, one with one processor, one with two processors, shown at the bottom of the figure. These two implementation platforms are modeled using the **CompositeResourceScheduler** actor, found in `MoreLibraries→ResourceScheduler`.

In this model, the Supervisor and Controller actors execute on one of the two processor architectures. Which one is determined by the value of the *useTwoProcessors* parameter in the model. If the value of this parameter is true, then the 2Processor aspect will be used to execute Supervisor and Controller. Otherwise, 1Processor will be used.

When this model is executed, the behavior changes with the value of *useTwoProcessors*. When two processors are used, there is no contention for resources, since Supervisor and Controller can execute simultaneously, as modeled by the two Server actors at the lower right in the figure. However, when only one processor is used, the Supervisor and Controller compete for the use a single processor, as modeled by the single server at the lower left. In that case, there is more delay in one of the two feedback loops, which changes the dynamics of the model. In particular, with certain choices of parameters and test conditions, the choice of processor architecture could affect whether the over voltage protection conditions shown in the plot in Figure 1.11 occurs.

Notice the use of RecordDisassembler actors in the composite execution aspects. The input to the submodel in the composite aspect is a record that contains the values of the decorator parameter *executionTime* of the actor requesting an execution resource. This execution time is extracted from the record and becomes the service time of the Server.

10.5 Ptides for Distributed Real-Time Systems

So far, this chapter has focused on modeling and simulating timing behavior in system implementations. Another role for timed models, however, is to *specify* timing behavior. That is, a timed model may give the *required* behavior of an implementation without completely describing the implementation. Towards this end, we focus for the remainder of this chapter on a programming model for distributed real-time systems called **Ptides**.*

Ptides models are designed to solve the problem identified in Example 10.9 above, where the behavior of a system depends on the details of the hardware and software platform that executes the system. A key goal in Ptides is to ensure that every *correct* execution of a system delivers exactly the same dynamic behavior.

Example 10.10: In Example 10.9, choosing to execute Supervisor and Controller on a single processor yields different dynamic behavior than choosing two processors. If these were Ptides models, the two behaviors would be identical, as long as the processor resources were sufficient to deliver a *correct* execution. Moreover, the execution times of the Supervisor and Controller will also not affect the dynamics until they get so large that a correct execution is no longer possible. Hence, Ptides has the potential to reduce the sensitivity that a system has to implementation details. Behavior is exactly the same over a range of implementations.

A Ptides model is a DE model with certain constraints on time stamps. Ptides is used to design event-triggered distributed real-time systems, where events may be occurring regularly (as in sampled-data systems) or irregularly. A key idea in Ptides is that, unlike DE, time stamps have a relationship with real time at sensors and actuators (which are the devices that bridge the cyber and the physical parts of cyber-physical systems). A second key idea in Ptides is that it leverages network time synchronization (Johannessen, 2004; Eidson, 2006) to provide a coherent global meaning to time stamps in distributed systems. The most interesting, subtle, and potentially confusing part about Ptides is the relationship between multiple time lines. But herein also lies its power.

10.5.1 Structure of a Ptides Model

A Ptides model consists of one or more **Ptides platforms**, each of which models a computer on a network. A Ptides platform is a composite actor that contains actors representing sensors, actuators, and network ports, and actors that perform computation and/or

*The name comes from the somewhat tortured acronym for "programming temporally integrated distributed embedded systems." The initial "P" is silent, as in Ptolemy, so the name is pronounced "tides."

Sidebar: Background of Ptides

Ptides leverages network **time synchronization** (Johannessen, 2004; Eidson, 2006) to provide a coherent global temporal semantics in distributed systems. The Ptides programming model was originally developed by Yang Zhao as part of her Ph.D. research (Zhao et al., 2007; Zhao, 2009). Zhao showed that, subject to assumed bounds on network latency, Ptides models are deterministic. The case for a time-centric approach like Ptides is elaborated by Lee et al. (2009b), and an overview of Ptides and an application to power-plant control is given by Eidson et al. (2012),

A number of implementations followed the initial work. A simulator is described by Derler et al. (2008), and an execution policy suitable for implementation in embedded software systems by Feng et al. (2008) and Zou et al. (2009b). Zou (2011) developed **PtidyOS**, a lightweight microkernel implementing Ptides on embedded computers, and a code generator producing embedded C programs from models. Matic et al. (2011) adapted PtidyOS and the code generator to demonstrate their use in smart grid technologies.

Feng and Lee (2008) extended Ptides with incremental checkpointing to provide a measure of fault tolerance. They showed conditions under which rollback can recover from errors, observing that the key constraint in Ptides is that actuator actions cannot be rolled back. Ptides has also been used to coordinate real-time components written in Java (Zou et al., 2009a).

A technique similar to Ptides was independently developed at Google for managing distributed databases (Corbett et al., 2012). In this work, clocks are synchronized across data centers, and messages sent between data centers are time stamped. The technique provides a measure of determinacy and consistency in database accesses and updates.

Assuming that the network latency bounds are met, a correct implementation of Ptides is deterministic in that a sequence of time-stamped events from sensors always results in a unique and well-defined sequence of time-stamped events delivered to actuators. However, this determinism does not provide any guarantee that events are delivered to actuators *on time* (prior to the deadline given by the time stamp). The problem of determining whether events can be delivered on time to actuators is called the **schedulability** problem. The question is, given a Ptides model, does there exist a schedule of the firing of actors such that deadlines are met. Zhao (2009) solved this problem for a limited class of models. The problem is further discussed by Zou et al. (2009b), and largely solved by Matsikoudis et al. (2013).

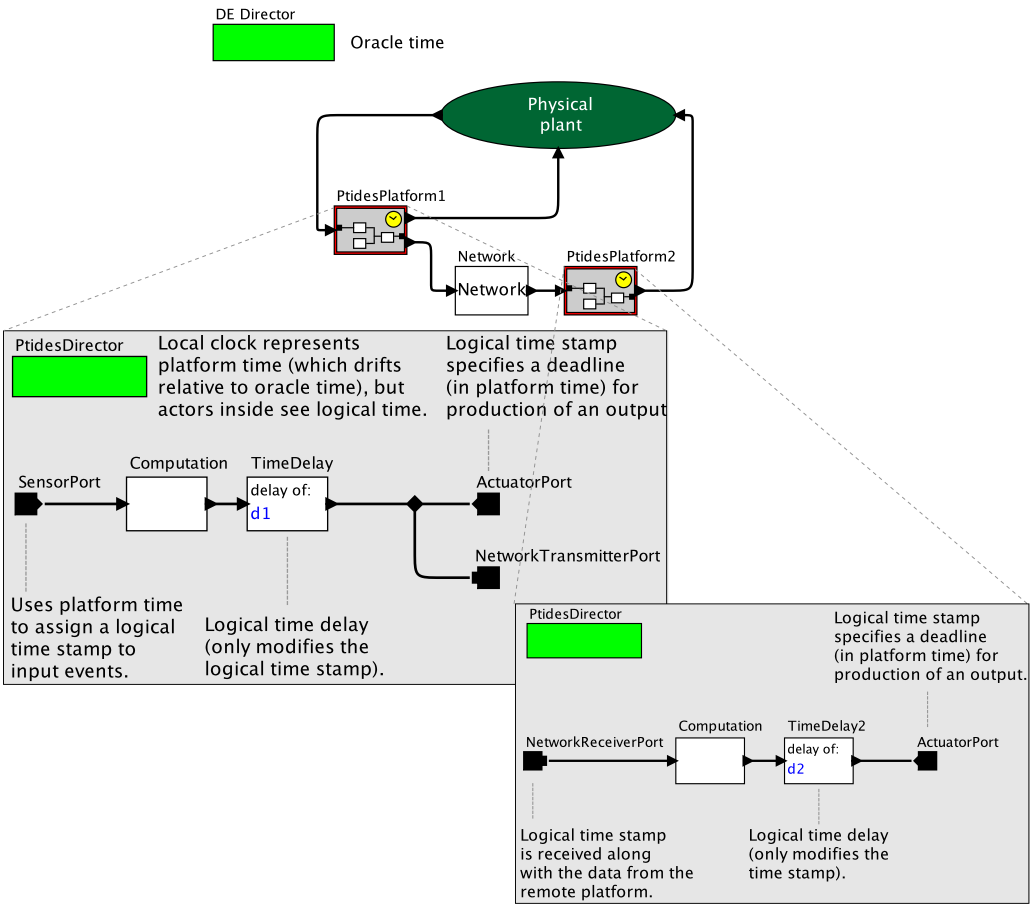


Figure 10.18: Ptides model with two Ptides platforms, sensor, actuator and network ports. [online]

modify time stamps. A Ptides platform contains a PtidesDirector and represents a single device in a distributed cyber-physical system, such as a circuit board containing a microcontroller and some set of sensor and actuator devices. For simulation purposes, a Ptides platform is placed within a DE model that models the physical environment of the platform.

Example 10.11: A simple Ptides model is shown in Figure 10.18. This model has two platforms connected to a physical plant (via sensors and actuators) and to a network. The top-level director is a DE director, whereas the platform directors are Ptides directors. The physical plant may internally be a Continuous model.

Ptides models leverage Ptolemy's multiform time mechanism. A common pattern in such models assumes the time line at the top-level of the model hierarchy represents an idealized physical time line that advances uniformly throughout the system. This time line cannot be directly observed by computational devices in the network, which must instead use clocks to approximately measure it. We refer to such an idealized time at the top level as the **oracle time**. In the MARTE time library, the same idealized concept of physical time is referred to simply as **ideal time** (André et al., 2007).

Within a platform, a local clock maintains a time line called **platform time**, which approximates oracle time. Platform time is **chronometric**, an imperfect measurement of oracle time. The builder of a Ptides model may choose to assume that platform time perfectly tracks oracle time or, more interestingly, to model imperfections in tracking and discrepancies across the network, as illustrated in Section 10.2 above.

Example 10.12: In the Ptides model of Figure 10.18, the top-level director's clock represents oracle time. The clocks of the Ptides directors represent platform time. These can be parameterized to drift with respect to each other and platform time and to have offsets.

A key innovation in Ptides, however, is that a second time line called **logical time** plays a key role in a platform. The notion of logical time in distributed systems was introduced by Lamport et al. (1978), and is applied in Ptides to achieve determinism in distributed real-time systems. Any actor that requests the current time from the PtidesDirector will be told about logical time, not about platform time. The only actors that have access to platform time are sensors, actuators, and network interfaces, i.e. the actors at the cyber-physical boundary. Specifically, when a sensor produces an event in a Ptides model, the time stamp of the event is a logical time value equal to the platform time at which the sensor takes its measurement. That is, Ptides binds logical time to physical time (as measured by platform time) at sensors.

Example 10.13: In the Ptides model of Figure 10.18, the sensor uses the platform time of PtidesPlatform1 to construct a time stamp for each event that it produces. Such an event represents a measurement made on the physical plant, and its (logical) time stamp is equal to the local measurement of time.

Let t_s be the platform time at which a measurement is made by a sensor. The event produced by that sensor actor will have logical time stamp t_s.

A TimeDelay actor, however, operates in logical time. It simply manipulates the logical time stamp. Platform time is not visible to it. If an input to a TimeDelay actor whose delay value is d_1 has (logical) time stamp t, then its output will have time stamp $t + d_1$.

Once an event from a sensor has been produced, it is processed by the Ptides model like any other discrete event in a DE system. That is, events with logical time stamps are processed by the PtidesDirector in time-stamp order, without particular concern for platform time or oracle time. Actors are fired as they would be in simulation. A key property of Ptides models is that this time-stamp-ordered processing of events is preserved despite the distributed architecture and imperfect clocks. This key property delivers determinism.

An actuator port inside a Ptides platform acts as an output from the platform. When it receives an event from the platform Ptides model, that event has a logical time stamp t. The actuator interprets t as a **deadline** relative to platform time. That is, an event with time stamp t sent to an actuator is a command to perform some physical action no later than the (platform) time equal to t. Hence, actuators, like sensors, also bridge logical and physical times.

Example 10.14: In the Ptides model of Figure 10.18, in PtidesPlatform1, assume the SensorPort produces an event with time stamp t_s. This represents the platform time at which a sensor measurement is made. Assume further that Computation is a zero-delay actor, and that it reacts to the event from SensorPort by producing an output event with the same time stamp t_s. The output of the TimeDelay actor, therefore, will have time stamp $t_s + d_1$, where d_1 is the delay of the TimeDelay actor. That event goes to the ActuatorPort, which interprets the time stamp $t_s + d_1$ as a deadline. That is, the actuator should produce its actuation at platform time no later than $t_s + d_1$.

By default, when executing a Ptides model, actors are assumed to be instantaneous (in platform time). Hence, the deadline at ActuatorPort in Figure 10.18 will never be violated. In fact, in the simulation, the ActuatorPort will be able to perform its actuation as early as platform time t_s. This is not very realistic, because any physical realization of this platform will incur some latency. It cannot react instantaneously to sensor events. More realistic simulation models can be constructed by combining the execution aspects of Section 10.4 with Ptides, but we will not do that here. Instead, here, we will assume that there is some variability in the latency introduced by the physical realization of the platform, but that the deadline will nevertheless be met. Verifying this is a schedulability problem.

The actuation of ActuatorPort in Figure 10.18 affects the physical plant, which in turn affects the SensorPort. There is a feedback loop, and the closed-loop behavior will be affected by the latency of the platform. If that latency is unknown or variable, then the overall closed-loop behavior of the system will be unknown or variable, yielding a nondeterministic model. To regain determinism, Ptides actuators can be configured to perform their actuation *at the deadline* rather than *by the deadline*. As long as events arrive at or before the deadline, the actuator will be able to produce its actuation deterministically, independent of the actual arrival time of the events, and hence independent of execution time variability. The response of PtidesPlatform1 to a sensor event will a *deterministic* actuator event (in platform time and oracle time). This makes the behavior of the entire closed-loop system independent of variability in execution times (and, as we will show below, network delays). To configure a Ptides actuator to provide this determinism, set the *actuateAtEventTimestamp* parameter of the *ActuatorPort* to `true`. Ptides, therefore, provides a mechanism to hide underlying uncertainty and variability (up to a failure threshold, when deadlines are not met), yielding deterministic closed-loop behavior.

A natural question arises now about what to do if the failure threshold is crossed. By default, an actuator port in Ptides will throw an exception if it receives an event with time stamp t and platform time has already exceeded t. Such an exception is an indication that assumptions about the ability of the platform to meet the deadline have been violated. A well-designed model will catch such exceptions, using for example error transitions in a modal model (see Section 8.2.3). How to handle such exceptions, of course, is application dependent. It might be necessary, for example, to switch to a safe but degraded mode of operation. Or it might be necessary to restart some portion of the system, or to switch to a backup system.

Multiple Ptides platforms in a model may communicate via a network. When such communication occurs, logical time stamps are conveyed along with the data. Unlike an actuator port, a network transmitter port always produces its output immediately when it becomes available, rather than waiting for platform time to match the time stamp. The logical time stamp of the event will be carried along with the event to the network receiver port, which will then produce on its output an event with that same time stamp.

Like an actuator port, a network transmitter port treats the time stamp as a deadline and will throw an exception if the platform time exceeds the time stamp value when the event arrives.†

Example 10.15: In the Ptides model of Figure 10.18, in PtidesPlatform1, assume that the sensor makes a measurement at platform time t_s, and that consequently the network transmitter port receives an event with time stamp $t_s + d_1$. Assume further that it receives this event at platform time t_s, because the execution time of actors is (by default) assumed to be zero. Hence, the Network actor in Figure 10.18 will received an event containing as its payload both a value (the value of the event delivered to the NetworkTransmitterPort) and a logical time stamp $t_s + d_1$.

The NetworkTransmitterPort will launch this payload into the network at platform time t_s. The network will incur some delay, simulated by the Network actor in the figure, and will arrive at PtidesPlatform2 at some time t_2, a local platform time at PtidesPlatform2. The NetworkReceiverPort on PtidesPlatform2 will produce an output event with (logical) time stamp $t_s + d_1$, extracted from the payload. In Figure 10.18, this event will pass through another Computation actor and another TimeDelay actor. Assuming the TimeDelay actor increments the time stamp by d_2, the *ActuatorPort* on PtidesPlatform2 will receive an event with time stamp $t_s + d_1 + d_2$. This deadline will be met if $t_s + d_1 + d_2 \geq t_2$.

If the *actuateAtEventTimestamp* parameter of the *ActuatorPort* is `true`, and all deadlines are met, then the overall latency from the sensor in platform 1 to the actuator in platform 2 is deterministic and independent of the actual network delay

†This deadline may be modified to be earlier or later by changing the *platformDelayBound* parameter of the network transmitter port, as explained below.

and actual computation times. This ability to have a fixed latency in a distributed system is central to the power of the Ptides model.

As with the actuator on platform 1, if the deadline is not met at the actuator on platform 2, the ActuatorPort will throw an exception. This exception is an indication that some timing assumption about the implementation has been violated; for example, an assumed bound on the network latency has not been actually met by the network. This should be handled by the model as an error condition, which could, for example, cause the model to switch into a safe but degraded mode of operation.

Although the end-to-end latency from the sensor on platform 1 to the actuator on platform 2 is deterministic, it is not exactly clear from this model what that latency is. Nominally, the latency is the logical time delay, $d_1 + d_2$. However, the time at which the actuation occurs, $t_s + d_1 + d_2$, is relative to the local platform clock at platform 2. This time, however, also depends on the clock on platform 1, since t_s is the time on platform 1 when the sensor measurement is taken. Hence, to be useful, a distributed Ptides system requires that clocks be synchronized (see Section 10.2). They need not be perfectly synchronized, but if the error between them is not bounded, then there is no bound on the end-to-end latency (in oracle time).

If these two platform clocks are perfectly synchronized, then the actual latency will be exactly $d_1 + d_2$, relative to these platform clocks. The latency in oracle time, of course, depends on the drift of these clocks relative to oracle time (see Figure 10.2). If these two clocks progress at exactly the rate of oracle time, then the actual latency will be exactly $d_1 + d_2$ in oracle time. Hence, with sufficiently good clocks and sufficiently good clock synchronization, Ptides gives an overall timing behavior that is precise and deterministic up to the precision of these clocks.

To help ensure that a realization meets the requirements of a specification, a network receiver port also imposes a constraint on timing. As mentioned above, the network transmitter port will throw an exception if it receives an event at a platform time greater than the time stamp of the event.[‡] So if a network receiver port receives a message, it knows that the message was transmitted at a platform time no later than the time stamp on the message it receives. The receiver has a parameter *networkDelayBound*, which is an upper bound on the network delay that it assumes. When the network receiver receives a

[‡]This deadline may be modified to be earlier or later by changing the *platformDelayBound* parameter of the network transmitter port, as explained below.

message, it checks that the platform time does not exceed the time stamp on the message plus the *networkDelayBound* plus a fudge factor to account for clock discrepancies and device delays, which also have assumed bounds specified by parameters, described below. If the platform time is too large, then the network receiver knows that one of these assumptions was violated (though it cannot know which one), and it throws an exception. Although this constraint is very subtle, the consequences on models are relatively easy to understand.

> **Example 10.16:** In the Ptides model of Figure 10.18, along the path from the sensor to the network receiver port, there cannot be a physical delay greater than the logical delay along the same path. The logical delay along this path is simply d_1, the parameter of the TimeDelay actor. The physical delay is the sum of the executions times of the actors along the path (which in simulation is assumed to be zero by default) and the network delay. Hence, if the network imposes a delay greater than d_1, the model in this figure will fail with an exception (by default, though other error handling strategies are also possible).

Notice that if we were to replace the Ptides directors in the platforms with DE directors, then the behavior would be significantly different. In this case, the latency from the sensor in platform 1 to the actuator in platform 2 would include the actual network delay. A key property of Ptides is that network delays and computation times are segregated from the logical timing of a model. The logical timing becomes a *specification* of timing behavior, whereas network delays and computation times are part of the *realization* of the system. Ptides models enable us to determine conditions under which realizations will meet the requirements of the specification. And the simulator enables evaluation of behavior under elaborate conditions that would be very difficult to validate analytically, for example taking into account the complicated dynamics of PTP clock synchronization protocols.

10.5.2 Ptides Components

Ptides ports. Ports in a Ptides platform represent devices that communicate with the environment or the network. Ptides ports can model device delays, although by default these delays are zero. Every Ptides port has a *deviceDelay* and a *deviceDelayBound* parameter. The *deviceDelay* d models delay of the device. For example, if a sensor makes a mea-

Sidebar: Safe-to-Process Analysis

The execution of actors inside a Ptides platform follows DE semantics. Actors must process events in time-stamp order (unless they are memoryless). In simulation, it is straightforward to ensure that events are processed in time-stamp order, but when Ptides models are deployed, things get more complicated. In particular, a deployed system cannot easily coordinate the scheduling of actor firings across platforms. Each platform must be able to make its own scheduling decisions.

Consider the platform model shown in Figure 10.19. This example has a sensor port and a network receiver port. Suppose that the sensor produces an event with time stamp t_s. If we assume that every sensor produces events in time-stamp order, then the scheduler can immediately fire Computation1. Suppose that firing produces another event with time stamp t_s, which then results in an event with time stamp $t_s + d_1$ available at the top input of Computation3. When can Computation3 be fired to react to that event? The scheduler has to be sure that no event with time stamp less than or equal to $t_s + d_1$ will later become available at the bottom input of Computation3.

A simple approach, developed by Chandy and Misra (1979) for distributed DE simulation, is to wait until there is an event available on the bottom input of Computation3 with time stamp greater than or equal to $t_s + d_1$. But this could result in quite a wait, particularly if a fault occurs and the source of events on this path fails.

An alternative approach due to Jefferson (1985) fires Computation3 speculatively, assuming no problematic event will later arrive, and if it does, reverses the computation by restoring the state of the actor. This approach is fundamentally limited by the inability to backtrack actuators.

The Ptides approach ensures that events are processed in order *as long as all deadlines are met.* In our example, an event at the top input of Computation3 with time stamp $t_s + d_1$ can be safely processed when the local platform time meets or exceeds $t_s + d_1$. This is because schedulability requires that an event with time stamp $t_s + d_1$ or earlier is required to arrive at the network receiver port at platform time $t_s + d_1$ or earlier.

Conversely, suppose an event with time stamp t_n is at the bottom input of Computation3. That event is **safe to process** when platform time meets $t_n - d_1 + s$, where s is a bound on the **sensor delay**, the time between a sensor event time stamp and the event becoming visible to the scheduler. See page 382 for citations that explain safe-to-process analysis in more detail.

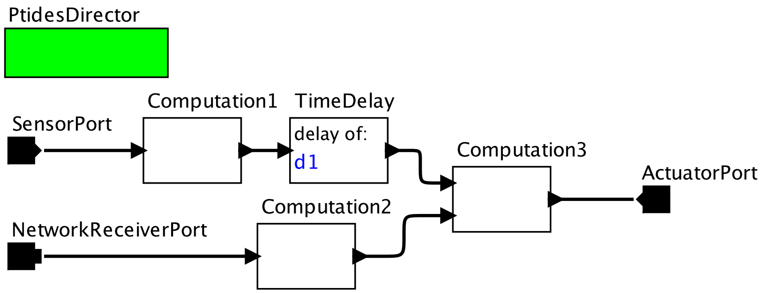
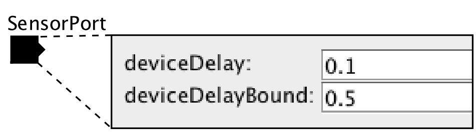


Figure 10.19: Simple Ptides example used to illustrate safe-to-process analysis.

surement at platform time t_s, it will produce an event with time stamp t_s. But that event will not appear until platform time $t_s + d$. For example, d might represent the amount of platform time that it takes for the sensor device to raise an interrupt request, and for the processor to respond to the interrupt request.

The *deviceDelayBound* d_B gives an upper bound on the *deviceDelay* d. The Ptides framework assumes that *deviceDelay* can vary during execution but will never exceed *deviceDelayBound*, which does not vary. This bound is used in safe to process analysis (see sidebar on page 390), which ensures that events are processed in time-stamp order.

Sensors. A **sensor port** is a particular kind of Ptides port that looks like this:

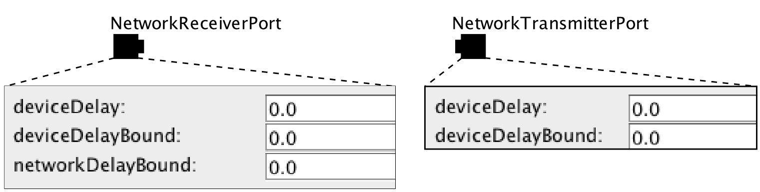


It receives inputs from the environment and creates new events with the time stamp equal to the current platform time (which is the current local time of the PtidesDirector) and posts this event on the event queue. An event that is received by a sensor at platform time t_s is produced with logical time stamp t_s at platform time $t_s + d$.

Actuators. An **actuator port** is a particular kind of Ptides port that looks like this:

By default, an actuator port produces events on the output of the platform when the time stamp of the event equals the current platform time. If you change *actuateAtEventtimestamp* to false, then the event may be produced earlier if it is available earlier. The *deviceDelayBound* parameter specifies a **setup time** for the device. Specifically, the deadline for delivery of an event with time stamp t to an actuator is platform time $t - d_B$, where d_B is the value of *deviceDelayBound*. An exception is thrown if this deadline is not met.

Network transmitters and receivers. **Network transmitter ports** and **network receiver ports** are also particular kinds of Ptides port that look like this:



The *NetworkTransmitterPort* takes an event from the inside of the Ptides platform and sends to the outside a record that encodes the time stamp of the event and its value (the **payload**). The *NetworkReceiverPort* extracts the time stamp and the payload and produces at the inside of the destination Ptides platform and event with the specified value and time stamp.

The parameter *networkDelayBound* (d_N) specifies the assumed maximum amount of time an incoming token spends in the network before it arrives at the receiver. It is used to determine whether an event can processed safely, or whether another event with a smaller time stamp may still be in the network. If the actual network delay exceeds this bound and the delay causes the message to be received too late, then the network receiver port will throw an exception.

10.6 Summary

Modeling of complex timed systems is not easy. We all harbor a naive notion of a uniform fabric of time, shared by all participants in the physical world. But such a notion is a fiction, and real systems are strongly affected by errors in time measurement and discrepancies between logical and physical notions of time. A major focus of recent work in the Ptolemy Project has been to provide a solid modeling foundation for the far-from-solid realities of time.

10.7 Acknowledgements

The authors would like to thank Yishai Feldman and Stavros Tripakis for very helpful suggestions for this chapter.

11

Ptera: An Event-Oriented Model of Computation

Thomas Huining Feng and Edward A. Lee

Contents

FSMs and DE models, covered in Chapters 6 and 7, focus on events and the causal relationships between those events. An event is atomic, conceptually occurring at an instant in time. An **event-oriented model** defines a collection of events in time. Specifically, it generates events, typically in chronological order, and defines how other events are triggered by those events. If there are externally provided events, the process also defines how those events may trigger additional events. In the DE domain, the timing of events is controlled by timed sources and delay actors (see sidebars on pages 241 and 243). Models in the FSM domain primarily react to externally provided events, but may also generate timed events internally using the *timeout* function in a guard (see Table 6.2 on page 196).

There are many ways other ways to specify event-oriented models (see sidebar on page 396). This chapter describes a novel one called **Ptera** (for Ptolemy event relationship actors), first given by Feng et al. (2010). Ptera is designed to interoperate well with other Ptolemy II models of computation, to provide model hierarchy, and to handle concurrency in a deterministic way.

11.1 Syntax and Semantics of Flat Models

A flat (i.e., non-hierarchical) Ptera model is a graph containing vertices connected with directed edges, such as shown in Figure 11.1, which contains two vertices and one edge. A vertex contains an **event**, and a directed edge represents the conditions under which one event will cause another event to occur. Vertices and edges can be assigned a range of attributes and parameters, as described later in this chapter.

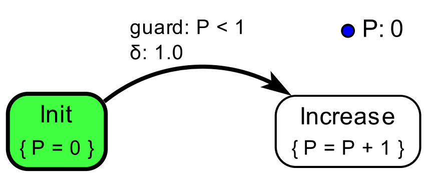


Figure 11.1: A simple Ptera model with two events. [online]

In a hierarchical Ptera model, vertices can also represent *submodel*s, which may be other Ptera models, FSMs, actor models using some other Ptolemy II director, or even custom Java code. The only requirement is that their behavior must be defined to conform with the actor abstract semantics, as explained below.

Sidebar: Notations, Languages for Event-Oriented Models

Many notations and languages have been developed for describing event-oriented models. A popular one today is the UML **activity diagram**, a derivative of the classical **flowcharts** that date back to the 1960s (see `http://en.wikipedia.org/wiki/Activity_diagram`). In an activity diagram, a block represents an **activity**, and an arrow from one activity to another designates the causality relationship between the two. A diamond-shaped activity tests for a condition, and causes the activity on one of its outgoing branches to occur. Activity diagrams include **split** and **join** activities, which spawn multiple concurrent activities, and wait for their completion. As is common with UML notations, the semantics of concurrency (and even of activities) is not clear. Activities may be interpreted as events, in which case they are atomic, or they may take time, in which case the triggering of an activity and its completion are events. The meaning of an activity diagram also becomes unclear when connections are made into and out of concurrent activities, and the model of time is not well defined. Nevertheless, activity diagrams are often easy to understand intuitively, and hence prove useful as a way to communicate event-oriented models.

Another relevant notation is the **business process model and notation** (**BPNL**), which, like UML, is now maintained and developed by the OMG. BPNL makes a distinction between events and activities, allowing both in a diagram. It also offers a form of hierarchy and concurrency, with rather complicated interaction and synchronization mechanisms.

A third related notation is **control flow graphs**, introduced by Allen (1970), which today are widely used in compiler optimizations, program analysis tools, and electronic design automation. In a CFG, the nodes in a graph represent **basic blocks** in a program, which are sequences of instructions with no branches or flow control structures. These basic blocks are treated as atomic, and hence can be considered events. Connections between basic blocks represent the possible flow control sequences that a program may follow.

Sidebar: Background of Ptera

Ptera is derived from **event graphs**, given by Schruben (1983). Blocks in an event graph (which he called vertices) contain events, which can include actions to be performed when that event is processed. Connections between events (called "directed edges") represent scheduling relations that can be guarded by Boolean and temporal expressions. Event graphs are timed, and time delays can be associated with scheduling relations. Each event graph has an event queue, although it is not explicitly shown in the visual representation. In multi-threaded execution, multiple event queues may be used, in principle. In each step of an execution, the execution engine removes the next event from the event queue and processes it. The event's associated actions are executed, and additional events specified by its scheduling relations are inserted into the event queue.

The original event graphs do not support hierarchy. Schruben (1995) gives two approaches for supporting hierarchy. One is to associate submodels with scheduling relations, in which the output of a submodel is a number used as the delay for the scheduling relation. Another approach is to associate submodels with events instead of scheduling relations (Som and Sargent, 1989). Processing such an event causes the unique start event in the submodel to be scheduled, which in turn may schedule further events in the submodel. When a predetermined end event is processed, the execution of the submodel terminates, and the event that the submodel is associated with is considered processed. Buss and Sanchez (2002) report a third attempt to support hierarchy, in which a listener pattern is introduced as an extra gluing mechanism for composing event graphs.

Ptera is based on event graphs, but extends them to support heterogeneous, hierarchical modeling. Composition of Ptera models forms a hierarchical model, which can be flattened to obtain an equivalent model without hierarchy. Ptera models conform with the actor abstract semantics, which permits them to contain or be contained by other types of models, thus enabling hierarchical heterogeneous designs. Ptera models can be freely composed with other models of computation in Ptolemy II.

Ptera models include an externally visible interface with parameters, input and output ports. Changes to parameters and the arrival of data at input ports can potentially trigger events within the Ptera model, in which case it becomes an actor. In addition, event actions can be customized by the designer with programs in an imperative language (such as Java or C) conforming to a protocol.

11.1.1 Introductory Examples

Example 11.1: An example Ptera model is shown in Figure 11.1. This model includes two vertices (events), *Init* and *Increase*, and one **variable**, P, with an initial value of 0. When the model is executed, this variable may be updated with new values. *Init* is an **initial event** (its *initial* parameter is set to true), as indicated by a filled rounded rectangle with a thick border.

At the start of execution, all initial events are scheduled to occur at model time 0. (As discussed later, even when events occur at the same time conceptually, there is still a well-defined order of execution.) The model's *event queue* holds a list of scheduled **event instances**. An event instance is removed from the event queue and processed when the model time reaches the time at which the event is scheduled to occur (i.e., at the time stamp of that event).

Ptera events may be associated with **actions**, which are shown inside brackets in the vertex.

Example 11.2: In Figure 11.1, for example, *Init* specifies the action "P = 0", which sets P to 0 when *Init* is processed. The edge (connection) from the *Init* event to the *Increase* event is called a **scheduling relation**. It is guarded by the Boolean expression "$P < 1$" (meaning that the transition will only be taken when this condition is met) and has a **delay** of 1.0 units of time (represented by the δ symbol). After the *Init* event is processed, if P's value is less than 1 (which is true in this case, since P is initially set to 0), then *Increase* will be executed at time 1.0. When *Increase* is processed at time 1.0, its action "P = P + 1" is executed and P's value is increased to 1.

After processing the *Increase* event, the event queue is empty. Since no more events are scheduled, the execution terminates.

In this simple example, there is at most one event in the event queue (either *Init* or *Increase*) at any time. In general, however, an unbounded number of events can be scheduled in the event queue.

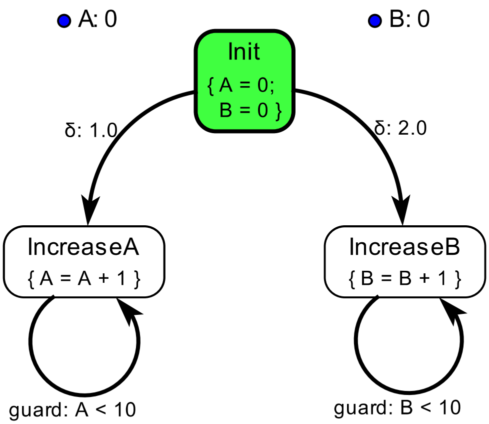


Figure 11.2: A model with multiple events in the event queue. [online]

Example 11.3: Figure 11.2 shows a slightly more complex Ptera model that requires an event queue of size greater than 1. In this model, the *Init* event schedules *IncreaseA* to occur after a 1.0-unit time delay, and *IncreaseB* to occur after a 2.0-unit delay. The guards of the two scheduling relations from *Init* have the default value "true," and are thus not shown in the visual representation. When *IncreaseA* is processed, it increases variable A by 1 and reschedules itself, creating another event instance on the event queue, looping until A's value reaches 10. (The model-time delay δ on the scheduling relation from A to itself is also hidden, because it takes the default value "0.0," which means the event is scheduled at the current model time, but the next microstep.) Similarly, *IncreaseB* repeatedly increases variable B at the current model time until B's value reaches 10.

11.1.2 Event Arguments

Like a C function, an event may include a list of formal *arguments*, where each argument is assigned a name and type.

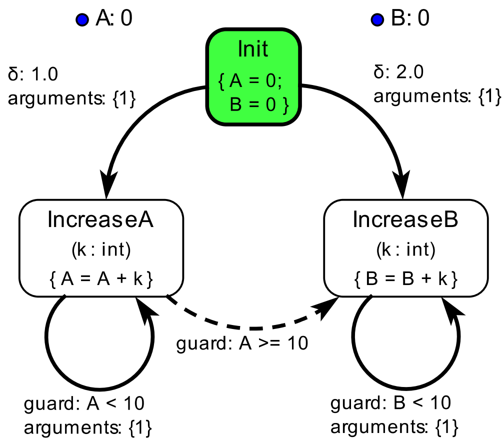


Figure 11.3: A model with arguments for the events and a canceling relation. [online]

Example 11.4: Figure 11.3 modifies Figure 11.2 by adding arguments k of type int to events *IncreaseA* and *IncreaseB*. These arguments are assigned values by the incoming relations and specify the increments to variables A and B. (The dashed edge in the figure is a canceling relation, which will be discussed in the next subsection.)

Each scheduling relation pointing to an event with arguments must specify a list of expressions in its *arguments* attribute. Those expressions are used to specify the actual values of the arguments when the event instance is processed. In this example, all scheduling relations pointing to *IncreaseA* and *IncreaseB* specify "$\{1\}$" in their argument attributes, meaning that k should take value 1 when those events are processed. Argument values can be used by event actions, guards, and delays.

11.1.3 Canceling Relations

A **canceling relation** is represented as a dashed line (edge) between events, and can be guarded by a Boolean expression. It cannot have any delays or arguments. When an event with an outgoing canceling relation is processed, if the guard is true and the target event has been scheduled in the event queue, the target event instance is removed from the event queue without being processed. In other words, a canceling relation cancels a previously scheduled event. If the target event is scheduled multiple times, i.e. multiple event instances are in the event queue, then the canceling relation causes only the first instance to be removed. If the target event is not scheduled, the canceling relation has no effect.

Example 11.5: Figure 11.3 provides an example of a canceling relation. Processing the last *IncreaseA* event (at time 1.0) causes *IncreaseB* (scheduled to occur at time 2.0 by the *Init* event) to be cancelled. As a result, variable B is never increased.

It should be noted that canceling relations do not increase expressiveness. In fact, a model with canceling relations can always be converted into a model without canceling relations, as is shown by Ingalls et al. (1996). Nonetheless, they can yield more compact and understandable models.

11.1.4 Simultaneous Events

Simultaneous events are defined as multiple event instances in an event queue that are scheduled to occur at the same model time.

Example 11.6: For example, in Figure 11.3, if both δs are set to 1.0, the model is as shown in Figure 11.4. Instances of *IncreaseA* and *IncreaseB* scheduled by *Init* become simultaneous events. Moreover, although multiple instances of *IncreaseA* occur at the same model time, they occur at different microsteps in superdense time, and they do not coexist in the event queue, so instances of *IncreaseA* are not simultaneous.

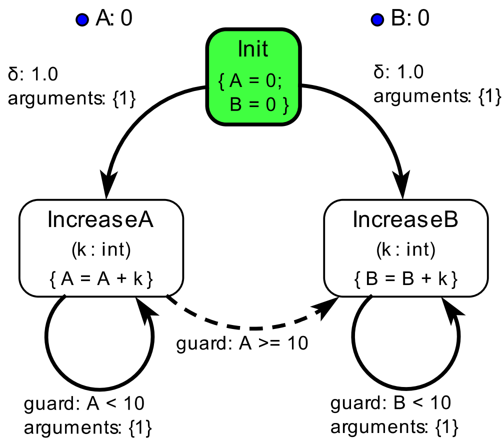


Figure 11.4: A model with simultaneous events. [online]

In general, it is a model checking (Clarke et al., 2000) problem to detect simultaneous events.

11.1.5 Potential Nondeterminism

When there are simultaneous events, there is potential nondeterminism introduced by the ambiguous order of event processing.

Example 11.7: For example, in Figure 11.3, what is the final value of the variables A and B? Suppose that all instances of *IncreaseA* are processed before any instance of *IncreaseB*. In that case, the final value of B will be 0. If instead, all instances of *IncreaseB* are processed before any instance of *IncreaseA*, then the final value of B will be 10.

Table 11.1 shows four possible execution traces that seem consistent with the model definition. The columns are arranged from left to right in the order of event processing. These traces include the case where *IncreaseA* always occurs before *IncreaseB*, where *IncreaseB* always occurs before *IncreaseA*, and where *IncreaseA*

1) *IncreaseA* is always scheduled before *IncreaseB*:

Time	0.0	1.0	1.0	...	1.0
Event	*Init*	*IncreaseA*	*IncreaseA*	...	*IncreaseA*
A	0	1	2	...	10
B	0	0	0	...	0

2) *IncreaseB* is always scheduled before *IncreaseA*:

Time	0.0	1.0	1.0	...	1.0	1.0
Event	*Init*	*IncreaseB*	*IncreaseB*	...	*IncreaseB*	*IncreaseA*
A	0	0	0	...	0	1
B	0	1	2	...	10	10

Time	1.0	...	1.0
Event	*IncreaseA*	...	*IncreaseA*
A	2	...	10
B	10	...	10

3) *IncreaseA* and *IncreaseB* are alternating, starting with *IncreaseA*:

Time	0.0	1.0	1.0	1.0	1.0	...
Event	*Init*	*IncreaseA*	*IncreaseB*	*IncreaseA*	*IncreaseB*	...
A	0	1	1	2	2	...
B	0	0	1	1	2	...

Time	1.0	1.0	1.0
Event	*IncreaseA*	*IncreaseB*	*IncreaseA*
A	9	9	10
B	8	9	9

4) *IncreaseA* and *IncreaseB* are alternating, starting with *IncreaseB*:

Time	0.0	1.0	1.0	1.0	1.0	...
Event	*Init*	*IncreaseB*	*IncreaseA*	*IncreaseB*	*IncreaseA*	...
A	0	0	1	1	2	...
B	0	1	1	2	2	...

Time	1.0	1.0	1.0	1.0
Event	*IncreaseB*	*IncreaseA*	*IncreaseB*	*IncreaseA*
A	8	9	9	10
B	9	9	10	10

Table 11.1: Four possible execution traces for the model in Figure 11.4.

and *IncreaseB* are alternating in two different ways. There are many other possible execution traces.

The traces end with different final values of A and B. The last instance of *IncreaseA*, which increases A to 10, always cancels the next *IncreaseB* in the event queue, if any. There are 10 instances of *IncreaseB* in total, and without a well-defined order, the one that is cancelled can be any one of them.

To avoid these nondeterministic execution results, we use the strategies discussed in the next sections.

11.1.6 LIFO and FIFO Policies

Ptera models can specify a **LIFO** (last in, first out) or **FIFO** (first in, first out) policy to control how event instances are accessed in the event queue and to help ensure deterministic outcomes. With LIFO (the default), the event scheduled later is processed sooner. The opposite occurs with FIFO. The choice between LIFO and FIFO is specified by a parameter *LIFO* in the Ptera model that defaults to value true.

If we use a LIFO policy to execute the model in Figure 11.4, then execution traces 3 and 4 in Table 11.1 are eliminated as possible outcomes. This leaves two possible execution traces, depending on whether *IncreaseA* or *IncreaseB* is processed first (that choice will depend on scheduling rules discussed later).

Example 11.8: Suppose *IncreaseA* is processed first. According to the LIFO policy, the second instance of *IncreaseA* scheduled by the first one should be processed before *IncreaseB*, which is scheduled by *Init*. The second instance again schedules the next one. In this way, processing of instances of *IncreaseA* continues until A's value reaches 10, when *IncreaseB* is cancelled. That leads to execution trace 1.

If instead *IncreaseB* is processed first, all 10 instances of *IncreaseB* are processed before *IncreaseA*. That yields execution trace 2.

With a FIFO policy, however, instances of *IncreaseA* and *IncreaseB* are interleaved, resulting in execution traces 3 and 4 in the table.

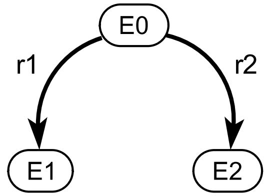


Figure 11.5: A scenario where event E0 schedules *E1* and *E2* after the same delay.

In practice, LIFO is more commonly used because it executes a chain of events; one event schedules the next without delay. This approach is atomic in the sense that no event that is in the chain interferes with the processing of events in the chain. This is convenient for specifying workflows where some tasks need to be finished sequentially without intervention.

11.1.7 Priorities

For events that are scheduled by the same event with the same delay δ, **priority** numbers can be assigned to the scheduling relations to force an ordering of event instances. A priority is an integer (which can be negative) that defaults to 0.

Example 11.9: Simultaneous instances of *E1* and *E2* in Figure 11.5 are scheduled by the scheduling relations r1 and r2 (with delay 0.0, since δ is not shown). If r1 has a higher priority (i.e., a smaller priority number) than r2, then *E1* is processed before *E2*, and vice versa.

Example 11.10: In Figure 11.4, if the priority of the scheduling relation from *Init* to *IncreaseA* is -1, and the priority of that from *Init* to *IncreaseB* is 0, then the first instance of *IncreaseA* is processed before *IncreaseB*. Execution traces 2 and 4 in Table 11.1 would not be possible. On the other hand, if the priority of the scheduling relation from *Init* to *IncreaseA* is 1, then the first instance of *IncreaseB* is processed earlier, making execution traces 1 and 3 impossible.

11.1.8 Names of Events and Scheduling Relations

Every event and every scheduling relation in a Ptolemy II Ptera model has a name. In Figure 11.4, for example, *Init*, *IncreaseA*, and *IncreaseB* are the names of the events. These names may be assigned by the builder of the model (see Figure 2.15). Vergil will assign a default name, and at each level of a hierarchical model, the names are unique. Ptera uses these names to assign an execution order for simultaneous events when the priorities of their scheduling relations are the same.

In Figure 11.5, if r1 and r2 have the same delay δ and the same priority, then we use names to determine the order of event processing.* The order of *E1* and *E2* is determined by first comparing the names of the events. In a flat model, these names are guaranteed to be different, so they have a well-defined alphabetical order. The earlier one in this order will be scheduled first.

In a hierarchical model (discussed below), it is possible for simultaneous events to have the same name. In that case, the names of the scheduling relations are used instead. These are not usually shown in the visual representation, but can be determined by hovering over a scheduling relation with the cursor.

11.1.9 Designs with Atomicity

In some applications, designers need to ensure atomic execution of a sequence of events in the presence of other simultaneous events. That is, the entire sequence should be processed before any other simultaneous event is processed. There are two design patterns shown in Figure 11.6 that can be used to ensure atomicity without requiring designers to explicitly control critical sections (as would be the case for imperative programming languages).

*It would be valid in a variant of Ptera to either choose nondeterministically or process the two events concurrently, but this could lead to nondeterminism, so in our current implementation, we have chosen to define the order.

The design pattern in Figure 11.6(a) is used to sequentially and atomically perform a number of tasks, assuming the LIFO policy is chosen. Even if other events exist in the

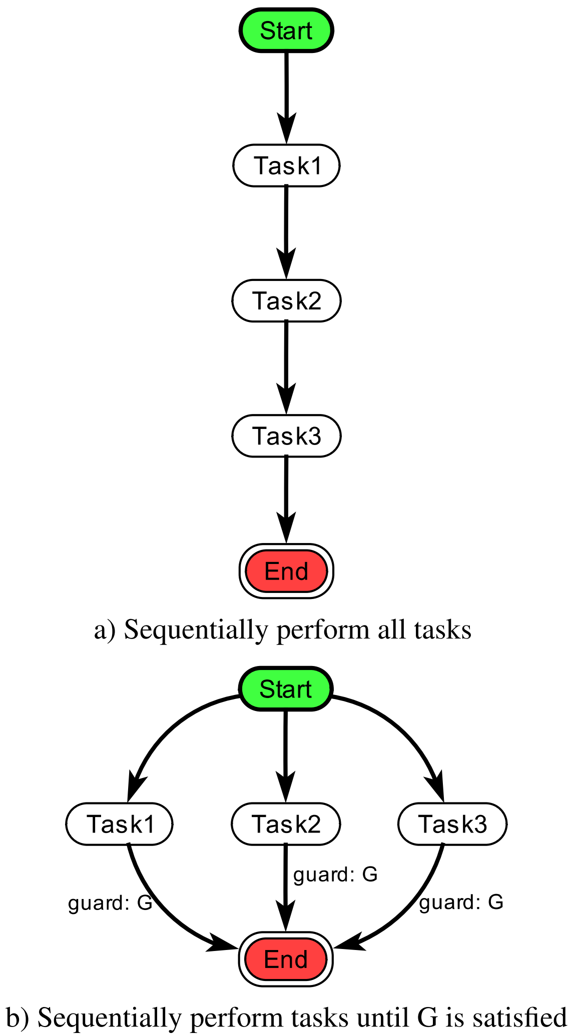


Figure 11.6: Two design patterns for controlling tasks.

model (which are not shown in the figure), those events cannot interleave with the tasks. As a result, intermediate state between tasks is not affected by other events.

The **final event**, End, is a special class of event that removes all events in the queue. Final events are used to force termination even when there are events remaining in the event queue. They are shown as filled vertices with double-line borders.

The design pattern in Figure 11.6(b) is used to perform tasks until the guard G is satisfied. This pattern again assumes a LIFO policy. After the Start event is processed, all tasks are scheduled. In this case, the first one to be processed is Task1, because of the alphabetical ordering. After Task1, if G is true, End is processed next, which terminates the execution. If G is not true, then Task2 is processed. The processing of tasks continues until either G becomes true at some point, or all tasks are processed but G remains false.

11.1.10 Application-Oriented Examples

In this section, we describe two simple Ptera models that have properties that are similar to many systems of interest. We begin with a simple multiple-server, single-queue system: a car wash.

Example 11.11: In this example, multiple car wash machines share a single queue. When a car arrives, it is placed at the end of the queue to wait for service. The machines serve cars in the queue one at a time in a first-come-first-served manner. The car arrival intervals and service times are generated by stochastic processes assigned to the edges.

The model shown in Figure 11.7 is designed to analyze the number of available servers and the number of cars waiting over an elapsed period of time. The *Servers* variable is initialized to 3, which is the total number of servers. The *Queue* variable starts with 0 to indicate that there are no cars waiting in the queue at the beginning. *Run* is an initial event. It schedules the *Terminate* final event to occur after the amount of time defined by a third variable, *SimulationTime*.

The *Run* event also schedules the first instance of the *Enter* event, causing the first car arrival to occur after delay “3.0 + 5.0 * random(),” where random() is a function that returns a random number in $[0, 1)$ with a uniform distribution. When *Enter* occurs, its action increases the queue size in the *Queue* variable by 1. The *Enter*

Sidebar: Model Execution Algorithm

We operationally define the semantics of a flat model with an execution algorithm. In the algorithm, symbol Q refers to the event queue. The algorithm terminates when Q becomes empty.

1. Initialize Q to be empty
2. For each initial event e in the $\leq_e$ order
    1. Create an instance i_e
    2. Set the time stamp of i_e to be 0
    3. Append i_e to Q
3. While Q is not empty
    - (a) Remove the first i_e from Q, which is an instance of some event e
    - (b) Execute the actions of e
    - (c) Terminate if e is a final event
    - (d) For each canceling relation c from e

      From Q, remove the first instance of the event that c points to, if any
    - (e) Let R be the list of scheduling relations from e
    - (f) Sort R by delays, priorities, target event IDs, and IDs of the scheduling relations in the order of significance
    - (g) Create an empty queue Q'
    - (h) For each scheduling relation r in R whose guard is true
        1. Evaluate parameters for the event e' that r points to
        2. Create an instance $i_{e'}$ of e' and associate it with the parameters
        3. Set the time stamp of $i_{e'}$ to be greater than the current model time by r's delay
        4. Append $i_{e'}$ to Q'
    - (i) Create Q'' by merging Q' with Q and preserving the order of events originally in Q' and Q. For any $i' \in Q'$ and $i \in Q$, i' appears before i in Q'' if and only if the LIFO policy is used and the time stamp of i' is less than or equal to that of i, or the FIFO policy is used and the time stamp of i' is strictly less than that of i.
    - (j) Let Q be Q''

event schedules itself to occur again. It also schedules the *Start* event if there are any available servers. The LIFO policy guarantees that both *Enter* and *Start* will be processed atomically, so it is not possible for the number of servers available to be changed by any other event in the queue after that value is tested by the guard of the scheduling relation from *Enter* to *Start*. In other words, once a car has entered the queue and has started being washed, its washing machine cannot be taken by another car.

The *Start* event simulates car washing by decreasing the number of available servers and the number of cars in the queue. The service time is "5.0 + 20.0 * random()." After that amount of time, the *Leave* event occurs, which represents the end of service for that car. Whenever a car leaves, the number of available servers must be greater than 0 (since at least one machine will become available at that point), so the *Leave* event immediately schedules *Start* if there is at least one car in the queue. Due to atomicity provided by the LIFO policy, the model will test the queue

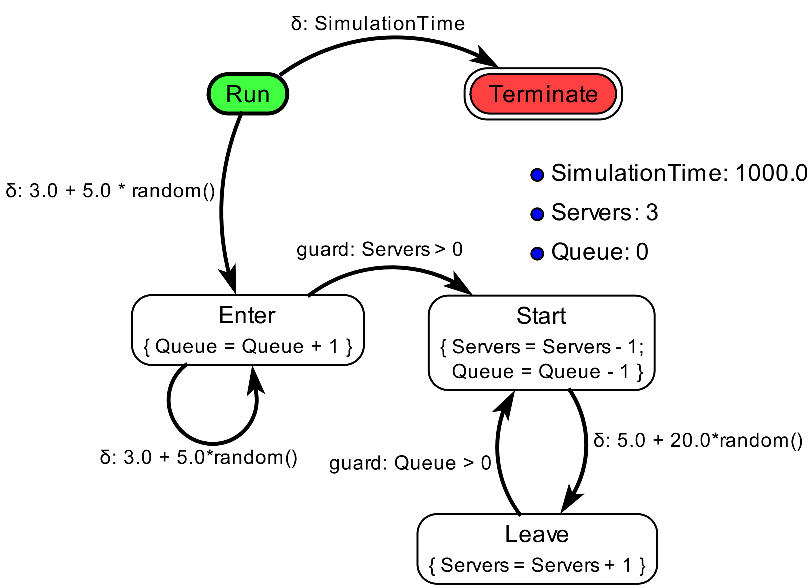


Figure 11.7: A model that simulates a car wash system. [online]

size and decrement its value during the subsequent *Start* event without allowing interruption by any other event in the queue.

The *Terminate* event, which halts execution, is prescheduled at the beginning of execution. Without this event, the model's execution would not terminate because the event queue would never be empty.

11.2 Hierarchical Models

Hierarchy can mitigate model complexity and improve reusability. Hierarchical multi-modeling enables combining multiple models of computation, each at its own level of the hierarchy. Here, we show how to construct hierarchical Ptera models. In the next section, we will show how to hierarchically combine Ptera models with other models of computation, such as those described in preceding chapters.

Example 11.12: Consider a car wash with two stations, one of which has one server, and one of which has three serves, where each station has its own queue. Figure 11.8 shows a hierarchical modification of Figure 11.7 with two such stations. Its top level simulates an execution environment, which has a *Run* event as the only initial event, a *Terminate* event as a final event, and a *Simulate* event associated with a submodel. The submodel simulates the car wash system with the given number of servers.

The two scheduling relations pointing to the *Simulate* event cause the submodel to trigger two instances of the *Init* event in the submodel's local event queue. These represent the start of two concurrent simulations, one with three servers (as indicated by the second argument on the left scheduling relation) and the other with one server. The priorities of the initializing scheduling relations are not explicitly specified. Because the two simulations are independent, the order in which they start has no observable effect. In fact, the two simulations may even occur concurrently.

Parameter i (the first argument on the two scheduling relations into *Init*) distinguishes the two simulations. Compared to the model in Figure 11.7, the *Servers* variable in the submodel has been extended into an array with two elements. *Servers*(0) refers to the number of servers in simulation $i = 0$, while *Servers*(1) is used in simulation $i = 1$. The *Queue* variable is extended in the same way. Each event in the submodel also takes a parameter i (which specifies the simulation number) and sends it to the next events that it schedules. This ensures that the events and variables in one simulation are not affected by those in the other simulation, even though they share the same model structure.

It is conceptually possible to execute multiple instances of a submodel by initializing it multiple times. However, the event queue and variables would not be copied. Therefore,

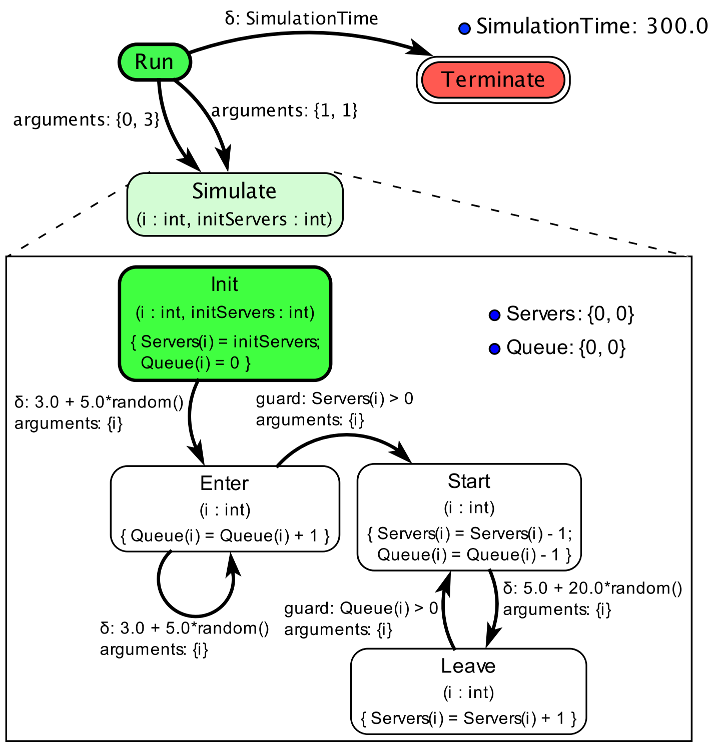


Figure 11.8: A hierarchical model that simulates a car wash system with two settings. [online]

the variables must be defined as arrays and an extra index parameter (i in this case) must be provided to every event.

Actor-oriented classes (see Section 2.6) provide an alternative approach to creating multiple executable instances of a submodel. The submodel can be defined as a class, with multiple instances executing in parallel.

11.3 Heterogeneous Composition

Ptera models can be composed with models built using other models of computation. Examples of such compositions are described in this section.

11.3.1 Composition with DE

Like Ptera models, discrete event (DE) models (discussed in Chapter 7) are based on events. But the notation is DE models is quite different. In DE, the components in the model, actors, consume input events and produce output events. In Ptera, an entire Ptera model may react to input events by producing output events, so Ptera submodels make natural actors in DE models. In fact, combinations of DE and Ptera can give nicely architected models with good separation of concerns.

Example 11.13: In Figure 11.8, the modeling of arrivals of cars is intertwined with the model of the servicing of cars. It is not easy, looking at the model, to separate these two. What if we wanted to, say, change the model so that cars arriving according to a Poisson process? Figure 11.9 shows a model that uses a DE director at its top level and separates the model of car arrivals from the servicing of the cars. This model has identical behavior to that in Figure 11.8, but it would be easy to replace the CarGenerator with a PoissonClock actor.

In Figure 11.9, in the CarGenerator, the *Init* event schedules the first *Arrive* event after a random delay. Each *Arrive* event schedules the next one. Whenever it is processed, the *Arrive* event generates a car arrival signal and sends it via the output port using the assignment “output = 1,” where “output” is the port name. In this case, the value 1 assigned to the output is unimportant, since only the timing of the output event is of interest.

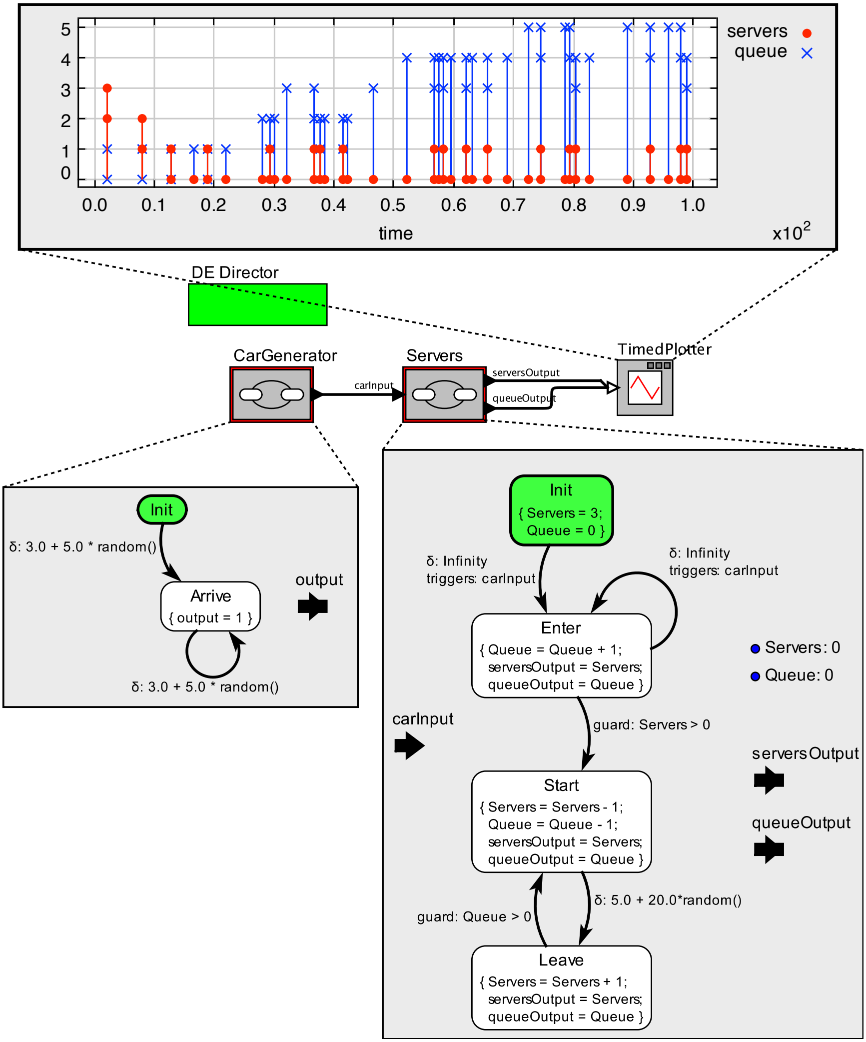


Figure 11.9: A car wash model using DE and Ptera in a hierarchical composition. [online]

Figure 11.9 also shows the internal design of Servers. It is similar to the previous car wash models, except that there is an extra *carInput* port to receive DE events representing car arrival signals from the outside and the *Enter* event is scheduled to handle inputs via that port. No assumption is made in the Servers component about the source of the car arrivals. At the top level, the connection from CarGenerator's output port to Servers' input port makes explicit the producer-consumer relationship, and leads to a more modular and reusable design.

The TimedPlotter shows the number of servers available and the number of waiting cars waiting in the queue over time. In the particular trace shown in the figure, the queue builds up to five cars over time.

A Ptera model within a DE model will execute either when an input event arrives from the DE model or when a timeout δ expires on a scheduling relation.

Example 11.14: In the Servers model in Figure 11.9, the relation from *Init* to *Enter* is labeled with δ: `Infinity`, which means the timeout will never expire. It is also labeled with `triggers: carInput`, which means that the *Enter* event will be scheduled to occur when an event arrives from the DE model on the *carInput* port.

A scheduling relation may be tagged with a *triggers* attribute that specifies port names separated by commas. This can be used to schedule an event in a Ptera submodel to react to external inputs. The attribute is used in conjunction with the delay δ to determine when the event is processed. Suppose that in a model the *triggers* parameter is "$p_1, p_2, \cdots, p_n$." The event is processed when the model time is δ-greater than the time at which the scheduling relation is evaluated *or* one or more DE events are received at *any* of $p_1, p_2, \cdots, p_n$. To schedule an event that indefinitely waits for input, `Infinity` may be used as the value of δ.

To test whether a port actually has an input, a special Boolean variable whose name is the port name followed by string "`_isPresent'`' can be accessed, similarly to FSMs. To refer to the input value available at a port, the port name may be used in an expression.

Example 11.15: The *Enter* event in Figure 11.9 is scheduled to indefinitely wait for DE events at the *carInput* port. When an event is received, the *Enter* event is processed ahead of its scheduled time and its action increases the queue size by 1. In that particular case, the value of the input is ignored.

To send DE events via output ports, assignments can be written in the action of an event with port names on the left hand side and expressions that specify the values on the right hand side. The time stamps of the outputs are always equal to the model time at which the event is processed.

11.3.2 Composition with FSMs

Ptera models can also be composed with untimed models such as FSMs. When a Ptera model contains an FSM submodel associated with an event, it can fire the FSM when that event is processed and when inputs are received at its input ports. FMSs are described in Chapter 6.

Example 11.16: To demonstrate composition of Ptera and FSM, consider the case where drivers may avoid entering a queue if there are too many waiting cars. This can lead to a lower arrival rate (or equivalently, longer average interarrival times). Conversely, if there are relatively few cars in the queue, the driver would always enter the queue, resulting in a higher arrival rate.

The model is modified for this scenario and shown in Figure 11.10. At the top level, the *queueOutput* port of Servers (whose internal design is the same as Figure 11.9) is fed back to the *queueInput* port of CarGenerator. The FSM submodel in Figure 11.10 refines the *Update* event in CarGenerator. It inherits the ports from its container, allowing the guards of its transitions to test the inputs received at the *queueInput* port. In general, actions in an FSM submodel can also produce data via the output ports.

At the time when the *Update* event of CarGenerator is processed, the FSM submodel is set to its initial state. When fired the first time, the FSM moves into the

Fast state and sets the minimum interarrival time to be 1.0. Subsequently, the interarrival time is generated using the expression "1.0 + 5.0 * random()." Notice that the *min* variable is defined in CarGenerator, and the scoping rules enable the contained FSM to read from and write to that variable.

When a Ptera model receives input at a port, all the initialized submodels are fired, regardless of the models of computation those submodels use.

The converse composition, in which Ptera submodels are refinements of states in an FSM, is also interesting. By changing states, the submodels may be disabled and enabled, and execution can switch between modes. That style of composition is provided by modal models, described in Chapter 8.

11.4 Summary

Ptera provides an alternative to FSMs and DE models, offering a complementary approach to modeling event-based systems. Ptera models are stylistically different from either. Components in the model are events, vs. states in FSMs and actors in DE models. The scheduling relations that connect events represent causality, where one event causes another under specified conditions (guard expressions, timeouts, and input events).

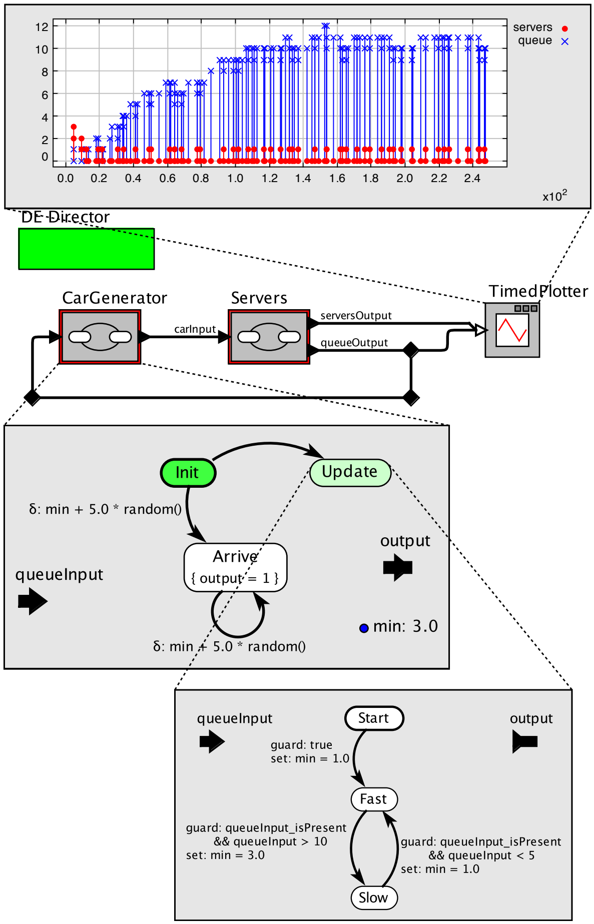


Figure 11.10: A car wash model using DE, Ptera and FSM in a hierarchical composition. [online]

Part III

Modeling Infrastructure

This part of this book focuses on the modeling infrastructure provided by Ptolemy II. Chapter 12 provides an overview of the software architecture with the goal of providing the reader with a good starting point for creating extensions of Ptolemy II. Chapter 13 describes the expression language used to set parameter values and perform computation in the Expression actor. Chapter 17 gives an overview of signal plotting capabilities provided in the actor library. Chapter 14 describes the Ptolemy II type system. Chapter 15 describes the ontology system. and Chapter 16 describes interfaces to the web, including mechanisms for exporting Ptolemy models to websites and mechanisms for creating custom web servers.

12

Software Architecture

Christopher Brooks, Joseph Buck, Elaine Cheong, John S. Davis II, Patricia Derler, Thomas Huining Feng, Geroncio Galicia, Mudit Goel, Soonhoi Ha, Edward A. Lee, Jie Liu, Xiaojun Liu, David Messerschmitt, Lukito Muliadi, Stephen Neuendorffer, John Reekie, Bert Rodiers, Neil Smyth, Yuhong Xiong, Haiyang Zheng

Contents

This chapter provides an overview of the software architecture of Ptolemy II to enable the reader to create custom software extensions, such as new directors or custom actors. Additional detail can be found in Brooks et al. (2004) and in the Ptolemy source code, which is well documented and designed for easy readability. This chapter assumes some familiarity with Java, the language in which most of Ptolemy II is written, and with UML class diagrams, which are used to depict key properties of the architecture.

12.1 Package Structure

Ptolemy II is a collection of Java classes organized into multiple packages. The package structure, shown in Figure 12.1, is carefully designed to ensure separability of the pieces.

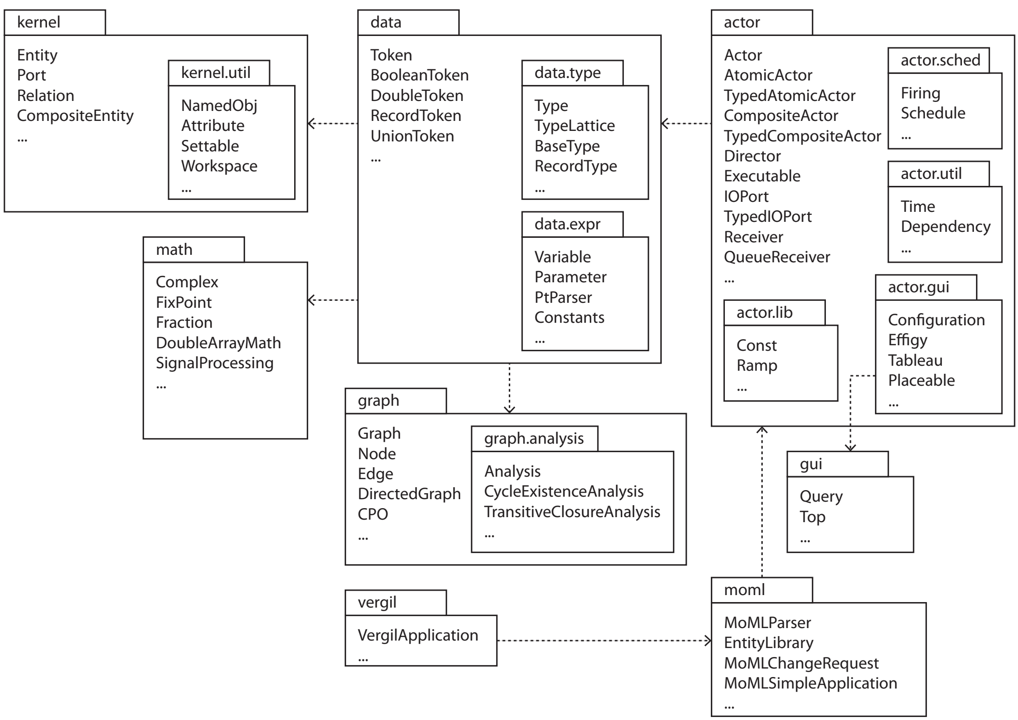


Figure 12.1: Key packages and their classes in Ptolemy II.

The kernel package. The kernel package and its subpackages are the heart of Ptolemy II. They contain the class definitions that serve as base classes for every part of a Ptolemy model. The kernel package itself is relatively small, and its design is described in Section 12.2. This package defines the structure of models; in particular, it specifies the hierarchy relationships between, for example, components and domains, and how the components of a model are interconnected.

The data package. The data package defines the classes that carry data from one component in a model to another. The Token class is of particular importance, because it is the base class for all units of data exchanged between components. The data.expr package defines the expression language, described in detail in Chapter 13. The expression language is used to assign values to parameters in a model and to establish interdependencies among parameters. The data.type package defines the type system (described in Chapter 14).

The math package. The math package contains mathematical functions and methods for operating on matrices and vectors. It also includes a complex number class, a class supporting fractions, and a set of classes supporting fixed-point numbers.

The graph package. The graph package and its subpackage, graph.analysis, provide algorithms for manipulating and analyzing mathematical graphs. This package supplies some of the core algorithms that are used in scheduling, in the type system, and in other model analysis tools.

The actor package. The actor package, described in more detail in Section 12.3, contains base classes for actors and I/O ports, where actors are defined as executable entities that receive and send data through I/O ports. The package also includes the base class Director that is customized for each domain to control model execution. The actor package contains several subpackages, including the following:

- The actor.lib package contains a large library of actors.
- The actor.sched package contains classes for representing and constructing schedules for executing actors.
- The actor.util package contains the core Time class, which implements model time, as described in Section 1.7.1. It also contains classes for keeping track of dependencies between output ports and input ports.
- The actor.gui package contains core classes for managing the user interface, including the Configuration class, which supports construction of customized, independently branded subsets of Ptolemy II. The Effigy and Tableau classes provide support for

opening and viewing models and submodels. The Placeable interface and associated classes provide support for actors with their own user interfaces.

The gui package. The gui package provides user interface components for interactively editing parameters of model components and for managing windows.

The moml package. The moml package provides a parser for **MoML** files (the **modeling markup language**, described in Lee and Neuendorffer (2000), which is the XML schema used to store Ptolemy II models.

The vergil package. The vergil package, which is quite large, provides the implementation of Vergil, the graphical user interface for Ptolemy II. Vergil is described further in Chapter 2.

There are many other packages and classes, but those discussed here provide a good overview of the overall system architecture. In the next section, we will explain how the kernel package defines the structure of models.

12.2 The Structure of Models

Computer scientists make a distinction between the syntax and the semantics of programming languages. The programming language syntax specifies the rules for constructing valid programs, and the semantics refers to the program's meaning. The same distinction is pertinent to models. The syntax of a model is how it is represented, while the semantics is what the model means.

Computer scientists further make a distinction between abstract syntax and concrete syntax. The **abstract syntax** of a model is the structure of its representation. For example, a model may be given as a **graph**, which is a collection of nodes and edges where the edges connect the nodes. Or it may be given as a **tree**, which is a type of graph that can be used to define a hierarchy. The abstract syntax of Ptolemy II models is (loosely) a tree (which represents the hierarchy of models) overlaid with a graph at each level of the hierarchy (for specifying the connections between components). The structure of a tree ensures that every node has exactly one **container**.

A **concrete syntax**, in contrast, is a specific notation for representing an abstract syntax. The block diagrams of Vergil are a concrete syntax. The MoML XML schema is a textual syntax (a syntax built from strings of characters) for the same models. The set of all structures that a concrete syntax can represent is its abstract syntax. Whereas an abstract syntax constrains the structure of a model, a concrete syntax provides a description of the model in text or pictures.

Both abstract and concrete syntaxes can be formally defined. A textual concrete syntax, for example, might be given in **Backus-Naur form** (**BNF**), familiar to computer scientists. In fact, BNF is a concrete syntax for describing concrete syntaxes. Compiler toolkits, such as the classic Yacc parser generator, take BNF as an input to automatically create a **parser**, which converts a textual concrete syntax into data structures in the memory of a computer that represent the abstract syntax.

Abstract syntax is more fundamental than concrete syntax. Given two concrete syntaxes for the same abstract syntax, translation from one concrete syntax to the other is always possible. Hence, for example, every Vergil block diagram can be represented in MoML and vice versa.

In general, a **meta model** is a model of a modeling language or notation. In the world of modeling, engineers use meta models to precisely define abstract syntaxes. A meta model can for example be given as a **UML class diagrams**, a notation for object-oriented designs (UML stands for the unified modeling language). A meta model in UML for the Ptolemy II abstract syntax is shown in Figure 12.2. The figure shows the relationships between object-oriented classes that are instantiated by the MoML parser to create a data structure representing a Ptolemy model. Instances of these classes comprise a **Ptolemy II model**.

Every component in a Ptolemy II model is an instance of the NamedObj class, which has a name and a container (the container encloses part of the hierarchical model; it is null for the **top-level** object). There are four subclasses of NamedObj. These are called Attribute, Entity, Port, and Relation. Instances of these subclasses are shown in a Ptolemy model in Figure 12.3.

A model consists of a top-level entity that contains other entities. The entities have ports through which they interact. Their interactions are mediated by relations, which represent communication paths. All of these objects (entities, ports, and relations) can be assigned attributes, which define their parameters or annotations. Ports have **links** to relations, represented in the meta model as an association between the Relation class and the Port class.

A NamedObj contains a (possibly empty) list of instances of Attribute. An Entity also contains a (possibly empty) collection of instances of Port. Ports are associated with instances of Relation, which define the connections between ports. A CompositeEntity is an Entity that contains instances of Entity and Relation. The resulting hierarchy of a model is illustrated in Figure 12.4. As described earlier, an actor is an executable entity, as indicated in Figure 12.2 by the fact that AtomicActor and CompositeActor implement the Executable interface. A director is an executable instance of the Director class, a subclass

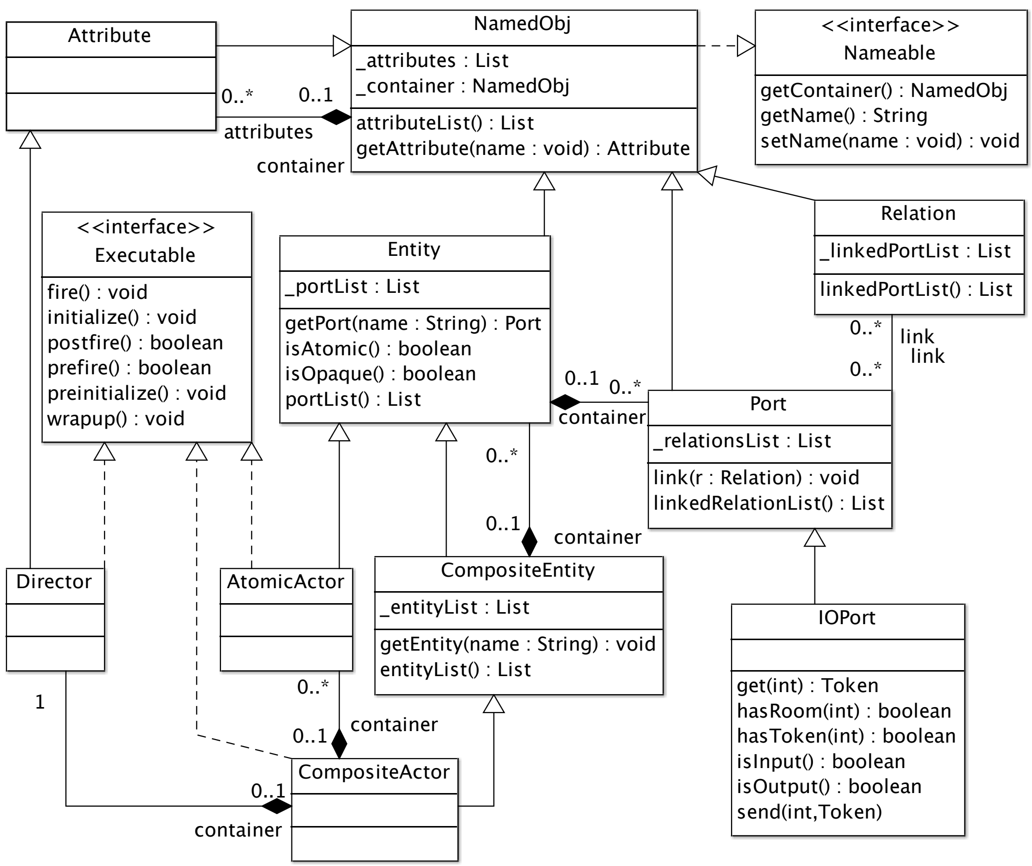


Figure 12.2: A meta model for Ptolemy II. This is a UML class diagram. The boxes represent classes with the class name, key members, and key methods shown. The lines with triangular arrowheads represent inheritance. The lines with diamond ends represent containment.

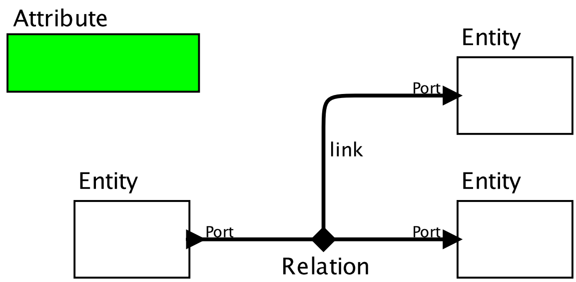


Figure 12.3: A Ptolemy II model showing the base meta-model class names for the objects in the model.

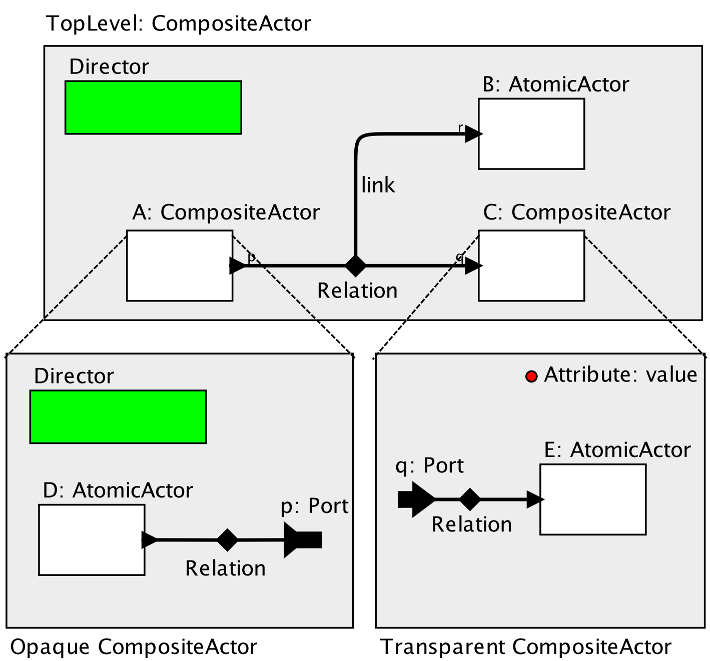


Figure 12.4: A hierarchical model showing meta-model class names for the objects in the model.

of Attribute. Each level of a hierarchical model has either one director or none; the top level always has one director.

Example 12.1: An example of a hierarchical Ptolemy II model is shown in Figure 12.4 using the concrete **visual syntax** of the Vergil visual editor.

The figure shows three distinct submodels and their hierarchical relationships. The top level of the hierarchy is labeled "TopLevel: CompositeActor," which means that its name is TopLevel and that it is an instance of CompositeActor. TopLevel contains an instance of Director, three actors, and one relation. Actors A and C are composite, whereas actor B is atomic. The ports of the three actors are linked to the relation. The ports of the composite actors appear twice in the diagram, once on the outside of the composite and once on the inside.

The block diagram in Figure 12.4 uses one of many possible concrete syntaxes for the same model. The model can also be defined in Java syntax, as shown in Figure 12.5, or in an XML schema known as MoML, as shown in Figure 12.6. All three syntaxes describe the model structure (which conforms to the abstract syntax). We will next give the structure some meaning (a semantics).

12.3 Actor Semantics and the MoC

Many Ptolemy II models are actor-oriented models; that is, they are based on connected groups of actors. In an actor-oriented model, actors execute concurrently and transfer data to each other via ports. What it means to "execute concurrently" and the manner in which data are passed between actors depend on the model of computation (MoC) in which the actor is running. In Ptolemy II, the model of computation is, in turn, defined by the director that is placed in that portion of the model.

An actor can itself be a Ptolemy II model, referred to as a composite actor. A composite actor that contains a director is said to be **opaque**; otherwise, it is **transparent**. An opaque composite actor behaves like a non-composite (i.e., atomic) actor, and its internal structure is not visible to the model in which it is used; it is a black box. In contrast, a transparent composite actor is fully visible from the outside, and is not executable on its

```java
import ptolemy.actor.AtomicActor;
import ptolemy.actor.CompositeActor;
import ptolemy.actor.Director;
import ptolemy.actor.IOPort;
import ptolemy.actor.IORelation;
import ptolemy.kernel.Relation;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;

public class TopLevel extends CompositeActor {
   public TopLevel()
           throws IllegalActionException,
           NameDuplicationException {
       super();
       // Construct top level.
       new Director(this, "Director");
       CompositeActor A = new CompositeActor(this, "A");
       IOPort p = new IOPort(A, "p");
       AtomicActor B = new AtomicActor(this, "B");
       IOPort r = new IOPort(B, "r");
       CompositeActor C = new CompositeActor(this, "C");
       IOPort q = new IOPort(C, "q");
       Relation relation = connect(p, q);
       r.link(relation);

       // Populate composite actor A.
       new Director(A, "Director");
       AtomicActor D = new AtomicActor(A, "D");
       IOPort D_p = new IOPort(D, "p");
       Relation D_r = new IORelation(A, "r");
       D_p.link(D_r);
       p.link(D_r);

       // Populate composite actor C.
       AtomicActor E = new AtomicActor(C, "E");
       IOPort E_p = new IOPort(E, "p");
       Relation E_r = new IORelation(C, "r");
       E_p.link(E_r);
       q.link(E_r);
    }
}
```

Figure 12.5: The model of Figure 12.4 given in the concrete syntax of Java.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="TopLevel" class="ptolemy.actor.CompositeActor">
    <property name="Director" class="ptolemy.actor.Director"/>
    <entity name="A" class="ptolemy.actor.CompositeActor">
        <property name="Director" class="ptolemy.actor.Director"/>
        <port name="p" class="ptolemy.actor.IOPort"/>
        <entity name="D" class="ptolemy.actor.AtomicActor">
            <port name="p" class="ptolemy.actor.IOPort"/>
        </entity>
        <relation name="r" class="ptolemy.actor.IORelation"/>
        <link port="p" relation="r"/>
        <link port="D.p" relation="r"/>
    </entity>
    <entity name="B" class="ptolemy.actor.AtomicActor">
        <port name="r" class="ptolemy.actor.IOPort"/>
    </entity>
    <entity name="C" class="ptolemy.actor.CompositeActor">
        <property name="Attribute"
                    class="ptolemy.kernel.util.Attribute"/>
        <port name="q" class="ptolemy.actor.IOPort"/>
        <entity name="E" class="ptolemy.actor.AtomicActor">
            <port name="p" class="ptolemy.actor.IOPort"/>
        </entity>
        <relation name="r" class="ptolemy.actor.IORelation"/>
        <link port="q" relation="r"/>
        <link port="E.p" relation="r"/>
    </entity>
    <relation name="r" class="ptolemy.actor.IORelation"/>
    <link port="A.p" relation="r"/>
    <link port="B.r" relation="r"/>
    <link port="C.q" relation="r"/>
</entity>
```

Figure 12.6: The model of Figure 12.4 given in the concrete syntax of MoML.

own. Opaque composite actors — black boxes — are key to **hierarchical heterogeneity**, because they allow different models of computation to be nested within a single model.

Just as we make a distinction between abstract syntax and concrete syntax, we also make a distinction between **abstract semantics** and **concrete semantics**. Consider, for example, the communication between actors. The abstract semantics captures the fact that a communication occurs (that is, one actor sends a token to another), whereas a concrete semantics captures *how* the communication occurs (e.g., whether it is rendezvous communication, asynchronous message passing, a fixed point, etc.). A director realizes a concrete semantics; the interaction between directors across levels of the hierarchy is governed by the abstract semantics.

Ptolemy II provides a particular abstract semantics, called the **actor abstract semantics**, that is central to the interoperability of directors and the ability to build heterogeneous models. The actor abstract semantics defines three distinct aspects of the actor's behavior: execution control, communication, and a model of time, each of which is discussed in detail below.*

12.3.1 Execution Control

The overall execution of a model is controlled by an instance of the Manager class. An example execution sequence for a hierarchical model with an opaque composite actor is shown in Figure 12.7. Each opaque composite actor has a director. As shown in the meta model of Figure 12.2, a Director is an Attribute that implements the Executable interface. It is rendered in Vergil as a green rectangle, as shown in Figure 12.4. Inserting a Director into a composite actor makes the composite actor executable, since it implements the Executable interface. Atomic actors also implement this interface.

The Executable interface defines the actions of the actor abstract semantics that perform computation. These actions are divided into three phases: setup, iterate, and wrapup. Each of these phases is further divided into subphases (or actions), as described below.

The **setup** phase is divided into preinitialize and initialize actions, implemented by methods of the Executable interface. In the **preinitialize** action, an actor performs any operations that may influence static analysis (including scheduling, type inference and checking, code generation, etc.). A composite actor may alter its own internal structure — by

*In this book we describe the actor abstract semantics informally. A formal framework can be found in Tripakis et al. (2013) and Lee and Sangiovanni-Vincentelli (1998).

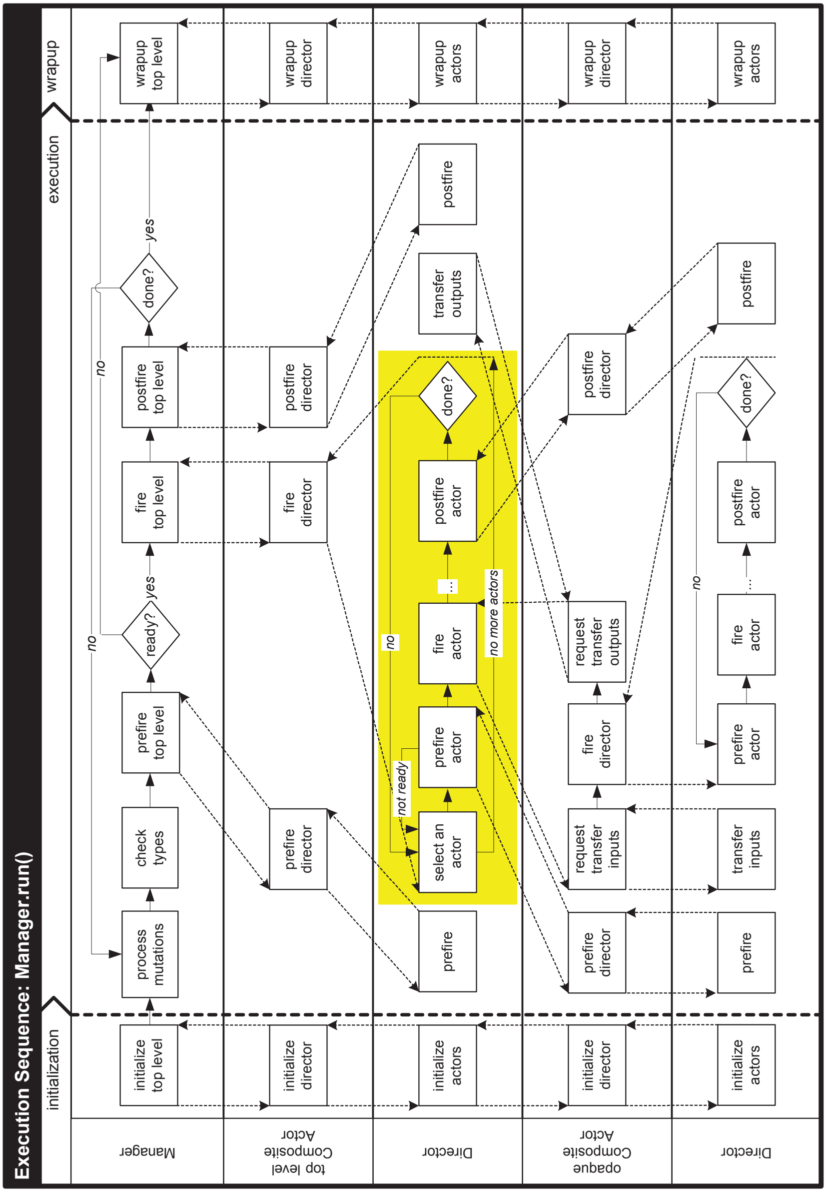


Figure 12.7: Execution of a hierarchical model with an opaque composite actor.

creating internal actors, for example — in this action. The **initialize** action of the setup phase initializes parameters, resets local state, and sends out any initial messages. Preinitialization of an actor is performed once, but initialization may be performed more than once during execution of a model. For example, initialization may be performed again if the semantics requires an actor to be re-initialized (as in the hybrid system formalism (Lee and Zheng, 2005)).

The **iterate** phase is the primary execution phase of a model. In this phase, each actor executes a sequence of **iteration**s, which are (typically finite) computations that lead the actor to a quiescent state. In a composite actor, the model of computation determines how the iteration of one actor relates to the iterations of other actors (whether they are concurrent or interleaved, how they are scheduled, etc.).

In order to coordinate the iterations among actors, an iteration is further broken into **prefire**, **fire**, and **postfire** actions. Prefire (optionally) tests the preconditions required for the actor to fire, such as the presence of sufficient inputs. The main computation of the actor is typically performed during the fire action, when it reads input data, performs computation, and produces output data. An actor may have persistent state that evolves during execution; the postfire action updates that state in response to any inputs. The fact that the state of an actor is updated only in postfire is an important part of the actor abstract semantics, as explained in the sidebar on page 433.

The **wrapup** phase is the final phase of execution. It is guaranteed to occur even if execution fails with an exception in a prior phase.

12.3.2 Communication

Like its execution control, an actor's communication capabilities are part of its abstract semantics. As described earlier, actors communicate via ports, which may be single ports or multiports. Each actor contains ports that are instances of IOPort, a subclass of Port, as shown in Figure 12.2. This subclass specifies whether a port is to be used for inputs or outputs. Two key methods that the IOPort subclass provides are `get` and `send`. As part of its fire action, an actor may use `get` to retrieve inputs, perform its computations, and use `send` to send the results to its output ports. For multiports, the integer arguments to `get` and `send` specify a channel. But what does it mean to get and send? Are communications

Sidebar: Why prefire, fire, and postfire?

Although it may not be immediately obvious, the division of each iteration of an actor's execution into prefire, fire, and postfire phases is essential for several Ptolemy II models of computation. As defined by the actor abstract semantics, the fire action reads inputs and produces outputs but does not change the state of the actor; state changes are only committed in the postfire phase. This approach is necessary for MoCs with a fixed-point semantics, which includes the synchronous-reactive (SR) and Continuous domains. Directors for such domains compute actor outputs by repeatedly firing the actors until a fixed point is reached. To ensure determinacy, it is essential that the state of each actor remain constant during these firings; the state of an actor can only be updated after the fixed point has been reached, at which point all the inputs to each actor are known. This does not occur until the postfire phase.

However, Ptolemy II does not strictly require every actor to follow this protocol. Goderis et al. (2009) classify actor-oriented MoCs into three categories of abstract semantics: **strict**, **loose**, and **loosest**. In the strict actor semantics, prefire, fire, and postfire are all finite computations, and only postfire changes the state. In the loose actor semantics, changes to the state may be made in the fire subphase. In the loosest actor semantics, the fire subphase may not even be finite; it may be a non-terminating computation.

An actor that conforms with the strict actor semantics is the most flexible type of actor in the sense that it may be used in any domain, including SR and Continuous. Such an actor is said to be **domain polymorphic**. Most actors in the library are domain polymorphic. An actor that conforms only with the loose actor semantics can be used with fewer directors (dataflow, for example). Those actors will be listed in domain-specific libraries. An actor that conforms only with the loosest actor semantics can be used with still fewer directors (process networks, for example). It is possible to define actors that will only work with a single type of director.

A director implements the same phases of execution as an actor. Thus, placing a director into a composite actor endows that composite actor with an executable semantics. If the director conforms to the strict actor semantics, then that composite actor is domain polymorphic. Such directors support the most flexible form of hierarchical heterogeneity in Ptolemy II because multiple directors with different MoCs may be combined hierarchically within a single model.

to be interpreted as a FIFO queue, a rendezvous communication, or other communication type? This meaning is specified by the director, not the actor.

A director determines how actors communicate by creating a **receiver** and placing it in an input port, with one receiver for each communication channel. A receiver is an object that implements the Receiver interface, as shown in Figure 12.8. That interface includes `put` and `get` methods. As illustrated in Figure 12.9, when one actor calls the `send` method of its output port, the output port delegates the call to the `put` method of the receiver in the designation input port(s). Similarly, when an actor calls the `get` method of an input port, the input port delegates the call to the `get` method of the receiver in the port. Thus, since the director provides the receiver, the director controls what it means to send and receive data.

Receivers can implement FIFO queues, mailboxes, proxies for a global queue, rendezvous, etc., all of which conform to the meta model shown in Figure 12.8. Directors provide receivers that implement a communication mechanism that is appropriate to the model of computation. Several receiver classes are shown in Figure 12.8.

Example 12.2: The PNReceiver, which is a subclass of QueueReceiver, is used by the process networks (PN) director. The `put` method of PNReceiver appends a data token t to a FIFO queue and then returns. When it returns, there is no assurance that the message has been received. The PN domain implements nonblocking writes; it delivers a token without waiting for the recipient to be ready to receive it, and thus will not block the model's continued execution.

In PN, every actor runs in its own Java thread. The actor receiving a message will therefore be running asynchronously in a different thread than the actor sending the token. The receiving actor will call the `get` method of its input port, which will delegate to the `get` method of the PNReceiver. The latter will block the calling thread until the FIFO queue has at least one token. It then returns the first token in the queue. Thus, the PNReceiver implements the blocking reads required by the PN domain. These blocking reads help ensure that a PN model is determinate.

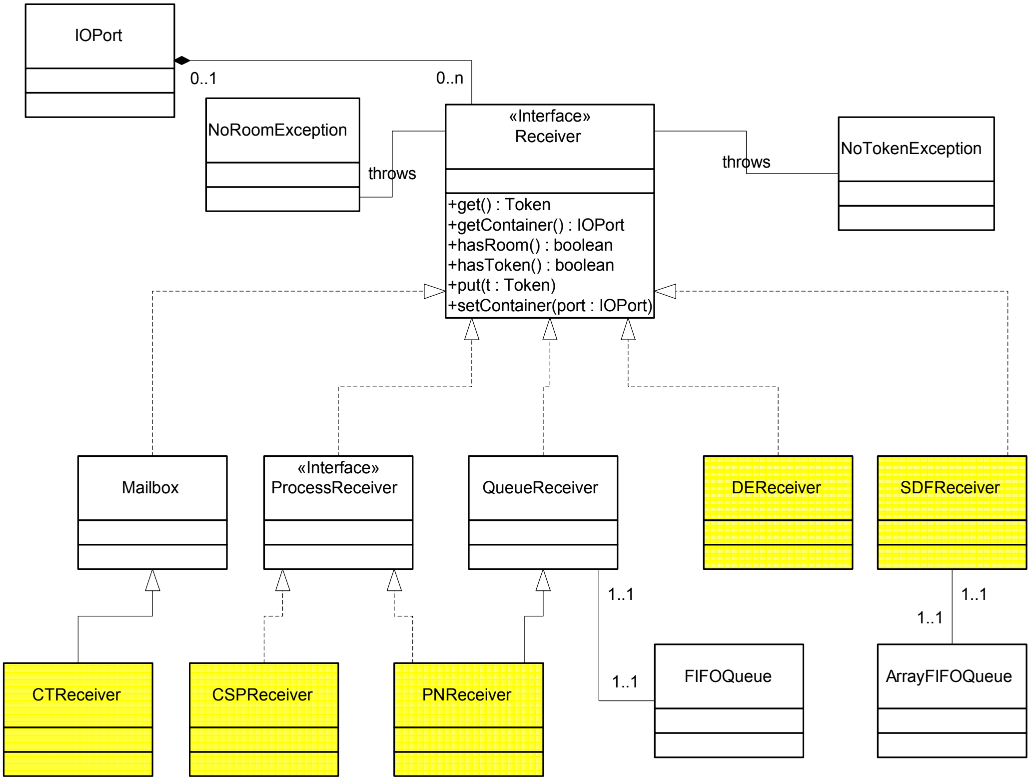


Figure 12.8: Meta model for communication in Ptolemy II.

12.3.3 Time

The final piece of the abstract actor semantics is the notion of time; that is, the way in which an actor views the passage of time when it is used in timed domains.

When an actor fires, it can ask its director for the current time. It does so with the following code:

```
Time currentTime = getDirector().getModelTime();
```

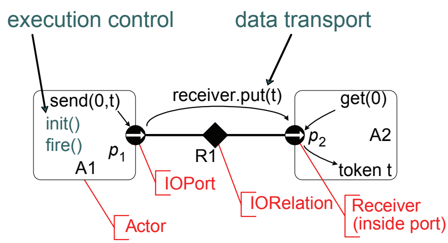


Figure 12.9: Communication mechanism in Ptolemy II.

The time returned by the director is called the **current** model time. If the actor requests the current model time only in its postfire subphase, then it is assured that the time value returned will be nondecreasing. If it requests the time in the fire subphase, however, then there is no such assurance, because some directors (such as the Continuous director, see Chapter 9) speculatively advance time while converging to a fixed point. In this case, the current model time may actually be *less* than its value in a prior invocation of `fire`.

An actor can request that it be fired at some future time by invoking the `fireAt` method of its director. The director is responsible for ensuring that the actor is fired at the requested future time. If it cannot honor the request, the director returns an alternative time at which it can fire the actor. Note, however, that the actor cannot assume that its *next* firing will be at that future time; there may be an arbitrary number of intervening firings. Moreover, if the execution of the model ends before time advances to the requested future time, then the actor will not be fired at the requested time.

The model hierarchy is central to the management of time. Typically, only the top-level director advances time. Other directors in a model obtain the current model time from their enclosing director. If the top-level director does not implement a timed model of computation, then time does not advance.

Perhaps counterintuitively, even untimed domains provide access to time. In a hierarchical model, unless the untimed domain is at the top level, it will delegate operations relating to time up the hierarchy to its container.

Timed and untimed models of computation may be interleaved in the hierarchy (see Section 1.7.1). There are certain combinations that do not make sense, however. For example, if the top-level director never advances time, and an actor requests a firing at a future time, then the request cannot be honored. The director's `fireAt` method will return the time at which it can fire the actor, which will be time zero, since it never advances time. It is up to the actor to either accept this or to throw an exception to indicate that it is incompatible with an enclosing director.

Time can also advance non-uniformly in a model, as explained in Section 1.7.1. In particular, in modal models (Chapter 8), the advancement of time can be temporarily suspended (Lee and Tripakis, 2010). Within the submodel, there is a monotonically non-decreasing gap between the local time and the time in the enclosing environment. This mechanism is used to model temporary suspension of a submodel, as explained in Section 8.5.

12.4 Designing Actors in Java†

The functionality of actors in Ptolemy II can be defined in a number of ways. The most basic mechanism is the use of hierarchy, in which an actor is defined as the composite of other actors. For actors that implement complex mathematical functionality, however, it is often more convenient to use the Expression actor, whose functionality is defined using the expression language described in Chapter 13. Actors can also be created using the **MatlabExpression** actor, by defining the behavior as a MATLAB script. An actor can be defined in the Python language using the **PythonActor** or **PythonScript** actor, or can be defined using the **Cal** actor definition language (Eker and Janneck, 2003). But the most flexible method is to define the actor in Java, which is the focus of our discussion.

As described earlier, some actors are designed to be domain polymorphic, meaning that they can operate in multiple domains. Here, we focus on designing actors that are domain polymorphic. We will also focus on designing polymorphic actors that operate on a wide variety of token data types. Domain and data polymorphism help maximize the reusability of actors and minimize duplicated code when building an actor library.

Code duplication can also be avoided by using object-oriented inheritance. Inheritance can also help to enforce consistency across a set of actors. Most actors in the default library extend a common set of base classes that enforce uniform naming of commonly used ports and parameters. Using common base classes avoids unnecessary heterogeneity such as input ports named "in" vs. "inputSignal" vs. "input." This makes actors easier to use, and their code easier to maintain. To this end, we recommend using a reasonably deep

†This section assumes some familiarity with Java and object-oriented design.

class hierarchy to promote consistency. It is better to subclass and override an existing actor than to copy it and modify the copy.

Note that the Java source code for existing Ptolemy II actors, which can be viewed using the `Open Actor` context menu item, can provide a useful reference for defining new actors[‡].

Each actor consists of a source code file written in Java. An example of the source code for a simple actor is shown in Figure 12.10. This text can be placed in a Java file, compiled, instantiated in Vergil, and used as an actor. To create a new actor and use it in Vergil, choose a Java development environment (such as Eclipse), create a Java file, save the Java file in your classpath,[§] and instantiate the actor in Vergil. The latter can be accomplished by specifying the fully qualified class name in the dialog opened by the [`Graph→Instantiate Entity`] menu item. For example, you could copy the text in Figure 12.10 into a file named Count.java, save the file in the home directory of your Ptolemy installation, and then create an instance of this actor in Vergil using [`Graph→ Instantiate Entity`].

In the source code shown in Figure 12.10, Lines 1 through 8 specify the Ptolemy classes on which this actor depends. The source code for those classes can also be viewed; Java development environments like Eclipse make it easy to view these files.

Line 10 defines a class named Count that subclasses TypedAtomicActor (the base class for most Ptolemy II actors whose ports and parameters have types). This particular actor could instead subclass Source or LimitedFiringSource, both of which would provide the needed ports. But here, for illustrative purposes, we include the port definitions. A particularly useful base class for actors is Transformer, shown in Figure 12.11. It is a reasonable choice for actors with one input and one output port.

Lines 12-20 give the constructor. The constructor is the Java procedure that creates instances of the class. The constructor takes arguments that define where the actor will be placed and the actor name. In the body of the constructor, line 15 creates an input port named *trigger*. The third and fourth arguments to the constructor for TypedIOPort designate that this port is an input and not an output. By convention, in Ptolemy II, every port

‡ If you plan to contribute custom actors to the open source collection of actors in Ptolemy II, please be sure to follow the coding style given by Brooks and Lee (2003).

§ The classpath is defined by an environment variable called CLASSPATH that Java uses to search for class definitions. By default, when you run Vergil, if you have a directory called ".ptolemyII" in your home directory, then that directory will be in your classpath. You can put Java class files there and Vergil will find them.

```
import ptolemy.actor.TypedAtomicActor;
import ptolemy.actor.TypedIOPort;
import ptolemy.data.IntToken;
import ptolemy.data.expr.Parameter;
import ptolemy.data.type.BaseType;
import ptolemy.kernel.CompositeEntity;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;

public class Count extends TypedAtomicActor {
    /** Constructor */
    public Count(CompositeEntity container, String name)
            throws NameDuplicationException,
            IllegalActionException {
        super(container, name);
        trigger = new TypedIOPort(this, "trigger", true, false);
        initial = new Parameter(this, "initial", new IntToken(0));
        initial.setTypeEquals(BaseType.INT);
        output = new TypedIOPort(this, "output", false, true);
        output.setTypeEquals(BaseType.INT);
    }
    /** Ports and parameters. */
    public TypedIOPort trigger, output;
    public Parameter initial;

    /** Action methods. */
    public void initialize() throws IllegalActionException {
        super.initialize();
        _count = ((IntToken)initial.getToken()).intValue();
    }
    public void fire() throws IllegalActionException {
        super.fire();
        if (trigger.getWidth() > 0 && trigger.hasToken(0)) {
             trigger.get(0);
        }
        output.send(0, new IntToken(_count + 1));
    }
    public boolean postfire() throws IllegalActionException {
        _count += 1;
        return super.postfire();
    }
    private int _count = 0; /** Local variable. */
}
```

Figure 12.10: A simple Count actor.

```java
public class Transformer extends TypedAtomicActor {

    /** Construct an actor with the given container and name.
     *  @param container The container.
     *  @param name The name of this actor.
     *  @exception IllegalActionException If the actor
     *   cannot be contained by the proposed container.
     *  @exception NameDuplicationException If the container
     *   alreadyhas an actor with this name.
     */
    public Transformer(CompositeEntity container, String name)
            throws NameDuplicationException,
            IllegalActionException {
        super(container, name);
        input = new TypedIOPort(this, "input", true, false);
        output = new TypedIOPort(this, "output", false, true);
    }

    ///////////////////////////////////////////////////////
    ////            ports and parameters           ////

    /** The input port. This base class imposes no type
     *  constraints except that the type of the input
     *  cannot be greater than the type of the output.
     */
    public TypedIOPort input;

    /** The output port. By default, the type of this output
     *  is constrained to be at least that of the input.
     */
    public TypedIOPort output;
}
```

Figure 12.11: Transformer is a useful base class for actors with one input and one output.

is visible as a public field (defined in this case on line 22), and the name of the public field matches the name given as a constructor argument on line 15. Matching these names is important for actor-oriented classes, explained in Section 2.6, to work correctly.

Line 16 defines a parameter named *initial*. Again, by convention, parameters are public fields with matching names, as shown on line 23. Line 17 specifies the data type of the parameter, constraining its possible values.

Lines 18 and 19 create the output port and set its data type. Nothing in this actor constrains the type of the *trigger* input, so any data type is acceptable.

The `initialize` method, given on lines 26 to 29, initializes the private local variable `_count` to the value of the *initial* parameter. By convention in Ptolemy II, the names of private and protected variables begin with an underscore. The cast to *IntToken* is safe here because the parameter type is constrained to be an integer.

The `fire` method on lines 30-36 reads the input port if it is connected (that is, if it has a width greater than zero) and has a token. In some domains, such as DE, it is important to read input tokens even if they are not going to be used. In particular, the DE director will repeatedly fire an actor that has unconsumed tokens on its inputs; failure to read an input will result an infinite sequence of firings. Line 35 sends an output token.

The `postfire` method on lines 37-40 updates the state of the actor by incrementing the private variable `_count`. As explained above, updating the state in `postfire` rather than `fire` enables the use of this actor with directors such as SR and Continuous that repeatedly fire an actor until reaching a fixed point.

12.4.1 Ports

By convention, ports are public members of actors. They represent a set of input and output channels through which tokens may pass to other ports. Figure 12.10 shows how to define ports as public fields and instantiate them in the constructor. Here, we describe a few options that may be useful when creating ports.

Multiports and Single Ports

A port can be a single port or a multiport. By default, a port is a single port. It can be declared to be a multiport as follows:

```
portName.setMultiport(true);
```

Each port has a width, which corresponds to its number of channels. If a port is not connected, the width is zero. If a port is a single port, the width can be zero or one. If a port is a multiport, the width can be larger than one.

Reading and Writing

Data (encapsulated in a token) can be sent to a particular channel of an output port using the following syntax:

```
portName.send(channelNumber, token);
```

where `channelNumber` begins with 0 for the first channel. The width of the port (which is the number of channels) can be obtained by

```
int width = portName.getWidth();
```

If the port is unconnected, then the token is not sent anywhere. The `send` method will simply return. Note that in general, if the channel number refers to a channel that does not exist, the `send` method simply returns without issuing an exception. In contrast, attempting to read from a nonexistent input channel will usually result in an exception.

A token can be sent to all output channels of a port by

```
portName.broadcast(token);
```

You can generate a token from a value and then send this token by

```
portName.send(channelNumber, new IntToken(integerValue));
```

A token can be read from a channel by

```
Token token = portName.get(channelNumber);
```

You can read from channel 0 of a port and extract the data value (assuming the type is known) by

```
double variableName = ((DoubleToken)
        portName.get(0)).doubleValue();
```

You can query an input port to determine whether a `get` will succeed (whether a token is available) by

```
boolean tokenAvailable = portName.hasToken(channelNumber);
```

You can also query an output port to see whether a `send` will succeed using

```
boolean spaceAvailable = portName.hasRoom(channelNumber);
```

although with many domains (like SDF and PN), the answer is always true.

Dependencies Between Ports

Many Ptolemy II domains perform analysis of the topology of a model as part of the process of scheduling the model's execution. SDF, for example, constructs a static schedule that sequences the invocations of actors. DE, SR, and Continuous all examine data dependencies between actors to prioritize reactions to simultaneous events. In all of these cases, the director requires additional information about the behavior of actors in order to perform the analysis. In this section, we explain how to provide that additional information.

Suppose you are designing an actor that does not require a token at its input port in order to produce a token on its output port when it fires. It is useful for the director to have access to this information, which can be conveyed within the actor's Java code. For example, the MicrostepDelay actor declares that its output port is independent of its input port by defining this method:

```
public void declareDelayDependency()
        throws IllegalActionException {
    _declareDelayDependency(input, output, 0.0);
}
```

By default, each output port is assumed to have a dependency on all input ports. By defining the above method, the MicrostepDelay actor alerts the director that this default is not applicable. There is a delay between the ports *input* and *output*. The delay here is declared to be 0.0, which is interpreted as a microstep delay. The scheduler can use this information to sequence the execution of the actors and to resolve causality loops. For domains that do not use dependency information (such as SDF), it is harmless to include the above method. Thus, these declarations help maximize the ability to reuse actors in a variety of domains.

Port Production and Consumption Rates

Some domains (notably SDF) make use of information about production and consumption rates at the ports of actors. If the author of an actor makes no specific assertion, the SDF director will assume that upon firing, the actor requires and consumes exactly one token on each input port, and produces exactly one token on each output port. To override this assumption, the author needs to include a parameter in the port that is named either *tokenConsumptionRate* (for input ports) or *tokenProductionRate* (for output ports). The value of these parameters is an integer that specifies the number of tokens consumed or produced in a firing. As always, the value of these parameters can be given by an expression that depends on other parameters of the actor. As in the previous example, these parameters have no effect in domains that do not use this information, but they enable actors to be used within domains that do (such as SDF).

Feedback loops in SDF require at least one actor in the loop to produce tokens in its `initialize` method. To alert the SDF scheduler that an actor includes this capability, the relevant output port must include an integer-valued parameter (named *tokenInitProduction*) that specifies the number of tokens initially produced. The SDF scheduler will use this information to determine that a model with cycles does not deadlock.

12.4.2 Parameters

Like ports, parameters are public members of actors by convention, and the name of the public member is required to match the name passed to the constructor of the parameter. Type constraints on parameters are specified in the same way as for ports.

An actor is notified when a parameter value has changed by having its method `attributeChang` called. If the actor needs to check parameter values for validity, it can do so by overriding this method. Consider the example shown in Figure 12.12, taken from the PoissonClock actor. This actor generates timed events according to a Poisson process. One of its parameters is *meanTime*, which specifies the mean time between events. This must be a double, as asserted in the constructor, but equally importantly, it is required to be positive. The actor can enforce this requirement as shown in lines 21-25, which will throw an exception if a non-positive value is given.

The `attributeChanged` method may also be used to cache the current value of a parameter, as shown on lines 26-28.

```java
public class PoissonClock extends TimedSource {
    public PoissonClock(CompositeEntity container, String name)
            throws NameDuplicationException,
            IllegalActionException {
        super(container, name);
        meanTime = new Parameter(this, "meanTime");
        meanTime.setExpression("1.0");
        meanTime.setTypeEquals(BaseType.DOUBLE);
        ...
    }
    public Parameter meanTime;
    public Parameter values;

    /** If the argument is the meanTime parameter,
     *  check that it is positive.
     */
    public void attributeChanged(Attribute attribute)
            throws IllegalActionException {
        if (attribute == meanTime) {
            double mean = ((DoubleToken)meanTime.getToken())
                    .doubleValue();
            if (mean <= 0.0) {
                throw new IllegalActionException(this,
                    "meanTime is required to be positive."
                    " Value given: " + mean);
            }
        } else if (attribute == values) {
            ArrayToken val = (ArrayToken)
                    (values.getToken());
            _length = val.length();
        } else {
            super.attributeChanged(attribute);
        }
    }
    ...
}
...
}
```

Figure 12.12: Illustration of the use of `attributeChanged` to validate parameter values.

12.4.3 Coupled Port and Parameter

Often, in the design of an actor, it is hard to decide whether a quantity should be specified by a port or by a parameter. Fortunately, you can easily design an actor to offer both options. An example of such an actor is the Ramp actor, which uses the code shown in Figure 12.13. This actor starts with an initial value given by the *init* parameter, which is then incremented by the value of *step*. The value of *step* can be specified by either a parameter named *step* or by a port named *step*. If the port is left unconnected, then the value will always be set by the parameter. If the port is connected, then the parameter provides the initial default value, and this value is subsequently replaced by any value that arrives on the port.

The parameter value is stored with the model containing the Ramp actor when it is saved to a MoML file. In contrast, any data that arrives on the port during execution of the model is not stored. Thus, the default value given by the parameter is persistent, while the values that arrive on the port are transient.

To support the use of both a parameter and a port, the Ramp actor creates an instance of the class PortParameter in its constructor, as shown in Figure 12.13. This is a parameter that creates an associated port with the same name. The `postfire` method first calls `update` on *step*, and then adds its value to the state. Calling `update` has the side effect of reading from the associated input port, and if a token is present there, updating the value of the parameter. It is essential to call `update` before reading the value of a PortParameter in order to ensure that any input token that might be available on the associated input port is consumed.

12.5 Summary

This chapter has provided a brief introduction to the software architecture of Ptolemy II. It explains the overall layout of the classes that comprise Ptolemy II and how key classes in the kernel package define the structure of a model. It also explains how key classes in the actor package define the execution of a model. And finally, it gives a brief introduction to writing custom actors in Java.

```
public class Ramp extends SequenceSource {
    public Ramp(CompositeEntity container, String name)
            throws NameDuplicationException,
            IllegalActionException {
        super(container, name);
        init = new Parameter(this, "init");
        init.setExpression("0");
        step = new PortParameter(this, "step");
        step.setExpression("1");
        ...
    }
    public Parameter init;
    public PortParameter step;
    public void attributeChanged(Attribute attribute)
            throws IllegalActionException {
        if (attribute == init) {
            _stateToken = init.getToken();
        } else {
            super.attributeChanged(attribute);
        }
    }
    public void initialize() throws IllegalActionException {
        super.initialize();
        _stateToken = init.getToken();
    }
    public void fire() throws IllegalActionException {
        super.fire();
        output.send(0, _stateToken);
    }
    ...
    public boolean postfire() throws IllegalActionException {
        step.update();
        _stateToken = _stateToken.add(step.getToken());
        return super.postfire();
    }
    private Token _stateToken = null;
}
```

Figure 12.13: Code segments from the Ramp actor.

13 Expressions

Christopher Brooks, Thomas Huining Feng, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Neil Smyth, Yuhong Xiong

Contents

In Ptolemy II, models specify computations by composing actors. Many computations, however, are awkward to specify this way. A common situation is where we wish to evaluate a simple algebraic expression, such as $\sin(2\pi(x-1))$. It is possible to express this computation by composing actors in a block diagram, but it is far more convenient to give it textually.

The Ptolemy II **expression language** provides infrastructure for specifying algebraic expressions textually and for evaluating them. The expression language is used to specify the values of parameters, guards and actions in state machines, and for the calculation performed by the Expression actor. In fact, the expression language is part of the generic infrastructure in Ptolemy II, and it can be used by programmers extending the Ptolemy II system. In this chapter, we describe how to use expressions from the perspective of a user rather than a programmer.

Vergil provides an interactive **expression evaluator**, which is accessed through the menu command [`File→New→Expression Evaluator`]. This operates like an interactive command shell, and is shown in Figure 13.1. It supports a command history. To access the previously entered expression, type the up arrow or Control-P. To go back, type the down arrow or Control-N. The expression evaluator is useful for experimenting with expressions.

13.1 Simple Arithmetic Expressions

13.1.1 Constants and Literals

The simplest expression is a constant, which can be given either by the symbolic name of the constant, or by a literal. By default, the symbolic names of constants supported are:

`PI`, `pi`, `E`, `e`, `true`, `false`, `i`, `j`, `NaN`, `Infinity`, `PositiveInfinity`, `NegativeInfinity`, `MaxUnsignedByte`, `MinUnsignedByte`, `MaxShort`, `MinShort`, `MaxInt`, `MinInt`, `MaxLong`, `MinLong`, `MaxFloat`, `MinFloat`, `MaxDouble`, and `MinDouble`. For example,

```
PI/2.0
```

is a valid expression that refers to the symbolic name "`PI`" and the literal "`2.0`." The constants `i` and `j` are the imaginary number with value equal to $\sqrt{-1}$. The constant `NaN` is "**not a number**," which for example is the result of dividing `0.0/0.0`. The constant `Infinity` is the result of dividing `1.0/0.0`. The constants that start with "`Max`" and "`Min`" are the maximum and minimum values for their corresponding types.

Numerical values without decimal points, such as "`10`" or "`-3`" are integers (type *int*). Numerical values with decimal points, such as "`10.0`" or "`3.14159`" are of type *double*. Numerical values followed by "`f`" or "`F`" are of type *float*. Numerical values without decimal points followed by the character "`l`" (el) or "`L`" are of type *long*. long. Numerical values without decimal points followed by the character "`s`" or "`S`" are of type *short*. Unsigned integers followed by "`ub`" or "`UB`" are of type *unsignedByte*, as in "`5ub`". An *unsignedByte* has a value between 0 and 255; note that it is not quite the same as the Java byte, which has a value between -128 and 127. Numbers of type *int*, *long*, *short* or *unsignedByte* can be specified in decimal, octal, or hexadecimal. Numbers beginning with a leading "`0`" are octal numbers. Numbers beginning with a leading "`0x`" are hexadecimal numbers. For example, "`012`" and "`0xA`" are both equal to the integer 10.

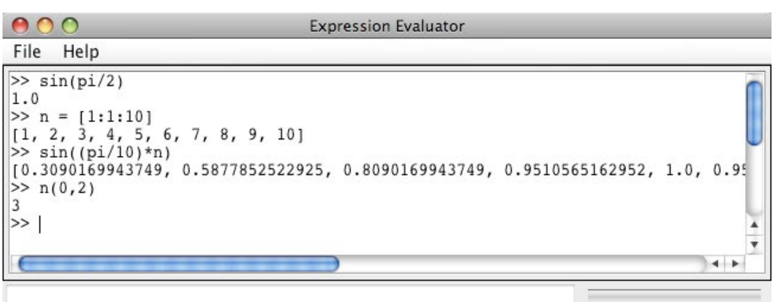


Figure 13.1: The Expression Evaluator.

A *complex* is defined by appending an "i" or a "j" to a *double* for the imaginary part. This gives a purely imaginary *complex* number which can then leverage the polymorphic operations in the Token classes to create a general *complex* number. Thus `2 + 3i` will result in the expected *complex* number. You can optionally write this `2 + 3*i`.

Literal string constants are also supported. Anything between *double* quotation marks, `"..."`, is interpreted as a *string* constant. The following built-in string-valued constants are defined:

variable name	**meaning**	**JVM property name**	**example value**
PTII	The directory in which Ptolemy II is installed	ptolemy.ptII.dir	c:\tmp
HOME	The user home directory	user.home	c:\Documents and Settings \you
CWD	The current working directory	user.dir	c:\ptII
TMPDIR	The temporary directory	java.io.tmpdir	c:\Documents and Settings \you\Local Settings\Temp\
USERNAME	The user account name	user.name	ptolemy

The value of these variables is the value given by the corresponding Java virtual machine (**JVM**) property, such as `user.home` for `HOME`. The properties `user.dir` and `user.home` are standard in Java. Their values are platform dependent; see the documentation for the method `getProperties` in the java.lang.System class for details.* Vergil will display all the Java properties if you invoke [View→JVM Properties] in the menu of a Graph Editor.

The `ptolemy.ptII.dir` property is set automatically when Vergil or any other Ptolemy II executable is started up. You can also set it when you start a Ptolemy II process using the `java` command by a syntax like the following:

```
java -Dptolemy.ptII.dir=${PTII} classname
```

where `classname` is the full class name of a Java application. You can similarly set the other variables in the table. For example, to invoke Vergil in a particular directory, use

*Note that `user.dir` and `user.home` are usually not readable in unsigned applets, in which case, attempts to use these variables in an expression will result in an exception.

variable name	value	variable name	value
CLASSPATH	”xxxxxxCLASSPATHxxxxxx”	CWD	”/Users/eal”
E	2.718281828459	HOME	”/Users/eal”
Infinity	Infinity	MaxDouble	1.797693134862316E308
MaxFloat	3.4028234663852886E38	MaxInt	2147483647
MaxLong	9223372036854775807L	MaxShort	32767s
MaxUnsignedByte	255ub	MinDouble	4.9E-324
MinFloat	1.401298464324817E-45	MinInt	-2147483648
MinLong	-9223372036854775808L	MinShort	-32768s
MinUnsignedByte	0ub	NaN	NaN
NegativeInfinity	-Infinity	PI	3.1415926535898
PTII	”/ptII”	PositiveInfinity	Infinity
TMPDIR	”/tmp”	USERNAME	”eal”
backgroundColor	{0.9, 0.9, 0.9, 1.0}	boolean	false
complex	0.0 + 0.0i	double	0.0
e	2.718281828459	false	false
fixedpoint	fix(0, 2, 2)	float	0.0f
general	present	i	0.0 + 1.0i
int	0	j	0.0 + 1.0i
long	0L	matrix	[]
nil	nil	null	object(null)
object	object(null)	pi	3.1415926535898
scalar	present	short	0s
string	””	true	true
unknown	present	unsignedByte	0ub
xmltoken	null		

Table 13.1: An example of the values returned by the `constants` function

```
java -cp /ptII -Duser.dir=/Users/eal \
      ptolemy.vergil.VergilApplication
```

The `-cp` option specifies the classpath (which has to include the root directory of Ptolemy II), the `-D` option specifies the property to set, and the final argument is the class that includes a `main` method that invokes Vergil.

The `constants` utility function returns a record with all the globally defined constants. If you open the expression evaluator and invoke this function, you will see that its value is similar to what is shown in Figure 13.1.

13.1.2 Variables

Expressions can contain identifiers that are references to variables within the **scope** of the expression. For example,

```
PI*x/2.0
```

is valid if "x" a variable in scope. In the expression evaluator, the variables that are in scope include the built-in constants plus any assignments that have been previously made. For example,

```
>> x = pi/2
1.5707963267949
>> sin(x)
1.0
```

In the context of Ptolemy II models, the variables in scope include all parameters defined at the same level of the hierarchy or higher. So for example, if an actor has a parameter named "x" with value `1.0`, then another parameter of the same actor can have an expression with value "`PI*x/2.0`", which will evaluate to $\pi/2$.

Consider a parameter *P* in actor X which is in turn contained by composite actor Y. The scope of an expression for *P* includes all the parameters contained by X and Y, plus those of the container of Y, its container, etc. That is, the scope includes any parameters defined above in the hierarchy.

You can add parameters to actors (composite or not) by right clicking on the actor, selecting [`Customize→Configure`] and then clicking on "`Add`," or by dragging in a parameter from the `Utilities` library. Thus, you can add variables to any scope, a capability that serves the same role as the "let" construct in many functional programming languages.

Occasionally, it is desirable to access parameters that are not in scope. The expression language supports a limited syntax that permits access to certain variables out of scope. In particular, if in place of a variable name `x` in an expression you write `A::x`, then instead of looking for `x` in scope, the interpreter looks for a container named `A` in the scope and a parameter named `x` in `A`. This allows reaching down one level in the hierarchy from either the current container or any of its containers.

13.1.3 Operators

The arithmetic operators are `+`, `-`, `*`, `/`, `^`, and `%`. Most of these operators operate on most data types, including arrays, records, and matrices. The `^` operator computes "to the power of" or exponentiation, where the exponent can only be a type that losslessly converts to an integer such as an *int*, *short*, or an *unsignedByte*.

The *unsignedByte*, *short*, *int* and *long* types can only represent integer numbers. Operations on these types are integer operations, which can sometimes lead to unexpected results. For instance, `1/2` yields `0`, since `1` and `2` are integers, whereas `1.0/2.0` yields `0.5`. The exponentiation operator "`^`" when used with negative exponents can similarly yield unexpected results. For example, `2^-1` is `0` because the result is computed as `1/(2^1)`.

The `%` operation is a modulo or remainder operation. The result is the remainder after division. The sign of the result is the same as that of the dividend (the left argument). For example,

```
>> 3.0 % 2.0
1.0
>> -3.0 % 2.0
-1.0
>> -3.0 % -2.0
-1.0
>> 3.0 % -2.0
1.0
```

The magnitude of the result is always less than the magnitude of the divisor (the right argument). Note that when this operator is used on doubles, the result is not the same as that produced by the `remainder` function (see Table 13.6). For instance,

```
>> remainder(-3.0, 2.0)
1.0
```

The `remainder` function calculates the IEEE 754 standard remainder operation. It uses a rounding division rather than a truncating division, and hence the sign can be positive or negative, depending on complicated rules (see Section 13.4.8). For example, counter-intuitively,

```
>> remainder(3.0, 2.0)
-1.0
```

When an operator involves two distinct types, the expression language has to make a decision about which type to use to implement the operation. If one of the two types can be converted without loss into the other, then it will be. For instance, *int* can be converted losslessly to *double*, so 1.0/2 will result in 2 being first converted to 2.0, so the result will be 0.5. Among the scalar types, *unsignedByte* can be converted to anything else, *short* can be converted to *int*, *int* can be converted to *double*, *float* can be converted to *double* and *double* can be converted to *complex*. Note that *long* cannot be converted to *double* without loss, nor vice versa, so an expression like `2.0/2L` yields the following error message:

```
Error evaluating expression "2.0/2L"
in .Expression.evaluator
Because:
divide method not supported between ptolemy.data.DoubleToken
'2.0' and ptolemy.data.LongToken '2L' because the types are
incomparable.
```

Just as *long* cannot be cast to *double*, *int* cannot be cast to *float* and vice versa.

All scalar types have limited precision and magnitude. As a result of this, arithmetic operations are subject to underflow and overflow.

- For *double* numbers, overflow results in the corresponding positive or negative infinity. Underflow (i.e. the precision does not suffice to represent the result) will yield zero.
- For *int* and *fixedpoint* types, overflow results in wraparound. For instance, the value of `MaxInt` is `2147483647`, but the expression `MaxInt + 1` yields `-2147483648`. Similarly, while `MaxUnsignedByte` has value `255ub`, `MaxUnsignedByte + 1ub` has value `0ub`. Note, however, that `MaxUnsignedByte + 1` yields `256`, which is an *int*, not an *unsignedByte*. This is because `MaxUnsignedByte` can be losslessly converted to an *int*, so the addition is *int* addition, not *unsignedByte* addition.

The bitwise operators are `&`, `|`, `#` and `~`. They operate on *boolean*, *unsignedByte*, *short*, *int* and *long* (but not *fixedpoint*, *float*, *double* or *complex*). The operator `&` is bitwise AND, `~` is bitwise NOT, and `|` is bitwise OR, and `#` is bitwise XOR (exclusive or, after MATLAB).

The relational operators are <, <=, >, >=, == and `!=`. They return type *boolean*. Note that these relational operators check the values when possible, irrespective of type. So, for example,

```
1 == 1.0
```

returns *true*. If you wish to check for equality of both type and value, use the `equals` method, as in

```
>> 1.equals(1.0)
false
```

Boolean-valued expressions can be used to give conditional values. The syntax for this is

```
boolean ? value1 : value2
```

If the boolean is true, the value of the expression is `value1`; otherwise, it is `value2`. The logical boolean operators are `&&`, `||`, `!`, `&` and `|`. They operate on type *boolean* and return type *boolean*. The difference between logical `&&` and logical `&` is that `&` evaluates all the operands regardless of whether their value is now irrelevant. Similarly for logical `||` and `|`. This approach is borrowed from Java. Thus, for example, the expression `false && x` will evaluate to false irrespective of whether `x` is defined. On the other hand, `false & x` will throw an exception if `x` is undefined.

The << and >> operators performs arithmetic left and right shifts respectively. The >>> operator performs a logical right shift, which does not preserve the sign. They operate on *unsignedByte*, *short*, *int*, and *long*.

13.1.4 Comments

In expressions, anything inside `/*...*/` is ignored, so you can insert comments.

13.2 Uses of Expressions

Expressions are used in Ptolemy II to assign values to parameters, to specify the input-output function realized by an Expression actor, and to specify guards and actions in state machines.

13.2.1 Parameters

The values of most parameters of actors can be given as expressions[†]. The variables in the expression refer to other parameters that are in scope, which are those contained by the same container or some container above in the hierarchy. They can also reference variables in a **scope-extending attribute**, which includes variables defining units, as explained below in section 13.7. Adding parameters to actors is straightforward, as explained in chapter 2.

13.2.2 Port Parameters

It is possible to define a parameter that is also a port. Such a **PortParameter** provides a default value, which is specified like the value of any other parameter. When the corresponding port receives data, however, the default value is overridden with the value provided at the port. Thus, this object functions like a parameter and a port. The current value of the PortParameter is accessed like that of any other parameter. Its current value will be either the default or the value most recently received on the port.

A PortParameter might be contained by an atomic actor or a composite actor. To put one in a composite actor, drag it into a model from the `Utilities` library, as shown in Figure 13.2.

To be useful, a PortParameter has to be given a name (the default name, "portParameter," is not very compelling). To change the name, right click on the icon and select [`Customize→Rename`], as shown in Figure 13.2. In the figure, the name is set to "noiseLevel." Then set the default value by double clicking. In the figure, the default value is set to 10.0.

An example of a library actor that uses a PortParameter is the Sinewave actor, which is found in the `Sources→SequenceSources` library in Vergil. It is shown in Figure 13.3. If you double click on this actor, you can set the default values for *frequency* and *phase*. But both of these values can also be set by the corresponding ports, which are shown with grey fill.

[†] The exceptions are parameters that are strictly string parameters, in which case the value of the parameter is the literal string, not the string interpreted as an expression, as for example the function parameter of the TrigFunction actor, which can take on only "sin," "cos," "tan", "asin", "acos", and "atan" as values.

13.2.3 String Parameters

Some parameters have values that are always strings of characters. Such parameters support a simple string substitution mechanism where the value of the string can reference other parameters in scope by name using the syntax $*name* or ${*name*} where *name* is the name of the parameter in scope. For example, the StringCompare actor in Figure 13.4 has as the value of *firstString* "The answer is $PI". This references the built-in constant PI. The value of *secondString* is "The answer is 3.1415926535898". As shown in the figure, these two strings are deemed to be equal because $PI is replaced with the value of PI.

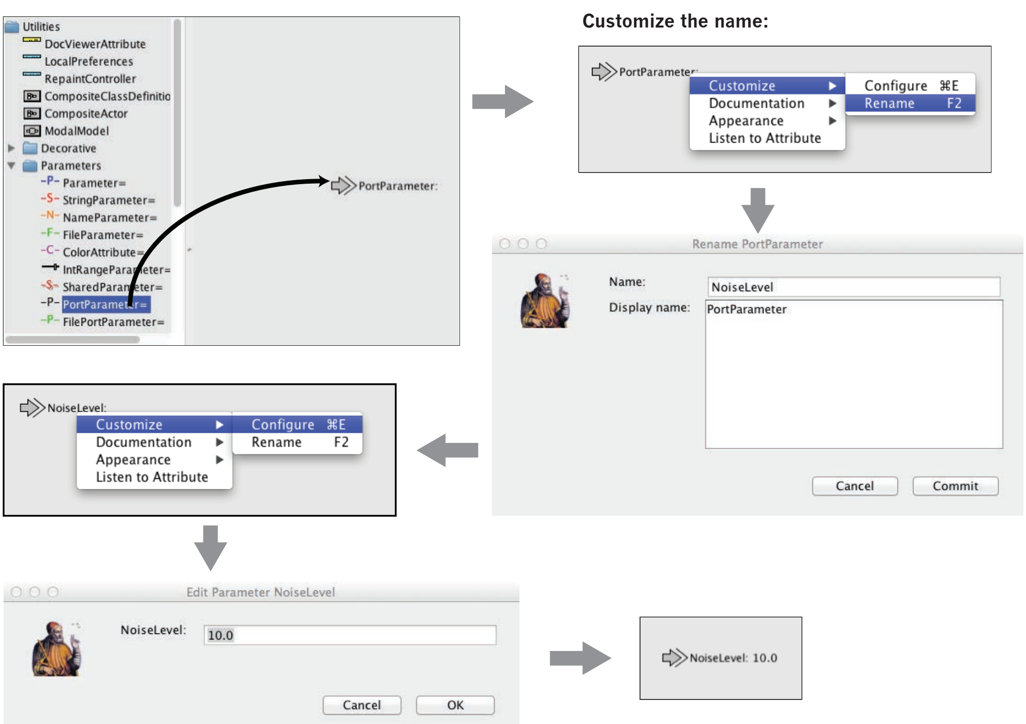


Figure 13.2: A *PortParameter* is both a port and a parameter. To use it in a composite actor, drag it into the actor, change its name to something meaningful and set its default value.

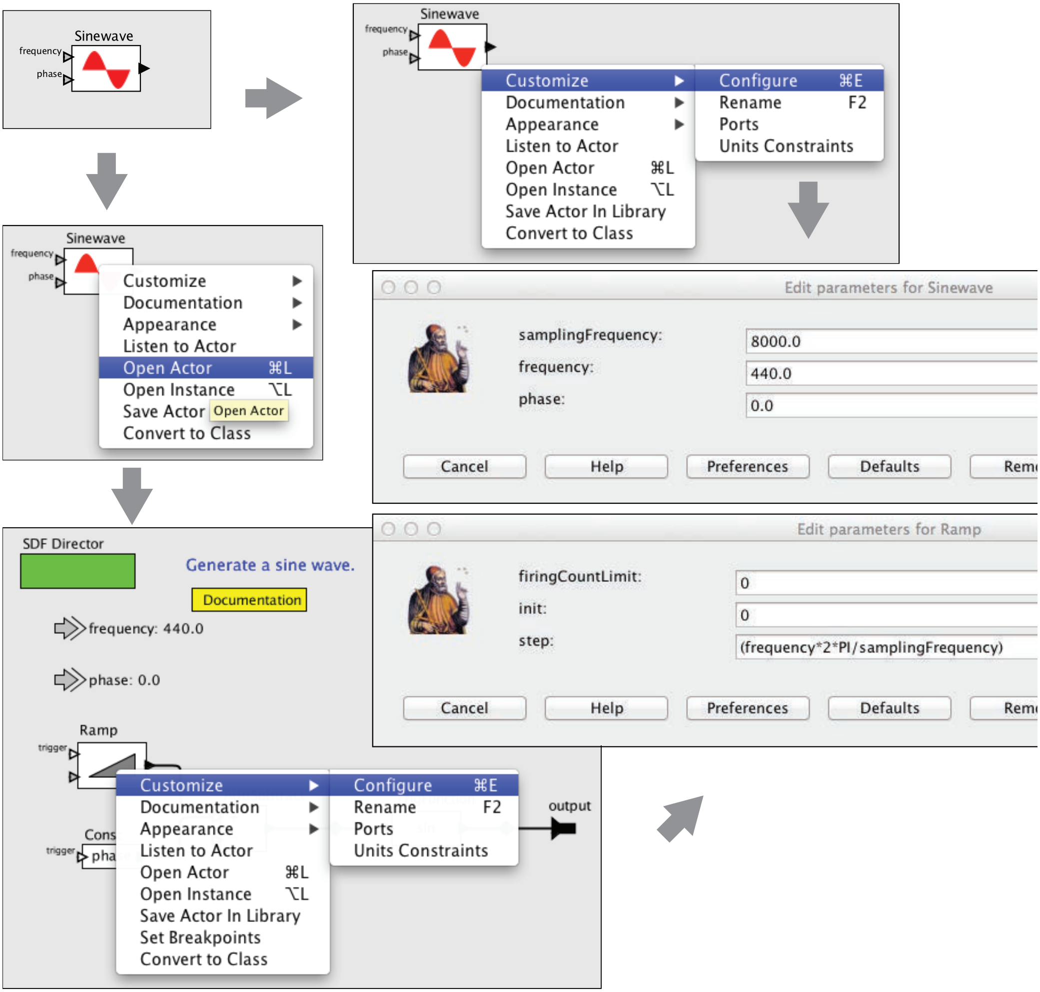


Figure 13.3: Sinewave actor, showing its port parameters, and their use at the lower level of hierarchy.

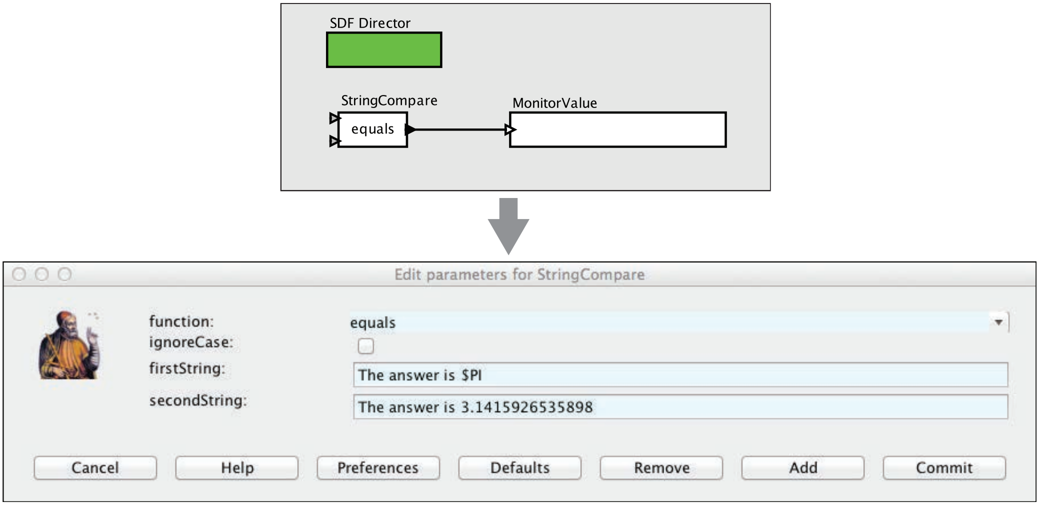


Figure 13.4: String parameters are indicated in the parameter editor boxes by a light blue background. A string parameter can include references to variables in scope with $*name*, where *name* is the name of the variable. In this example, the built-in constant $PI is referenced by name in the first parameter.

13.2.4 Expression Actor

The **Expression** actor is a particularly useful actor found in the `Math`. By default, it has one output and no inputs, as shown in Figure 13.5(a). The first step in using it is to add ports, as shown in (b) and (c). Click on Add to add a port, and then type in a unique name for the port. You then specify an expression using the port names as variables, as shown in (d), resulting in the icon shown in (e).

13.2.5 State Machines

Expressions give the guards for state transitions, as well as the values used in actions that produce outputs and actions that set values of parameters in the refinements of destination states. This mechanism was explained in the previous chapter.

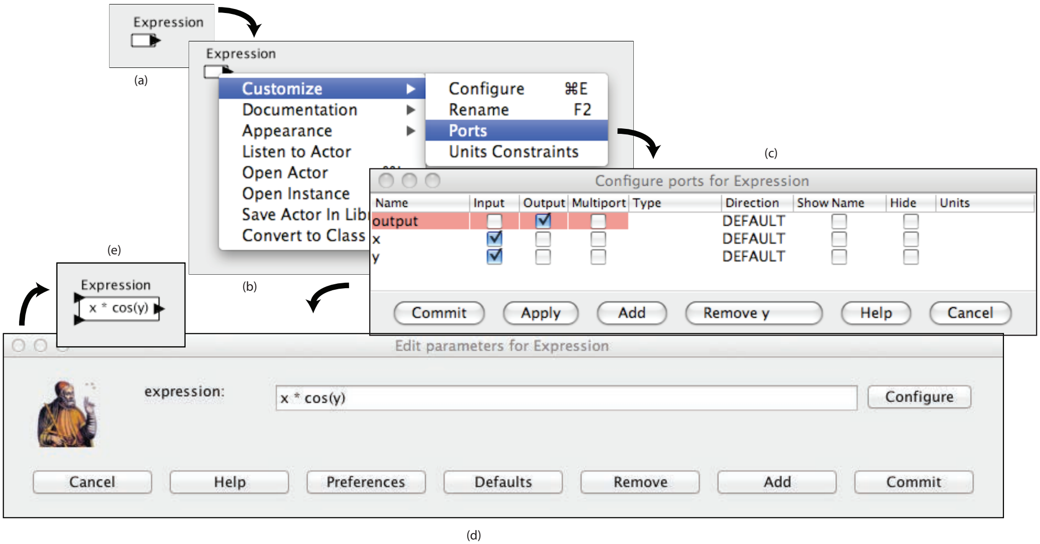


Figure 13.5: Illustration of the Expression actor.

13.3 Composite Data Types

A **composite data type** is a data type that aggregates some collection of other data types. The composite data types in the expression language include arrays, matrices, records, and union types.

13.3.1 Arrays

Arrays are specified with curly brackets, e.g., `{1, 2, 3}` is an array of *int*, while `{"x", "y", "z"}` is an array of type *string*. The types are denoted `arrayType(int,3)` and `arrayType(string,3)` respectively. An array is an ordered list of tokens of any type, with the primary constraint being that the elements all have the same type. If an array is given with mixed types, the expression evaluator will attempt to losslessly convert the elements to a common type. Thus, for example,

```
{1, 2.3}
```

has value

```
{1.0, 2.3}
```

Its type is `arrayType(double,2)`. The common type might be `arrayType(scalar)`, which is a union type (a type that can contain multiple distinct types). For example,

```
{1, 2.3, true}
```

has value

```
{1, 2.3, true}
```

The value is unchanged, but the type of the array is now `arrayType(scalar,3)`.

In Figure 13.5(c), the "Type" column may be used to specify the type of a port. Usually, it is not necessary to set the type, since the type inference mechanism will determine the type from the connections (see Chapter 14). Occasionally, however, it is necessary or helpful to force a port to have a particular type.

The Type column accepts expressions like `arrayType(int)`, which specifies an array with an unknown length. It is better, however, to specify an array length, if it is known. To do that, use an expression like `arrayType(int,`*n*`)`, where *n* is a positive integer that is the length of the array that is expected on the port.

The elements of the array can be given by expressions, as in the example `{2*pi, 3*pi}`. Arrays can be nested; for example, `{{1, 2}, {3, 4, 5}}` is an array of arrays of integers. The elements of an array can be accessed as follows:

```
>> {1.0, 2.3}(1)
2.3
```

Note that indexing begins at 0. Of course, if *name* is the name of a variable in scope whose value is an array, then its elements may be accessed similarly, as shown in this example:

```
>> x = {1.0, 2.3}
```

```
{1.0, 2.3}
>> x(0)
1.0
```

Arithmetic operations on arrays are carried out element-by-element, as shown by the following examples:

```
>> {1, 2}*{2, 2}
{2, 4}
>> {1, 2}+{2, 2}
{3, 4}
>> {1, 2}-{2, 2}
{-1, 0}
>> {1, 2}^2
{1, 4}
>> {1, 2}%{2, 2}
{1, 0}
```

Addition, subtraction, multiplication, division, and modulo of arrays by scalars is also supported, as in the following examples:

```
>> {1.0, 2.0} / 2.0
{0.5, 1.0}
>> 1.0 / {2.0, 4.0}
{0.5, 0.25}
>> 3 * {2, 3}
{6, 9}
>> 12 / {3, 4}
{4, 3}
```

Arrays of length 1 are equivalent to scalars, as illustrated below:

```
>> {1.0, 2.0} / {2.0}
{0.5, 1.0}
>> {1.0} / {2.0, 4.0}
{0.5, 0.25}
>> {3} * {2, 3}
```

```
{6, 9}
>> {12} / {3, 4}
{4, 3}
```

A significant subtlety arises when using nested arrays. Note the following example:

```
>> {{1.0, 2.0}, {3.0, 1.0}} / {0.5, 2.0}
{{2.0, 4.0}, {1.5, 0.5}}
```

In this example, the left argument of the divide is an array with two elements, and the right argument is also an array with two elements. The divide is thus element wise. However, each division is the division of an array by a scalar.

An array can be checked for equality with another array as follows:

```
>> {1, 2}=={2, 2}
false
>> {1, 2}!={2, 2}
true
```

For other comparisons of arrays, use the `compare` function (see Table 13.5). As with scalars, testing for equality using the == or != operators tests the values, independent of type. For example,

```
>> {1, 2}=={1.0, 2.0}
true
```

You can obtain the length of an array as follows,

```
>> {1, 2, 3}.length()
3
```

You can extract a subarray by invoking the `subarray` method as follows:

```
>> {1, 2, 3, 4}.subarray(2, 2)
{3, 4}
```

The first argument is the starting index of the subarray, and the second argument is the length.

You can also extract non-contiguous elements from an array using the `extract` method. This method has two forms. The first form takes a *boolean* array of the same length as the original array which indicates which elements to extract, as in the following example:

```
>> {"red", "green", "blue"}.extract({true,false,true})
{"red", "blue"}
```

The second form takes an array of integers giving the indices to extract, as in the following example:

```
>> {"red","green","blue"}.extract({2,0,1,1})
{"blue", "red", "green", "green"}
```

You can create an empty array with a specific element type using the function `emptyArray`. For example, to create an empty array of integers, use:

```
>> emptyArray(int)
{}
```

You can combine arrays into a single array using the `concatenate` function. For example,

```
>> concatenate({1, 2}, {3})
{1, 2, 3}
```

You can update an element of an array using the `update` function, for example,

```
>> {1, 2, 3}.update(0, 4)
{4, 2, 3}
```

The `update` function creates a new array[‡]

[‡]Actually, `update` creates a new token of type UpdatedArrayToken which keeps track of updated elements in a token while preserving the unchanged elements. The alternative would be to produce a new ArrayToken, which would result allocating memory and copying the entire source array.

13.3.2 Matrices

In Ptolemy II, arrays are ordered sets of tokens. Ptolemy II also supports **matrices**, which are more specialized than arrays. They contain only certain primitive types, currently *boolean*, *complex*, *double*, *fixedpoint*, *int*, and *long*. Currently *float*, *short* and *unsignedByte* matrices are not supported. Matrices cannot contain arbitrary tokens, so they cannot, for example, contain matrices. They are intended for data intensive computations. Matrices are specified with square brackets, using commas to separate row elements and semicolons to separate rows. E.g., `[1, 2, 3; 4, 5, 5+1]` gives a two by three integer matrix (2 rows and 3 columns). Note that an array or matrix element can be given by an expression. A row vector can be given as `[1, 2, 3]` and a column vector as `[1; 2; 3]`. Some MATLAB-style array constructors are supported. For example, `[1:2:9]` gives an array of odd numbers from 1 to 9, and is equivalent to `[1, 3, 5, 7, 9]`. Similarly, `[1:2:9; 2:2:10]` is equivalent to `[1, 3, 5, 7, 9; 2, 4, 6, 8, 10]`. In the syntax `[p:q:r]`, *p* is the first element, *q* is the step between elements, and *r* is an upper bound on the last element. That is, the matrix will not contain an element larger than *r*. If a matrix with mixed types is specified, then the elements will be converted to a common type, if possible. Thus, for example, `[1.0, 1]` is equivalent to `[1.0, 1.0]`, but `[1.0, 1L]` is illegal (because there is no common type to which both elements can be converted losslessly, see Chapter 14).

Elements of matrices are referenced using *`matrixname`*`(`*`n`*`, `*`m`*`)`, where *`matrixname`* is the name of a matrix variable in scope, *`n`* is the row index, and *`m`* is the column index. Index numbers start with zero, as in Java, not 1, as in MATLAB. For example,

```
>> [1, 2; 3, 4](0,0)
1
>> a = [1, 2; 3, 4]
[1, 2; 3, 4]
>> a(1,1)
4
```

Matrix multiplication works as expected. For example,

```
>> [1, 2; 3, 4]*[2, 2; 2, 2]
[6, 6; 14, 14]
```

Of course, if the dimensions of the matrix don't match, then you will get an error message. To do element wise multiplication, use the `multipyElements` function (see Table 13.8). Matrix addition and subtraction are element wise, as expected, but the division operator is not supported. Element wise division can be accomplished with the `divideElements` function, and multiplication by a matrix inverse can be accomplished using the `inverse` function (see Table 13.8). A matrix can be raised to an *int*, *short* or *unsignedByte* power, which is equivalent to multiplying it by itself some number of times. For instance,

```
>> [3, 0; 0, 3]^3
[27, 0; 0, 27]
```

A matrix can also be multiplied or divided by a scalar, as follows:

```
>> [3, 0; 0, 3]*3
[9, 0; 0, 9]
```

A matrix can be added to a scalar. It can also be subtracted from a scalar, or have a scalar subtracted from it. For instance,

```
>> 1-[3, 0; 0, 3]
[-2, 1; 1, -2]
```

A matrix can be checked for equality with another matrix as follows:

```
>> [3, 0; 0, 3]!=[3, 0; 0, 6]
true
>> [3, 0; 0, 3]==[3, 0; 0, 3]
true
```

For other comparisons of matrices, use the `compare` function (see Table 13.7). As with scalars, testing for equality using the == or `!=` operators tests the values, independent of type. For example,

```
>> [1, 2]==[1.0, 2.0]
true
```

To get type-specific equality tests, use the `equals` method, as in the following examples:

```
>> [1, 2].equals([1.0, 2.0])
false
>> [1.0, 2.0].equals([1.0, 2.0])
true
```

13.3.3 Records

A **record** token is a composite type containing named fields, where each field has a value. The value of each field can have a distinct type. Records are delimited by curly braces, with each field given a name. For example, `{a=1, b="foo"}` is a record with two fields, named "a" and "b", with values 1 (an integer) and "foo" (a string), respectively. The key of a field can be an arbitrary string, provided that it is quoted. Only strings that qualify as valid Java identifiers can be used without quotation marks. Note that quotation marks within a quoted string must be escaped using a backslash. The value of a field can be an arbitrary expression, and records can be nested (a field of a record token may be a record token).

An **ordered record** is similar to a normal record except that it preserves the original ordering of the labels. Ordered records are delimited using square brackets rather than curly braces. For example, `[b="foo", a=1]` is an ordered record token in which "b" will remain the first label.

Fields that are valid Java identifiers may be accessed using the period operator, optionally with braces—as if it were a method call. For example, the following two expressions:

```
{a=1,b=2}.a
{a=1,b=2}.a()
```

both yield 1.

An alternative syntax to access fields uses the `get()` method. Note that this is the only way to access fields for which the key demands the use of quotation marks. For example:

```
{" a"=1, "\"b"=2}.get("\"b")
```

yields 2.

The arithmetic operators `+`, `-`, `*`, `/`, and `%` can be applied to records. If the records do not have identical fields, then the operator is applied only to the fields that match, and the result contains only the fields that match. Thus, for example,

```
{foodCost=40, hotelCost=100}
   + {foodCost=20, taxiCost=20}
```

yields the result

```
{foodCost=60}
```

You can think of an operation as a set intersection, where the operation specifies how to merge the values of the intersecting fields. You can also form an intersection without applying an operation. In this case, using the `intersect` function, you form a record that has only the common fields of two specified records, with the values taken from the first record. For example,

```
>> intersect({a=1, c=2}, {a=3, b=4})
{a=1}
```

Records can be joined (think of a set union) without any operation being applied by using the `merge` function. This function takes two arguments, both of which are record tokens. If the two record tokens have common fields, then the field value from the first record is used. For example,

```
merge({a=1, b=2}, {a=3, c=3})
```

yields the result {`a=1, b=2, c=3`}.

Records can be compared, as in the following examples:

```
>> {a=1, b=2}!={a=1, b=2}
false
>> {a=1, b=2}!={a=1, c=2}
true
```

Note that two records are equal only if they have the same field labels and the values match. As with scalars, the values match irrespective of type. For example:

```
>> {a=1, b=2}=={a=1.0, b=2.0+0.0i}
true
```

The order of the fields is irrelevant for normal (unordered) records. Hence

```
>> {a=1, b=2}=={b=2, a=1}
true
```

Moreover, normal record fields are reported in alphabetical order, irrespective of the order in which they are defined. For example,

```
>> {b=2, a=1}
{a=1, b=2}
```

Equality comparisons for ordered records respect the original order of the fields. For example,

```
>> [a=1, b=2]==[b=2, a=1]
false
```

Additionally, ordered record fields are always reported in the order in which they are defined. For example,

```
>> [b=2, a=1]
[b=2, a=1]
```

To get type-specific equality tests, use the `equals` method, as in the following examples:

```
>> {a=1, b=2}.equals({a=1.0, b=2.0+0.0i})
false
>> {a=1, b=2}.equals({b=2, a=1})
true
```

Finally, You can create an empty record using the `emptyRecord` function:

```
>> emptyRecord()
{}
```

13.3.4 Union Types

Occasionally, more than one distinct data type will be sent over the same connection, or a variable may take on values with one of several data types. Ptolemy II provides a **union** type to accommodate this. A union type is designated as in the following example:

```
{|a = int, b = complex |}
```

This indicates a port or variable that may have type *int* or *complex*. A typical use of union types uses a **UnionMerge** and/or **UnionDisassembler** actor.

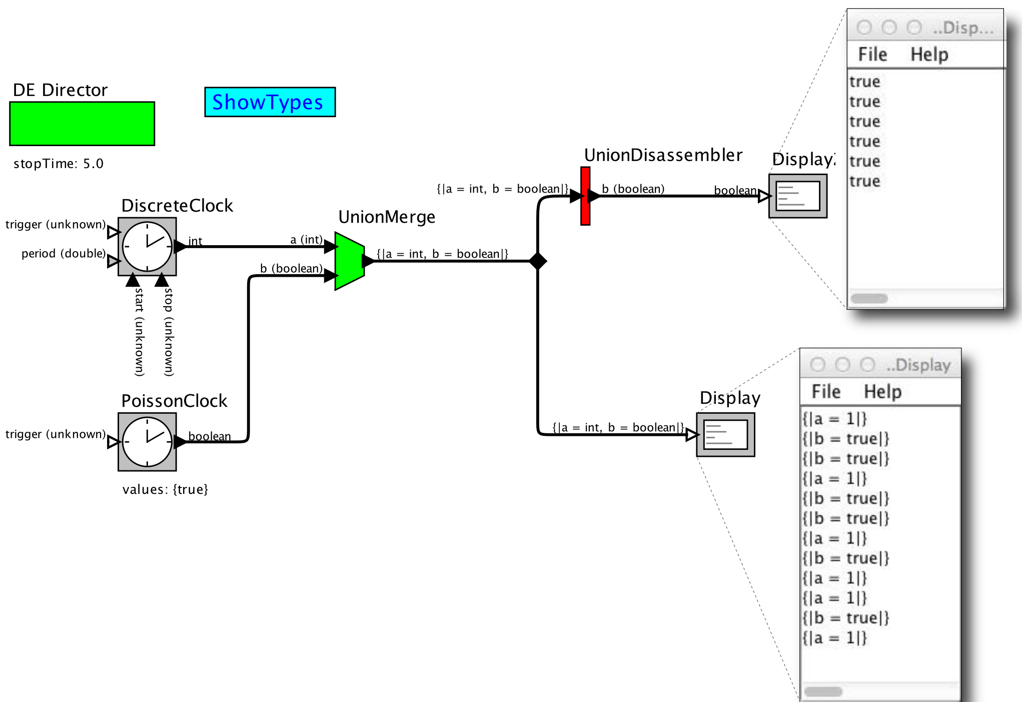


Figure 13.6: Union types allow types to resolve to more than one type. [online]

Example 13.1: Consider the example shown in Figure 13.6. This is a DE model with two data sources, a DiscreteClock, which produces outputs of type *int*, and a PoissonClock, which produces outputs of type *boolean*. These two streams of values are merged by the UnionMerge, whose output type becomes a union type. The names of the types in the union are determined by the names of the input ports of the UnionMerge, which are added when building the model. The lower Display actor displays the merged stream, showing that each token displayed has a single value, either an *int* or a *boolean*.

Along the upper path, a UnionDisassembler actor is used to extract from the stream the "b" types, which are *boolean*. Only those types are passed to the output, and the type of the output is inferred to be *boolean*.

13.4 Operations on Tokens

Every element and subexpression in an expression represents an instance of the Token class in Ptolemy II (or more likely, a class derived from Token). The expression language supports a number of operations on tokens that give access to the underlying Java code.

13.4.1 Invoking Methods

The expression language supports invocation of any method of a given token, as long as the arguments of the method are of type Token and the return type is Token (or a class derived from Token, or something that the expression parser can easily convert to a token, such as a string, *double*, *int*, etc.). The syntax for this is `(token.methodName(args)`, where `methodName` is the name of the method and `args` is a comma-separated set of arguments. Each argument can itself be an expression. Note that the parentheses around the `token` are not required, but might be useful for clarity. As an example, the ArrayToken and RecordToken classes have a `length` method, illustrated by the following examples:

```
{1, 2, 3}.length()
{a=1, b=2, c=3}.length()
```

each of which returns the integer 3.

The MatrixToken classes have three particularly useful methods, illustrated in the following examples:

```
[1, 2; 3, 4; 5, 6].getRowCount()
```

which returns 3, and

```
[1, 2; 3, 4; 5, 6].getColumnCount()
```

which returns 2, and

```
[1, 2; 3, 4; 5, 6].toArray()
```

which returns 1, 2, 3, 4, 5, 6. The latter function can be particularly useful for creating arrays using MATLAB-style syntax. For example, to obtain an array with the integers from 1 to 100, you can enter:

```
[1:1:100].toArray()
```

13.4.2 Accessing Model Elements

Expressions in a model can reference elements of the model and invoke methods on them. The expression giving a parameter its value may reference by name any object that is contained by the container of the parameter.

Example 13.2: Figure 13.7 shows a model with four parameters *P1* through *P4*. The parameter *P1* has expression

```
Const2.value
```

The `Const2` here refers to the actor named "Const2" that is contained by the container of *P1*, and hence `Const2.value` refers to the *value* parameter of the actor Const2. The value of the parameter *P1* is therefore equal to the value of the *value* parameter of Const2.

The keyword `this` in a parameter expression refers to the object that contains the parameter.

Example 13.3: In Figure 13.7, Const2 has a *value* parameter with the expression (shown in its icon):

```
this.getName() + ": " + P2
```

Here, `this` refers to Const2, so `this.getName()` returns a string that is the name "Const2." The rest of the expression performs string concatenation, appending a colon and the value parameter *P2*.

The parameter *P2* has expression

```
this.entityList().size()
```

In this case, `this` refers to the container of *P2*, which is the top-level model. Hence, `this.entityList()` returns a list of entities (actors) contained by this top-level model. And finally, `this.entityList().size()` returns the number of actors contained by this top-level model, which is 5.

Now, the first two outputs of this model should be easy to understand:

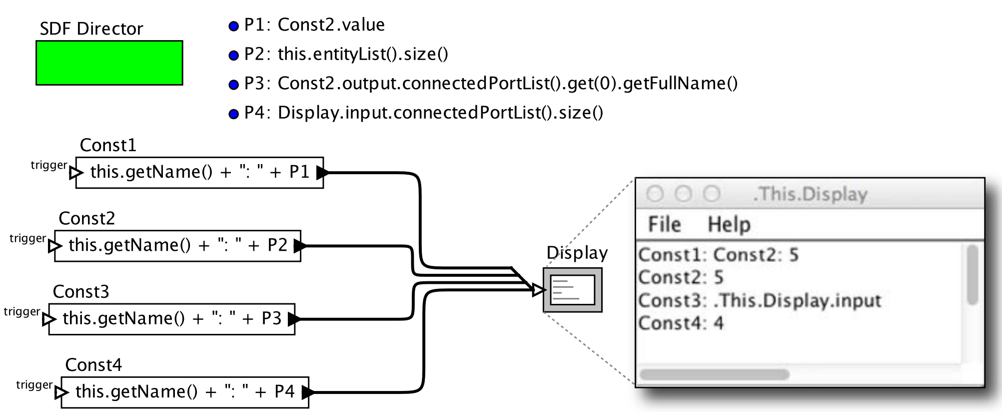


Figure 13.7: Expressions can access elements of the model, as shown here. [online]

```
Const1: Const2: 5
Const2: 5
```

The second output is the value of *P2* (namely 5) prepended by the name of the Const actor generating the output and a colon. The first output is the value *P1* (which is the string `"Const2: 5"`) prepended by the name of the Const actor generating the output and a colon.

Parameters can use method invocation to traverse the connections in the model, as illustrated next.

Example 13.4: In Figure 13.7, parameter *P3* has the expression:

```
Const2.output.connectedPortList().get(0)
         .getFullName()
```

This gets the full name of the first port in the list of ports that the *output* port of actor Const2 is connected to. Specifically, it returns the string `".This.Display.input"`, as displayed by actor Const3.

Similarly, *P4* has expression

```
Display.input.connectedPortList().size()
```

which returns the number of sources connected to the Display input.

For an overview of some of the methods that can be invoked on actors, parameters, and ports, see Chapter 12. For a complete listing, see the code documentation for Ptolemy II.

13.4.3 Casting

The cast function can be used to explicitly cast a value into a type. When the cast function is invoked with `cast(type, value)`, where `type` is the target type and `value` is the value to be cast, a new value is returned (if a predefined casting is applicable) that is in the specified type. For example, `cast(long, 1)` yields 1L, which is equal to 1 but is in the long data type, and `cast(string, 1)` yields `"1"`, which is in the string data type.

13.4.4 Defining Functions

Users can define new functions in the expression language. The syntax is:

```
function(arg1:Type, arg2:Type...)
   function body
```

where **function** is the keyword for defining a function. The type of an argument can be left unspecified, in which case the expression language will attempt to infer it. The function body gives an expression that defines the return value of the function. The return type is always inferred based on the argument type and the expression. For example:

```
function(x:double) x*5.0
```

defines a function that takes a *double* argument, multiplies it by 5.0, and returns a *double*. The return value of the above expression is the function itself. Thus, for example, the expression evaluator yields:

```
>> function(x:double) x*5.0
(function(x:double) (x*5.0))
```

To apply the function to an argument, simply do

```
>> (function(x:double) x*5.0) (10.0)
50.0
```

Alternatively, in the expression evaluator, you can assign the function to a variable, and then use the variable name to apply the function. For example,

```
>> f = function(x:double) x*5.0
(function(x:double) (x*5.0))
>> f(10)
50.0
```

13.4.5 Higher-Order Functions

Functions can be passed as arguments to certain **higher-order functions** that have been defined (see Table 13.15). For example, the `iterate` function takes three arguments, a function, an integer, and an initial value to which to apply the function. It applies the function first to the initial value, then to the result of the application, then to that result, collecting the results into an array whose length is given by the second argument. For example, to get an array whose values are multiples of 3, try

```
>> iterate(function(x:int) x+3, 5, 0)
{0, 3, 6, 9, 12}
```

The function given as an argument simply adds three to its argument. The result is the specified initial value (0) followed by the result of applying the function once to that initial value, then twice, then three times, etc.

Another useful higher-order function is the `map` function. This one takes a function and an array as arguments, and simply applies the function to each element of the array to construct a result array. For example,

```
>> map(function(x:int) x+3, {0, 2, 3})
{3, 5, 6}
```

Ptolemy II also supports a fold function, which can be used to program a loop in an expression. The fold function applies a function to each element of an array, accumulating a result as it goes. The function that is folded over the array takes two arguments, the accumulated result so far and an array element. When the function to be folded is applied to the first element of the array, the accumulated result is an initial value.

Example 13.5:

```
fold(
   function(x:int, e:int) x + 1,
   0, {1, 2, 3}
)
```

This computes the length of array $\{1, 2, 3\}$. The result is 3, which is equal to $\{1, 2, 3\}$.length(). Specifically, the function to be folded is `function(x:int,` `e:int) x + 1`. Given arguments `x` and `e`, it returns `x + 1`, ignoring the second argument `e`. It is first applied to the initial value, `0`, and the first element of the array, `1`, yielding `1`. It is then applied to the accumulated result, `1`, and the second element of the array, the value of which it ignores, yielding `2`. It is invoked the number of times equal to the number of elements in array $\{1, 2, 3\}$. Therefore, `x` is increased 3 times from the starting value 0.

Example 13.6: The following variant does not ignore the values of the array elements:

```
fold(
   function(x:int, e:int) x + e,
   0, {1, 2, 3}
)
```

This computes the sum of all elements in array $\{1, 2, 3\}$, yielding `6`.

Example 13.7:

```
fold(
   function(x:arrayType(int), e:int)
      e % 2 == 0 ?  x :  x.append({e}),
   {}, {1, 2, 3, 4, 5}
)
```

This computes a subarray of array $\{1, 2, 3, 4, 5\}$ that contains only odd numbers. The result is $\{1, 3, 5\}$.

Example 13.8: Let C be an actor.

```
fold(
   function(list:arrayType(string),
      port:object("ptolemy.kernel.Port"))
```

```
        port.connectedPortList().isEmpty() ?
            list.append({port}) :  list,
    {}, C.portList()
)
```

This returns a list of `C`'s ports that are not connected to any other port (with `connectedPortList()` being empty). Each port in the returned list is encapsulated in an ObjectToken.

13.4.6 Using Functions in a Model

A typical use of functions in a Ptolemy II model is to define a parameter in a model whose value is a function. Suppose that the parameter named `f` has value

```
function(x:double)x*5.0
```

Then within the scope of that parameter, the expression `f(10.0)` will yield result 50.0.

Functions can also be passed along connections in a Ptolemy II model.

Example 13.9: Consider the model shown in Figure 13.8. In that example, the Const actor defines a function that simply squares the argument. Its output, therefore, is a token with type function. That token is fed to the "f" input of the Expression actor. The expression uses this function by applying it to the token provided on the "y" input. That token, in turn, is supplied by the Ramp actor, so the result is the curve shown in the plot on the right.

Example 13.10: A more elaborate use is shown in Figure 13.9 In that example, the Const actor produces a function, which is then used by the Expression actor to create new function, which is then used by Expression2 to perform a calculation. The calculation performed here multiplies the output of the Ramp to the square of the output of the Ramp, thus computing the cube.

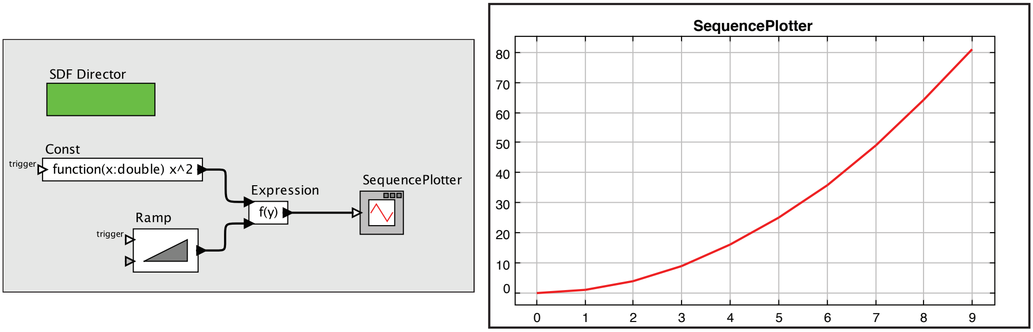


Figure 13.8: Example of a function being passed from one actor to another.

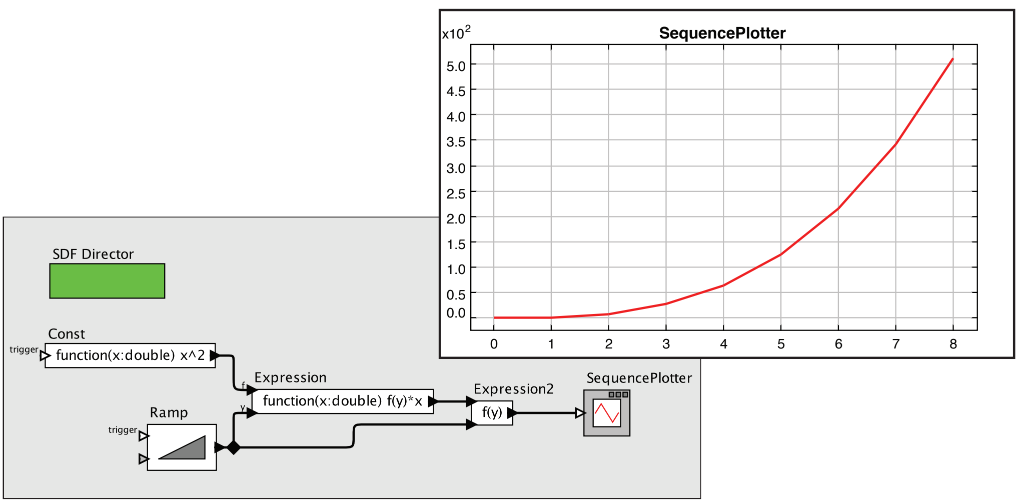


Figure 13.9: More elaborate example with functions passed between actors.

13.4.7 Recursive Functions

Functions can be recursive, as illustrated by the following (rather arcane) example:

```
>> fact = function(x:int,f:(function(x,f) int)) (x<1?1:x*f(x-1,f))
(function(x:int, f:function(a0:general, a1:general) int)
       (x<1)?1:(x*f((x-1), f)))
>> factorial = function(x:int) fact(x,fact)
(function(x:int) (function(x:int, f:function(a0:general, a1:general) int)
 (x<1)?1:(x*f((x-1), f)))(x, (function(x:int,
 f:function(a0:general, a1:general) int) (x<1)?1:(x*f((x-1), f)))))
>> map(factorial, [1:1:5].toArray())
{1, 2, 6, 24, 120}
```

The first expression defines a function named "fact" that takes a function as an argument, and if the argument is greater than or equal to 1, uses that function recursively. The second expression defines a new function "factorial" using "fact." The final command applies the factorial function to an array to compute factorials.

13.4.8 Built-In Functions

The expression language includes a set of functions, such as `sin`, `cos`, etc. The functions that are built in include all static methods [§] of the classes shown in Table 13.2, which together provide a rich set [¶].

The functions currently available are shown in the tables at the end of this chapter, which also show the argument types and return types. The argument and return types are the widest type that can be used. For example, `acos` will take any argument that can be losslessly cast to a *double*, such as unsigned byte, short, integer, float. *long* cannot be cast losslessly cast to *double*, so acos(1L) will fail. **Trigonometric functions** are given in Table 13.4. Basic **mathematical functions** are given in Tables 13.5 and 13.6. Functions that take or return matrices, arrays, or records are given in Tables 13.7 through 13.9. Utility functions for evaluating expressions are given in Table 13.10. Functions performing signal processing operations are given in Tables 13.11 through 13.13. I/O and other miscellaneous functions are given in Tables 13.15 and 13.16.

[§] Note that calling methods such as String.format() that have an argument of Object [] can be difficult because of problems specifying an array of Java Objects such as java.lang.Double instead of ptolemy.data.type.DoubleToken

[¶] Moreover, the set of available methods can easily be extended if you are writing Java code by registering another class that includes static methods (see the PtParser class in the ptolemy.data.expr package).

In most cases, a function that operates on scalar arguments can also operate on arrays and matrices. Thus, for example, you can fill a row vector with a sine wave using an expression like

```
sin([0.0:PI/100:1.0])
```

Or you can construct an array as follows,

```
sin({0.0, 0.1, 0.2, 0.3})
```

Functions that operate on type *double* will also generally operate on *int*, *short*, or *unsignedByte*, because these can be losslessly converted to *double*, but not generally on *long* or *complex*. Tables of available functions are shown in the appendix. For example, Table 13.4 shows trigonometric functions. Note that these operate on *double* or *complex*, and hence on *int*, *short* and *unsignedByte*, which can be losslessly converted to *double*. The result will always be *double*. For example,

```
>> cos(0)
1.0
```

Table 13.2: The classes whose static methods are available as functions in the expression language.

java.lang.Math	ptolemy.math.IntegerMatrixMath
java.lang.Double	ptolemy.math.DoubleMatrixMath
java.lang.Integer	ptolemy.math.ComplexMatrixMath
java.lang.Long	ptolemy.math.LongMatrixMath
java.lang.String	ptolemy.math.IntegerArrayMath
ptolemy.data.MatrixToken.	ptolemy.math.DoubleArrayStat
ptolemy.data.RecordToken.	ptolemy.math.ComplexArrayMath
ptolemy.data.expr.UtilityFunctions	ptolemy.math.LongArrayMath
ptolemy.data.expr.FixPointFunctions	ptolemy.math.SignalProcessing
ptolemy.math.Complex	ptolemy.math.FixPoint
ptolemy.math.ExtendedMath	ptolemy.data.ObjectToken

These functions will also operate on matrices and arrays, in addition to the scalar types shown in the table, as illustrated above. The result will be a matrix or array of the same size as the argument, but always containing elements of type *double*.

Table 13.7 shows other arithmetic functions beyond the trigonometric functions. As with the trigonometric functions, those that indicate that they operate on *double* will also work on *int*, *short* and *unsignedByte*, and unless they indicate otherwise, they will return whatever they return when the argument is *double*. Those functions in the table that take scalar arguments will also operate on matrices and arrays. For example, since the table indicates that the `max` function can take *int*, *int* as arguments, then by implication, it can also take *int*, *int*. For example,

```
>> max({1, 2}, {2, 1})
{2, 2}
```

Notice that the table also indicates that `max` can take *int* as an argument. E.g.

```
>> max({1, 2, 3})
3
```

In the former case, the function is applied pointwise to the two arguments. In the latter case, the returned value is the maximum over all the contents of the single argument.

Table 13.7 shows functions that only work with matrices, arrays, or records (that is, there is no corresponding scalar operation). Recall that most functions that operate on scalars will also operate on arrays and matrices. Table 13.10 shows utility functions for evaluating expressions given as strings or representing numbers as strings. Of these, the `eval` function is the most flexible.

A few of the functions have sufficiently subtle properties that they require further explanation. That explanation is here.

eval() and traceEvaluation()

The built-in function `eval` will evaluate a string as an expression in the expression language. For example,

```
eval("[1.0, 2.0; 3.0, 4.0]")
```

will return a matrix of *doubles*. The following combination can be used to read parameters from a file:

```
eval(readFile("filename"))
```

where the filename can be relative to the current working directory (where Ptolemy II was started, as reported by the Java Virtual Machine property `user.dir`), a user's home directory (as reported by the property `user.home`), or the classpath, which includes the directory tree in which Ptolemy II is installed. Note that if `eval` is used in an Expression actor, then it will be impossible for the type system to infer any more specific output type than general. If you need the output type to be more specific, then you will need to cast the result of `eval`. For example, to force it to type *double*:

```
>> cast(double, eval("pi/2"))
1.5707963267949
```

The `traceEvaluation` function evaluates an expression given as a string, much like `eval`, but instead of reporting the result, reports exactly how the expression was evaluated. This can be used to debug expressions, particularly when the expression language is extended by users.

random(), gaussian()

The `random` and `gaussian` functions shown in Table 13.5 and Table 13.6 return one or more random numbers. With the minimum number of arguments (zero or two, respectively), they return a single number. With one additional argument, they return an array of the specified length. With a second additional argument, they return a matrix with the specified number of rows and columns.

There is a key subtlety when using these functions in Ptolemy II. In particular, they are evaluated only when the expression within which they appear is evaluated. The result of the expression may be used repeatedly without re-evaluating the expression. Thus, for example, if the value parameter of the Const actor is set to `random()`, then its output will be a random constant, i.e., it will not change on each firing. The output will change, however, on successive runs of the model. In contrast, if this is used in an Expression actor, then each firing triggers an evaluation of the expression, and consequently will result in a new random number.

property()

The `property` function accesses Java Virtual Machine system properties by name. Some possibly useful system properties are:

- ptolemy.ptII.dir: The directory in which Ptolemy II is installed.
- ptolemy.ptII.dirAsURL: The directory in which Ptolemy II is installed, but represented as a URL.
- user.dir: The current working directory, which is usually the directory in which the current executable was started.

For a complete list of Java Virtual Machine properties, see the Java documentation for `java.lang.System.getProperties`.

remainder()

This function computes the remainder operation on two arguments as prescribed by the IEEE 754 standard, which is not the same as the modulo operation computed by the % operator. The result of `remainder(x, y)` is $x - yn$, where n is the integer closest to the exact value of x/y. If two integers are equally close, then n is the integer that is even. This yields results that may be surprising, as indicated by the following examples:

```
>> remainder(1,2)
1.0
>> remainder(3,2)
-1.0
```

Compare this to

```
>> 3%2
1
```

which is different in two ways. The result numerically different and is of type *int*, whereas `remainder` always yields a result of type *double*. The `remainder()` function is implemented by the java.lang.Math class, which calls it `IEEEremainder()`. The documentation for that class gives the following special cases:

- If either argument is NaN, or the first argument is infinite, or the second argument is positive zero or negative zero, then the result is NaN.
- If the first argument is finite and the second argument is infinite, then the result is the same as the first argument.

DCT() and IDCT()

The discrete cosine transform (DCT) function can take one, two, or three arguments. In all three cases, the first argument is an array of length $N > 0$ and the DCT returns an

$$X_k = s_k \sum_{n=0}^{N-1} x_n \cos((2n+1)k\frac{\pi}{2D}) \tag{13.1}$$

for k from 0 to $D-1$, where N is the size of the specified array and D is the size of the DCT. If only one argument is given, then D is set to equal the next power of two larger than N. If a second argument is given, then its value is the order of the DCT, and the size of the DCT is 2^{order}. If a third argument is given, then it specifies the scaling factors s_k according to the following table:

Table 13.3: Normalization options for the DCT function.

Name	Third argument	Normalization
Normalized	0	$s_k = \begin{cases} \frac{1}{\sqrt{2}}; & k=0 \\ 1, & otherwise \end{cases}$
Unnormalized	1	$s_k = 1$
Orthonormal	2	$s_k = \begin{cases} \frac{1}{\sqrt{D}}; & k=0 \\ \sqrt{\frac{2}{D}}; & otherwise \end{cases}$

The default, if a third argument is not given, is “Normalized.” The IDCT function is similar, and can also take one, two, or three arguments. The formula in this case is

$$x_n = \sum_{k=0}^{N-1} s_k X_k \cos((2n+1)k\frac{\pi}{2D}). \tag{13.2}$$

13.5 Nil Tokens

Null or missing tokens are common in analytical systems like R and SAS where they are used to handle sparsely populated data sources. In database parlance, missing tokens are sometimes called null tokens. Since null is a Java keyword, we use the term "**nil**." Nil tokens are useful for analyzing real world data such as temperature where the value is not measured during every interval. In principle, one may want, for example, a TolerantAverage actor that does not require all data values to be present — when the TolerantAverage actor sees a nil token, it would ignore it. Note that this can lead to uncertainty. For example, if `average` is expecting 30 values and 29 of them are nil, then the average will not be very accurate.

In Ptolemy II, operations on tokens yield a nil token if any argument is a nil token. Thus, the Average actor is not like TolerantAverage. Upon receiving a nil token, all subsequent results will be nil. When an operation yields a nil value, the resulting nil token will have the same type that would normally have resulted from the operation, so type safety is preserved. Not all data types, however, support nil tokens. In particular, the various matrix types cannot have nil values because the underlying matrices are Java native type matrices that do not support nil.

The expression language defines a constant named `nil` that has value nil and type *niltype* (see Chapter 14). The `cast` expression language function can be used to generate nil values of other types. For example, "`cast(int, nil)`" will return a token with value nil and type *int*.

13.6 Fixed Point Numbers

Ptolemy II includes a fixed point data type. We represent a fixed point value in the expression language using the following format:

```
fix(value, totalBits, integerBits)
```

Thus, a fixed point value of 5.375 that uses 8 bit precision of which 4 bits are used to represent the (signed) integer part can be represented as:

```
fix(5.375, 8, 4)
```

The value can also be a matrix of *doubles*. The values are rounded, yielding the nearest value representable with the specified precision. If the value to represent is out of range, then it is saturated, meaning that the maximum or minimum fixed point value is returned, depending on the sign of the specified value. For example,

```
fix(5.375, 8, 3)
```

will yield 3.968758, the maximum value possible with the (8/3) precision.

In addition to the `fix` function, the expression language offers a `quantize` function. The arguments are the same as those of the `fix` function, but the return type is a DoubleToken or DoubleMatrixToken instead of a FixToken or FixMatrixToken. This function can therefore be used to quantize double-precision values without ever explicitly working with the fixed-point representation.

To make the FixToken accessible within the expression language, the following functions are available:

- To create a single FixPoint Token using the expression language:

  ```
  fix(5.34, 10, 4)
  ```

  This will create a FixToken. In this case, we try to fit the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.

- To create a Matrix with FixPoint values using the expression language:

  ```
  fix([ -.040609, -.001628, .17853 ], 10, 2)
  ```

  This will create a FixMatrixToken with 1 row and 3 columns, in which each element is a FixPoint value with precision(10/2). The resulting FixMatrixToken will try to fit each element of the given *double* matrix into a 10 bit representation with 2 bits used for the integer part. By default the round quantizer is used.

- To create a single DoubleToken, which is the quantized version of the *double* value given, using the expression language:

  ```
  quantize(5.34, 10, 4)
  ```

This will create a DoubleToken. The resulting DoubleToken contains the *double* value obtained by fitting the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.

- To create a Matrix with *doubles* quantized to a particular precision using the expression language:

```
quantize([ -.040609, -.001628, .17853 ], 10, 2)
```

This will create a DoubleMatrixToken with 1 row and 3 columns. The elements of the token are obtained by fitting the given matrix elements into a 10 bit representation with 2 bits used for the integer part. Instead of being a fixed point value, the values are converted back to their *double* representation and by default the round quantizer is used.

13.7 Units

Ptolemy II supports **units systems**, which are built on top of the expression language. Units systems allow parameter values to be expressed with units, such as "1.0 * cm", which is equal to "0.01 * meters". These are expressed this way (with the * for multiplication) because "cm" and "meters" are actually variables that become in scope when a units system icon is dragged in to a model. A few simple units systems are provided (mainly as examples) in the utilities library.

A model using one of the simple provided units systems is shown in Figure 13.10 This unit system is called *BasicUnits*; the units it defines can be examined by double clicking on its icon, or by invoking "Customize" |"Configure", as shown in Figure 13.11. In that figure, we see that "meters", "meter", and "m" are defined, and are all synonymous. Moreover, "cm" is defined, and given value "0.01*meters", and "in", "inch" and "inches" are defined, all with value "2.54*cm".

In the example in Figure 13.10, a constant with value "1.0 * meter" is fed into a Scale actor with scale factor equal to "2.0/ms". This produces a result with dimensions of length over time. If we feed this result directly into a Display actor, then it is displayed as "2000.0 meters/seconds", as shown in Figure 13.12, top display. The canonical units for length are meters, and for time are seconds.

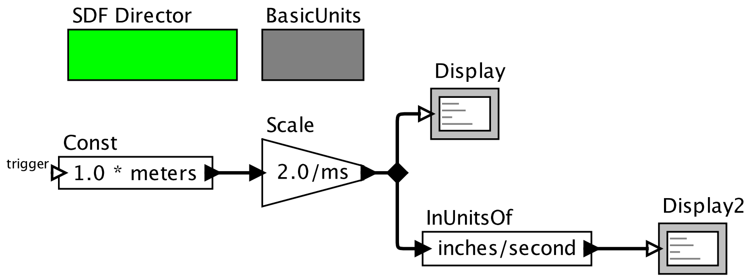


Figure 13.10: Example of a model that includes a unit system. [online]

In Figure 13.10, we also take the result and feed it to the InUnitsOf actor, which divides its input by its argument, and checks to make sure that the result is unitless. This tells us that 2 meters/ms is equal to about 78,740 inches/second.

The InUnitsOf actor can be used to ensure that numbers are interpreted correctly in a model, which can be effective in catching certain kinds of critical errors. For example, if in Figure 13.10, we had entered "seconds/inch" instead of "inches/second" in the InUnitsOf actor, we would have gotten the exception in Figure 13.13 instead of the execution in Figure 13.12.

Units systems are built entirely on the expression language infrastructure in Ptolemy II. The units system icons actually represent instances of scope-extending attributes, which are attributes whose parameters are in scope as if those parameters were directly contained by the container of the scope extending attribute. That is, scope-extending attributes can define a collection of variables and constants that can be manipulated as a unit. Two fairly extensive units systems are provided, CGSUnitBase and ElectronicUnitBase. Nonetheless, these are intended as examples only, and can no doubt be significantly improved and extended.

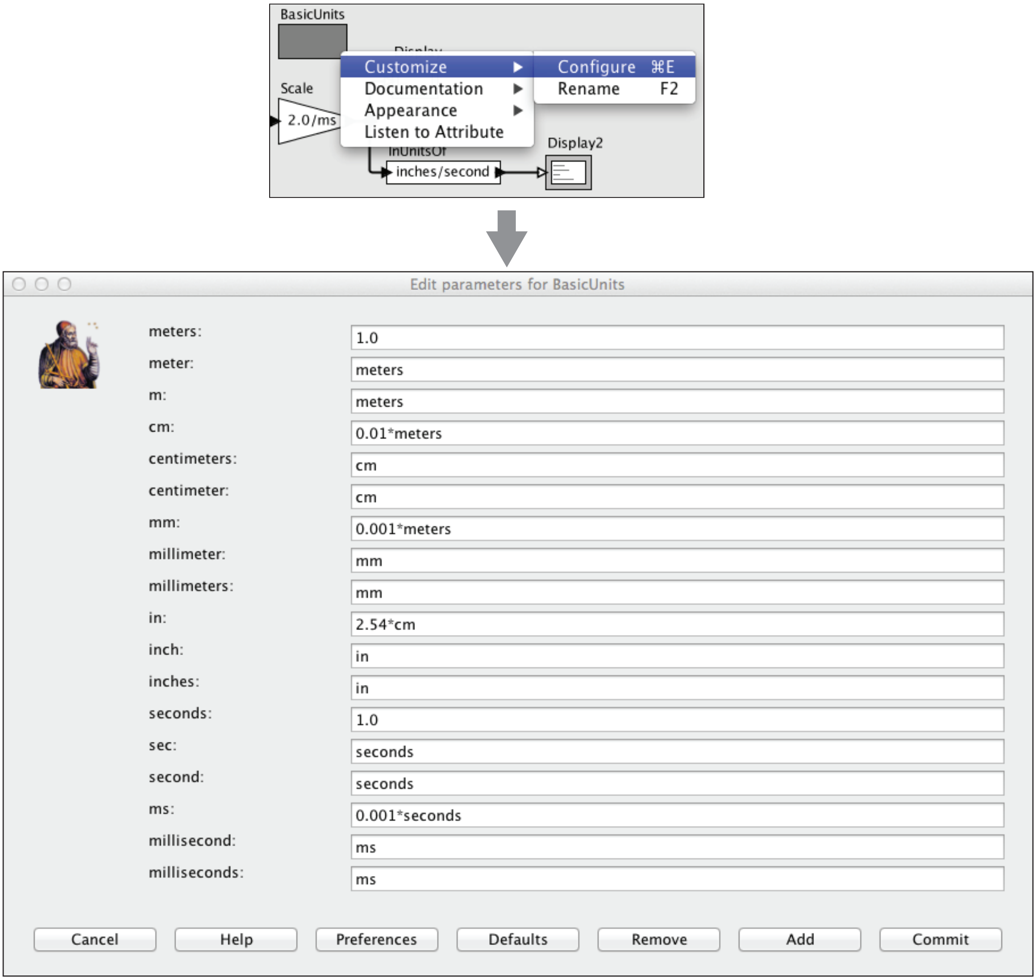


Figure 13.11: Units defined in a units system can be examined by double clicking or by right clicking and selecting Customize and Configure.

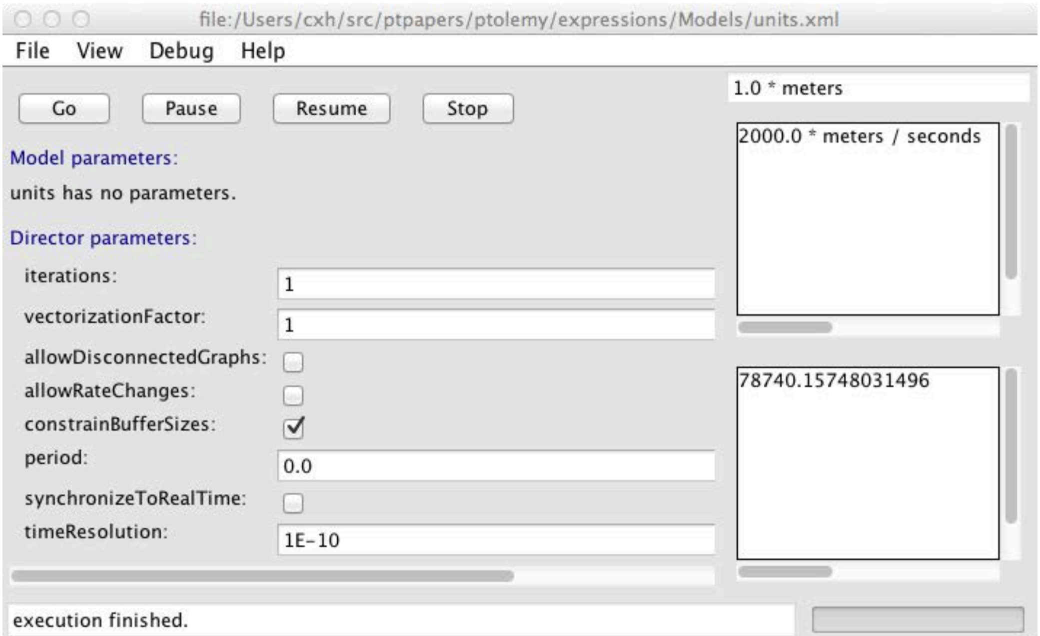


Figure 13.12: Result of running the model in Figure 13.10.

Figure 13.13: Example of an exception resulting from a units mismatch.

13.8 Tables of Functions

Table 13.4: Trigonometric functions.

Name	Argument type(s)	Return type	Description
acos	*double* in the range [-1.0, 1.0] or *complex*	*double* in the range [0.0, π] or NaN if out of range or *complex*	arc cosine *complex* case: $acos(z) = -i\log(z + i\sqrt{i - z^2})$
asin	*double* in the range [-1.0, 1.0] or *complex*	*double* in the range $[-\pi/2, \pi/2]$ or NaN if out of range or *complex*	arc sine *complex* case: $asin(z) = -i\log(iz + \sqrt{i - z^2})$
atan	*double* or *complex*	*double* in the range $[-\pi/2, \pi/2]$ or *complex*	arc tangent *complex* case: $atan(z) = -\frac{i}{2}\log\left(\frac{i-z}{i+z}\right)$
atan2	*double*, *double*	*double* in the range $[-\pi, \pi]$	angle of a vector (note: the arguments are (y, x), not (x, y) as one might expect).
acosh	*double* greater than 1 or *complex*	*double* or *complex*	hyperbolic arc cosine, defined for both *double* and *complex* case by: $acosh(z) = \log\left(z + \sqrt{z^2 - 1}\right)$
asinh	*double* or *complex*	*double* or *complex*	hyperbolic arc sine *complex* case: $asinh(z) = \log\left(z + \sqrt{z^2 + 1}\right)$
cos	*double* or *complex*	*double* in the range [-1, 1], or *complex*	cosine *complex* case: $cos(z) = \frac{exp(iz)+exp(-iz)}{2}$
cosh	*double* or *complex*	*double* or *complex*	hyperbolic cosine, defined for *double* or *complex* by: $cosh(z) = \frac{exp(z)+exp(-z)}{2}$
sin	*double* or *complex*	*double* or *complex*	sine function *complex* case: $sin(z) = \frac{exp(iz)-exp(-iz)}{2i}$
sinh	*double* or *complex*	*double* or *complex*	hyperbolic sine, defined for *double* or *complex* by: $sinh(z) = \frac{exp(z)-exp(-z)}{2}$
tan	*double* or *complex*	*double* or *complex*	tangent function, defined for *double* or *complex* by: $tan(z) = \frac{\sin(z)}{\cos(z)}$
tanh	*double* or *complex*	*double* or *complex*	hyperbolic tangent, defined for *double* or *complex* by: $tanh(z) = \frac{sinh(z)}{cosh(z)}$

Table 13.5: Basic mathematical functions, part 1.

Function	Argument type(s)	Return type	Description
abs	*double* or *complex*	*double* or *int* or *long*(complex returns *double*	absolute value *complex* case: $abs(a+ib) = \|z\| = \sqrt{a^2+b^2}$
angle	*complex*	*double* in the range $[-\pi, \pi]$	angle or argument of the *complex* number: $\angle z$
ceil	*double* or float	*double*	ceiling function, which returns the smallest (closest to negative infinity) *double* value that is not less than the argument and is an integer.
compare	*double*, *double*	*int*	compare two numbers, returning -1, 0, or 1 if the first argument is less than, equal to, or greater than the second.
conjugate	*complex*	*complex*	*complex* conjugate
exp	*double* or *complex*	*double* in the range [0.0, ∞] or complex	exponential function ($e^{argument}$) complex case: $e^{a+ib} = e^a(\cos(b) + i\sin(b))$
floor	*double*	*double*	floor function, which is the largest (closest to positive infinity) value not greater than the argument that is an integer.
gaussian	*double*, *double* or *double*, *double*, *int*, or *double*, *double*, *int*, *int*	*double* or *arrayType(double)* or [*double*]	one or more Gaussian random variables with the specified mean and standard deviation (see 13.4.8).
imag	*complex*	*double*	imaginary part
isInfinite	*double*	*boolean*	return true if the argument is infinite
isNaN	*double*	*boolean*	return true if the argument is "not a number"
log	*double* or *complex*	*double* or *complex*	natural logarithm *complex* case: $log(z) = log(abs(z) + i\angle z)$
log10	*double*	*double*	log base 10
log2	*double*	*double*	log base 2
max	*double*, *double* or double	a scalar of the same type as the arguments	maximum
min	*double*, *double* or double	a scalar of the same type as the arguments	minimum
pow	*double*, *double* or *complex*, *complex*	*double* or complex	first argument to the power of the second

Table 13.6: Basic mathematical functions, part 2.

Function	Argument type(s)	Return type	Description
random	no arguments or int or *int*, *int*	*double* or double or [*double*]	one or more random numbers between 0.0 and 1.0 (see 13.4.8)
real	*complex*	*double*	real part
remainder	*double*, *double*	*double*	remainder after division, according to the IEEE 754 floating-point standard (see 13.4.8).
round	*double*	*long*	round to the nearest long, choosing the next greater integer when exactly in between, and throwing an exception if out of range. If the argument is NaN, the result is 0L. If the argument is out of range, the result is either MaxLong or MinLong, depending on the sign.
roundToInt	*double*	*int*	round to the nearest *int*, choosing the next greater integer when exactly in between, and throwing an exception if out of range. If the argument is NaN, the result is 0. If the argument is out of range, the result is either MaxInt or MinInt, depending on the sign.
sgn	*double*	*int*	-1 if the argument is negative, 1 otherwise
sqrt	*double* or complex	*double* or complex	square root. If the argument is *double* with value less than zero, then the result is NaN. complex case: $sqrt(z) = \sqrt{\lvert z\rvert}(\cos(\frac{\angle z}{2}) + i\sin(\frac{\angle z}{2}))$
toDegrees	*double*	*double*	convert radians to degrees
toRadians	*double*	*double*	convert degrees to radians
within	*type*, *type*, *double*	*boolean*	return true if the first argument is in the neighborhood of the second, meaning that the distance is less than or equal to the third argument. The first two arguments can be any type for which such a distance is defined. For composite types, arrays, records, and matrices, then return true if the first two arguments have the same structure, and each corresponding element is in the neighborhood.

Table 13.7: Functions that take or return matrices, arrays, or records, part 1.

Function	Argument type(s)	Return type	Description
arrayToMatrix	*arrayType(type)*, *int*, *int*	[*type*]	Create a matrix from the specified array with the specified number of rows and columns
concatenate	*arrayType(type)*, *arrayType(type)*	*arrayType(type)*	Concatenate two arrays.
concatenate	*arrayType(arrayType(type))*	*arrayType(type)*	Concatenate arrays in an array of arrays.
conjugateTranspose	[*complex*]	[*complex*]	Return the conjugate transpose of the specified matrix.
createSequence	*type*, *type*, *int*	*arrayType(type)*	Create an array with values starting with the first argument, incremented by the second argument, of length given by the third argument.
crop	[*int*], *int*, *int*, *int*, *int* or [*double*], *int*, *int*, *int*, *int* or [*complex*], *int*, *int*, *int*, *int* or [*long*], *int*, *int*, *int*, *int*	[*int*] or [*double*] or [*complex*] or [*long*] or	Given a matrix of any type, return a submatrix starting at the specified row and column with the specified number of rows and columns.
determinant	[*double*] or [*complex*]	*double* or complex	Return the determinant of the specified matrix.
diag	*arrayType(type)*	[*type*]	Return a diagonal matrix with the values along the diagonal given by the specified array.
divideElements	[*type*], [*type*]	[*type*]	Return the element-by-element division of two matrices
emptyArray	*type*	*arrayType(type)*	Return an empty array whose element type matches the specified token.
emptyRecord		*record*	Return an empty record.
find	*arrayType(type)*, *type*	*arrayType(int)*	Return an array of the indices where elements of the specified array match the specified token.
find	*arrayType(boolean)*	*arrayType(int)*	Return an array of the indices where elements of the specified array have value true.
hilbert	*int*	[*double*]	Return a square Hilbert matrix, where $A_{ij} = \frac{1}{i+j+1}$. A Hilbert matrix is nearly, but not quite singular.

Table 13.8: Functions that take or return matrices, arrays, or records, part 2.

Function	Argument type(s)	Return type	Description
identityMatrixComplex	*int*	[*complex*]	Return an identity matrix with the specified dimension.
identityMatrixDouble	*int*	[*double*]	Return an identity matrix with the specified dimension.
identityMatrixInt	*int*	[*int*]	Return an identity matrix with the specified dimension.
identityMatrixLong	*int*	[*long*]	Return an identity matrix with the specified dimension.
intersect	*record*, *record*	*record*	Return a record that contains only fields that are present in both arguments, where the value of the field is taken from the first record.
inverse	[*double*] or [*complex*]	[*double*] or [*complex*]	Return the inverse of the specified matrix, or throw an exception if it is singular.
matrixToArray	[*type*]	*arrayType(type)*	Create an array containing the values in the matrix
merge	*record*, *record*	*record*	Merge two records, giving priority to the first one when they have matching record labels.
multiplyElements	[*type*], [*type*]	[*type*]	Multiply element wise the two specified matrices.
orthonormalizeColumns	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthonormal columns.
orthonormalizeRows	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthonormal rows.
repeat	*int*, *type*	*arrayType(type)*	Create an array by repeating the specified token the specified number of times.
sort	*arrayType(string)* or *arrayType(realScalar)*	*arrayType(string)* or *arrayType(realScalar)*	Return the specified array, but sorted in ascending order. realScalar is any scalar token except *complex*.
sortAscending	*arrayType(string)* or *arrayType(realScalar)*	*arrayType(string)* or *arrayType(realScalar)*	Return the specified array, but sorted in ascending order. realScalar is any scalar token except *complex*.

Table 13.9: Functions that take or return matrices, arrays, or records, part 3.

Function	Argument type(s)	Return type	Description
sortDescending	*arrayType(string)* or *arrayType(realScalar)*	*arrayType(string)* or *arrayType(realScalar)*	Return the specified array, but sorted in descending order. realScalar is any scalar token except *complex*.
subarray	*arrayType(type)*, *int*, *int*	*arrayType(type)*	Extract a subarray starting at the specified index with the specified length.
sum	*arrayType(type)* or [*type*]	*type*	Sum the elements of the specified array or matrix. This throws an exception if the elements do not support addition or if the array is empty (an empty matrix will return zero).
trace	[*type*]	*type*	Return the trace of the specified matrix.
transpose	[*type*]	[*type*]	Return the transpose of the specified matrix.
update	*int*, *arrayType(type)*	*arrayType(type)*	Update an element in a an array.
zeroMatrixComplex	*int*, *int*	[*complex*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixDouble	*int*, *int*	[*double*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixInt	*int*, *int*	[*int*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixLong	*int*, *int*	[*long*]	Return a zero matrix with the specified number of rows and columns.

Table 13.10: Utility functions for evaluating expressions

Function	Argument type(s)	Return type	Description
eval	*string*	any type	evaluate the specified expression (see 13.4.8).
parseInt	*string* or *string*, *int*	*int*	return an *int* read from a string, using the given radix if a second argument is provided.
parseLong	*string* or *string*, *int*	*int*	return a long read from a string, using the given radix if a second argument is provided.
toBinaryString	*int* or *long*	*string*	return a binary representation of the argument
toOctalString	*int* or *long*	*string*	return an octal representation of the argument
toString	*double* or *int* or *int*, *int* or *long*	*string*	return a string representation of the argument, using the given radix if a second argument is provided.
traceEvaluation	*string*	*string*	evaluate the specified expression and report details on how it was evaluated (see 13.4.8).

Table 13.11: Functions performing signal processing operations, part 1.

Function	Argument type(s)	Return type	Description
close	*double*, *double*	*boolean*	Return true if the first argument is close to the second (within EPSILON, where EPSILON is a static public variable of this class).
convolve	*arrayType(double)*, *arrayType(double)* or *arrayType(complex)*, *arrayType(complex)*	*arrayType(double)* or *arrayType(complex)*	Convolve two arrays and return an array whose length is sum of the lengths of the two arguments minus one. Convolution of two arrays is the same as polynomial multiplication.
DCT	*arrayType(double)* or *arrayType(double)*, *int* or *arrayType(double)*, *int*, *int*	*arrayType(double)*	Return the Discrete Cosine Transform of the specified array, using the specified (optional) length and normalization strategy (see 13.4.8).
downsample	*arrayType(double)*, *int* or *arrayType(double)*, *int*, *int*	*arrayType(double)*	Return a new array with every n-th element of the argument array, where n is the second argument. If a third argument is given, then it must be between 0 and $n-1$, and it specifies an offset into the array (by giving the index of the first output).
FFT	*arrayType(double)* or *arrayType(complex)* or *arrayType(double)*, int *arrayType(complex)*, *int*	*arrayType(complex)*	Return the Fast Fourier Transform of the specified array. If the second argument is given with value n, then the length of the transform is 2^n. Otherwise, the length is the next power of two greater than or equal to the length of the input array. If the input length does not match this length, then input is padded with zeros.
generateBartlett Window	*int*	*arrayType(double)*	Bartlett (rectangular) window with the specified length. The end points have value 0.0, and if the length is odd, the center point has value 1.0. For length M + 1, the formula is: $w(n) = \begin{cases} 2\frac{n}{M}; & if 0 \leq n \leq \frac{M}{2} \\ 2 - 2\frac{n}{M}; & if \frac{M}{2} \leq n \leq M \end{cases}$

Table 13.12: Functions performing signal processing operations, part 2.

Function	Argument type(s)	Return type	Description
generateBlackman Window	*int*	*arrayType(double)*	Return a Blackman window with the specified length. For length M + 1, the formula is: $w(n) = 0.42 + 0.5\cos\left(\frac{2\pi n}{M}\right) + 0.08\cos\left(\frac{4\pi n}{M}\right)$
generateBlackman HarrisWindow	*int*	*arrayType(double)*	Return a Blackman-Harris window with the specified length. For length M + 1, the formula is: $w(n) = 0.35875 + 0.48829\cos\left(\frac{2\pi n}{M}\right) + 0.14128\cos\left(\frac{4\pi n}{M}\right) + 0.01168\cos\left(\frac{6\pi n}{M}\right)$
generateGaussian Curve	*arrayType(double)*, *arrayType(double)*, *int*	*arrayType(double)*	Return a Gaussian curve with the specified standard deviation, extent, and length. The extent is a multiple of the standard deviation. For instance, to get 100 samples of a Gaussian curve with standard deviation 1.0 out to four standard deviations, use generateGaussianCurve(1.0, 4.0, 100).
generateHamming Window	*int*	*arrayType(double)*	Return a Hamming window with the specified length. For length M + 1, the formula is: $w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M}\right)$
generateHanning Window	*int*	*arrayType(double)*	Return a Hanning window with the specified length. For length M + 1, the formula is: $w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{/} M\right)$
generatePolynomial Curve	*arrayType(double)*, *double*, *double*, *int*	*arrayType(double)*	Return samples of a curve specified by a polynomial. The first argument is an array with the polynomial coefficients, beginning with the constant term, the linear term, the squared term, etc. The second argument is the value of the polynomial variable at which to begin, and the third argument is the increment on this variable for each successive sample. The final argument is the length of the returned array.

Table 13.13: Functions performing signal processing operations, part 3.

Function	Argument type(s)	Return type	Description
generateRaised CosinePulse	*double*, *double*, *int*	*arrayType(double)*	Return an array containing a symmetric raised-cosine pulse. This pulse is widely used in communication systems, and is called a "raised cosine pulse" because the magnitude its Fourier transform has a shape that ranges from rectangular (if the excess bandwidth is zero) to a cosine curved that has been raised to be non-negative (for excess bandwidth of 1.0). The elements of the returned array are samples of the function: $h(t) = \frac{\sin(\frac{\pi t}{T})}{\frac{\pi t}{T}} \times \frac{\cos(\frac{x \pi t}{T})}{1-(\frac{2xt}{T})^2}$, where x is the excess bandwidth (the first argument) and T is the number of samples from the center of the pulse to the first zero crossing (the second argument). The samples are taken with a sampling interval of 1.0, and the returned array is symmetric and has a length equal to the third argument. With an excess Bandwidth of 0.0, this pulse is a sinc pulse.
generate Rectangular Window	*int*	*arrayType(double)*	Return an array filled with 1.0 of the specified length. This is a rectangular window.
IDCT	*arrayType(double)* or *arrayType(double)*, *int* or *arrayType(double)*, *int*, *int*	*arrayType(double)*	Return the inverse discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see 13.4.8).
IFFT	*arrayType(double)* or *arrayType(complex)* or *arrayType(double)*, *int* *arrayType(complex)*, *int*	*arrayType(complex)*	inverse fast Fourier transform of the specified array. If the second argument is given with value n, then the length of the transform is 2^n. Otherwise, the length is the next power of two greater than or equal to the length of the input array. If the input length does not match this length, then input is padded with zeros.

Table 13.14: Functions performing signal processing operations, part 4.

Function	Argument type(s)	Return type	Description
nextPowerOfTwo	*double*	*int*	Return the next power of two larger than or equal to the argument.
poleZeroTo Frequency	*arrayType(complex)*, *arrayType(complex)*, *complex*, *int*	*arrayType(complex)*	Given an array of pole locations, an array of zero locations, a gain term, and a size, return an array of the specified size representing the frequency response specified by these poles, zeros, and gain. This is calculated by walking around the unit circle and forming the product of the distances to the zeros, dividing by the product of the distances to the poles, and multiplying by the gain.
sinc	*double*	*double*	Return the sinc function, $\sin(x)/x$, where special care is taken to ensure that 1.0 is returned if the argument is 0.0.
toDecibels	*double*	*double*	Return $20 \times \log_{10}(z)$, where z is the argument.
unwrap	*arrayType(double)*	*arrayType(double)*	Modify the specified array to unwrap the angles. That is, if the difference between successive values is greater than π in magnitude, then the second value is modified by multiples of 2π until the difference is less than or equal to π. In addition, the first element is modified so that its difference from zero is less than or equal to π in magnitude.
upsample	*arrayType(double)*, *int*	*arrayType(double)*	Return a new array that is the result of inserting $n-1$ zeroes between each successive sample in the input array, where n is the second argument. The returned array has length nL, where L is the length of the argument array. It is required that $n > 0$.

Table 13.15: Miscellaneous functions, part 1.

Function	Argument type(s)	Return type	Description
asURL	*string*	*string*	Return a URL representation of the argument.
cast	type1, type2	type1	Return the second argument converted to the type of the first, or throw an exception if the conversion is invalid.
constants	none	*record*	Return a record identifying all the globally defined constants in the expression language.
findFile	*string*	*string*	Given a file name relative to the user directory, current directory, or classpath, return the absolute file name of the first match, or return the name unchanged if no match is found.
filter	function, *arrayType(type)*	*arrayType(type)*	Extract a sub-array consisting of all of the elements of an array for which the given predicate function returns true.
filter	function, *arrayType(type)*, *int*	*arrayType(type)*	Extract a sub-array with a limited size consisting of all of the elements of an array for which the given predicate function returns true.
freeMemory	none	*long*	Return the approximate number of bytes available for future memory allocation.
iterate	function, *int*, *type*	*arrayType(type)*	Return an array that results from first applying the specified function to the third argument, then applying it to the result of that application, and repeating to get an array whose length is given by the second argument.
map	function, *arrayType(type)*	*arrayType(type)*	Return an array that results from applying the specified function to the elements of the specified array.
property	*string*	*string*	Return a system property with the specified name from the environment, or an empty string if there is none. Some useful properties are java.version, ptolemy.ptII.dir, ptolemy.ptII.dirAsURL, and user.dir.

Table 13.16: Miscellaneous functions, part 2.

Function	Argument type(s)	Return type	Description
readFile	*string*	*string*	Get the string text in the specified file, or throw an exception if the file cannot be found. The file can be absolute, or relative to the current working directory (user.dir), the user's home directory (user.home), or the classpath. The readfile function is often used with the eval function.
readResource	*string*	*string*	Get the string text in the specified resource (which is a file found relative to the classpath), or throw an exception if the file cannot be found.
totalMemory	none	*long*	Return the approximate number of bytes used by current objects plus those available for future object allocation.
yesNoQuestion	*string*	*boolean*	Query the user for a yes-no answer and return a boolean. This function will open a dialog if a GUI is available, and otherwise will use standard input and output.

14

The Type System

Edward A. Lee, Marten Lohstroh, and Yuhong Xiong

Contents

In a programming language, a **type system** associates a type with each variable. A **type** is logically a family of values that the variable can take on. For example, the type *double* is the set of all double-precision floating point numbers represented by 64 bits according to the IEEE 754 standard for floating-point arithmetic. A **strong type system** will prevent a program from using the 64-bit value of a *double* variable as, for example, a pointer into memory or a *long* integer (Liskov and Zilles, 1974). Java has a strong type system; C does not. A good introduction to type systems is given by Cardelli and Wegner (1985).

The Ptolemy II type system associates a type with each port and parameter in a model. Ptolemy II uses **type inference**, where the types of parameters and ports are inferred based on their usage. Types need not be declared by the model builder, usually.

A programming language where types are checked at compile time is said to be **statically typed**. In Ptolemy II, types are checked just prior to execution of a model, between the preinitialize and initialize phases of execution. Since this happens once, before execution of the model, we consider Ptolemy II to be statically typed.

Although the type system is a strong one (a port will not receive a token that is incompatible with its declared type, for example), there are loopholes. In particular, users can define their own actors in Java, and these actors may not behave well. For example, an actor may declare an output port to be of type *int* and then attempt to send a *string* through that port. To catch such errors, Ptolemy II also performs run-time type checks. Although it is rare (unless you write your own actors), it is possible to build models that will exhibit type errors only during execution. These errors will not be detected by the static type checker.

Fortunately, with the help of static type checking, run-time type checks can be performed automatically when a token is sent out from a port. The run-time type checker simply compares the type of a produced token against the (static) type of the output port. This way, a type error is detected at the earliest possible time (when the token is produced, rather than when it is used). However, this does not guarantee that a type of token that an actor accepts is indeed compatible with the operation it implements. A run-time type error may therefore also be thrown by the actor itself, particularly if the actor is written incorrectly.

The Ptolemy II type system supports **polymorphism**, where actors can operate on a variety of data types. To facilitate the construction of polymorphic actors, the type system offers a mechanism called **automatic type conversion**, which allows a component to receive multiple data types by automatically converting them to a single data type, assuming that the conversion can be done without loss of information. Polymorphism greatly increases the reusability of actors in the presence of static typing, especially in combination with **type inference**. In this chapter we describe how these mechanisms are integrated into the Ptolemy II static type checking framework, with emphasis on how they help build correct models.

14.1 Type Inference, Conversion, and Conflict

In Ptolemy II models, types of ports are inferred from the model, subject to constraints imposed by the actors and the infrastructure. We will explain exactly how these constraints come about and how the inference is performed, but first we can develop an intuitive understanding of what happens.

Example 14.1: Consider the example shown in Figure 14.1. This model calculates and displays $1 - \pi$. (This is a silly model, since the entire model could be replaced by the expression `1 - PI`, but it serves to illustrate the type system.) The model includes an attribute called *ShowTypes*, which can be found in the `Utilities→Analysis` library. This attribute causes Vergil to display the type of each port next to the port (in addition to the name of the port, if the name would otherwise be displayed). As you can see in the figure, initially the type of each port is *unknown*.

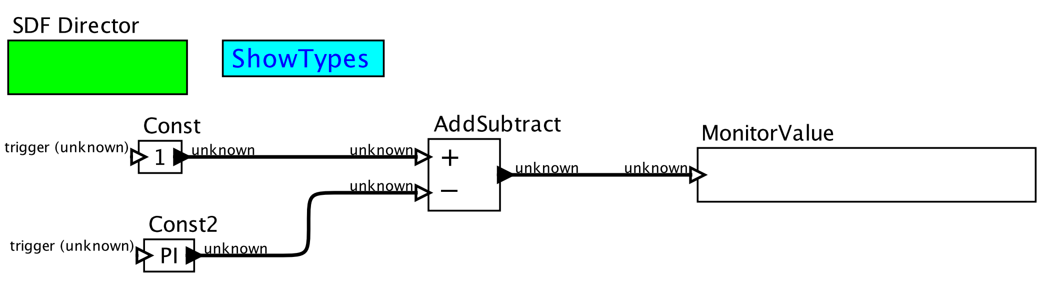


Figure 14.1: A simple example for illustrating type inference and conversion. [online]

During the first stage of execution, preinitialize, the types of the ports are inferred, after which the display is eventually updated as shown in Figure 14.2. The types of the output ports of the Const actors are determined by their *value* parameters, which are `1` and `PI`, respectively. If you change a *value* parameter to, for example, `1+i`, then the output type will become *complex*. If you change it to `{1, 2}`, then the output type will become *arrayType(int, 2)*, an array with *int* elements and length 2.

Notice in Figure 14.2 that the input ports of the AddSubtract actor have both resolved to *double*. The AddSubtract actor imposes a constraint that its two input ports must have the same type. When the AddSubtract actor receives an *int* token from the Const actor, the input port will automatically convert the token to type *double*. We will explain in detail what conversions are allowed, but intuitively, a conversion is allowed if no information is lost.

Notice in Figure 14.2 that the output port of the AddSubtract actor and the input port of MonitorValue have also resolved to *double*. The MonitorValue actor can accept any input type, since it simply displays a string representation of the token, and every token has a string representation.

The previous example illustrates that the type of a parameter in one part of a model can have far-reaching consequences in other parts of the model. The type system ensures consistency. It is common when building models to raise type errors, as illustrated by the following example.

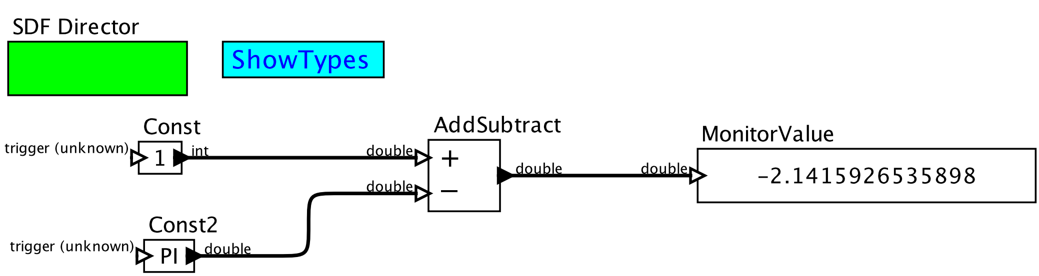


Figure 14.2: After execution.

Example 14.2: Consider the example shown in Figure 14.3. In that example, we have replaced MonitorValue with SequencePlotter, and we have changed the Const actor to produce a *complex* value. Type inference determines that the output of AddSubtract is *complex*. But SequencePlotter requires an input of type *double*. A *complex* token cannot be losslessly converted to a *double* token, so upon executing the model you will get the following exception:

```
ptolemy.actor.TypeConflictException:  Type conflicts occurred on
the following inequalities:
  (port .TypeConflict.AddSubtract.output:  complex) <=
  (port .TypeConflict.SequencePlotter.input:  double)
  in .TypeConflict
```

This error message reports that a type constraint in the model cannot be satisfied. That type constraint is that the type of the output port of AddSubtract must be *less than or equal to* ($\leq$) the type of the input port of SequencePlotter. Further, it reports that the output port has type *complex*, while the input port has type *double*. The offending ports and their containers are then highlighted in the model as shown in the figure.

In the above example, a type constraint is given as an inequality, an assertion that the type of one port must be “less than or equal to” the type of another. What does this mean?

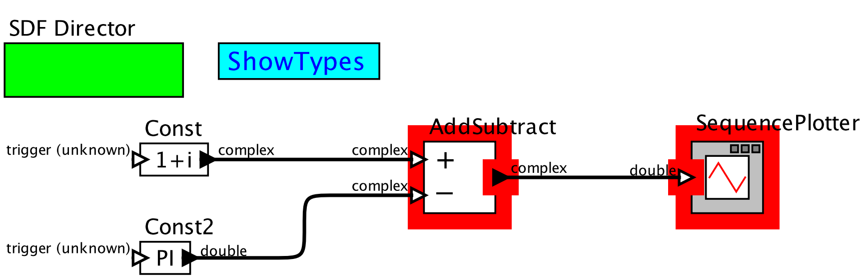


Figure 14.3: Type conflict.

Intuitively, one type is less than another if it can be lossless converted to that other type. We examine this inequality relation next.

14.1.1 Automatic Type Conversion

The allowed automatic type conversions are represented in Figure 14.4, which depicts the **type lattice** of Ptolemy II. In this diagram, a conversion from a first type to a second type is allowed if there is an upward path from the first type to the second type in the diagram. This relationship implies a partial order on types (see sidebar on page 513), so we might say that a conversion is allowed if the first type is less than or equal to the second type. This partial order has an elegant mathematical structure called a lattice (see sidebar on page 513) that enables efficient type inference and type checking.

Automatic conversions occur when an actor retrieves data from its input port.* The types of ports are determined prior to execution, and run-time type checking guarantees that tokens sent through an output port are compatible with the types of downstream input ports (i.e., a conversion to such type is allowed by the type lattice). This is due to the type constraints that are imposed by connections between ports. These type constraints are explained in Section 14.1.2. If a token is not compatible, the run-time type checker will throw an exception before the token is sent. Hence, run-time type errors are detected as early as possible.

A type conversion can also be forced in the expression language using the `cast` function, one of many built-in functions available in the expression language. An expression of the form `cast(newType, value)` will convert the specified value into the specified type. See Section 13.4.3 and Table 13.15 for information about the `cast` function.

The type lattice is constructed based on a principle of lossless conversion. A conversion is allowed automatically as long as important information about value of data tokens is not lost. Such conversions are referred to as **widening conversions** in Java. For instance, converting a 32-bit signed integer to a 64-bit IEEE double precision floating point number is allowed, since every integer can be represented exactly as a floating point number. On the other hand, data type conversions that lose information are not automatic.

* Some actors disable automatic type conversion on their input ports because they do not need the conversion. For example, AddSubtract and Display accept any token type, because they make use of methods that are inherited by all token types. These actors disable automatic conversion by invoking:
`portName.setAutomaticTypeConversion(false).`

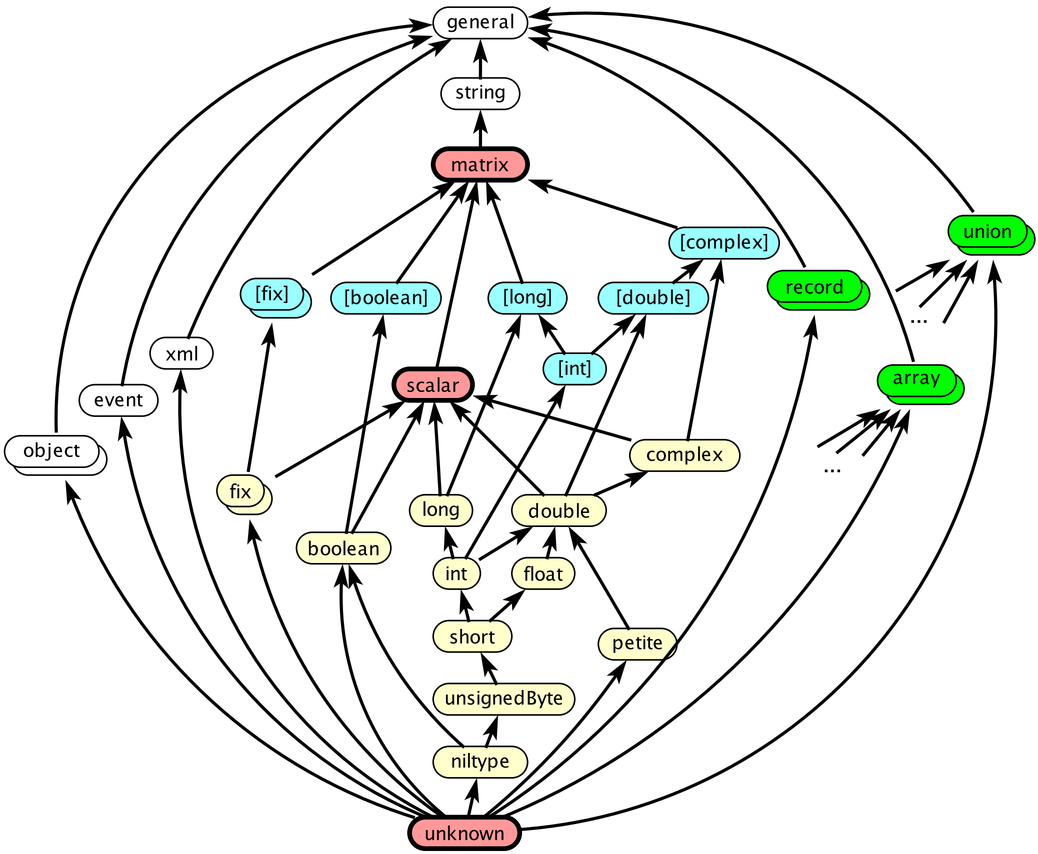


Figure 14.4: The Type Lattice. Types that cannot be instantiated are shown with bold outlines (and pink fill). Scalar types are lightly filled (in yellow), matrix types are slightly more darkly filled (in cyan), and composite types still more darkly filled (in green). The composite types and the *fix* types are infinite sublattices, as suggested by their double icons.

Sidebar: What is a Lattice?

A lattice is a mathematical structure that has properties that enable efficiently solving type constraints. A lattice is a set with particular kind of **order relation** related to CPOs (see sidebar on page 182). See Davey and Priestly (2002) for more details.

First, a total order is an ordering over a set S, denoted $\leq$, where any two elements of the set are ordered. Specifically, for any $x, y \in S$, either $x \leq y$ or $y \leq x$ (or both, in which case $x = y$). For example, if the set S is the set of integers, and $\leq$ denotes the ordinary arithmetic ordering, then $(S, \leq)$ is a total order.

A partial order relaxes the constraint that any two elements be ordered. An example of a partial order is $(S, \leq)$, where S is a set of sets and $\leq$ is the subset relation, usually denoted $\subseteq$. Specifically, if A, B are both sets in S, then it may be that neither $A \subseteq B$ nor $B \subseteq A$. E.g., let $A = \{1, 2\}$ and $B = \{2, 3\}$; then neither is a subset of the other.

Another partial order is the **prefix order** on strings. Let S be the set of sequences of alphabetic characters, for example. Then for two strings $x, y \in S$, we write $x \leq y$ is x is a prefix of y. E.g., if $x =$abc and $y =$abcd, then $x \leq y$. If $z =$bc, then neither $x \leq z$ nor $z \leq x$ holds. Formally, a partial order is a set S and a relation $\leq$, such that for all $x, y, z \in S$,

- $x \leq x$ (the order is reflexive),
- if $x \leq y$ and $y \leq z$, then $x \leq z$ (the order is transitive), and
- if $x \leq y$ and $y \leq x$, then $x = y$ (the order is antisymmetric).

The least upper bound (LUB), if it exists, of a subset $U \subseteq S$ of a partial order $(S, \leq)$ is the least element $x \in S$ such that for every $u \in U$, $u \leq x$. The **greatest lower bound** (**GLB**) of U, if it exists, is the greatest element $x \in S$ such that for every $u \in U$, $x \leq u$. E.g., in the prefix order, if $x =$abc, $y =$abcd, and $z =$bc, then the LUB of $\{x, y\}$ is y. The LUB of $\{x, z\}$ does not exist. The GLB of $\{x, y\}$ is x. The GLB of $\{x, z\}$ is the empty string, which is a prefix of all strings.

A **lattice** is a partial order $(S, \leq)$ for which every subset of S has a GLB and a LUB. The subset order is a lattice, because the LUB can be found with set union, and the GLB can be found with set intersection. The prefix order on strings, however, is not a lattice, because two strings may not have a LUB. The prefix order is a **lower semi-lattice**, however, because the GLB of a set of strings always exists.

14.1.2 Type Constraints

A model imposes a number of constraints that drive type inference. A constraint is expressed as an inequality between the types of two ports. It requires one port to have a type that is less than or equal to (losslessly convertible to) the type of the other port.

In a Ptolemy II topology, the **type compatibility rule** requires an output port to have a type that is less than or equal to all inputs to which it is connected, as follows:

$$outType \leq inType \tag{14.1}$$

This constraint guarantees that there is an allowed automatic conversion that can be performed during data transfer. Every connection between an output port and an input port establishes a type constraint that enforces this rule.

Example 14.3: In Figure 14.2, the Const actor produces type *int*, while the AddSubtract actor receives type *double*. In Figure 14.4, we see that *int* is less than *double*, so the type compatibility rule is satisfied.

In addition to the constraints imposed by the connections between actors, most actors also impose constraints. For example, the AddSubtract actor declares that its output type is greater than or equal to its input types, and that the types of its two input ports are equal. An equality constraint is equivalent to two inequality constraints, as in:

$$\begin{aligned} plus &\leq minus \\ minus &\leq plus, \end{aligned}$$

where *plus* and *minus* are the input ports of the AddSubtract actor.

Many actors impose a **default type constraint** that requires an unconstrained input to be less than or equal every unconstrained output. By default, actors implicitly include this type constraint for every set of input and output ports that have no explicit type constraint.

Example 14.4: Some actors operate on tokens without regard for the actual types of the tokens. For example, the DownSample does not care about the type of token

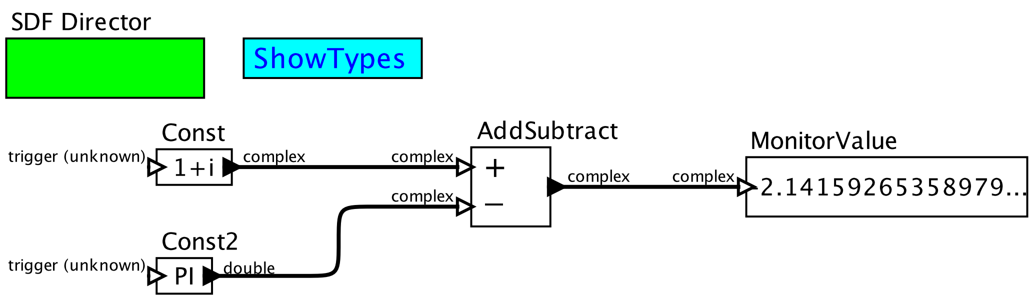


Figure 14.5: Type conflict of Figure 14.3 resolved by using an actor that can accept any input type, MonitorValue. [online]

going through it, so it does not explicitly declare any type constraints. The default type constraint enables type information to propagate through this actor from its input to its output.

By default, actors that leave their input ports undeclared will have the type of the input port determined by the upstream model (unless the model has enabled backward type inference, as explained below in Section 14.1.4).

Example 14.5: The MonitorValue actor, which displays the value of tokens it receives, can accept any type of input. By default, it leaves its input undeclared, which results in the type resolving to whatever is provided upstream. For example, Figure 14.5 resolves the type conflict of Figure 14.3 by replacing the SequencePlotter with a MonitorValue actor. The resolved type of the input, *complex*, is determined by the upstream actors.

14.1.3 Type Declarations

Sometimes, there is not enough information in a model to infer types from the sources of data.

Example 14.6: Consider for example the model in Figure 14.6. This model is intended to evaluate expressions entered by a user in a shell, but type resolution fails, as shown. The ExpressionToToken actor takes an input string, which is expected to be an expression in the Ptolemy expression language (see Chapter 13), and evaluates the expression. The result of evaluation is produced on the output. There is no way to anticipate what the user might type in the shell, so there is not enough information to infer types. The type of the output of the ExpressionToToken remains *unkonwn*.

Such difficulties can be fixed by enabling backward type inference (discussed below), or by explicitly declaring the type of a port, as illustrated in the next example.

Example 14.7: We can force the type of output of the ExpressionToToken actor to be of type *general*, as shown in Figure 14.7. Downstream types resolve to *general* as well.

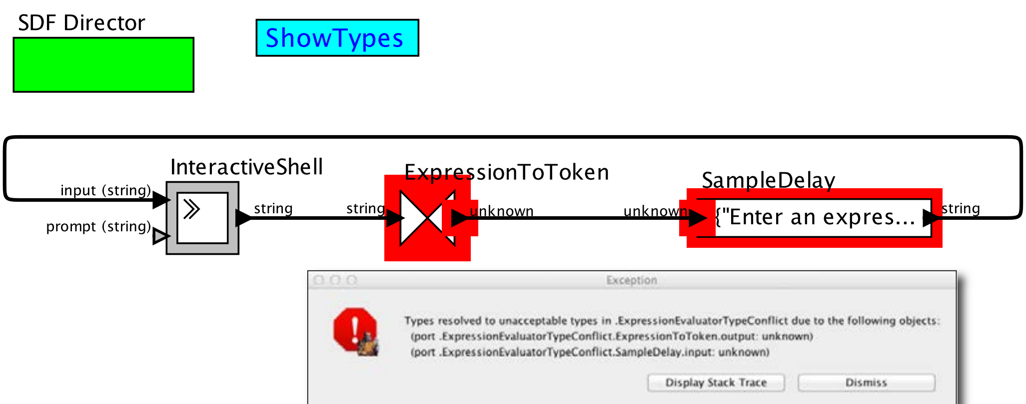


Figure 14.6: A model intended to evaluate expressions entered by a user, but for which there is not enough information for types to be inferred from the sources of data.

In the type column of the port dialog, you can enter any expression in the expression language. Whatever type that expression resolves to will be the declared type of the corresponding port. For clarity, Ptolemy II provides some built in variables that designate a type. The variable named `general`, for instance, evaluates to a token of type *general*. Similarly, the variable named `double` evaluates to a token of type *double* (which happens to have value 0.0, but the value immaterial). Table 14.8 lists the predefined variable names and their corresponding types.

14.1.4 Backward Type Inference

In all the examples discussed so far, type inference propagates forward in models, with each constant or fixed output type causing downstream types to resolve. That is, type information travels in the same direction that the tokens are sent during execution. The type compatibility rule given by (14.1) imposes no useful constraints on output ports, because it is always satisfied if *outType* has type *unknown*, the bottom element of the type lattice. Nevertheless, **forward type inference** is usually sufficient because sources of data in most models provide specific type information about those data.

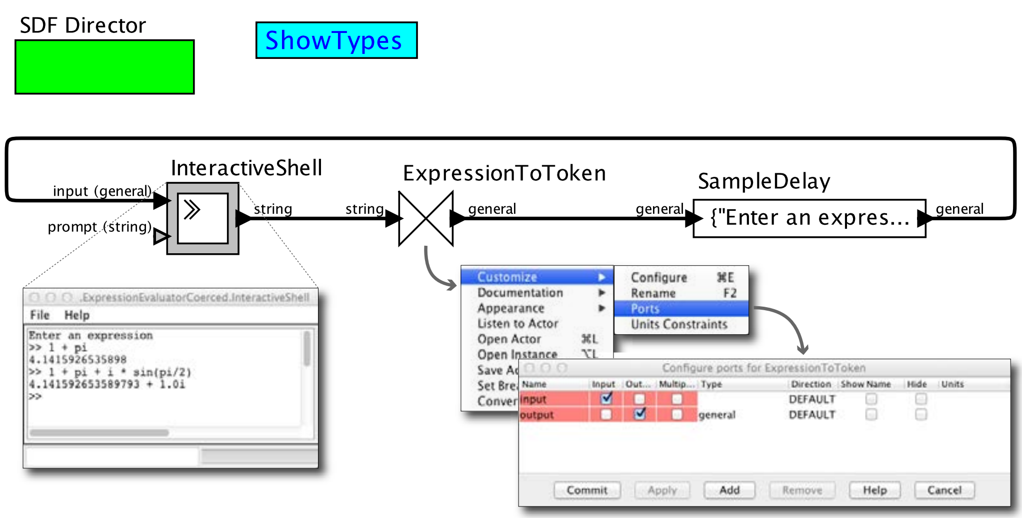


Figure 14.7: Types of ports declared by entering a type into the type column of the port configuration dialog. [online]

BaseType field	Expression	Description
UNKNOWN	unknown	bottom element of the data type lattice
ARRAY_BOTTOM		*array* of unknown type
BOOLEAN	boolean	boolean (true or false)
BOOLEAN_MATRIX	[boolean]	matrix of booleans
UNSIGNED_BYTE	unsignedByte	unsigned byte
COMPLEX	complex	complex number
COMPLEX_MATRIX	[complex]	complex matrix
FLOAT	float	32-bit IEEE floating-point number
DOUBLE	double	64-bit IEEE floating-point number
DOUBLE_MATRIX	[double]	matrix of doubles
FIX	fixedpoint	fixed-point data type
FIX_MATRIX	[fixedpoint]	matrix of fixed-point numbers
SHORT	short	16-bit integer
INT	int	32-bit integer
INT_MATRIX	[int]	matrix of 32-bit integers
LONG	long	64-bit integer
LONG_MATRIX	[long]	matrix of 64-bit integers
OBJECT	object	object type
ACTOR		actor type
XMLTOKEN		XML type
SCALAR	scalar	scalar number
MATRIX		matrix of unknown type
STRING	string	string
GENERAL	general	any type
EVENT		event (empty token)
PETITE		a double constrained to be between -1 and 1
NIL	nil	nil type
RECORD		record type

Figure 14.8: Type constants defined in the BaseType class with their corresponding name in the expression language, if there is one.

The models in Figures 14.6 and 14.7, however, do not have this property. First, since the model forms a loop, there is no clear "source" of data. Every actor is both upstream and downstream of every other actor. Moreover, the ExpressionToToken actor, by nature of what it does, cannot provide any specific information about the type of its output.

The Ptolemy II type system optionally provides **backward type inference** to solve this problem. To enable backward type inference, set the *enableBackwardTypeInference* parameter to true at the top level of the model, as shown in Figure 14.9. This has three effects. First, it causes certain actors that do not impose restrictions on the data received at input ports to declare those ports to have type *general*. This includes InteractiveShell and Display, for example. Second, it allows type constraints to propagate upstream. Specifically, it adds an additional constraint to the type compatibility rule of (14.1). The additional constraint is that the type of each output port is required to be greater than or equal to the greatest lower bound (GLB) of the types of all input ports to which it is connected. Third, for each actor that does not explicitly constrain the type relationships of its port, it imposes a default type constraint that the types of its input ports are greater than or equal to the GLB of the types of its output ports. These additional constraints are sufficient for the types to resolve to the same solution that we achieved with type coercion in Figure 14.7.

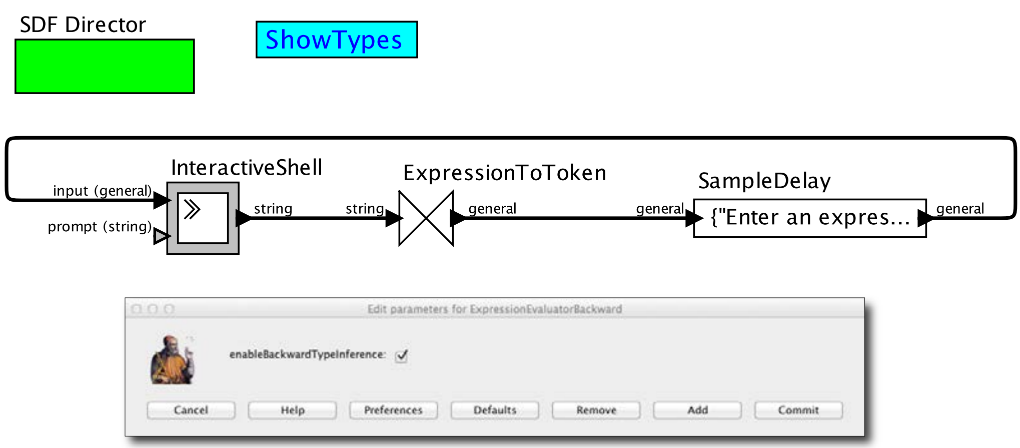


Figure 14.9: Enabling backward type inference allows type constraints to propagate upstream. [online]

The root of the problem is the ExpressionToToken actor, which cannot itself be specific about its output type. Any constraints on its output type would have to result from how its output tokens are used, rather than from the actor itself. In Figure 14.9, the InteractiveShell input accepts type *general*, which therefore propagates upstream to the output of the ExpressionToToken actor. When there are undeclared output port types, such as on the ExpressionToToken actor, backward type inference finds the the most general type that is compatible with downstream type constraints.

14.2 Structured Types

14.2.1 Arrays

Structured types include those tokens which aggregate other tokens of arbitrary type, such as array and record types. As described in Section 13.3, an array is an ordered list of tokens, all of which have the same type. Records contain a set of labeled tokens, like a struct in the C language. It is useful for grouping multiple pieces of related information together. In the type lattice in Figure 14.4, record types are incomparable with all the base types except *unknown*, *string*, and *general*. Array types are a bit more complex because any type is less than an array of that type in the type lattice. This is hinted at in the figure with the disconnected lines at the bottom of the array type. Note that the lattice nodes *array* and *record* actually represent an infinite number of types, so the type lattice is infinite.

For any type a, the following type relation holds,

$$a < \{a\}.$$

A value can be losslessly converted to an array of values. Moreover,

$$a < b \quad \Rightarrow \quad \{a\} < \{b\}.$$

Combining these, we see that the definition is recursive, so

$$a < \{a\} < \{\{a\}\} < \{\{\{a\}\}\} \cdots$$

Example 14.8: *int* $\leq$ *double*, so the following all hold:

$$\begin{array}{rcl} int & < & \{int\} \\ \{int\} & < & \{double\} \\ int & < & \{double\} \\ int & < & \{\{double\}\} \\ \cdots & & \end{array}$$

A consequence of these type relations is that there is an infinite path from any particular array type to the top of the type lattice. This can result in situations where type inference does not converge.

Example 14.9: In the model in Figure 14.10, the Expression actor constructs an array consisting of one element, its input token. When you attempt to run this model, an exception occurs that includes the message

```
Large type structure detected during type resolution
```

The reason for this is that there is no (finite) type that satisfies all the constraints. The SampleDelay actor requires its output to be at least an *int*, because it produces an initial *int*. But it also requires that its output be greater than equal to its input. The first input it will receive will have type $\{int\}$, and the second input will have type $\{\{int\}\}$, etc. The only possible type is an infinite nesting of arrays.

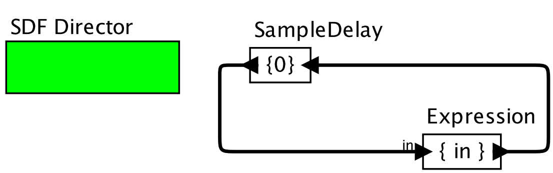


Figure 14.10: Example where type inference does not converge.

The Ptolemy II type system often (but not always) includes the *length* of an array in its type. If you explicitly query the type of a token, you can see this, as in the following command in the expression evaluator:

```
>> {1, 2}.getType()
object(arrayType(int,2))
```

Thus, `arrayType(int,2)` is the type of an array of length two, whereas `arrayType(int)` is the type of an array with indeterminate length. Including the length in the type allows the type system to detect more errors than otherwise.

Generally speaking, array types with specific length are incomparable with array types with different lengths, and can be converted to an array type with unknown length (and compatible element type). Scalars are convertible to array types with length 1.

One subtlety is that when you specify an array type with an expression `{ int }`, you are actually giving the type `arrayType(int, 1)`, which is more specific than you probably want. For this reason, unless you specifically want to constrain array types, it is better to specify an array type with `arrayType(int)`.

14.2.2 Records

The order relation between two record types follows the standard **depth subtyping** and **width subtyping** relations commonly used for such types (Cardelli, 1997). In depth subtyping, a record type c is a subtype of a record type d (i.e., $c \leq d$) if the fields of c are a subtype of the corresponding fields in d. For example,

```
{x = string, y = int} <= {x = string, y = double}
```

In width subtyping, a record with more fields is a subtype of a record with fewer fields. For example, we have:

```
{x = string, y = double, z = int} <= {x = string, y = double}
```

The width subtyping rule is a bit counterintuitive, as it implies a type conversion which loses information, discarding the extra fields of a record. However, it conforms with the "**is a**" interpretation of types, where $a \leq b$ if a is a b. In this case, the record with more fields "is an" instance of the record with fewer fields, whereas the reverse is not true.

14.2.3 Unions

Another structured type is the union type. It allows the user to create a token that can hold data of various types, but only one at a time. This is like the union construct in C. The union type is also called a **variant type** in the type system literature. The width subtyping relation for union type is the opposite to that of the record type. That is, a shorter union is a subtype of a longer one. Again, this corresponds with the "is a" interpretation of type relations.

A consequence of this width subtyping relation is that there are an infinite number of types from a particular union type to the top of the type lattice. This again means that type inference may not converge. The Ptolemy II type system truncates type inference after a finite number of steps, providing a heuristic that is a likely indicator of a type error.

14.2.4 Functions

One final structured type is the expression language function, described in Section 13.4.4. Functions can take several arguments and return a single value. The type system supports **function types**, where the arguments have declared types, and the return type is known. Function types are related in a way that is contravariant (oppositely related) between inputs and outputs. Namely, if `function(x:int, y:int) int` is a function with two integer arguments that returns an integer, then

```
function(x:int, y:int) int <= function(x:int, y:int) double
function(x:int, y:double) int <= function(x:int, y:int) int
```

The contravariant notion here is easiest to think about in terms of the automatic type conversion of one function into another. A function that returns *int* can be converted into a function that returns *double* by adding a conversion of the returned value from *int* to *double*. On the other hand, a function that takes an *int* cannot be converted into a function that takes a *double*, since that would mean that the function is suddenly able to accept *double* arguments when it could not before, and there is no automatic conversion from *double* to *int*. Functions that are lower in the type lattice assume less about their inputs and guarantee more about their outputs.

The names of arguments do not affect the relation between two function types, since argument binding is by the order of arguments only. Additionally, functions with different numbers of arguments (different **arity**) are considered incomparable.

The presence of function types that can be used as any other token results in what is commonly termed a higher-order type system. An example of the use of function tokens is given in Figure 2.44 and discussed in the sidebar on page 89.

14.3 Type Constraints in Actor Definitions

Section 12.4 introduces how to write actors in Java. In this section, we explain how to constrain types in the Java definition of an actor.

Prior to the execution of a model, during the setup phase, type constraints are gathered from all entities in the model (e.g., instances of TypedIOPort, TypedAtomicActor, or Parameter) that impose restrictions on their types. Actors can either set type constraints by storing them in the object instances of the concerning ports or parameters, or by setting them up using the `_customTypeConstraints` method of TypedAtomicActor.

A simple type constraint that is common to many actors is to ensure that the type of an output is greater than or equal to the type of a parameter. You can do so by putting the following statement in the constructor:

```
portName.setTypeAtLeast(parameterName);
```

or equivalently, by defining:

```
protected Set<Inequality> _customTypeConstraints() {
  Set<Inequality> result = new HashSet<Inequality>();
  result.add(new Inequality(parameterName.getTypeTerm(),
    portName.getTypeTerm()));
  return result;
}
```

This is called a **relative type constraint** because it constrains the type of one object relative to the type of another. Another form of relative type constraint forces two objects to have the same type, but without specifying what that type should be:

```
portName.setTypeSameAs(parameterName);
```

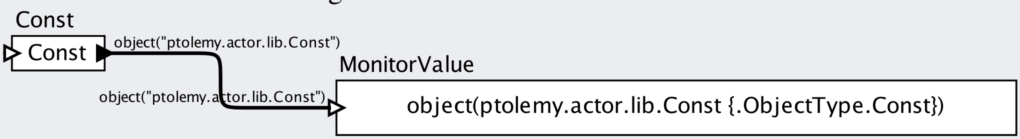
These constraints could be specified in reverse order,

```
parameterName.setTypeSameAs(portName);
```

which obviously means the same thing.

Sidebar: Object Types

The *object* type at the far left in Figure 14.4 is particularly powerful (and should be used with caution). A token of type *object* represents a Java object, such as a Ptolemy actor. Consider the following model:



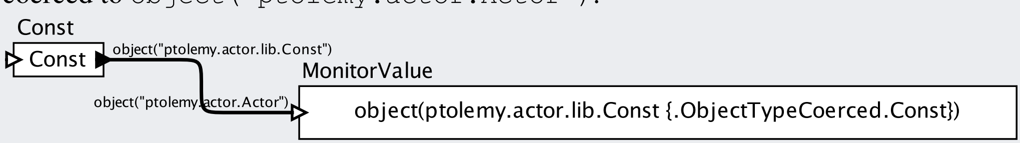
The *value* of the Const actor is set to `Const`, which evaluates to the Const actor itself. The Const actor's name is "`Const`," and the expression `Const` evaluates to the actor itself. Hence, the inferred type of the output port of Const is *object*, and its output will be the actor itself.

The *object* type is not one type, but an infinite number of types, as suggested by the double oval in Figure 14.4. There is a particular *object* type for every distinct Java class. The type of the output port above is `object("ptolemy.actor.lib.Const")`, because the Const actor is an instance of the Java class `ptolemy.actor.lib.Const`.

A type `object("A")` is less than `object("B")` if the Java class `A` is a subclass of Java class `B`. For example, `ptolemy.actor.lib.Const` implements the Java interface, `ptolemy.actor.Actor`, so

```
object("ptolemy.actor.lib.Const") <= object("ptolemy.actor.Actor")
```

In the following variant of the above model, the input port of the MonitorValue actor is coerced to `object("ptolemy.actor.Actor")`:



A similar conversion can be accomplished with the `cast` function by doing

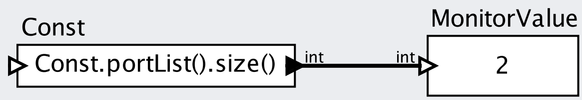
```
cast(object("ptolemy.actor.Actor"), Const)
```

The most general object type is *object* (without any argument). The pre-defined object token called `null` has this type.

Sidebar: Invoking Methods on Object Tokens

The expression language permits methods defined in the Java class of an object token (see sidebar on page 525). For example, in a model that contains an actor named C, the term `C` in an expression may refer to that actor.

Java methods may be invoked on the objects encapsulated in object tokens. For example, in the following model, the Const actor outputs the number of ports contained by the Const actor:



Sidebar: Petite and Unsigned Byte Data Types

The **petite** data type is used to represent real numbers in the range between −1.0 and 1.0 inclusive. It is used to emulate the behavior in certain specialized processors such as DSP processors (Digital Signal Processing), which sometimes use fixed-point arithmetic limited to this range of values. The *petite* type approximates this as a *double* limited to the range between −1.0 and 1.0 inclusive. In the expression language, a *petite* number is indicated by the suffix "p", and arithmetic operations saturate at −1.0 and 1 when results would lie outside this range. For example, using the expression evaluator, we get

```
>> 0.5p + 1.0p
1.0p
```

A data type that is sometimes useful for operating on raw data (e.g. packets arriving from a network) is the **unsigned byte**, designated as follows:

```
>> 1ub
1ub
>> -1ub
255ub
```

The same constraints can be expressed like this:

```
protected Set<Inequality> _customTypeConstraints() {
  result.add(new Inequality(parameterName.getTypeTerm(),
      portName.getTypeTerm());
  result.add(new Inequality(portName.getTypeTerm(),
      parameterName.getTypeTerm());
  return result;
}
```

The `_customTypeConstraints` method is particularly useful for actors that establish type constraints between ports that may be dynamically removed or added. For those actors, it is not safe to store the type constraints in the respective ports because constraints associated with no longer existing ports can persist and inadvertently cause type errors.

Another common type constraint is an **absolute type constraint**, which fixes the type of the port (i.e. making it **monomorphic** rather than polymorphic). This can specified as follows:

```
portName.setTypeEquals(BaseType.DOUBLE);
```

The above line declares that the port can only handle doubles. Figure 14.8 lists the type constants defined in the BaseType class. Another form of absolute type constraint imposes an upper bound on the type:

```
portName.setTypeAtMost(BaseType.COMPLEX);
```

which declares that any type that can be losslessly converted to ComplexToken is acceptable. By default, for any input port that has no declared type constraints, a default type constraint is automatically created that declares its type to be less than or equal to that of any output ports that have no declared type constraints. If there are input ports with no constraints, but no output ports lacking constraints, then those input ports will remain unconstrained. Conversely, if there are output ports with no constraints, but no input ports lacking constraints, then those output ports will remain be unconstrained. The latter is unacceptable, unless backward type inference is enabled. Default type constraints can be disabled by overriding the `_defaultTypeConstraints` method and having it return null.

A port can be declared to accept any token using following type constraint:

```
portName.setTypeAtMost(BaseType.GENERAL);
```

Example 14.10: An extension of Transformer of Figure 12.11 is shown in Figure 14.11. This SimplerScale is a simplified version of the Scale actor in the standard Ptolemy II library. This actor produces an output token on each firing with a value that is equal to a scaled version of the input. The actor is polymorphic in that it can support any token type that supports multiplication by the factor parameter. In the constructor, the output type is constrained to be at least as general as both the input and the factor parameter.

```
public class SimplerScale extends Transformer {
     public SimplerScale(CompositeEntity container,
               String name)
               throws NameDuplicationException,
               IllegalActionException {
          super(container, name);
          factor = new Parameter(this, "factor");
          factor.setExpression("1");
          // set the type constraints.
          output.setTypeAtLeast(input);
          output.setTypeAtLeast(factor);
     }
     public Parameter factor;
     public Object clone(Workspace workspace)
               throws CloneNotSupportedException {
          SimplerScale newObject = (SimplerScale)super.
                    clone(workspace);
          newObject.output.setTypeAtLeast(newObject.input);
          newObject.output.setTypeAtLeast(newObject.factor);
          return newObject;
     }
     public void fire() throws IllegalActionException {
          if (input.hasToken(0)) {
               Token in = input.get(0);
               Token factorToken = factor.getToken();
               Token result = factorToken.multiply(in);
               output.send(0, result);
          }
     }
}
```

Figure 14.11: Actor with non-trivial type constraints.

Notice in Figure 14.11 how the `fire` method uses `hasToken` to ensure that no output is produced if there is no input. Furthermore, only one token is consumed from each input channel, even if there is more than one token available. This is generally the behavior of domain polymorphic actors. Notice also how it uses the `multiply` method of the `Token` class. This method is polymorphic. Thus, this scale actor can operate on any token type that supports multiplication, including all the numeric types and matrices.

An awkward complication when customizing type constraints is illustrated by the `clone` method in lines 12-18. In order for the actor to work properly in actor-oriented classes, relative type constraints that are set up in the constructor have to be repeated in the `clone` method.

The `setTypeAtLeast`, `setTypeAtMost`, `setTypeEquals`, and `setTypeSameAs` methods are part of the Typeable interface, which is implemented by ports and parameters. The `setTypeAtMost` method is usually invoked on input ports to declare a requirement that *input* tokens must satisfy, while the `setTypeAtLeast` method is usually invoked on *output* ports to declare a guarantee of the type of the output. The methods `_customTypeConstraints` and `_defaultTypeConstraints` are part of the base class TypedAtomicActor, and its subclasses can override those methods to customize the type constraints they impose.

Example 14.11: The constraint that the type of an input port can be no greater than double might be declared as:

```
inputPort.setTypeAtMost(BaseType.DOUBLE);
```

Note that the argument to `setTypeAtMost` and `setTypeEquals` is a type, whereas the argument to `setTypeAtLeast` is a Typeable object. This reflects the common usage, where `setTypeAtLeast` is declaring a dependency on externally provided types, whereas both `setTypeAtMost` and `setTypeEquals` declare constraints on externally defined types. The forms of the type inequalities that are specifiable by these methods also ensures that type inference is efficient and that the result of type inference is deterministic.

More complex type constraints arise from structured types, such as arrays and records. To declare that a parameter is an array of doubles, use:

```
parameter.setTypeEquals(new ArrayType(BaseType.DOUBLE));
```

This declares that a parameter or a port has a particular array type. A more flexible parameter might be able to contain an array of any type. This is expresses as follows:

```
parameter.setTypeAtLeast(ArrayType.ARRAY_BOTTOM);
```

In a more elaborate example, we might constrain the type of an output port to be no less than the element type of the array contained by a parameter (or an input port):

```
outputPort.setTypeAtLeast(ArrayType.arrayOf(parameter));
```

To declare that an output has a type greater than or equal to that of the elements of an input (or parameter) array, use:

```
outputPort.setTypeAtLeast(ArrayType.elementType(inputPort));
```

The above code implicitly constrains the input port to have an array type, but does not constrain the element types of that array. The above kinds of constraints appear in source actors such as DiscreteClock and Pulse, ArrayToSequence and SequenceToArray. Examining the source code for those actors can be instructive.

Another common constraint is that an input port of an actor receives a record with unconstrained fields. This constraint can be declared using the following code:

```
inputPort.setTypeAtMost(RecordType.EMPTY_RECORD);
```

Suppose you have an output port that may produce records with arbitrary fields. The above construct will not be sufficient since it does not declare any lower bound on the type, so at run time, the type will not be resolved to something useful. Instead, do:

```
outputPort.setTypeEquals(BaseType.RECORD);
```

This forces the type to resolve to the empty record. Any record with fields is a subtype of the empty record type, so this effectively declares the output to produce any record. Alternatively, enabling backward type inference allows the element type of the record to be inferred from the type constraints imposed by downstream actors.

To declare that a parameter can have a value that is any record, you can do:

```
param.setTypeAtMost(BaseType.RECORD);
```

but you will also need to specify a value for the parameter so that the type resolves to something concrete. To give a default value that is an empty record you can do:

```
param.setToken(RecordToken.EMPTY_RECORD);
```

Two of the types, *matrix* and *scalar*, are union types. This means that an instance of this type can be any of the types immediately below them in the lattice. An actor may, for example, declare that an input port must be of type no greater than scalar:

```
inputPort.setTypeAtMost(BaseType.SCALAR);
```

In this case, inputs of any type immediately below scalar in the type lattice will not be converted, except that the type of the input tokens will be reported as scalar. This is useful, for example, in actors that need to compare tokens, such as the Limiter actor. The `fire` method of that actor contains the code

```
if (input.hasToken(0)) {
    ScalarToken in = (ScalarToken) input.get(0);
    if ((in.isLessThan((ScalarToken) bottom.getToken()))
            .booleanValue()) {
        output.send(0, bottom.getToken());
    } else if ((in.isGreaterThan((ScalarToken) top.getToken()))
            .booleanValue()) {
        output.send(0, top.getToken());
    } else {
        output.send(0, in);
    }
}
```

This code relies on input port *in* and parameter *bottom* being declared to be at most scalar type, and ScalarToken being a base class for every token with type immediately below scalar. It then uses comparison methods defined in the ScalarToken class.

Type constraints in actors can get much more sophisticated than what we describe here. As always, the source code (and its extensive documentation) is the ultimate reference.

14.4 Summary

Ptolemy II includes a sophisticated type system that performs inference and checks for errors. It uses an efficient algorithm given by Rehof and Mogensen (1999), who prove that their algorithm has complexity that is linear in the number of occurrences of symbols in the type constraints. As a consequence, the algorithm scales well to large models. Most of the time, the type system makes it unnecessary for the builder of models to think about types. At it makes it easy to define actors that operate on a multiplicity of types, as most of the actors in the standard library do.

Sidebar: Monotonic Functions in Type Constraints

More sophisticated type constraints can be expressed using a **monotonic function** on the left-hand side of an inequality. A monotonic function f preserves the order of its arguments; that is

$$x_1 \leq x_2 \Rightarrow f(x_1) \leq f(x_2). \tag{14.2}$$

Using a monotonic function, it is possible to define a type that is dependent on other types in complicated ways. For example, the actor RecordDisassembler sets up a type constraint that forces each field of its input record to be of the same type as the output port with the same name as the field.

A monotonic function is specified by subclassing the abstract class MonotonicFunction and implementing the methods `getVariables` and `getValue`. The `getVariables` method returns the variables that the function takes as arguments. The `getValue` method returns the result of applying the function to the current value of the variables it depends on.

For example, the ConstructAssociativeType class subclasses MonotonicFunction. The variables returned by `getVariables` are the variables holding the types of a list of ports, such as the output ports of a RecordDisassembler actor. The `getValue` method returns a record type with fields matching the names of those ports and types matching the types of those ports.

It should be noted that the base class MonotonicFunction does not guarantee that its subclasses indeed behave monotonically. If a function that is not actually monotonic is used in a type constraint, then type resolution is no longer guaranteed to yield a unique result. The result may depend on the order in which constraints are applied.

15

Ontologies

Patricia Derler, Elizabeth A. Latronico, Edward A. Lee, Man-Kit Leung, Ben Lickly, Charles Shelton

Contents

An **ontology** in information science refers to an explicit organization of knowledge. An ontology can be organized into a graph as a set of **concepts** and the relations between those concepts. By constructing an ontology over a specific domain, a user is formalizing information of that domain in a way that can be shared with others. Models can have annotations added to them that express how they are used with respect to an ontology. Ontology-based annotations are a form of model documentation. Like type signatures, they can express the intended use of a model, but with respect to the domain of the ontology rather than to the type system.

A **static analysis** of a program or model is a check that can be run at compile time. Ptolemy II's type checker (see Chapter 14) is one example of a static analysis. It infers the data types used throughout a model and checks for consistency. In fact, the Ptolemy II type system is an ontology. It is an organization of knowledge about the data that a model operates on. The ontology checker described in this chapter also performs inference based on the annotations, and then checks consistency. But it is not constrained to checking data types. Instead, ontologies can be used to express static analyses from a variety of user-defined domains, including, for example:

- **units checking**: determining whether the units of data are consistent;
- **constant analysis**: determining what data in a model is constant, and what data varies in time;
- **taint analysis**: determining whether values in a data stream are influenced by an untrusted source; and
- **semantics checking**: determining whether the meaning of data produced by one component is consistent with the meaning assumed by another component that uses the data.

Such analyses can expose a variety of modeling errors.

Example 15.1: A portion of a model of a multi-tank fuel system in an aircraft (Moir and Seabridge, 2008) is shown in Figure 15.1. This model has three actors, where the ports are labeled with names that suggest the intended meaning and units of the data that are exchanged between actors. The model has three types of errors. It has units errors, where for example one component gives the level of a tank in gallons to a component that assumes that the level is being given in kilograms. (The latter is often a better choice, since amount of fuel in gallons varies with temperature, whereas the amount in kilograms does not.) It also has semantics

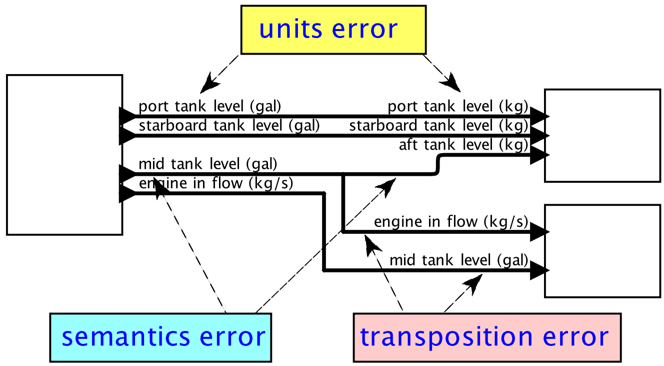


Figure 15.1: An illustration of some of the sorts of errors that can be caught by an ontology system.

errors, where a component gives the level of mid tank to a component that assumes it is seeing the level of the aft tank. And it has a transposition error, where a level and a flow are exchanged.

Such modeling errors are extremely easy to make and can have devastating consequences. This chapter gives an overview of how to construct ontologies and use them to prevent such errors.

15.1 Creating and Using Ontologies

The ontologies package provides an analysis that can be run on top of an existing model, so the first step is to create a Ptolemy II model on which we can run our analysis. In this section, we use a rather trivial model and a rather trivial ontology to illustrate the mechanics of construction of an ontology and the use of a solver. We will then illustrate a less trivial ontology that is practical and useful for catching certain kinds of modeling errors.

Example 15.2: Figure 15.2, shows a simple model with both constant and non-constant actors. The constant actors produce a sequence of output values that are all the same. For this model, we will show how to create a simple analysis that checks which signals in the model are constant. To do this, we will first define an ontology that distinguishes the concept of "constant" from "non constant." We will then define constraints for actors used by the model, and finally, we will invoke the solver.

Sidebar: Background on the Ontology Framework

The approach to ontologies described in this chapter was first given by Leung et al. (2009). They build on the theory of Hindley-Milner type systems (Milner, 1978), the efficient inference algorithm of Rehof and Mogensen Rehof and Mogensen (1996), the implementation of this algorithm in Ptolemy II (Xiong, 2002), and the application of similar mathematical foundations to formal concept analysis (Ganter and Wille, 1998).

An interesting extension of this basic mechanism, devised by Feng (2009), uses ontologies to guide model-based **model transformation**, where a Ptolemy II model modifies the structure of another Ptolemy II model. For example, the constant analysis described in Section 15.1 can guide a model optimization that replaces all constant subsystems with a Const actor. Also, Lickly et al. (2011) show how an infinite lattice can be used to not just infer that a signal is constant, but also to infer its value. Lickly et al. (2011) also show various other ways to use infinite lattices, including unit systems. They also show how ontologies work with structured types such as records.

The **Web Ontology Language** (**OWL**) is a widely-used family of languages endorsed by the World Wide Web Consortium (**W3C**) for specifying ontologies. OWL ontologies, like ours, form a partial order with a top and bottom element, but unlike ours, they are not constrained to be a lattice. Hence, the efficient inference algorithm of Rehof and Mogensen cannot always be applied. Nevertheless, a very useful extension of the mechanisms described in this chapter would be to export and import OWL ontologies. The **Eclipse Modeling Framework** (**EMF**) also specifies ontologies through the notion of classes and subclasses. Many Eclipse-based tools have been developed supporting it, so again it would be useful to build bridges.

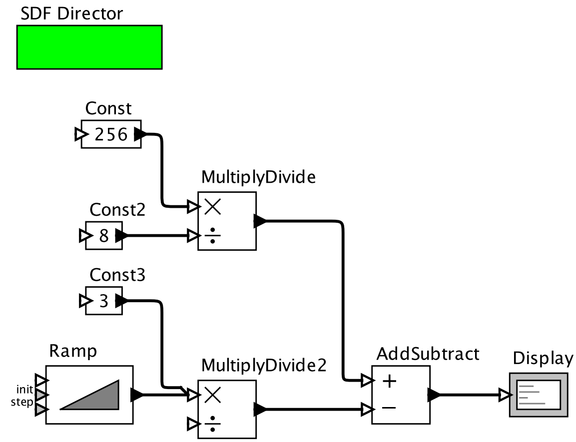


Figure 15.2: A simple Ptolemy model made of constant and non-constant actors and containing an ontology solver that can determine which signals are constant.

15.1.1 Creating an Ontology

In order to create the analysis, the first step will be to add the solver that will perform our analysis. In this case, we will drag in the **LatticeOntologySolver** actor to our model, as shown in Figure 15.3. This is where we will add all of the details of how our analysis works. These include the lattice that represents the concepts that we are interested in, and the constraints that actors impose on those concepts. In our case, the lattice will specify whether a signal is constant or not, and the constraints will provide information about which actors produce constant or non-constant signals.

As shown in Figure 15.3, if you open the LatticeOntologySolver, you get an editor with a customized library for building analyses. At a minimum, an analysis requires an ontology. Figure 15.4 illustrates the steps in constructing one. First, drag into the blank editor an **Ontology**. Open the ontology and drag **Concept**s into it from the library provided by the ontology editor.

First, we should assign meaningful names to our concepts. In Figure 15.5, we have renamed Top to NonConstant, Bottom to Unused, and Concept to Constant. We have also edited the parameters to the NonConstant concept to change its color to a light red, and to check the *isAcceptable* parameter, which visually removes the bold outline. These con-

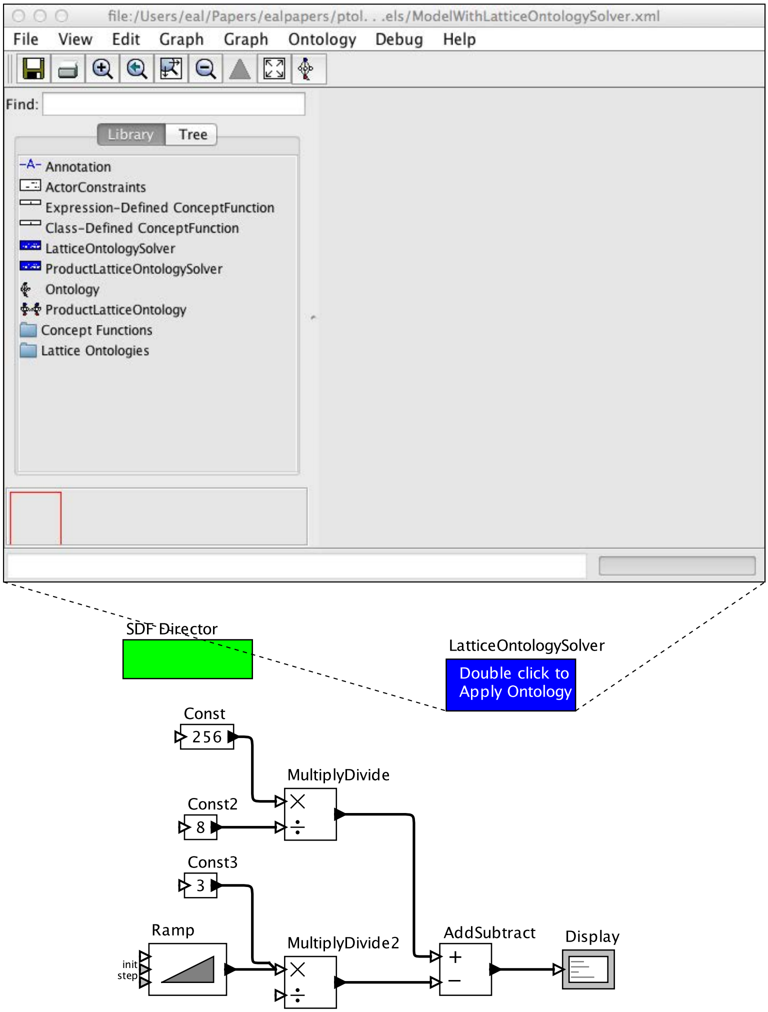


Figure 15.3: A model with a blank ontology solver.

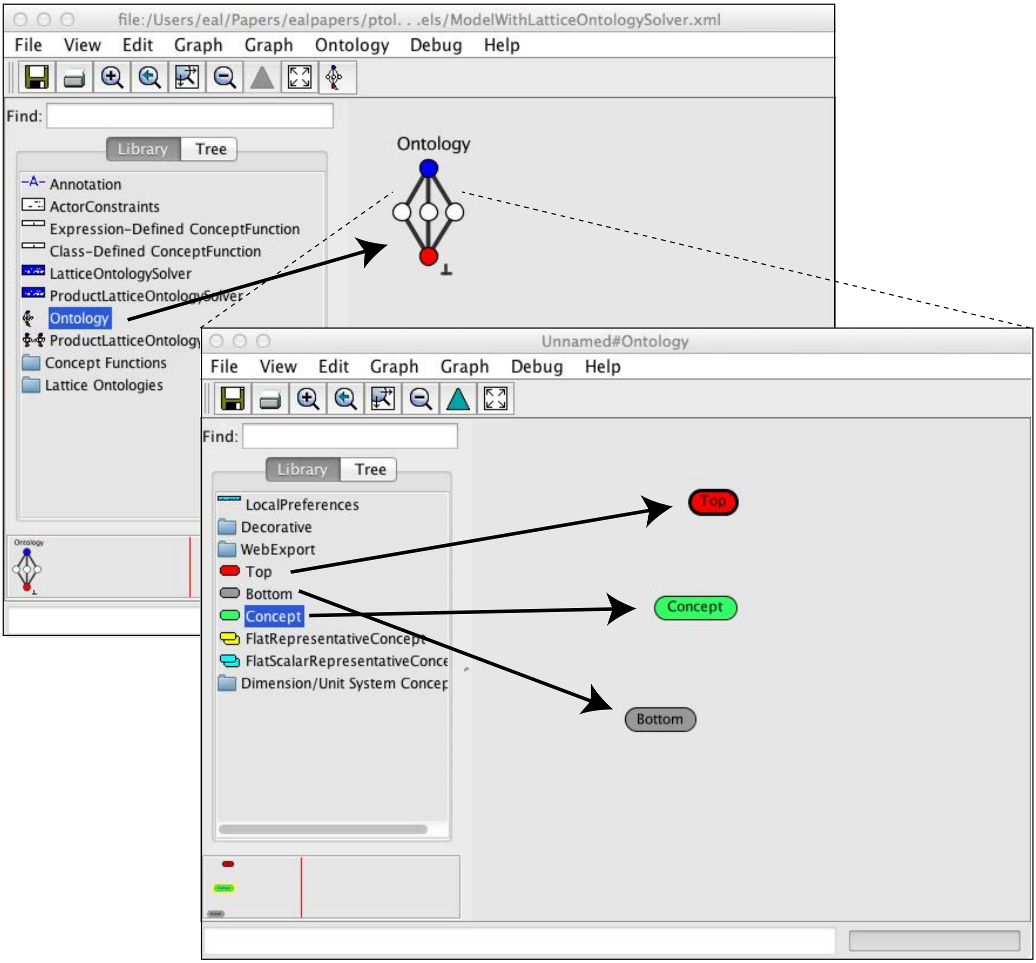


Figure 15.4: Steps in the construction of an ontology.

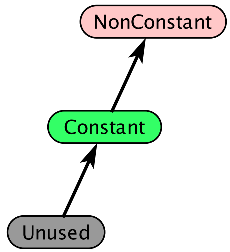


Figure 15.5: The lattice used for constant analysis.

cepts will be associated with ports in our model, and when *isAcceptable* is unchecked (the outline on the concept is bold), then it is an error in the model for any port to be associated with the concept. In our case, it is not an error for a port to be NonConstant, so we make the value of this parameter true.

Here, we include not only the concepts of Const and NonConst, but also explicitly include a notion of Unused. This concept will be associated with ports that are simply not participating in the analysis. We will use the NonConstant concept to represent any signal that may or may not be constant, so a better name might be PotentiallyNonConst or NotNecessarilyConst.

The last step in building an ontology is to establish relationships between the concepts. Do this by holding the control key (command key on a Mac) and dragging from the lower concept to the higher one. In this case, the relation between Constant and NonConstant is a **generalization relation**. The relations define an order relation between concepts (see sidebar on page 513), where if the arrow goes from concept a to concept b, then $a \leq b$. In this case, Constant $\leq$ NonConstant, and Unused $\leq$ Constant.

The meanings of these relations can vary with ontologies, but it is common for them to represent subclassing, a subset relation, or as an "is a" relation. In the subset interpretation, a concept represents a set, and concept A is less than another B if everything in A is also in B. The "is a" relation interpretation is similar, but doesn't require a formal notion of a set. E.g., if the concept A represents "dog" and the concept B represents "mammal," then it is reasonable to establish on ontology where $A \leq B$ under the "is a" interpretation. A dog is a mammal.

Example 15.3: In Figure 15.5, NonConstant represents anything that may or may not be constant. Hence, something that is actually constant “is a” NonConstant.

The interpretation of the bottom element, Unused in the example, is a bit trickier. Presumably, anything that makes no assertion about whether it is constant or not “is a” NonConstant. Indeed, since the order relation is transitive, this statement is implied by Figure 15.5. But why make a distinction between Unused and NonConstant? We could build an ontology that makes no such distinction, but in such an ontology, the inference engine would infer Constant for any port that imposes no constraints at all. This is probably an error. The bottom element, therefore, is used to indicate that the inference engine has no usable information at all. If a port resolves to Unused, then it is not playing the game. If we wish to force all ports to play the game, then we should set the *isAcceptable* parameter of Unused to false.

The Ptolemy II type system is an ontology, as shown in Figure 14.4. Here, the order relations represent subtyping, which in the case of Ptolemy II is based on the principle of lossless type conversion.

With the concepts as nodes and the relations as edges, the ontology forms a mathematical graph. The structure of this graph is required to conform with that a mathematical lattice (see sidebar on page 513). Specifically, the structure is a lattice if given any two concepts in the ontology, these two concepts have a least upper bound and a greatest lower bound. In this case, conformance is trivial. For example, the least upper bound of Constant and NonConstant is NonConstant. The greatest lower bound is Constant. But it is easy to construct an ontology that is not a lattice.

Example 15.4: Figure 15.6 shows an ontology that is not a lattice. Consider the two concepts, Dog and Cat. There are three concepts that are upper bounds for these two concepts, namely Pet, Mammal, and Animal. But of those three, there is no least upper bound. A least upper bound must be less than all other upper bounds.

If you build an ontology that is not a lattice, then upon invoking the solver, you will get an error message.

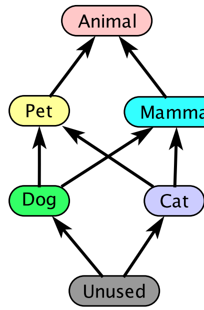


Figure 15.6: An ontology that is not a lattice.

15.1.2 Creating Constraints

Now that we have an ontology, to use it, we need to create constraints on the association between objects in the model and the concepts in the ontology. The most straightforward way to do this is with manual annotations in the model. With large models, however, this technique does not scale well. A more scalable technique is to define constraints that apply broadly to all actors of a class, and to specify default constraints that apply when no other constraints are specified. We begin with the manual annotations, because they are conceptually simplest.

Manual Annotations

The simplest way to relate objects in a model to concepts in the ontology is through manual annotations. Manual annotations take the form of inequality constraints. To create such constraints, find the **Constraint** annotation in the `MoreLibraries→Ontologies` library, and drag it into the model. Then specify an inequality constraint of the form `object >= concept` or `concept >= object`, where `object` is an object in the model (a port or parameter) and `concept` is a concept in the ontology.

Example 15.5: Figure 15.7 elaborates the model of Figure 15.3 by adding four annotations. Each of these has the form

```
port >= concept.
```

Specifically, the output ports of each Const actor are constrained to be greater than or equal to Constant, whereas the output port of the Ramp actor is constrained to be greater than or equal to NonConstant. The latter constraint, in effect, forces the port to NonConstant, since there is nothing greater than NonConstant in the ontology. The former constraints could be elaborated to force the ports to resolve to Constant by adding the complementary inequality, like

```
Constant >= Const.output.
```

However, this additional constraint is unnecessary. The solver will find the *least* solution that satisfies all the constraints, so in this case, since there are no other constraints on the output ports of the Const actors, they will resolve to Constant anyway.

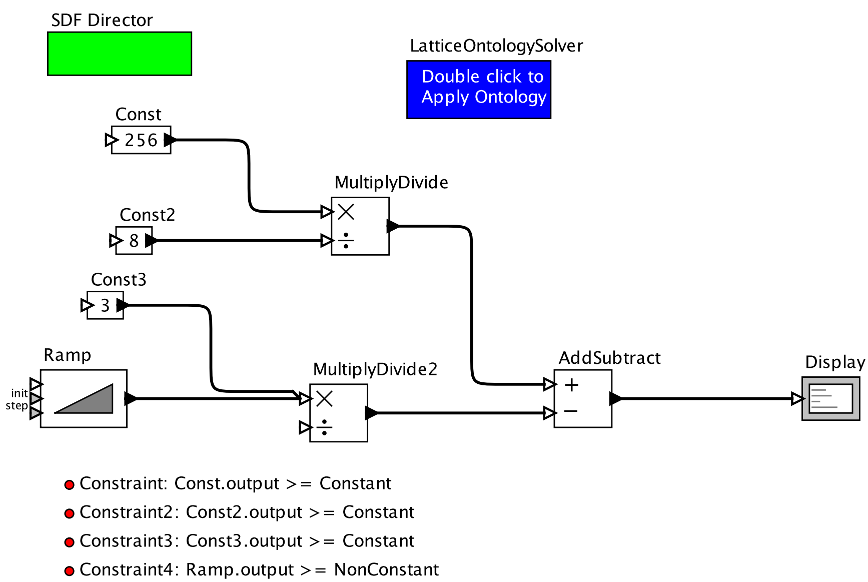


Figure 15.7: A model with manual constraints, using the ontology in Figure 15.5.

Once we have created all the constraints needed in our model, the next step is to run the analysis. The analysis can be run by right-clicking on the LatticeOntologySolver and selecting `Resolve Concepts`, as shown in Figure 15.8. Because this is a common operation, double-clicking on the LatticeOntologySolver will also run the analysis.

Example 15.6: Figure 15.8 includes the results of running the analysis. Notice that each port is annotated with the concept that it is now associated with. In addition, the port is highlighted with the same color specified for the concept in the ontology of Figure 15.5. The output ports of the Const actors, as expected, have resolved to Constant, and the the output of the Ramp has resolved to NonConstant. But more interestingly, downstream ports have also resolved in a reasonable way. The output of the MultiplyDivide is Constant (because both its inputs are Constant), and the the output of MultiplyDivide2 is NonConstant (because one of its inputs is NonConstant). These results are due to default constraints associated with actors, which as explained below, can be customized in an ontology.

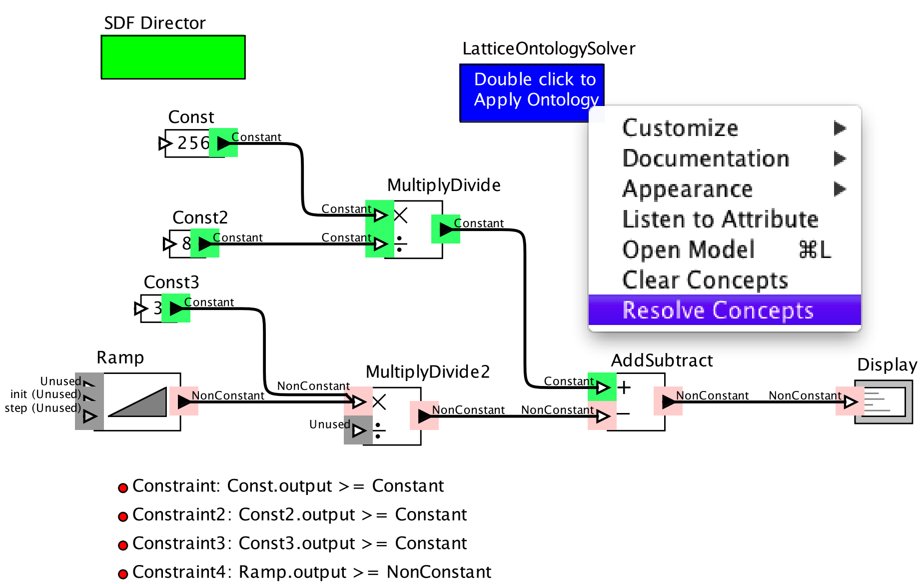


Figure 15.8: Running an analysis. [online]

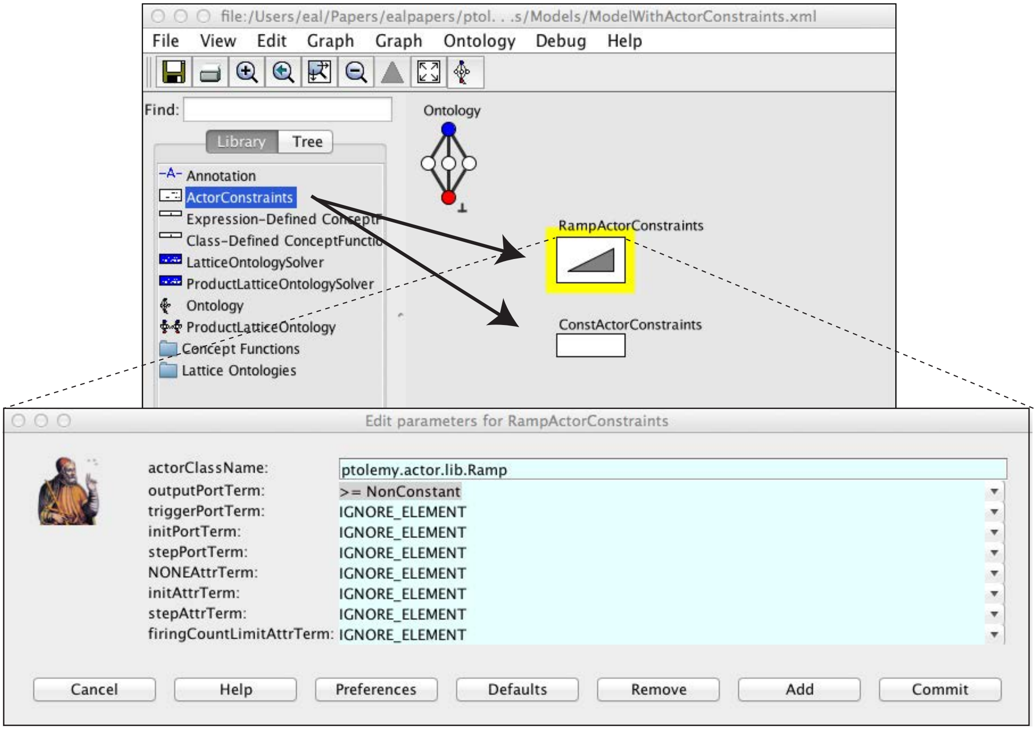


Figure 15.9: Adding actor constraints to an ontology.

Actor-level Constraints

The manual annotations in Figure 15.7 could become very tedious to enter in a large model. Fortunately, there is a convenient shortcut. As part of defining an ontology, we can specify constraints that will be associated with all instances of a particular actor class. Figure 15.9 shows how to do this. Inside the LatticeOntologySolver, we add one instance of **ActorConstraints** for each actor for which we want to specify default constraints.

Setting the *actorClassName* parameter of the ActorConstraints to the fully qualified class name of the actors to be constrained causes both the name and the icon of the instance of ActorConstraints to change to match the class of actors that it will constrain.

Example 15.7: Figure 15.9 shows two instances of ActorConstraints that have been dragged into the LatticeConstraintsSolver. The top one has the *actorClassName* parameter set to `ptolemy.actor.lib.Ramp`, the class name of the Ramp actor. (To see the class name of an actor, linger over it in Vergil.)

Once the class name has been set, upon re-opening the parameter dialog, a new set of parameters will have appeared, one for each port belonging to instances of the actor class, and one for each parameter of the actor.

Example 15.8: Figure 15.9 shows these parameters for the Ramp actor. Here we can see that constraints have been set that will force the *output* port to be greater than or equal to NonConstant, and that all other constraints have been set to `IGNORE_ELEMENT`. Once a similar constraint has been added to the ConstActorConstraints component, constraining the output ports of instances of Const to be `>= Constant`, then the four constraints at the bottom of Figure 15.7 are no longer necessary. They could be removed, and the results of the analysis will be the same as in Figure 15.8.

The constraints associated with a port or parameter can take any one of the following forms:

- `NO_CONSTRAINTS` (the default)
- `IGNORE_ELEMENT`
- >= *concept*
- <= *concept*
- == *concept*

By default, the ActorConstraints actor will set `NO_CONSTRAINTS` as the constraint of each port and parameter. This means that instances of the actor allow their ports and parameters to be associated with any concept. In this ontology, the association will always result in Unused, so we could equally well have left all the constraints at the default `NO_CONSTRAINTS`, except those for the output ports.

Default Constraints

Notice in Figure 15.8 that not only have the outputs of the Const and Ramp actors resolved to the appropriate concepts, but so have those of downstream actors, including MultiplyDivide and AddSubtract. How did this come about?

With respect to this analysis, both the MultiplyDivide actor and the AddSubtract actors behave the same way. Given only constant inputs, they produce a constant output, but given any non-constant input, they produce a (potentially) non-constant output. In terms of the ontology lattice (Figure 15.5), the output concept will always be greater than or equal to all of the input concepts. In other words, the output should be constrained to be greater than or equal to the least upper bound of all the inputs. It turns out that this type of inference behavior is a very common one. For this reason it is the default constraint for all actors. Actors that do not have explicit constraints will inherit this default constraint. This means that we can omit specifying any more ActorConstraints, since the global default constraint is sufficient for all of our remaining actors.

15.1.3 Abstract Interpretation

Identifying signals as either constant or non-constant is a particularly simple form of **abstract interpretation** (Cousot and Cousot, 1977). In abstract interpretation, instead of actually computing the values of variables, we classify the variables in more abstract terms, such as whether their values vary over time. Ontologies can be used to systematically apply more sophisticated abstractions, determining for example whether variables are always positive, negative, or zero. This can be used to expose errors in designs, and also to optimize design by removing unnecessary computations.

> **Example 15.9:** The model in Figure 15.10 produces a constant stream of zeros. Were we to apply the same Constant-NonConstant analysis as before to this model, we would be able to determine that the output is constant. But it would not tell us that the output is a constant stream of *zeros*.

To address this problem, we can use the more elaborate ontology shown in Figure 15.11. This ontology is used to abstract numeric variables as positive, negative, or zero-valued

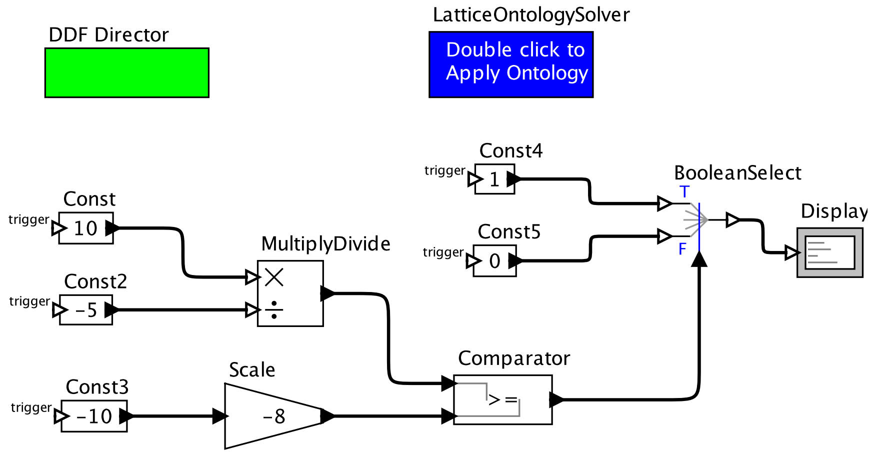


Figure 15.10: A model that produces only zeros.

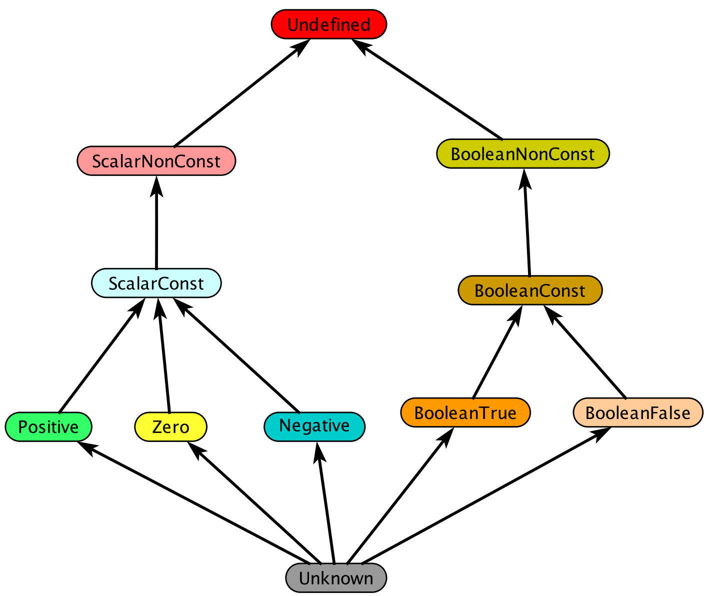


Figure 15.11: An ontology that tracks the sign of numeric variables and the value of boolean variables.

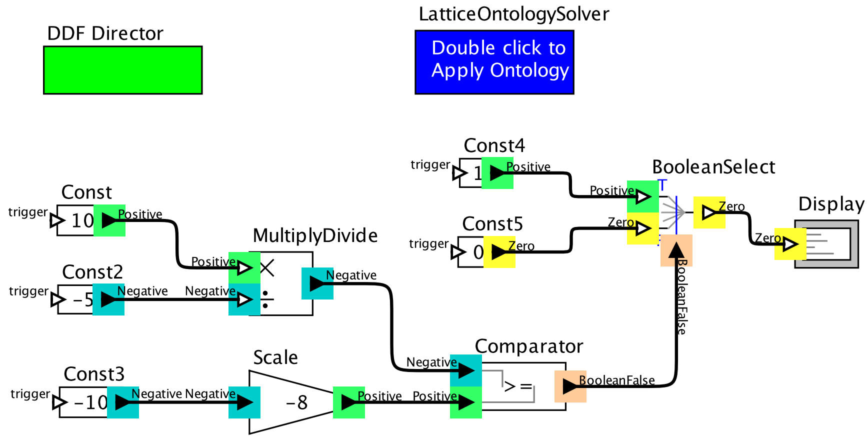


Figure 15.12: Result of applying an analysis based on the ontology in Figure 15.11 to the model in Figure 15.10. [online]

numbers, in addition to whether they are constant or non constant. For boolean-valued variables, if the variable is constant, then it also tracks whether the constant is true or false. With appropriate actor constraints, this ontology can be used to produce the result shown in Figure 15.12, which determines that the output is a constant stream of zeros.

15.2 Finding and Minimizing Errors

In this section, we discuss how to use an analysis to identify errors, and discuss tools available to help in correcting the errors. For this discussion, let us consider the ontology in Figure 15.13. This ontology models physical dimensions. Specifically, it distinguishes the concepts of time, position, velocity, and acceleration. We will show that with appropriate actor constraints, it can use properties of these dimensions in inference.

Example 15.10: Figure 15.14 shows a piece of a larger model that shows interesting inference of dimensions. This is a model of a car with a **cruise control**, where the input is a desired speed, and the outputs are acceleration, speed, and position. In this model, when velocity is divided by time, the result is acceleration. When acceleration is integrated over time, the result is velocity. When velocity is integrated over time, the result is position. This analysis relies on actor constraints for arithmetic operations and integrators.

The ontology in Figure 15.13 explicitly includes the concepts Unknown and Conflict. The difference between Unknown and Conflict is subtle and deserves mention. Conflict

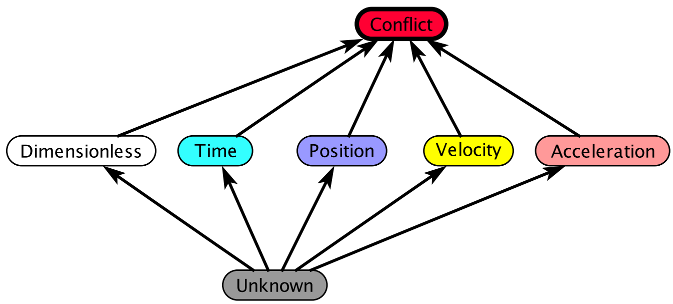


Figure 15.13: An ontology for analyzing physical dimensions.

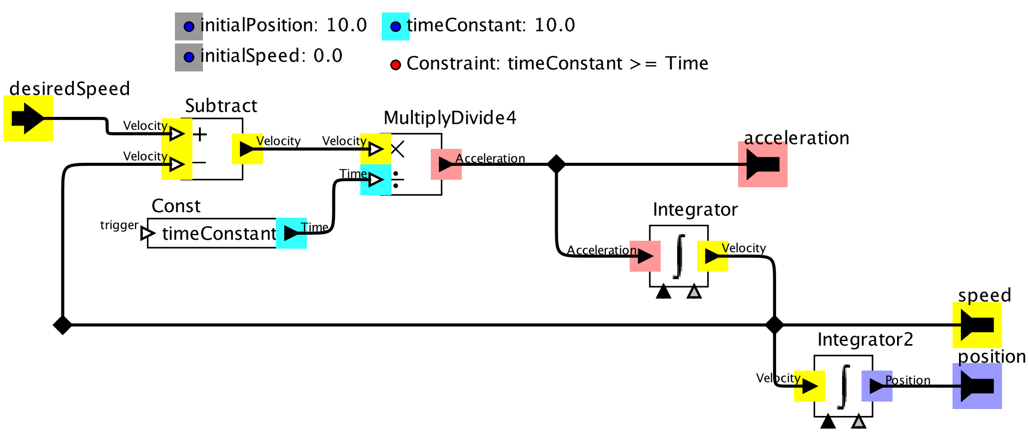


Figure 15.14: A piece of a larger model that shows interesting inference of dimensions. [online]

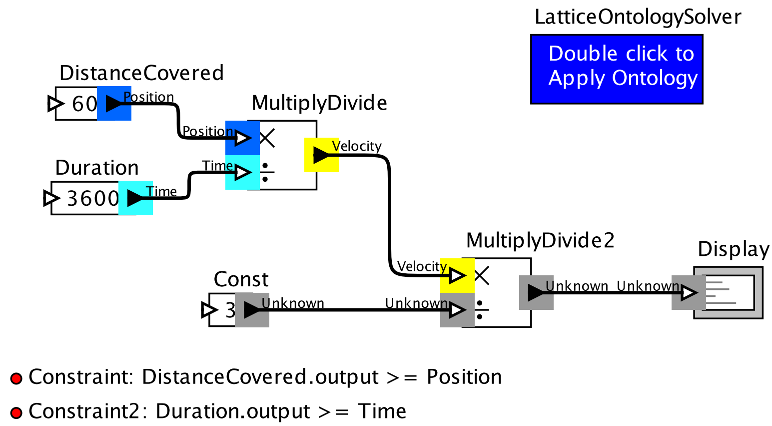


Figure 15.15: A model analyzing physical dimensions with too few annotations. Running the analysis reveals where additional annotations are needed. [online]

represents a situation where the analysis has detected that a given signal cannot have any of the dimensions; thus, the analysis has found an error in the model due to conflicting use of dimensions. For this reason, in this ontology, the *isAcceptable* parameter of Conflict is set to false, resulting in a bold outline in Figure 15.13. In addition, if any port resolves to Conflict, then running the analysis will report an error.

In the ontology in Figure 15.13, Unknown represents a case where the analysis cannot say conclusively anything about the given signal; this means that the property being analyzed cannot be proved with the given assumptions. Unknown in this case plays a similar role as Unused in Figure 15.5. When a port resolves to Unknown, this can point to there not being enough constraints in the model. This may or may not be an error, so *isAcceptable* is left at its default value of true.

Example 15.11: An example of an underconstrained model is shown in Figure 15.15. Here, the model divides a velocity by a time to get an acceleration, but the constraint specifying the dimension of the time value produced by the Const actor has been omitted. When running the analysis, we get results shown the figure. The lack of information propagates throughout the model. This can be fixed, of course, by adding a manual annotation to the model as shown in Figure 15.16.

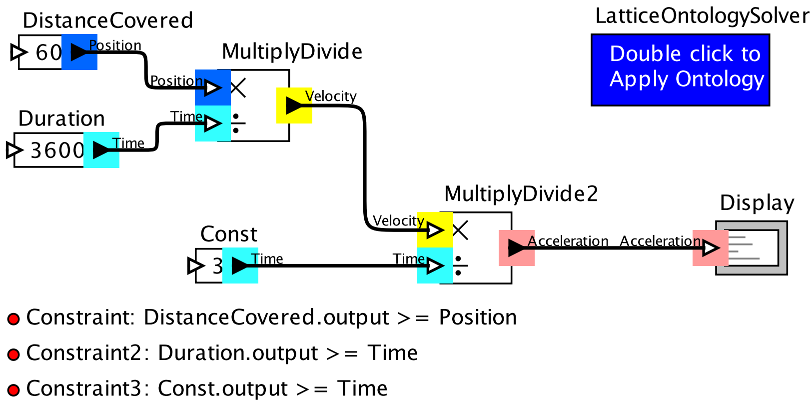


Figure 15.16: Adding an additional constraint allows for a complete analysis. [online]

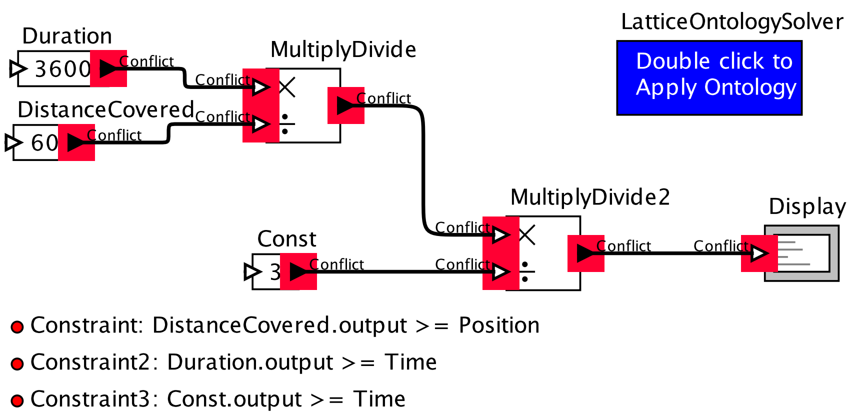


Figure 15.17: An example model with conflicting dimensions due to an error in the model. Running the analysis on this model shows that the whole model is in conflict. [online]

In general, overconstrained models can be more difficult to deal with.

Example 15.12: An example of an overconstrained model is shown in Figure 15.17. Here, the model builder has incorrectly divided a time by a position, where presumably the reverse operation was intended. The result is conflicts throughout the model.

The conflict in the previous example arises because of the ActorConstraints that are defined for the MultiplyDivide actor as part of the ontology (see Figure 15.9). Specifically, the ontology gives for the *outputPortTerm* the following expression:

```
>=
(multiply == Unknown || divide == Unknown) ? Unknown :
(multiply == Position && divide == Time) ? Velocity :
(multiply == Velocity && divide == Time) ? Acceleration :
(multiply == Position && divide == Velocity) ? Time :
(multiply == Velocity && divide == Acceleration) ? Time :
(divide == Dimensionless) ? multiply :
Conflict
```

This constrains the output concept as a function of the input concepts. If the *multiply* input is Position and the *divide* input is Time, for example, this expression evaluates to Velocity. If *divide* input is Dimensionless, then it evaluates to whatever *multiply* is. If *multiply* is Time and *divide* is Position, it evaluates to Conflict.

In the example of Figure 15.17, notice that conflicts propagate upstream as well as downstream. In this example, we have set the *solverStrategy* of the LatticeOntologySolver to `bidirectional`.* The default value of this parameter is `forward`, which means that when there is a connection from an output port to an input port, then the concept associated with the input port is constrained to be greater than or equal to the concept associated with the output port. When the parameter value is set to `bidirectional`, then the two concepts are required to be equal. With the `bidirectional` setting, constraints propagate upstream in a model just as easily as downstream. So although this is reasonable to

*Since double clicking on the LatticeOntologySolver runs the analysis, double clicking cannot be used to access the parameters. Instead, hold the alt key while you click to access the parameters.

do for dimension analysis, it makes it difficult to see which part of the model is causing the error.

By default, the solver finds the least solution that satisfies all the constraints.[†] Hence, the way that our analysis deals with conflicting information is to promote signals to the least upper bound of the conflicting concepts (or greatest lower bound when computing greatest fixed points).

The analysis is able to correctly detect the error, as shown in Figure 15.17, but there is a problem. Unlike the underconstrained case in which the error was relatively contained, in this case Conflict propagates throughout the model, making it very difficult to see where the source of the error is. In this simple example it is not too difficult to find the error, but as models grow, so does the difficulty in tracking down the source of an error of this type.

In order to address this problem, we have introduced an error minimization algorithm. This algorithm is implemented in the **DeltaConstraintSolver** actor, which is a subclass of LatticeOntologySolver. It can be found in the same `MoreLibraries→Ontologies` library. The DeltaConstraintSolver offers a `Resolve Conflicts` item in the context menu in addition to the `Resolve Concepts` that we used before.

Given a model in which at least one signal resolves to an unacceptable solution, the algorithm realized by `Resolve Conflicts` finds a subset of those constraints with the property that removing any additional constraint does not produce any error. Highlighting the result of running the analysis with these constraints will highlight only a subsection of the model that contains an error. In practice, this means in contrast to the full analysis, where the highlighted result shows many errors throughout the model as shown in Figure 15.17, the modified algorithm highlights only a single path through the model that contains an error, as shown in Figure 15.18. In this example, only the Duration, DistanceCovered, and MultiplyDivide actors (each of which is an instance of Const) are highlighted, since they are sufficient to cause the error. The unhighlighted actors (in this case the Const, MultiplyDivide2, and Display) are not necessary to cause the error, so the model builder can ignore them in trying to find the cause of the error.

In this case, the port that was highlighted with the erroneous concept (Conflict) was the location of the error, but in general, this need not be the case. The only guarantee that is made is that there is an error somewhere in the path of all highlighted signals. This

[†]This can be changed by changing the *solvingFixedPoint* parameter of the LatticeOntologySolver from the default `least` to `greatest`. In that case, the solver will find the highest solution in the lattice that satisfies all the constraints.

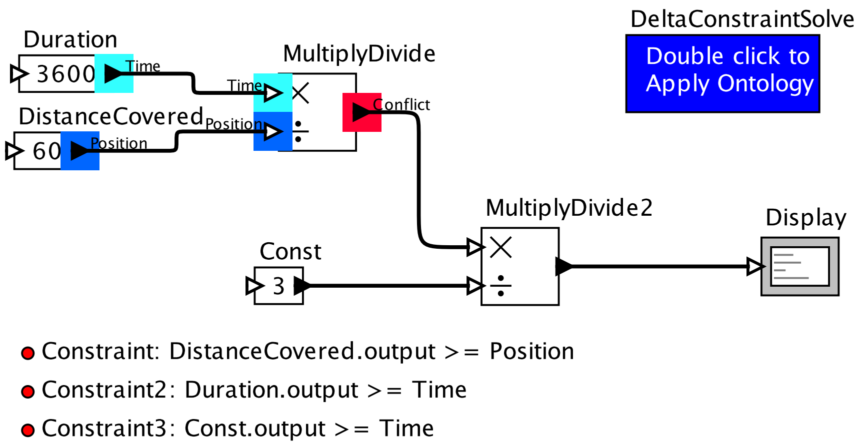


Figure 15.18: With our error minimization algorithm, finding errors is easier. [online]

means that in general, all of the signals that are highlighted may need to be checked in order to find the source of the error. The gain of this technique comes from the many unhighlighted actors in the model that can be completely ignored.

15.3 Creating a Unit System

In the previous section, we saw an analysis that checked that the dimensions of a system were being used in a consistent way, to avoid errors such as integrating a velocity and expecting to get an acceleration value out. A similar but more insidious type of error is when two signals have the same dimensions, but different units, such as feet vs. meters, or pounds vs. kilograms. Since this is a more subtle problem – results can deviate by only a small scaling factor – it is even more desirable to check these types of properties automatically. Section 13.7 describes a built-in unit system provided in Ptolemy II. But a units system is just another ontology, and the ontology framework can be used to create specialized units systems.

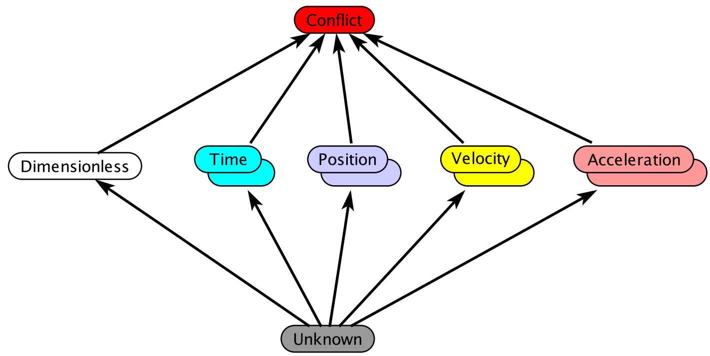


Figure 15.19: A unit system for physical units.

15.3.1 What are Units?

In order to discuss what units are, we first need to discuss how units differ from one another. There are two ways that units can differ. They can be different measures of the same quantity, like feet and meters, or they can be measures of fundamentally different quantities, like feet and seconds. Differences of the second type are exactly what is captured in the dimension ontology, so our approach to units will be similar, but extended to deal with different units within a single dimension.

Ptolemy II supports creating a **units system ontology** with the same freedom as other ontologies. Users can choose the units and dimensions that are appropriate to analyze the model at hand, and thus make the ontology only as complicated as it needs to be to perform the desired analysis. Since we have already discussed a physical dimension ontology that had concepts for Position, Velocity, Acceleration, etc., we will continue to use an example unit ontology where the units are drawn from the same dimensions, as shown in Figure 15.19. However, instead of building our ontology using only simple instances of Concept, we use instances of **DerivedDimension** and **Dimensionless**. These are concepts specialized to support units, and are selected from the `Dimension/Unit System Concepts` sublibrary provided by the ontology editor (see Figure 15.4).

As shown in Figure 15.19, DerivedDimension has a slightly different icon, a double oval, suggestive that it is in fact a **representative concept**, a single object that stands in for a family of concepts. DerivedDimension has some parameters that we set and parameters that we must add to build a unit system. We outline what this looks like here, but to build a unit system from scratch, you will need to refer to the documentation.

15.3.2 Base and Derived Dimensions

The dimensions from which units are drawn are split into two types, **base dimensions**, and **derived dimensions**. Base dimensions are dimensions that cannot be broken down into any smaller components, such as Time, whereas derived dimensions can be expressed in terms of other dimensions. For example, Acceleration can be expressed in terms of Position and Time. Note that there is no restriction on which dimensions can be base dimensions or derived dimensions. There is no technical reason that a model builder could not define Acceleration to be a base dimension and derive Position, although it is not the natural choice in this situation. When defining a base dimension, the user only needs to specify the units within that dimension. This is done by adding parameters to the DerivedDimension concept.

Example 15.13: Figure 15.20 shows parameters that have been added to the Time dimension of Figure 15.19. Here, the user specifies the scaling factor of all units

secFactor:	1.0
hrFactor:	3600*secFactor
dayFactor:	24*hrFactor
sec:	{ Factor = secFactor }
ms:	{ Factor = 0.001*secFactor }
us:	{ Factor = 1E-06*secFactor }
ns:	{ Factor = 1E-09*secFactor }
minute:	{ Factor = 60*secFactor }
hr:	{ Factor = hrFactor }
day:	{ Factor = dayFactor }
yrCalendar:	{ Factor = 365.2425*dayFactor }
yrSidereal:	{ Factor = 31558150*secFactor }
yrTropical:	{ Factor = 31556930*secFactor }

Figure 15.20: An example of the specification of the Time base dimension.

within that dimension with respect to one another, using named constants as needed to make the calculations clearer.

The Time dimension is a base dimension. The definition of a derived dimension is slightly more involved, since a derived dimension must explicitly state which dimensions it is derived from. Derived dimensions must specify both how the derived dimension is derived from other dimensions, and how each of its individual units are derived from units of those other dimensions.

Example 15.14: Figure 15.21 shows an example of the specification of the Acceleration dimension of Figure 15.19. Here, the first lines show that an Acceleration is built from the Time and Position base dimensions, and that an acceleration has units of $position/time^2$. The rest of the specification shows how individual units of acceleration are related to units of position and time.

The main benefit of specifying units this way is that we can infer the constraints for multiplication, division, and integration, which are used in many actors.

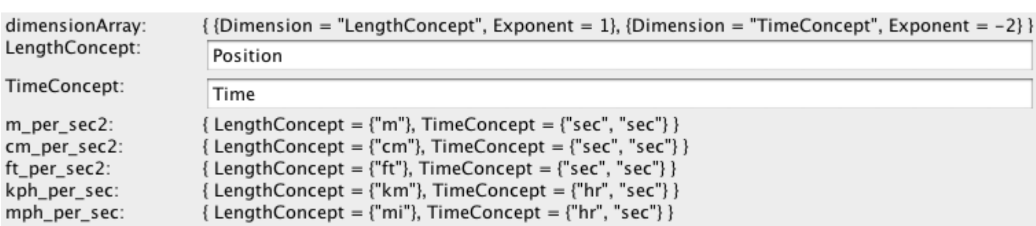


Figure 15.21: An example of the specification of the Acceleration derived dimension.

15.3.3 Converting Between Dimensions

In the case that units are used inconsistently, there is a error in the model. Not all units errors are the same, however. Units errors that result from interchanging signals with different dimensions point to a model whose connections must be changed, but unit errors from interchanging signals with different units of the same dimensions can be fixed by simply converting the units from one form to another. While it is technically possible to perform this type of conversion automatically, Ptolemy does not do this. Unit errors are an error in modeling, and model errors should never be concealed from model builders.

In keeping with this philosophy, a **UnitsConverter** actor[‡] must be explicitly added to a model in order to perform conversion between two units of the same dimension. This conversion happens both during the analysis, when the actor creates constraints on the units of its input and output ports, and during runtime, when the actor performs the linear transform from one unit to the other. Since the ontology knows the conversion factors between components, model builders need only specify the units to convert between, rather than the logic for conversion that would need to be specified in order to do conversion completely manually.

The parameters of the UnitsConverter actor, shown in Figure 15.22, include the *unitSystemOntologySolver*, which refers to the name of the LatticeOntologySolver of the units system analysis. (The name in this case is DimensionAnalysis.) Since it is only possible

[‡]which can be found in MoreLibraries→Ontologies

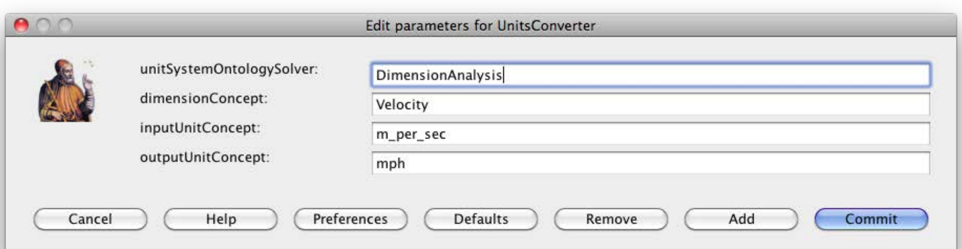


Figure 15.22: Using the UnitsConverter requires setting the name of the associated solver, as well as the input and output units.

to convert between units of the same dimension, the dimension is only specified once, in the *dimensionConcept* parameter, and the individual units of the input and output are specified in the *inputUnitConcept* and *outputUnitConcept* parameters respectively.

Example 15.15: Assume that we have a model that makes the unit error shown in Figure 15.23, where a velocity is provided in meters per second rather than miles per hour. By adding in a UnitsConverter as shown in Figure 15.24, and setting the parameters to convert from Velocity_mph to Velocity_m_per_sec as shown in Figure 15.22, we can create a model that both passes our units analysis and performs the conversion at runtime. Since the ontology analysis is not required to pass before running a model, the version of the model in Figure 15.23 without the UnitsConverter actor can still run. It produces an incorrect output value, however, since it treats the velocity value in Const as expressed in miles per hour as one in meters per second.

15.4 Summary

One of the key challenges in building large, heterogeneous models is ensuring correct composition of components. The components are often designed by different people, and their assumptions are not always obvious. Customized, domain-specific ontologies offer a powerful way to make assumptions clearer. Applying such ontologies in practice, however, can be very tedious, because they typically require the model builders to extensively annotate the model, decorating every element of the model with ontology information. The infrastructure described in this chapter leverages a very efficient inference algorithm that can significantly reduce the effort required to apply an ontology to a model. Far fewer annotations are required than is typical because most ontology associations are inferred.

Domain-specific ontologies and the associated constraints, however, can get quite sophisticated. The vision here is that libraries of ontologies, constraints, and analyses will be built up and re-used. This is certainly possible with the unit systems and dimension systems, but it also seems possible to construct libraries of analyses that are industry- or application-specific. Since these analyses are simply model components, they are easy to share among models within an enterprise.

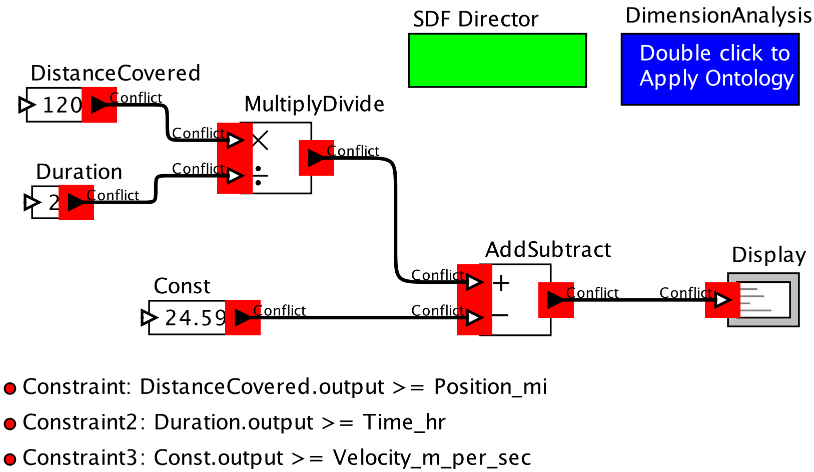


Figure 15.23: An example that adds a quantity in miles per hour to one in meters per second without conversion, resulting in conflicts throughout the model. [online]

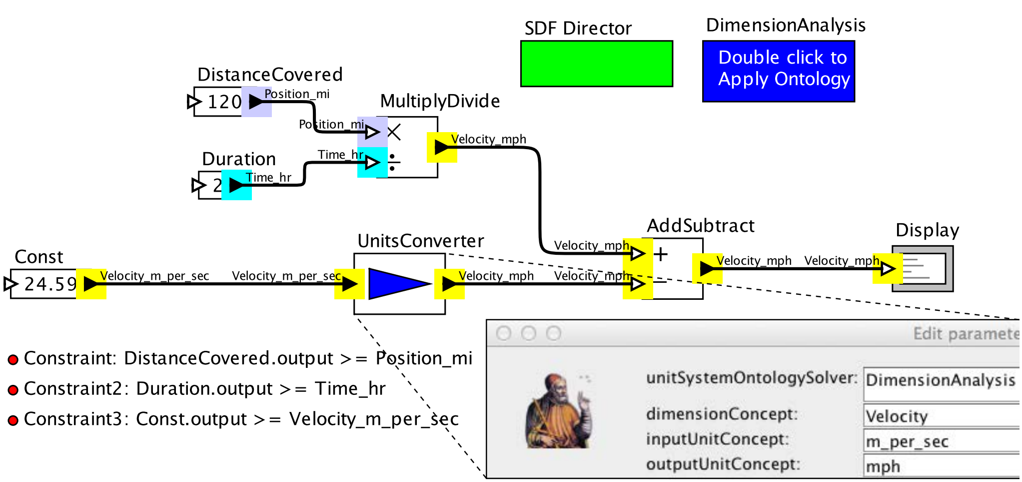


Figure 15.24: A model that uses the UnitsConverter actor to convert from meters per second to miles per hour. [online]

More interestingly, simple ontologies can be systematically combined to create more sophisticated ontologies, including ones where there are constraints that reference more than one ontology. For example, the ontology in Figure 15.11 could be factored into two simpler ontologies, once that refers to scalars and one that refers to booleans, and a product ontology could be defined in terms of these two simpler ontologies. The interested reader is referred to the Combining Ontologies chapter of Lickly (2012).

16 Web Interfaces

Christopher Brooks, Edward A. Lee, Elizabeth A. Latronico, Baobing Wang, and Roxana Gheorghui

Contents

Ptolemy II includes a flexible mechanism for creating web pages from models and for building web services. The more basic mechanism is the **export to web**, which simply makes a model available as a web page for browsing using a web browser. Such a web page provides easy access and documentation for models that archives both the structure of the models and the results of executing the models. It can be used to share information about models or their execution without requiring installation of any software, since an ordinary web browser is sufficient. More interestingly, the mechanism is extensible and customizable, allowing for creation of fairly sophisticated web pages. You can associate hyperlinks or actions defined in JavaScript* with icons in a model. The customization can be done for individual icons in a model or for sets of icons in a model.

The more advanced mechanism described in this chapter turns a model into a web service. The machine on which the model executes becomes a web server, and the model defines how the server reacts to HTTP requests that come in over the Internet. A web service can be created that does anything that can be done in a Ptolemy II model. Some care is required, of course, to ensure that such a web service does not create unacceptable security vulnerabilities for the web server machine.

16.1 Export to Web

To export a model to the web, select [`File→Export→Export to Web`], as shown in Figure 16.1. This will open a dialog that enables you to select a directory (or create new directory). That directory will be populated with a file called index.html, some image files, and some subdirectories. One image file shows whatever portion of the model is visible when you perform the export. In addition, there will be an image file for each open plot window. Moreover, there will be one subdirectory for each composite actor that is open at the time of export.

The export dialog offers a number of options, as follows.

- *directoryToExportTo*: The directory into which to put the web files. If no directory is given, then a new directory is created in the same directory that stores the MoML file for the model. The new directory will have the same name as the model, with any special characters replaced so that the name is a legal file name.
- *backgroundColor*: The background color to use for the image model. By default, this is blank, which means that the image will use whatever background color the model has (typically gray). But white is a good option for web pages, as shown in Figure 16.1.
- *openCompositesBeforeExport*: If this is true, then composite actors in the model are opened before exporting. Each composite actor will also be exported into its own web page, and hyperlinks will be created in the top-level image to allow navigation to those web pages in the browser. If you want only some of the composite actors to be included in the export, then you can manually open the ones you want. Only open windows will be included in the export.

*By default, the export to web facility uses JavaScript to display the parameters of actors. JavaScript may be disabled in your web browser. To enable JavaScript. See http://support.microsoft.com/gp/howtoscript.

- *runBeforeExport*: If this is true, then the model is run before exporting. This has the side effect of opening plot windows, which will therefore be included in the export. If you want only some of the plot windows to be included in the export, then you can run the model and close the ones you don't want. Only open plot windows will be included in the export.
- *showInBrowser*: If this is true, then once the export is complete, the resulting web page will be displayed in your default browser.
- *copyJavaScriptFiles*: If this is true, then additional files will be included in the exported page so that the page does not depend on any files from the internet. The files include

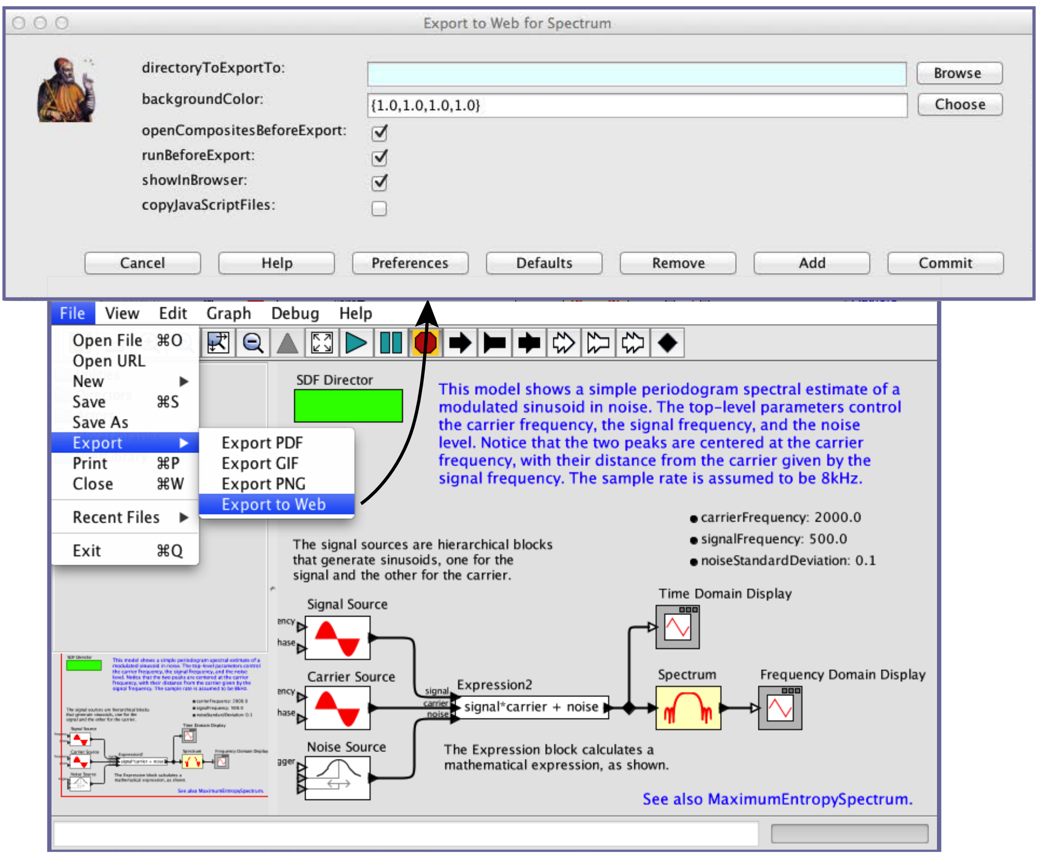


Figure 16.1: Menu command to export a model to the web.

JavaScript code and image files that affect the interactivity and look-and-feel of the web page. By default, these files are not included and are instead retrieved by the web page from `http://ptolemy.org`.

For the example shown in Figure 16.1, the resulting web page is displayed by the Safari web browser as shown in Figure 16.2. This page exhibits some of the default behavior of export to web. A title for the page is shown at the top; this is, by default, the name of the model. Moreover, in the image shown in Figure 16.2, the mouse is hovering over the Signal Source actor, which is outlined; when the mouse hovers over an actor, then by default, a table with the parameter values of the actor is displayed at the bottom of the page, as shown in Figure 16.2.

The generated web page shows the portion of the model visible in the viewing pane. Therefore, parts of the model can be hidden by resizing the viewing pane. For example, one might wish to hide a long list of parameters or attributes. Simply resize the pane, then perform the export.

In Figure 16.1, *openCompositesBeforeExport* and *runBeforeExport* are both set to true (the default is false). Hence, the model is executed before the export, opening plot windows. Hyperlinks to the plot windows are created, and clicking on a plot actor on the web page image will display the plot, as shown in Figure 16.3. In addition, the composite actors in the model, Signal Source, Carrier Source, and Spectrum, all have hyperlinks to a page showing the inner structure of the composite.

All these functions can be customized, as we will explain next.

16.1.1 Customizing the Export

As shown in Figure 16.4, the `Utilities→WebExport` library provides attributes that, when dragged into a model, customize the exported web page. This section explains each of the items in this library, shown on the left in the figure. In each case, you can right click (or control click on a Mac) and select `Get Documentation` to view documentation about the attribute. The attributes are related to one another as shown in the UML class diagram in Figure 16.5.

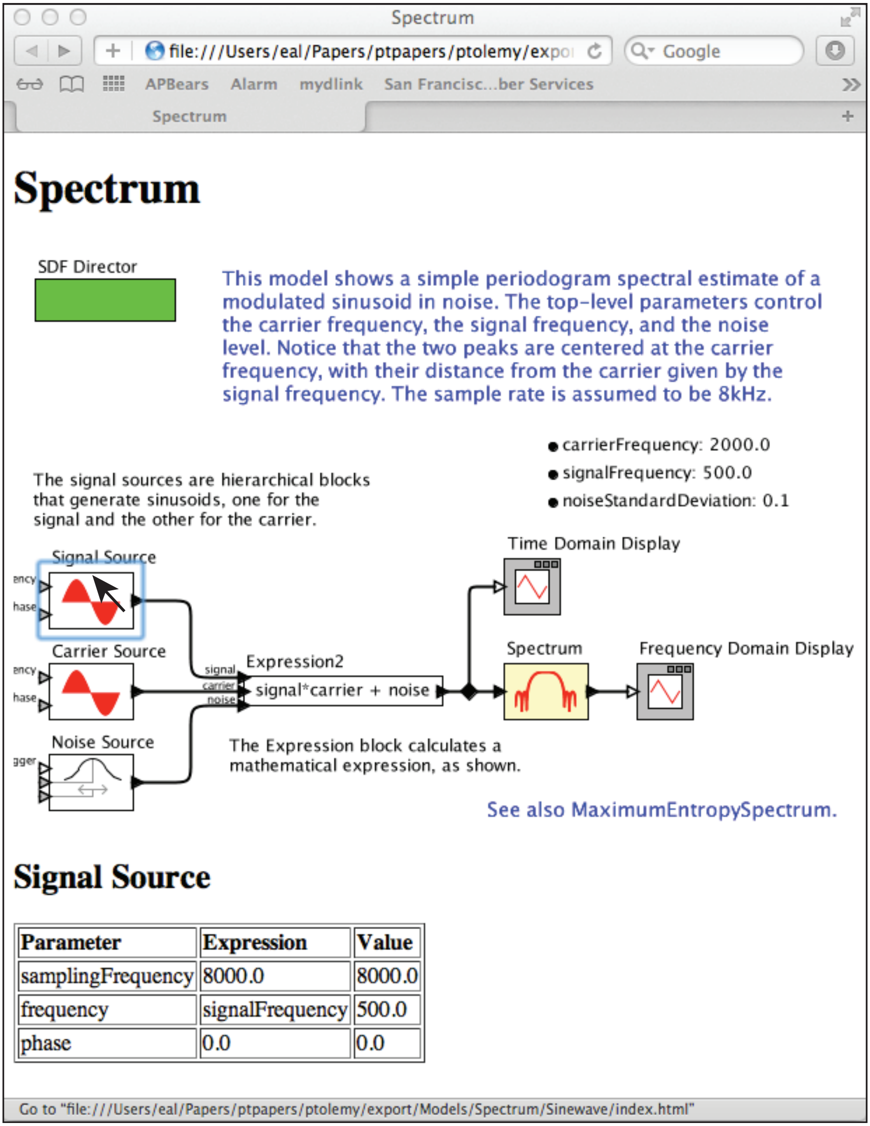


Figure 16.2: Web page exported from the model shown in Figure 16.1.

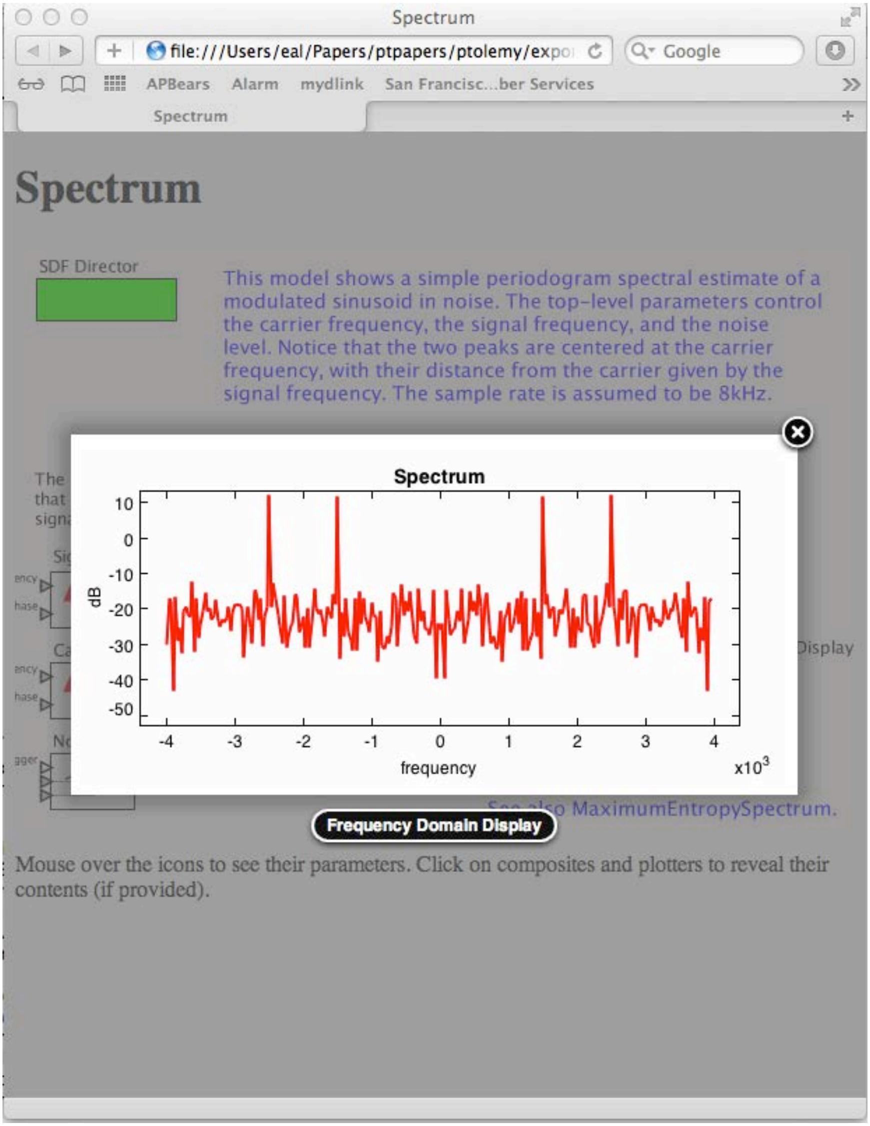


Figure 16.3: Clicking on the Frequency Domain Display actor in Figure 16.2 displays the plot generated by running the model.

HTMLText: Adding Text to Web Pages

The *HTMLText* attribute inserts HTML text into the page exported by Export to Web. Drag the attribute onto the background of a model, as shown in Figure 16.4, and double click on its icon to specify the HTML text to export. To specify the text to include in the HTML page, double click on the icon for the *HTMLText* attribute (which by default is a textual icon reading "Content for Export to Web"), as shown in Figure 16.6. You can type in the text to export, including any HTML content you like such as hyperlinks and

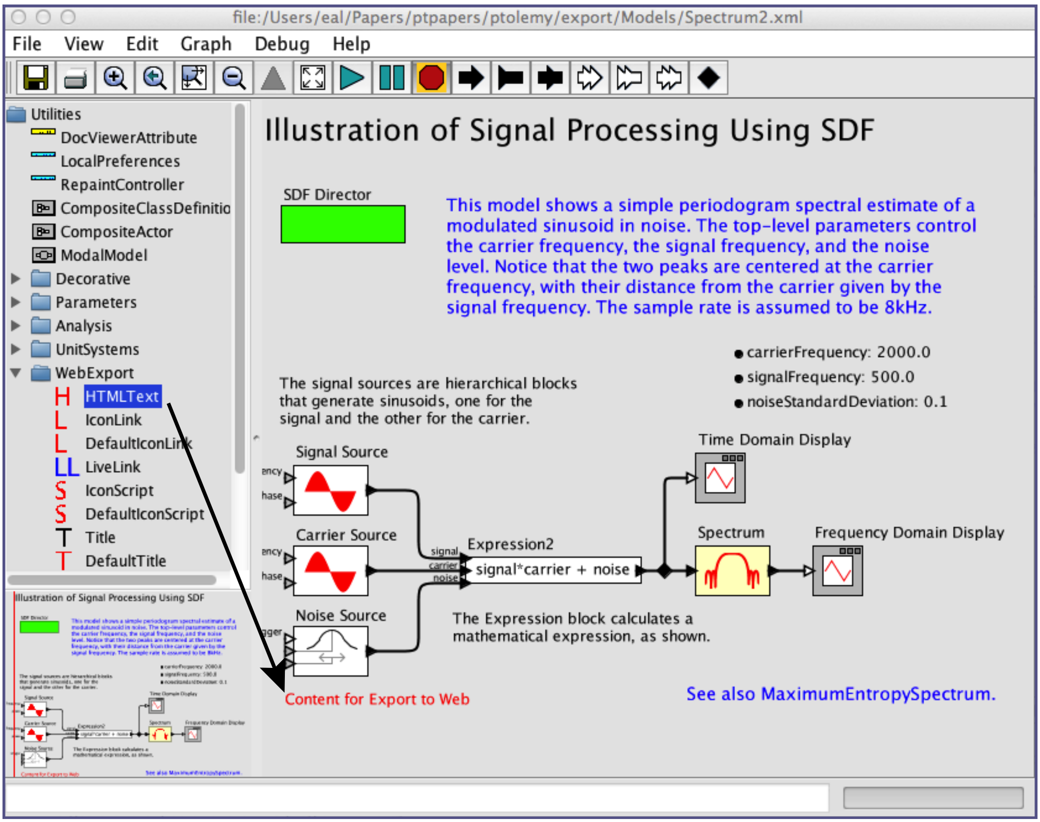


Figure 16.4: The `Utilities→Web Export` library provides attributes that, when dragged into a model, customize the exported web page. [online]

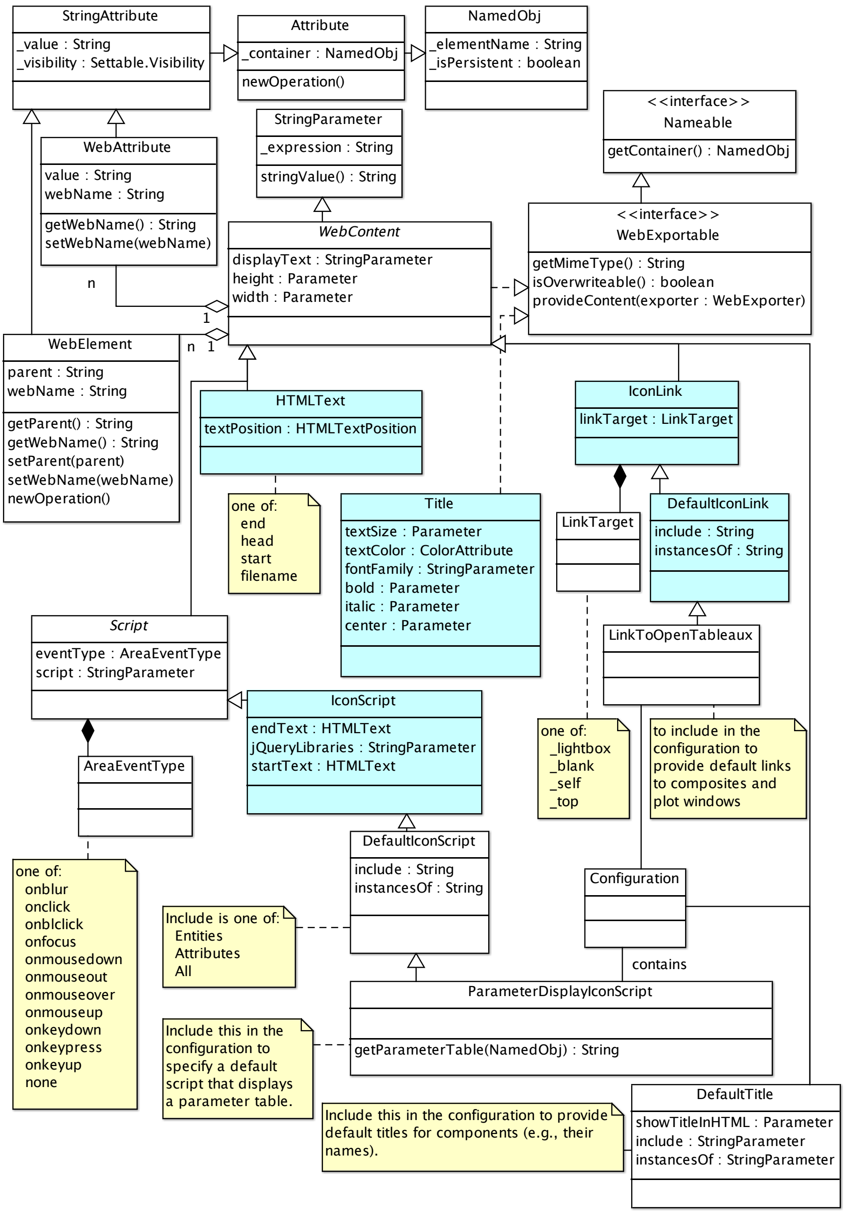


Figure 16.5: UML class diagram for the attributes for customization of exported web pages. The shaded attributes are the most commonly used in models.

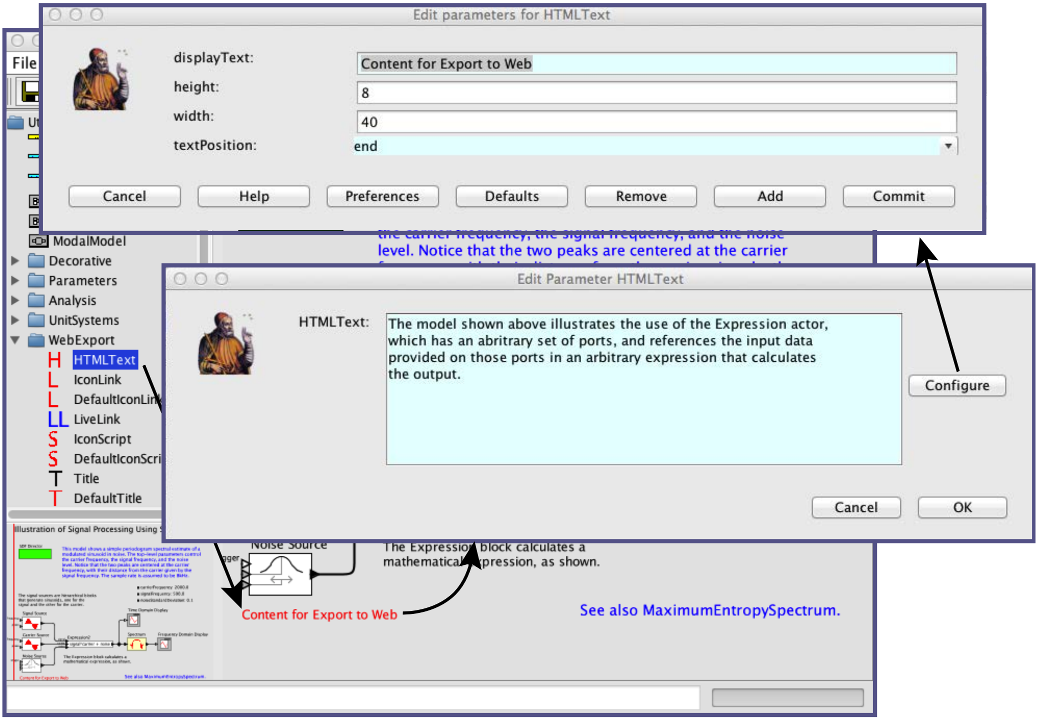


Figure 16.6: Dialog for customizing HTML text to include in an exported web page.

formatting directives. The web page including the text specified in Figure 16.6 is shown in Figure 16.7.

By default, this text will be placed before the image for the model, but you can change the position by setting the *textPosition* parameter, as shown in Figure 16.6. In that figure, you can see that the *HTMLText* attribute is configured to put the text at the end of the HTML file, which explains why that text appears at the bottom of the page in Figure 16.7.

The *HTMLText* attribute has several options for customizing it:

- *displayText*: This parameter determines what shows up in the model itself. By default, this is the text "Content for Export to Web." Notice that this text also appears in the exported web page in Figure 16.7, which is a bit odd. This text is not an interesting part of the model; it is simply a placeholder for an attribute that customizes the exported web page. If you do not want this attribute to show up in an exported web page, you can simply move the attribute out of the field of view before doing the export. Alternatively, you can set *displayText* to an empty string, but this technique has the disadvantage of making it slightly more difficult to find the attribute to edit or customize the exported text. In Figure 16.8, the *displayText* has been set to the empty string. The *HTMLText* parameter is still present and can be selected (the small yellow box that is barely visible at the lower left in the figure is the *HTMLText* parameter), but since there is no visible icon, it is hard to find. An easier way to edit the *HTMLText* parameter is to right click on the background of the model, as shown in Figure 16.8. The *HTMLText* parameter appears as a parameter of the model, along with whatever other parameters have been defined in the model.

- *height*: The height of the editing box for specifying the text to export. If you change this value, close and re-open the dialog for the change to take effect.
- *width*: The width of the editing box for specifying the text to export. If you change this value, close and re-open the dialog to see the change.
- *textPosition*: As mentioned above, this parameter determines the position of the exported text. The built-in options are end, start, and head. Choosing "end" puts the text after the exported model image. Choosing "start" puts the text before the exported model image. Choosing "head" puts the text in the header section of the HTML page. If you specify any other value for *textPosition*, then that value is assumed to be the name of a file, and a file with that name is created in the same directory as the export. The specified text is then exported to that file.

IconLink: Specifying Hyperlinks for Icons

The *IconLink* parameter shown in the `Utilities→WebExport` library can be used to specify a hyperlink for an icon in the model. To use it, drag it from the library onto the icon that you would like to have the link. In the example of Figure 16.9, we have done such a drag onto the text annotation shown at the lower right that reads "See also MaximumEntropySpectrum." Double clicking on the text annotation reveals an *IconLink* parameter that can be set to a URL. The exported web page will include a hyperlink from the text annotation to that specified page.

The *IconLink* parameter can be customized (click on the Configure button at the lower right of the dialog in Figure 16.9). The parameters *displayText*, *width*, and *height* are the

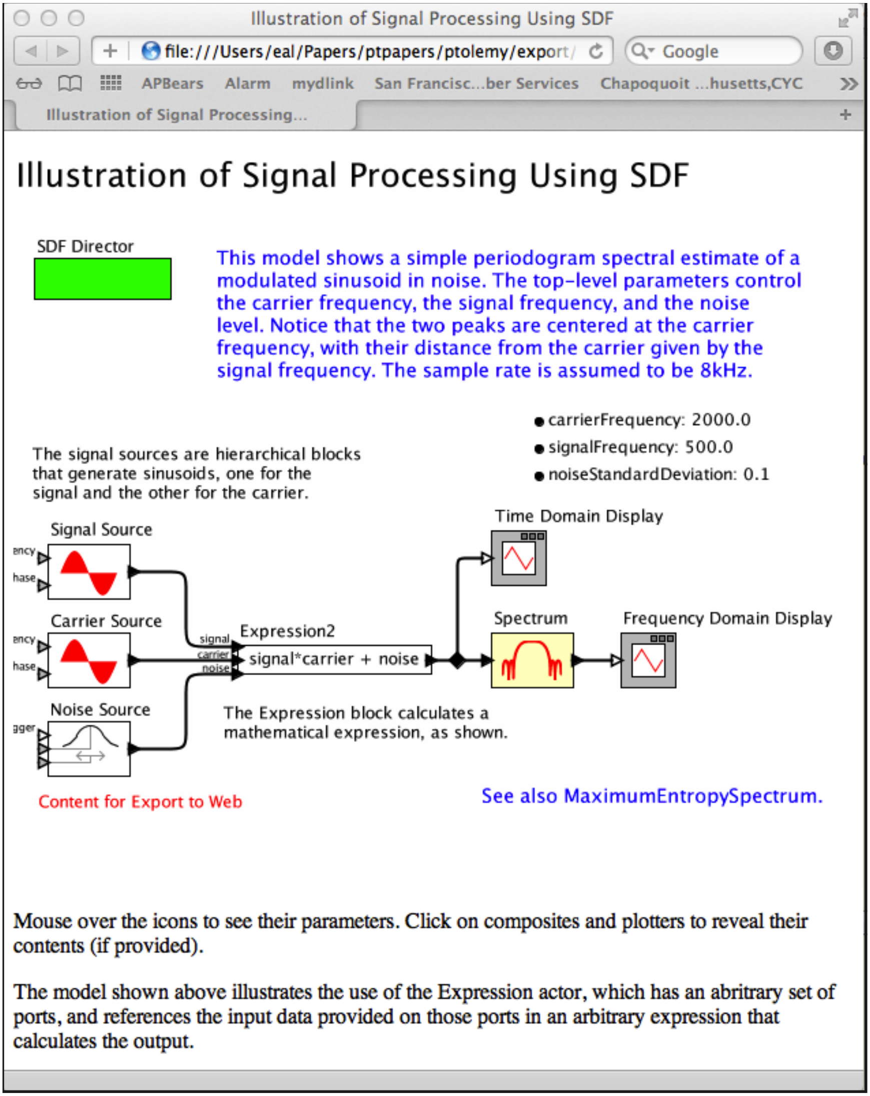


Figure 16.7: Page resulting from inserting an HTMLText attribute into the example of Figure 16.4 and configuring it as shown in Figure 16.6.

same as those for *HTMLText*, described above. A new parameter is *linkTarget*. This has four allowed values:

- `_lightbox`: Display the link in a pop-up lightbox.
- `_blank` (the default): Display the link in new blank window of the browser.
- `_self`: Display the link in the same window, replacing the current page or frame.
- `_top`: Display the link in the same window, replacing the current page.

An example of the lightbox display is the plot shown in Figure 16.3.

In addition, if the *linkTarget* parameter is given any other value, then that value is assumed to be the name of a frame in the web page, and that frame becomes the target.

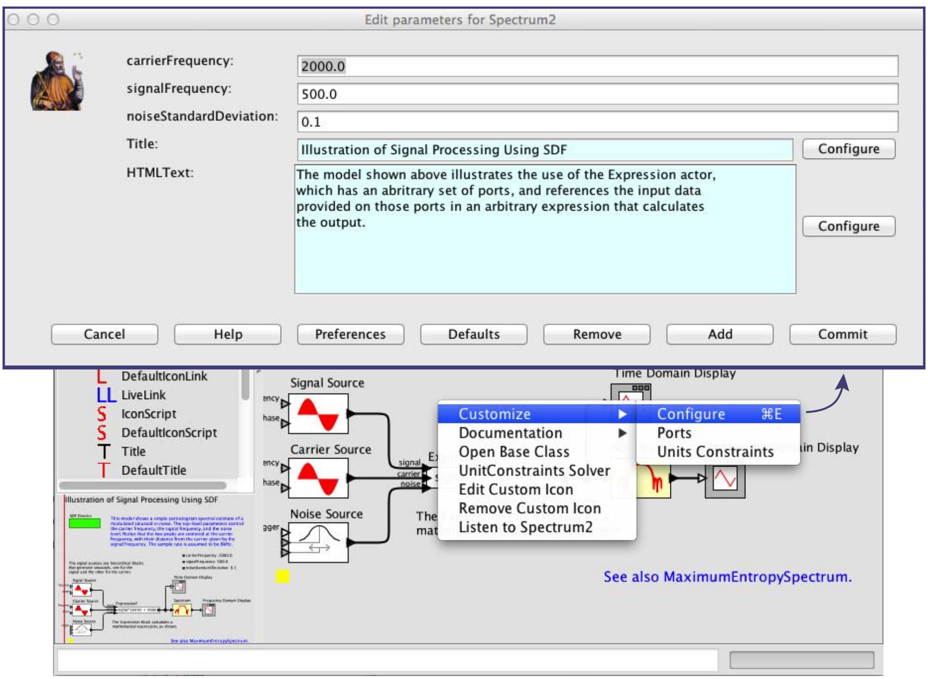


Figure 16.8: The *HTMLText* attribute can be hidden by setting its *displayText* parameter to the empty string. It can still be edited by right clicking on the background of the model. Notice that *HTMLText* appears in the list of model parameters.

DefaultIconLink: Default Hyperlinks for Icons

The *DefaultIconLink* parameter shown in the `Utilities→WebExport` library on the left in Figure 16.4 can be used to specify a default hyperlink for any icon in a model that does not contain an *IconLink*. In addition to the parameters of *IconLink*, *DefaultIconLink* has two additional parameters:

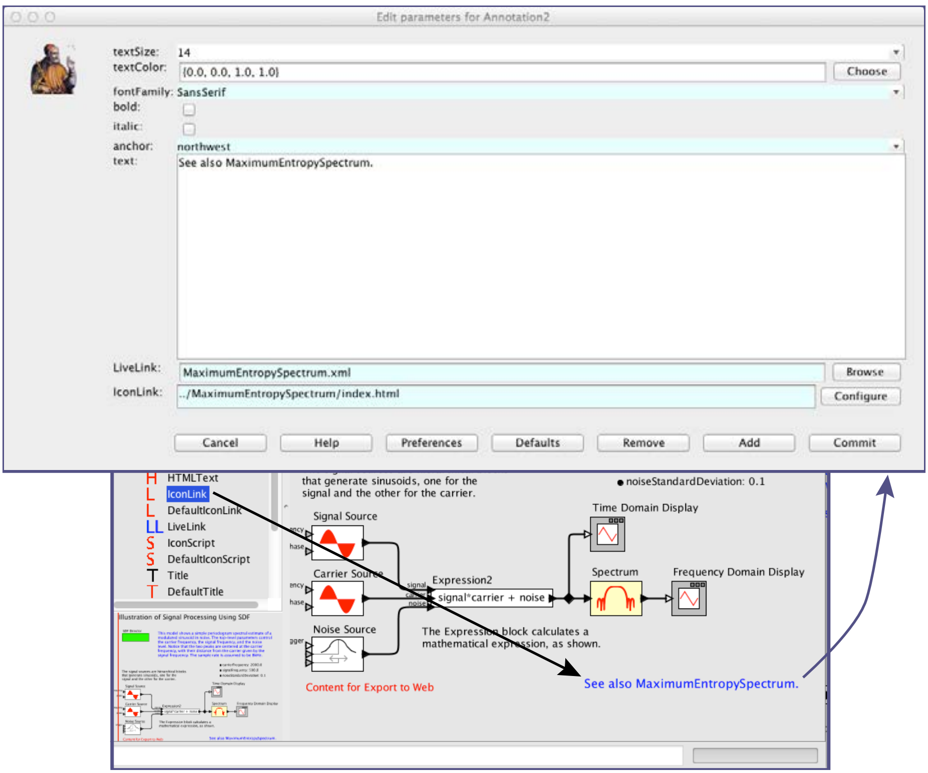


Figure 16.9: The *IconLink* attribute can be dragged onto an icon. The object onto which it is dragged acquires a parameter that can be used to specify a web page to link to from that icon when the model is exported to the web. Here, the exported web page will have a link on this icon to "../../MaximumEntropySpectrum/index.html".

- *include*: This parameter can be used to restrict icons to which the default applies. Specifically, the defaults may be specified for icons for attributes, entities, or both.
- *instancesOf*: If non-empty, this attribute specifies a class name. Only entities or attributes (depending on the *include* parameter) implementing the specified class will be assigned the default link.

LiveLink: Hyperlinks in Vergil

Although not directly related to web page exporting, the *LiveLink* parameter is included in the library because it works particularly well with *IconLink*. In particular, if you drop an instance of *LiveLink* onto an icon, then you can specify a file or URL to be opened when a user double clicks on the icon in Vergil (vs. clicking on an icon in a browser showing the exported web page). This does not automatically result in a hyperlink in an exported web page because typically a model will want to specify a different file or URL to be opened by Vergil than what would be opened by a browser. Vergil can open and display MoML files, for example, whereas a browser will simply display the XML content.

Example 16.1: Notice that in Figure 16.9, the annotation that reads "See also MaximumEntropySpectrum" contains both an instance of *IconLink* and an instance of *LiveLink*. The *LiveLink* references a MoML file, MaximumEntropySpectrum.xml, assumed to be stored in the same directory as the Spectrum model. The *IconLink* parameter, however, references an HTML file. That reference assumes that both Spectrum and MaximumEntropySpectrum will have exported web pages, and that the relative locations of these pages on a server are such that the specified path will provide a link to the HTML file for the MaximumEntropySpectrum.

Assuming all files are arranged appropriately in the file system, the Vergil hyperlink and the web page hyperlink will do essentially the same thing. They will each open the referenced model, MaximumEntropySpectrum. But Vergil will open it in Vergil, whereas a browser will open its exported web page in the browser.

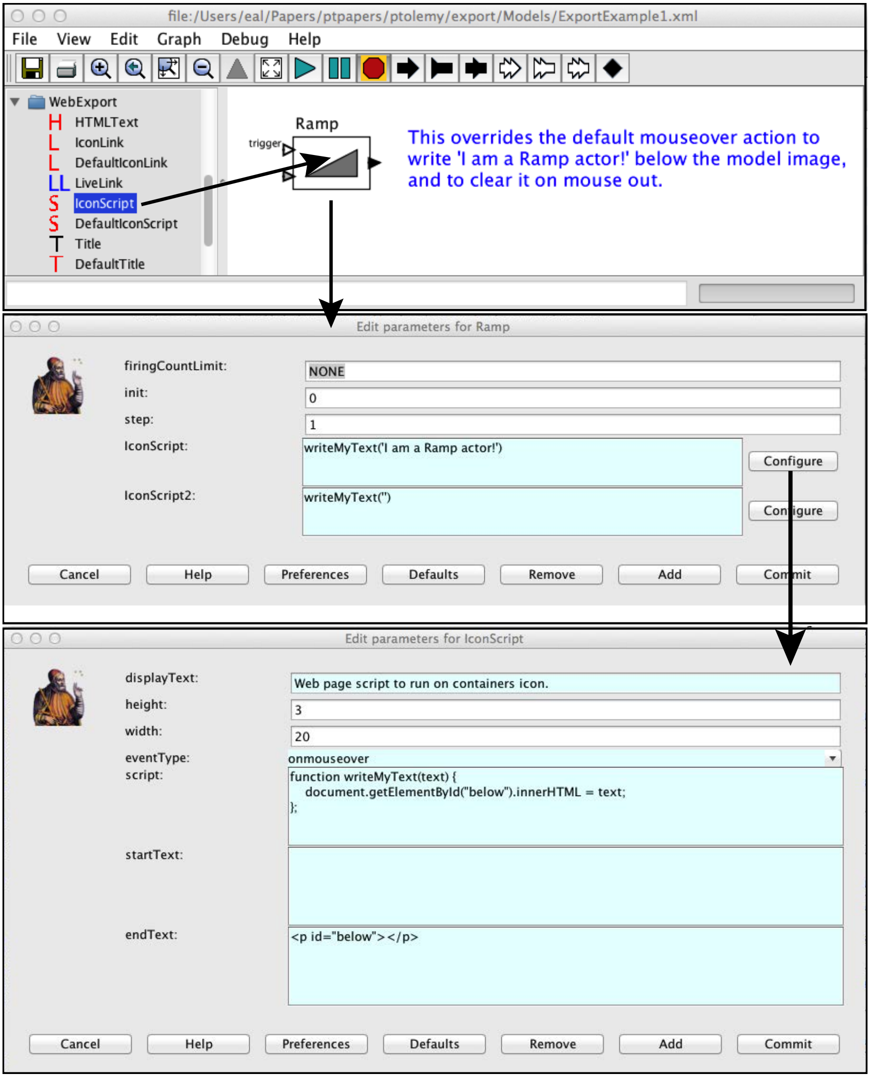


Figure 16.10: Here, two instances of the *IconScript* parameter have been dragged onto the icon for a Ramp actor. These parameters have been customized to display "I am a Ramp actor!" when the mouse enters the icon on the exported web page, and to clear the display when the mouse leaves the icon, as shown in Figure 16.11.

IconScript: Scripted Actions for Icons

The *IconScript* parameter is used to provide a scripted action associated with an icon in a model. Specifically, an action can be associated with mouse movement over the icon, mouse clicks, or keyboard actions. The action is specified as a JavaScript script.

Example 16.2: An example using *IconScript* is shown in Figures 16.10 and 16.11. In this example, two instances of the *IconScript* parameter have been dragged onto the icon for a Ramp actor. These parameters have been customized to display "I am a Ramp actor!" when the mouse enters the icon on the exported web page, and to clear the display when the mouse leaves the icon, as shown in Figure 16.11.

The way that this works is that the value of the first *IconScript* parameter is the JavaScript code:

```
writeMyText('I am a Ramp actor!')
```

This invokes a JavaScript procedure `writeMyText`, which is defined in the *script* parameter of the *IconScript* parameter to be:

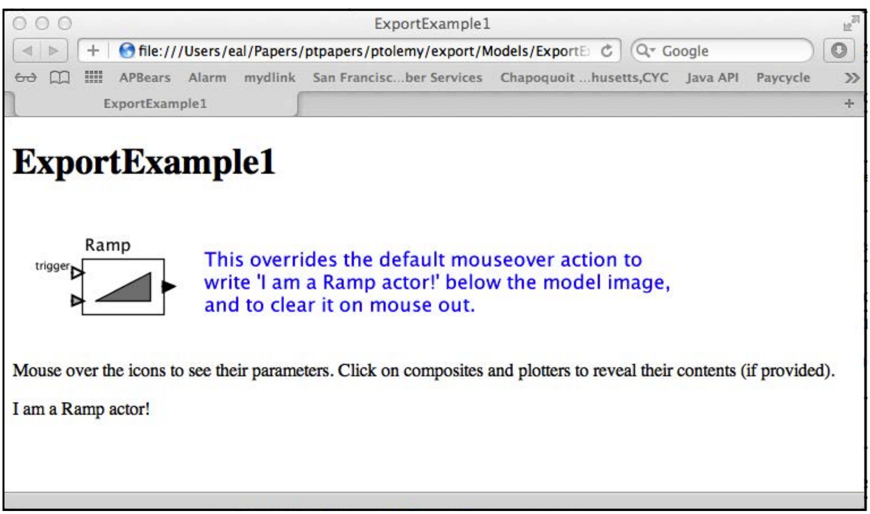


Figure 16.11: Web page exported by the model in Figure 16.11, shown with the mouse lingering over the Ramp icon.

```
function writeMyText(text) {
    document.getElementById("below").innerHTML = text;
};
```

This procedure takes one argument, `text`, and writes the value of this argument into the `innerHTML` field of the element with ID `below`. That element is defined in the *endText* parameter of the *IconScript* parameter as follows:

```
<p id="below"></p>
```

This is an HTML paragraph with ID `below`. This paragraph will be inserted into the exported web page below the model image. Finally, the *eventType* parameter of the *IconScript* is set to `onmouseover`, which results in the script being invoked when the mouse enters the area of the web page displaying the Ramp icon, as shown in Figure 16.11.

The second instance of *IconScript*, named IconScript2, specifies the following script:

```
writeMyText('')
```

This uses the same JavaScript procedure to clear the display when mouse exits the Ramp icon. The *eventType* parameter of this second *IconScript* is set to `onmouseout`.

If multiple instances of *IconScript* have exactly the same *script* parameter, then the value of that parameter will be included only once in the head section of the exported HTML page. Hence, the value of the *script* parameter is required JavaScript definitions. The web page exporter is smart enough to include those definitions only once if they are required at least once in the model.

DefaultIconScript: Default Scripted Actions for Icons

DefaultIconScript is similar to *IconScript*, except that it gets dragged onto the background of a model rather than onto an icon, and it specifies actions for many icons instead of just one. It has the same parameters as *IconScript*, but like *DefaultIconLink* described above, it also has *include* and *instancesOf* parameters, which have the same meaning described above in Section 16.1.1.

DefaultIconScript can be used, for example, to override the default behavior that causes parameters to be displayed on mouse over, as shown in Figure 16.2.

Title: Title for Icons

The *Title* parameter is used to customize the title displayed in a web page. This parameter also appears as a title in the Vergil window. The title in Figure 16.7 is actually given by an instance of *Title* inserted into the model, with the default title changed to read "Illustration of Signal Processing Using SDF." This replaces the default title provided by the web export, which is the name of the model. This title also becomes the title defined in the header of the exported HTML file.

The default value of the *Title* parameter is the expression

```
$(this.getName())
```

which is an expression in the Ptolemy II expression language for string parameters (see Chapter 13). This expression invokes the `getName` method on the container object, so the default title that is displayed is the name of the model.

DefaultTitle: DefaultTitle for Icons

The *DefaultTitle* parameter is used to customize the title associated with each icon in a model. This title is what shows up on the exported web page as a tooltip when the mouse lingers over an icon. Like *DefaultIconLink* described above, it also has *include* and *instancesOf* parameters, which have the same meaning described above in Section 16.1.1. These can be used to specify default titles for subsets of icons.

16.2 Web Services

Ptolemy allows models to be run as web services. A **web service** runs on a server and is accessible on the Internet via a uniform resource locator (**URL**). Typically, a web service responds to requests by providing either a web page (typically formatted in **HTML**, the hypertext markup language) or by providing data in some other standard Internet format such as **XML** (the extensible markup language) or **JSON** (the JavaScript object notation). The standard Ptolemy II library includes an attribute that turns a model into a web server, an actor to respond to HTTP requests, actors that facilitate constructing an HTML response, and actors for a model to access and use a web service.

16.2.1 Architecture of a Web Server

Figure 16.12 illustrates the operation of a web server. The URL for accessing the web server consists of the protocol, host name and port number (if other than the default, 80). For example, the URL `http://localhost:8078/` sends an HTTP request to the web server running on the local machine at port 8078.

A web server hosts one or more web applications (or web services). In our case, each application will be realized by one Ptolemy II model containing an instance of the WebServer attribute (see box on page 584). Each application registers an **application path** with the server. The application will handle an HTTP request for URLs that include the application path immediately after the hostname and port. For example, if application 2 registers the application path `/app2`, then the URL `http://localhost:8078/app2` will be handled by application 2. The application path can be the empty string, in which case all HTTP requests to this host on this port will be delegated to the

Sidebar: Command-Line Export

Given a MoML file for a model, you can generate a web page using a command-line program called `ptweb`. The command should have the following form:

```
ptweb [options] model [targetDirectory]
```

The "model" argument should be a MoML file. If no target directory is specified, then the name of the model becomes the name of the target directory (after any special characters have been replaced by characters that are allowed in file names). The options include:

- `-help`: Print a help message.
- `-run`: Run the model before the web page is exported, so that plot windows are included the export.

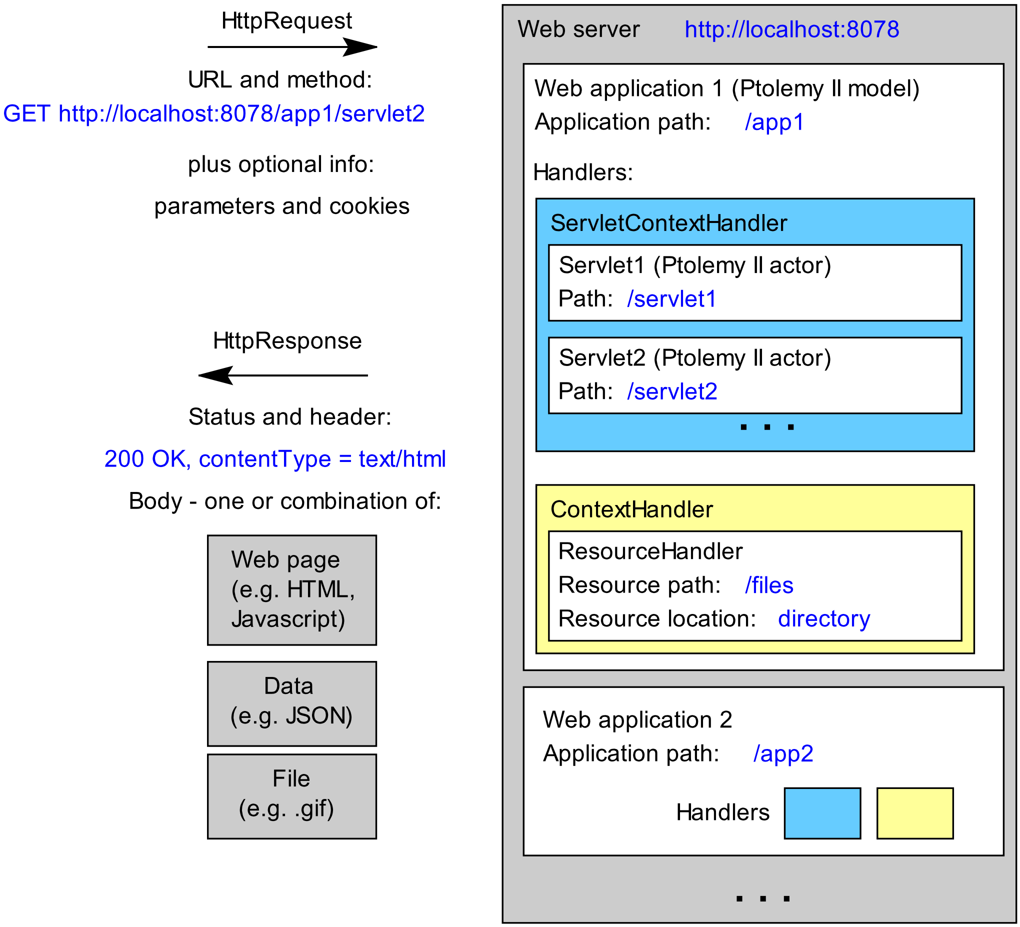


Figure 16.12: A web server hosts one or more web applications. Each application contains one or more request handlers. The web server receives HttpRequests and, according to the URL of the request, delegates the request to the appropriate request handler. The handler returns an HttpResponse.

application. When multiple applications are running on the same server, each application should have a unique application path prefix, so that the server can determine where to delegate requests.

Each application contains one or more request handlers. In our case, these handlers can be instances of the HttpActor actor (see box on page 584). Each handler registers a path prefix with the web application (this path prefix can again be the empty string). For HttpActor, the path prefix is given by the *path* parameter. When multiple handlers are running in the same application, each should have a unique path prefix, so that the application can determine where to delegate the request. For example, in Figure 16.12, the URL `http://localhost:8078/app1/servlet2` will be handled by application 1, which will delegate it to a Ptolemy II actor that has registered the prefix `servlet2`. If more that one prefix matches, then the server will delegate to handler with the most specific prefix. For example, if one handler has a blank prefix and the other has the prefix `/foo`, then all requests of the form `http://hostname:port/applicationPath/foo/...` will be delegated to the second handler, and all other requests to the first.

A second type of handler called a **resource handler** is also provided by the WebServer to handle requests for static resources such as files (Jetty class `ResourceHandler`). Again, this has a prefix which must appear in the URL. For example, in Figure 16.12, the URL `http://localhost:8078/app1/files/foo.png` references a file named `foo.png` that is stored on the server in a directory identified by a resource location attribute of the WebServer.

The response produced by a handler contains a status code, header, and the response body. The response body is the content for the user, for example, a web page, a file, or data formatted in JSON. The status code indicates whether the operation was successful, and if not, why not. There is a standard set of response codes for HTTP requests[†] The header contains information such as the content format (the **MIME type**[‡], the content length, and other useful information.

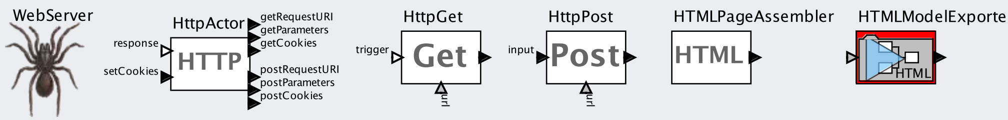
16.2.2 Constructing Web Services

The use of the WebServer and HttpActor are illustrated by the following example.

[†] See `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.
[‡] See `http://www.iana.org/assignments/media-types/index.html`.

Sidebar: Components for Web Services and Web Pages

Some components that are particularly useful for constructing web services, accessing web pages, and building web pages are shown below:



- **WebServer**. An attribute that starts a Jetty web server (see `http://www.eclipse.org/jetty/`) when the model containing it is executed. This attribute routes incoming HTTP requests to objects in the model that implement an HttpService interface, such as HttpActor. This attribute has parameters for specifying the port on which to receive HTTP requests, an application path to be included in the URL accessing this server, directories in which to find resources that are requested, and a directory in which to store temporary files.

- **HttpActor**. An actor that handles **HTTP GET** and **HTTP POST** requests that match its path. This actor is designed to work with the DE director. The outputs contain the details of the request time stamped by the elapsed time (in seconds) since the server model started executing. This actor expects that for each output it produces, the model in which it resides will provide an input that is the response to the HTTP request.

- **HttpGet**. An actor that issues an HTTP GET request to a specified URL. This actor is similar to FileReader, but it only handles URLs, and not files.

- **HttpPost**. An actor that issues an HTTP POST request to a specified URL. The contents of the post are specified by an input record.

- **HTMLPageAssembler**. An actor that assembles an HTML page by inserting input text at appropriate places in a template file.

- **HTMLModelExporter**. An extension of the VisualModelReference actor that not only displays and executes a referenced model, but also exports that model to a web page using the techniques discussed in Section 16.1.

Example 16.3: The model in Figure 16.13 is a web service that asks the user to type in some text, then returns the "Ptolemnized" text, where all leading 'p's (but not including instances of 'th') are replaced with 'pt'. For example, 'text' becomes 'ptext', as shown in Figure 16.14. The text manipulation is accomplished by the PythonScript actor, which executes the Python code shown in Figure 16.15.

First, notice that the *stopWhenQueueIsEmpty* parameter of the DE director is set to false. Were this not the case, the model would halt immediately when run because there would be no pending events to process. Second, notice that the *enable-*

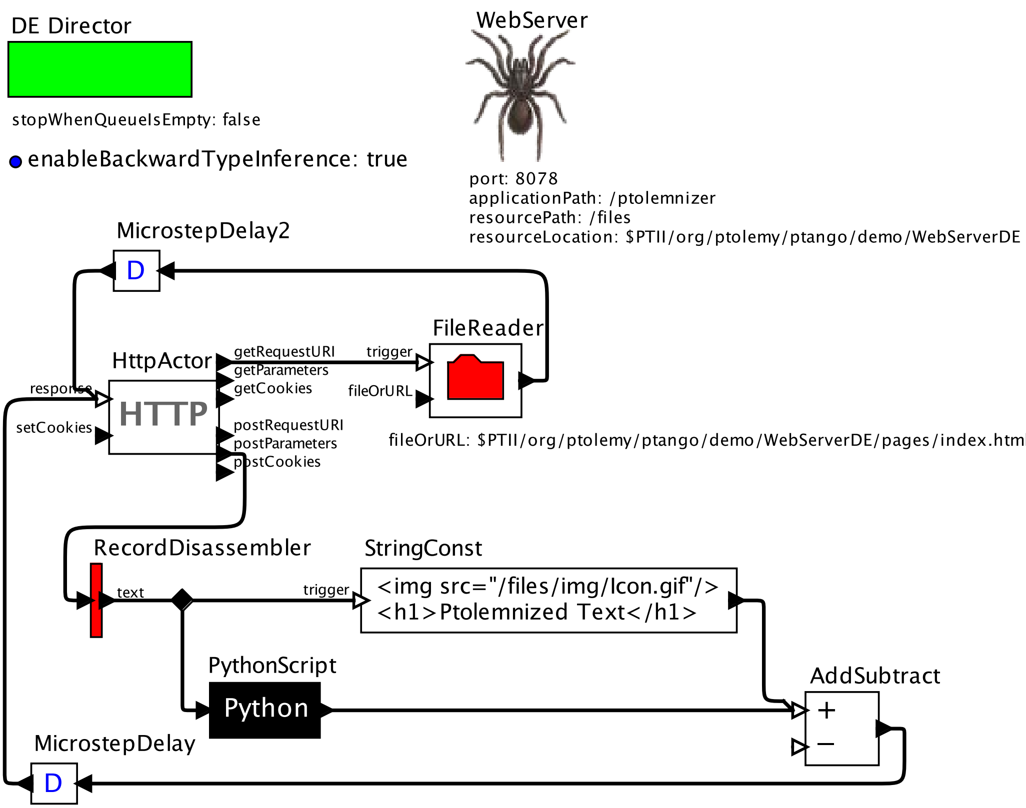


Figure 16.13: A simple web service implemented in Ptolemy II. [online]

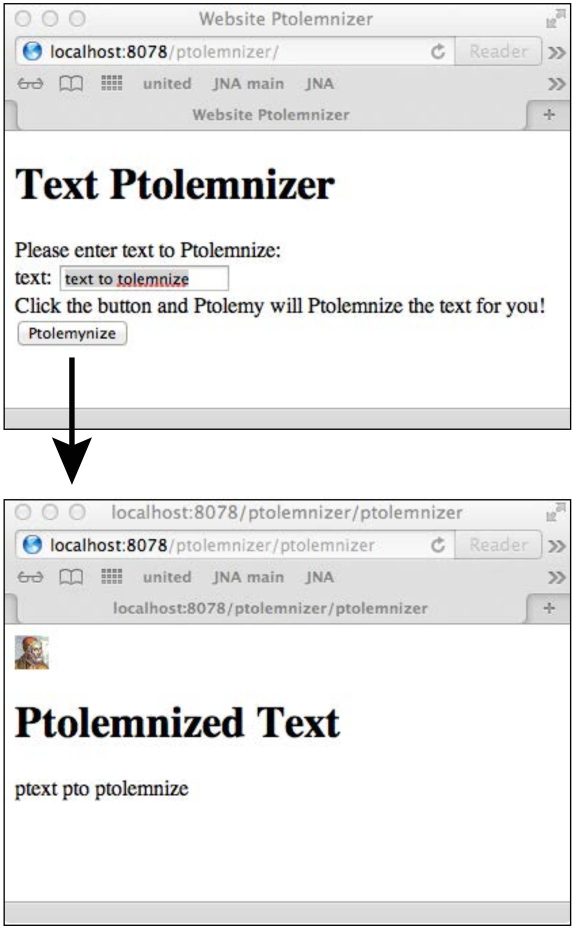


Figure 16.14: The web page returned by model in Figure 16.13 in response to an HTTP GET, and the page returned in response to a POST (triggered by the button).

```
from ptolemy.data import StringToken
class Main :
  "ptolemizer"
  def fire(self) :
    # read input, compute, send output
    t = self.in.get(0)
    s = t.stringValue()
    s = self.ptolemize(s)
    t = StringToken(s)
    self.out.broadcast(t)
    return

  def ptolemize(self, s) :
    l = list(s)
    length = len(l)
    if length == 0 :
      return ''
    if length == 1 :
      if l[0] == 't' :
        return 'pt'
      else :
        return l[0]
    if l[0] == 't' and l[1] != 'h' :
      l[0] = 'pt'
    i = 1
    while i < length - 1 :
      if l[i-1] == ' ' and l[i] == 't' and l[i+1] != 'h' :
        l[i] = 'pt'
      i = i + 1
    if l[-2] == ' ' and l[-1] == 't' :
      l[-1] = 'pt'
    return reduce(lambda x,y: x+y, l, '')
```

Figure 16.15: The Python code for the PythonScript actor in Figure 16.13.

BackwardTypeInference parameter of the model is set to true. This enables backward type inference, which in this case, results in the *postParameters* output of the HttpActor to have a record type with a single field called "test" of type string. The PythonScript actor specifies the type *string* on its input port, because the Python code expects a string.

When this model is executed, the WebServer launches a web service with an application path of `/ptolemnizer` on port 8078 of the local machine. The service is therefore available at `http://localhost:8078/ptolemnizer`. Accessing that URL in a web browser results in the top web page of Figure 16.14. How does this work?

When the web server receives an HTTP GET request with a matching application path, it delegates the request to the HttpActor. The actor requests of the director to be fired, and when the director fires it, it produces information about the GET request on its top three output ports. This model uses the URL of the GET request to trigger the FileReader actor, which simply reads a file on the local file system, the contents of which are shown in Figure 16.16. The contents of that file are sent back to the *response* input of the HttpActor, which then fires again. On that second firing, it collaborates with the WebServer to serve the response shown at the top of Figure 16.14. Note that MicrostepDelay actor is required in the feedback loop, as usual for DE models (see Section 7.3.2).

As you can see in Figure 16.14 and Figure 16.16, the web page that is served has a form, and pushing the "Ptolemnize" button results in an HTTP POST with the contents of the form. When this POST occurs, the WebServer again delegates to the HttpActor, which outputs the details of the POST on its lower three output ports. The *postParameters* port will produce a record token with a single field called "text." The RecordDisassembler extracts the value of this field, which is the text entered by the user into the form. The StringConst, PythonScript, and AddSubtract actor then construct an HTML response, which is sent back to the HttpActor. That response results in the page at the bottom of Figure 16.14.

The response to the POST includes an "img" element (see the StringConst actor in Figure 16.13). When the browser parses this response, this img element will trigger another HTTP GET. The WebServer has its *resourcePath* parameter set to `/files`, so the img src URL `/files/img/Icon.gif` will be handled by the resource handler rather than being delegated to an HttpActor (see Figure 16.12). That resource handler will search for a file named `img/Icon.gif` in the directory given by the *resourceLocation* parameter. The small Ptolemy icon on the bottom page of Figure 16.14 is the result.

This example constructs a web service by composing a number of capabilities. It uses HTML to construct an interactive web page, and Python to process data submitted by a user. In effect, the Ptolemy model is serving as an orchestrator for a number of distinct software components.

16.2.3 Storing Data on the Client using Cookies

A **cookie** is small piece of data — specifically a (name, value) pair plus expiration and visibility information — that is stored by a web browser on the client side and returned to the web server along with subsequent HTTP requests. A web service can store state on the client using a cookie; for example, a web service can use a cookie to remember that the user has logged in. A **persistent cookie** is stored for a specified period of time (including indefinitely), whereas a **session cookie** is only stored until the browser window is closed.

HttpActor has basic support for getting and setting session cookies from a client browser. Specifically, HttpActor has a *requestedCookies* parameter whose value is an array of strings. This specifies the names of cookies that the web service sets or gets. It also has an input port *setCookies*, which accepts a record that assigns values to each of the named cookies. Finally, it has output ports *getCookies* and *postCookies* that provide a record with cookie values along with each HTTP GET or POST request.

Example 16.4: The model shown in Figure 16.17 uses cookies. The web service that this model implements uses cookies to remember the identity of a client over a sequence of HTTP accesses. The pages shown in Figure 16.18 illustrate how the service responds to an initial HTTP GET, an HTTP POST that stores the identity of a client "Claudius Ptolemaeus" as a cookie, a subsequent HTTP GET, and finally, an HTTP POST that deletes the cookie.

The model has two instances of HttpActor. The first one, labeled HttpActor1, has the default *path* parameter, which matches all requests. The second one, labeled

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <title> Website Ptolemnizer </title>
</head>
<body>

<div data-role="page" data-theme="c">
  <div data-role="header">
    <h1> Text Ptolemnizer </h1>
  </div>
  <div data-role="content">
    Please enter text to Ptolemnize:
    <form action="ptolemnizer" method="post" >

      <div data-role="fieldcontain" class="ui-hide-label">
        <label for="text">text:</label>
        <input type="text" name="text" id="text" value=""
                   width="80" placeholder="text to tolemnize"/>
        </br>
      </div>

      <div>
        Click the button and Ptolemy will
        Ptolemnize the text for you!
        <br/>
        <button type="submit" id="ptolemnize">
          Ptolemynize
        </button>
      </div>
    </form>
  </div>
</div>
</body>
</html>
```

Figure 16.16: The HTML code read by the FileReader actor in Figure 16.13.

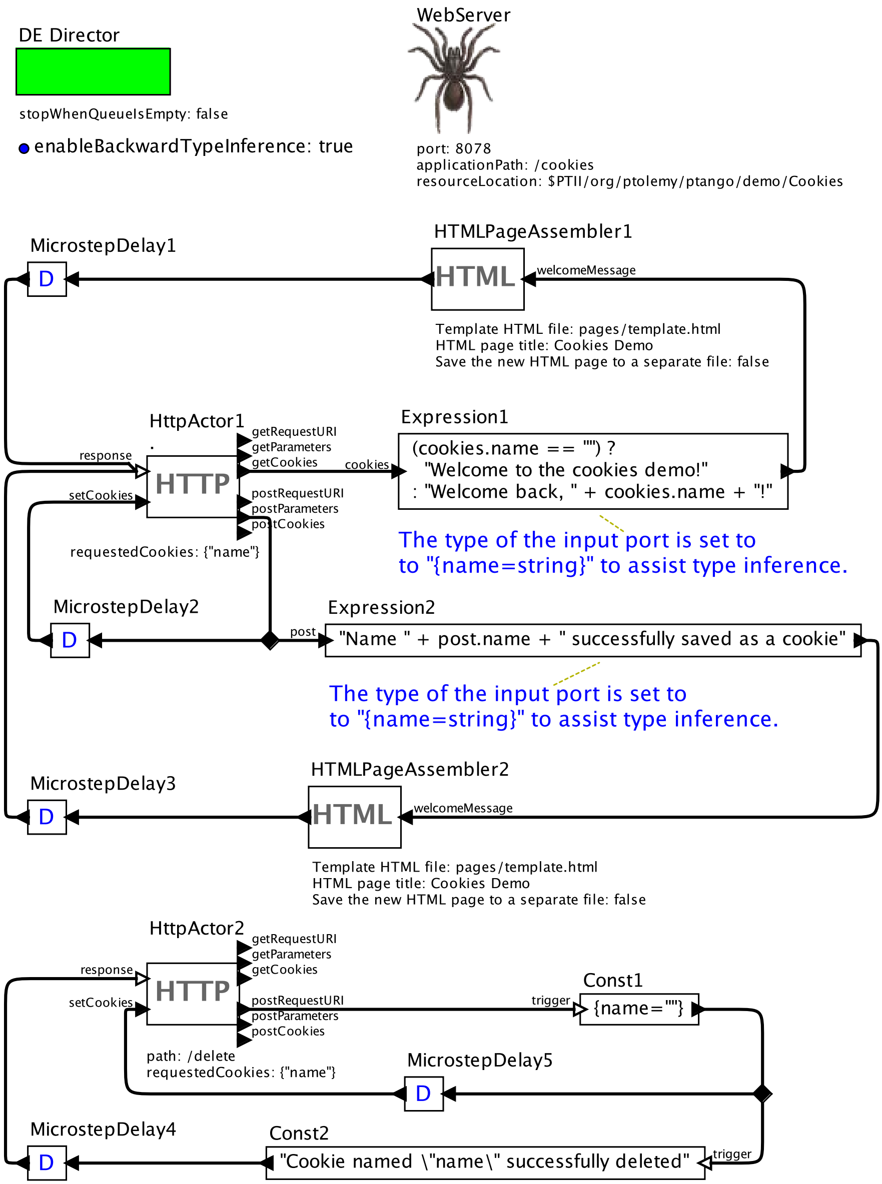


Figure 16.17: A model that gets, sets, and deletes a cookie on the client. [online]

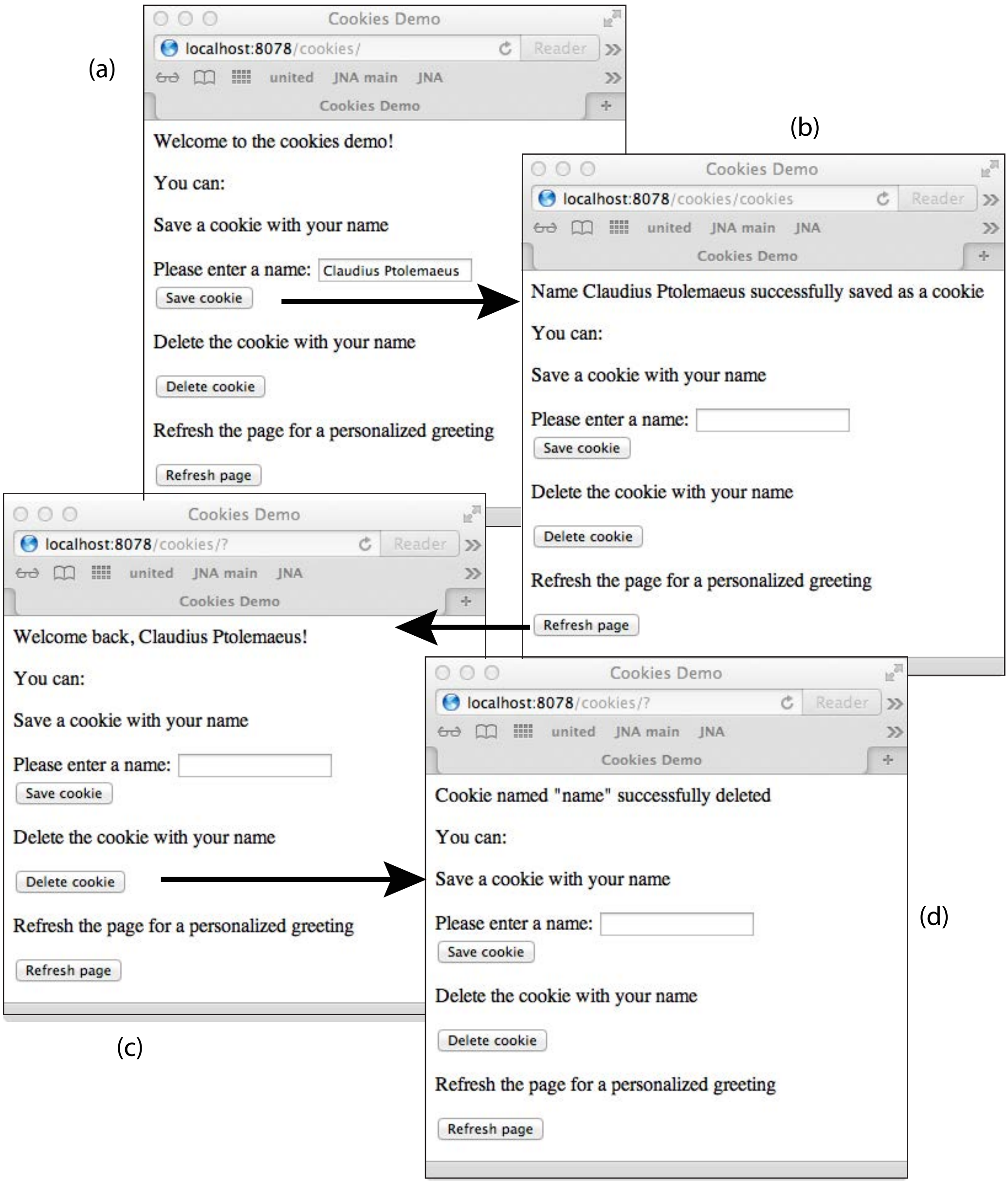


Figure 16.18: A sequence of web pages created by the model in Figure 16.17.

HttpActor2, has *path* set to `/delete`, so it will handle requests with URLs of the form `http://localhost:8078/cookies/delete`.

Both instances of HttpActor have parameter *requestedCookies* set to `{"name"}`, and array with one string. This instructs the HttpActor to check the incoming HTTP request for a cookie with the label `name`. The HttpActor produces a record on its *getCookies* or *postCookies* output port with the label `name` and the value provided by the cookie. If no cookie is found, the value is an empty string.

Note that an HttpActor actor always produces a record with the fields specified in *requestedCookies*, so downstream actors can always assume a record with the specified field. Hence, for example, the Expression actor named Expression1 in Figure 16.17 extracts the `name` field of the record using the syntax `cookies.name`. If value of the field is an empty string, then the model generates a generic welcome message, as shown in Figure 16.18(a). Otherwise, it customizes the page, as shown in Figure 16.18(c).

From the initial page, Figure 16.18(a), the user can specify a name and save a cookie with the name, which yields the response Figure 16.18(b). This is accomplished using an HTTP POST with parameter `name`. Notice in Figure 16.17 that the *postParameters* output port is fed back to the *setCookies* input port, so the response to this HTTP POST will be to set a cookie in the browser with whatever value is provided by the POST.

Clicking on the "Refresh page" button causes another HTTP GET, which now yields the customized page, Figure 16.18(c).

Clicking on the "Delete cookie" button sends a POST request to `http://localhost:8078/cookies/delete`. This request is mapped to HttpActor2. The response has two parts. First, Const1 sends a record with the label `name` and an empty string value to the *setCookies* port on HttpActor2. HttpActor2 interprets this as a request to delete the cookie. Note that, because of this implementation, the HttpActor actor will interpret any `RecordToken` label with an empty string value as a request to delete the cookie with that label. Hence, a missing cookie is equivalent to a cookie with an empty value. In addition, the model will generate a response confirming deletion of the cookie, Figure 16.18(d).

Assembling Web Pages

The model in Example 16.4 and Figure 16.17 serves some non-trivial web pages. To facilitate construction of these web pages, the model uses the HTMLPageAssembler actor. This actor inserts contents from its input ports into a specified template file, and outputs the resulting HTML page. The names of the input ports match HTML tag IDs in the template file.

Example 16.5: Figure 16.19 shows the HTML template referenced by the HTML-PageAssembler actors in Figure 16.17. Notice the **div** tag with ID "welcomeMessage." Notice further that the actors each have an input port named *welcomeMessage*, which has been added by the builder of the model. Whatever is received on this input port will be inserted into this div tag position in the response HTML page.

Note that the `Save cookie` and `Refresh page` buttons are HTML forms. These buttons perform the action specified when clicked. For example, the `Save cookie` button generates a POST request to the relative URL `cookies`, at `http://localhost:8078/cookies`, as specified by line 7. The `Refresh page` button generates a GET request to that same URL, as specified by line 24.

An alternative technique, also used in Figure 16.17, is to use JavaScript to update a page instead of returning a new page. This technique is known as **AJAX** (for asynchronous JavaScript and XML).

Example 16.6: The `Delete cookie` button calls the JavaScript function `deleteCookie()`, as shown on lines 17-18 of Figure 16.19. Figure 16.20 shows the `deleteCookie()` function definition. The function submits a POST request to the relative URL `cookies/delete`. If the request is successful, the response data are inserted into the HTML element with the ID `welcomeMessage` (overwriting any previous data). If the request is not successful, an error message is inserted into this element.

This example illustrates two reasons for using Ajax. First, returning a whole page is not necessary for the delete case. A simple message is sufficient. There are many cases where a developer might want to insert a small update into a larger page. This promotes separation of concerns, where one developer could be responsible for the main page, and a second could be responsible for updates without having to know the structure of the rest of the main page. The second developer might also want to create a web service to provide data to many different pages.

```
    <body>
    <div>
        <div id="welcomeMessage">
        </div>

        <div> <p> You can: </p>  </div>
        <form accept-charset="UTF-8" action="cookies
                       method="post">
            <p> Save a cookie with your name </p>
            <p> Please enter a name:
                <input type="text" name="name" id="name"/>
                <br>
                <input type="submit" value="Save cookie"/>
            </p>
        </form>

        <div> <p> Delete the cookie with your name </p>
           <input type="button" value="Delete cookie"
            onclick="deleteCookie()"/>
        </div>

        <div> <p>
             Refresh the page for a personalized greeting
        </p> </div>

        <form name="input" action="/cookies" method="get">
            <input type="submit" value="Refresh page" />
        </form>

    </div>
    </body>
</html>
```

Figure 16.19: The HTML template referenced by the HTMLPageAssembler actors in Figure 16.17.

```
<!DOCTYPE HTML>
<html>
    <head>
        <script type="text/javascript"
          src="http://code.jquery.com/jquery-1.6.4.min.js">
        </script>
        <script type="text/javascript">
            function deleteCookie() {
                jQuery.ajax({
                     url: "/cookies/delete",
                     type: "post",
                     success: function(data) {
                         jQuery('#welcomeMessage')
                           .html(data);
                         },
                     error: function(data) {
                         jQuery('#welcomeMessage')
                           .html("Error deleting cookie.");
                     }
                    });
            }
        </script>
        <title>Cookies demo</title>

    </head>
```

Figure 16.20: The head section of the HTML template page used in Figure 16.17.

A more subtle reason for using Ajax is that the URL of the website remains unchanged, at `http://localhost:8078/cookies`, while still being able to use a URL structure for the delete web service, `cookies/delete`. If the URL were to change to `http://localhost:8078/cookies/delete`, this would cause problems when the user clicks on further buttons, because the button URLs are defined as relative URLs. E.g., the URL would then be `http://localhost:8078/cookies/delete/cookies`.

There are, of course, many other ways to create web pages to respond to HTTP requests. A particularly interesting possibility is to use the techniques covered in Section 16.1 above to generate web pages from Ptolemy II models. In fact, a web service model could include an instance of the HTMLModelExporter actor, which refers to another Ptolemy II model, executes it, generates a web page with the results, and returns the web page. This offers a particularly powerful way to combine models to provide sophisticated services.

16.3 Summary

Building web pages and web services by constructing models offers a potentially very powerful way to combine sophisticated components in a modular way. At a minimum, the ability to export a web page that documents a model is valuable, enabling teams of designers to more effectively communicate with one another. But more interestingly, the ability to incorporate web servers into models offers a particularly powerful way to combine distributed services.

Exercises

1. Figure 4.3, discussed in Example 4.3 of Chapter 4, implements a simple chat client that uses HTTP Get and HTTP Post to enable a client to chat with other clients on the Internet. In this exercise, we build a simple (and rather limited) web server that supports this client. This server will support exactly two clients, one that will use URL
   `http://localhost:8078/chat/Claudius`
   for its Get requests, and one that will use URL
   `http://localhost:8078/chat/Ptolemaeus`
   for its Get requests. Both will use the same URL,
   `http://localhost:8078/chat/post`
   to post chat data.

   (a) A key property of this server is that it must implement long polling, where it sits on an HTTP Get request until a chat client issues an HTTP Post, which provides some chat text, and then it responds to all clients that have pending Gets with the contents of the Post. To support this, create an actor-oriented class composite actor with two input ports, *get* and *post*, and one output port *response*. This class should queue a get request (at most one) and when a post arrives, if there is a pending get request in the queue, then it should respond with the contents of the post.

   (b) Use the class definition created in part (a) to build a web server that supports the two clients.

   (c) A limitation of the chat client in Figure 4.3 is that it does not stop gracefully. The stop button in the Vergil window eventually stops it, but not until the FileReader actor times out, which can take a long time. In a better design, the server would always respond to an HTTP Get request within some amount of time, given by parameter *maximumResponseTime*. It could respond with an empty string, and the client could then filter out empty strings so that it does not display them to the user. In this design, stopping the client will succeed within the *maximumResponseTime*. Modify your server and the client to implement this.

   (d) (Open-ended question) One of the limitations of the web server you have been asked to design is that only exactly two clients are supported. Another is that there is no authentication of clients. Discuss how to address these limitations, and implement a more elaborate server that addresses at least one of these limitations.

17 Signal Display

Christopher Brooks and Edward A. Lee

Contents

Ptolemy II includes a number of **signal plotters**, shown in Figure 17.1. These can be found in the `Sinks` library, as shown in Figure 17.2. This appendix gives an overview of these capabilities, with emphasis on how to customize the plots. Once plots have been customized, saving the model containing the plotter will make the customization persistent.

17.1 Overview of Available Plotters

The plotters shown in Figure 17.1 provide a number of capabilities, and are all built on a common infrastructure. The most basic is **SequencePlotter**, which simply plots data values received on the input port. An example of a plot is shown in Figure 17.3. The mechanisms for customizing the title, axes, legend and signal plots will be covered in the following section.

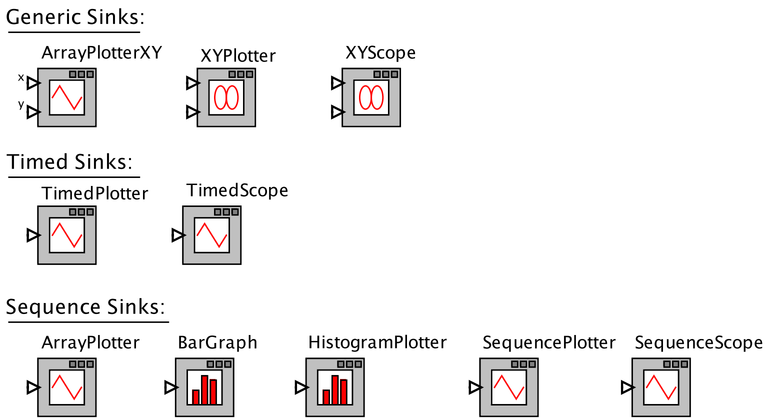


Figure 17.1: Available signal plotters.

The actors shown in the libraries in Figure 17.1 provide a variety of ways of displaying data. **ArrayPlotter** is similar except that it operates on input arrays rather than sequences. Whereas the SequencePlotter actor plots all of the input data over the entire run of the the model, the SequenceScope actor plots windows of input data, optionally overlaying them, as shown in Figure 17.4. The **SequenceScope** actor is more useful with long or infinite runs. It functions more like an oscilloscope in that it forgets old data, plotting only recent data, and overlaying windows of the data on each other.

The **TimedPlotter** actor plots input data as a function of the time stamps of the inputs, as shown in Figure 17.5. This plotter is useful in domains that advance model time, such as DE and Continuous. **TimedScope** is similar, though like SequenceScope, it functions like an oscilloscope and forgets old data.

The **XYPlotter** actor plots input data from one input port vs. input data from its other input port, as shown in Figure 17.6. **XYScope** is similar, though like SequenceScope, it functions like an oscilloscope and forgets old data. **ArrayPlotterXY** is similar, except that it operates on arrays of data rather than sequences.

BarGraph plots input arrays in the form of a bar graph. **HistogramPlotter** calculates a histogram of input data and then plots it as a bar graph, as shown in Figure 17.7. As shown in Figure 17.2, there is also a **ComputeHistogram** actor which calculates a histogram without plotting it.

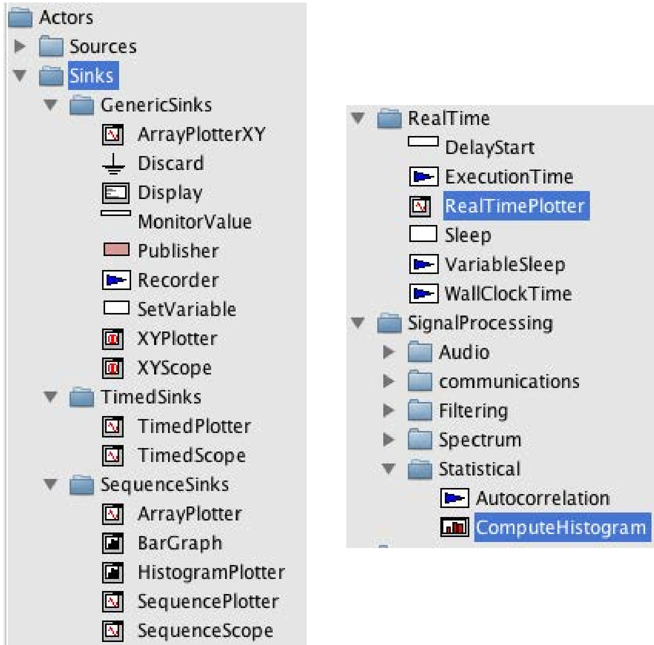


Figure 17.2: A variety of signal plotters can be found in the `Sinks` library.

Also shown in Figure 17.2 is **RealTimePlotter**, which plots input values as a function of real time elapsed on the computer executing the model.

17.2 Customizing a Plot

When used with its default configuration, the SequencePlotter will produce a plot like that shown in Figure 17.8, which is also shown in Figure 3.1. The default title is uninformative, the axes are not labeled, and the horizontal axis ranges from 0 to 255, which is not meaningful.* In the model that created this plot, which is shown in Figure 3.1, in one iteration, the Spectrum actor produces 256 output tokens. By default, the SequencePlotter just numbers these samples 0 to 255, and uses those numbers as the horizontal axis for the plot. But the horizontal axis may have more meaning in the model. In this particu-

*Hint: Notice the "$x10^2$" at the bottom right, which indicates that the label "2.5" stands for "250".

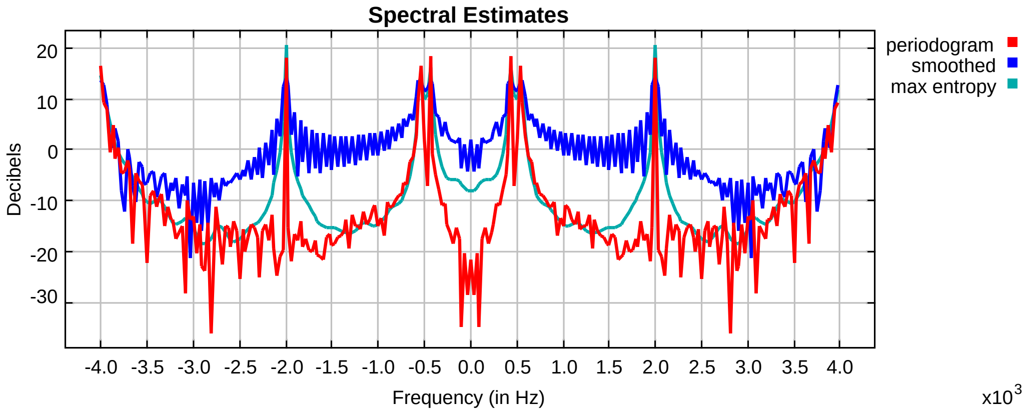


Figure 17.3: Example of a plot produced by the SequencePlotter actor.

lar example plot, the plotted data represent frequency bins that range between $-\pi$ and π radians per second.

The SequencePlotter actor has some pertinent parameters, shown in Figure 17.9, that can be used to improve the labeling of the plot. The *xInit* parameter specifies the value to use on the horizontal axis for the first token. The *xUnit* parameter specifies the value to increment this by for each subsequent token. Setting these to "-PI" and "PI/128" respectively results in the plot shown in Figure 17.10.

This plot is better, but still missing useful information. To control more precisely the visual appearance of the plot, click on the second button from the right in the row of buttons at the top right of the plot. This button brings up a format control window. It is shown in Figure 17.11, filled in with values that result in the plot shown in Figure 17.12. Most of these are self-explanatory, but the following pointers may be useful:

- The grid is turned off to reduce clutter.
- Titles and axis labels have been added.
- The X range and Y range are determined by the fill button at the upper right of the plot.
- Stem plots can be had by clicking on "Stems"
- Individual tokens can be shown by clicking on "dots"
- Connecting lines can be eliminated by deselecting "connect"

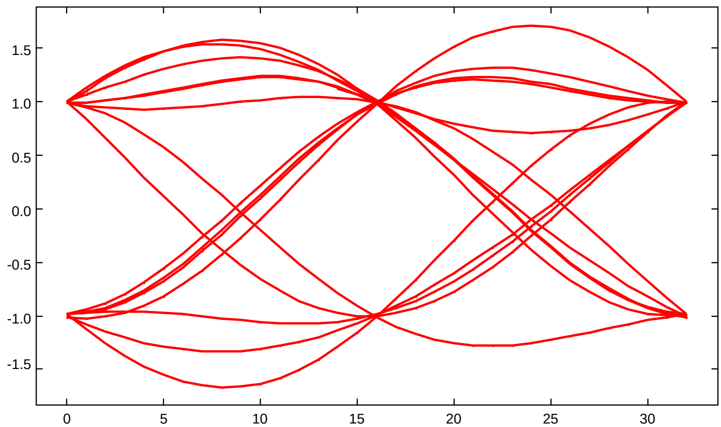


Figure 17.4: Example of a plot produced by the SequenceScope actor, which overlays successive windows of data, like what an oscilloscope does.

- The X axis label has been changed to symbolically indicate multiples of PI/2. This is done by entering the following in the X Ticks field:

  `-PI -3.14159, -PI/2 -1.570795, 0 0.0, PI/2 1.570795, PI 3.14159`

  The syntax in general is: *label value, label value,* ..., where the label is any string (enclosed in quotation marks if it includes spaces), and the value is a number.

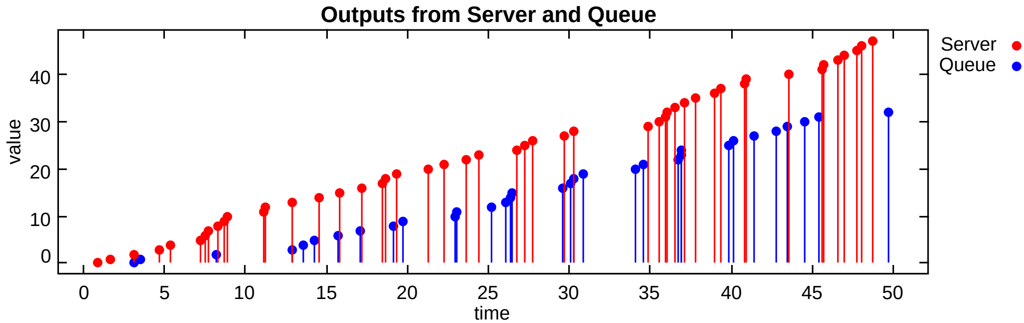


Figure 17.5: Example of a plot produced by the TimedPlotter actor, which plots input data as a function of the time stamps of the inputs.

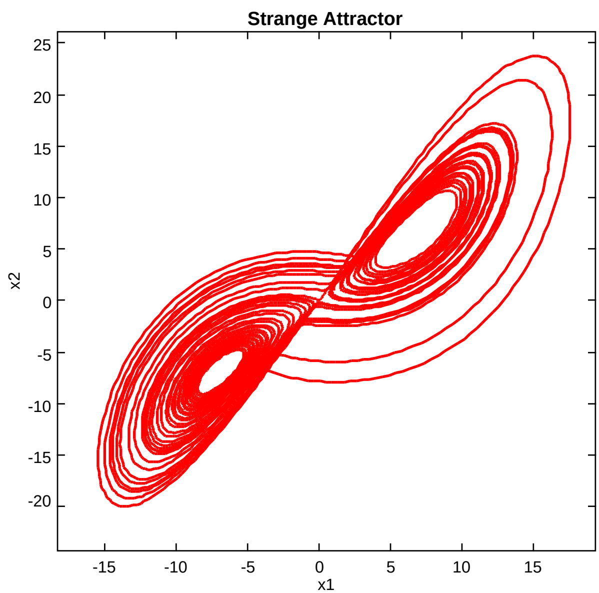


Figure 17.6: Example of a plot produced by the XYPlotter actor, which plots input data on one input port vs. input data on the other port.

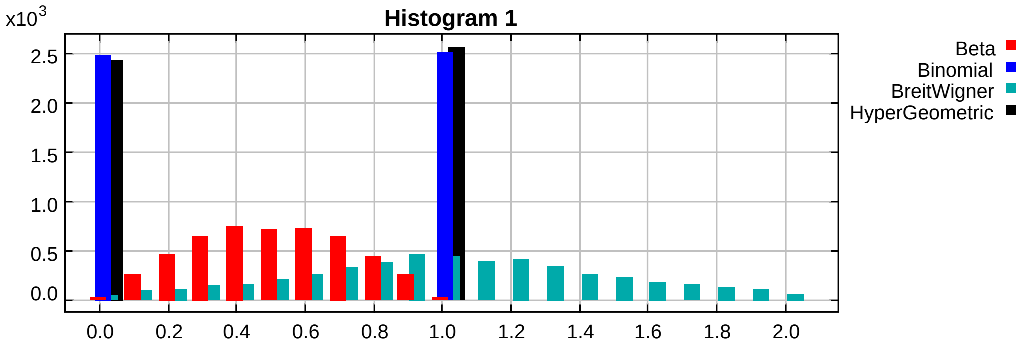


Figure 17.7: Example of a plot produced by the HistogramPlotter actor, which calculates and plots a histogram based on input data.

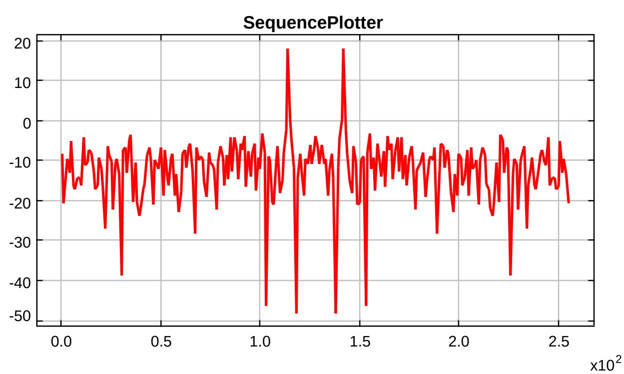


Figure 17.8: In its default configuration, a plot produced by SequencePlotter does not have informative labels.

Figure 17.9: Parameters for the SequencePlotter actor.

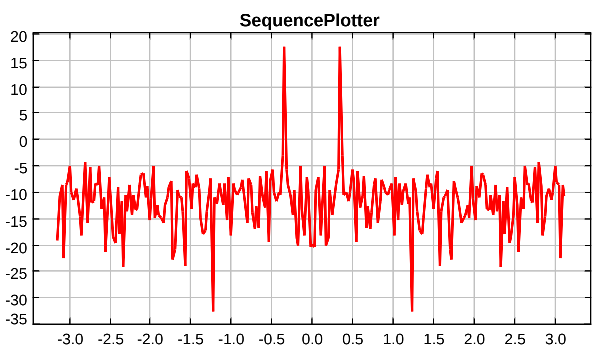


Figure 17.10: Better labeled plot, where the horizontal axis now properly represents the frequency values.

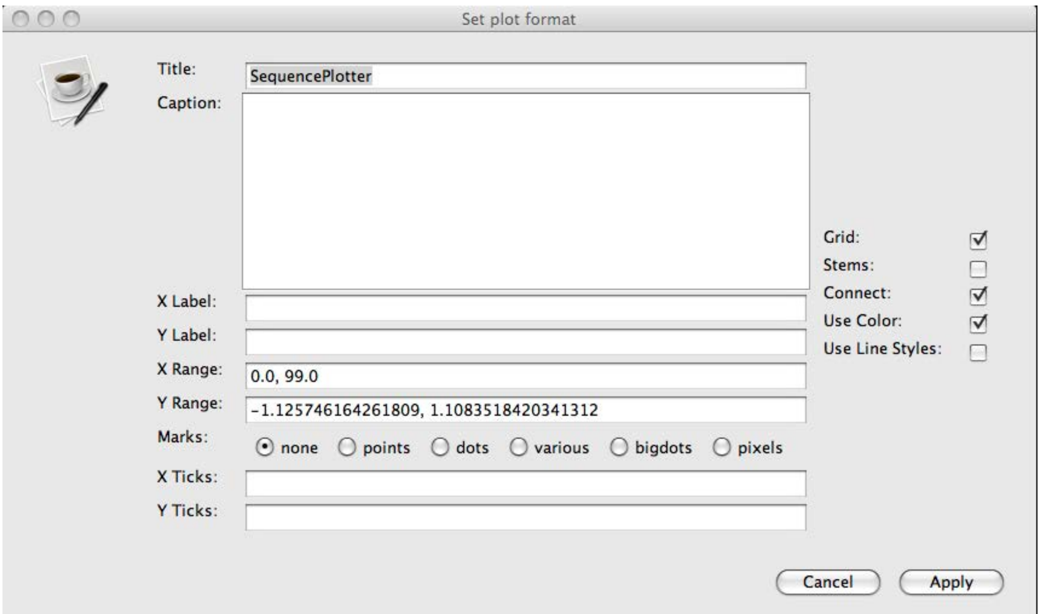


Figure 17.11: Format control window for a plot.

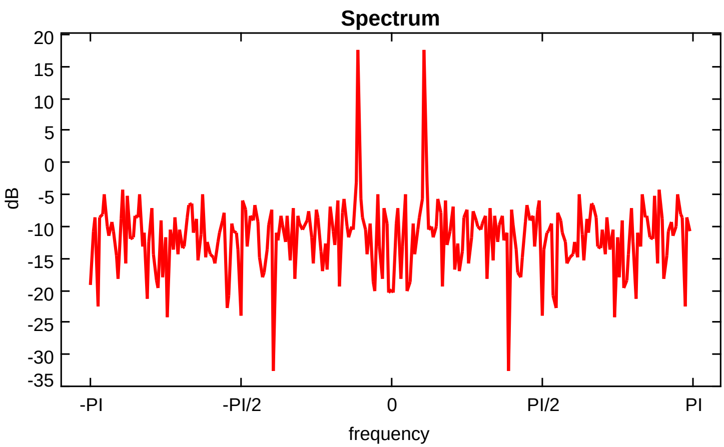


Figure 17.12: Still better labeled plot.

Bibliography

Agha, G. A., I. A. Mason, S. F. Smith, and C. L. Talcott, 1997: A foundation for actor computation. *Journal of Functional Programming*, **7(1)**, 1–72.

Allen, F. E., 1970: Control flow analysis. *SIGPLAN Notices*, **5(7)**, 1–19.

Alur, R., S. Kannan, and M. Yannakakis, 1999: Communicating hierarchical state machines. In *26th International Colloquium on Automata, Languages, and Programming*, Springer, vol. LNCS 1644, pp. 169–178.

Andalam, S., P. S. Roop, and A. Girault, 2010: Predictable multithreading of embedded applications using PRET-C. In *Formal Methods and Models for Codesign (MEMOCODE)*, IEEE/ACM, Grenoble, France, pp. 159–168. doi:10.1109/MEMCOD.2010.5558636.

André, C., 1996: SyncCharts: a visual representation of reactive behaviors. Tech. Rep. RR 95–52, revision: RR (96–56), University of Sophia-Antipolis. Available from: http://www-sop.inria.fr/members/Charles.Andre/CA%20Publis/SYNCCHARTS/overview.html.

André, C., F. Mallet, and R. d. Simone, 2007: Modeling time(s). In *Model Driven Engineering Languages and Systems (MoDELS/UML)*, Springer, Nashville, TN, vol. LNCS 4735, pp. 559–573. doi:10.1007/978-3-540-75209-7_38.

Arbab, F., 2006: A behavioral model for composition of software components. *L'Object, Lavoisier*, **12(1)**, 33–76. doi:10.3166/objet.12.1.33-76.

Arvind, L. Bic, and T. Ungerer, 1991: Evolution of data-flow computers. In Gaudiot, J.-L. and L. Bic, eds., *Advanced Topics in Data-Flow Computing*, Prentice-Hall.

Baccelli, F., G. Cohen, G. J. Olster, and J. P. Quadrat, 1992: *Synchronization and Linearity, An Algebra for Discrete Event Systems*. Wiley, New York.

Baier, C. and M. E. Majster-Cederbaum, 1994: Denotational semantics in the CPO and metric approach. *Theoretical Computer Science*, **135(2)**, 171–220.

Balarin, F., H. Hsieh, L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, and Y. Watanabe, 2003: Metropolis: an integrated electronic system design environment. *Computer*, **36(4)**.

Baldwin, P., S. Kohli, E. A. Lee, X. Liu, and Y. Zhao, 2004: Modeling of sensor nets in Ptolemy II. In *Information Processing in Sensor Networks (IPSN)*, Berkeley, CA, USA. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/04/VisualSense/.

—, 2005: Visualsense: Visual modeling for wireless and sensor network systems. Technical Report UCB/ERL M05/25, EECS Department, University of California. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/05/visualsense/index.htm.

Basu, A., M. Bozga, and J. Sifakis, 2006: Modeling heterogeneous real-time components in BIP. In *International Conference on Software Engineering and Formal Methods (SEFM)*, Pune, pp. 3–12.

Benveniste, A. and G. Berry, 1991: The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, **79(9)**, 1270–1282.

Benveniste, A., P. Caspi, P. Le Guernic, and N. Halbwachs, 1994: Data-flow synchronous languages. In Bakker, J. W. d., W.-P. d. Roever, and G. Rozenberg, eds., *A Decade of Concurrency Reflections and Perspectives*, Springer-Verlag, Berlin, vol. 803 of *LNCS*, pp. 1–45.

Benveniste, A. and P. Le Guernic, 1990: Hybrid dynamical systems theory and the SIGNAL language. *IEEE Tr. on Automatic Control*, **35(5)**, 525–546.

Berry, G., 1976: Bottom-up computation of recursive programs. *Revue Franaise dAutomatique, Informatique et Recherche Oprationnelle*, **10(3)**, 47–82.

—, 1999: *The Constructive Semantics of Pure Esterel - Draft Version 3*. Book Draft. Available from: http://www-sop.inria.fr/meije/esterel/doc/main-papers.html.

—, 2003: The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies. Available from: http://www.esterel-technologies.com.

Berry, G. and G. Gonthier, 1992: The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, **19(2)**, 87–152. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.5606.

Bhattacharya, B. and S. S. Bhattacharyya, 2000: Parameterized dataflow modeling of DSP systems. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Istanbul, Turkey, pp. 1948–1951.

Bhattacharyya, S. S., J. T. Buck, S. Ha, and E. Lee, 1995: Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, **42(3)**, 138–150. doi:10.1109/81.376876.

Bhattacharyya, S. S., J. T. Buck, S. Ha, and E. A. Lee, 1993: A scheduling framework for minimizing memory requirements of multirate DSP systems represented as dataflow graphs. In *VLSI Signal Processing VI*, IEEE, Veldhoven , The Netherlands, pp. 188–196. doi:10.1109/VLSISP.1993.404488.

Bhattacharyya, S. S. and E. A. Lee, 1993: Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI Signal Processing Systems*, **6(3)**, 271–288. doi:10.1007/BF01608539.

Bhattacharyya, S. S., P. Murthy, and E. A. Lee, 1996a: APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems*, **2(1)**, 33–60. doi:10.1023/A:1008806425898.

Bhattacharyya, S. S., P. K. Murthy, and E. A. Lee, 1996b: *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, Mass.

Bilsen, G., M. Engels, R. Lauwereins, and J. A. Peperstraete, 1996: Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, **44(2)**, 397–408. doi:10.1109/78.485935.

Bock, C., 2006: SysML and UML 2 support for activity modeling. *Systems Engineering*, **9(2)**, 160 –185.

Booch, G., I. Jacobson, and J. Rumbaugh, 1998: *The Unified Modeling Language User Guide*. Addison-Wesley.

Boussinot, F., 1991: Reactive c: An extension to c to program reactive systems. *Software Practice and Experience*, **21(4)**, 401–428.

Box, G. E. P. and N. R. Draper, 1987: *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Statistics, Wiley.

Brock, J. D. and W. B. Ackerman, 1981: Scenarios, a model of non-determinate computation. In *Conference on Formal Definition of Programming Concepts*, Springer-Verlag, vol. LNCS 107, pp. 252–259.

Broenink, J. F., 1997: Modelling, simulation and analysis with 20-Sim. *CACSD*, **38(3)**, 22–25.

Brooks, C., C. Cheng, T. H. Feng, E. A. Lee, and R. von Hanxleden, 2008: Model engineering using multimodeling. In *International Workshop on Model Co-Evolution and Consistency Management (MCCM)*, Toulouse, France. Available from: http://chess.eecs.berkeley.edu/pubs/486.html.

Brooks, C., E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, 2004: Heterogeneous concurrent modeling and design in Java. Tech. Rep. Technical Memorandum UCB/ERL M04/16, University of California. Available from: http://ptolemy.eecs.berkeley.edu/papers/04/ptIIDesignSoftware/.

Brooks, C. H. and E. A. Lee, 2003: Ptolemy II coding style. Tech. Rep. Technical Memorandum UCB/ERL M03/44, University of California at Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/03/codingstyle/.

Broy, M., 1983: Applicative real time programming. In *Information Processing 83, IFIP World Congress*, North Holland Publ. Company, Paris, pp. 259–264.

Broy, M. and G. Stefanescu, 2001: The algebra of stream processing functions. *Theoretical Computer Science*, **258**, 99–129.

Bryant, V., 1985: *Metric Spaces - Iteration and Application*. Cambridge University Press.

Buck, J. T., 1993: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. Thesis Tech. Report UCB/ERL 93/69, University of California, Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/93/jbuckThesis/.

Buck, J. T., S. Ha, E. A. Lee, and D. G. Messerschmitt, 1994: Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"*, **4**, 155–182. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/94/JEurSim/.

Burch, J. R., R. Passerone, and A. L. Sangiovanni-Vincentelli, 2001: Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *International Conference on Application of Concurrency to System Design*, p. 13.

Buss, A. H. and P. J. Sanchez, 2002: Building complex models with LEGOs (listener event graph objects). *Winter Simulation Conference (WSC 02)*, **1**, 732–737.

Cardelli, L., 1997: Type systems. In Tucker, A. B., ed., *The Computer Science and Engineering Handbook*, CRC Press, chap. 103, pp. 2208–2236, http://lucacardelli.name/Papers/TypeSystems

Cardelli, L. and P. Wegner, 1985: On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, **17(4)**, 471 – 523.

Carloni, L. P., R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli, 2006: Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, **1(1/2)**. doi:10.1561/1000000001.

Caspi, P., P. Raymond, and S. Tripakis, 2007: Synchronous Programming. In Lee, I., J. Leung, and S. Son, eds., *Handbook of Real-Time and Embedded Systems*, Chapman & Hall, pp. 14–1 — 14–21. Available from: http://www-verimag.imag.fr/~tripakis/papers/handbook07.pdf.

Cassandras, C. G., 1993: *Discrete Event Systems, Modeling and Performance Analysis*. Irwin.

Cataldo, A., E. A. Lee, X. Liu, E. Matsikoudis, and H. Zheng, 2006: A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems (WODES)*, Ann Arbor, Michigan. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/06/constructive/.

Chandy, K. M. and J. Misra, 1979: Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, **5(5)**, 440–452.

Clarke, E. M., O. Grumberg, and D. A. Peled, 2000: *Model checking*. MIT Press, ISBN 0-262-03270-8.

Coffman, E. G., Jr. (Ed), 1976: *Computer and Job Scheduling Theory*. Wiley.

Conway, M. E., 1963: Design of a separable transition-diagram compiler. *Communications of the ACM*, **6(7)**, 396–408.

Corbett, J. C., J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, 2012: Spanner: Googles globally-distributed database. In *OSDI*.

Cousot, P. and R. Cousot, 1977: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, ACM Press, pp. 238–252.

Creeger, M., 2005: Multicore CPUs for the masses. *ACM Queue*, **3(7)**, 63–64.

Davey, B. A. and H. A. Priestly, 2002: *Introduction to Lattices and Order*. Cambridge University Press, second edition ed.

de Alfaro, L. and T. Henzinger, 2001: Interface automata. In *ESEC/FSE 01: the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*.

Dennis, J. B., 1974: First version data flow procedure language. Tech. Rep. MAC TM61, MIT Laboratory for Computer Science.

Derler, P., E. A. Lee, and S. Matic, 2008: Simulation and implementation of the ptides programming model. In *IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, Vancouver, Canada.

Dijkstra, E. W., 1968: Go to statement considered harmful (letter to the editor). *Communications of the ACM*, **11(3)**, 147–148.

Edwards, S. A. and E. A. Lee, 2003a: The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, **48(1)**, 21–42. doi:10.1016/S0167-6423(02)00096-5.

—, 2003b: The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, **48(1)**, 21–42. Available from: http://ptolemy.eecs.berkeley.edu/papers/03/blockdiagram/.

Eidson, J. C., 2006: *Measurement, Control, and Communication Using IEEE 1588*. Springer. doi:10.1007/1-84628-251-9.

Eidson, J. C., E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, 2012: Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, **100(1)**, 45–59. doi:10.1109/JPROC.2011.2161237.

Eker, J. and J. W. Janneck, 2003: Cal language report: Specification of the cal actor language. Tech. Rep. Technical Memorandum No. UCB/ERL M03/48, University of California, Berkeley, CA. Available from: http://ptolemy.eecs.berkeley.edu/papers/03/Cal/index.htm.

Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, 2003: Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, **91(2)**, 127–144. Available from: http://www.ptolemy.eecs.berkeley.edu/publications/papers/03/TamingHeterogeneity/.

Encyclopedia Britannica, 2010: Ockham's razor. *Encyclopedia Britannica Online, Retrieved June 24, 2010*. Available from: http://www.britannica.com/EBchecked/topic/424706/Ockhams-razor.

Falk, J., J. Keiner, C. Haubelt, J. Teich, and S. S. Bhattacharyya, 2008: A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. In *Embedded Software (EMSOFT)*, ACM, Atlanta, Georgia, USA.

Faustini, A. A., 1982: An operational semantics for pure dataflow. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming (ICALP)*, Springer-Verlag, vol. Lecture Notes in Computer Science (LNCS) Vol. 140, pp. 212–224.

Feng, T. H., 2009: Model transformation with hierarchical discrete-event control. PhD Thesis UCB/EECS-2009-77, EECS Department, UC Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-77.html.

Feng, T. H. and E. A. Lee, 2008: Real-time distributed discrete-event execution with fault tolerance. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, St. Louis, MO, USA. Available from: http://chess.eecs.berkeley.edu/pubs/389.html.

Feng, T. H., E. A. Lee, H. D. Patel, and J. Zou, 2008: Toward an effective execution policy for distributed real-time embedded systems. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, St. Louis, MO, USA. Available from: https://chess.eecs.berkeley.edu/pubs/402.

Feng, T. H., E. A. Lee, and L. W. Schruben, 2010: Ptera: An event-oriented model of computation for heterogeneous systems. In *EMSOFT*, ACM Press, Scottsdale, Arizona, USA. doi:10.1145/1879021.1879050.

Feredj, M., F. Boulanger, and A. M. Mbobi, 2009: A model of domain-polymorph component for heterogeneous system design. *The Journal of Systems and Software*, **82**, 112–120.

Fitzgerald, J., P. G. Larsen, K. Pierce, M. Verhoef, and S. Wolff, 2010: Collaborative modelling and co-simulation in the development of dependable embedded systems. In *Integrated Formal Methods (IFM)*, Springer-Verlag, vol. LNCS 6396, pp. 12–26. doi:10.1007/978-3-642-16265-7_2.

Fitzgerald, J. S., P. G. Larsen, and M. Verhoef, 2008: Vienna development method. In *Wiley Encyclopedia of Computer Science and Engineering*, John Wiley & Sons, Inc. doi:10.1002/9780470050118.ecse447.

Foley, J., A. van Dam, S. Feiner, and J. Hughes, 1996: *Computer Graphics, Principles and Practice*. Addisson-Wesley, 2nd ed.

Friedman, D. P. and D. S. Wise, 1976: CONS should not evaluate its arguments. In *Third Int. Colloquium on Automata, Languages, and Programming*, Edinburg University Press.

Fritzson, P., 2003: *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley.

Fuhrmann, H. and R. v. Hanxleden, 2010: Taming graphical modeling. In *Model Driven Engineering Languages and Systems (MODELS) 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010*, Springer, Oslo, Norway, vol. 6394, pp. 196–210. doi:10.1007/978-3-642-16145-2_14.

Fuhrmann, H. and R. von Hanxleden, 2008: On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development – Monterey Workshop*, Springer, Budapest, Hungary, vol. LNCS 6028, pp. 116–140. doi:10.1007/978-3-642-12566-9_7.

Fujimoto, R., 2000: *Parallel and Distributed Simulation Systems*. John Wiley and Sons.

Gaderer, G., P. Loschmidt, E. G. Cota, J. H. Lewis, J. Serrano, M. Cattin, P. Alvarez, P. M. Oliveira Fernandes Moreira, T. Wlostowski, J. Dedic, C. Prados, M. Kreider, R.Baer, S.Rauch, and T.Fleck, 2009: The white rabbit project. In *Int. Conf. on Accelerator and Large Experimental Physics Control Systems*, Kobe, Japan.

Galletly, J., 1996: *Occam-2*. University College London Press, 2nd ed.

Ganter, B. and R. Wille, 1998: *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, Berlin.

Geilen, M. and T. Basten, 2003: Requirements on the execution of Kahn process networks. In *European Symposium on Programming Languages and Systems*, Springer, LNCS, pp. 319–334. Available from: http://www.ics.ele.tue.nl/~tbasten/papers/esop03.pdf.

Geilen, M., T. Basten, and S. Stuijk, 2005: Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Design Automation Conference (DAC)*, ACM, Anaheim, California, USA, pp. 819–824. doi:10.1145/1065579.1065796.

Geilen, M. and S. Stuijk, 2010: Worst-case performance analysis of synchronous dataflow scenarios. In *CODES+ISSS*, ACM, Scottsdale, Arizona, USA, pp. 125–134.

Girault, A., B. Lee, and E. A. Lee, 1999: Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, **18(6)**, 742–760.

Goderis, A., C. Brooks, I. Altintas, E. A. Lee, and C. Goble, 2009: Heterogeneous composition of models of computation. *Future Generation Computer Systems*, **25(5)**, 552–560. doi:doi:10.1016/j.future.2008.06.014.

Goessler, G. and A. Sangiovanni-Vincentelli, 2002: Compositional modeling in Metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Springer-Verlag, Grenoble, France.

Golomb, S. W., 1971: Mathematical models: Uses and limitations. *IEEE Transactions on Reliability*, **R-20(3)**, 130–131. doi:10.1109/TR.1971.5216113.

Gu, Z., S. Wang, S. Kodase, and K. G. Shin, 2003: An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software. In *Real-Time Systems Symposium (RTSS)*, pp. 78 – 81.

Ha, S. and E. A. Lee, 1991: Compile-time scheduling and assignment of dataflow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, **40(11)**, 1225–1238. doi:10.1109/12.102826.

Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud, 1991: The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, **79(9)**, 1305–1319.

Hardebolle, C. and F. Boulanger, 2007: ModHel'X: A component-oriented approach to multi- formalism modeling. In *MODELS 2007 Workshop on Multi- Paradigm Modeling*, Elsevier Science B.V., Nashville, Tennessee, USA.

Harel, D., 1987: Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8(3)**, 231–274.

Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, 1990: STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, **16(4)**, 403 – 414. doi:10.1109/32.54292.

Harel, D. and A. Pnueli, 1985: On the development of reactive systems. In Apt, K. R., ed., *Logic and Models for Verification and Specification of Concurrent Systems*, Springer-Verlag, vol. F13 of *NATO ASI Series*, pp. 477–498.

Henzinger, T. A., 2000: The theory of hybrid automata. In Inan, M. and R. Kurshan, eds., *Verification of Digital and Hybrid Systems*, Springer-Verlag, vol. 170 of *NATO ASI Series F: Computer and Systems Sciences*, pp. 265–292.

Henzinger, T. A., B. Horowitz, and C. M. Kirsch, 2001: Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, Springer-Verlag, Tahoe City, CA, vol. LNCS 2211, pp. 166–184.

Herrera, F. and E. Villar, 2006: A framework for embedded system specification under different models of computation in SystemC. In *Design Automation Conference (DAC)*, ACM, San Francisco.

Hewitt, C., 1977: Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, **8(3)**, 323–363.

Hoare, C. A. R., 1978: Communicating sequential processes. *Communications of the ACM*, **21(8)**, 666–677.

Hopcroft, J. and J. Ullman, 1979: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Hsu, C.-J., F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya, 2004: DIF: An interchange format for dataflow-based design tools. In *International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece.

Hu, T. C., 1961: Parallel sequencing and assembly line problems. *Operations Research*, **9(6)**, 841–848.

Ingalls, R. G., D. J. Morrice, and A. B. Whinston, 1996: Eliminating canceling edges from the simulation graph model methodology. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, IEEE Computer Society, Washington, DC, USA, ISBN 0-7803-3383-7, pp. 825–832.

Jantsch, A., 2003: *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann.

Jantsch, A. and I. Sander, 2005: Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, **152(2)**, 114–129.

Jefferson, D., 1985: Virtual time. *ACM Trans. Programming Languages and Systems*, **7(3)**, 404–425.

Johannessen, S., 2004: Time synchronization in a local area network. *IEEE Control Systems Magazine*, 61–69.

Johnston, W. M., J. R. P. Hanna, and R. J. Millar, 2004: Advances in dataflow programming languages. *ACM Computing Surveys*, **36(1)**, 1–34.

Kahn, G., 1974: The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., pp. 471–475.

Kahn, G. and D. B. MacQueen, 1977: Coroutines and networks of parallel processes. In Gilchrist, B., ed., *Information Processing*, North-Holland Publishing Co., pp. 993–998.

Karsai, G., A. Lang, and S. Neema, 2005: Design patterns for open tool integration. *Software and Systems Modeling*, **4(2)**, 157–170. doi:10.1007/s10270-004-0073-y.

Kay, S. M., 1988: *Modern Spectral Estimation: Theory & Application*. Prentice-Hall, Englewood Cliffs, NJ.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, 1997: Aspect-oriented programming. In *ECOOP, European Conference in Object-Oriented Programming*, Springer-Verlag, Finland, vol. LNCS 1241.

Kienhuis, B., E. Deprettere, P. van der Wolf, and K. Vissers, 2001: A methodology to design programmable embedded systems. In Deprettere, E., J. Teich, and S. Vassiliadis, eds., *Systems, Architectures, Modeling, and Simulation (SAMOS)*, Springer-Verlag, vol. LNCS 2268.

Kodosky, J., J. MacCrisken, and G. Rymar, 1991: Visual programming using structured data flow. In *IEEE Workshop on Visual Languages*, IEEE Computer Society Press, Kobe, Japan, pp. 34–39.

Kopetz, H., 1997: *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Springer.

Kopetz, H. and G. Bauer, 2003: The time-triggered architecture. *Proceedings of the IEEE*, **91(1)**, 112–126.

Lamport, L., R. Shostak, and M. Pease, 1978: Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21(7)**, 558–565.

Landin, P. J., 1965: A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, **8(2)**, 89–101.

Le Guernic, P., T. Gauthier, M. Le Borgne, and C. Le Maire, 1991: Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, **79(9)**, 1321 – 1336. doi:10.1109/5.97301.

Lee, E. A., 1986: A coupled hardware and software architecture for programmable digital signal processors. PhD Thesis UCB/ERL M86/54, University of California. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/86/LeePhDThesis/.

—, 1999: Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, **7**, 25–45. doi:10.1023/A:1018998524196.

—, 2006: The problem with threads. *Computer*, **39(5)**, 33–42. doi:10.1109/MC.2006.180.

—, 2008a: Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, IEEE, Orlando, Florida, pp. 363 – 369. doi:10.1109/ISORC.2008.25.

—, 2008b: ThreadedComposite: A mechanism for building concurrent and parallel Ptolemy II models. Technical Report UCB/EECS-2008-151, EECS Department, University of California, Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-151.html.

—, 2009: Finite state machines and modal models in Ptolemy II. Tech. Rep. UCB/EECS-2009-151, EECS Department, University of California, Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html.

—, 2010a: CPS foundations. In *Design Automation Conference (DAC)*, ACM, Anaheim, California, USA, pp. 737–742. doi:10.1145/1837274.1837462.

—, 2010b: Disciplined heterogeneous modeling. In Petriu, D. C., N. Rouquette, and O. Haugen, eds., *Model Driven Engineering, Languages, and Systems (MODELS)*, IEEE, pp. 273–287. Available from: http://chess.eecs.berkeley.edu/pubs/679.html.

Lee, E. A., E. Goei, H. Heine, and W. Ho, 1989: Gabriel: A design environment for programmable DSPs. In *Design Automation Conference (DAC)*, Las Vegas, NV, pp. 141–146. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/89/gabriel/.

Lee, E. A. and S. Ha, 1989: Scheduling strategies for multiprocessor real-time DSP. In *Global Telecommunications Conference (GLOBECOM)*, vol. 2, pp. 1279 –1283. doi:10.1109/GLOCOM.1989.64160.

Lee, E. A., X. Liu, and S. Neuendorffer, 2009a: Classes and inheritance in actor-oriented design. *ACM Transactions on Embedded Computing Systems (TECS)*, **8(4)**, 29:1–29:26. doi:10.1145/1550987.1550992.

Lee, E. A., S. Matic, S. A. Seshia, and J. Zou, 2009b: The case for timing-centric distributed software. In *IEEE International Conference on Distributed Computing Systems Workshops: Workshop on Cyber-Physical Systems*, IEEE, Montreal, Canada, pp. 57–64. Available from: http://chess.eecs.berkeley.edu/pubs/607.html.

Lee, E. A. and E. Matsikoudis, 2009: The semantics of dataflow with firing. In Huet, G., G. Plotkin, J.-J. Lévy, and Y. Bertot, eds., *From Semantics to Computer Science: Essays in memory of Gilles Kahn*, Cambridge University Press. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/.

Lee, E. A. and D. G. Messerschmitt, 1987a: Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, **C-36(1)**, 24–35. doi:10.1109/TC.1987.5009446.

—, 1987b: Synchronous data flow. *Proceedings of the IEEE*, **75(9)**, 1235–1245. doi:10.1109/PROC.1987.13876.

Lee, E. A. and S. Neuendorffer, 2000: MoML - a modeling markup language in XML. Tech. Rep. UCB/ERL M00/12, UC Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/00/moml/.

Lee, E. A., S. Neuendorffer, and M. J. Wirthlin, 2003: Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, **12(3)**, 231–260. Available from: http://ptolemy.eecs.berkeley.edu/papers/03/actorOrientedDesign/.

Lee, E. A. and T. M. Parks, 1995: Dataflow process networks. *Proceedings of the IEEE*, **83(5)**, 773–801. doi:10.1109/5.381846.

Lee, E. A. and A. Sangiovanni-Vincentelli, 1998: A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*,

17(12), 1217–1229. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/98/framework/.

Lee, E. A. and S. A. Seshia, 2011: *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. LeeSeshia.org, Berkeley, CA. Available from: http://LeeSeshia.org.

Lee, E. A. and S. Tripakis, 2010: Modal models in Ptolemy. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, Linköping University Electronic Press, Linköping University, Oslo, Norway, vol. 47, pp. 11–21. Available from: http://chess.eecs.berkeley.edu/pubs/700.html.

Lee, E. A. and P. Varaiya, 2011: *Structure and Interpretation of Signals and Systems*. LeeVaraiya.org, 2nd ed. Available from: http://LeeVaraiya.org.

Lee, E. A. and H. Zheng, 2005: Operational semantics of hybrid systems. In Morari, M. and L. Thiele, eds., *Hybrid Systems: Computation and Control (HSCC)*, Springer-Verlag, Zurich, Switzerland, vol. LNCS 3414, pp. 25–53. doi:10.1007/978-3-540-31954-2_2.

—, 2007: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, ACM, Salzburg, Austria, pp. 114 – 123. doi:10.1145/1289927.1289949.

Leung, M.-K., T. Mandl, E. A. Lee, E. Latronico, C. Shelton, S. Tripakis, and B. Lickly, 2009: Scalable semantic annotation using lattice-based ontologies. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ACM/IEEE, Denver, CO, USA. Available from: http://chess.eecs.berkeley.edu/pubs/611.html.

Lickly, B., 2012: Static model analysis with lattice-based ontologies. PhD Thesis Technical Report No. UCB/EECS-2012-212, EECS Department, University of California, Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-212.html.

Lickly, B., C. Shelton, E. Latronico, and E. A. Lee, 2011: A practical ontology framework for static model analysis. In *International Conference on Embedded Software (EMSOFT)*, ACM, pp. 23–32. Available from: http://chess.eecs.berkeley.edu/pubs/862.html.

Lin, Y., R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner, 2006: SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*, Orlando. Available from: http://www.eecs.umich.edu/~sdrg/publications.php.

Liskov, B. and S. Zilles, 1974: Programming with abstract data types. *ACM Sigplan Notices*, **9(4)**, 50–59. doi:10.1145/942572.807045.

Liu, J., B. Wu, X. Liu, and E. A. Lee, 1999: Interoperation of heterogeneous CAD tools in Ptolemy II. In *Symposium on Design, Test, and Microfabrication of MEMS/MOEMS*, Paris, France. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/99/toolinteraction/.

Liu, X. and E. A. Lee, 2008: CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, **409(1)**, 110–125. doi:10.1016/j.tcs.2008.08.044.

Liu, X., E. Matsikoudis, and E. A. Lee, 2006: Modeling timed concurrent systems. In *CONCUR 2006 - Concurrency Theory*, Springer, Bonn, Germany, vol. LNCS 4137, pp. 1–15. doi:10.1007/11817949_1.

Lynch, N., R. Segala, F. Vaandrager, and H. Weinberg, 1996: Hybrid I/O automata. In Alur, R., T. Henzinger, and E. Sontag, eds., *Hybrid Systems III*, Springer-Verlag, vol. LNCS 1066, pp. 496–510.

Lzaro Cuadrado, D., A. P. Ravn, and P. Koch, 2007: Automated distributed simulation in Ptolemy II. In *Parallel and Distributed Computing and Networks (PDCN)*, Acta Press.

Maler, O., Z. Manna, and A. Pnueli, 1992: From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, Springer-Verlag, pp. 447–484.

Malik, S., 1994: Analysis of cyclic combinational circuits. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, **13(7)**, 950–956.

Manna, Z. and A. Pnueli, 1992: *The Temporal Logic of Reactive and Concurrent Systems*. Springer, Berlin.

—, 1993: Verifying hybrid systems. In *Hybrid Systems*, vol. LNCS 736, pp. 4–35.

Maraninchi, F. and T. Bhouhadiba, 2007: 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *6th ACM International Conference on Generative Programming and Component Engineering (GPCE)*, Salzburg, Austria, pp. 1–3.

Maraninchi, F. and Y. Rémond, 2001: Argos: an automaton-based synchronous language. *Computer Languages*, **(27)**, 61–92.

Matic, S., I. Akkaya, M. Zimmer, J. C. Eidson, and E. A. Lee, 2011: Ptides model on a distributed testbed emulating smart grid real-time applications. In *Innovative Smart Grid Technologies (ISGT-EUROPE)*, IEEE, Manchester, UK. Available from: http://chess.eecs.berkeley.edu/pubs/857.html.

Matsikoudis, E., C. Stergiou, and E. A. Lee, 2013: On the schedulability of real-time discrete-event systems. In *International Conference on Embedded Software (EMSOFT)*, ACM, Montreal, Canada.

Matthews, S. G., 1995: An extensional treatment of lazy data flow deadlock. *Theoretical Computer Science*, **151(1)**, 195–205.

Messerschmitt, D. G., 1984: A tool for structured functional simulation. *IEEE Journal on Selected Areas in Communications*, **SAC-2(1)**.

Mills, D. L., 2003: A brief history of NTP time: confessions of an internet timekeeper. *ACM Computer Communications Review*, **33**.

Milner, R., 1978: A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**, 348–375.

—, 1980: *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer.

Misra, J., 1986: Distributed discrete event simulation. *ACM Computing Surveys*, **18(1)**, 39–65.

Modelica Association, 2009: Modelica®- a unified object-oriented language for physical systems modeling: Language specification version 3.1. Report. Available from: http://www.Modelica.org.

Moir, I. and A. Seabridge, 2008: *Aircraft Systems: Mechanical, Electrical, and Avionics Subsystems Integration*. AIAA Education Series, Wiley, third edition ed.

Moreira, O., T. Basten, M. Geilen, and S. Stuijk, 2010: Buffer sizing for rate-optimal single-rate dataflow scheduling revisited. *IEEE Transactions on Computers*, **59(2)**, 188–201. doi:10.1109/TC.2009.155.

Morris, J. H. and P. Henderson, 1976: A lazy evaluator. In *Conference on the Principles of Programming Languages (POPL)*, ACM.

Mosterman, P. J. and H. Vangheluwe, 2004: Computer automated multi-paradigm modeling: An introduction. *Simulation: Transactions of the Society for Modeling and Simulation International Journal of High Performance Computing Applications*, **80(9)**, 433–450.

Motika, C., H. Fuhrmann, and R. v. Hanxleden, 2010: Semantics and execution of domain specific models. In *Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) at conference INFORMATIK 2010*, Bonner Köllen Verlag, Leipzig, Germany, vol. GI-Edition – Lecture Notes in Informatics (LNI),.

Murata, T., 1989: Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, **77(4)**, 541–580. doi:10.1109/5.24143.

Murthy, P. K. and S. S. Bhattacharyya, 2006: *Memory Management for Synthesis of DSP Software*. CRC Press.

Murthy, P. K. and E. A. Lee, 2002: Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, **50(8)**, 2064–2079. doi:10.1109/TSP.2002.800830.

Object Management Group (OMG), 2007: A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, OMG. Available from: http://www.omg.org/omgmarte/.

—, 2008a: System modeling language specification v1.1. Tech. rep., OMG. Available from: http://www.sysmlforum.com.

—, 2008b: A UML profile for MARTE, beta 2. OMG Adopted Specification ptc/08-06-09, OMG. Available from: http://www.omg.org/omgmarte/.

Olson, A. G. and B. L. Evans, 2005: Deadlock detection for distributed process networks. In *ICASSP*.

Parks, T. M., 1995: Bounded scheduling of process networks. Ph.D. Thesis Tech. Report UCB/ERL M95/105, UC Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/papers/95/parksThesis.

Parks, T. M. and D. Roberts, 2003: Distributed process networks in Java. In *International Parallel and Distributed Processing Symposium*, Nice, France.

Patel, H. D. and S. K. Shukla, 2004: *SystemC Kernel Extensions for Heterogeneous System Modelling*. Kluwer.

Pino, J. L., T. M. Parks, and E. A. Lee, 1994: Automatic code generation for heterogeneous multiprocessors. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Adelaide, Australia, pp. 445–448. doi:10.1109/ICASSP.1994.389626.

Pree, W. and J. Templ, 2006: Modeling with the timing definition language (TDL). In *Automotive Software Workshop San Diego (ASWSD) on Model-Driven Development of Reliable Automotive Services*, Springer, San Diego, CA, LNCS.

Press, W. H., S. Teukolsky, W. T. Vetterling, and B. P. Flannery, 1992: *Numerical Recipes in C: the Art of Scientific Computing*. Cambridge University Press.

Prochnow, S. and R. von Hanxleden, 2007: Statechart development beyond WYSIWYG. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, ACM/IEEE, Nashville, TN, USA.

Ramadge, P. and W. Wonham, 1989: The control of discrete event systems. *Proceedings of the IEEE*, **77(1)**, 81–98.

Reed, G. M. and A. W. Roscoe, 1988: Metric spaces as models for real-time concurrency. In *3rd Workshop on Mathematical Foundations of Programming Language Semantics*, London, UK, pp. 331–343.

Rehof, J. and T. A. Mogensen, 1996: Tractable constraints in finite semilattices. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, Springer-Verlag, London, UK, ISBN 3-540-61739-6, pp. 285–300.

Rehof, J. and T. . Mogensen, 1999: Tractable constraints in finite semilattices. *Science of Computer Programming*, **35(2-3)**, 191–221.

Ritchie, D. M. and K. L. Thompson, 1974: The UNIX time-sharing system. *Communications of the ACM*, **17(7)**, 365 – 375.

Rodiers, B. and B. Lickly, 2010: Width inference documentation. Technical Report UCB/EECS-2010-120, EECS Department, University of California. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-120.html.

Sander, I. and A. Jantsch, 2004: System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, **23(1)**, 17–32.

Schruben, L. W., 1983: Simulation modeling with event graphs. *Communications of the ACM*, **26(11)**, 957–963.

—, 1995: Building reusable simulators using hierarchical event graphs. In *Winter Simulation Conference (WSC 95)*, IEEE Computer Society, Los Alamitos, CA, USA, ISBN 0-7803-3018-8, pp. 472–475.

Shapiro, F. R., 2006: *The Yale Book of Quotations*. Yale University Press.

Sih, G. C. and E. A. Lee, 1993a: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, **4(2)**, 175–187. doi:10.1109/71.207593.

—, 1993b: Declustering : A new multiprocessor scheduling technique. *IEEE Transactions on Parallel and Distributed Systems*, **4(6)**, 625–637. doi:10.1109/71.242160.

Simitci, H., 2003: *Storage Network Performance Analytics*. Wiley.

Smith, N. K., 1929: *Immanuel Kant's Critique of Pure Reason*. Macmillan and Co. Available from: http://www.hkbu.edu.hk/~ppp/cpr/toc.html.

Som, T. K. and R. G. Sargent, 1989: A formal development of event graph models as an aid to structured and efficient simulation programs. *ORSA Journal on Computing*, **1(2)**, 107–125.

Spönemann, M., H. Fuhrmann, R. v. Hanxleden, and P. Mutzel, 2009: Port constraints in hierarchical layout of data flow diagrams. In *17th International Symposium on Graph Drawing (GD)*, Springer, Chicago, IL, USA, vol. LNCS. Available from: http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/gd09.pdf.

Srini, V., 1986: An architectural comparison of dataflow systems. *Computer*, **19(3)**.

Sriram, S. and S. S. Bhattacharyya, 2009: *Embedded Multiprocessors: Scheduling and Synchronization*. CRC press, 2nd ed.

Stark, E. W., 1995: An algebra of dataflow networks. *Fundamenta Informaticae*, **22(1-2)**, 167–185.

Stephens, R., 1997: A survey of stream processing. *Acta Informatica*, **34(7)**.

Stuijk, S., M. C. Geilen, and T. Basten, 2008: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, **57(10)**, 1331–1345. doi:10.1109/TC.2008.58.

Thies, W., M. Karczmarek, and S. Amarasinghe, 2002: StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, Springer-Verlag, Grenoble, France, vol. LNCS 2304. doi:10.1007/3-540-45937-5_14.

Thies, W., M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, 2005: Teleport messaging for distributed stream programs. In *Principles and Practice of Parallel Programming (PPoPP)*, ACM, Chicago, USA. doi:10.1145/1065944.1065975.

Tripakis, S., C. Stergiou, C. Shaver, and E. A. Lee, 2013: A modular formal semantics for Ptolemy. *Mathematical Structures in Computer Science*, **23**, 834–881. Available from: http://chess.eecs.berkeley.edu/pubs/999.html, doi:10.1017/S0960129512000278.

Turjan, A., B. Kienhuis, and E. Deprettere, 2003: Solving out-of-order communication in Kahn process networks. *Journal on VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, **40**, 7 – 18. doi:10.1007/s11265-005-4935-5.

University of Pennsylvania MoBIES team, 2002: HSIF semantics (version 3, synchronous edition). Tech. Rep. Report, University of Pennsylvania.

von der Beeck, M., 1994: A comparison of Statecharts variants. In Langmaack, H., W. P. de Roever, and J. Vytopil, eds., *Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag, Lübeck, Germany, vol. 863 of *Lecture Notes in Computer Science*, pp. 128–148.

von Hanxleden, R., 2009: SyncCharts in C - A proposal for light-weight deterministic concurrency. In *ACM Embedded Software Conference (EMSOFT)*, pp. 11–16. doi:10.1145/1629335.1629366.

Wiener, N., 1948: *Cybernetics: Or Control and Communication in the Animal and the Machine*. Librairie Hermann & Cie, Paris, and MIT Press.Cambridge, MA.

Xiong, Y., 2002: An extensible type system for component-based design. Ph.D. Thesis Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720. Available from: http://ptolemy.eecs.berkeley.edu/papers/02/typeSystem.

Yates, R. K., 1993: Networks of real-time processes. In Best, E., ed., *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, Springer-Verlag, vol. LNCS 715.

Zeigler, B., 1976: *Theory of Modeling and Simulation*. Wiley Interscience, New York.

Zeigler, B. P., H. Praehofer, and T. G. Kim, 2000: *Theory of Modeling and Simulation*. Academic Press, 2nd ed.

Zhao, Y., 2009: On the design of concurrent, distributed real-time systems. Ph.D. Thesis Technical Report UCB/EECS-2009-117, EECS Department, UC Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-117.html.

Zhao, Y., E. A. Lee, and J. Liu, 2007: A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, Bellevue, WA, USA, pp. 259 – 268. doi:10.1109/RTAS.2007.5.

Zou, J., 2011: From ptides to ptidyos, designing distributed real-time embedded systems. PhD Dissertation Technical Report UCB/EECS-2011-53, UC Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-53.html.

Zou, J., J. Auerbach, D. F. Bacon, and E. A. Lee, 2009a: PTIDES on flexible task graph: Real-time embedded system building from theory to practice. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, ACM, Dublin, Ireland. Available from: http://chess.eecs.berkeley.edu/pubs/531.html.

Zou, J., S. Matic, E. A. Lee, T. H. Feng, and P. Derler, 2009b: Execution strategies for Ptides, a programming model for distributed embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, San Francisco, CA. Available from: http://chess.eecs.berkeley.edu/pubs/529.html.

Actor Index

A small fraction of the actors in the standard Ptolemy II library are described in detail in this book. Below is a summary of all the actors in the library.

Sources Library		
Const	produce a constant sequence	48
CurrentMicrostep	produce the current microstep when fired	242
CurrentTime	produce the current model time when fired	242
DiscreteClock	produce a timed sequence	241
InteractiveShell	shell for interaction with a user	136
Interpolator	produce a signal by interpolating given values	48
Pulse	produce a sequence pattern	48
Ramp	produce a counting sequence	48
Sequence	produce a sequence of specified values	48
SketchedSource	produce a sketched signal	48
StringConst	produce a constant string-valued sequence	48
Subscriber	output tokens published by a Publisher	48
SubscriptionAggregator	output tokens published by a Publisher	48
PoissonClock	produce a random timed sequence	241
TriggeredSinewave	produce samples of a sine wave	48
Sinewave	produce a sine wave	48

Conversions		
AnythingToDouble	cast to double (with NaN for non-doubles)	—
BooleanToAnything	convert a boolean to two arbitrary values	169
BitsToInt	convert a sequence of bits to an integer	—
CartesianToComplex	convert two doubles to complex	—
CartesianToPolar	convert cartesian to magnitude and phase	—
ComplexToCartesian	convert complex to two doubles	—
ComplexToPolar	convert complex to magnitude and phase	—
DoubleToFix	convert a double to a fixed-point number	—
ExpressionToToken	parse and evaluate a string expression	253
FixToDouble	convert a fixed-point number to a double	—
FixToFix	convert a fixed-point number to another	—
IntToBits	convert an integer to a sequence of bits	—
InUnitsOf	reinterpret a value in new units	—
LongToDouble	coerce a long into a double	—
PolarToCartesian	convert magnitude and phase to cartesian	—
Round	convert a double to an int	—
StringToUnsignedByteArray	convert a string to an array of bytes	86
StringToXML	parse an XML-formatted string	—
TokenToExpression	produce a string representation of a token	634
UnsignedByteArrayToString	convert a byte array to a string	86

Flow Control		
BooleanMultiplexor	multiplexor with two inputs	119
BooleanSelect	Select with two inputs	119
BooleanSwitch	Switch with two outputs	119
BusAssembler	aggregate signals into a bus	65
BusDisassembler	disaggregate signals from a bus	65
Chop	break a sequence into chunks	106
Commutator	interleave streams in round robin	106
ConfigurationSelect	select inputs based on a parameter	119
ConfigurationSwitch	switch signals based on a parameter	119
CountTrues	output the number of trues received	—
Distributor	divide a stream in round robin	106
Exit	stop the current Ptolemy process and exit	—
Multiplexor	build a stream with elements from streams	119
OrderedRecordAssembler	construct an ordered record	253
RecordAssembler	construct a record	253
RecordDisassembler	extract fields from a record	253
RecordUpdater	update fields in a record	253

Flow Control		
Repeat	repeat an input token	106
Sampler	sample a signal	244
SampleDelay	output initial tokens	103
Select	interleave streams	119
Sequencer	sequence tokens by sequence number	143
SingleTokenCommutator	commutator that outputs one token at a time	110
Stop	stop a model execution	143
Switch	split streams	119
Synchronizer	synchronize a set of streams	143
ThrowException	throw an exception	—
ThrowModelError	throw a model error	—
UnionDisassembler	extract a particular type from a union	471
UnionMerge	merge types into a union	471
VectorAssembler	construct a column or row matrix	—
VectorDisassembler	deconstruct a column or row matrix	—

Higher Order Actors		
ApplyFunction	apply a function to the inputs	89
Case	apply one of n models to the inputs	120
IterateOverArray	apply a model to each element of an input array	84
MobileModel	apply a model provided as an input	89
ModelDisplay	display a model	—
ModelReference	execute a model defined in another file	87
MultiInstanceComposite	execute copies of a model on inputs	81
RunCompositeActor	execute a submodel to completion	87
PtalonActor	construct a model using Ptalon	—
ThreadedComposite	execute a model in a new thread	258
VisualModelReference	execute a model defined in another file	87

IO Actors		
ArrowKeySensor	report keystrokes on the arrow keys	128
CSVReader	read comma-separated values	128
CSVWriter	write comma-separated values	128
DatagramReader	read packets from the network	—
DatagramWriter	write packets to a network	—
DirectoryListing	list files in a directory keys	128
FileReader	read a file or URL	128
FileWriter	write a file	128

Matrix Actors		
MatrixSplit	split a matrix by tiles	—
MatrixToArray	convert a matrix to an array	86
MatrixToSequence	convert a matrix to a sequence of elements	—
MatrixViewer	display a matrix	—
SequenceToMatrix	convert a sequence to a matrix	—
SubMatrix	extract a submatrix	—

Random Number Generators		
Bernoulli	random true or false	114
ColtBeta	Beta random number	—
ColtBinomial	Binomial random number	—
ColtBinomialSelector	Binomial selection random number	—
ColtBreitWigner	Breit-Wigner random number	—
ColtChiSquare	Chi-squared random number	—
ColtExponential	exponential random number	248
ColtExponentialPower	exponential power random number	—
ColtGamma	gamma random number	—
ColtHyperGeometric	hyper-geometric random number	—
ColtLogarithmic	logarithmic random number	—
ColtNegativeBinomial	negative binomial random number	—
ColtNormal	normal random number	—
ColtPoisson	Poisson random number	—
ColtPoissonSlow	Poisson slow random number	—
ColtStudentT	student T random number	—
ColtVonMises	VonMises random number	—
ColtZeta	zeta random number	—
DiscreteRandomSource	discrete random number	—
Gaussian	Gaussian random number	62
RandomInteger	random 32-bit integer	—
Rician	Rician random number	—
Triangular	random number with a triangular distribution	—
Uniform	random number with a uniform distribution	147

Real-Time Actors		
DelayStart	delay start of execution of a model	—
ExecutionTime	simulate execution time	—
RealTimePlotter	plot a signal vs. real time	602
Sleep	sleep the calling thread	147

Real-Time Actors		
VariableSleep	sleep the calling thread	—
WallClockTime	output the current time of day	242

Signal Processing		
AudioCapture	capture audio from a microphone	98
AudioReader	output sampled audio from a file	—
AudioPlayer	play input audio samples	—
AudioWriter	write audio samples to an audio file	—
Autocorrelation	estimate the autocorrelation of the input	—
ClipPlayer	play an audio clip on each firing	—
ComputeHistogram	compute a histogram	601
ConvolutionalCoder	encode a binary sequence	—
DB	convert values to decibels	—
DelayLine	output arrays with a sliding window	107
DeScrambler	unrandomize a binary sequence	—
DownSample	downsample a signal	106
FFT	fast Fourier transform	101
FIR	finite impulse response filter	107
GradientAdaptiveLattice	gradient adaptive lattice filter	—
HadamardCode	produce a Hadamard code	—
HammingCoder	encode a binary sequence	—
HammingDecoder	decode a binary sequence	—
HuffmanCoder	encode a sequence with a Huffman code	—
HuffmanDecoder	decode a sequence with a Huffman code	—
IFFT	inverse fast Fourier transform	101
IIR	infinite impulse response filter	107
Lattice	FIR filter with a lattice structure	—
LempelZivCoder	encode a sequence with Lempel Ziv	—
LempelZivDecoder	decode a sequence with Lempel Ziv	—
LevinsonDurbin	Levinson-Durbin spectral estimation	—
LinearDifferenceEquationSystem	linear difference equation filter	—
LineCoder	binary sequence to symbol sequence	—
LMSAdaptive	least mean square adaptive filter	107
MaximumEntropySpectrum	maximum entropy spectral estimation	101
PhaseUnwrap	phase unwrap algorithm	—
PowerEstimate	estimate the power of the input	—
RaisedCosine	raised cosine frequency response filter	—
RecursiveLattice	IIR filter with a lattice structure	—
Scrambler	randomize a binary sequence	—

DomainSpecific - Synchronous Reactive		
InstantaneousDialogGenerator	test for instantaneous dialogs	—
IsPresent	true output on present input	167
NonStrictDelay	one-tick delay	167
NonStrictDisplay	display that shows absent inputs	203
NonStrictLogicGate	parallel or and and	169
NonStrictThreeBitAdder	test actor	—
When	gate a signal with another	167
TrueGate	gate a signal	167

Although they are not part of the Ptolemy II standard library, the Ptolemy distribution includes a collection of useful actors in the `MoreLibraries` library. The table below highlights a few of these.

More Libraries		
ArrayOfRecordsRecorder	display an array of records	—
CalInterpreter	actor defined in the Cal actor language	—
DatabaseInsert	database insert via a DatabaseManager	—
DatabaseManager	interface to a database	—
DatabaseQuery	perform a query via a DatabaseManager	—
DatabaseSelect	perform a select via a DatabaseManager	—
Exec	execute a command-line statement	—
HttpActor	react to HTTP requests via a WebServer actor	584
JSONToToken	convert a JSON string to a record	—
KeyWriter	write a key out to a key store	—
SQLStatement	issue an SQL statement via a DatabaseManager	—
FSMActor	finite state machine without hierarchy	188
MatlabExpression	compute an output using a MATLAB script	437
NonStrictTest	test inputs against expected values	126
PrivateKeyReader	produce a private key from a key store	—
PublicKeyReader	produce a public key from a key store	—
PythonActor	actor specified in the Python language	437
PythonScript	actor specified in the Python language	437
SecretKey	create a secret key	—
SecretKeyReader	produce a secret key from a key store	—
SendMail	send email	—
SerialComm	read from or write to a serial port	—
SignatureSigner	sign data using a private key	—
SignatureVerifier	verify signed data	—
Simulator	run a program with socket communication	—

More Libraries		
StringToXML	convert XML-formatted string to an XML token	634
SymmetricDecryption	decrypt data	—
SymmetricEncryption	encrypt data	—
SystemCommand	invoke an external program and report results	—
Test	test inputs against expected values	126
TestExceptionAttribute	check exceptions against an expected exception	126
TokenToJSON	convert a record to a JSON string	—
WebServer	start a web server on the local machine	584
XMLInclusion	combine XML tokens into one	—
XSLTransformer	transform XML using XSLT	—

Index