PTOLEMY II
HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

VOLUME 1: INTRODUCTION TO PTOLEMY II

Edited by

Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Steve, Neuendorffer, Yuhong Xiong and Haiyang Zheng

Authors: Shuvra S. Bhattacharyya, Elaine Cheong, John Davis, II, Mudit Goel, Bart Kienhuis, Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, Brian Vogel, Winthrop Williams, Yuhong Xiong, Yang Zhao and Haiyang Zheng

Memorandum No. UCB/ERL M03/27

16 July 2003

PTOLEMY II
HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

VOLUME 1: INTRODUCTION TO PTOLEMY II

Edited by

Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu,
Steve Neuendorffer, Yuhong Xiong, Haiyang Zheng

Authors: Shuvra S. Bhattacharyya, Elaine Cheong, John Davis, II, Mudit Goel,
Bart Kienhuis, Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu,
Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay,
Brian Vogel, Winthrop Williams, Yuhong Xiong, Yang Zhao and Haiyang Zheng

Memorandum No. UCB/ERL M03/27

16 July 2003

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PTOLEMY II

HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

Edited by:
Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Steve Neuendorffer, Yuhong Xiong, Haiyang Zheng

VOLUME 1: INTRODUCTION TO PTOLEMY II

Authors:
Shuvra S. Bhattacharyya
Elaine Cheong
John Davis, II
Mudit Goel
Bart Kienhuis
Christopher Hylands
Edward A. Lee
Jie Liu
Xiaojun Liu
Lukito Muliadi
Steve Neuendorffer
John Reekie
Neil Smyth
Jeff Tsay
Brian Vogel
Winthrop Williams
Yuhong Xiong
Yang Zhao
Haiyang Zheng

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
http://ptolemy.eecs.berkeley.edu

Document Version 3.0
for use with Ptolemy II 3.0
July 16, 2003

Memorandum UCB/ERL M03/27
Earlier versions:
- *UCB/ERL M02/23*
- *UCB/ERL M01/12*
- *UCB/ERL M99/40*

This project is supported by the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation, Chess (the Center for Hybrid and Embedded Software Systems), the State of California MICRO program, and the following companies: Agilent, Atmel, Cadence, Hitachi, Honeywell, National Semiconductor, Philips, and Wind River Systems.

[illegible]

VOLUME 1

INTRODUCTION TO PTOLEMY II

This volume describes how to construct Ptolemy II models for web-based modeling or building applications. The first chapter includes an overview of Ptolemy II software, and a brief description of each of the models of computation that have been implemented. It describes the package structure of the software, and includes as an appendix a brief tutorial on UML notation, which is used throughout the documentation to explain the structure of the software. The second chapter is a tutorial on building models using Vergil, a graphical user interface where models are built pictorially. The third chapter discusses the Ptolemy II expression language, which is used to set parameter values. The next chapter gives an overview of actor libraries. These three chapters, plus one of the domain chapters, will be sufficient for users to start building interesting models in the selected domain. The fifth chapter gives a tutorial on designing actors in Java.The sixth chapter explains MoML, the XML schema used by Vergil to store models. And the seventh chapter, the final one in this part, explains how to construct custom applets.

Volume 2 describes the software architecture of Ptolemy II, and volume 3 describes the domains, each of which implements a model of computation.

Contents

1

Introduction

Author: *Edward A. Lee*
Contributors: *The entire Ptolemy team*

1.1 Purpose

This document is the first of three volumes describing the Ptolemy II software. This first volume introduces the software and explains how to use it. The second volume documents the software architecture and explains how to extend it. The third volume documents the "domains," which realize models of computation. Some of these are relatively mature and established, and some are highly experimental. Indeed, a major part of the Ptolemy Project is experimentation with models of computation.

Ptolemy II is the current software infrastructure of the Ptolemy Project. For the participants in the Ptolemy Project, Ptolemy II is first and foremost a laboratory for experimenting with design techniques. It is published freely and in open-source form for several reasons. First, the software complements more traditional publication media, and serves as a clear, unambiguous, and complete description our research results. Second, the open architecture and open source encourages researchers to build their own methods, leveraging and extending the core infrastructure provided by the software. This creates a community where much of the dialog is through the software. Third, the freely available software encourages designers to try out the new design techniques that are introduced and give feedback to the Ptolemy Project. This helps guide further research. Fourth, the open source software encourages commercial providers of software tools to commercialize the research results, which then helps to maximize the impact of the work.

The Ptolemy Project is an informal group of researchers at U.C. Berkeley. There have been many participants in the project over the years (see "Acknowledgements" on page 28 for a list contributors to Ptolemy II). Ptolemy II is the third generation of design software to emerge from this group, with each generation bringing a new set of problems being addressed, new emphasis, and (largely) a new group of contributors.

1.1.1 Gabriel (1986-1991)

The first generation of software created by this was group was called Gabriel [16]. It was written in Lisp and aimed at signal processing. It was during the construction of Gabriel that synchronous dataflow (SDF) block diagrams and both sequential and parallel scheduling techniques for SDF models matured. Gabriel included code generators for programmable DSPs that produced efficient assembly code for certain processors (notably, Motorola processors). Gabriel included hardware/software co-simulators, where parallel code generators would produce assembly code which then ran on instruction set simulators within a hardware simulation of a multiprocessor architecture. Gabriel had a graphical user interface built on top of Vem, written by Rick Spickelmeyer, which was originally designed for schematic capture in VLSI CAD. It used Oct, which was the design database developed by the Berkeley CAD group under the leadership of Professor Richard Newton.

1.1.2 Ptolemy Classic (1990-1997)

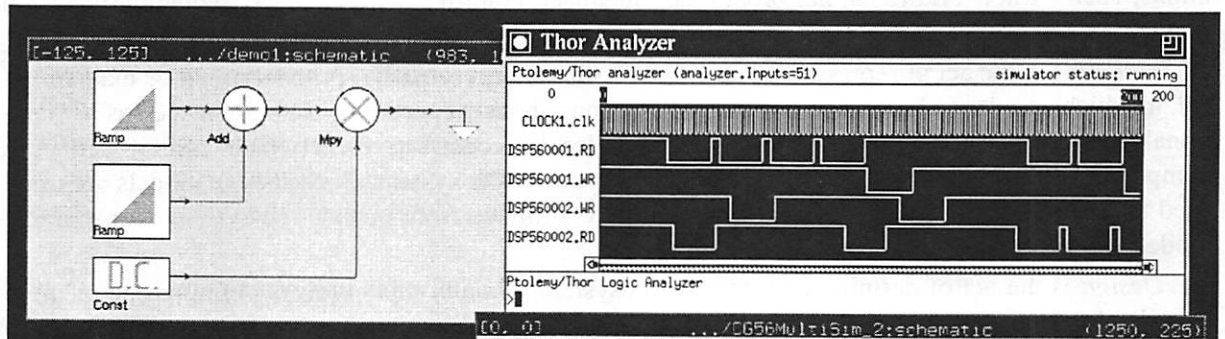
Ptolemy Classic, started jointly by Professors Edward Lee and Dave Messerschmitt in 1990, was written in C++ [19]. It was the first modeling environment to systematically support multiple models of computation, hierarchically combined. We ported the SDF capabilities from Gabriel, and also developed boolean dataflow (BDF), dynamic dataflow (DDF), multidimensional synchronous dataflow (MDSDF) and process networks (PN) domains. We also ported the DSP code generators, and created C and VHDL code generators as well. We developed the concept of "targets," which encapsulated knowledge about specific hardware platforms, and demonstrated construction of models that executed on attached embedded processors (such as S-bus cards with DSPs), including models that executed jointly on a Unix host and the attached embedded processor. We developed a discrete-event domain and demonstrated joint modeling of communication networks and signal processing, and also developed a hardware simulation domain called Thor, which was adapted from an open-source hardware simulator by the same name (see figure 1.1). We made major contributions to SDF scheduling techniques, including introducing the concept of "single appearance schedules" (which minimize generated code size and enable extensive use of inlining of generated code). We also introduced "higher-order components," which greatly increased the expressiveness of visual syntaxes. The Ptolemy Classic user interface was an extension of the Gabriel interface, still based on Oct and Vem, but extended by Tycho (written in Itcl, an object-oriented extension of Tcl/Tk). Portions of Ptolemy Classic were commercialized as part of the Agilent ADS system, and methods from Ptolemy Classic were used in Cadence's SPW system.

1.1.3 Ptolemy II (1996-?)

The Ptolemy Project (as it was now known) began working on Ptolemy II in 1996. The major reasons for starting over were to exploit the network integration, migrating code, built-in threading, and user-interface capabilities of Java. Ptolemy II introduced the notion of domain polymorphism (where components could be designed to be able to operate in multiple domains) and modal models (where finite state machines are combined hierarchically with other models of computation). We built a continuous-time domain, which combined with the modal modeling capability, yields hybrid system modeling. Ptolemy II has a sophisticated type system with type inference and data polymorphism (where components can be designed to operate on multiple data types), and a rich expression language. The concept of behavioral types emerged (where components and domains could have interface definitions that describe not just static structure, as with traditional type systems, but also dynamic behavior). Some (but not all) of the SDF capabilities from Ptolemy Classic were ported, and the heterochronous dataflow model was introduced. We contributed to a user-interface toolkit (called Diva) based on Java, built a user interface for Ptolemy II (called Vergil) based on Diva, designed a Java plotter (PtPlot), and introduced a 3-D animation domain. We built models that could be used as applets in a web browser. And we built numerous experimental domains that explored real-time and distributed computing (distributed discrete events (DDE), timed multitasking (TM), Giotto, and component interaction (CI)). As for code generation, the tactic in Ptolemy II is significantly different than that in Gabriel or Ptolemy Classic. Instead of components as generators, Ptolemy II uses a component-specialization framework built on top of a Java compiler toolkit called Soot. Ptolemy II uses XML for its persistent data representation, and has introduced the concept of migrating models.

1.1.4 Organization of this Document

This document is the first of three volumes. This first volume introduces the software and explains how to use it. It begins with a description of the rationale for the design in this chapter. The second


[-125, 125] .../demo1:schematic (983,
Ramp
Add
Mpy
Ramp
D.C.
Const
Thor Analyzer
Ptolemy/Thor analyzer (analyzer.Inputs=51)
simulator status: running
0
200
CLOCK1.clk
DSP560001.RD
DSP560001.WR
DSP560002.WR
DSP560002.RD
Ptolemy/Thor Logic Analyzer
[0, 0] .../CG56MultiSim_2:schematic (1250, 225)
Vcc
NC
DSP56000
DSP56000
F
DSP560002
DSP560

```
MOTOROLA DSP56000 SIMULATOR:

command1.cmd
RPC connection with Thor esta
load code1
Loading file:code1.lod
go
SIMULATION ABORTED
  x=        $000000000000 y=
  a=       $00000001000000 b=
             x1=   $000000  x0=
             y1=   $000000  y0=
 a2=   $00 a1=   $000001  a0=
 b2=   $00 b1=   $000000  b0=

 pc= $0011    sr=  $0310 omr=
 la= $0000    lc=  $0000
ssh= $0000   ssl=  $0000  sp=
cyc=000068  ictr= 000012 cnt1
P:$0010 0C0001          = JMP <
go
0>go
SIMULATION IN PROGRESS. Enter Ctrl-C to Halt. dev:0 pc:0011 cyc:68
GO [(from)location/R(reset)] [(to break number)#bn] [(occurrence):count]
```

```
MOTOROLA

command0.
RPC conne
load code
Loading f
go

SIMULATION IN PROGRESS. Enter Ctrl-C to Halt. dev:0 pc:0000 cyc:0
```

FIGURE 1.1. Ptolemy Classic screen image (from 1993) showing an SDF graph at the upper left that is automatically mapped and scheduled onto the two processor architecture, whose model is at the lower right (in the "Thor" domain). Assembly code for the two processor is generated, and then ISA simulators of the processors (provided by Motorola, lower left) interact with the Thor-domain simulation, resulting in the logic analyzer trace at the upper right.

chapter is a tutorial that explains how to use Ptolemy II via the Vergil visual editor. The third chapter explains the expression language, which is used extensively in Ptolemy II. The fourth chapter provides an overview of the actor libraries; note, however, that the most complete documentation for the actors is built in to the software, accessed through the "Get Documentation" command, obtained by right clicking on the actor icon. The fifth chapter explains how to write actors. The sixth chapter describes MoML, the XML schema used to store Ptolemy II models.

1.2 Modeling and Design

The Ptolemy project studies heterogeneous modeling, simulation, and design of concurrent systems. The focus is on *embedded systems* [66], particularly those that mix technologies including, for example, analog and digital electronics, hardware and software, and electronics and mechanical devices. The focus is also on systems that are complex in the sense that they mix widely different operations, such as networking, signal processing, feedback control, mode changes, sequential decision making, and user interfaces.

Modeling is the act of representing a system or subsystem formally. A model might be mathematical, in which case it can be viewed as a set of assertions about properties of the system such as its functionality or physical dimensions. A model can also be constructive, in which case it defines a computational procedure that mimics a set of properties of the system. Constructive models are often used to describe behavior of a system in response to stimulus from outside the system. Constructive models are also called executable models.

Design is the act of defining a system or subsystem. Usually this involves defining one or more models of the system and refining the models until the desired functionality is obtained within a set of constraints.

Design and modeling are obviously closely coupled. In some circumstances, models may be immutable, in the sense that they describe subsystems, constraints, or behaviors that are externally imposed on a design. For instance, they may describe a mechanical system that is not under design, but must be controlled by an electronic system that is under design.

Executable models are sometimes called *simulations*, an appropriate term when the executable model is clearly distinct from the system it models. However, in many electronic systems, a model that starts as a simulation mutates into a software implementation of the system. The distinction between the model and the system itself becomes blurred in this case. This is particularly true for embedded software.

1.2.1 Embedded Software

Embedded software is software that resides in devices that are not first-and-foremost computers. It is pervasive, appearing in automobiles, telephones, pagers, consumer electronics, toys, aircraft, trains, security systems, weapons systems, printers, modems, copiers, thermostats, manufacturing systems, appliances, etc. A technically active person probably interacts regularly with more pieces of embedded software than conventional software. A key feature of embedded software is that it engages the physical world, and hence has temporal constraints that desktop software does not share.

A major emphasis in Ptolemy II is on the methodology for defining and producing embedded software together with the systems within which it is embedded.

Executable models are constructed under a *model of computation*, which is the set of "laws of physics" that govern the interaction of components in the model. If the model is describing a mechanical system, then the model of computation may literally be the laws of physics. More commonly, however, it is a set of rules that are more abstract, and provide a framework within which a designer builds models. A set of rules that govern the interaction of components is called the *semantics* of the model of computation. A model of computation may have more than one semantics, in that there might be distinct sets of rules that impose identical constraints on behavior.

The choice of model of computation depends strongly on the type of model being constructed. For example, for a purely computational system that transforms a finite body of data into another finite body of data, the imperative semantics that is common in programming languages such as C, C++, Java, and MATLAB will be adequate. For modeling a mechanical system, the semantics needs to be able to handle concurrency and the time continuum, in which case a continuous-time model of computation such that found in Simulink, Saber, Hewlett-Packard's ADS, and VHDL-AMS is more appropriate.

The ability of a model to mutate into an implementation depends heavily on the model of computation that is used. Some models of computation, for example, are suitable for implementation only in customized hardware, while others are poorly matched to customized hardware because of their intrinsically sequential nature. Choosing an inappropriate model of computation may compromise the quality of design by leading the designer into a more costly or less reliable implementation.

A principle of the Ptolemy project is that the choices of models of computation strongly affect the quality of a system design.

For embedded systems, the most useful models of computation handle concurrency and time. This is because embedded systems consist typically of components that operate simultaneously and have multiple simultaneous sources of stimuli. In addition, they operate in a timed (real world) environment, where the timeliness of their response to stimuli may be as important as the correctness of the response.

The objective in Ptolemy II is to support the construction and interoperability of executable models that are built under a wide variety of models of computation.

Ptolemy II takes a component view of design, in that models are constructed as a set of interacting components. A model of computation governs the semantics of the interaction, and thus imposes a discipline on the interaction of components.

Component-based design in Ptolemy II involves disciplined interactions between components governed by a model of computation.

1.2.2 Actor-Oriented Design

Most (but not all) models of computation in Ptolemy II support *actor-oriented design*. This contrasts with (and complements) object-oriented design by emphasizing concurrency and communication between components. Components called actors execute and communicate with other actors in a model, as illustrated in figure 1.2. Like objects, actors have a well defined component interface. This interface abstracts the internal state and behavior of an actor, and restricts how an actor interacts with its environment. The interface includes ports that represent points of communication for an actor, and parameters which are used to configure the operation of an actor. Often, parameter values are part of the *a priori* configuration of an actor and do not change when a model is executed, but not always. The "port/parameters" shown in figure 1.2 function as both ports and parameters.

Central to actor-oriented design are the communication channels that pass data from one port to another according to some messaging scheme. Whereas with object-oriented design, components interact primarily by transferring control through method calls, in actor-oriented design, they interact by sending messages through channels. The use of channels to mediate communication implies that actors interact only with the channels that they are connected to and not directly with other actors.

Like actors, a model may also define an external interface; this interface is called its *hierarchical abstraction*. This interface consists of external ports and external parameters, which are distinct from the ports and parameters of the individual actors in the model. The external ports of a model can be connected by channels to other external ports of the model or to the ports of actors that compose the model. External parameters of a model can be used to determine the values of the parameters of actors inside the model.

Taken together, the concepts of models, actors, ports, parameters and channels describe the *abstract syntax* of actor-oriented design. This syntax can be represented concretely in several ways, such as graphically, as in figure 4, in XML as in figure 1.3, or in a program designed to a specific API (as in SystemC). Ptolemy II offers all three alternatives.

It is important to realize that the syntactic structure of an actor-oriented design says little about the semantics. The semantics is largely orthogonal to the syntax, and is determined by a model of computation. The model of computation might give operational rules for executing a model. These rules determine when actors perform internal computation, update their internal state, and perform external communication. The model of computation also defines the nature of communication between components.

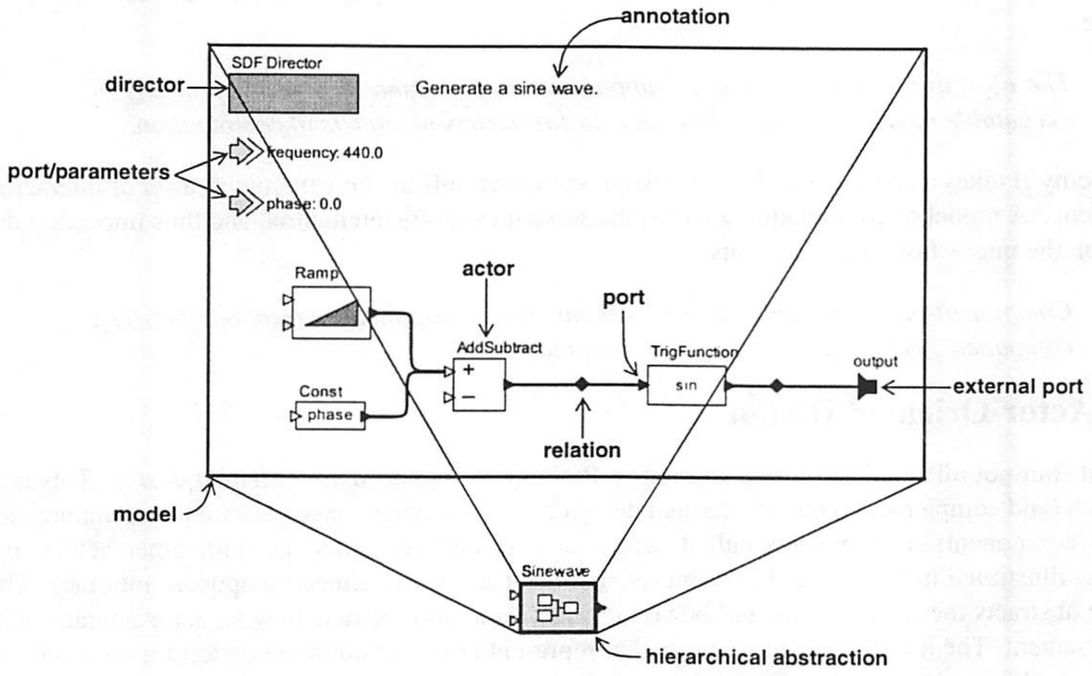


FIGURE 1.2. Illustration of an actor-oriented model (above) and its hierarchical abstraction (below).

Our notion of actor-oriented modeling is related to the work of Gul Agha and others. The term actor was introduced in the 1970's by Carl Hewitt of MIT to describe the concept of autonomous reasoning agents [48]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [1-5]. Agha's actors each have an independent thread of control and communicate via asynchronous message passing. We have further developed the term to embrace a larger family of models of concurrency that are often more constrained than general message passing. Our actors are still conceptually concurrent, but unlike Agha's actors, they need not have their own thread of control. Moreover, although communication is still through some form of message passing, it need not be strictly asynchronous.

Actor-oriented modeling has been around for some time, and is in widespread use through such programs as Simulink, from The Mathworks, LabView, from National Instruments, and many others. It is gaining broader legitimacy through the efforts of OMG in UML-2 [101], for example, some of which has its roots in the actor-oriented framework ROOM [116]. Many research projects are based on some form of actor-oriented modeling, but the Ptolemy Project is unique in the breadth of exploration of semantic alternatives and in the commitment made to a particular model of computation within a domain.

1.2.3 Architecture Design

Architecture description languages (ADLs), such as Wright [6] and Rapide [85], focus on formalisms for describing the rich sorts of component interactions that commonly arise in software architecture. Ptolemy II, by contrast, might be called an *architecture design language*, because its objective is not so much to describe existing interactions, but rather to promote coherent software architecture by imposing some structure on those interactions. Thus, while an ADL might focus on the compatibility of a sender and receiver in two distinct components, we would focus on a pattern of interactions among a set of components. Instead of, for example, verifying that a particular protocol in a single port-to-port interaction does not deadlock [6], we would focus on whether an assemblage of components can deadlock.

It is arguable that our approach is less modular, because components must be designed to the framework. Typical ADLs can describe pre-existing components, whereas in Ptolemy II, such pre-

```
<class name="Sinewave">
   <property name="samplingFrequency" value="8000.0"/>
   <property name="frequency" value="440.0"/>
   <property name="phase" value="0.0"/>
   <property name="SDF Director" class="ptolemy.domains.sdf.kernel.SDFDirector"/>
   <port name="output"><property name="output"/>
   <entity name="Ramp" class="ptolemy.actor.lib.Ramp">
      <property name="init" value="phase"/>
      <property name="step" value="frequency*2*PI/samplingFrequency"/>
   </entity>
   <entity name="TrigFunction" class="ptolemy.actor.lib.TrigFunction">
      <property name="function" value="sin" class="ptolemy.kernel.util.StringAttribute"/>
   </entity>
   <relation name="relation"/>
   <relation name="relation2"/>
   <link port="output" relation="relation2"/>
   <link port="Ramp.output" relation="relation"/>
   <link port="TrigFunction.input" relation="relation"/>
   <link port="TrigFunction.output" relation="relation2"/>
</class>
```

FIGURE 1.3. An XML representation of a simplified sinewave source.

existing components would have to wrapped in Ptolemy II actors. Moreover, designing components to a particular interface may limit their reusability, and in fact the interface may not match their needs well. All of these are valid points, and indeed a major part of our research effort is to ameliorate these limitations. The net effect, we believe, is an approach that is much more powerful than ADLs.

First, we design components to be *domain polymorphic*, meaning that they can interact with other components within a wide variety of domains. In other words, instead of coming up with an ADL that can describe a number of different interaction mechanisms, we have come up with an architecture where components can be easily designed to interact in a number of ways. We argue that this makes the components more reusable, not less, because disciplined interaction within a well-defined semantics is possible. By contrast, with pre-existing components that have rigid interfaces, the best we can hope for is ad-hoc synthesis of adapters between incompatible interfaces, something that is likely to lead to designs that are very difficult to understand and to verify. Whereas ADLs draw an analogy between compatibility of interfaces and type checking [6], we use a technique much more powerful than type checking alone, namely polymorphism [68].

Second, to avoid the problem that a particular interaction mechanism may not fit the needs of a component well, we provide a rich set of interaction mechanisms embodied in the Ptolemy II domains. The domains force component designers to think about the overall pattern of interactions, and trade off uniformity for expressiveness. Where expressiveness is paramount, the ability of Ptolemy II to hierarchically mix domains offers essentially the same richness of more ad-hoc designs, but with much more discipline. By contrast, a non-trivial component designed without such structure is likely to use a *melange*, or ad-hoc mixture of interaction mechanisms, making it difficult to embed it within a comprehensible system.

Third, whereas an ADL might choose a particular model of computation to provide it with a formal structure, such as CSP for Wright [6], we have developed a more abstract formal framework that describes models of computation at a meta level [71]. This means that we do not have to perform awkward translations to describe one model of computation in terms of another. For example, stream based communication via FIFO channels are awkward in Wright [6].

We make these ideas concrete by describing the models of computation implemented in the Ptolemy II domains.

1.3 Models of Computation

There is a rich variety of models of computation that deal with concurrency and time in different ways. Each gives an interaction mechanism for components. The utility of a model of computation stems from the modeling properties that apply to all similar models. For many models of computation these properties are derived through formal mathematics. Depending on the model of computation, the model may be determinate [55], statically schedulable [72], or time safe [47]. Because of its modeling properties, a model of computation represents a style of modeling that is useful in any circumstance where those properties are desirable. In other words, models of computation form design patterns of component interaction, in the same sense that Gamma, et al. describe design patterns in object oriented languages [35].

For a particular application, an appropriate model of computation does not impose unnecessary constraints, and at the same time is constrained enough to result in useful derived properties. For example, by restricting the design space to synchronous designs, Scenic [74] enables cycle-driven simulation [41], which greatly improves execution efficiency over more general discrete-event models of computation (such as that found in VHDL). However, for applications with multirate behavior, synchronous design can be constraining. In such cases, a less constrained model of computation, such as synchronous dataflow [72] or Kahn process networks [55] may be more appropriate. One drawback of this relaxation of synchronous design constraints is that buffering becomes more difficult to analyze. On the other hand, techniques exist for synchronous dataflow that allow co-optimization of memory usage and execution latency [118] that would otherwise be difficult to apply to a multirate system. Selecting an appropriate model of computation for a particular application is often difficult, but this is a problem we should embrace instead of avoiding.

In this section, we describe models of computation that are implemented in Ptolemy II domains. Our focus has been on models of computation that are most useful for embedded systems. All of these can lend a semantics to the same bubble-and-arc, or block-and-arrow diagram shown in figure 1.4. Ptolemy II models are (clustered, or hierarchical) graphs of the form of figure 1.4, where the nodes are *entities* and the arcs are *relations*. For most domains, the entities are *actors* (entities with functionality) and the relations connecting them represent communication between actors.

1.3.1 Component Interaction - CI

The component interaction (CI) domain, created by Xiaojun Liu and Yang Zhao, models systems that blend data-driven and demand-driven styles of computation. As an example, the interaction between a web server and a browser is mostly demand-driven. When the user clicks on a link in the browser, it pulls the corresponding page from the web server. A stock-quote service can use a data-driven style of computation. The server generates events when stock prices change. The data drive the clients to update their displayed information. Such push/pull interaction between a data producer and consumer is common in distributed systems, and has been included in middleware services, most notably in the CORBA event service. These services motivated the design of this domain to study the interaction models in distributed systems, such as stock-quote services, traffic or weather information systems. Other applications include database systems, file systems, and the Click modular router [57].

An actor in a CI model can be active, which means it possesses its own thread of execution. For example, an interrupt source of an embedded system can be modeled as an active source actor. Such a source generates events asynchronously with respect to the software execution on the embedded processor. CI models can be used to simulate and study how the embedded software handles the asynchronous events, such as external interrupts and asynchronous I/O.

1.3.2 Communicating Sequential Processes - CSP

In the CSP domain (communicating sequential processes), created by Neil Smyth [117], actors

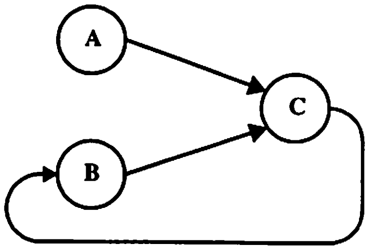


FIGURE 1.4. A single *syntax* (bubble-and-arc or block-and-arrow diagram) can have a number of possible *semantics* (interpretations).

represent concurrently executing processes, implemented as Java threads. These processes communicate by atomic, instantaneous actions called *rendezvous* (or sometimes, *synchronous message passing*). If two processes are to communicate, and one reaches the point first at which it is ready to communicate, then it stalls until the other process is ready to communicate. "Atomic" means that the two processes are simultaneously involved in the exchange, and that the exchange is initiated and completed in a single uninterruptable step. Examples of rendezvous models include Hoare's *communicating sequential processes* (CSP) [51] and Milner's *calculus of communicating systems* (CCS) [90]. This model of computation has been realized in a number of concurrent programming languages, including Lotos and Occam.

Rendezvous models are particularly well-matched to applications where resource sharing is a key element, such as client-server database models and multitasking or multiplexing of hardware resources. A key feature of rendezvous-based models is their ability to cleanly model nondeterminate interactions. The CSP domain implements both conditional send and conditional receive. It also includes an experimental timed extension.

1.3.3 Continuous Time - CT

In the CT domain (continuous time), created Jie Liu [81], actors represent components that interact via continuous-time signals. Actors typically specify algebraic or differential relations between inputs and outputs. The job of the director in the domain is to find a fixed-point, i.e., a set of continuous-time functions that satisfy all the relations.

The CT domain includes an extensible set of differential equation solvers. The domain, therefore, is useful for modeling physical systems with linear or nonlinear algebraic/differential equation descriptions, such as analog circuits and many mechanical systems. Its model of computation is similar to that used in Simulink, Saber, and VHDL-AMS, and is closely related to that in Spice circuit simulators.

Mixed Signal Models. Embedded systems frequently contain components that are best modeled using differential equations, such as MEMS and other mechanical components, analog circuits, and microwave circuits. These components, however, interact with an electronic system that may serve as a controller or a recipient of sensor data. This electronic system may be digital. Joint modeling of a continuous subsystem with digital electronics is known as *mixed signal modeling* [82]. The CT domain is designed to interoperate with other Ptolemy domains, such as DE, to achieve mixed signal modeling. To support such modeling, the CT domain models of discrete events as Dirac delta functions. It also includes the ability to precisely detect threshold crossings to produce discrete events.

Modal Models and Hybrid Systems. Physical systems often have simple models that are only valid over a certain regime of operation. Outside that regime, another model may be appropriate. A *modal model* is one that switches between these simple models when the system transitions between regimes. The CT domain interoperates with the FSM domain to create modal models. Such modal models are often called *hybrid systems.*

1.3.4 Discrete-Events - DE

In the discrete-event (DE) domain, created by Lukito Muliadi [94], the actors communicate via sequences of events placed in time, along a real time line. An *event* consists of a *value* and *time stamp.* Actors can either be processes that react to events (implemented as Java threads) or functions that fire when new events are supplied. This model of computation is popular for specifying digital hardware and for simulating telecommunications systems, and has been realized in a large number of simulation environments, simulation languages, and hardware description languages, including VHDL and Verilog.

DE models are excellent descriptions of concurrent hardware, although increasingly the globally consistent notion of time is problematic. In particular, it over-specifies (or over-models) systems where maintaining such a globally consistent notion is difficult, including large VLSI chips with high clock rates. Every event is placed precisely on a globally consistent time line.

The DE domain implements a fairly sophisticated discrete-event simulator. DE simulators in general need to maintain a global queue of pending events sorted by time stamp (this is called a *priority queue*). This can be fairly expensive, since inserting new events into the list requires searching for the right position at which to insert it. The DE domain uses a calendar queue data structure [17] for the global event queue. A calendar queue may be thought of as a hashtable that uses quantized time as a hashing function. As such, both enqueue and dequeue operations can be done in time that is independent of the number of events in the queue.

In addition, the DE domain gives deterministic semantics to simultaneous events, unlike most competing discrete-event simulators. This means that for any two events with the same time stamp, the order in which they are processed can be inferred from the structure of the model. This is done by analyzing the graph structure of the model for data precedences so that in the event of simultaneous time stamps, events can be sorted according to a secondary criterion given by their precedence relationships. VHDL, for example, uses delta time to accomplish the same objective.

1.3.5 Distributed Discrete Events - DDE

The distributed discrete-event (DDE) domain, created by John Davis [26], can be viewed either as a variant of DE or as a variant of PN (described below). Still highly experimental, it addresses a key problem with discrete-event modeling, namely that the global event queue imposes a central point of control on a model, greatly limiting the ability to distribute a model over a network. Distributing models might be necessary either to preserve intellectual property, to conserve network bandwidth, or to exploit parallel computing resources.

The DDE domain maintains a local notion of time on each connection between actors, instead of a single globally consistent notion of time. Each actor is a process, implemented as a Java thread, that can advance its local time to the minimum of the local times on each of its input connections. The domain systematizes the transmission of null events, which in effect provide guarantees that no event will be supplied with a time stamp less than some specified value.

1.3.6 Discrete Time - DT

The discrete-time (DT) domain, written by Chamberlain Fong [32], extends the SDF domain (described below) with a notion of time between tokens. Communication between actors takes the form of a sequence of tokens where the time between tokens is uniform. Multirate models, where distinct connections have distinct time intervals between tokens, are also supported. There is considerable subtlety in this domain when multirate components are used. The semantics is defined so that component behavior is always causal, in that outputs whose values depend on inputs are never produced at times prior to those of the inputs.

1.3.7 Finite-State Machines - FSM

The finite-state machine (FSM) domain, written by Xiaojun Liu, is radically different from the other Ptolemy II domains. The entities in this domain represent not actors but rather *state*, and the connections represent *transitions* between states. Execution is a strictly ordered sequence of state transitions. The FSM domain leverages the built-in expression language in Ptolemy II to evaluate *guards*, which determine when state transitions can be taken.

FSM models are excellent for expressing control logic and for building modal models (models with distinct modes of operation, where behavior is different in each mode). FSM models are amenable to in-depth formal analysis, and thus can be used to avoid surprising behavior.

**Charts*. FSM models have some key weaknesses. First, at a very fundamental level, they are not as expressive as the other models of computation described here. They are not sufficiently rich to describe all partial recursive functions. However, this weakness is acceptable in light of the formal analysis that becomes possible. Many questions about designs are decidable for FSMs and undecidable for other models of computation. A second key weakness is that the number of states can get very large even in the face of only modest complexity. This makes the models unwieldy.

Both problems can often be solved by using FSMs in combination with concurrent models of computation. This was first noted by David Harel, who introduced that Statecharts formalism. Statecharts combine a loose version of synchronous-reactive modeling (described below) with FSMs [42]. FSMs have also been combined with differential equations, yielding the so-called *hybrid systems* model of computation [45].

The FSM domain in Ptolemy II can be hierarchically combined with other domains. We call the resulting formalism "*charts" (pronounced "starcharts") where the star represents a wildcard [38]. Since most other domains represent concurrent computations, *charts model concurrent finite state machines with a variety of concurrency semantics. When combined with CT, they yield hybrid systems and modal models. When combined with SR (described below), they yield something close to Statecharts. When combined with process networks, they resemble SDL [115].

1.3.8 Process Networks - PN

In the process networks (PN) domain, created by Mudit Goel [39], processes communicate by sending messages through channels that can buffer the messages. The sender of the message need not wait for the receiver to be ready to receive the message. This style of communication is often called asynchronous message passing. There are several variants of this technique, but the PN domain specifically implements one that ensures determinate computation, namely Kahn process networks [55].

In the PN model of computation, the arcs represent sequences of data values (tokens), and the entities represent functions that map input sequences into output sequences. Certain technical restrictions on these functions are necessary to ensure determinacy, meaning that the sequences are fully specified. In particular, the function implemented by an entity must be *prefix monotonic*. The PN domain realizes a subclass of such functions, first described by Kahn and MacQueen [56], where *blocking reads* ensure monotonicity.

PN models are loosely coupled, and hence relatively easy to parallelize or distribute. They can be implemented efficiently in both software and hardware, and hence leave implementation options open. A key weakness of PN models is that they are awkward for specifying control logic, although much of this awkwardness may be ameliorated by combining them with FSM.

The PN domain in Ptolemy II has a highly experimental timed extension. This adds to the blocking reads a method for stalling processes until time advances. We anticipate that this timed extension will make interoperation with timed domains much more practical.

1.3.9 Synchronous Dataflow - SDF

The synchronous dataflow (SDF) domain, created by Steve Neuendorffer, handles regular computations that operate on streams. Dataflow models, popular in signal processing, are a special case of process networks (for the complete explanation of this, see [70]). Dataflow models construct processes of a process network as sequences of atomic actor *firings*. Synchronous dataflow (SDF) is a particularly restricted special case with the extremely useful property that deadlock and boundedness are decidable. Moreover, the schedule of firings, parallel or sequential, is computable statically, making SDF an extremely useful specification formalism for embedded real-time software and for hardware.

Certain generalizations sometimes yield to similar analysis. Boolean dataflow (BDF) models sometimes yield to deadlock and boundedness analysis, although fundamentally these questions are undecidable. Dynamic dataflow (DDF) uses only run-time analysis, and thus makes no attempt to statically answer questions about deadlock and boundedness. Neither a BDF nor DDF domain has yet been written in Ptolemy II. Process networks (PN) serves in the interim to handle computations that do not match the restrictions of SDF.

1.3.10 Giotto

The Giotto domain, created by Christoph Meyr Kirsch, realizes a model of computation developed by Tom Henzinger, Christoph Kirsch, Ben Horowitz and Haiyang Zheng [44]. This domain has a time-triggered flavor, where each actor is invoked periodically with a specified period. The domain is designed to work with the FSM domain to realize modal models. It is intended for hard-real-time systems, where resource allocation is precomputed.

1.3.11 Synchronous/Reactive - SR

In the synchronous/reactive (SR) domain, written by Paul Whitaker [121] implements a model of computation [11] where the arcs represent data values that are aligned with global clock ticks. Thus, they are discrete signals, but unlike discrete time, a signal need not have a value at every clock tick. The entities represent relations between input and output values at each tick, and are usually partial functions with certain technical restrictions to ensure determinacy. Examples of languages that use the SR model of computation include Esterel [13], Signal [12], Lustre [23], and Argos [86].

SR models are excellent for applications with concurrent and complex control logic. Because of the tight synchronization, safety-critical real-time applications are a good match. However, also because of the tight synchronization, some applications are overspecified in the SR model, limiting the implementation alternatives. Moreover, in most realizations, modularity is compromised by the need to seek a global fixed point at each clock tick. The SR domain implementation in Ptolemy II is similar to the SR implementation in Ptolemy Classic by Stephen Edwards[28].

1.3.12 Timed Multitasking - TM

The timed multitasking (TM) domain, created by Jie Liu [80], supports the design of concurrent real-time software. It assumes an underlying priority-driven preemptive scheduler, such as that typically found in a real-time operating systems (RTOS). But the behavior of models is more deterministic than that obtained by more ad hoc uses of an RTOS.

In TM, each actor executes (conceptually) as a concurrent task. It is a timed domain, meaning that there is a notion of "model time" that advances monotonically and uniformly. Each actor has a specified execution time *T*, and it delays the production of the outputs until it has had access to the CPU for that specified amount of time (in model time, which may or may not match real time). Actors execute when they receive new inputs, so the execution is event driven. Conceptually, the actor begins execution at some time t, and its output is produced at time $t + T + P$, where T is the declared execution time, and P is the amount of time where the actor is suspended due to being preempted by a higher priority actor. At any given model time t, the task with the highest priority that has received inputs but not yet produced its outputs has the CPU. All other tasks are suspended.

TM offers a way to design real-time systems that is more deterministic than ad hoc uses of an RTOS. In particular, typically, a task produces outputs at a time that depends on the actual execution time of the task, rather than on some declared parameter. This means that consumers of that data may or may not see updates to the data, depending on when their execution occurs relative to the actual execution time. Thus, the computational results that are produced depend on the actual execution time. TM avoids this by declaring the time that elapses before production of the outputs. By maintaining model time correctly, TM ensures that the data computation is deterministic, irrespective of actual execution time.

1.4 Choosing Models of Computation

The rich variety of concurrent models of computation outlined in the previous section can be daunting to a designer faced with having to select them. Most designers today do not face this choice because they get exposed to only one or two. This is changing, however, as the level of abstraction and domain-specificity of design software both rise. We expect that sophisticated and highly visual user interfaces will be needed to enable designers to cope with this heterogeneity.

An essential difference between concurrent models of computation is their modeling of time. Some are very explicit by taking time to be a real number that advances uniformly, and placing events on a time line or evolving continuous signals along the time line. Others are more abstract and take time to be discrete. Others are still more abstract and take time to be merely a constraint imposed by causality. This latter interpretation results in time that is partially ordered, and explains much of the expressiveness in process networks and rendezvous-based models of computation. Partially ordered time provides a mathematical framework for formally analyzing and comparing models of computation [71].

A grand unified approach to modeling would seek a concurrent model of computation that serves all purposes. This could be accomplished by creating a *melange*, a mixture of all of the above, but such a mixture would be extremely complex and difficult to use, and synthesis and simulation tools would be difficult to design.

Another alternative would be to choose one concurrent model of computation, say the rendezvous model, and show that all the others are subsumed as special cases. This is relatively easy to do, in theory. It is the premise of Wright [6] and Metropolis (http://www.gigascale.org/metropolis/), for example. Most of these models of computation are sufficiently expressive to be able to subsume most of the others. However, this fails to acknowledge the strengths and weaknesses of each model of computation. Wright, for example, uses rendezvous, which is very good at resource management, but very awkward for loosely coupled data-oriented computations. Asynchronous message passing is the reverse, where resource management is awkward, but data-oriented computations are natural[1]. Thus, to design interesting systems, designers need to use heterogeneous models.

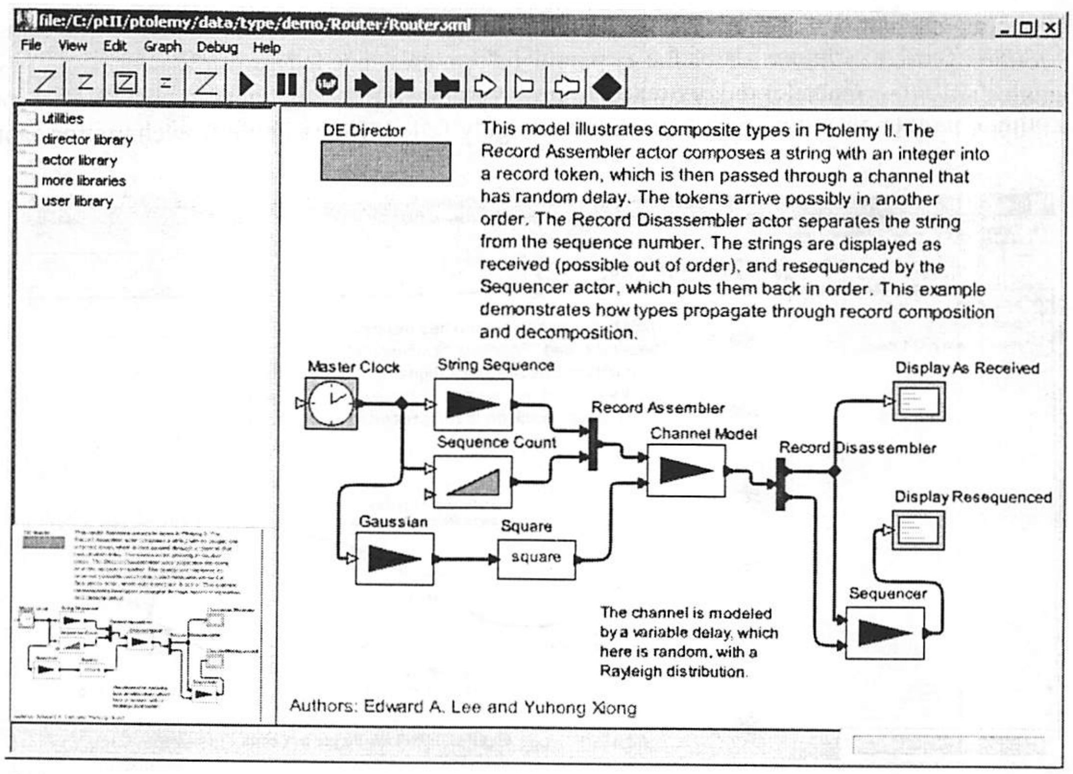
1.5 Visual Syntaxes

Visual depictions of systems have always held a strong human appeal, making them extremely effective in conveying information about a design. Many of the domains of interest in the Ptolemy project use such depictions to completely and formally specify models.

One of the principles of the Ptolemy project is that visual depictions of systems can help to offset the increased complexity that is introduced by heterogeneous modeling.

These visual depictions offer an alternative *syntax* to associate with the semantics of a model of computation. Visual syntaxes can be every bit as precise and complete as textual syntaxes, particularly when they are judiciously combined with textual syntaxes.

Figures 1.5 and 1.6 show two different visual renditions of Ptolemy II models. Both renditions are constructed in Vergil, the visual editor framework in Ptolemy II designed by Steve Neuendorffer. In figure 1.5, a Ptolemy II model is shown as a block diagram, which is an appropriate rendition for many discrete event models. In this particular example, records are constructed at the left by composing strings with integers representing a sequence number. The records are launched into a network that



This model illustrates composite types in Ptolemy II. The Record Assembler actor composes a string with an integer into a record token, which is then passed through a channel that has random delay. The tokens arrive possibly in another order. The Record Disassembler actor separates the string from the sequence number. The strings are displayed as received (possible out of order), and resequenced by the Sequencer actor, which puts them back in order. This example demonstrates how types propagate through record composition and decomposition.

The channel is modeled by a variable delay, which here is random, with a Rayleigh distribution.

Authors: Edward A. Lee and Yuhong Xiong

FIGURE 1.5. Visual rendition of a Ptolemy II model as a block diagram in Vergil (in the DE domain).

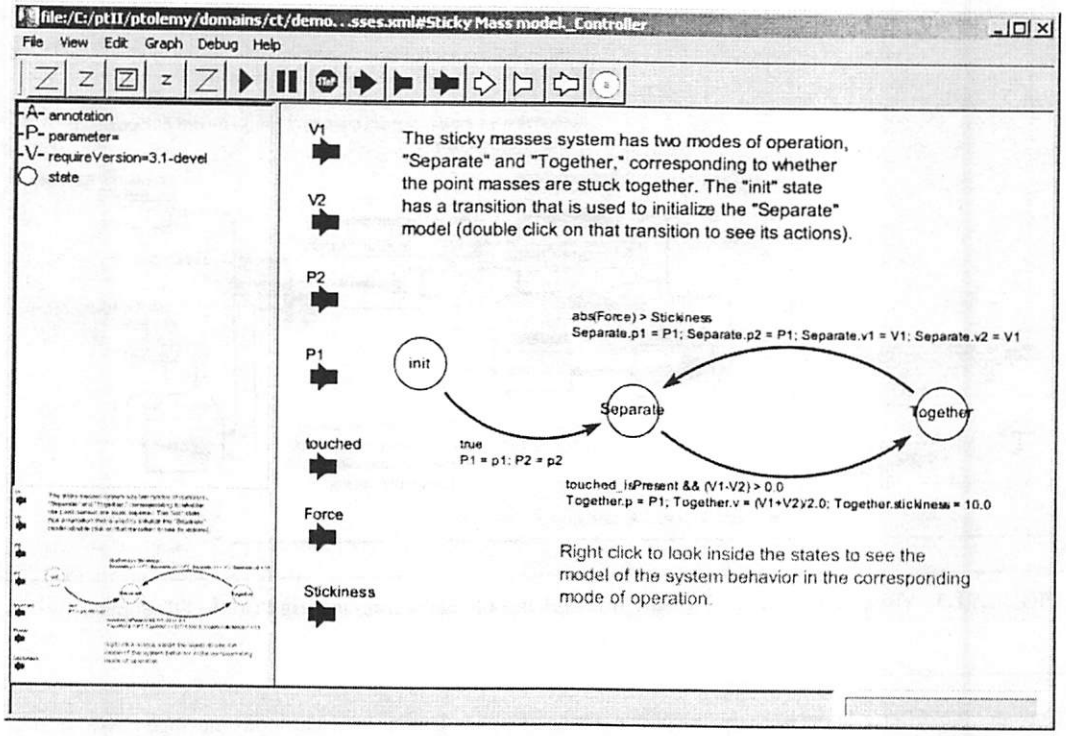
1. Consider the difference between the telephone (rendezvous) and email (asynchronous message passing). If you are trying to schedule a meeting between four busy people, getting them all on a conference call would lead to a quick resolution of the meeting schedule. Scheduling the meeting by email could take several days, and may in fact never converge. Other sorts of communication, however, are far more efficient by email.

introduces random delay. The records may arrive at the right out of order, but the Sequence actor is used to re-order them using the sequence number.

Figure 1.6 also shows a visual rendition of a Ptolemy II model, but now, the components are represented by circles, and the connections between components are represented by labeled arcs. This visual syntax is a familiar way to represent finite state machines (FSMs). Each circle represents a state of the model, and the arcs represent transitions between states. The particular example in the figure comes from a hybrid system model, where the two states, Separate and Together, represent two different modes of operation of a continuous-time system. The arcs are labeled with two lines, the first of which is a *guard*, and the second of which is an *action*. The guard is a boolean-valued textual expression that specifies when the transition should be taken, and the action is a sequence of commands that are executed when the transition is taken.

The visual renditions in figures 1.5 and 1.6 are both constructed using the same underlying infrastructure, Vergil, built by Stephen Neuendorffer. Vergil, in turn, in built on top of a GUI package called Diva, developed by John Reekie and Michael Shilman at Berkeley. Diva, in turn, is built on top of Swing and Java 2D, which are part of the Java platform from Sun Microsystems. In Vergil, a visual editor is constructed as an assembly of components in a Ptolemy II model. Thus, the system is configurable and customizable, and a great deal of infrastructure can be shared between the two distinct visual editors of figures 1.5 and 1.6.

Visual representations of models have a mixed history. In circuit design, schematic diagrams used to be routinely used to capture all of the essential information needed to implement some systems. Schematics are often replaced today by text in hardware description languages such as VHDL or Verilog. In other contexts, visual representations have largely failed, for example flowcharts for capturing


file:/C:/ptII/ptolemy/domains/ct/demo...sses.xml#Sticky Mass model._Controller
File
View
Edit
Graph
Debug
Help
-A- annotation
-P- parameter=
-V- requireVersion=3.1-devel
state
V1
V2
P2
P1
touched
Force
Stickiness
init
Separate
Together
abs(Force) > Stickiness
Separate.p1 = P1; Separate.p2 = P1; Separate.v1 = V1; Separate.v2 = V1
true
P1 = p1; P2 = p2
touched_isPresent && (V1-V2) > 0.0
Together.p = P1; Together.v = (V1+V2)/2.0; Together.stickiness = 10.0

The sticky masses system has two modes of operation, "Separate" and "Together," corresponding to whether the point masses are stuck together. The "init" state has a transition that is used to initialize the "Separate" model (double click on that transition to see its actions).

Right click to look inside the states to see the model of the system behavior in the corresponding mode of operation.

FIGURE 1.6. Visual rendition of a Ptolemy II model as a state transition diagram in Vergil (FSM domain).

the behavior of software. Recently, a number of innovative visual formalisms have been garnering support, including visual dataflow, hierarchical concurrent finite state machines, and object models. The UML visual language for object modeling has been receiving a great deal of attention. The static structure diagrams of UML, in fact, are used fairly extensively in the design of Ptolemy II itself (see appendix A of this chapter). Moreover, the Statecharts diagrams of UML are very similar to a hierarchical composition of the FSM and SR domains in Ptolemy II.

A subset of visual languages that are recognizable as "block diagrams" represent concurrent systems. There are many possible concurrency semantics (and many possible models of computation) associated with such diagrams. Formalizing the semantics is essential if these diagrams are to be used for system specification and design. Ptolemy II supports exploration of the possible concurrency semantics. A principle of the project is that the strengths and weaknesses of these alternatives make them complementary rather than competitive. Thus, interoperability of diverse models is essential.

1.6 Ptolemy II Architecture

Ptolemy II offers a unified infrastructure for implementations of a number of models of computation. The overall architecture consists of a set of packages that provide generic support for all models of computation and a set of packages that provide more specialized support for particular models of computation. Examples of the former include packages that contain math libraries, graph algorithms, an interpreted expression language, signal plotters, and interfaces to media capabilities such as audio. Examples of the latter include packages that support clustered graph representations of models, packages that support executable models, and *domains*, which are packages that implement a particular model of computation.

Ptolemy II is modular, with a careful package structure that supports a layered approach. The *core packages* support the data model, or *abstract syntax*, of Ptolemy II designs. They also provide the *abstract semantics* that allows domains to interoperate with maximum information hiding. The *UI packages* provide support for our XML file format, called MoML, and a visual interface for constructing models graphically. The *library packages* provide actor libraries that are *domain polymorphic*, meaning that they can operate in a variety of domains. And finally, the *domain packages* provide domains, each of which implements a model of computation, and some of which provide their own, domain-specific actor libraries.

1.6.1 Core Packages

The core packages are shown in figure 1.7. This is a UML package diagram. The name of each package is in the tab at the top of each box. Subpackages are contained within their parent package. Dependencies between packages are shown by dotted lines with arrow heads. For example, *actor* depends on *kernel* which depends on *kernel.util*. *Actor* also depends on *data* and *graph*. The role of each package is explained below.

actor	This package supports executable entities that receive and send data through ports. It includes both untyped and typed actors. For typed actors, it implements a sophisticated type system that supports polymorphism. It includes the base class Director that is extended in domains to control the execution of a model.
actor.lib	This subpackage and its subpackages contain domain polymorphic actors. The actor.lib package is discussed further in section 1.6.4.
actor.parameters	

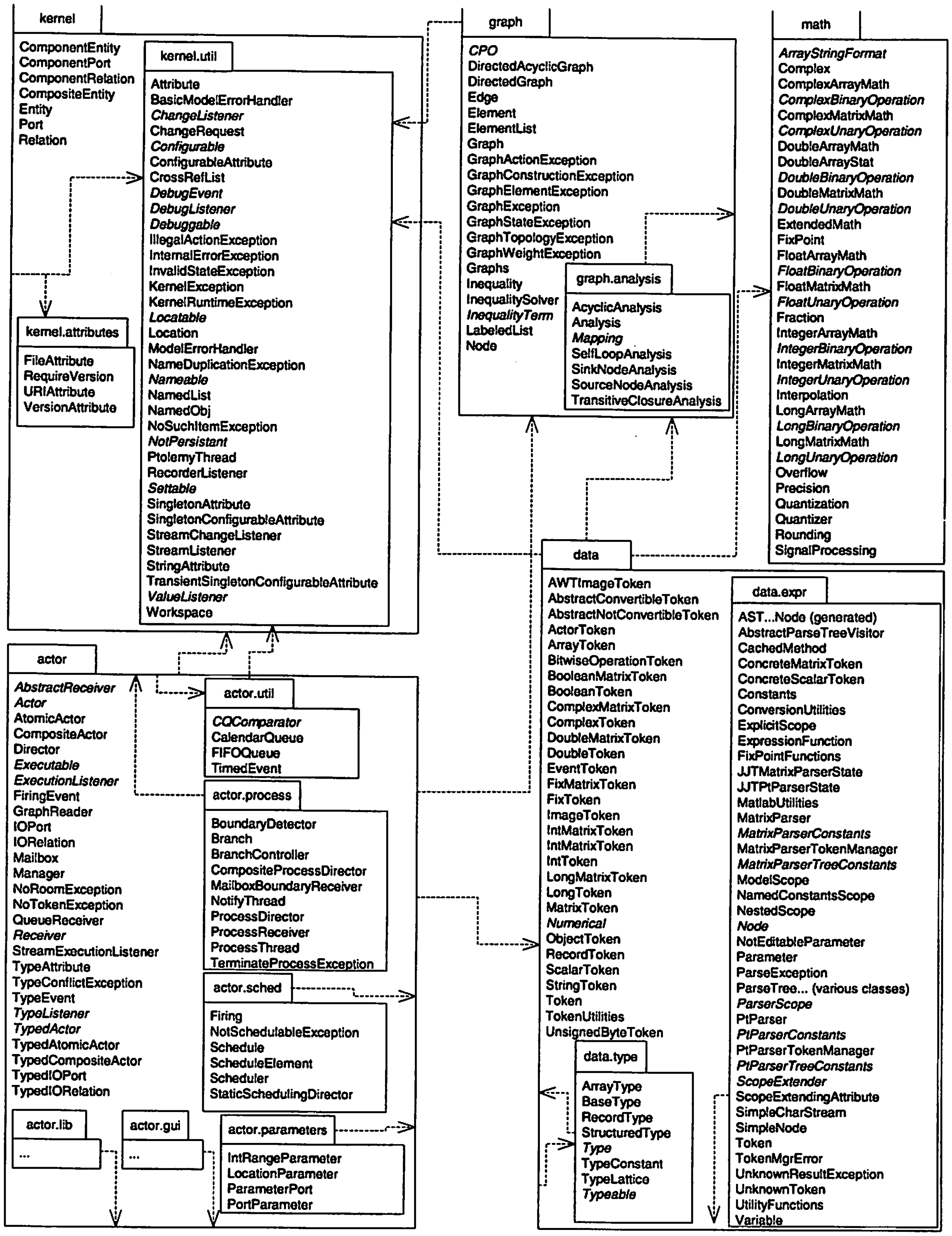


FIGURE 1.7. The core packages shown here support the data model (abstract syntax) and the actor model, (abstract semantics) of Ptolemy II designs.

	This subpackage provides specialized parameters for specifying locations, ranges of values, etc.
actor.process	This subpackage provides infrastructure for domains where actors are processes implemented on top of Java threads.
actor.sched	This subpackage provides infrastructure for domains where actors are statically scheduled by the director, or where there is static analysis of the topology of a model associated with scheduling.
actor.util	This subpackage contains utilities that support directors in various domains. Specifically, it contains a simple FIFO Queue and a sophisticated priority queue called a calendar queue.
copernicus	This subpackage contains the "actor specialization" infrastructure (Java code generation).
data	This package provides classes that encapsulate and manipulate data that is transported between actors in Ptolemy models. The key class is the Token class, which defines a set of polymorphic methods for operating on tokens, such as add(), subtract(), etc.
data.expr	This class supports an extensible expression language and an interpreter for that language. Parameters can have values specified by expressions. These expressions may refer to other parameters. Dependencies between parameters are handled transparently, as in a spreadsheet, where updating the value of one will result in the update of all those that depend on it.
data.type	This package contains classes and interfaces for the type system.
graph	This package provides algorithms for manipulating and analyzing mathematical graphs. This package is expected to supply a growing library of algorithms. These algorithms support scheduling and analysis of Ptolemy II models.
kernel	This package provides the software architecture for the Ptolemy II data model, or *abstract syntax*. This abstract syntax has the structure of clustered graphs. The classes in this package support *entities* with *ports*, and *relations* that connect the ports. Clustering is where a collection of entities is encapsulated in a single *composite entity*, and a subset of the ports of the inside entities are exposed as ports of the composite entity.
kernel.attributes	This subpackage of the kernel package provides specialized attributes such as FileAttribute, which is used in actors to specify a file or URL.
kernel.util	This subpackage of the kernel package provides a collection of utility classes that do not depend on the kernel package. It is separated into a subpackage so that these utility classes can be used without the kernel. The utilities include a collection of exceptions, classes supporting named objects with attributes, lists of named objects, a specialized cross-reference list class, and a thread class that helps Ptolemy keep track of executing threads.
math	This package encapsulates 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.
matlab	This package contains the MATLAB interface.

util This package contains various Ptolemy-independent utilities, such as string utilities and XML utilities.

1.6.2 Overview of Key Classes

Some of the key classes in Ptolemy II are shown in figure 1.8. This is a UML static structure diagram (see appendix A of this chapter). The key syntactic elements are boxes, which represent classes, the hollow arrow, which indicates generalization (or subclassing), and other lines, which indicate associations. Some lines have a small diamond, which indicates aggregation. The details of these classes will be discussed in subsequent chapters.

Instances of all of the classes shown can have names; they all implement the Nameable interface. Most of the classes generalize NamedObj, which in addition to being nameable can have a list of attributes associated with it. Attributes themselves are instances of NamedObj.

Entity, Port, and Relation are three key classes that extend NamedObj. These classes define the

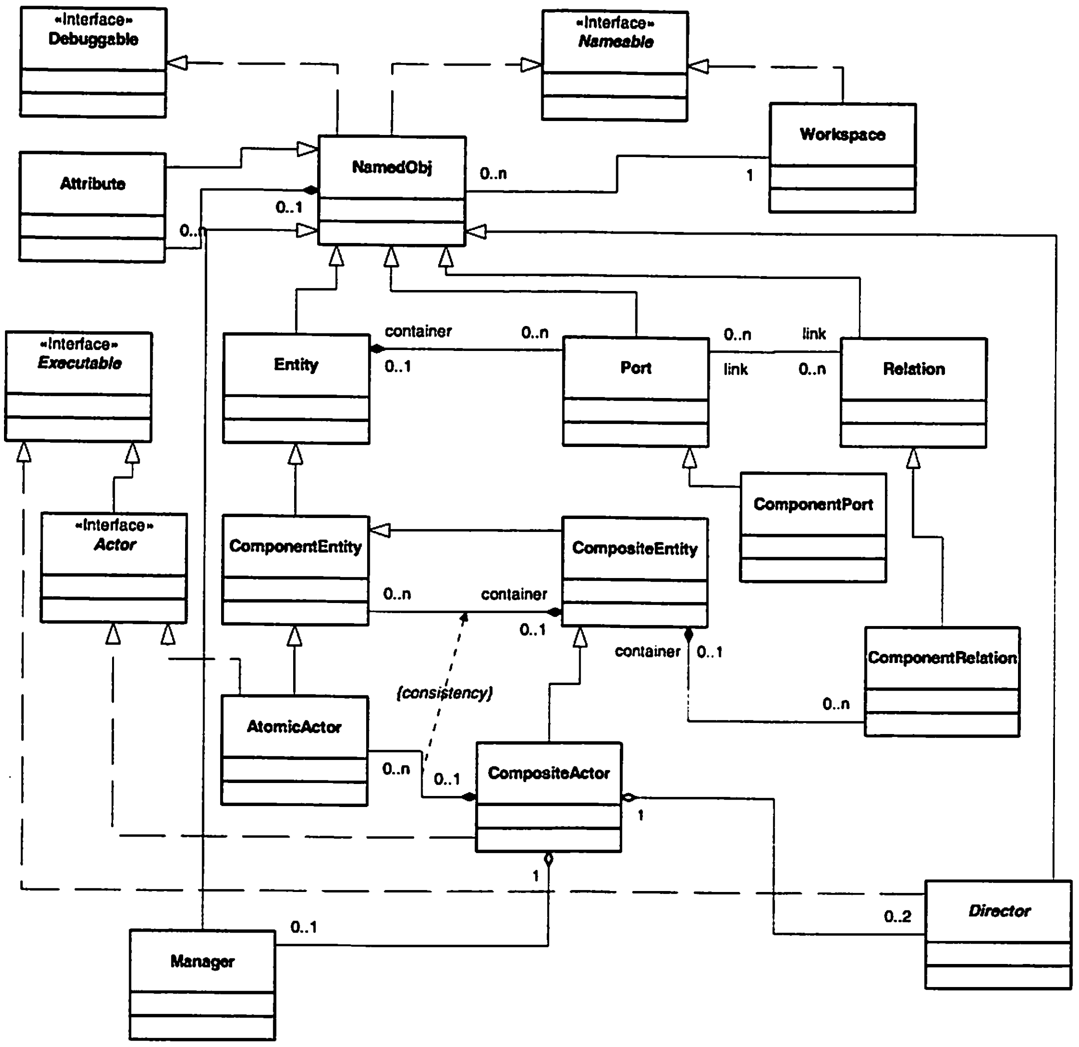


FIGURE 1.8. Some of the key classes in Ptolemy II. These are defined in the *kernel*, *kernel.util*, and *actor* packages. They define the Ptolemy II abstract syntax and abstract semantics.

primitives of the abstract syntax supported by Ptolemy II. They are fully explained in the kernel chapter. ComponentPort, ComponentRelation, and ComponentEntity extend these classes by adding support for clustered graphs. CompositeEntity extends ComponentEntity and represents an aggregation of instances of ComponentEntity and ComponentRelation.

The Executable interface, explained in the actors chapter, defines objects that can be executed. The Actor interface extends this with capability for transporting data through ports. AtomicActor and CompositeActor are concrete classes that implement this interface. The Executable and Actor interfaces are key to the Ptolemy II abstract semantics.

An executable Ptolemy II model consists of a top-level CompositeActor with an instance of Director and an instance of Manager associated with it. The manager provides overall control of the execution (starting, stopping, pausing). The director implements a semantics of a model of computation to govern the execution of actors contained by the CompositeActor.

Director is the base class for directors that implement models of computation. Each such director is associated with a domain. We have defined in Ptolemy II directors that implement continuous-time modeling (ODE solvers), process networks, synchronous dataflow, discrete-event modeling, and communicating sequential processes.

1.6.3 Domains

The domains in Ptolemy II are subpackages of the ptolemy.domains package. The more mature and frequently used domains are shown in figure 1.9. The experimental domains and less commonly used domains are not shown, but the examples in figure 1.9 are illustrative of their structure. These packages generally contain a kernel subpackage, which defines classes that extend those in the actor or kernel packages of Ptolemy II. The lib subpackage, when it exists, includes domain-specific actors.

1.6.4 Library Packages

Most domains extend classes in the actor package to give a specific semantic interpretation to an interconnection of actors. It is possible, and strongly encouraged, to define actors in such a way that they can operate in multiple domains. Such actors are said to be *domain polymorphic*. Actor that are domain polymorphic are organized in the packages shown in figure 1.10. These packages are briefly described below:

actor.lib	This package is the main library of polymorphic actors.
actor.lib.comm	This is a library of actors for modeling communications systems.
actor.lib.conversions	This package provides domain polymorphic actors that convert data between different types.
actor.lib.gui	This package is a library of polymorphic actors with user interface components, such as plotters.
actor.lib.hoc	This package is a library of higher-order components, which are components that construct portions of a model.
actor.lib.image	This package is a library of image processing actors.

actor.lib.io This package provides file I/O.

actor.lib.io.comm

This package provides an actor that communicate via the serial ports. This actor works only under Windows.

actor.lib.jai This is a library of image processing actors based on the *Java advanced imaging* API.

actor.lib.javasound

This package provides sound actors.

actor.lib.jmf This is a library of image processing actors based on the *Java media framework*

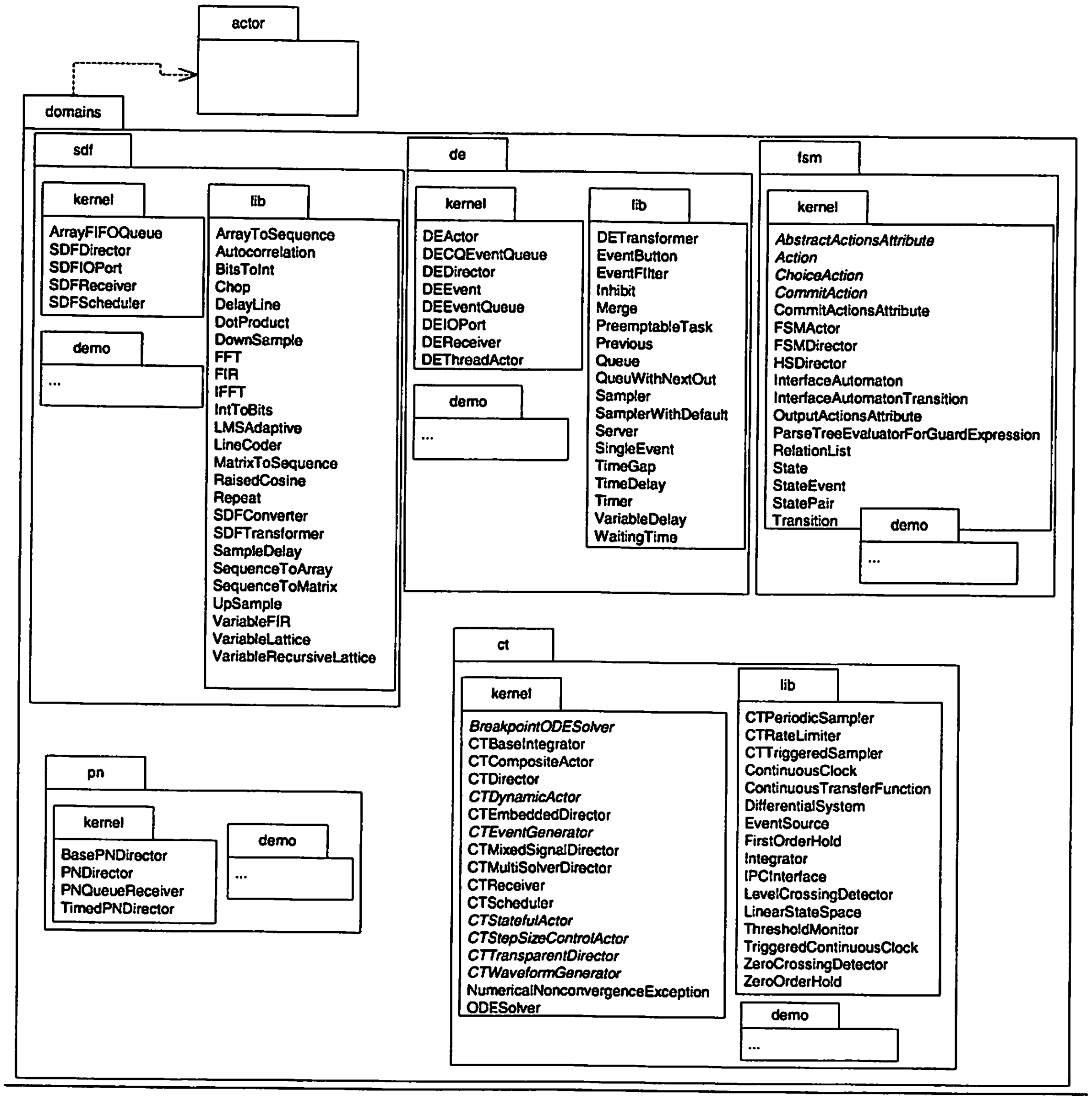


FIGURE 1.9. Package structure of common Ptolemy II domains.

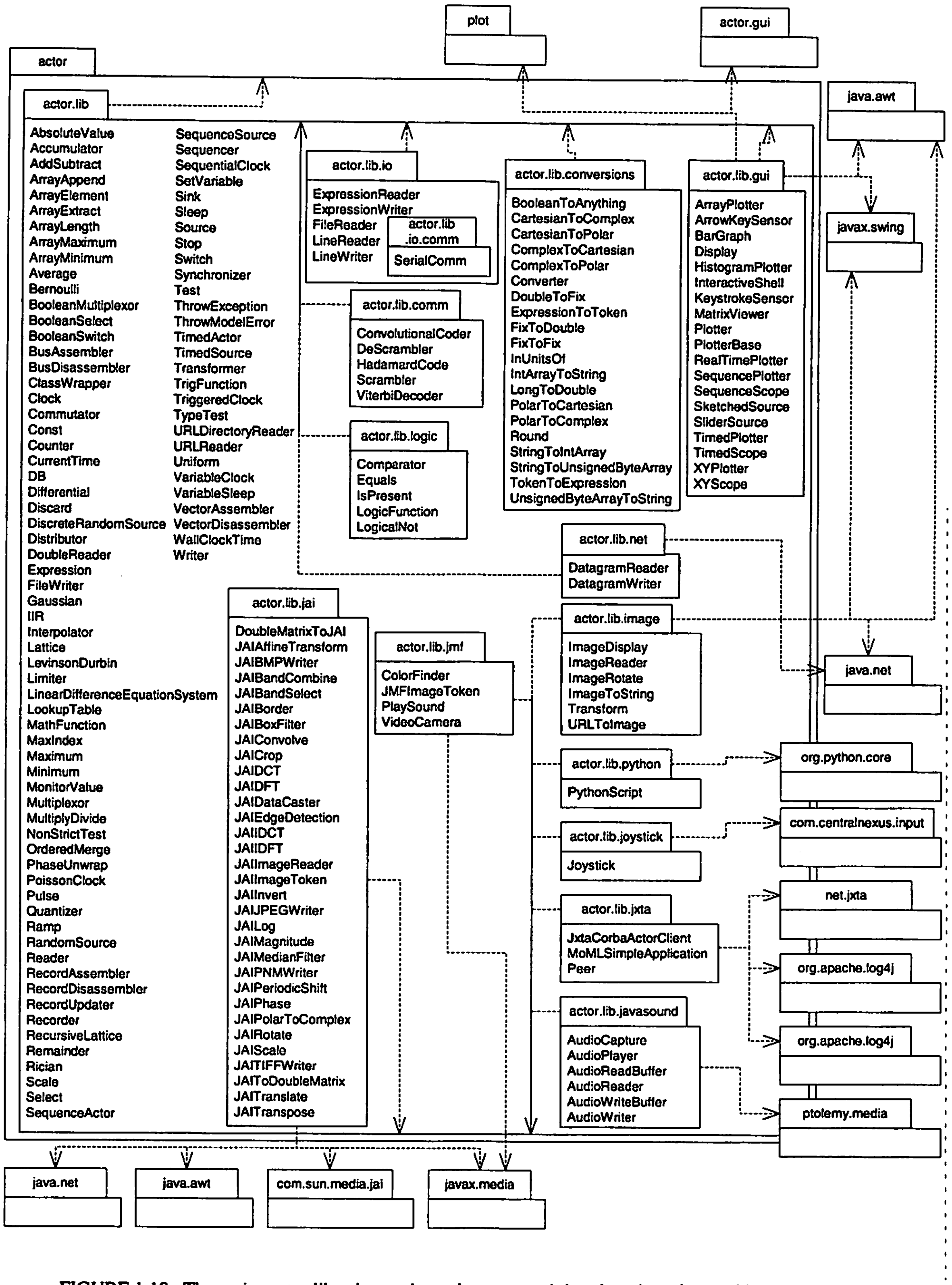


FIGURE 1.10. The major actor libraries are in packages containing domain-polymorphic actors.

	API.
actor.lib.joystick	This package provides an example actor that communicates with a particular I/O device, a joystick.
actor.lib.jxta	This is a library of experimental actors supporting the JXTA discovery mechanism from Sun Microsystems.
actor.lib.logic	This package provides actors that perform logical functions like AND, OR and NOT.
actor.lib.net	This package provides actors that communicate using datagrams.
actor.lib.python	This package provides an actor whose operation can be specified in Python.

1.6.5 User Interface Packages

The UI packages provide support for our XML file format, called MoML, and a visual interface for constructing models graphically, called Vergil. These packages are organized as shown in figures 1.11 and 1.12. The intent of each package is described below:

actor.gui	This package contains the configuration infrastructure, which supports modular construction of user interfaces that are themselves Ptolemy II models.
actor.gui.style	This package contains classes that decorate attributes to serve as hints to a user interface about how to present these attributes to the user.
gui	This package contains generically useful user interface components.
media	This package encapsulates a set of classes supporting audio and image processing.
moml	This package contains classes support our XML modeling markup language (MoML), which is used to describe Ptolemy II models.
moml.filter	This package provides backward compatibility between Ptolemy release. We hope to replace it with an XSL based solution in a future release.
plot	This package and its packages provides two-dimensional signal plotting widgets.
vergil	This package and its packages contains the Ptolemy II graphical user interface. It builds on Diva, a toolkit that extends Java 2D. For more information about Diva, see http://www.gigascale.org/diva

1.6.6 Capabilities

Ptolemy II is a third generation system. Its immediate predecessor, Ptolemy Classic, still has active users and developers, particularly through a commercial product that is based partly on it, Agilent's ADS. Ptolemy II has a somewhat different emphasis, and through its use of Java, concurrency, and integration with the network, is aggressively experimental. Some of the major capabilities in Ptolemy II that we believe to be new technology in modeling and design environments include:

- *Higher level concurrent design in Java^{TM}.* Java support for concurrent design is very low level, based on threads and monitors. Maintaining safety and liveness can be quite difficult [60]. Ptolemy II includes a number of domains that support design of concurrent systems at a much higher level

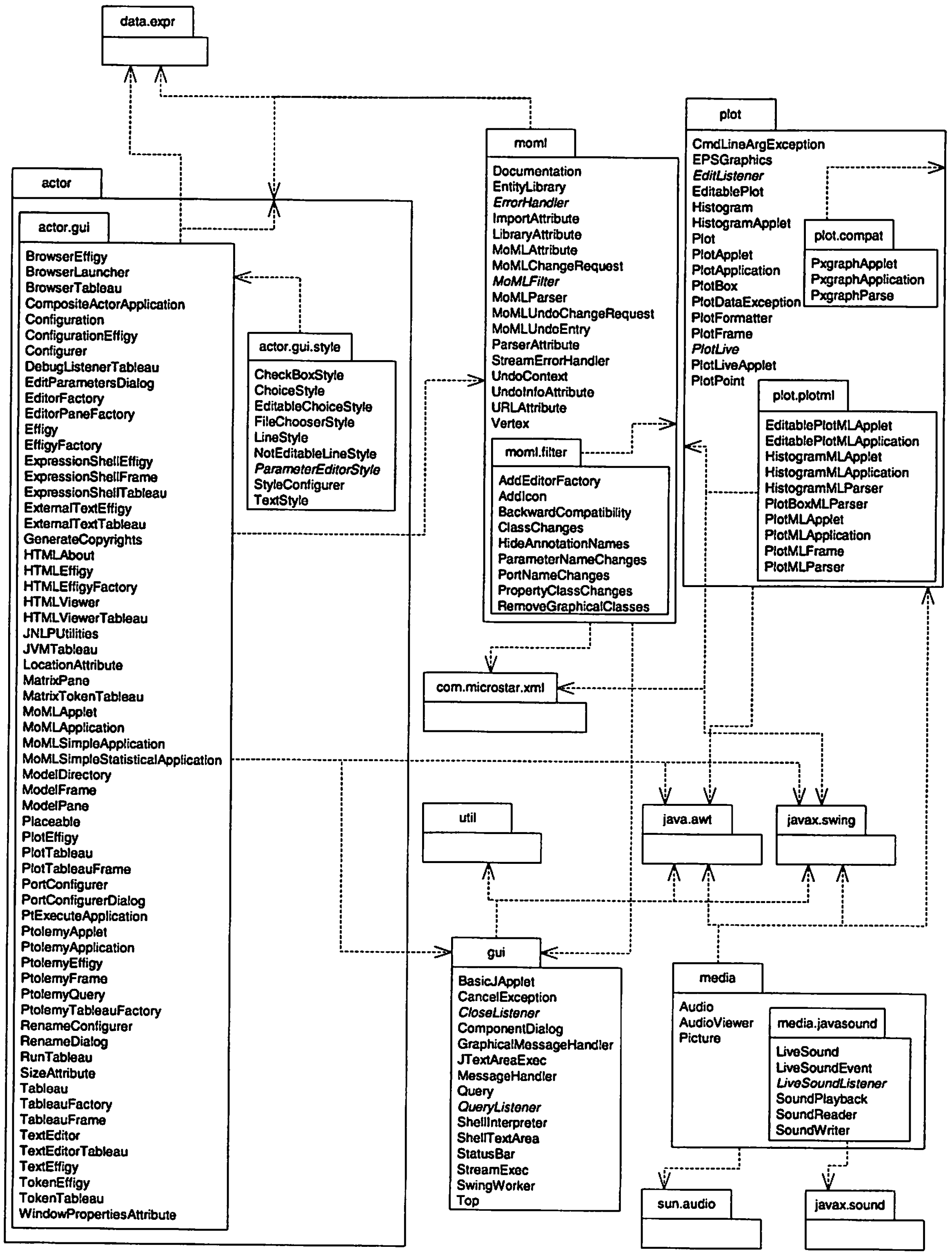


FIGURE 1.11. Packages in Ptolemy II that support the user interfaces, including the MoML XML schema, plotters and other display infrastructure, and support for windows and application configurations.

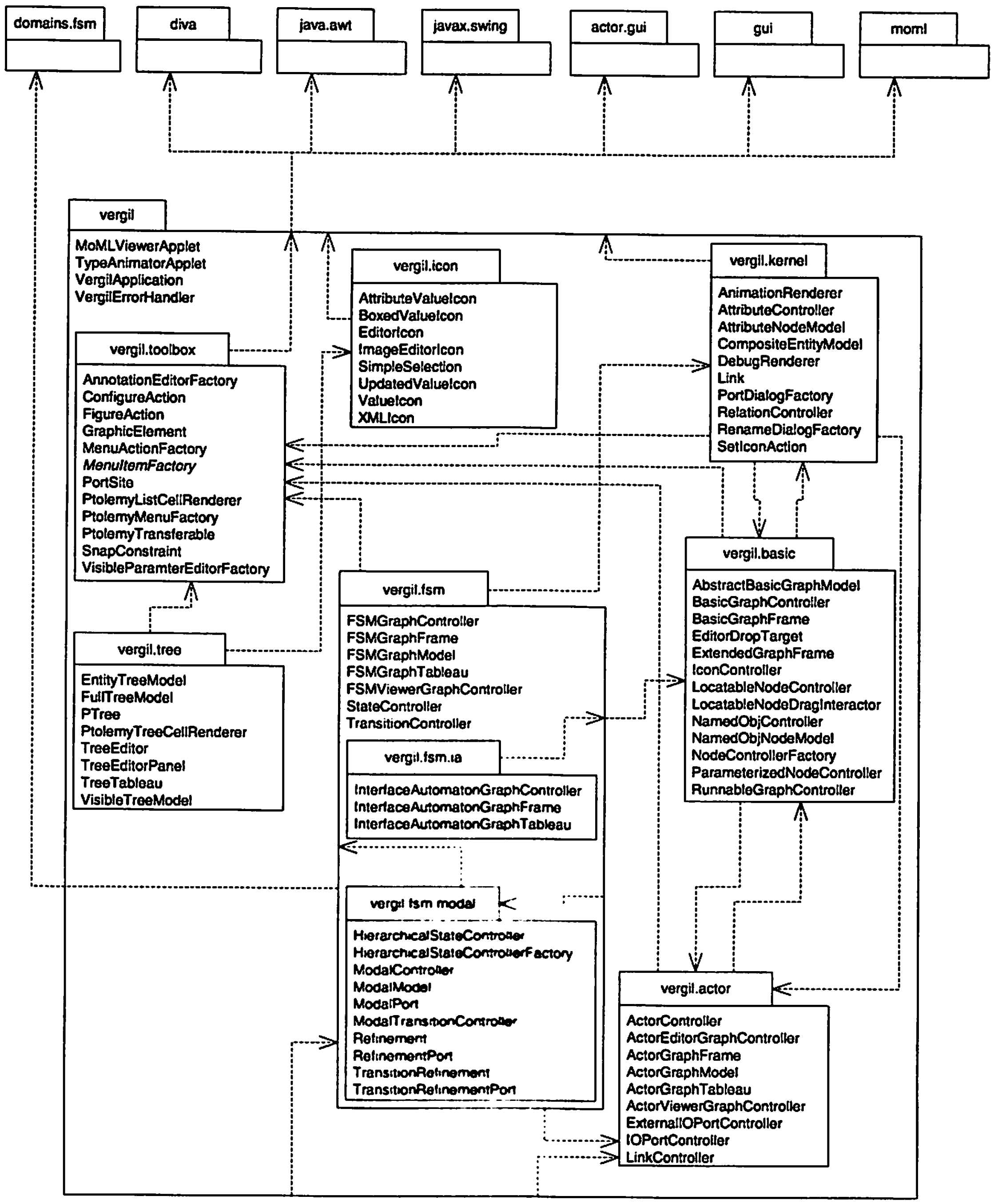


FIGURE 1.12. Packages in Ptolemy II provide the Vergil visual editor.

of abstraction, at the level of their software architecture. Some of these domains use Java threads as an underlying mechanism, while others offer an alternative to Java threads that is much more efficient, scalable, and understandable.

- *Better modularization through the use of packages.* Ptolemy II is divided into packages that can be used independently and distributed on the net, or drawn on demand from a server. This breaks with tradition in design software, where tools are usually embedded in huge integrated systems with interdependent parts.
- *Complete separation of the abstract syntax from the semantics.* Ptolemy designs are structured as clustered graphs. Ptolemy II defines a clean and thorough abstract syntax for such clustered graphs, and separates into distinct packages the infrastructure supporting such graphs from mechanisms that attach semantics (such as dataflow, analog circuits, finite-state machines, etc.) to the graphs.
- *Improved heterogeneity via a well-defined abstract semantics.* Ptolemy Classic provided a wormhole mechanism for hierarchically coupling heterogeneous models of computation. This mechanism is improved in Ptolemy II through the use of opaque composite actors, which provide better support for models of computation that are very different from dataflow, the best supported model in Ptolemy Classic. These include hierarchical concurrent finite-state machines and continuous-time modeling techniques.
- *Thread-safe concurrent execution.* Ptolemy models are typically concurrent, but in the past, support for concurrent execution of a Ptolemy model has been primitive. Ptolemy II supports concurrency throughout, allowing for instance for a model to mutate (modify its clustered graph structure) while the user interface simultaneously modifies the structure in different ways. Consistency is maintained through the use of monitors and read/write semaphores [51] built upon the lower level synchronization primitives of Java.
- *A software architecture based on object modeling.* Since Ptolemy Classic was constructed, software engineering has seen the emergence of sophisticated object modeling [89][110][113] and design pattern [35] concepts. We have applied these concepts to the design of Ptolemy II, and they have resulted in a more consistent, cleaner, and more robust design. We have also applied a simplified software engineering process that includes systematic design and code reviews [107].
- *A truly polymorphic type system.* Ptolemy Classic supported rudimentary polymorphism through the "anytype" particle. Even with such limited polymorphism, type resolution proved challenging, and the implementation is ad-hoc and fragile. Ptolemy II has a more modern type system based on a partial order of types and monotonic type refinement functions associated with functional blocks. Type resolution consists of finding a fixed point, using algorithms inspired by the type system in ML [92]. The type system is described in [124] and [125].
- *Domain-polymorphic actors.* In Ptolemy Classic, actor libraries were separated by domain. Through the notion of subdomains, actors could operate in more than one domain. In Ptolemy II, this idea is taken much further. Actors with intrinsically polymorphic functionality can be written to operate in a much larger set of domains. The mechanism they use to communicate with other actors depends on the domain in which they are used. This is managed through a concept that we call a *process level type system.*
- *Extensible XML-based file formats.* XML is an emerging standard for representation of information that focuses on the logical relationships between pieces of information. Human-readable representations are generated with the help of style sheets. Ptolemy II will use XML as its primary format for persistent design data.

- *Distributed models.* Ptolemy II has (still preliminary) infrastructure supporting distributed modeling using CORBA, Java RMI, or lower-level networking primitives. Ptolemy II has (still preliminary) support for migrating software components.
- *Component specialization.* Ptolemy II has an evolving code generation mechanism that is very different from that in Ptolemy Classic. In Ptolemy Classic, each component has to have a definition in the target language, and the code generator merely stitches together these components. In Ptolemy II, components are defined in Java, and the Java definition is parsed. An API for performing optimization transformations on the abstract syntax tree is defined, and then compiler back ends can be used to generate target code. A preliminary implementation of this approach is described in [98], [119] and [120].
- *Fully integrated expression language.* The Ptolemy II expression language is a higher-order, richly expressive language that is fully integrated with actor-oriented modeling. The type system inference mechanism propagates through expressions, parameters, and actor ports seamlessly.

1.6.7 Future Capabilities

Capabilities that we anticipate making available in the future include:

- *Integrated verification tools.* Modern verification tools based on model checking [46] could be integrated with Ptolemy II at least to the extent that finite state machine models can be checked. We believe that the separation of control logic from concurrency will greatly facilitate verification, since only much smaller cross-sections of the system behavior will be offered to the verification tools.
- *Reflection of dynamics.* Java supports reflection of static structure, but not of dynamic properties of process-based objects. For example, the data layout required to communicate with an object is available through the reflection package, but the communication protocol is not. We plan to extend the notion of reflection to reflect such dynamic properties of objects.
- *Meta modeling.* The domains in Ptolemy II are constructed based on an intuitive understanding of a useful class of modeling techniques, and then the support infrastructure for specifying and executing models in the domain are built by hand by writing Java code. Others have built tools that have the potential of improving on this situation by *meta modeling*. In Dome (from Honeywell) and GME (from Vanderbilt), for example, a modeling strategy itself is modeled, and user interfaces supporting that modeling strategy are synthesized from that model. We can view the current component-based architecture of Vergil as a starting point in this direction. In the future, we expect to see much more use of Ptolemy II itself to define and construct Ptolemy II domains and their user interfaces.

1.7 Acknowledgements

There have been many participants in the Ptolemy Project over the years. So many, that we may (inadvertently) omit some names here. With apologies to those people, we list the contributors. The principal authors of version 3.0 are

- Christopher Hylands,
- Edward Lee,
- Jie Liu,
- Xiaojun Liu,

- Stephen Neuendorffer,
- Yuhong Xiong,
- Yang Zhao, and
- Haiyang Zheng.

Major contributors to earlier versions of Ptolemy II include:

- John Reekie,
- Chamberlain Fong,
- John Davis, II,
- Mudit Goel,
- Bilung Lee,
- Lukito Muliadi,
- Neil Smyth, and
- Jeff Tsay.

Other contributors include Vincent Arnould, Frederic Boulanger, Adam Cataldo, Chris Chang, Albert Chen, Elaine Cheong, Brieuc Desoutter, Pedro Domecq, Johan Eker, Geroncio Galicia, Ben Horowitz, Heloise Hse, Jörn Janneck, Zoltan Kemenczy, Bart Kienhuis, Christoph Meyer Kirsch, Sanjeev Kohli, Robert Kroeger, David Lee, Michael Leung, John Li, Andrew Mihal, Eleftherios Matsikoudis, Aleksandar Necakov, Sarah Packman, Shankar Rao, Sonia Sachs, Michael Shilman, Sean Simmons, Mandeep Singh, Peter N. Steinmetz, Dick Stevens, Mary Stewart, Ned Stoffel, Manda Sutijono, Neil Turner, Guillaume Vibert, Kees Vissers, Brian K. Vogel, Yuke Wang, Xavier Warzee, Scott Weber, Paul Whitaker, Winthrop Williams, Ed Willink, Michael Wirthlin, William Wu, Paul Yang, James Yeh, Nick Zamora, and Rachel Zhou.

The Ptolemy Project has received financial support the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation, Chess (the Center for Hybrid and Embedded Software Systems), the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of California MICRO program, and the following companies: Agilent, Atmel, Cadence, Hewlett-Packard, Hitachi, National Semiconductor, and Philips.

Appendix A: UML — Unified Modeling Language

UML (the unified modeling language) [33][106] defines a suite of visual syntaxes for describing various aspects of software architecture. We make heavy use of two of these visual syntaxes, package diagrams and static structure diagrams. These syntaxes are summarized here. As with most descriptive syntaxes, any use of the syntax involves certain stylistic choices. These stylistic choices are not part of UML, but nonetheless can be important to understanding the diagrams. We explain the style that we use here.

A.1 Package Diagrams

Figures 1.7 and 1.9 show UML *package diagrams,* which have a simple syntax. A package is given as a box with a tab, with the tab containing the name of the package. Subpackages are enclosed in the box of the parent package, and package dependencies are indicated with arrows. A package dependency occurs when a Java file in a package includes a class in another package (using import in Java).

A.2 Static Structure Diagrams

Figure 1.8 is a different kind of UML diagram, called a *static structure diagram* or *class diagram.* It represents the relationships between classes, including inheritance relationships, containment relationships, and cross references. These relationships are called an *object model,* and represent many essential features about the design.

A.2.1 Classes

A simplified static structure diagram for some Ptolemy II classes is shown in figure 1.13. In this diagram, each class is shown in a box. The class name is at the top of each box, its *attributes* are below that, and its methods below that. Thus, each box is divided into three segments separated by horizontal lines. The attributes are members of the Java classes, which may be public, package friendly, protected, or private. Private members are prefixed by a minus sign "-", as for example the _container attribute of Port. Although private members are not visible directly to users of the class, they may nonetheless be a useful part of the object model because they indicate the state information contained by an instance of the class. Public members have a leading "+" and protected methods a leading "#" in a UML diagram. There are no public or protected members shown in figure 1.13. The type of a member is indicated after a colon, so for example, the _container method of Port is of type Entity.

Methods, which are shown below attributes, also have a leading "+" for public, "#" for protected, and "-" for private. Our object models do not show private methods, since they are not inherited and are not visible in the interface to the object. Figure 1.13 shows a number of public methods and one protected method, _link() in Port. The return value of a method is given after a colon, so for example, getContainer() of Port returns an Entity.

Although not usually included in UML diagrams, our diagrams show class constructors. They are listed first among the methods and have names that are the same as the name of the class. No return type is shown. For completeness, our object models typically show all public and protected methods of these classes, although a proper object model might only show those relevant to the issues being discussed. Figure 1.13 does not show all methods, so that we can simplify the discussion of UML. Our diagrams do not include deprecated methods or methods that are present in parent classes.

Arguments to a method or constructor are shown in parentheses, with the types after a colon, so for example, ComponentEntity shows a single constructor that takes two arguments, one of type CompositeEntity and the other of type String.

A.2.2 Inheritance

Subclasses are indicated by lines with white triangles (or outlined arrow heads). The class on the side of the arrow head is the *superclass* or *base class*. The class on the other end is the *subclass* or *derived class*. The derived class is said to *specialize* the base class, or conversely, the base class to *generalize* the derived class. The derived class *inherits* all the methods shown in the base class and may *override* or some of them. In our object models, we do not explicitly show methods that override those defined in a base class or are inherited from a base class. For example, in figure 1.13, ComponentEntity has all the methods of Entity and NamedObj, and may override some of those methods, but only

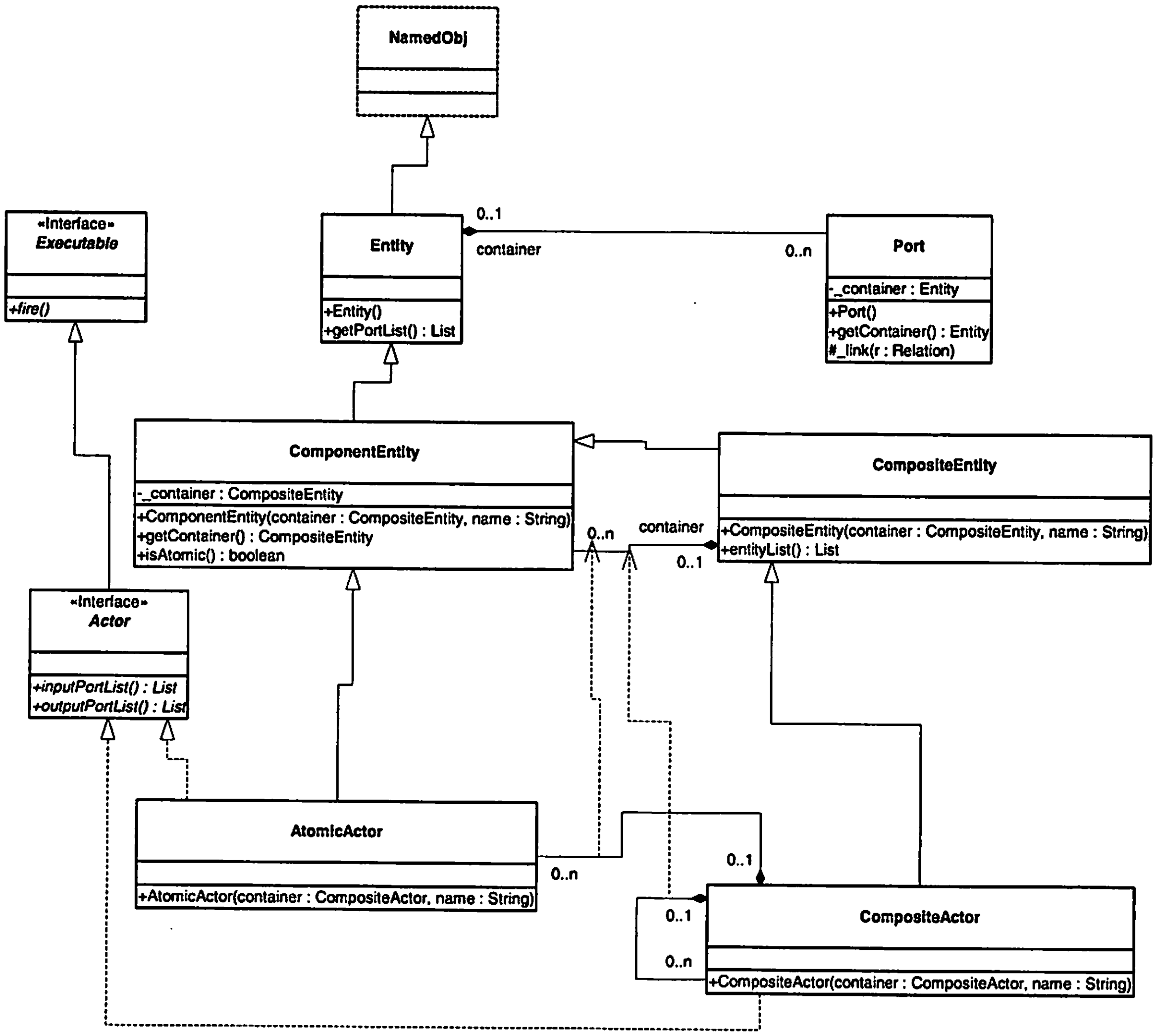


FIGURE 1.13. Simplified static structure diagram for some Ptolemy II classes. This diagram illustrates features of UML syntax that we use.

shows the one method it adds. Thus, the complete set of methods of a class is cumulative. Every class has its own methods plus those of all its superclasses.

An exception to this is constructors. In Java, constructors are not inherited. Thus, in our class diagrams, the only constructors available for a class are those shown in the box defining the class. Figure 1.13 does not show all the constructors of these classes, for simplicity.

Classes shown in boxes outlined with dashed lines, such as NamedObj in figure 1.13, are fully described elsewhere. This is not standard UML notation, but it gives us a convenient way to partition diagrams. Often, these classes belong to another package.

A.2.3 Interfaces

Figure 1.13 also shows two examples of *interfaces*, Executable and Actor. An interface is indicated by the label "<<Interface>>" and by italics in the name. An interface defines a set of methods without providing an implementation for them. It cannot be instantiated, and therefore has no constructors. When a class *implements* an interface, the object model shows the relationship with a dotted-line with an arrow. Any *concrete class* (one that can be instantiated) that implements an interface must provide implementations of all its methods. In our object models, we do not show those methods explicitly in the concrete class, just like inherited methods, but their presence is implicit in the relationship to the interface.

One interface can extend another. For example, in figure 1.13, Actor extends Executable. This means that any concrete class that implements Actor must implement the methods of Actor and Executable.

We will occasionally show *abstract classes*, which are like interfaces in that they cannot be instantiated, but unlike interfaces in that they may provide default implementations for some methods and may even have private members. Abstract classes are indicated by italics in the class name. There are no abstract classes in figure 1.13.

A.2.4 Associations

Inheritance and implementation are types of *associations* between entities in the object model. Associations of other types are indicated by other lines, often annotated with ranges like "0..n" or with diamonds on one end or the other.

Aggregations are shown as associations with diamonds. For example, an Entity is an aggregation of any number (0..n) instances of Port. More strongly, we say that a Port is *contained* by 0 or 1 instances of Entity. By containment, we mean that a port can only be contained by a single Entity. In a weaker form of aggregation, more than one aggregate may refer to the same component. The stronger form of aggregation (containment) is indicated by the filled diamond, while the weaker form is indicated by the unfilled diamond. There are no unfilled diamonds in figure 1.13. In fact, they are fairly rare in Ptolemy II, since many of its architectural features depend on containment relationships, where an object can have at most one container.

The relationship between ComponentEntity and CompositeEntity is particularly interesting. An instance of CompositeEntity can contain any number of instances of ComponentEntity, but CompositeEntity is derived from ComponentEntity. Thus, a CompositeEntity can contain any number of instances of either ComponentEntity or CompositeEntity. This is the classic Composite design pattern [35], which supports arbitrarily deeply nested containment hierarchies.

In figure 1.13, a CompositeActor is an aggregation of AtomicActors and CompositeActors. These two aggregation relations are derived from the aggregation relationship between ComponentEntity and CompositeEntity. This derived association is indicated with a dashed line with an open arrowhead.

Appendix B: Ptolemy II Naming Conventions

We have made an effort to be consistent about naming of classes, methods and members. This appendix describes our policy.

B.1 Classes

Class names are capitalized with internal word boundaries also capitalized (as in "CompositeEntity"). Most names are made up of complete words ("CompositeEntity" rather than "CompEnt")[1]. Interface names suggest their potential (as in "Executable," which means "can be executed").

Despite having packages to divide up the namespace, we attempt nonetheless to keep class names unique. This helps avoid confusion and bugs that may arise from having Java import statements in the wrong order. In many cases, a domain includes a specialized version of some more generic class. In this case, we create a unique name by prefixing the generic name with the domain name. For example, while Director is a base class in the actor package, DEDirector is a derived class in the DE domain.

For the most part, we try to avoid prefixing actor names with the domain name. e.g., we define Delay rather than DEDelay. Occasionally however, the domain prefix is useful to distinguish two versions of some similar functionality, both of which might be useful in a domain. For example, the DE domain can use actors derived from Transformer or from DETransformer, where the latter is specialized to DE.

B.2 Members

Member names are not capitalized, although internal word boundaries usually are (e.g. "declaredType"). If the member is private or protected, then its name begins with a leading underscore (e.g. "_declaredType").

B.3 Methods

Method names are similar to member names, in that they are not capitalized, except on internal word boundaries. Private and protected methods have a leading underscore. In text referring to methods, the method name is followed by open and close parentheses, as in "getName()." Usually, no arguments are given, even if the method takes arguments.

Method names that are plural, such as insideRelations(), usually return an enumeration (or sometimes an array, or an iterator). Methods that return Lists are usually of the form portList().

1. There are some (perhaps regrettable) exceptions to this, such as NamedObj.

2

Using Vergil

Authors: *Edward A. Lee*
Steve Neuendorffer

2.1 Introduction

There are many ways to use Ptolemy II. It can be used as a framework for assembling software components, as a modeling and simulation tool, as a block-diagram editor, as a system-level rapid prototyping application, as a toolkit supporting research in component-based design, or as a toolkit for building Java applications. This chapter introduces its use as a modeling and simulation tool.

In this chapter, we describe how to graphically construct models using Vergil, a graphical user interface (GUI) for Ptolemy II. Figure 2.1 shows a simple Ptolemy II model in Vergil, showing the graph editor, one of several editors available in Vergil. Keep in mind as you read this document that graphical entry of models is only one of several possible entry mechanisms available in Ptolemy II. Moreover, only some of the execution engines (called *domains*) are described here. A major emphasis of Ptolemy II is to provide a framework for the construction of modeling and design tools, so the specific modeling and design tools described here should be viewed as representative of our efforts.

2.2 Quick Start

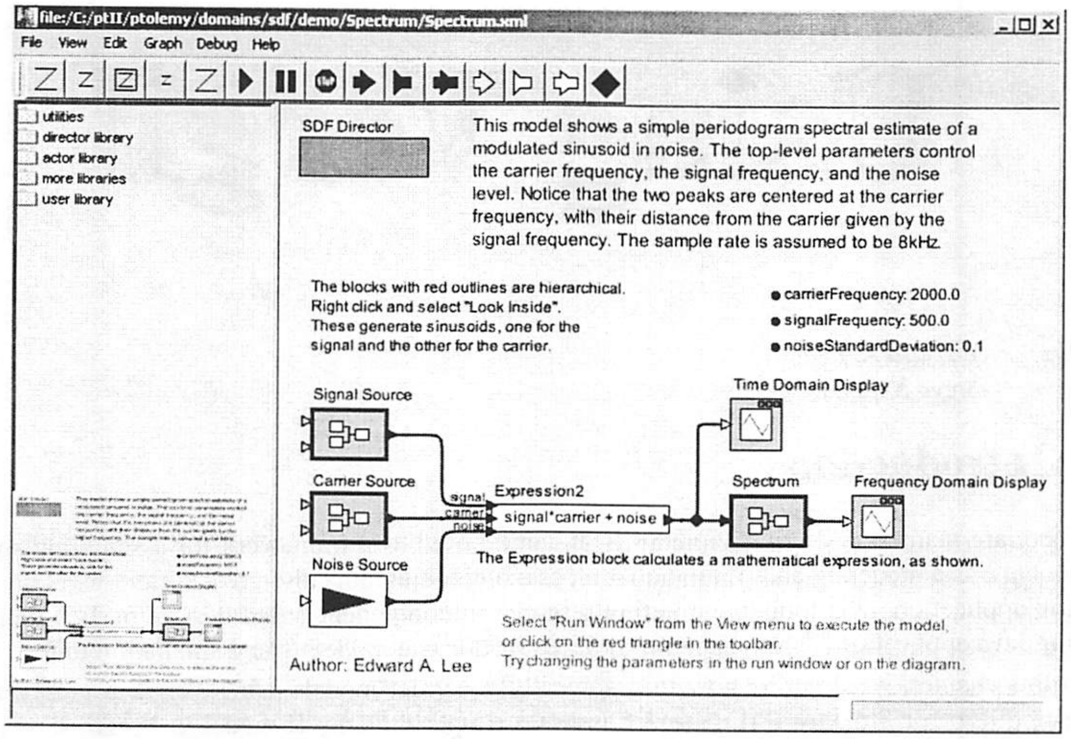
This section shows how to start Vergil, how to execute and explore pre-built models, and how to construct your own models.

2.2.1 Starting Vergil

First start Vergil. From the command line, enter "vergil", or select Vergil or Ptolemy II in the Start menu, or click on a Web Start link on a web page supporting the web edition. You should see an initial welcome window that looks like the one in figure 2.2. Feel free to explore the links in this window. The "Quick tour" link takes you to the page shown in figure 2.3.

2.2.2 Executing a Pre-Built Model: A Signal Processing Example

The very first example on the quick tour page is the model shown in figure 2.1, which creates a sinusoidal signal, multiplies it by a sinusoidal carrier, and adds noise. You can execute this model in either of two ways. First, you can select Run Window in the View menu, and then click on Go. The result is shown in figure 2.4. The upper plot shows the spectrum of the time-domain signal shown in the lower plot. Note the four peaks, which indicate the modulated sinusoid. In the run window you can



This model shows a simple periodogram spectral estimate of a modulated sinusoid in noise. The top-level parameters control the carrier frequency, the signal frequency, and the noise level. Notice that the two peaks are centered at the carrier frequency, with their distance from the carrier given by the signal frequency. The sample rate is assumed to be 8kHz.

The blocks with red outlines are hierarchical. Right click and select "Look Inside". These generate sinusoids, one for the signal and the other for the carrier.

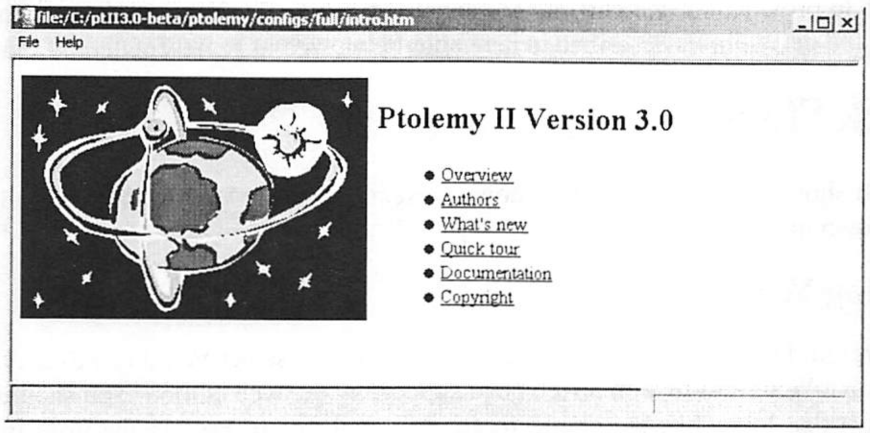
- carrierFrequency: 2000.0
- signalFrequency: 500.0
- noiseStandardDeviation: 0.1

The Expression block calculates a mathematical expression, as shown.

Select "Run Window" from the View menu to execute the model, or click on the red triangle in the toolbar. Try changing the parameters in the run window or on the diagram.

Author: Edward A. Lee

FIGURE 2.1. Example of a Vergil window.



Ptolemy II Version 3.0

- Overview
- Authors
- What's new
- Quick tour
- Documentation
- Copyright

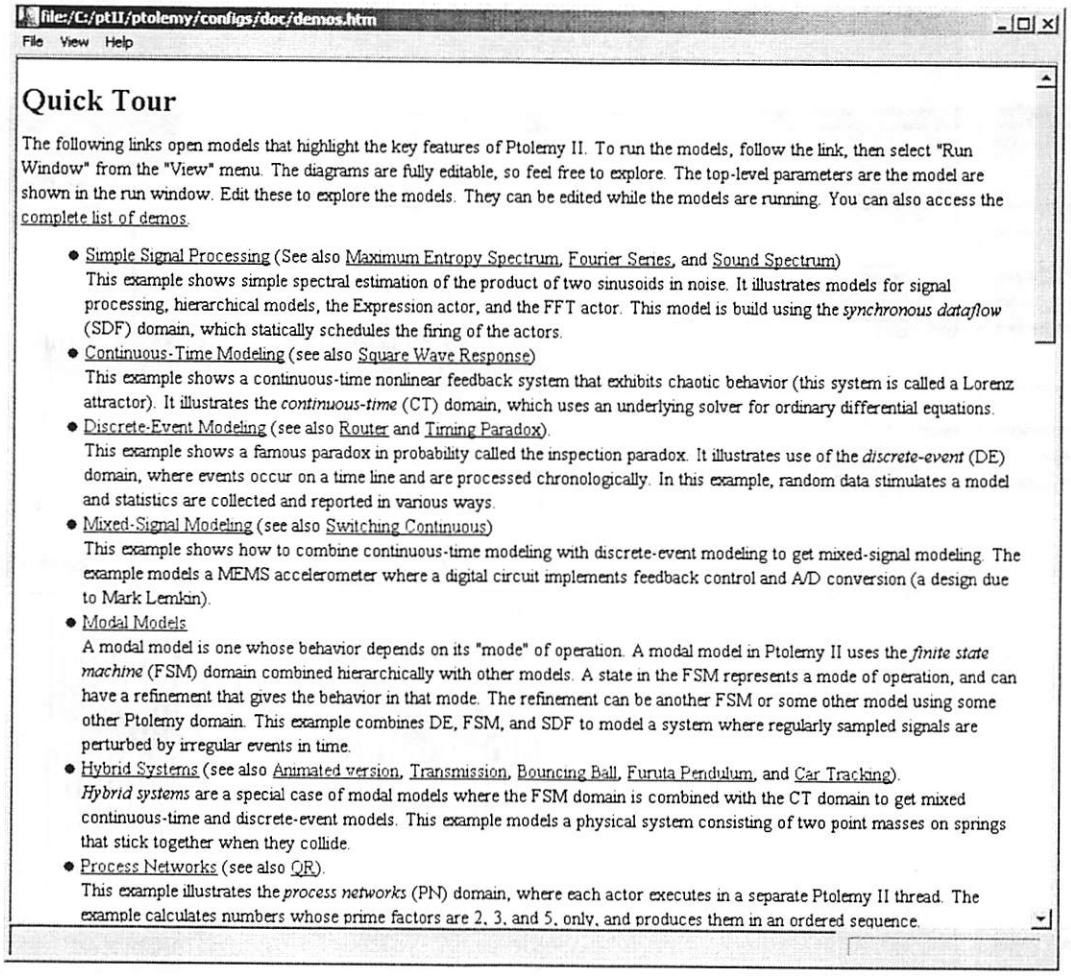
FIGURE 2.2. Initial welcome window.

adjust the frequencies of the signal and the carrier as well as the amount of noise. These can also be adjusted in the block diagram in figure 2.1 by double clicking on the bulleted parameters near the upper right of the window.

The second alternative for running the model is to click on the run button in the toolbar, which is indicated by a red triangle pointing to the right. If you use this alternative, then the two signal plots are displayed in their own windows.

You can study the way the model is constructed in figure 2.1. Note the Expression actor in the middle, whose icon indicates the expression being calculated: "`signal*carrier + noise`". The identifiers in this expression, `signal`, `carrier`, and `noise` refer to the input ports by name. The names of these ports are shown in the diagram. The Expression actor is a very flexible actor in the Ptolemy II actor library. It can have any number of input ports, with arbitrary names, and uses a rich and expressive expression language to specify the value of the output as a function of the inputs (and parameters of the containing model, if desired).

Three of the actors in figure 2.1 are *composite actors*, which means that their implementation is itself given as a block diagram. Composite actors are indicated visually by the red outline. You can look inside to reveal the implementation, as shown in figure 2.5, which shows the implementation of



Quick Tour

The following links open models that highlight the key features of Ptolemy II. To run the models, follow the link, then select "Run Window" from the "View" menu. The diagrams are fully editable, so feel free to explore. The top-level parameters are the model are shown in the run window. Edit these to explore the models. They can be edited while the models are running. You can also access the complete list of demos.

- Simple Signal Processing (See also Maximum Entropy Spectrum, Fourier Series, and Sound Spectrum)
  This example shows simple spectral estimation of the product of two sinusoids in noise. It illustrates models for signal processing, hierarchical models, the Expression actor, and the FFT actor. This model is build using the *synchronous dataflow* (SDF) domain, which statically schedules the firing of the actors.
- Continuous-Time Modeling (see also Square Wave Response)
  This example shows a continuous-time nonlinear feedback system that exhibits chaotic behavior (this system is called a Lorenz attractor). It illustrates the *continuous-time* (CT) domain, which uses an underlying solver for ordinary differential equations.
- Discrete-Event Modeling (see also Router and Timing Paradox).
  This example shows a famous paradox in probability called the inspection paradox. It illustrates use of the *discrete-event* (DE) domain, where events occur on a time line and are processed chronologically. In this example, random data stimulates a model and statistics are collected and reported in various ways.
- Mixed-Signal Modeling (see also Switching Continuous)
  This example shows how to combine continuous-time modeling with discrete-event modeling to get mixed-signal modeling. The example models a MEMS accelerometer where a digital circuit implements feedback control and A/D conversion (a design due to Mark Lemkin).
- Modal Models
  A modal model is one whose behavior depends on its "mode" of operation. A modal model in Ptolemy II uses the *finite state machine* (FSM) domain combined hierarchically with other models. A state in the FSM represents a mode of operation, and can have a refinement that gives the behavior in that mode. The refinement can be another FSM or some other model using some other Ptolemy domain. This example combines DE, FSM, and SDF to model a system where regularly sampled signals are perturbed by irregular events in time.
- Hybrid Systems (see also Animated version, Transmission, Bouncing Ball, Furuta Pendulum, and Car Tracking).
  Hybrid systems are a special case of modal models where the FSM domain is combined with the CT domain to get mixed continuous-time and discrete-event models. This example models a physical system consisting of two point masses on springs that stick together when they collide.
- Process Networks (see also QR).
  This example illustrates the *process networks* (PN) domain, where each actor executes in a separate Ptolemy II thread. The example calculates numbers whose prime factors are 2, 3, and 5, only, and produces them in an ordered sequence.

FIGURE 2.3. The quick-tour page.

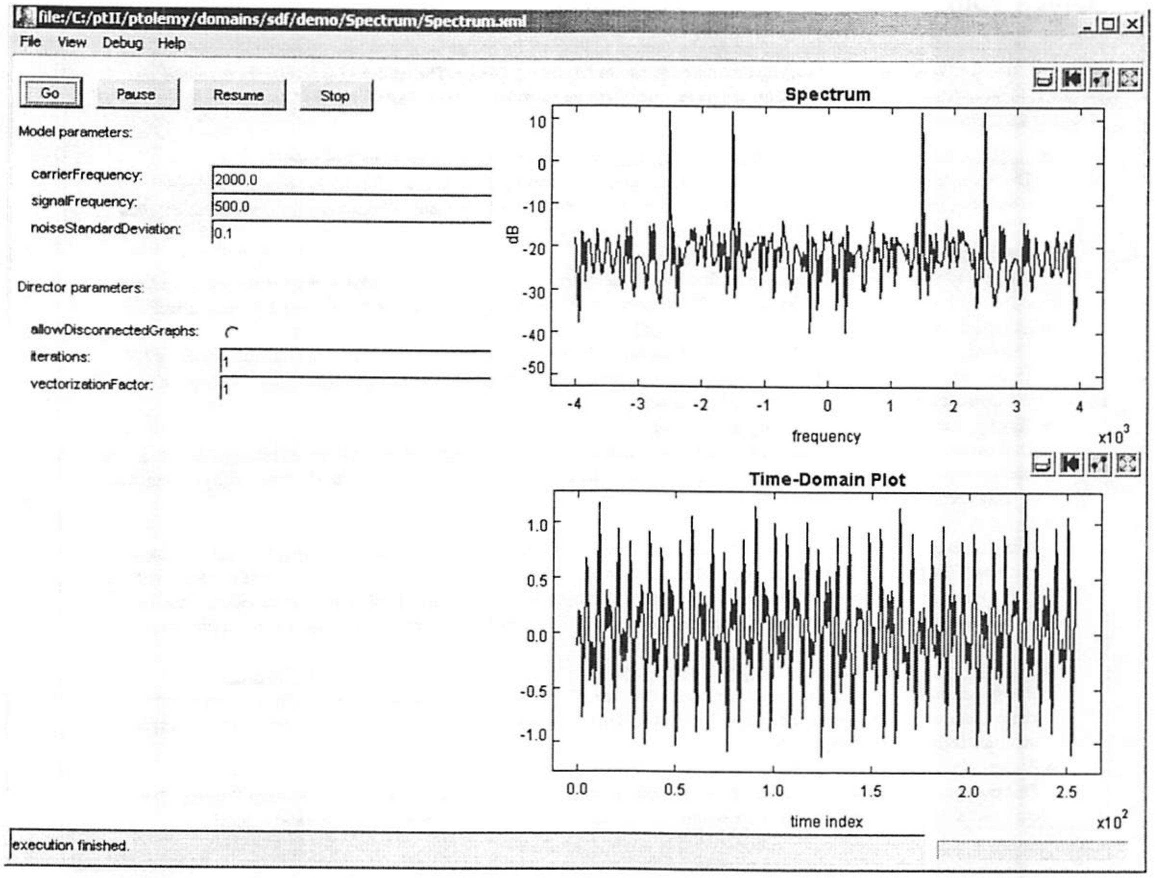
the Signal Source in figure 2.1. It is evident from the block diagram how a sinusoidal signal is generated.

2.2.3 Executing a Pre-Built Model: A Continuous-Time Example

A key principle of the Ptolemy II system is that the model of computation that defines the meaning of a block diagram is not built-in, but is rather specified by the *director* component that is included in the model. The box labeled "SDF Director" in figures 2.1 and 2.5 specifies that these block diagrams have *synchronous dataflow* semantics, which is explained further below. The second example in the quick tour of figure 2.3, by contrast, has continuous-time semantics (the one labeled "Continuous-Time Modeling"). The example is the well-known *Lorenz attractor*, a non-linear feedback system that exhibits chaotic behavior.

The Lorenz attractor model, shown in figure 2.1, is a block diagram representation of a set of non-linear ordinary differential equations. The blocks with integration signs in their icons are integrators. At any given time *t*, their output is given by

$$x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau)d\tau, \quad (1)$$


file:/C:/ptII/ptolemy/domains/sdf/demo/Spectrum/Spectrum.xml

Model parameters:

carrierFrequency: 2000.0

signalFrequency: 500.0

noiseStandardDeviation: 0.1

Director parameters:

allowDisconnectedGraphs:

iterations: 1

vectorizationFactor: 1

execution finished.

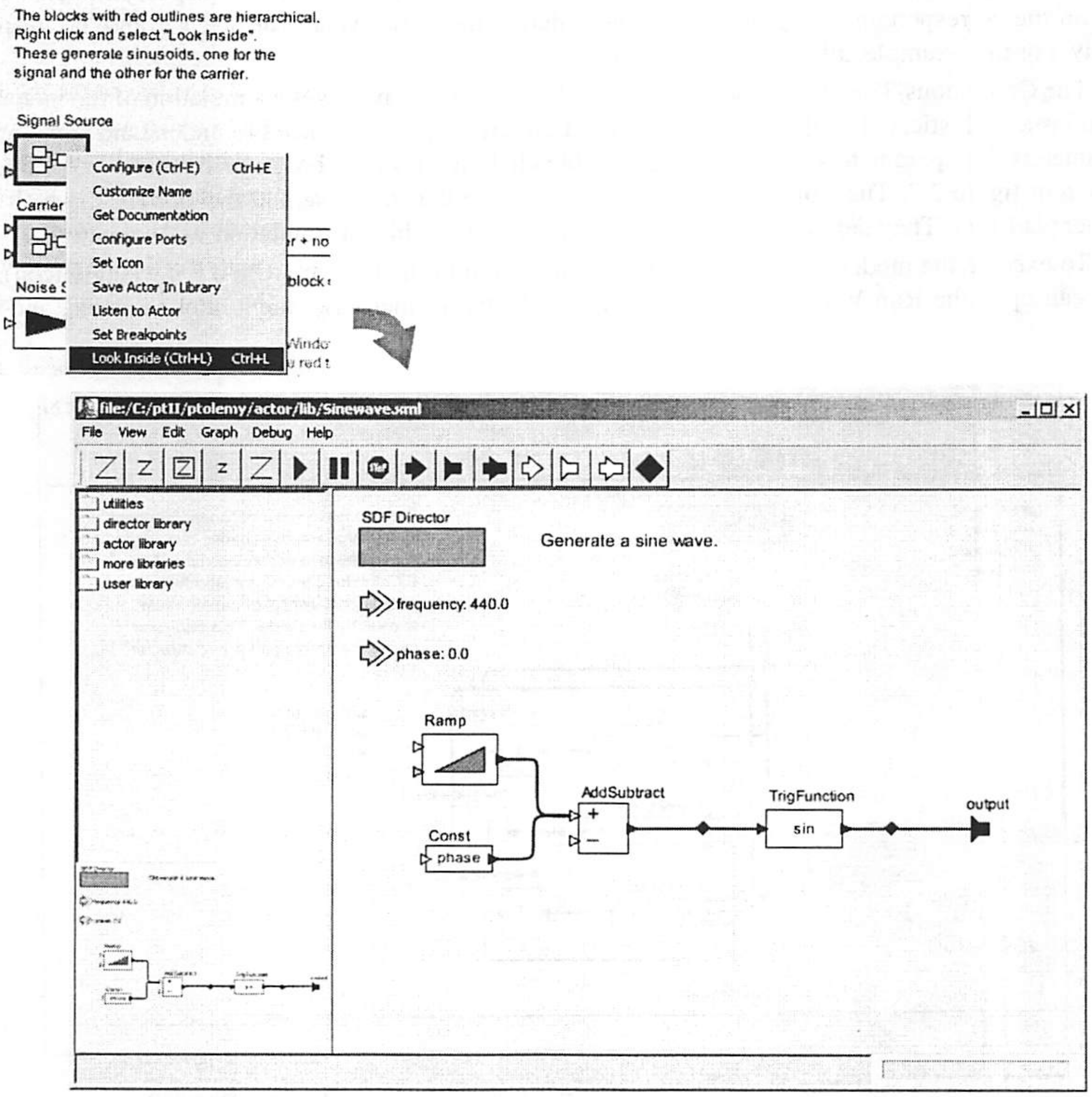
FIGURE 2.4. The run window for the model shown in figure 2.1.

where $x(t_0)$ is the initial state of the integrator, t_0 is the start time of the model, and $\dot{x}$ is the input signal. Note that since the output is the integral of the input, then at any given time, the input is the derivative of the output,

$$\dot{x}(t) = \frac{d}{dt}x(t). \tag{2}$$

Thus, the system describes either an integral equation or a differential equation, depending on which of these two forms you use.

Let the output of the top integrator in figure 2.1 be x_1, the output of the middle integrator be x_2, and the output of the bottom integrator be x_3. Then the equations described by figure 2.1 are


The blocks with red outlines are hierarchical. Right click and select "Look Inside". These generate sinusoids, one for the signal and the other for the carrier.

Generate a sine wave.

frequency: 440.0

phase: 0.0

FIGURE 2.5. Look inside composite actors to reveal their implementation.

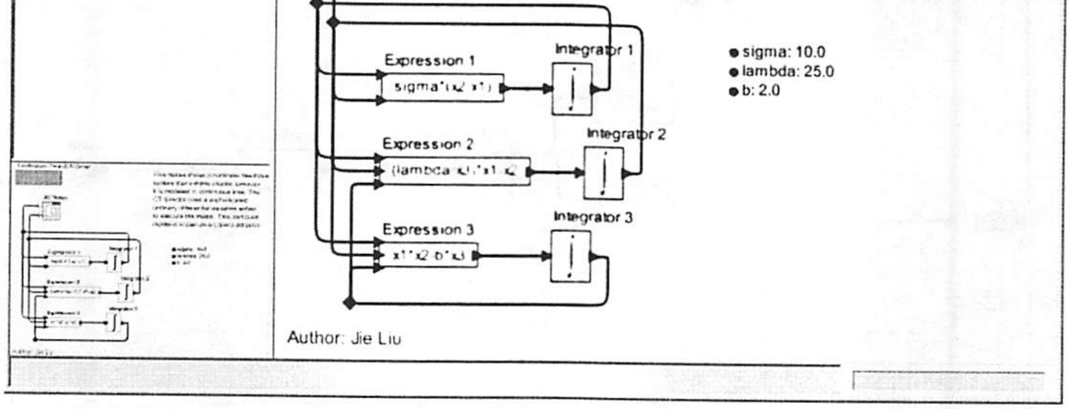
$$
\begin{aligned}
\dot{x}_1(t) &= \sigma(x_2(t) - x_1(t)) \\
\dot{x}_2(t) &= (\lambda - x_3(t))x_1(t) - x_2(t)\,. \\
\dot{x}_3(t) &= x_1(t)x_2(t) - bx_3(t)
\end{aligned}
\tag{3}
$$

For each equation, the expression on the right 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 names of the input ports are not shown in the diagram, but if you linger over them with the mouse cursor, the name will pop up in a tooltip. 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 on the right.

The integrators each also have initial values, which you can examine and change by double clicking on the corresponding integrator icon. These define the initial values of x_1, x_2, and x_3, respectively. For this example, all three are set to 1.0.

The Continuous-Time (CT) Solver, shown at the upper right, manages a simulation of the model. It contains a sophisticated ODE solver, and to use it effectively, you will need to understand some of its parameters. The parameters are accessed by double clicking on solver box, which results in the dialog shown in figure 2.7. The simplest of these parameters are the *startTime* and the *stopTime*, which are self-explanatory. They define the region of the time line over which a simulation will execute.

To execute the model, you can click on the run button in the toolbar (with a red triangle icon), or you can open the Run Window in the View menu. In the former case, the model executes, and the



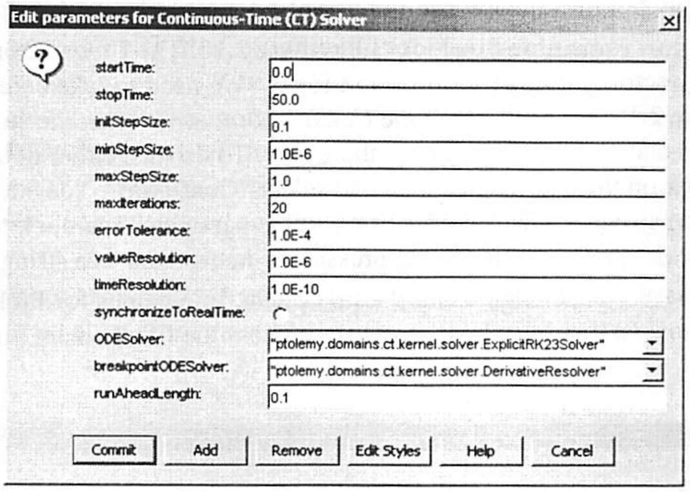
This model shows a nonlinear feedback system that exhibits chaotic behavior. It is modeled in continuous time. The CT director uses a sophisticated ordinary differential equation solver to execute the model. This particular model is known as a Lorenz attractor.

- sigma: 10.0
- lambda: 25.0
- b: 2.0

FIGURE 2.6. A block diagram representation of a set of nonlinear ordinary differential equations.

results are plotted in their own window, as shown in figure 2.8. What is plotted is $x_1(t)$ vs. $x_2(t)$ for values of t in between *startTime* and *stopTime*.

Like the Lorenz model, a typical continuous-time model contains integrators in feedback loops, or more elaborate blocks that realize linear and non-linear dynamical systems given abstract mathematical representations of them (such as Laplace transforms). In the next section, we will explore how to build a model from scratch.



startTime:	0.0
stopTime:	50.0
initStepSize:	0.1
minStepSize:	1.0E-6
maxStepSize:	1.0
maxIterations:	20
errorTolerance:	1.0E-4
valueResolution:	1.0E-6
timeResolution:	1.0E-10
synchronizeToRealTime:	
ODESolver:	"ptolemy.domains.ct.kernel.solver.ExplicitRK23Solver"
breakpointODESolver:	"ptolemy.domains.ct.kernel.solver.DerivativeResolver"
runAheadLength:	0.1

FIGURE 2.7. Dialog box showing solver parameters for the model in figure 2.1.

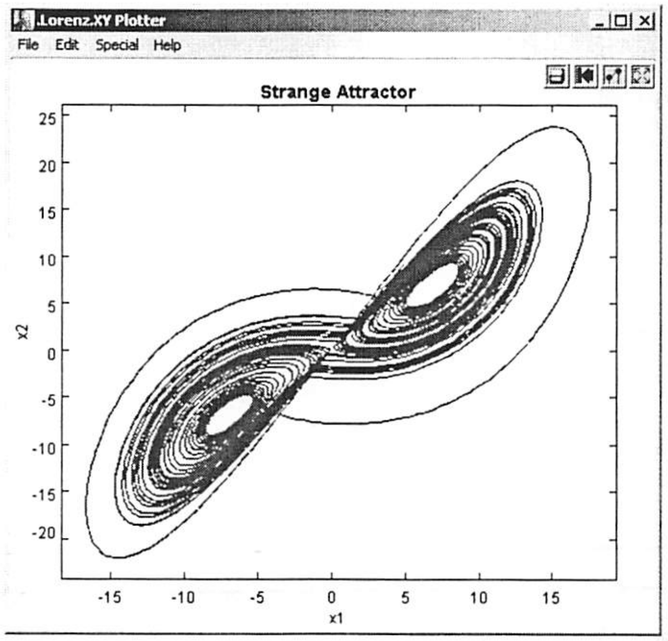


FIGURE 2.8. Result of running the Lorenz model using the run button in the toolbar.

2.2.4 Creating a New Model

Create a new model by selecting File->New->Graph Editor in the welcome window. You should see something like the window shown in figure 2.9. Ignoring the menus and toolbar for a moment, on the left is a palette of objects that can be dragged onto the page on the right. To begin with, the page on the right is blank. Open the *actor library* in the palette, and go into the *sources* library. Find the *Const* actor under *generic sources* and drag an instance over onto the blank page. Then go into the *sinks* library (*generic sinks* sublibrary) and drag a *Display* actor onto the page. Each of these actors can be dragged around on the page. However, we would like to connect one to the other. To do this, drag a connection from the output port on the right of the *Const* actor to the input port of the *Display* actor. Lastly, open the *director library* and drag an *SDF Director* onto the page. The director gives a meaning (semantics) to the graph, but for now we don't have to be concerned about exactly what that is.

Now you should have something that looks like figure 2.10. The *Const* actor is going to create our string, and the *Display* actor is going to print it out for us. We need to take care of one small detail to make it look like figure 2.10: we need to tell the *Const* actor that we want the string "Hello World". To do this we need to edit one of the parameters of the *Const*. To do this, either double click on the *Const* actor icon, or right click on the *Const* actor icon and select "Configure". You should see the dialog box in figure 2.11. Enter the string "Hello World" for the value parameter and click the Commit button. Be sure to include the double quotes, so that the expression is interpreted as a string.

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 you open that file.

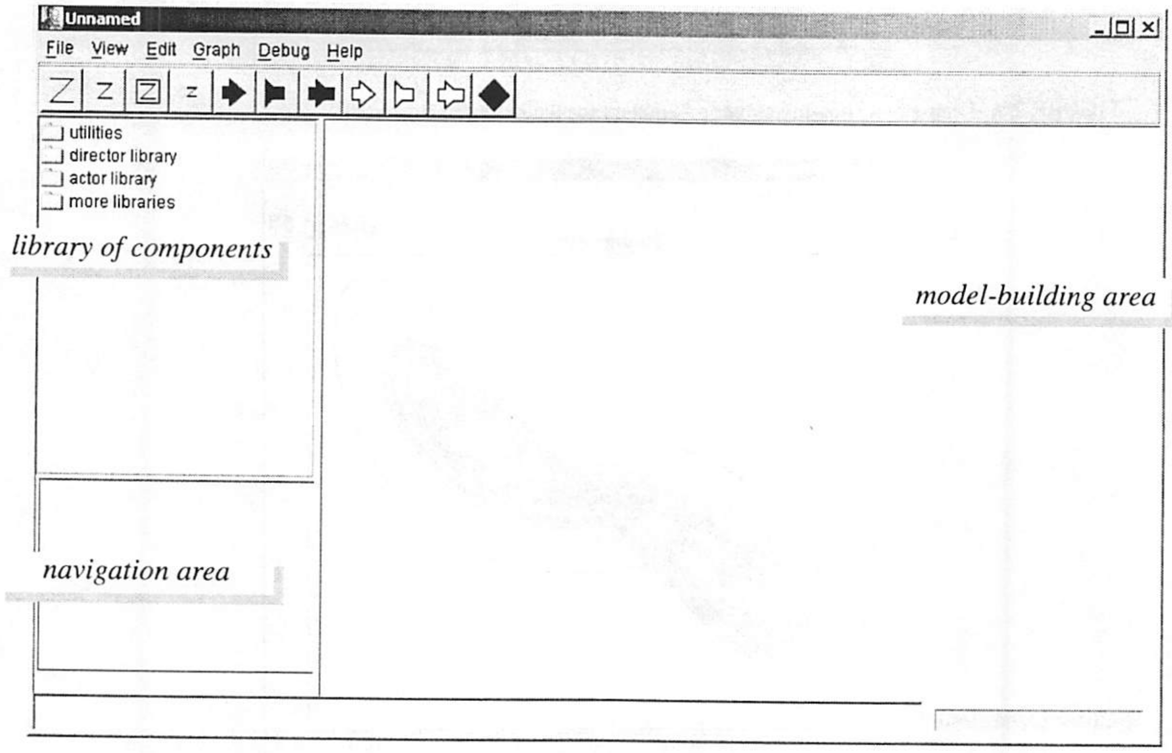


FIGURE 2.9. An empty Vergil Graph Editor.

2.2.5 Running the Model

To run the example, go to the View menu and select the Run Window. If you click the "Go" button, you will see a large number of strings in the display at the right. To stop the execution, click the "Stop" button. To see only one string, change the *iterations* parameter of the SDF Director to 1, which can be done in the run window, or in the graph editor in the same way you edited the parameter of the *Const* actor before. The run window is shown in figure 2.12.

2.2.6 Making Connections

The model constructed above contained only two actors and one connection between them. If you

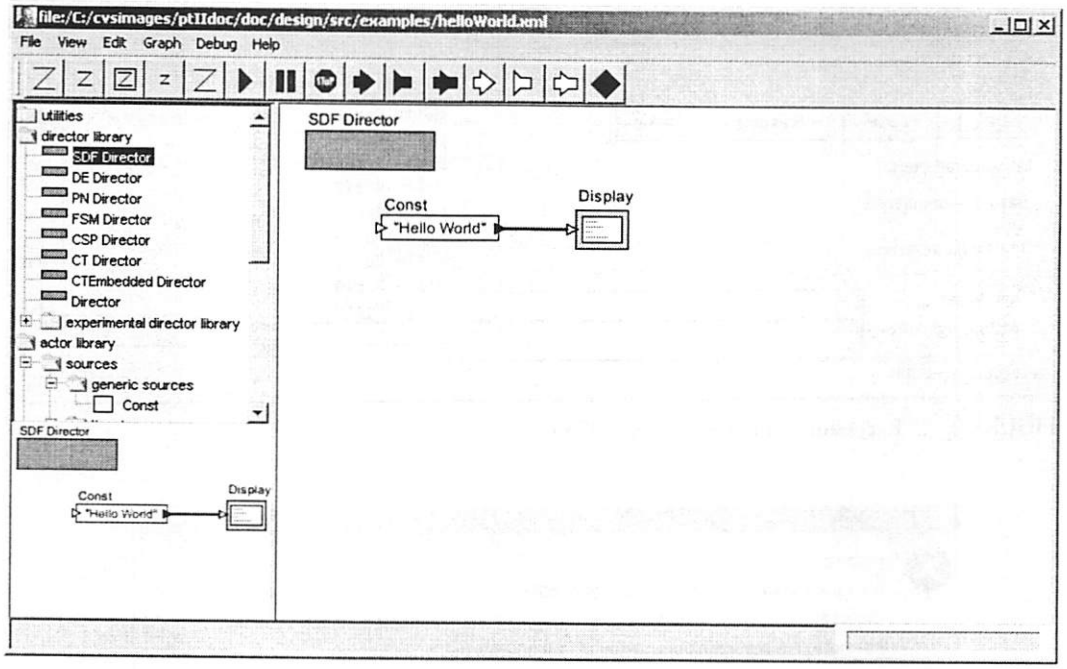


FIGURE 2.10. The Hello World example.

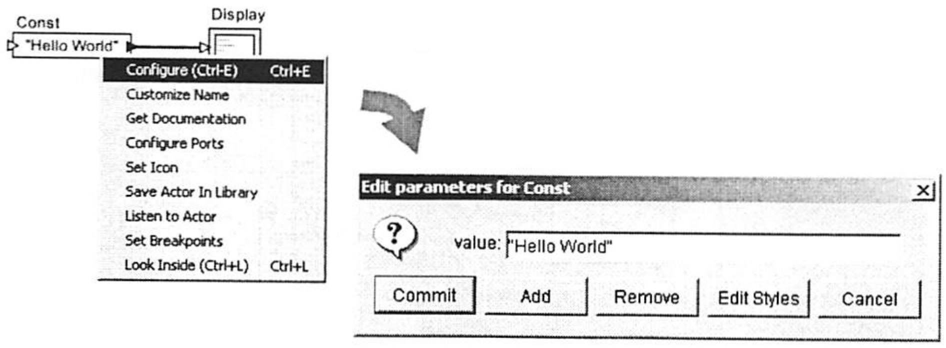
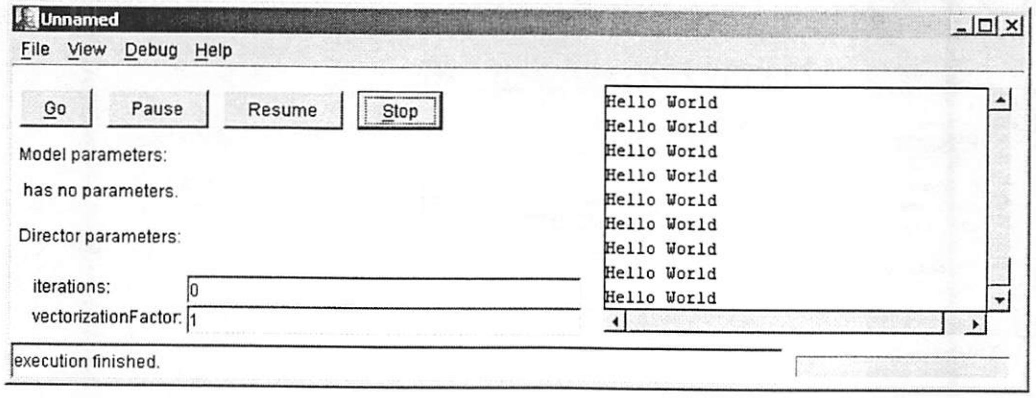


FIGURE 2.11. The Const parameter editor.

move either actor (by clicking and dragging), you will see that the connection is 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 *SDFDirector*, a *Ramp* actor (found in the *sources*) library, a *Display* actor, and a *SequencePlotter* actor, found in the *sinks* library, as shown in figure 2.13. Suppose we wish to route the output of the *Ramp* to both the *Display* and the *SequencePlotter*. If we simply attempt to make the connections, we get the exception shown in figure 2.13. Don't panic! Exceptions are normal and common. The key line in this exception report is the last one, which says

```
Attempt to link more than one relation to a single port.
```



Model parameters:

has no parameters.

Director parameters:

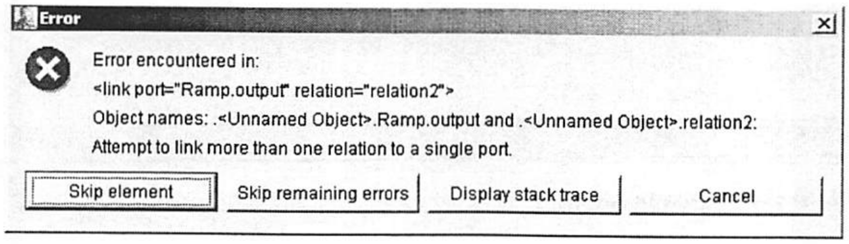
iterations: 0

vectorizationFactor: 1

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

execution finished.

FIGURE 2.12. Execution of the Hello World example.



Error encountered in:
<link port="Ramp.output" relation="relation2">
Object names: .<Unnamed Object>.Ramp.output and .<Unnamed Object>.relation2:
Attempt to link more than one relation to a single port.

FIGURE 2.13. Exception that occurs if you attempt to simply wire the output of the *Ramp* in figure 2.14 to the inputs of the other two actors.

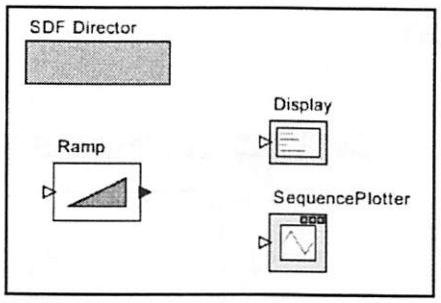


FIGURE 2.14. Three unconnected actors in a model.

The line above that gives the names of the objects involved, which are

```
    Object names: .<Unnamed Object>.Ramp.output and .<Unnamed
                                                  Object>.relation2
```

In Ptolemy II models, all objects have a dotted name. The dots separate elements in the hierarchy. Thus, ".<Unnamed Object>.Ramp.output" is an object named "output" contained by an object named "Ramp", which is contained by an unnamed object (the model itself). The model has no name because we have not assigned one (it acquires a name when we save it).

Why did this exception occur? Ptolemy II supports two distinct flavors of ports, indicated in the diagrams by a filled triangle or an unfilled triangle. The output port of the *Ramp* actor is a *single port*, indicated by a filled triangle, which means that it can only support a single connection. The input port of the *Display* and *SequencePlotter* actors are *multiports*, indicated by unfilled triangles, which means that they can support multiple connections. Each connection is treated as a separate *channel*, which is a path from an output port to an input port (via relations) that can transport a single stream of tokens.

So how do we get the output of the *Ramp* to the other two actors? We need an explicit *relation* in the diagram. A relation is represented in the diagram by a black diamond, as shown in figure 2.15. It can be created by either control-clicking on the background or by clicking on the button in the toolbar with the black diamond on it.

Making a connection to a relation can be tricky, since if you just click and drag on the relation, the relation gets selected and moved. To make a connection, hold the control button while clicking and dragging on the relation.

In the model shown in figure 2.15, 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 to it, a connection to a relation. Relations can also be used to control the routing of wires in the diagram. However, as of the 3.0 release of Ptolemy II, a connection can only have a single relation on it, so the degree to which routing can be controlled is limited.

To explore multiports, try putting some other signal source in the diagram and connecting it to the *SequencePlotter* or to the *Display*. If you explore this fully, you will discover that the *SequencePlotter* can only accept inputs of type *double*, or some type that can be losslessly converted to *double*, such as *int*. These data type issues are explored next.

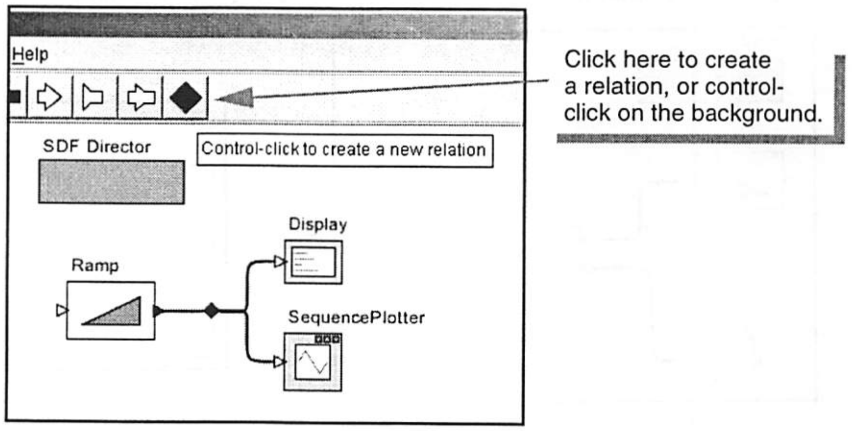

Help
SDF Director
Control-click to create a new relation
Display
Ramp
SequencePlotter
Click here to create a relation, or control-click on the background.

FIGURE 2.15. A relation can be used to broadcast an output from a single port.

2.3 Tokens and Data Types

In the example of figure 2.10, the *Const* actor creates a sequence of values on its output port. The values are encapsulated as *tokens*, and sent to the *Display* actor, which consumes them and displays them in the run window.

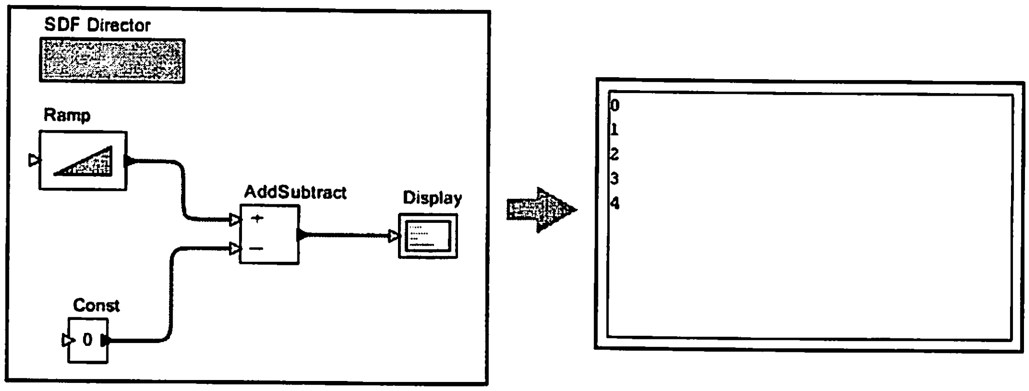
The tokens produced by the *Const* actor can have any value that can be expressed in the Ptolemy II *expression language*. We will say more about the expression language in chapter 3, "Expressions", but for now, try giving the value 1 (the integer with value one), or 1.0 (the floating-point number with value one), or {1.0} (An array containing a one), 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 expressions.

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*, meaning that they can operate on or produce data with multiple types. The behavior may even be different for different types. Multiplying matrices, for example, is not the same as multiplying integers, but both are accomplished by the *MultiplyDivide* actor in the *math library*. Ptolemy II includes a sophisticated type system that allows this to be done efficiently and safely.

To explore data types a bit further, try creating the model in figure 2.16. The *Ramp* actor is listed under *sources*, *sequence sources*, and the *AddSubtract* actor is listed under *math*. Set the *value* parameter of the constant to be 0 and the *iterations* parameter of the director to 5. Running the model should result in 5 numbers between 0 and 4, as shown in the figure. These are the values produced by the *Ramp*, which are having the value of the *Const* actor subtracted from them. Experiment with changing the value of the *Const* actor and see how it changes the 5 numbers at the output.

Now for the real test: 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 2.17. Do not worry; exceptions are a normal part of constructing (and debugging) models. In this case, the exception window is telling you that you have tried to subtract a string value from an integer value, which doesn't make much sense at all (following Java, adding strings *is* allowed). This is an example of a type error.

Exceptions can be a very useful debugging tool, particularly if you are developing your own components in Java. To illustrate how to use them, click on the Display Stack Trace button in the exception window of figure 2.17. You should see the stack trace shown in figure 2.18. This window displays the execution sequence that resulted in the exception. For example, the line



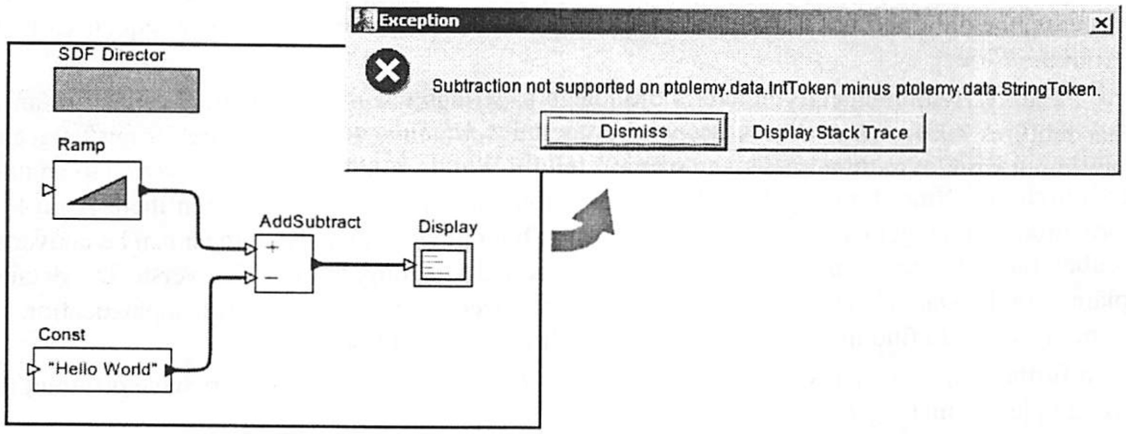
```
0
1
2
3
4
```

FIGURE 2.16. Another example, used to explore data types in Ptolemy II.

```
at ptolemy.data.IntToken.subtract(IntToken.java:547)
```

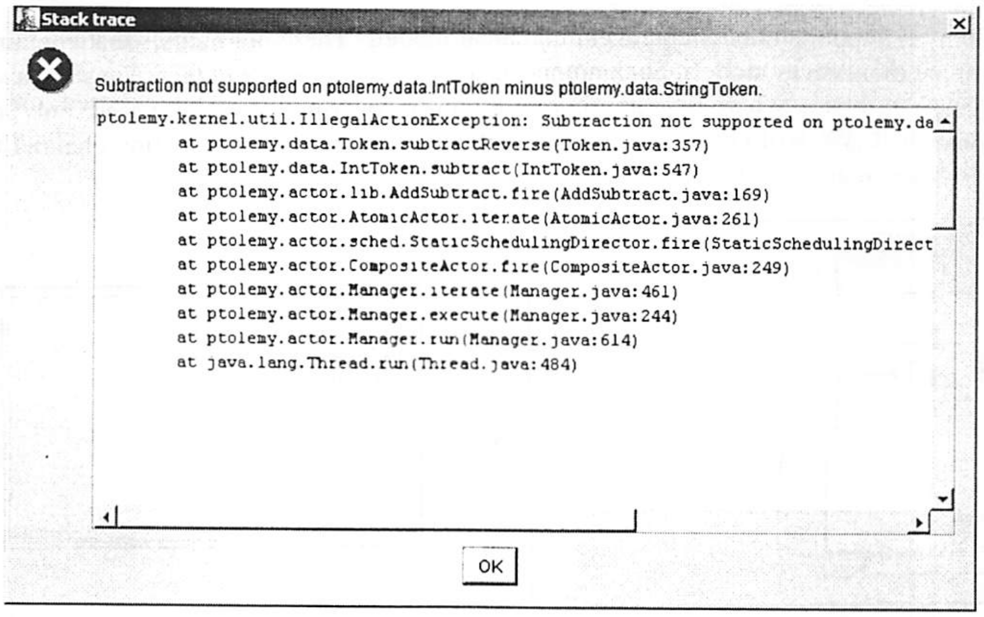
indicates that the exception occurred within the subtract() method of the class ptolemy.data.IntToken, at line 547 of the source file IntToken.java. Since Ptolemy II is distributed with source code (except in the Windows installer version and the Web Start Web Edition), this can be very useful information. 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 IntToken.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



Subtraction not supported on ptolemy.data.IntToken minus ptolemy.data.StringToken.

FIGURE 2.17. An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from integers.



Subtraction not supported on ptolemy.data.IntToken minus ptolemy.data.StringToken.

```
ptolemy.kernel.util.IllegalActionException: Subtraction not supported on ptolemy.da
        at ptolemy.data.Token.subtractReverse(Token.java:357)
        at ptolemy.data.IntToken.subtract(IntToken.java:547)
        at ptolemy.actor.lib.AddSubtract.fire(AddSubtract.java:169)
        at ptolemy.actor.AtomicActor.iterate(AtomicActor.java:261)
        at ptolemy.actor.sched.StaticSchedulingDirector.fire(StaticSchedulingDirect
        at ptolemy.actor.CompositeActor.fire(CompositeActor.java:249)
        at ptolemy.actor.Manager.iterate(Manager.java:461)
        at ptolemy.actor.Manager.execute(Manager.java:244)
        at ptolemy.actor.Manager.run(Manager.java:614)
        at java.lang.Thread.run(Thread.java:484)
```

FIGURE 2.18. Stack trace for the exception shown in figure 2.17.

replaced by slashes; in this case, it is at $PTII/ptolemy/data/IntToken.java (the slashes might be backslashes under Windows).

Let's try a small change to the model to get something that does not trigger an exception. Disconnect the *Const* from the lower port of the *AddSubtract* actor and connect it instead to the upper port, as shown in figure 2.19. You can do this by selecting the connection and deleting it (using the delete key), then 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 it is a *multiport*, meaning that you can make more than one connection to it. Now when you run the model you should see strings like "0HelloWorld", as shown in the figure.

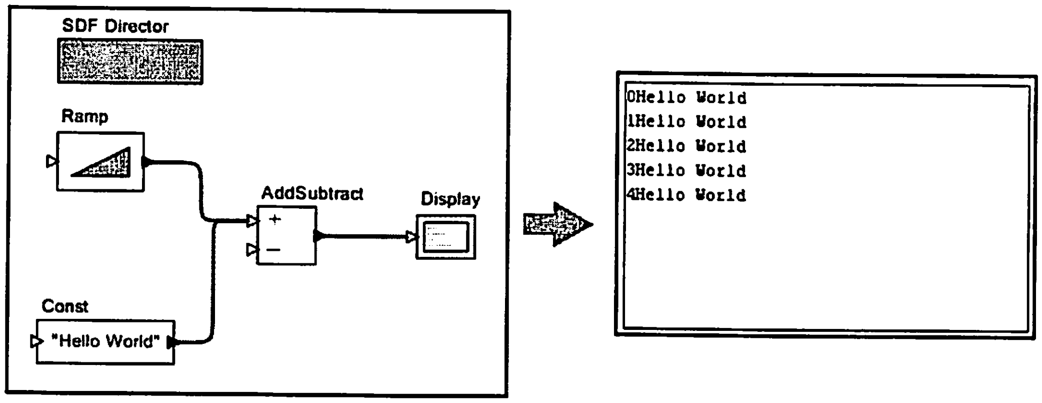
There are two interesting things going on here. The first is that, as in Java, strings are added by concatenating them. The second is that the integers from the *Ramp* are converted to strings and concatenated with the string "Hello World". All the connections to a multiport must have the same type. In this case, the multiport has a sequence of integers coming in (from the *Ramp*) and a sequence of strings (from the *Const*).

Ptolemy II automatically converts the integers to strings when integers are provided to an actor that requires strings. But in this case, why does the *AddSubtract* actor require strings? Because it would not work to require integers; the string "Hello World" would have to be converted to an integer. As a rough guideline, Ptolemy II will perform automatic type conversions when there is no loss of information. An integer can be converted to a string, but not vice versa. An integer can be converted to a double, but not vice versa. An integer can be converted to a long, but not vice versa. The details are explained in the Data chapter, but many users will not need to understand the full sophistication of the system. You should find that most of the time it will just do what you expect.

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

2.4 Hierarchy

Ptolemy II supports (and encourages) hierarchical models. These are models that contain components that are themselves models. Such components are called *composite actors*. Consider a small signal processing problem, where we are interested in recovering a signal based only on noisy measurements of it. We will create a composite actor modeling a communication channel that adds noise, and then use that actor in a model.



```
0Hello World
1Hello World
2Hello World
3Hello World
4Hello World
```

FIGURE 2.19. Addition of a string to an integer.

2.4.1 Creating a Composite Actor

First open a new graph editor and drag in a *Typed Composite Actor* from the *utilities* library. This actor is going to add noise to our measurements. First, using the context menu (obtained by right clicking over the composite actor), select "Customize Name", and give the composite a better name, like "Channel", as shown in figure 2.20. Then, using the context menu again, select "Look Inside" on the actor. You should get a blank graph editor, as shown in figure 2.21. The original graph editor is still open. To see it, move the new graph editor window by dragging the title bar of the window.

2.4.2 Adding Ports to a Composite Actor

First we have to add some ports to the composite actor. There are several ways to do this, but clicking on the port buttons in the toolbar is probably the easiest. You can explore the ports in the toolbar by lingering with the mouse over each button in the toolbar. A tool tip pops up that explains the button.

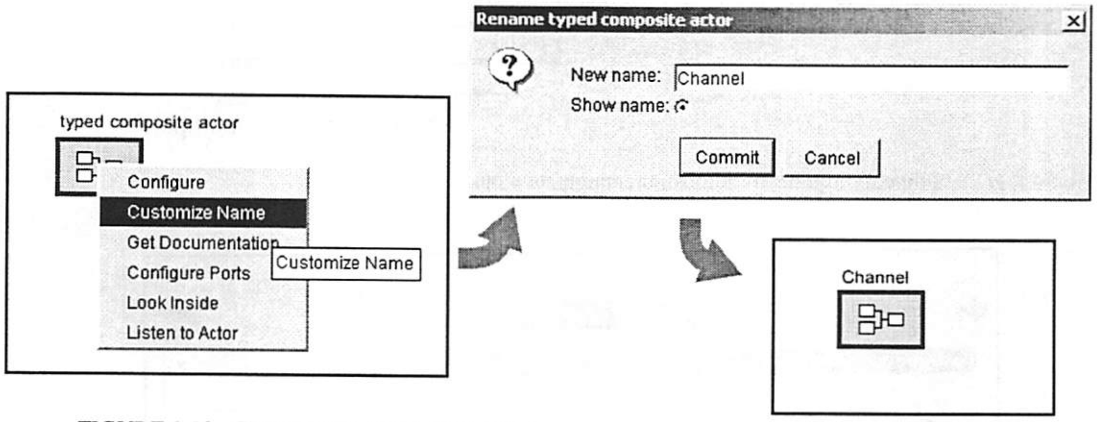


FIGURE 2.20. Changing the name of an actor.

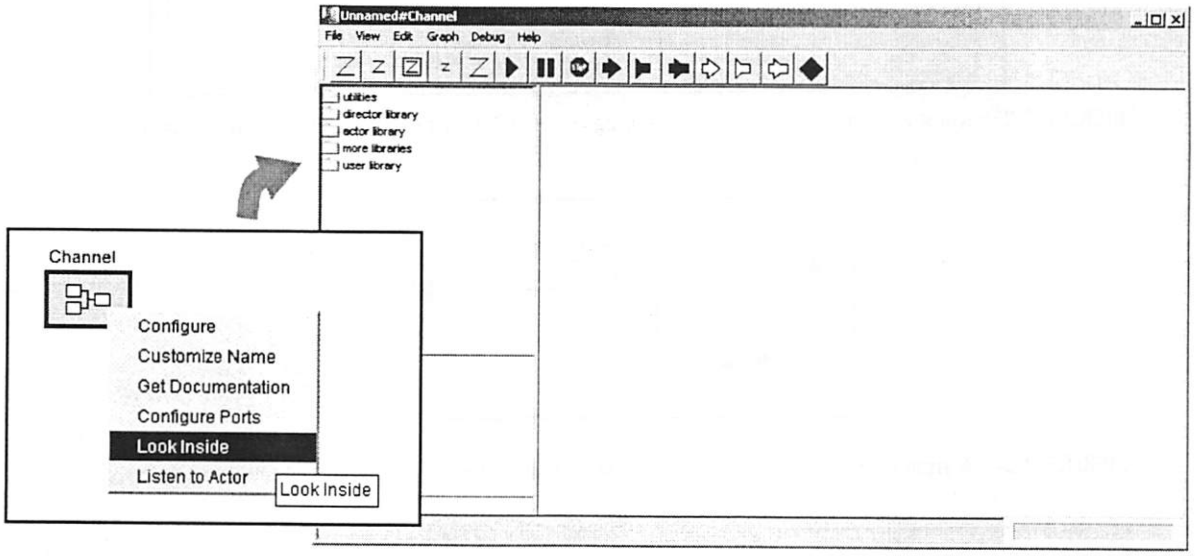


FIGURE 2.21. Looking inside a composite actor.

The buttons are summarized in figure 2.22. Create an input port and an output port and rename them *input* and *output* right by clicking on the ports and selecting "Customize Name". Note that, as shown in figure 2.23, you can also right click on the background of the composite actor and select *Configure Ports* to change whether a port is an input, an output, or a multiport. The resulting dialog also allows you to set the type of the port, although much of the time you will not need to do this, since the type inference mechanism in Ptolemy II will figure it out from the connections.

Then using these ports, create the diagram shown in figure 2.24[1]. The *Gaussian* actor creates values from a Gaussian distributed random variable, and is found in the *random* library. Now if you close

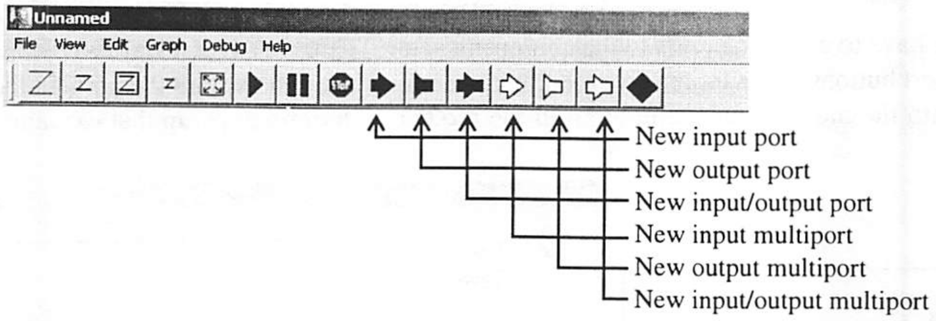
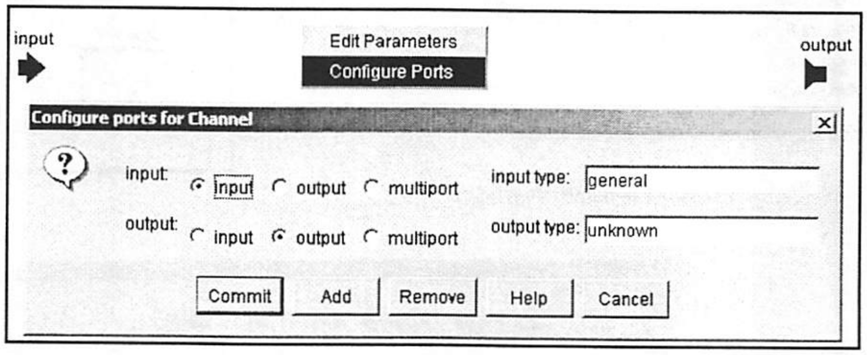


FIGURE 2.22. Summary of toolbar buttons for creating new ports.



input: (• input) (output) (multiport) input type: general
output: (input) (• output) (multiport) output type: unknown

FIGURE 2.23. Right clicking on the background brings up a dialog that can be used to configure ports.

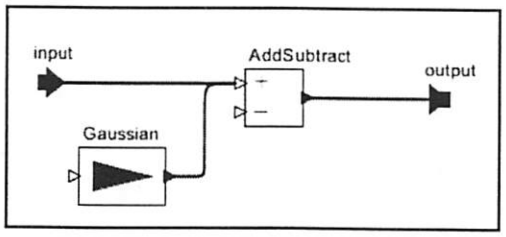


FIGURE 2.24. A simple channel model defined as a composite actor.

1. **Hint:** to create a connection starting on one of the external ports, hold down the control key when dragging.

this editor and return to the previous one, you should be able to easily create the model shown in figure 2.25. The *Sinewave* actor is listed under *sources*, and the *SequencePlotter* actor is found in *sinks*. Notice that the *Sinewave* actor is also a hierarchical model, as suggested by its red outline (try looking inside). If you execute this model (you will probably want to set the iterations to something reasonable, like 100), you should see something like figure 2.26.

2.4.3 Setting the Types of Ports

In the above example, we never needed to define the types of any ports. The types were inferred from the connections. Indeed, this is usually the case in Ptolemy II, but occasionally, you will need to set the types of the ports. Notice in figure 2.23 that there is a position in the dialog box that configures ports for specifying the type. Thus, to specify that a port has type *boolean*, you could enter *boolean* into the dialog box. There are other commonly used types: *complex*, *double*, *fixedpoint*, *general*, *int*, *long*, *matrix*, *object*, *scalar*, *string*, and *unknown*. Let's take a more complicated case. How would you specify that the type of a port is a double matrix? Easy:

```
    [double]
```

This expression actually creates a 1 by 1 matrix containing a double (the value of which is irrelevant). It thus serves as a prototype to specify a double matrix type. Similarly, we can specify an array of com-

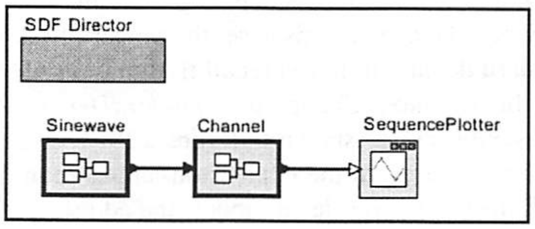
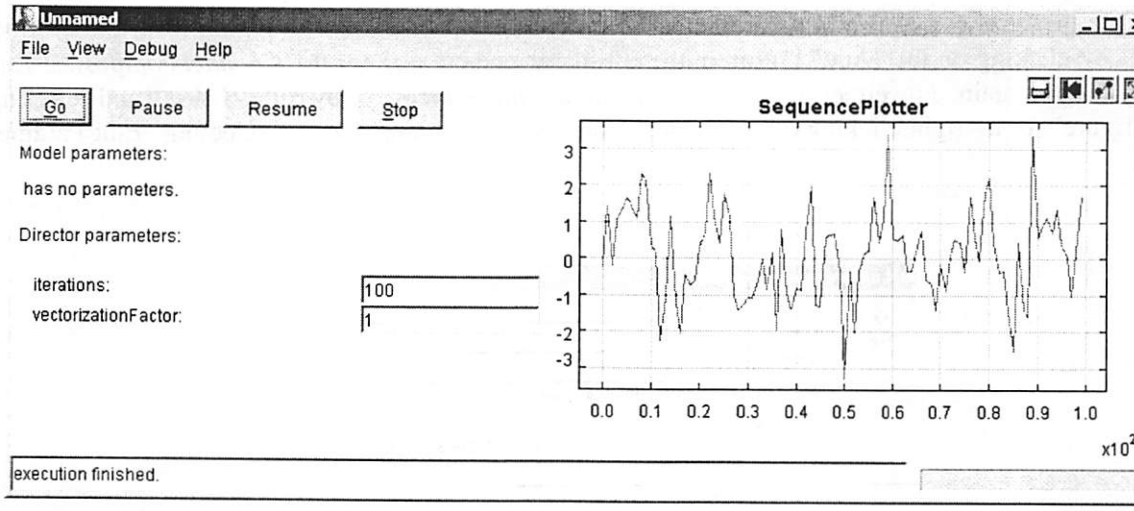


FIGURE 2.25. A simple signal processing example that adds noise to a sinusoidal signal.



Model parameters:
has no parameters.
Director parameters:
iterations: 100
vectorizationFactor: 1

execution finished.

FIGURE 2.26. The output of the simple signal processing model in figure 2.25.

plex numbers as

```
    {complex}
```

In the Ptolemy II expression language, square braces are used for matrices, and curly braces are used for arrays. What about a record containing a string named "name" and an integer named "address"? Easy:

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

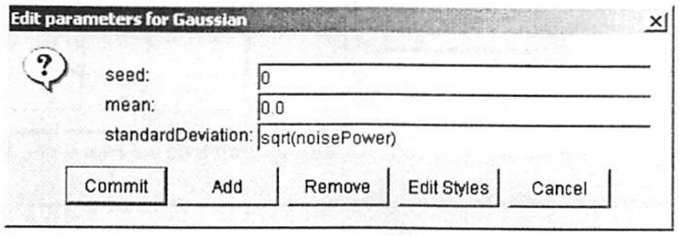
2.5 Annotations and Parameterization

In this section, we will enhance the model in figure 2.25 in a number of ways.

First, notice from figure 2.26 that the noise overwhelms the sinusoid, making it barely visible. A useful channel model would have a parameter that sets the level of the noise. Look inside the channel model, and add a parameter by dragging one in from the *utilities* library, as shown in figure 2.27. Right click on the parameter to change its name to "noisePower". (In order to be able to use this parameter in expressions, the name cannot have any spaces in it.) Also, right click or double click on the parameter to change its default value to 0.1.

Now we can use this parameter. First, let's use it to set the amount of noise. The *Gaussian* actor has a parameter called *standardDeviation*. In this case, the power of the noise is equal to the variance of the Gaussian, not the standard deviation. If you recall from basic statistics, the standard deviation is equal to the square root of the variance. Change the *standardDeviation* parameter of the *Gaussian* actor so its value is "sqrt(noisePower)", as shown in figure 2.28. This is an expression that references the *noisePower* parameter. We will explain the expression language in the next chapter. But first, let check our improved model. Return to the top-level model, and edit the parameters of the *Channel* actor (by either double clicking or right clicking and selecting "Configure"). Change the noise power from the default 0.1 to 0.01. Run the model. You should now get a relatively clean sinusoid like that shown in figure 2.29.

Note that you can also add parameters to a composite actor without dragging from the *utilities* library by clicking on the "Add" button in the edit parameters dialog for the *Channel* composite. This dialog can be obtained by either double clicking on the *Channel* icon, or by right clicking and selecting "Configure", or by right clicking on the background inside the composite and selecting "Edit Parameters".


Edit parameters for Gaussian

seed: 0

mean: 0.0

standardDeviation: sqrt(noisePower)

FIGURE 2.28. The standard deviation of the *Gaussian* actor is set to the square root of the noise power.

There are several other useful enhancements you could make to this model. Try dragging an *annotation* from the *utilities* library and creating a title on the diagram. Also, try setting the title of the plot by clicking on the second button from the right in the row of buttons at the top right of the plot. This button produces the tool tip "Set the plot format" and bring up the format control window.

2.6 Navigating Larger Models

Sometimes, a model gets large enough that it is not convenient to view it all at once. There are four toolbar buttons, shown in figure 2.27 that help. These buttons permit zooming in and out. The "Zoom reset" button restores the zoom factor to the "normal" one, and the "Zoom fit" calculates the zoom factor so that the entire model is visible in the editor window.

In addition, it is possible to pan over a model. Consider the window shown in figure 2.31. Here, we have zoomed in so that icons are larger than the default. The *pan window* at the lower left shows the

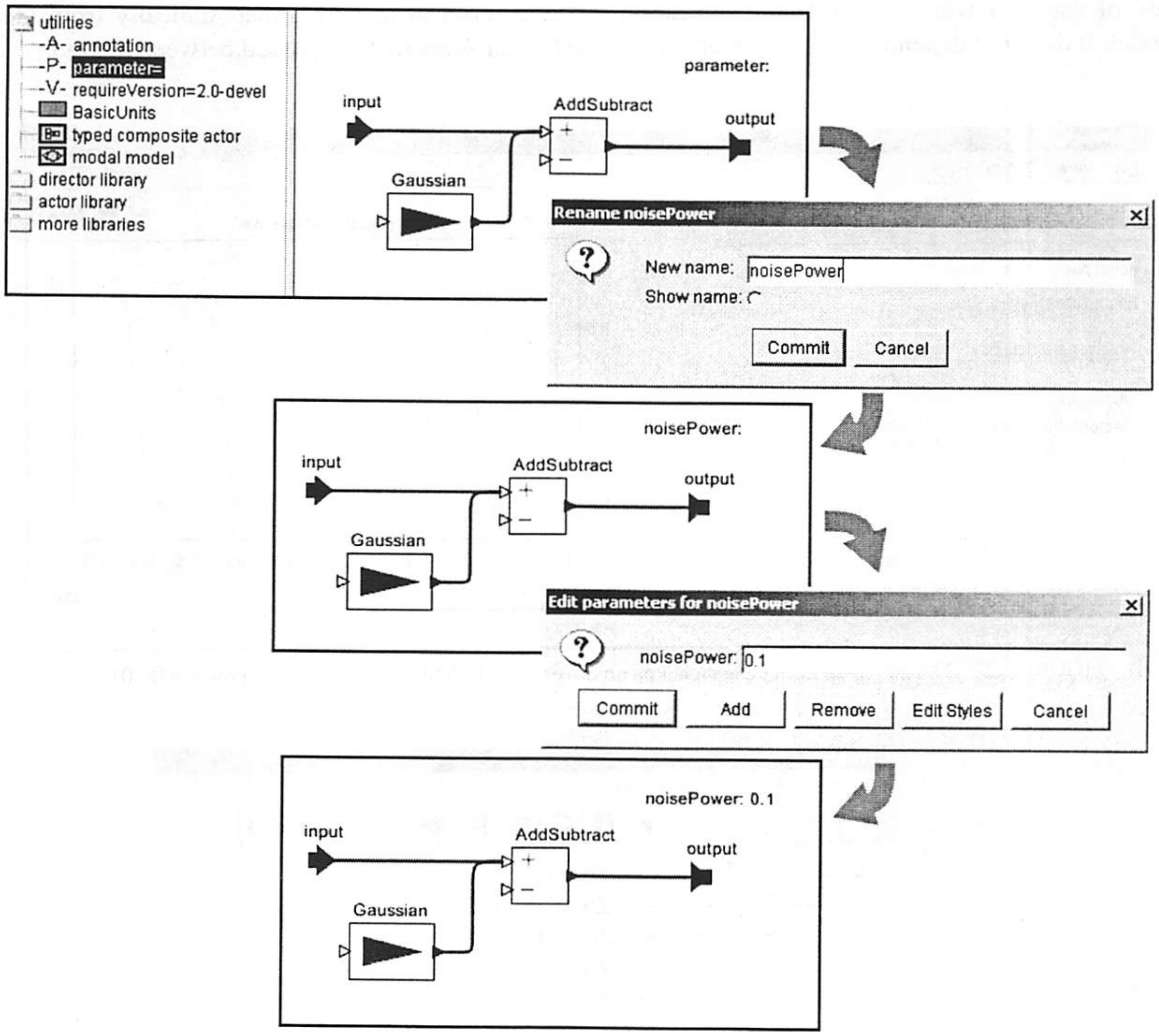


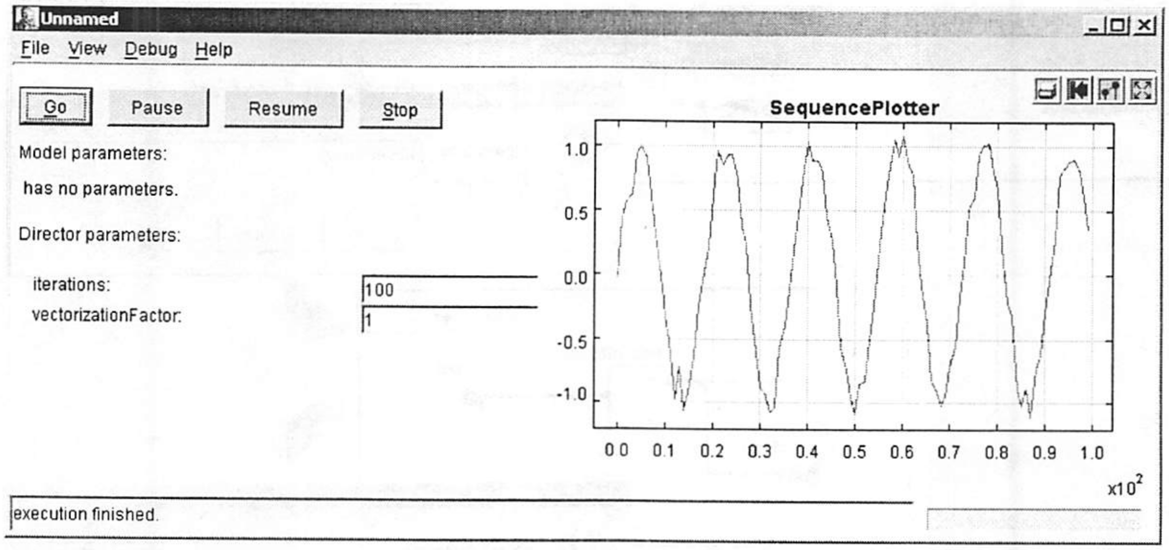
FIGURE 2.27. Adding a parameter to the channel model.

entire model, with a red box showing the visible portion of the model. By clicking and dragging in the pan window, it is easy to navigate around the entire model. Clicking on the "Zoom fit" button in the toolbar results in the editor area showing the entire model, just as the pan window does.

2.7 Domains

A key innovation in Ptolemy II is that, unlike other design and modeling environments, there are several available *models of computation* that define the meaning of a diagram. In the above examples, we directed you to drag in an *SDF Director* without justifying why. A director in Ptolemy II gives meaning (semantics) to a diagram. It specifies what a connection means, and how the diagram should be executed. In Ptolemy II terminology, the director realizes a *domain*. Thus, when you construct a model with an SDF director, you have constructed a model "in the SDF domain."

The SDF director is fairly easy to understand. "SDF" stands for "synchronous dataflow." In dataflow models, actors are invoked (fired) when their input data is available. SDF is particularly simple case of dataflow where the order of invocation of the actors can be determined statically from the model. It does not depend on the data that is processed (the tokens that are passed between actors).



Model parameters:
has no parameters.
Director parameters:
iterations: 100
vectorizationFactor: 1

execution finished.

FIGURE 2.29. The output of the simple signal processing model in figure 2.25.with noise power = 0o.01

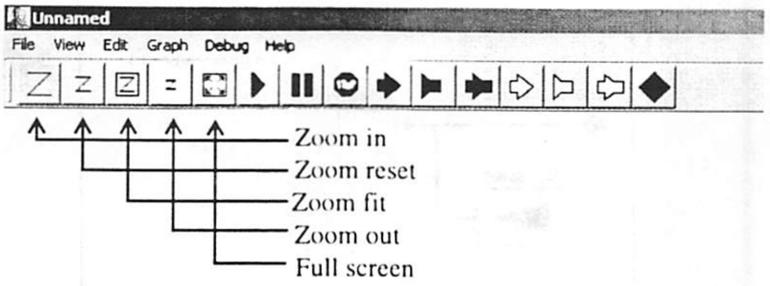


FIGURE 2.30. Summary of toolbar buttons for zooming and fitting.

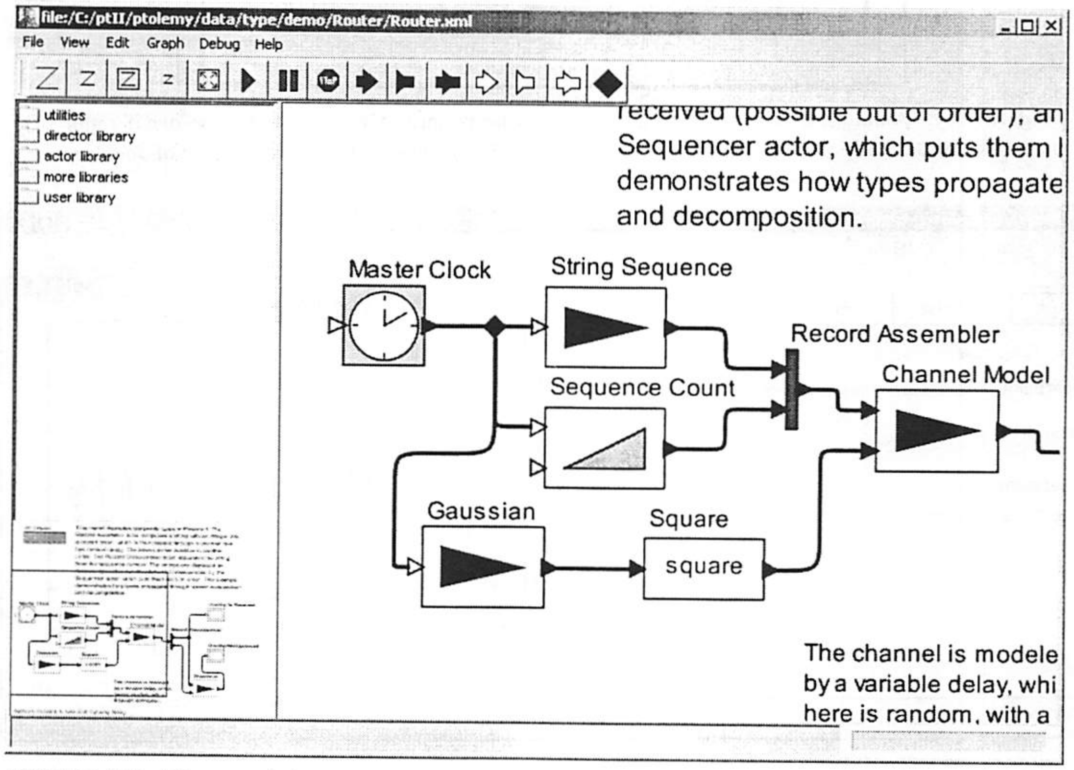
But there are other models of computation available in Ptolemy II. It can be difficult to determine which one to use without having experience with several. Moreover, you will find that although most actors in the library do *something* in any domain in which you use them, they do not always do something useful. It is important to understand the domain you are working with and the actors you are using. Here, we give a very brief introduction to some of the domains. But we begin first by explaining some of the subtleties in SDF.

2.7.1 SDF and Multirate Systems

So far we have been dealing with relatively simple systems. They are simple in the sense that each actor produces and consumes one token from each port at a time. In this case, the SDF director simply ensures that an actor fires after the actors whose output values it depends on. The total number of output values that are created by each actor is determined by the number of iterations, but in this simple case only one token would be produced per iteration.

It turns out that the SDF scheduler is actually much more sophisticated. It is capable of scheduling the execution of actors with arbitrary prespecified data rates. Not all actors produce and consume just a single sample each time they are fired. Some require several input token before they can be fired, and produce several tokens when they are fired.

One such actor is a spectral estimation actor. figure 2.32 shows a system that computes the spectrum of the same noisy sine wave that we constructed in figure 2.25. The *Spectrum* actor has a single parameter, which gives the *order* of the FFT used to calculate the spectrum. Figure 2.33 shows the output of the model with *order* set to 8 and the number of *iterations* set to 1. **Note that there are 256 out-**



received (possible out of order), an
Sequencer actor, which puts them i
demonstrates how types propagate
and decomposition.

The channel is modele
by a variable delay, whi
here is random, with a

FIGURE 2.31. The pan window at the lower left has a red box representing the visible are of the model in the main editor window. This red box can be moved around to view different parts of the model.

put samples output from the *Spectrum* actor. This is because the *Spectrum* actor requires 2^8, or 256 input samples to fire, and produces 2^8, or 256 output samples when it fires. Thus, one iteration of the model produces 256 samples. The *Spectrum* actor makes this a *multirate* model, because the firing rates of the actors are not all identical.

It is common in SDF to construct models that require exactly one iteration to produce a useful result. In some multirate models, it can be complicated to determine how many firings of each actor occur per iteration of the model. See the SDF chapter for details.

A second subtlety with SDF models is that if there is a feedback loop, as in figure 2.34, then the loop must have at least one instance of the *SampleDelay* actor in it (found in the *flow control* library). Without this actor, the loop will deadlock. The *SampleDelay* actor produces initial tokens on its output, before the model begins firing. The initial tokens produced are given by a the *initialOutputs* parameter, which specifies an array of tokens. These initial tokens enable downstream actors and break the circular dependencies that would result otherwise from a feedback loop.

A final issue to consider with the SDF domain is time. Notice that in all the examples above we have suggested using the *SequencePlotter* actor, not the *TimedPlotter* actor, which is in the same *sinks* library. This is because the SDF domain does not include in its semantics a notion of time. Time does

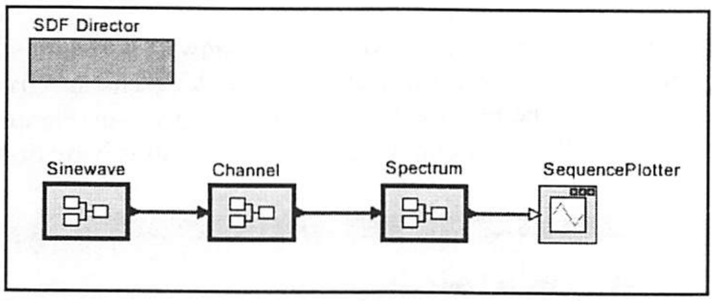
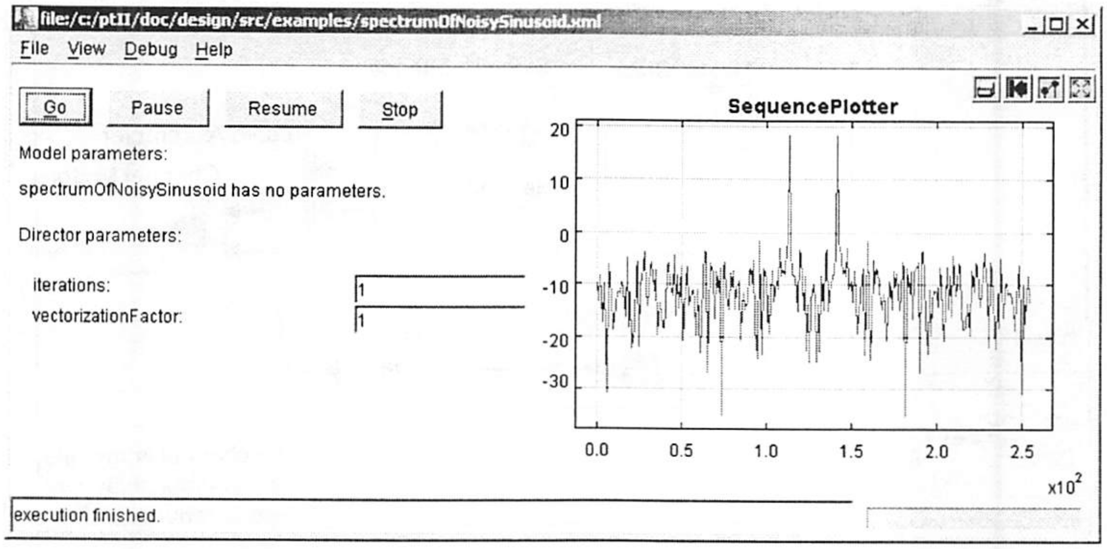


FIGURE 2.32. A multirate SDF model. The *Spectrum* actor requires 256 tokens to fire, so one iteration of this model results in 256 firings of *Sinewave, Channel,* and *SequencePlotter*, and one firing of *Spectrum*.



Model parameters:

spectrumOfNoisySinusoid has no parameters.

Director parameters:

iterations: 1

vectorizationFactor: 1

execution finished.

FIGURE 2.33. A single iteration of the SDF model in figure 2.32 produces 256 output tokens.

not advance as an SDF model executes, so the *TimedPlotter* actor would produce very uninteresting results, where the horizontal axis value would always be zero. The *SequencePlotter* actor uses the index in the sequence for the horizontal axis. The first token received is plotted at horizontal position 0, the second at 1, the third at 2, etc. The next domain we consider, DE, includes much stronger notion of time, and it is almost always more appropriate in the DE domain to use the *TimedPlotter* actor.

2.7.2 Discrete-Event Systems

In discrete-event (DE) systems, the 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 its time stamp is a double-precision floating-point number. This is different from dataflow, where a signal consists of a sequence of tokens, and there is no time significance in the signal.

A DE model executes chronologically, processing the oldest events first. Time advances as events are processed. There is potential confusion, however, between *model time*, the time that evolves in the model, and *real time*, the time that elapses in the real world while the model executes (also called *wall-clock time*). Model time may advance more rapidly than real time or more slowly. The DE director has a parameter, *synchronizeToRealTime*, that, when set to true, attempts to synchronize the two notions of time. It does this by delaying execution of the model, if necessary, allowing real time to catch up with model time.

Consider the DE model shown in figure 2.35. This model includes a *PoissonClock* actor, a *CurrentTime* actor, and a *WallClockTime* actor, all found in the *sources* library. The *PoissonClock* actor generates a sequence of events with random times, where the time between events is exponentially distributed. Such an event sequence is known as a Poisson process. The value of the events produced by the *PoissonClock* actor is a constant, but the value of that constant is ignored in this model. Instead, these events trigger the *CurrentTime* and *WallClockTime* actors. The *CurrentTime* actor outputs an event with the same time stamp as the input, but whose value is the current model time (equal to the time stamp of the input). The *WallClockTime* actor an event with the same time stamp as the input, but whose value is the current real time, in seconds since initialization of the model.

The plot in figure 2.35 shows an execution. Note that model time has advanced approximately 10 seconds, but real time has advanced almost not at all. In this model, model time advances much more rapidly than real time. If you build this model, and set the *synchronizeToRealTime* parameter of the director to true, then you will find that the two plots coincide almost perfectly.

A significant subtlety in using the DE domain is in how simultaneous events are handled. Simultaneous events are simply events with the same time stamp. We have stated that events are processed in chronological order, but if two events have the same time stamp, then there is some ambiguity. Which one should be processed first? If the two events are on the same signal, then the answer is simple: pro-

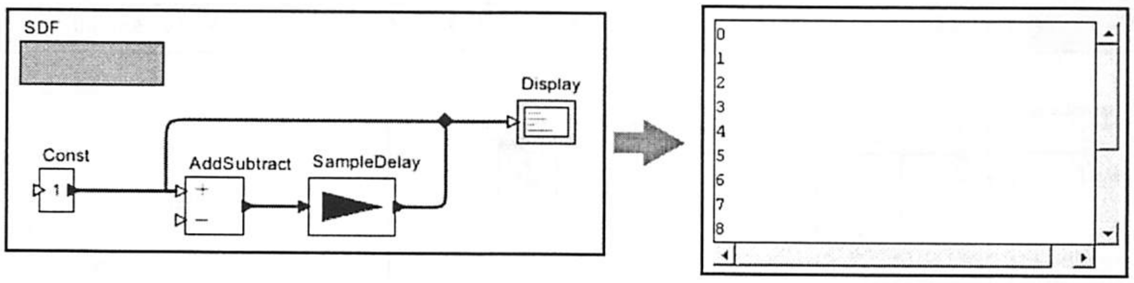


FIGURE 2.34. An SDF model with a feedback loop must have at least one instance of the *SampleDelay* actor in it.

cess first the one that was produced first. However, if the two events are on different signals, then the answer is not so clear.

Consider the model shown in figure 2.36, which produces a histogram of the interarrival times of events from the *PoissonClock* actor. In this model, we calculate the difference between the current

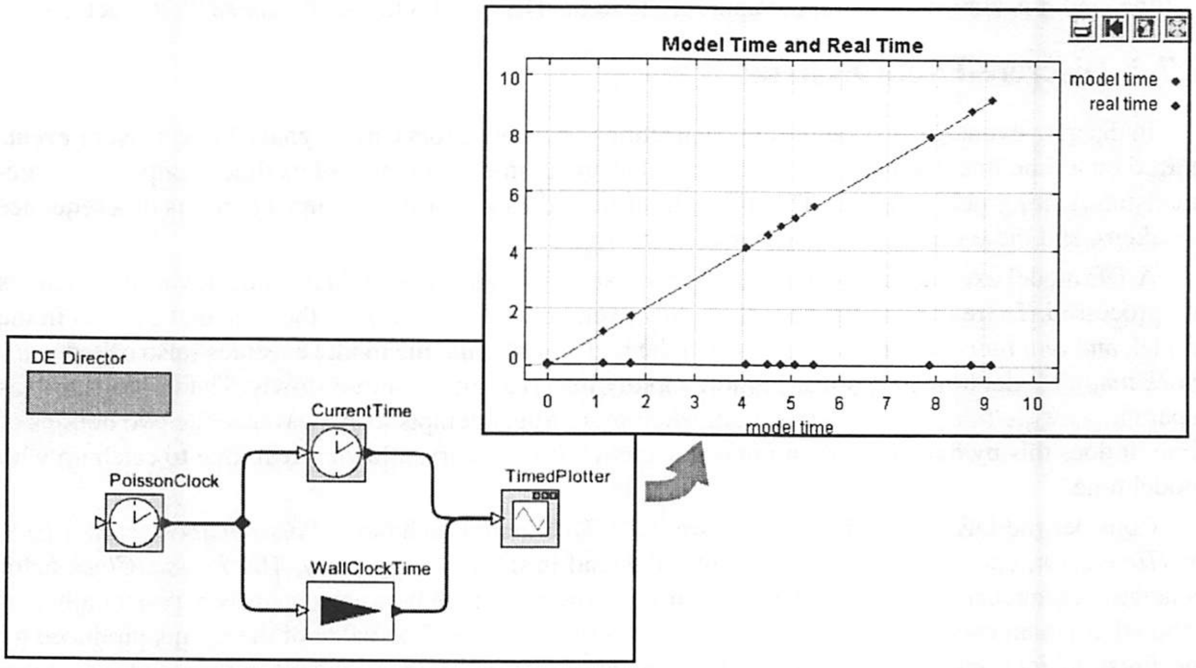


FIGURE 2.35. Model time vs. real time (wall clock time).

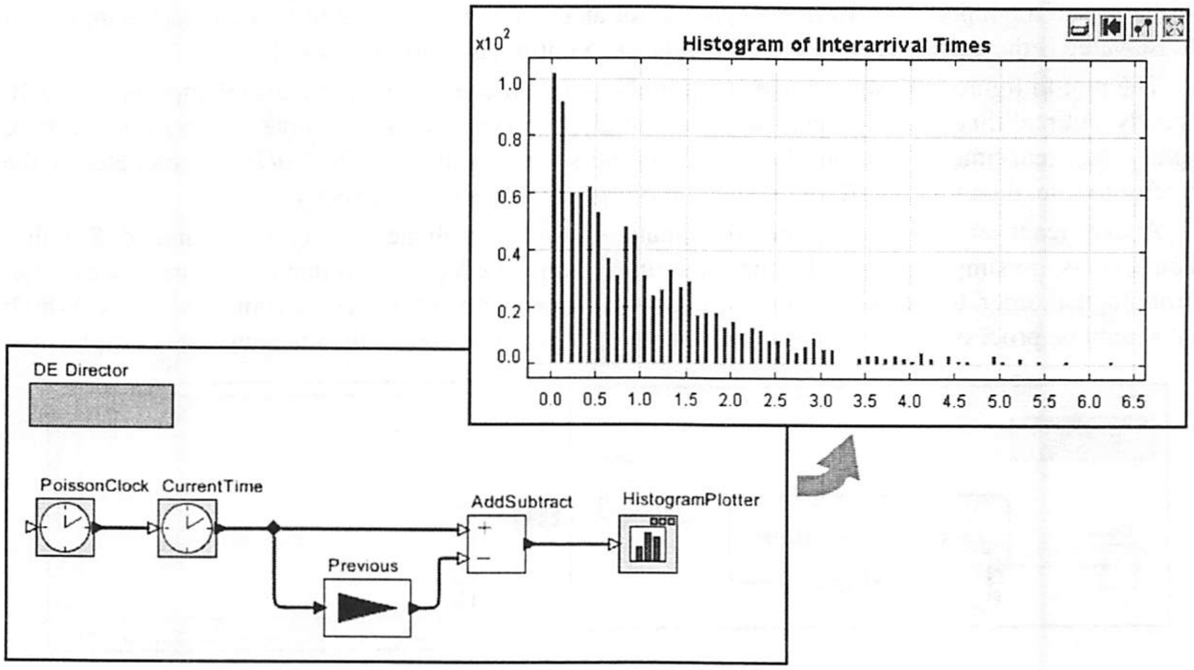


FIGURE 2.36. Histogram of interarrival times, illustrating handling of simultaneous events.

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). Thus, when the *PoissonClock* actor produces an output, there will be two simultaneous events, one at the input to the *plus* port of the *AddSubtract* actor, and one at the input of the *Previous* actor. Should the director fire the *AddSubtract* actor or the *Previous* actor? Either seems OK if it is to respect chronological order, but it seems intuitive that the *Previous* actor should be fired first.

It is helpful to know how the *AddSubtract* actor works. When it fires, it adds all available tokens on the *plus* port, and subtracts all available tokens on the *minus* port. If the *AddSubtract* actor fires before the *Previous* actor, then the only available token will be the one on the *plus* port, and the expected subtraction will not occur. Intuitively, we would expect the director to invoke the *Previous* actor before the *AddSubtract* actor so that the subtraction occurs.

How does the director deliver on the intuition that the *Previous* actor should be fired first? Before executing the model, the DE director constructs a *topological sort* of the model. A topological sort is simply a list of the actors in data-precedence order. For the model in figure 2.36, there is only one allowable topological sort:

- *PoissonClock, CurrentTime, Previous, AddSubtract, HistogramPlotter*

In this list, *AddSubtract* is after *Previous*. So the when they have simultaneous events, the DE director fires *Previous* first.

Thus, the DE director, by analyzing the structure of the model, usually delivers the intuitive behavior, where actors that produce data are fired before actors that consume their results, even in the presence of simultaneous events.

There remains one key subtlety. If the model has a directed loop, then a topological sort is not possible. In the DE domain, every feedback loop is required to have at least one actor in it that introduces a time delay, such as the *TimedDelay* actor, which can be found in the *domain specific* library under *discrete-event* (this library is shown on the left in figure 2.37). Consider for example the model shown in figure 2.37. That model has a *Clock* actor, which is set to produce 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.

Occasionally, you will need to put a *TimedDelay* actor in a feedback loop with a delay of 0.0. This is particularly true if you are building complex models that mix domains, and there is a delay inside a composite actor that the DE director cannot recognize as a delay. The *TimedDelay* actor with a delay of 0.0 can be thought of as a way to let the director know that there is a time delay in the preceding actor, without specifying the amount of the time delay.

2.7.3 Continuous-Time Systems

The continuous-time domain (CT) is another relatively mature domain with semantics considerably different from either DE or SDF. In CT, the signals sent along connections between actors are usually continuous-time signals. A CT example is described above in section 2.2.3.

The CT domain can also handle discrete events. These events are usually related to a continuous-time signal, for example representing a zero-crossing of the continuous-time signal. The CT director is quite sophisticated in its handling of such mixed signal systems.

2.8 Hybrid Systems and Modal Models

Hybrid systems are models that combine continuous dynamics with discrete mode changes. They are created in Ptolemy II by creating a *modal model*, found in the *utilities* library. We start by examining a pre-built modal model, and conclude by illustrating how to construct one. Modal models can be constructed with other domains besides CT, but this section will concentrate on CT. Feel free to examine other examples of modal models given in the quick tour, figure 2.3.

2.8.1 Examining a Pre-Built Model

Consider the bouncing ball example, which can be found under "Hybrid Systems" in figure 2.3. The top-level contents of this model is shown in figure 2.38. It contains only two actors, a *Ball Model* and a *TimedPlotter*. The *Ball Model* is an instance of the *modal model* found in the *utilities* library, but renamed. If you execute the model, you should see a plot like that in the figure. The continuous dynamics correspond to the times when the ball is in the air, and the discrete events correspond to the times when the ball hits the surface and bounces.

If you look inside the *Ball Model*, you will see something like figure 2.39. Figure 2.39 shows a state-machine editor, which has a slightly different toolbar and a significantly different library at the

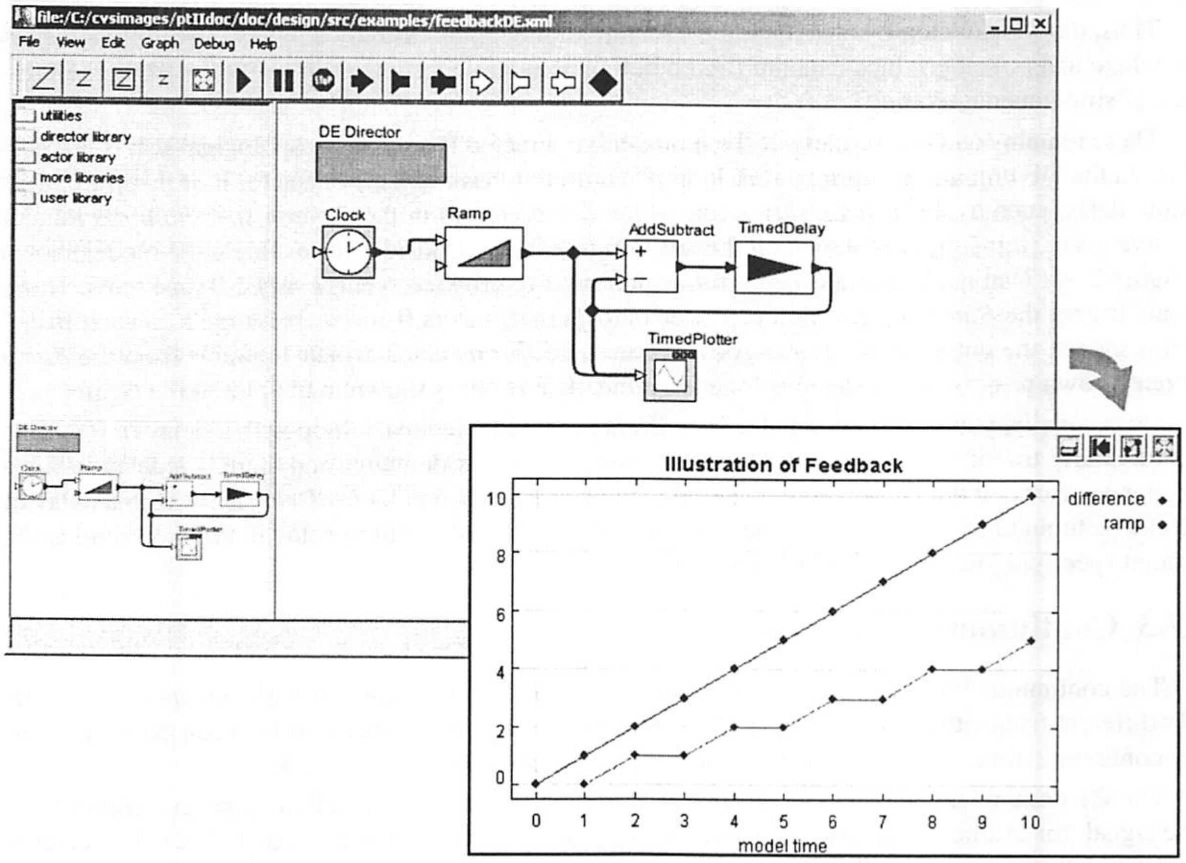
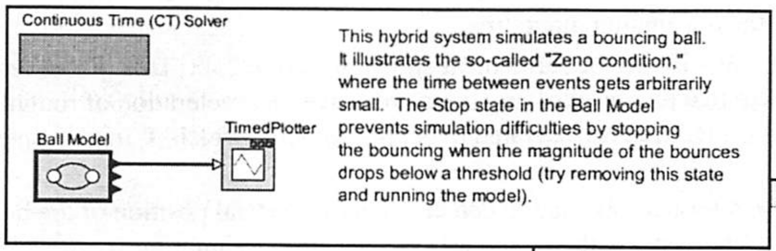


FIGURE 2.37. Discrete-event model with feedback, which requires a delay actor such as *TimedDelay*. Notice the library of domain-specific actors at the left.

left. The circles in figure 2.39 are states, and the arcs between circles are *transitions* between states. A modal model is one that has *modes*, which represent regimes of operation. Each mode in a modal model is represented by a state in a finite-state machine.

The state machine in figure 2.39 has three states, named *init*, *free*, and *stop*. The *init* state is the initial state, which is set as shown in figure 2.40. The *free* state represents the mode of operation where the ball is in free fall, and the *stop* state represents the mode where the ball has stopped bouncing.

At any time during the execution of the model, the modal model is in one of these three states. When the model begins executing, it is in the *init* state. During the time a modal model is in a state, the



This hybrid system simulates a bouncing ball.
It illustrates the so-called "Zeno condition,"
where the time between events gets arbitrarily
small. The Stop state in the Ball Model
prevents simulation difficulties by stopping
the bouncing when the magnitude of the bounces
drops below a threshold (try removing this state
and running the model).

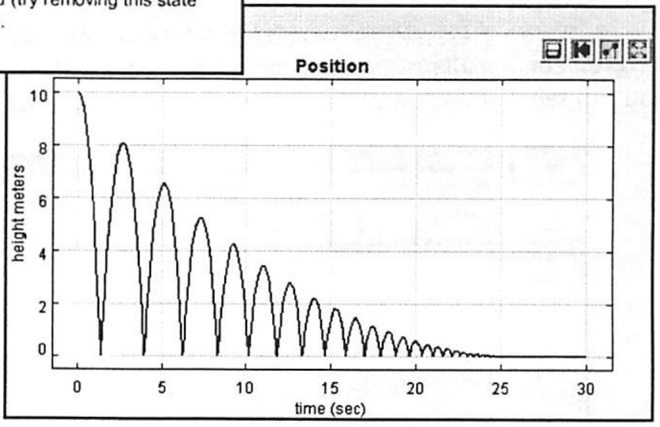
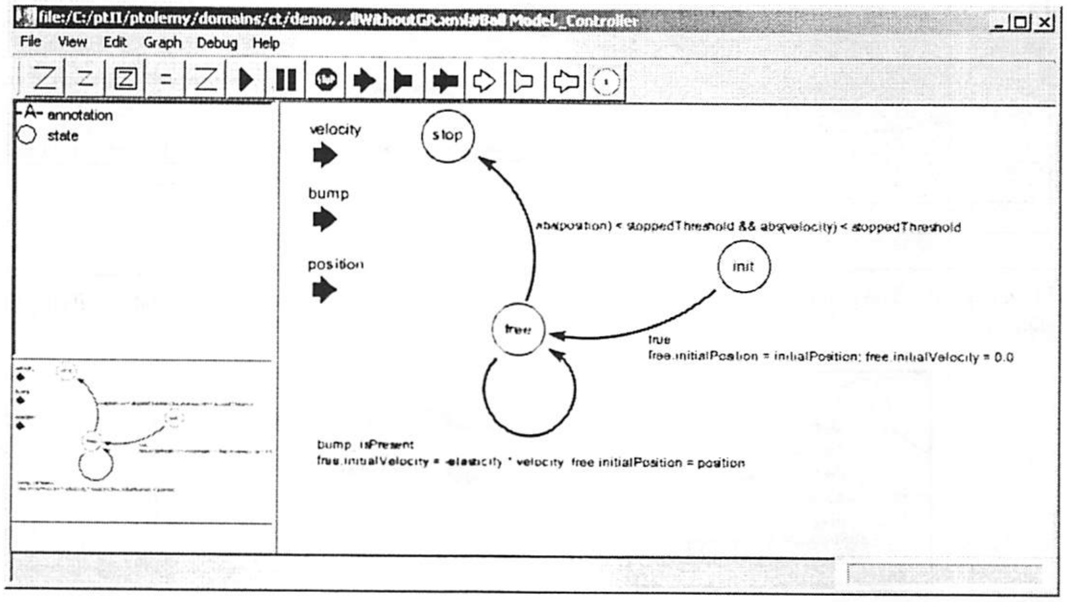


FIGURE 2.38. Top level of the bouncing ball example.


file:/C:/ptII/ptolemy/domains/ct/demo.../WithoutGR.xml#Ball Model_Controller
File View Edit Graph Debug Help
annotation
state
velocity
bump
position
stop
init
free

abs(position) < stoppedThreshold && abs(velocity) < stoppedThreshold

true
free.initialPosition = initialPosition; free.initialVelocity = 0.0

bump_isPresent
free.initialVelocity = -elasticity * velocity; free.initialPosition = position

FIGURE 2.39. Inside the *Ball Model* of figure 2.38.

behavior of the modal model is specified by the *refinement* of the state. The refinement can be examined by looking inside the state. As shown in figure 2.41, the *init* state has no refinement.

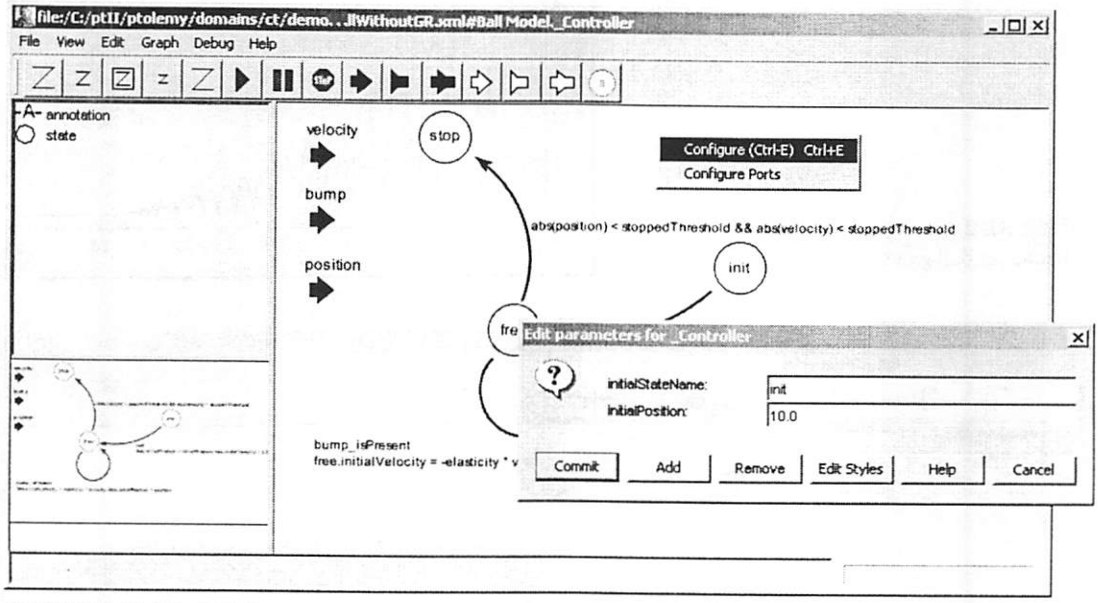
Consider the transition from *init* to *free*. It is labeled as follows:

```
true
free.initialPosition = initialPosition; free.initialVelocity = 0.0
```

The first line is a *guard*, which is predicate that determines when the transition is enabled. In this case, the transition is always enabled, since the predicate has value *true*. Thus, the first thing this model will do is take this transition and change modes to *free*. The second line specifies a sequence of *actions*, which in this case set parameters of the destination mode *free*.

If you look inside the *free* state, you will see the refinement shown in figure 2.42. This model represents the laws of gravity, which state that an object of any mass will have an acceleration of roughly -10 meters/second^2 (roughly). The acceleration is integrated to get the velocity. which is, in turn, integrated to get the vertical position.

In figure 2.42, a *ZeroCrossingDetector* actor is used to detect when the vertical position of the ball is zero. This results in production of an event on the (discrete) output *bump*. Examining figure 2.39, you can see that this event triggers a state transition back to the same *free* state, but where the *initialVe-*

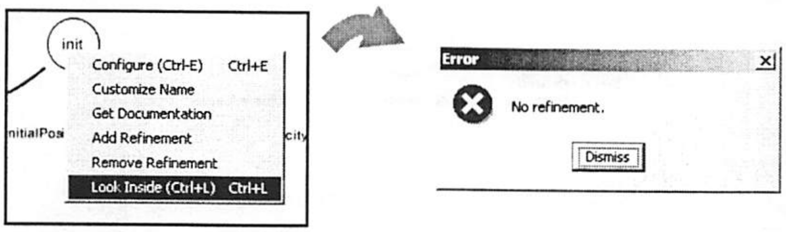

file:/C:/ptII/ptolemy/domains/ct/demo...JIWithoutGR.xml#Ball Model._Controller
File View Edit Graph Debug Help
annotation
state
velocity
bump
position
stop
init
fre
Configure (Ctrl-E) Ctrl+E
Configure Ports
Edit parameters for _Controller
Commit
Add
Remove
Edit Styles
Help
Cancel

abs(position) < stoppedThreshold && abs(velocity) < stoppedThreshold

bump_isPresent
free.initialVelocity = -elasticity * v

initialStateName: init
initialPosition: 10.0

FIGURE 2.40. The initial state of a state machine is set by right clicking on the background and specifying the state name.



No refinement.

FIGURE 2.41. A state may or may not have a refinement, which specified the behavior of the model while the model is in that state. In this case, *init* has no refinement.

locity parameter is changed to reverse the sign and attenuate it by the *elasticity*. This results in the ball bouncing, and losing energy, as shown by the plot in figure 2.38.

As you can see from figure 2.39, 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. This results in the model producing no further output.

2.8.2 Numerical Precision and Zeno Conditions

The bouncing ball model of figures 2.38 and 2.39 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. If you remove the *stop* state from the FSM, and re-run the model, you get the result shown in figure 2.43. The ball, in effect, 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 some fundamental pitfalls with hybrid system modeling. The event detected by the *ZeroCrossingDetector* 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 time at that time. However, when the numbers get small enough, numerical errors take over, and the event is missed.

A related phenomenon is called the Zeno phenomenon. In the case of the bouncing ball, the time between bounces gets smaller as the simulation progresses. Since the simulator is attempting to capture every bounce event with a time step, we could encounter the problem where the number of time steps

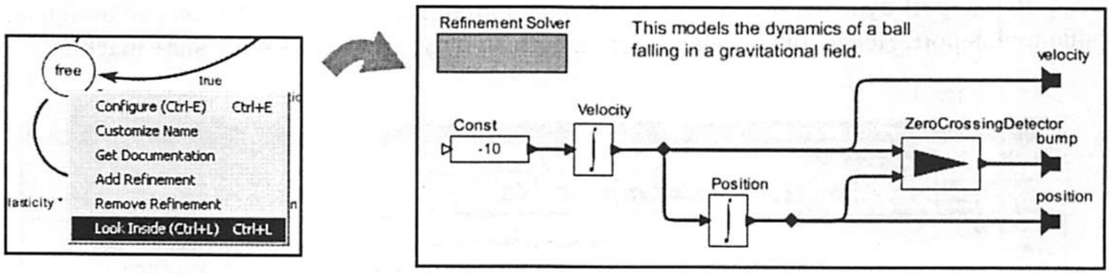

This models the dynamics of a ball falling in a gravitational field.

FIGURE 2.42. The refinement of the *free* state, shown here, is a continuous-model representing the laws of gravity.

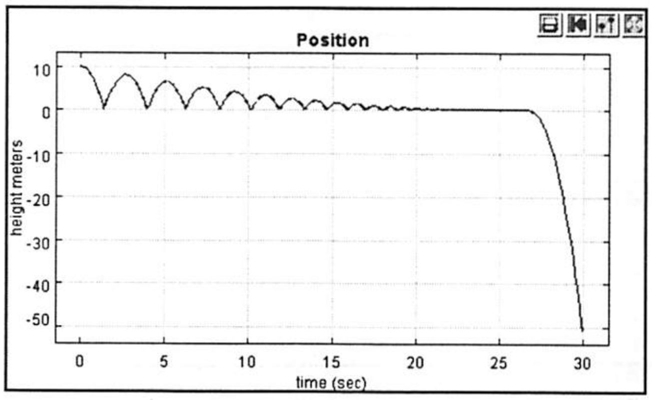


FIGURE 2.43. Result of running the bouncing ball model without the *stop* state.

becomes infinite over a finite time interval. This makes it impossible for time to advance. In fact, in theory, the bouncing ball example exhibits this Zeno phenomenon. However, numerical precision errors take over, since the simulator cannot possibly keep decreasing the magnitude of the time increments.

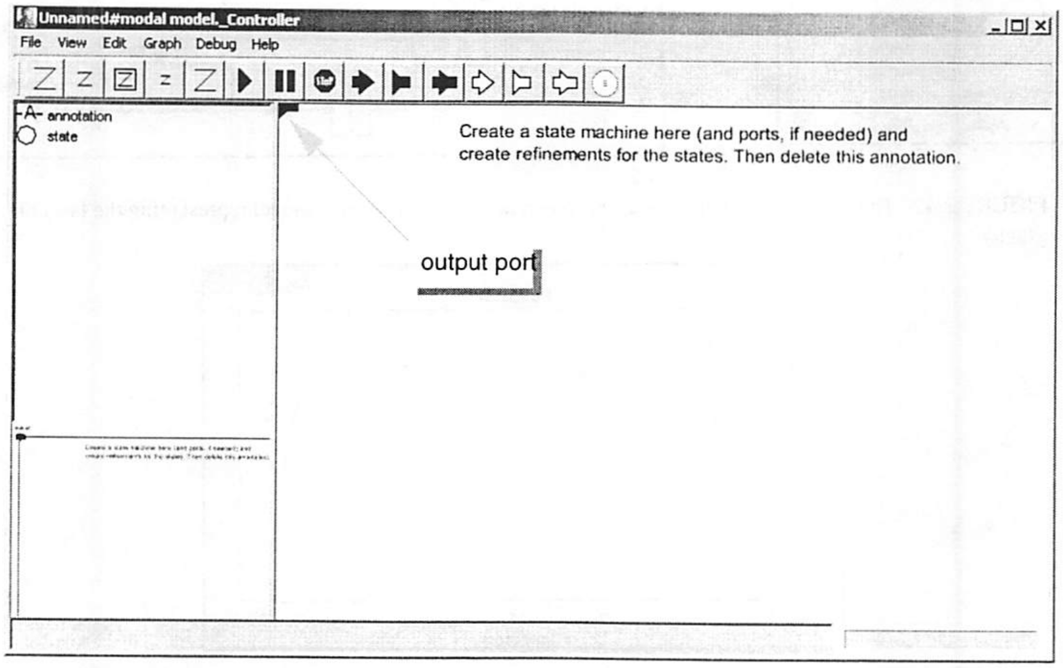
The lesson is that some caution needs to be exercised when relying on the results of a simulation of a hybrid system. Use your judgement.

2.8.3 Constructing Modal Models

A modal model is a component in a larger continuous-time (or other kind of) model. You can create a modal model by dragging one in from the *utilities* library. By default, it has no ports. To make it useful, you will need to add ports. The mechanism for doing that is identical to adding ports to a composite model, and is explained in section 2.4.2. Figure 2.38 shows a top-level continuous-time model with a single modal model that has been renamed *Ball Model*. Three output ports have been added to that modal model, but only the top one is used. It gives the vertical distance of the ball from the surface on which it bounces.

If you create a new modal model by dragging it in from the *utilities* library, create an output port and name it *output*, and then look inside, you will get an FSM editor like that shown in figure 2.44. Note that the output port is (regrettably) located at the upper left, and is only partially visible. The annotation text suggests delete once you no longer need it. You may want to move the port to a more reasonable location (where it is visible).

The output port that you created is in fact indicated in the state machine as being both an output and input port. The reason for this is that guards in the state machine can refer to output values that are produced on this port by refinements. In addition, the output actions of a transition can assign an output value to this port. Hence, the port is, in fact, both an output and input for the state machine.


Unnamed#modal model._Controller
File View Edit Graph Debug Help
annotation
state
output port

Create a state machine here (and ports, if needed) and create refinements for the states. Then delete this annotation.

FIGURE 2.44. Inside of a new modal model that has had a single output port added.

To create a finite-state machine like that in figure 2.39, drag in states (white circles), or click on the state icon in the toolbar. You can rename these states by right clicking on them and selecting "Customize Name". Choose names that are pertinent to your application. In figure 2.39, there is an *init* state for initialization, a *free* state for when the ball is in the air, and a *stop* state for when the ball is no longer bouncing. You must specify the initial state of the FSM by right clicking on the background of the FSM Editor, selecting "Edit Parameters", and specifying an initial state name, as shown in figure 2.40. In that figure, the initial state is named *init.*

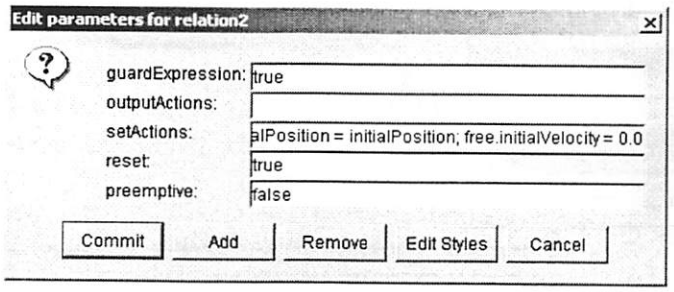
Creating Transitions. To create transitions, you must hold the control button on the keyboard while clicking and dragging from one state to the next (a transition can also go back to the same state). The handles on the transition can be used to customize its curvature and orientation. Double clicking on the transition (or right clicking and selecting "Configure") allows you to configure the transition. The dialog for the transition from *init* to *free* is shown in figure 2.45. In that dialog, we see the following:

- The guard expression is *true*, so this transition is always enabled. The transition will be taken as soon as the model begins executing. A guard expression can be any boolean-valued expression that depends on the inputs, parameters, or even the outputs of any refinement of the current state (see below). Thus, this transition is used to initialize the model.
- The output actions are empty, meaning that when this transition is taken, no output is specified. This parameter can have a list of assignments of values to output ports, separated by semicolons. Those values will be assigned to output ports when the transition is taken.
- The set actions field contains the following statements:

  ```
  free.initialPosition = initialPosition; free.initialVelocity = 0.0
  ```

  The "free" in these expressions refers to the mode refinement in the *free* state. Thus, *free.initialPosition* is a parameter of that mode refinement. Here, its value is assigned to the value of the parameter *initialPosition*. The parameter *free.initialVelocity* is set to zero.
- The *reset* parameter is set to *true*, meaning that the destination mode refinement will be initialized when the transition is taken.
- The *preemptive* parameter is set to *false*. In this case, it makes no difference, since the *init* state has no refinement. Normally, if a transition out of a state is enabled and *preemptive* is *true*, then the transition will be taken without first executing the refinement. Thus, the refinement will not affect the outputs of the modal model.

A state may have several outgoing transitions. However, it is up to the model builder to ensure that at


Edit parameters for relation2

guardExpression:	true
outputActions:	
setActions:	alPosition = initialPosition; free.initialVelocity = 0.0
reset:	true
preemptive:	false

FIGURE 2.45. Transition dialog for the transition from *init* to *free* in figure 2.39.

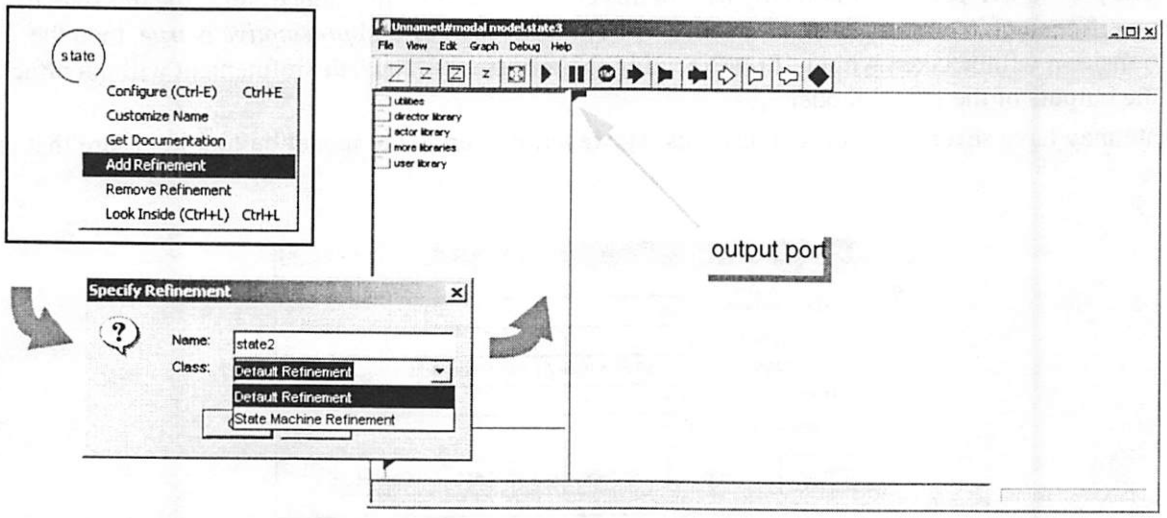
no time does more than one guard on these transitions evaluate to true. In other words, Ptolemy II does not allow nondeterministic state machines, and will throw an exception if it encounters one.

Creating Refinements. Both states and transitions can have *refinements.* To create a refinement, right click on the state or transition, and select "Add Refinement." You will see a dialog like that in figure 2.46. As shown in the figure, you will be offered the alternatives of a "Default Refinement" or a "State Machine Refinement." The first of these provides a block diagram model as the refinement. The second provides another finite state machine as the refinement. In the former case (the default), a blank refinement model will open, as shown in the figure. As before, the output port will appear in an inconvenient location. You will almost certainly want to move it to a more convenient location. You will have to create a director in the refinement. The modal model will not operate without a director in the refinement.

You can also create refinements for transitions, but these have somewhat different behavior. They will execute exactly once when the transition is taken. For this reason, only certain directors make sense in such refinements. The most commonly useful is the SDF director. Such refinements are typically used to perform arithmetic computations that are too elaborate to be conveniently specified as an action on the transition.

Once you have created a refinement, you can look inside a state or transition. For the bouncing ball example, the refinement of the *free* state is shown in figure 2.42. This model exhibits certain key properties of refinements:

- Refinements must contain directors. In this case, the CTEmbeddedDirector is used. When a continuous-time model is used inside a mode, this director must be used instead of the default CTDirector (see the CT domain documentation for details).
- The refinement has the same ports as the modal model, and can read input value and specify output values. When the state machine is in the state of which this is the refinement, this model will be executed to read the inputs and produce the outputs.



Name: state2
Class: Default Refinement

FIGURE 2.46. Adding a refinement to a state.

2.8.4 Execution Semantics

The behavior of a refinement is simple. When the modal model is executed, the following sequence of events occurs:

- For any transitions out of the current state for which *preemptive* is *true*, the guard is evaluated. If exactly one such guard evaluates to *true*, then that transition is chosen. The *output actions* of the transition are executed, and the *refinements* of the transition (if any) are executed, followed by the *set actions*.
- If no preemptive transition evaluated to true, then the refinement of the current state, if there is one, is evaluated at the current time step.
- Once the refinement has been evaluated (and it has possibly updated its output values), the guard expressions on all the outgoing transitions of the current state are evaluated. If none is true, the execution is complete. If one is true, then that transition is taken. If more than one is true, then an exception is thrown (the state machine is nondeterministic). What it means for the transition to be "taken" is that its *output actions* are executed, its *refinements* (if any) are executed, and its *set actions* are executed.
- If *reset* is true on a transition that is taken, then the refinement of the destination mode (if there is one) is initialized.

There is a subtle distinction between the *output actions* and the *set actions*. The intent of these two fields on the transition is that *output actions* are used to define the values of output ports, while *set actions* are used to define state variables in the refinements of the destination modes. The reason that these two actions are separated is that while solving a continuous-time system of equations, the solver may speculatively execute models at certain time steps before it is sure what the next time step will be. The *output actions* make no permanent changes to the state of the system, and hence can be executed during this speculative phase. The *set actions*, however, make permanent changes to the state variables of the destination refinements, and hence are not executed during the speculative phase.

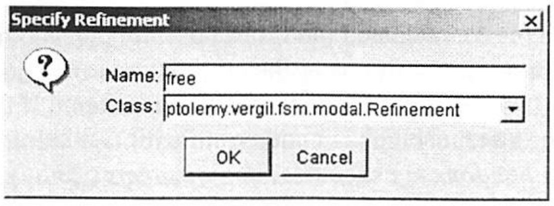
2.9 Using the Plotter

Several of the plots shown above have flaws that can be fixed using the features of the plotter. For instance, the plot shown in figure 2.33 has the default (uninformative) title, the axes are not labeled, and the horizontal axis ranges from 0 to 255[1], because in one iteration, the *Spectrum* actor produces 256 output tokens. These outputs represent frequency bins that range between $-\pi$ and π radians per second.

1. **Hint:** Notice the "x10^2" at the bottom right, which indicates that the label "2.5" stands for "250".

The *SequencePlotter* actor has some pertinent parameters, shown in figure 2.47. 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 2.48.

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 2.49, filled in with values that result in the plot shown in figure 2.50. Most of these are self-explanatory, but the following pointers may be useful:



Name: free
Class: ptolemy.vergil.fsm.modal.Refinement

FIGURE 2.47. Dialog for creating a refinement of a state.

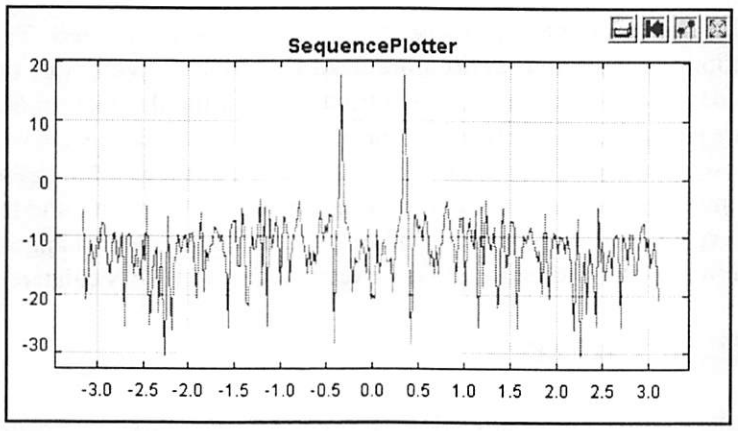
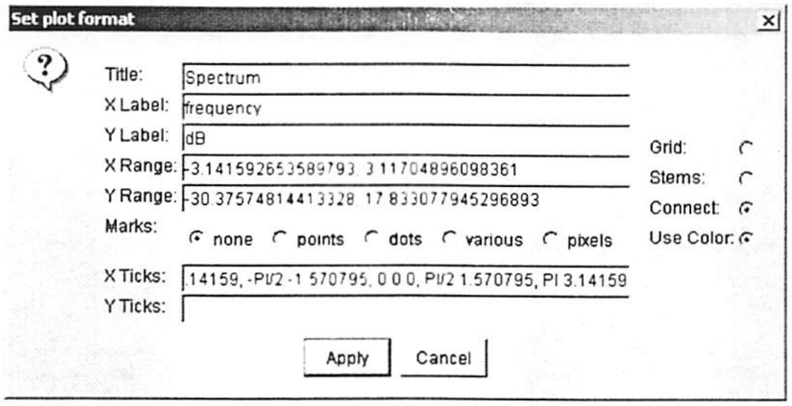


FIGURE 2.48. Better labeled plot, where the horizontal axis now properly represents the frequency values.



Title: Spectrum
X Label: frequency
Y Label: dB
X Range: -3.141592653589793, 3.11704896098361
Y Range: -30.37574814413328, 17.833077945296893
Marks: [✓ none] points dots various pixels
X Ticks: .14159, -PI/2 -1.570795, 0 0.0, PI/2 1.570795, PI 3.14159
Y Ticks:
Grid:
Stems:
[✓ Connect:]
[✓ Use Color:]

FIGURE 2.49. Format control window for a plot.

- 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"
- 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.

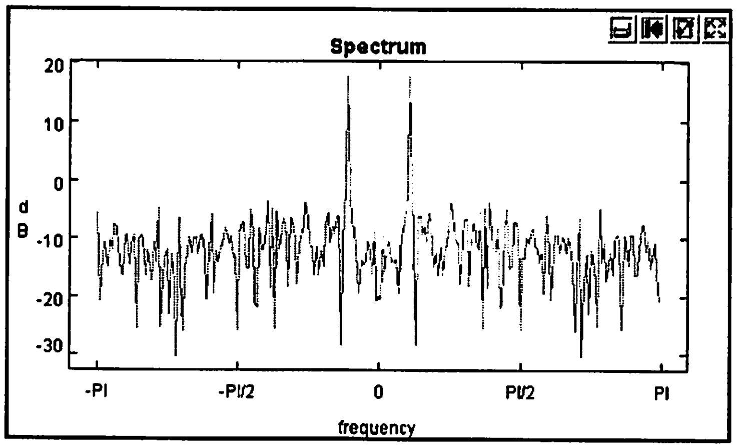


FIGURE 2.50. Still better labeled plot.

3

Expressions

Authors: *Edward A. Lee*
Xiaojun Liu
Steve Neuendorffer
Neil Smyth
Yuhong Xiong

3.1 Introduction

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(2π (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.

3.1.1 Expression Evaluator

Vergil provides an interactive *expression evaluator*, which is accessed through the File:New menu. This operates like an interactive command shell, and is shown in figure 3.1. 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.

3.2 Simple Arithmetic Expressions

3.2.1 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, MaxInt, MinInt, MaxLong, MinLong, MaxDouble, MinDouble. For example,

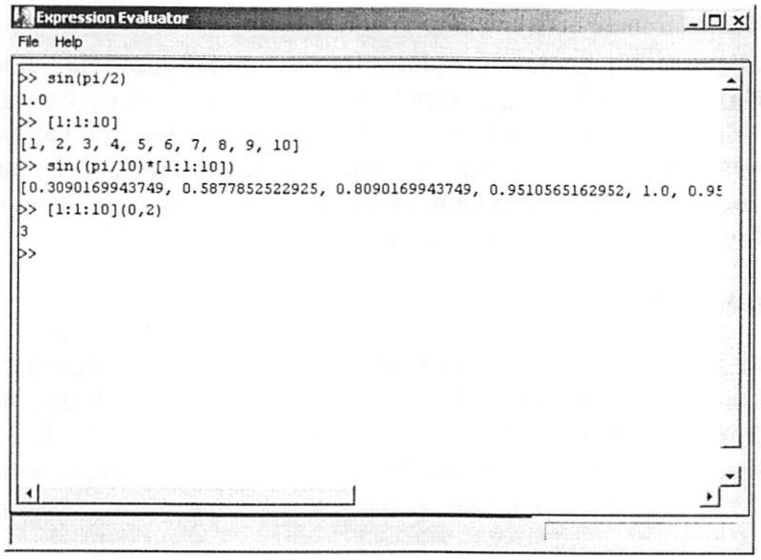
```
PI/2.0
```

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 the square root of –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 without decimal points followed by the character "l" (el) or "L" are of type *long*. 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 not quite the same as the Java byte, which has a value between -128 and 127.

Numbers of type *int*, *long*, 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.

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



```
>> sin(pi/2)
1.0
>> [1:1:10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>> sin((pi/10)*[1:1:10])
[0.3090169943749, 0.5877852522925, 0.8090169943749, 0.9510565162952, 1.0, 0.95
>> [1:1:10](0,2)
3
>>
```

FIGURE 3.1. Expression evaluator, which is accessed through the File:New menu.

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 quotes, "...", is interpreted as a string constant. The following built-in string-valued constants are defined:

TABLE 1: String-valued constants defined in the expression language.

Variable name	Meaning	Property name	Example under Windows
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

The value of these variables is the value of the Java virtual machine property, such as *user.home*. The properties *user.dir* and *user.home* are standard in Java. Their values are platform dependent; see the documentation for the java.lang.System.getProperties() method for details. 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. Vergil will display all the Java properties if you invoke JVM Properties in the View 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.

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 something like:

```
{CWD="C:\ptII\ptolemy\data\expr", E=2.718281828459, HOME="C:\Documents
and Settings\eal", Infinity=Infinity, MaxDouble=1.7976931348623E308,
MaxInt=2147483647, MaxLong=9223372036854775807L,
MaxUnsignedByte=255ub, MinDouble=4.9E-324, MinInt=-2147483648,
MinLong=-9223372036854775808L, MinUnsignedByte=0ub, NaN=NaN,
NegativeInfinity=-Infinity, PI=3.1415926535898, PTII="c:\ptII",
PositiveInfinity=Infinity, boolean=false, complex=0.0 + 0.0i,
double=0.0, e=2.718281828459, false=false, fixedpoint=fix(0.0,2,1),
general=present, i=0.0 + 1.0i, int=0, j=0.0 + 1.0i, long=0L, matrix=[],
object=object(null),pi=3.1415926535898, scalar=present, string="",
true=true, unknown=present, unsignedByte=0ub}
```

3.2.2 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" is 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 "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.

3.2.3 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 an *int* or an *unsignedByte*.

The *unsignedByte*, *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 if 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 5 on page 96). 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 page 90). 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, *int* 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.
```

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 integer types and *fixedpoint*, overflow results in wraparound. For instance, while the value of `MaxInt` is 2147483647, 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*, *int* and *long* (but not *fixedpoint*, *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.

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*, *int*, and *long*.

3.2.4 Comments

In expressions, anything inside `/*...*/` is ignored, so you can insert comments.

3.3 Uses of Expressions

3.3.1 Parameters

The values of most parameters of actors can be given as expressions[1]. 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 3.9. Adding parameters to actors is straightforward, as explained in the previous chapter.

3.3.2 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

1. 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.

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.

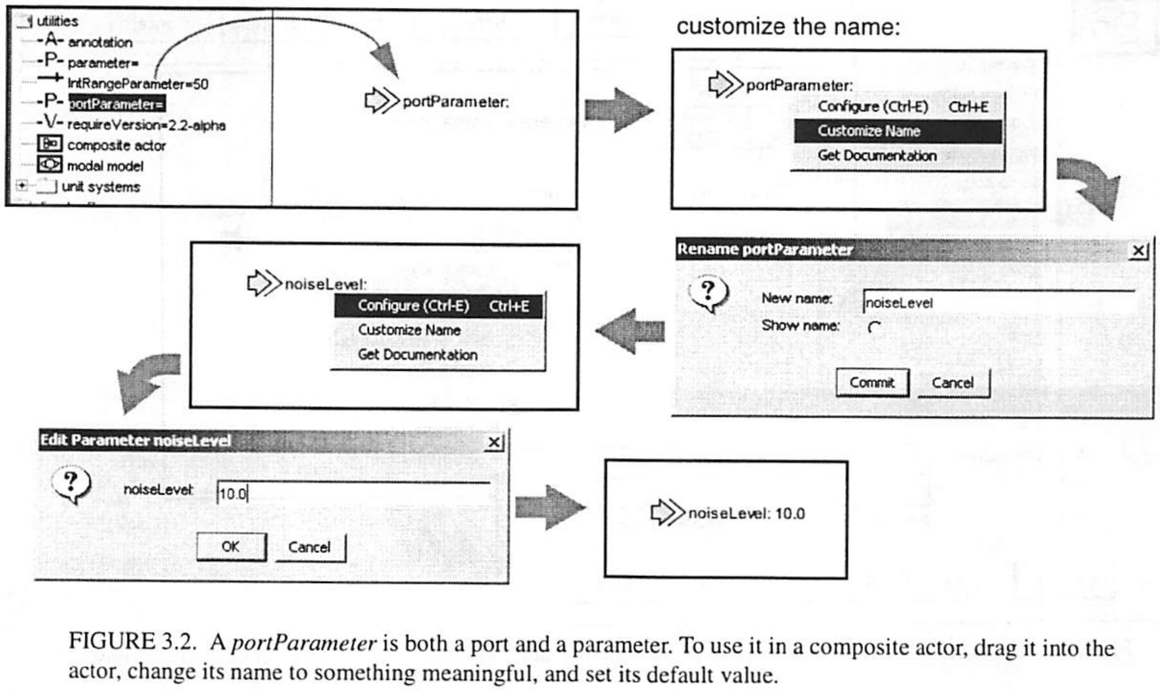
A PortParameter might be contained by an atomic actor or a composite actor. To put one in a composite actor, drag it into a model from the *utilities* library, as shown in figure 3.2. The resulting icon is actually a combination of two icons, one representing the port, and the other representing the parameter. These can be moved separately, but doing so might create confusion, so we recommend selecting both by clicking and dragging over the pair and moving both together.

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 Name," as shown in figure 3.2. In the figure, the name is set to "noiseLevel." Then set the default value by either double clicking or selecting "Configure." 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* library in Vergil. It is shown in figure 3.3. 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.

3.3.3 Expression Actor

The *Expression* actor is a particularly useful actor found in the *math* library. By default, it has one output and no inputs, as shown in Figure 3.4(a). The first step in using it is to add ports, as shown in (b) and (c), resulting in a new icon as shown in (d). Note: In (c) when you click on Add, you will be prompted for a Name (pick one) and a Class. Leave the Class entry blank and click OK. You then specify an expression using the port names, as shown in (e), resulting in the icon shown in (f).



New name: noiseLevel

noiseLevel: 10.0

FIGURE 3.2. 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.

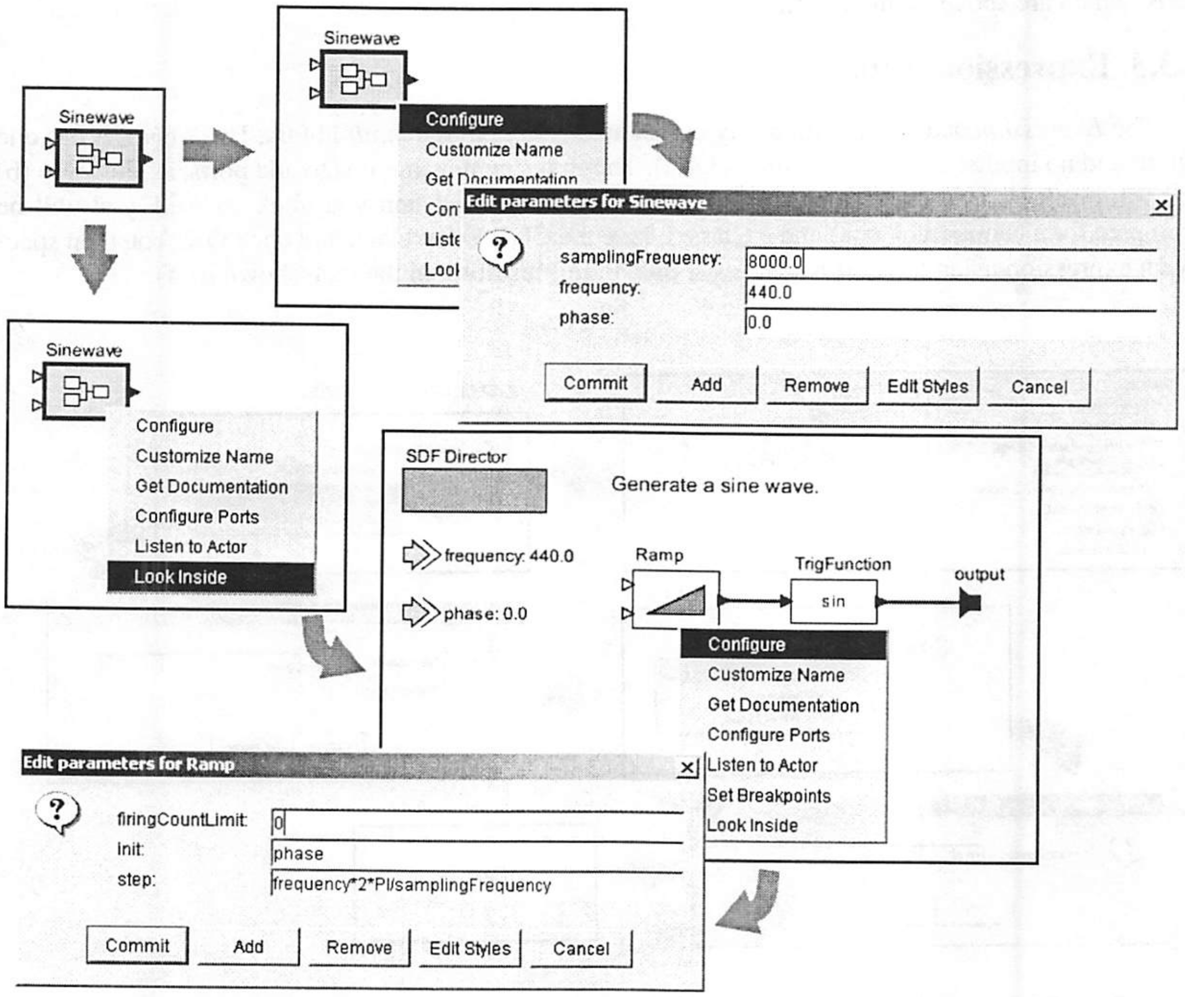
3.3.4 State Machines

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.

3.4 Composite Data Types

3.4.1 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 *string*. The types are denoted "`{int}`" and "`{string}`" respectively. An array is an ordered list of tokens of any type, with the only constraint being that the elements all have


Edit parameters for Sinewave

samplingFrequency: 8000.0
frequency: 440.0
phase: 0.0

Generate a sine wave.

frequency: 440.0
phase: 0.0

Edit parameters for Ramp

firingCountLimit: 0
init: phase
step: frequency*2*PI/samplingFrequency

FIGURE 3.3. Sinewave actor, showing its port parameters, and their use at the lower level of the hierarchy.

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}
```

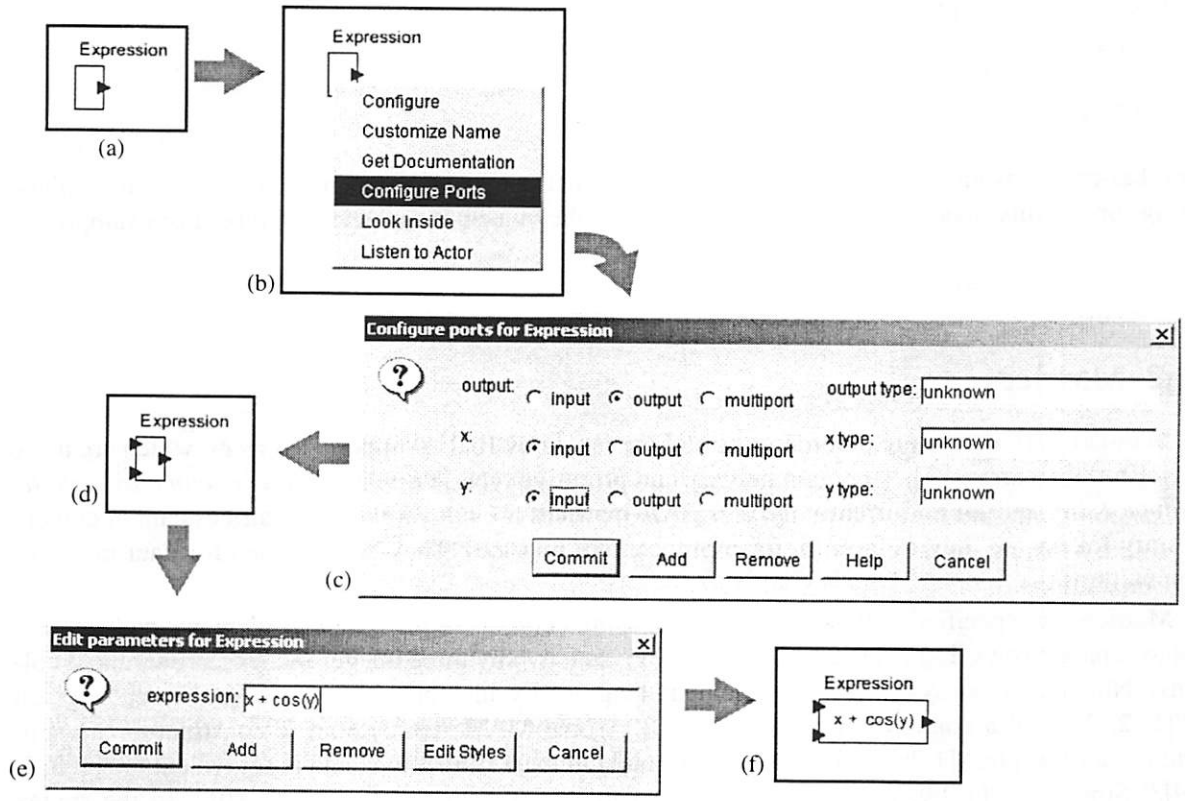
has value

```
{1.0, 2.3}
```

Its type is `{double}`. 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
```

which yields 2.3. Note that indexing begins at 0. Of course, if *name* is the name of a variable in scope



output: input [✓ output] multiport output type: unknown
x: [✓ input] output multiport x type: unknown
y: [✓ input] output multiport y type: unknown

expression: x + cos(y)

FIGURE 3.4. Illustration of the *Expression* actor.

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}
```

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 5 on page 96). 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
```

3.4.2 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 *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).

Reference to elements of matrices have the form "*matrix*(*n*, *m*)" or "*name*(*n*, *m*)" where *name* 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, as seen in the expression evaluator (see figure 3.1),

```
>> [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 6 on page 98). 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 6 on page 98). A matrix can be raised to an *int* 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 5 on page 96). 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
>>
```

3.4.3 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 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).

Fields may be accessed using the period operator. For example,

```
{a=1,b=2}.a
```

yields 1. You can optionally write this as if it were a method call:

```
{a=1,b=2}.a()
```

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. Hence

```
>> {a=1, b=2}=={b=2, a=1}
true
```

Moreover, 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}
```

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
>>
```

3.5 Invoking Methods

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 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()
```

3.6 Defining Functions

The expression language supports definition of functions. 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
>>
```

Functions can be passed as arguments to certain "higher-order functions" that have been defined (see table Table 9 on page 102). 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}
```

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. Consider the model shown in figure 3.5. 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.

A more elaborate use is shown in figure 3.6. 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 adds the output of the Ramp to the square of the output of the Ramp.

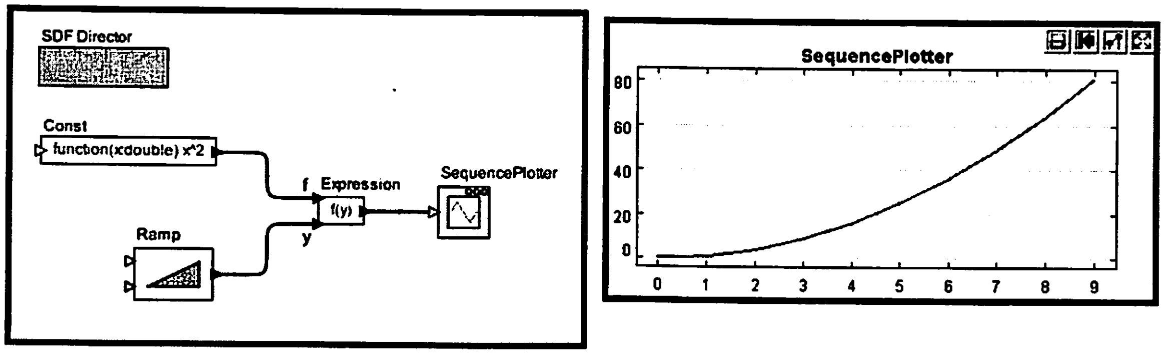


FIGURE 3.5. Example of a function being passed from one actor to another.

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:gen-
eral, 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.

3.7 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 2 on page 88, which together provide a rich set[1]. The functions currently available are shown in the tables in the appendix, which also show the argument types and return types.

In most cases, a function that operates on scalar arguments can also operate on arrays and matrices.

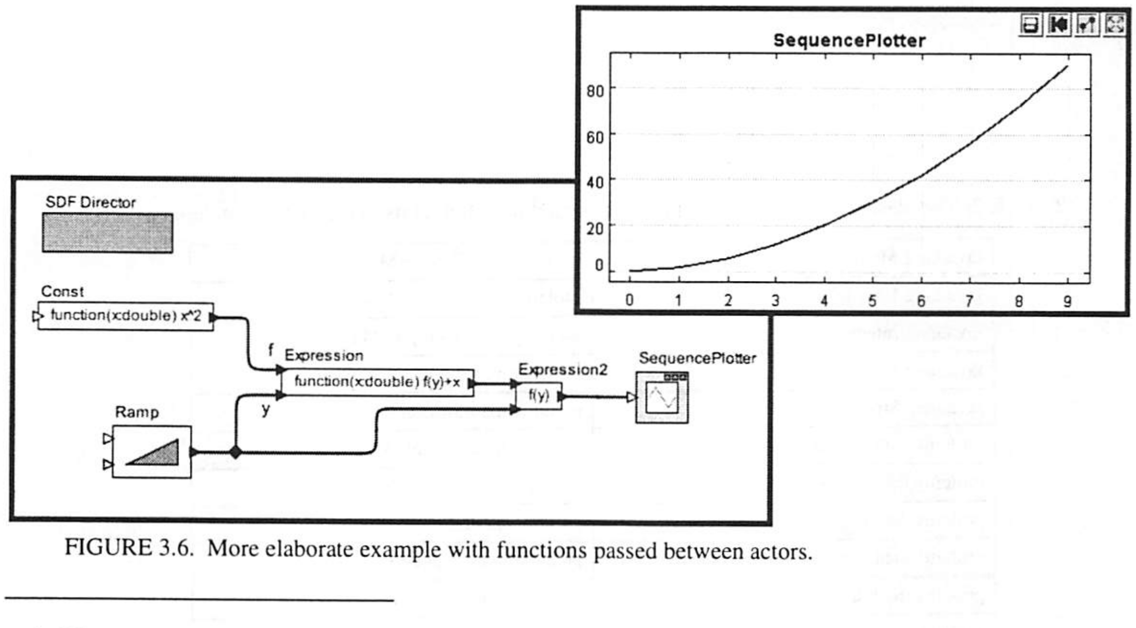


FIGURE 3.6. More elaborate example with functions passed between actors.

1. Moreover, the set of available 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).

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* 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 4 on page 95 shows trigonometric functions. Note that these operate on *double* or *complex*, and hence on *int* and *unsignedByte*, which can be losslessly converted to *double*. The result will always be double. For example,

```
    >> cos(0)
    1.0
```

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 5 on page 96 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* 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,

TABLE 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	

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 6 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 matricesTable 7 shows utility functions for evaluating expressions given as strings or representing numbers as strings. Of these, the eval() function is the most flexible (see page 89).

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 property user.dir), the 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 functions random() and gaussian() shown in Table 5 on page 96 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 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.

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 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

$$X_k = s_k \sum_{n=0}^{N-1} x_n \cos\left((2n+1)k\frac{\pi}{2D}\right) \quad (4)$$

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 3: Normalization options for the DCT function

Name	Third argument	Normalization
Normalized	0	$s_k = \begin{cases} 1/\sqrt{2}; & k=0 \\ 1; & \text{otherwise} \end{cases}$
Unnormalized	1	$s_k = 1$
Orthonormal	2	$s_k = \begin{cases} 1/\sqrt{D}; & k=0 \\ \sqrt{2/D}; & \text{otherwise} \end{cases}$

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

$$x_n = \sum_{k=0}^{N-1} s_k X_k \cos\left((2n+1)k\frac{\pi}{2D}\right). \quad (5)$$

3.8 Fixed Point Numbers

Ptolemy II includes a preliminary 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.

3.9 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 3.7. This unit system is called BasicUnits; the units it defines can be examined by double clicking on its icon, or by invoking Configure, as shown in figure 3.8. 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".

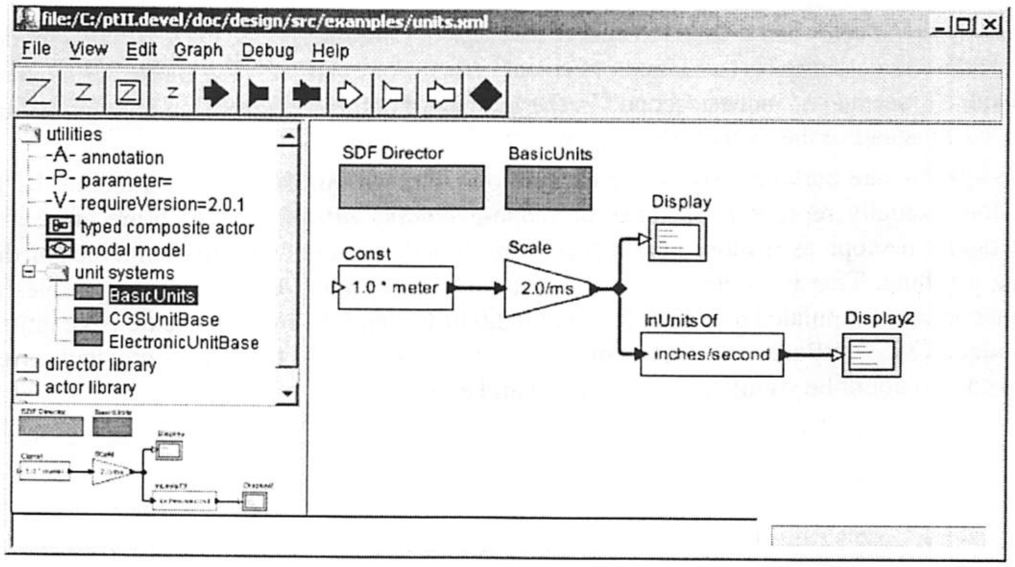
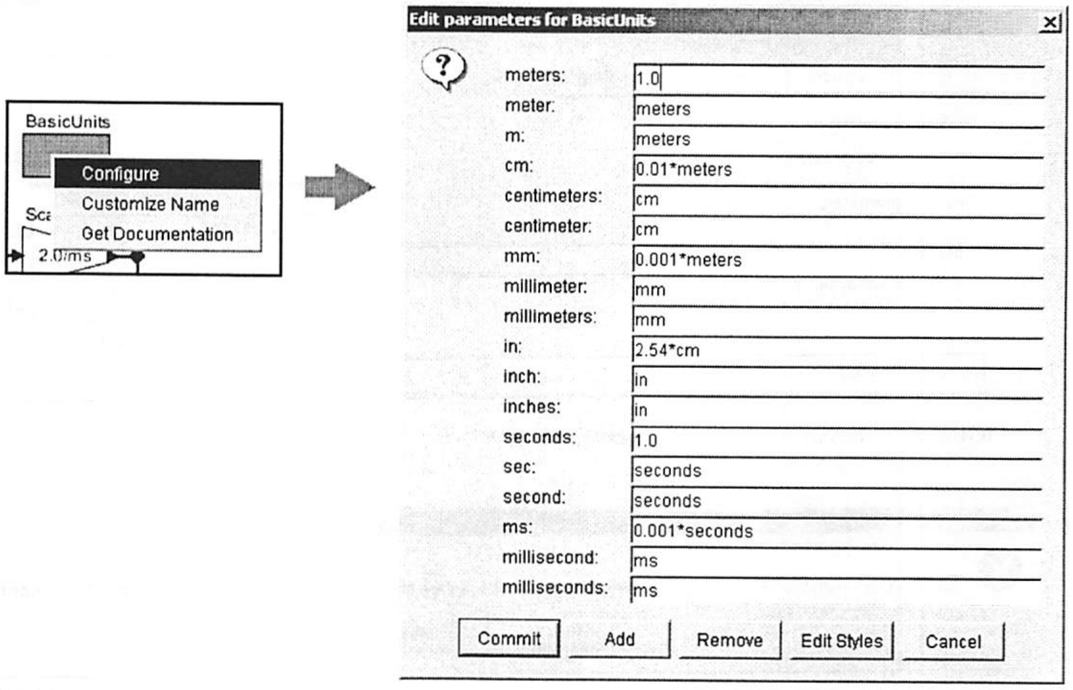


FIGURE 3.7. Example of a model that includes a unit system.



meters:	1.0
meter:	meters
m:	meters
cm:	0.01*meters
centimeters:	cm
centimeter:	cm
mm:	0.001*meters
millimeter:	mm
millimeters:	mm
in:	2.54*cm
inch:	in
inches:	in
seconds:	1.0
sec:	seconds
second:	seconds
ms:	0.001*seconds
millisecond:	ms
milliseconds:	ms

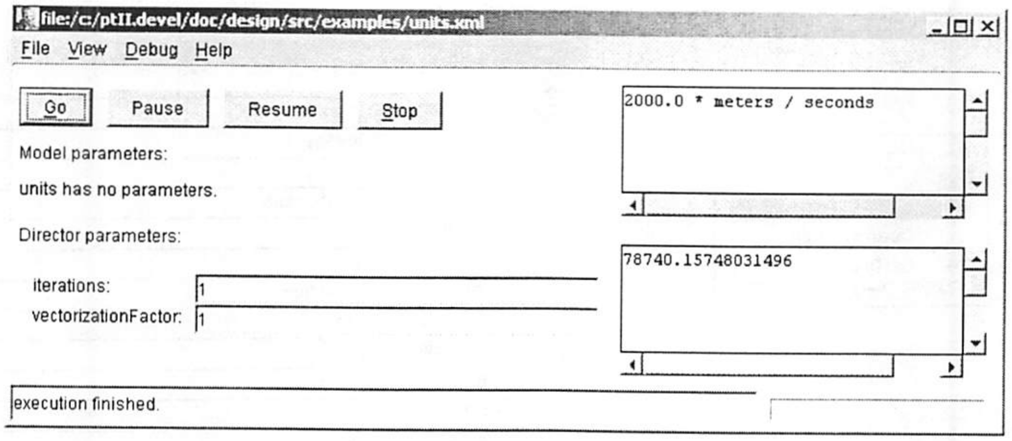
FIGURE 3.8. Units defined in a units system can be examined by invoking Configure on its icon.

In the example in figure 3.7, 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 3.9, top display. The canonical units for length are meters, and for time are seconds.

In figure 3.7, we also take the result and feed it to the *InUnitsOf* actor, which performs 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 3.7 we had entered "seconds/inch" instead of "inches/second" in the InUnitsOf actor, we would have gotten the exception in figure 3.10 instead of the execution in figure 3.9.

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. In version 2.0 of Ptolemy II, 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.



Model parameters:

units has no parameters.

Director parameters:

iterations: 1

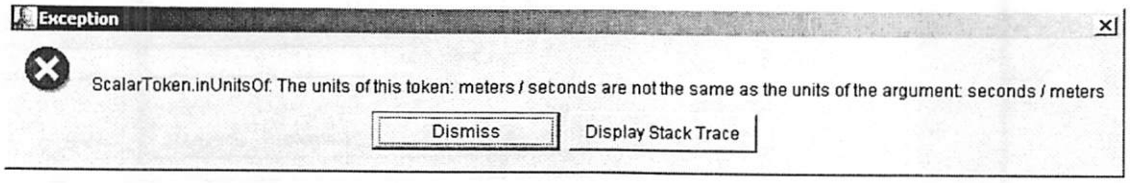
vectorizationFactor: 1

```
2000.0 * meters / seconds
```

```
78740.15748031496
```

execution finished.

FIGURE 3.9. Result of running the model in figure 3.7.



ScalarToken.inUnitsOf: The units of this token: meters / seconds are not the same as the units of the argument: seconds / meters

FIGURE 3.10. Example of an exception resulting from a units mismatch.

Appendix A. Tables of Functions

In this appendix, we tabulate the functions available in the expression language. Further explanation of many of these functions is given in section section 3.7 above.

A.1 Trigonometric Functions

TABLE 4: Trigonometric functions.

function	argument type(s)	return type	description
acos	*double* in the range [-1.0, 1.0] or *complex*	*double* in the range [0.0, pi] or NaN if out of range or *complex*	arc cosine *complex* case: $\mathrm{acos}(z) = -i\log(z + i\,\mathrm{sqrt}(1 - z^2))$
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: $\mathrm{asin}(z) = -i\log(iz + \mathrm{sqrt}(1 - z^2))$
atan	*double* or *complex*	*double* in the range [-pi/2, pi/2] or *complex*	arc tangent *complex* case: $\mathrm{atan}(z) = -\frac{i}{2}\log\left(\frac{i-z}{i+z}\right)$
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: $\mathrm{acosh}(z) = \log(z + \mathrm{sqrt}(z^2 - 1))$
asinh	*double* or *complex*	*double* or *complex*	hyperbolic arc sine *complex* case: $\mathrm{asinh}(z) = \log(z + \mathrm{sqrt}(z^2 + 1))$
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: $\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$

A.2 Basic Mathematical Functions

TABLE 5: Basic mathematical functions

function	argument type(s)	return type	description
abs	*double* or *int* or *long* or *complex*	*double* or *int* or *long* (*complex* returns double)	absolute value complex case: $\text{abs}(a + ib) = \lvert z\rvert = \sqrt{a^2 + b^2}$
angle	*complex*	*double* in the range [-pi, pi]	angle or argument of the complex number: $\angle z$
ceil	*double*	*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, infinity] 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 {*double*} or [*double*]	one or more Gaussian random variables with the specified mean and standard deviation (see page 89).
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: $\log(z) = \log(\text{abs}(z) + i\text{angle}(z))$
log10	*double*	*double*	log base 10
log2	*double*	*double*	log base 2
max	*double, double* or *int, int* or *long, long* or *unsignedByte, unsignedByte* or {*double*} or {*int*} or {*long*} or {*unsignedByte*}	*double* or *int* or *long* or *unsignedByte*	maximum
min	*double, double* or *int, int* or *long, long* or *unsignedByte, unsignedByte* or {*double*} or {*int*} or {*long*} or {*unsignedByte*}	*double* or *int* or *long* or *unsignedByte*	minimum

TABLE 5: Basic mathematical functions

function	argument type(s)	return type	description
neighborhood	*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.
pow	*double, double* or *complex, complex*	double or complex	first argument to the power of the second
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 page 89)
real	*complex*	*double*	real part
remainder	*double, double*	*double*	remainder after division, according to the IEEE 754 floating-point standard (see page 90).
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: $\mathrm{sqrt}(z) = \sqrt{\lvert z\rvert}\left(\cos\left(\frac{\angle z}{2}\right) + i\sin\left(\frac{\angle z}{2}\right)\right)$
toDegrees	*double*	*double*	convert radians to degrees
toRadians	*double*	*double*	convert degrees to radians

3.10 Matrix, Array, and Record Functions.

TABLE 6: Functions that take or return matrices, arrays, or records.

function	argument type(s)	return type	description
arrayToMatrix	{*type*}, int, int	[*type*]	Create a matrix from the specified array with the specified number of rows and columns
conjugateTranspose	[*complex*]	[*complex*]	Return the conjugate transpose of the specified matrix.
createSequence	*type*, *type*, int	{*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* or	[*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	{*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
hilbert	*int*	[*double*]	Return a square Hilbert matrix, where $A_{ij} = 1/(i+j+1)$. A Hilbert matrix is nearly, but not quite singular.
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*]	{*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.
orthogonalizeColumns	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthogonal columns.
orthogonalizeRows	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthogonal rows.
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*	{*type*}	Create an array by repeating the specified token the specified number of times.
sum	{*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.

TABLE 6: Functions that take or return matrices, arrays, or records.

function	argument type(s)	return type	description
transpose	[*type*]	[*type*]	Return the transpose of the specified matrix.
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.

A.3 Functions for Evaluating Expressions

TABLE 7: Utility functions for evaluating expressions

function	argument type(s)	return type	description
eval	*string*	any type	evaluate the specified expression (see page 89).
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* or *long*, *int*	*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 page 89).

A.4 Signal Processing Functions

TABLE 8: Functions performing signal processing operations

function	argument type(s)	return type	description
convolve	*{double}*, *{double}* or *{complex}*, *{complex}*	*{double}* or *{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	*{double}* or *{double}*, *int* or *{double}*, *int*, *int*	*{double}*	Return the discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see page 90).
downsample	*{double}*, *int* or *{double}*, *int*, *int*	*{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	*{double}* or *{complex}* or *{double}*, *int* *{complex}*, *int*	*{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.
generateBartlettWindow	*int*	*{double}*	Return a 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: $w(n) = \begin{cases} 2\frac{n}{M}; & \text{if } 0 \le n \le \frac{M}{2} \\ 2-2\frac{n}{M}; & \text{if } \frac{M}{2} \le n \le M \end{cases}$
generateBlackmanWindow	*int*	*{double}*	Return a Blackman window with the specified length. For length $M + 1$, the formula is: $w(n) = 0.42 + 0.5\cos(2\pi n/M) + 0.08\cos(4\pi n/M)$
generateBlackmanHarrisWindow	*int*	*{double}*	Return a Blackman-Harris window with the specified length. For length $M + 1$, the formula is: $w(n) = 0.35875 + 0.48829\cos(2\pi n/M) + 0.14128\cos(4\pi n/M) + 0.01168\cos(6\pi n/M)$
generateGaussianCurve	*double, double, int*	*{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).
generateHammingWindow	*int*	*{double}*	Return a Hamming window with the specified length. For length $M + 1$, the formula is: $w(n) = 0.54 - 0.46\cos(2\pi n/M)$
generateHanningWindow	*int*	*{double}*	Return a Hanning window with the specified length. For length $M + 1$, the formula is: $w(n) = 0.5 - 0.5\cos(2\pi n/M)$

TABLE 8: Functions performing signal processing operations

function	argument type(s)	return type	description
generatePolynomialCurve	*{double}, double, double, int*	*{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.
generateRaisedCosinePulse	*double, double, int*	*{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: $$h(t) = \frac{\sin(\pi t/T)}{\pi t/T} \times \frac{\cos(x\pi t/T)}{1-(2xt/T)^2},$$ 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 excessBandwidth of 0.0, this pulse is a sinc pulse.
generateRectangularWindow	*int*	*{double}*	Return an array filled with 1.0 of the specified length. This is a rectangular window.
IDCT	*{double}* or *{double}, int* or *{double}, int, int*	*{double}*	Return the inverse discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see page 90).
IFFT	*{double}* or *{complex}* or *{double}, int* *{complex}, int*	*{complex}*	Return the 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.
nextPowerOfTwo	*double*	*int*	Return the next power of two larger than or equal to the argument.
poleZeroToFrequency	*{complex}, {complex}, complex, int*	*{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.

TABLE 8: Functions performing signal processing operations

function	argument type(s)	return type	description
toDecibels	*double*	*double*	Return $20 \times \log_{10}(z)$, where z is the argument.
unwrap	*{double}*	*{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	*{double}, int*	*{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 $n > 0$.

A.5 I/O Functions and Other Miscellaneous Functions

TABLE 9: Miscellaneous functions.

function	argument type(s)	return type	description
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.
freeMemory	none	*long*	Return the approximate number of bytes available for future memory allocation.
iterate	*function, int, type*	*{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, {type}*	*{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.
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.
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.

PTOLEMY II

HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

Edited by:
Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Steve Neuendorffer, Yuhong Xiong, Haiyang Zheng

VOLUME 1: INTRODUCTION TO PTOLEMY II

Authors:
Shuvra S. Bhattacharyya
Elaine Cheong
John Davis, II
Mudit Goel
Bart Kienhuis
Christopher Hylands
Edward A. Lee
Jie Liu
Xiaojun Liu
Lukito Muliadi
Steve Neuendorffer
John Reekie
Neil Smyth
Jeff Tsay
Brian Vogel
Winthrop Williams
Yuhong Xiong
Yang Zhao
Haiyang Zheng

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
http://ptolemy.eecs.berkeley.edu

Document Version 3.0
for use with Ptolemy II 3.0
July 16, 2003

Memorandum UCB/ERL M03/TBA
Earlier versions:
- *UCB/ERL M02/23*
- *UCB/ERL M01/12*
- *UCB/ERL M99/40*

This project is supported by the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation, Chess (the Center for Hybrid and Embedded Software Systems), the State of California MICRO program, and the following companies: Agilent, Atmel, Cadence, Hitachi, Honeywell, National Semiconductor, Philips, and Wind River Systems.

[illegible]

[illegible]

VOLUME 1

INTRODUCTION TO PTOLEMY II

This volume describes how to construct Ptolemy II models for web-based modeling or building applications. The first chapter includes an overview of Ptolemy II software, and a brief description of each of the models of computation that have been implemented. It describes the package structure of the software, and includes as an appendix a brief tutorial on UML notation, which is used throughout the documentation to explain the structure of the software. The second chapter is a tutorial on building models using Vergil, a graphical user interface where models are built pictorially. The third chapter discusses the Ptolemy II expression language, which is used to set parameter values. The next chapter gives an overview of actor libraries. These three chapters, plus one of the domain chapters, will be sufficient for users to start building interesting models in the selected domain. The fifth chapter gives a tutorial on designing actors in Java.The sixth chapter explains MoML, the XML schema used by Vergil to store models. And the seventh chapter, the final one in this part, explains how to construct custom applets.

Volume 2 describes the software architecture of Ptolemy II, and volume 3 describes the domains, each of which implements a model of computation.

Contents

1 Introduction

Author: *Edward A. Lee*
Contributors: *The entire Ptolemy team*

1.1 Purpose

This document is the first of three volumes describing the Ptolemy II software. This first volume introduces the software and explains how to use it. The second volume documents the software architecture and explains how to extend it. The third volume documents the "domains," which realize models of computation. Some of these are relatively mature and established, and some are highly experimental. Indeed, a major part of the Ptolemy Project is experimentation with models of computation.

Ptolemy II is the current software infrastructure of the Ptolemy Project. For the participants in the Ptolemy Project, Ptolemy II is first and foremost a laboratory for experimenting with design techniques. It is published freely and in open-source form for several reasons. First, the software complements more traditional publication media, and serves as a clear, unambiguous, and complete description our research results. Second, the open architecture and open source encourages researchers to build their own methods, leveraging and extending the core infrastructure provided by the software. This creates a community where much of the dialog is through the software. Third, the freely available software encourages designers to try out the new design techniques that are introduced and give feedback to the Ptolemy Project. This helps guide further research. Fourth, the open source software encourages commercial providers of software tools to commercialize the research results, which then helps to maximize the impact of the work.

The Ptolemy Project is an informal group of researchers at U.C. Berkeley. There have been many participants in the project over the years (see "Acknowledgements" on page 28 for a list contributors to Ptolemy II). Ptolemy II is the third generation of design software to emerge from this group, with each generation bringing a new set of problems being addressed, new emphasis, and (largely) a new group of contributors.

1.1.1 Gabriel (1986-1991)

The first generation of software created by this was group was called Gabriel [16]. It was written in Lisp and aimed at signal processing. It was during the construction of Gabriel that synchronous dataflow (SDF) block diagrams and both sequential and parallel scheduling techniques for SDF models matured. Gabriel included code generators for programmable DSPs that produced efficient assembly code for certain processors (notably, Motorola processors). Gabriel included hardware/software co-simulators, where parallel code generators would produce assembly code which then ran on instruction set simulators within a hardware simulation of a multiprocessor architecture. Gabriel had a graphical user interface built on top of Vem, written by Rick Spickelmeyer, which was originally designed for schematic capture in VLSI CAD. It used Oct, which was the design database developed by the Berkeley CAD group under the leadership of Professor Richard Newton.

1.1.2 Ptolemy Classic (1990-1997)

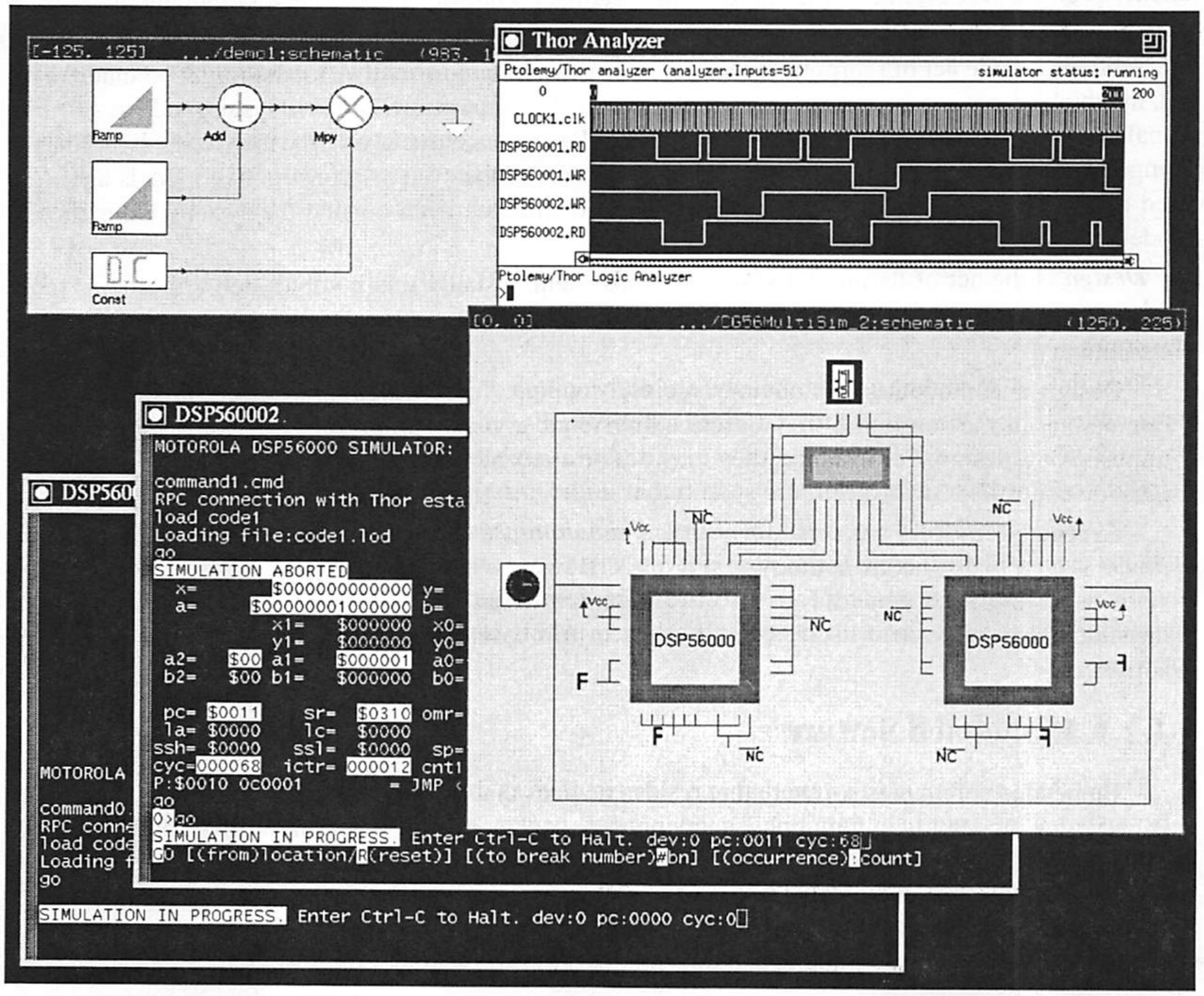
Ptolemy Classic, started jointly by Professors Edward Lee and Dave Messerschmitt in 1990, was written in C++ [19]. It was the first modeling environment to systematically support multiple models of computation, hierarchically combined. We ported the SDF capabilities from Gabriel, and also developed boolean dataflow (BDF), dynamic dataflow (DDF), multidimensional synchronous dataflow (MDSDF) and process networks (PN) domains. We also ported the DSP code generators, and created C and VHDL code generators as well. We developed the concept of "targets," which encapsulated knowledge about specific hardware platforms, and demonstrated construction of models that executed on attached embedded processors (such as S-bus cards with DSPs), including models that executed jointly on a Unix host and the attached embedded processor. We developed a discrete-event domain and demonstrated joint modeling of communication networks and signal processing, and also developed a hardware simulation domain called Thor, which was adapted from an open-source hardware simulator by the same name (see figure 1.1). We made major contributions to SDF scheduling techniques, including introducing the concept of "single appearance schedules" (which minimize generated code size and enable extensive use of inlining of generated code). We also introduced "higher-order components," which greatly increased the expressiveness of visual syntaxes. The Ptolemy Classic user interface was an extension of the Gabriel interface, still based on Oct and Vem, but extended by Tycho (written in Itcl, an object-oriented extension of Tcl/Tk). Portions of Ptolemy Classic were commercialized as part of the Agilent ADS system, and methods from Ptolemy Classic were used in Cadence's SPW system.

1.1.3 Ptolemy II (1996-?)

The Ptolemy Project (as it was now known) began working on Ptolemy II in 1996. The major reasons for starting over were to exploit the network integration, migrating code, built-in threading, and user-interface capabilities of Java. Ptolemy II introduced the notion of domain polymorphism (where components could be designed to be able to operate in multiple domains) and modal models (where finite state machines are combined hierarchically with other models of computation). We built a continuous-time domain, which combined with the modal modeling capability, yields hybrid system modeling. Ptolemy II has a sophisticated type system with type inference and data polymorphism (where components can be designed to operate on multiple data types), and a rich expression language. The concept of behavioral types emerged (where components and domains could have interface definitions that describe not just static structure, as with traditional type systems, but also dynamic behavior). Some (but not all) of the SDF capabilities from Ptolemy Classic were ported, and the heterochronous dataflow model was introduced. We contributed to a user-interface toolkit (called Diva) based on Java, built a user interface for Ptolemy II (called Vergil) based on Diva, designed a Java plotter (PtPlot), and introduced a 3-D animation domain. We built models that could be used as applets in a web browser. And we built numerous experimental domains that explored real-time and distributed computing (distributed discrete events (DDE), timed multitasking (TM), Giotto, and component interaction (CI)). As for code generation, the tactic in Ptolemy II is significantly different than that in Gabriel or Ptolemy Classic. Instead of components as generators, Ptolemy II uses a component-specialization framework built on top of a Java compiler toolkit called Soot. Ptolemy II uses XML for its persistent data representation, and has introduced the concept of migrating models.

1.1.4 Organization of this Document

This document is the first of three volumes. This first volume introduces the software and explains how to use it. It begins with a description of the rationale for the design in this chapter. The second


[-125, 125]   .../demo1:schematic   (983, 1
Ramp
Add
Mpy
Ramp
D.C.
Const
Thor Analyzer
Ptolemy/Thor analyzer (analyzer.Inputs=51)
simulator status: running
0
200
CLOCK1.clk
DSP560001.RD
DSP560001.WR
DSP560002.WR
DSP560002.RD
Ptolemy/Thor Logic Analyzer
[0, 0]   .../CG56MultiSim_2:schematic   (1250, 225)
Vcc
NC
DSP56000
DSP56000
F
DSP56002.
DSP560

```
MOTOROLA DSP56000 SIMULATOR:

command1.cmd
RPC connection with Thor esta
load code1
Loading file:code1.lod
go
SIMULATION ABORTED
  x=       $000000000000 y=
  a=     $00000001000000 b=
           x1=   $000000  x0=
           y1=   $000000  y0=
 a2=   $00 a1=   $000001  a0=
 b2=   $00 b1=   $000000  b0=

 pc= $0011    sr=  $0310 omr=
 la= $0000    lc=  $0000
ssh= $0000   ssl=  $0000  sp=
cyc=000068  ictr= 000012 cnt1
P:$0010 0c0001         = JMP <
go
0>go
SIMULATION IN PROGRESS. Enter Ctrl-C to Halt. dev:0 pc:0011 cyc:68
GO [(from)location/R(reset)] [(to break number)#bn] [(occurrence):count]
```

```
MOTOROLA

command0.
RPC conne
load code
Loading f
go

SIMULATION IN PROGRESS. Enter Ctrl-C to Halt. dev:0 pc:0000 cyc:0
```

FIGURE 1.1. Ptolemy Classic screen image (from 1993) showing an SDF graph at the upper left that is automatically mapped and scheduled onto the two processor architecture, whose model is at the lower right (in the "Thor" domain). Assembly code for the two processor is generated, and then ISA simulators of the processors (provided by Motorola, lower left) interact with the Thor-domain simulation, resulting in the logic analyzer trace at the upper right.

chapter is a tutorial that explains how to use Ptolemy II via the Vergil visual editor. The third chapter explains the expression language, which is used extensively in Ptolemy II. The fourth chapter provides an overview of the actor libraries; note, however, that the most complete documentation for the actors is built in to the software, accessed through the "Get Documentation" command, obtained by right clicking on the actor icon. The fifth chapter explains how to write actors. The sixth chapter describes MoML, the XML schema used to store Ptolemy II models.

1.2 Modeling and Design

The Ptolemy project studies heterogeneous modeling, simulation, and design of concurrent systems. The focus is on *embedded systems* [66], particularly those that mix technologies including, for example, analog and digital electronics, hardware and software, and electronics and mechanical devices. The focus is also on systems that are complex in the sense that they mix widely different operations, such as networking, signal processing, feedback control, mode changes, sequential decision making, and user interfaces.

Modeling is the act of representing a system or subsystem formally. A model might be mathematical, in which case it can be viewed as a set of assertions about properties of the system such as its functionality or physical dimensions. A model can also be constructive, in which case it defines a computational procedure that mimics a set of properties of the system. Constructive models are often used to describe behavior of a system in response to stimulus from outside the system. Constructive models are also called executable models.

Design is the act of defining a system or subsystem. Usually this involves defining one or more models of the system and refining the models until the desired functionality is obtained within a set of constraints.

Design and modeling are obviously closely coupled. In some circumstances, models may be immutable, in the sense that they describe subsystems, constraints, or behaviors that are externally imposed on a design. For instance, they may describe a mechanical system that is not under design, but must be controlled by an electronic system that is under design.

Executable models are sometimes called *simulations*, an appropriate term when the executable model is clearly distinct from the system it models. However, in many electronic systems, a model that starts as a simulation mutates into a software implementation of the system. The distinction between the model and the system itself becomes blurred in this case. This is particularly true for embedded software.

1.2.1 Embedded Software

Embedded software is software that resides in devices that are not first-and-foremost computers. It is pervasive, appearing in automobiles, telephones, pagers, consumer electronics, toys, aircraft, trains, security systems, weapons systems, printers, modems, copiers, thermostats, manufacturing systems, appliances, etc. A technically active person probably interacts regularly with more pieces of embedded software than conventional software. A key feature of embedded software is that it engages the physical world, and hence has temporal constraints that desktop software does not share.

A major emphasis in Ptolemy II is on the methodology for defining and producing embedded software together with the systems within which it is embedded.

Executable models are constructed under a *model of computation*, which is the set of "laws of physics" that govern the interaction of components in the model. If the model is describing a mechanical system, then the model of computation may literally be the laws of physics. More commonly, however, it is a set of rules that are more abstract, and provide a framework within which a designer builds models. A set of rules that govern the interaction of components is called the *semantics* of the model of computation. A model of computation may have more than one semantics, in that there might be distinct sets of rules that impose identical constraints on behavior.

The choice of model of computation depends strongly on the type of model being constructed. For example, for a purely computational system that transforms a finite body of data into another finite body of data, the imperative semantics that is common in programming languages such as C, C++, Java, and MATLAB will be adequate. For modeling a mechanical system, the semantics needs to be able to handle concurrency and the time continuum, in which case a continuous-time model of computation such that found in Simulink, Saber, Hewlett-Packard's ADS, and VHDL-AMS is more appropriate.

The ability of a model to mutate into an implementation depends heavily on the model of computation that is used. Some models of computation, for example, are suitable for implementation only in customized hardware, while others are poorly matched to customized hardware because of their intrinsically sequential nature. Choosing an inappropriate model of computation may compromise the quality of design by leading the designer into a more costly or less reliable implementation.

A principle of the Ptolemy project is that the choices of models of computation strongly affect the quality of a system design.

For embedded systems, the most useful models of computation handle concurrency and time. This is because embedded systems consist typically of components that operate simultaneously and have multiple simultaneous sources of stimuli. In addition, they operate in a timed (real world) environment, where the timeliness of their response to stimuli may be as important as the correctness of the response.

The objective in Ptolemy II is to support the construction and interoperability of executable models that are built under a wide variety of models of computation.

Ptolemy II takes a component view of design, in that models are constructed as a set of interacting components. A model of computation governs the semantics of the interaction, and thus imposes a discipline on the interaction of components.

Component-based design in Ptolemy II involves disciplined interactions between components governed by a model of computation.

1.2.2 Actor-Oriented Design

Most (but not all) models of computation in Ptolemy II support *actor-oriented design*. This contrasts with (and complements) object-oriented design by emphasizing concurrency and communication between components. Components called actors execute and communicate with other actors in a model, as illustrated in figure 1.2. Like objects, actors have a well defined component interface. This interface abstracts the internal state and behavior of an actor, and restricts how an actor interacts with its environment. The interface includes ports that represent points of communication for an actor, and parameters which are used to configure the operation of an actor. Often, parameter values are part of the *a priori* configuration of an actor and do not change when a model is executed, but not always. The "port/parameters" shown in figure 1.2 function as both ports and parameters.

Central to actor-oriented design are the communication channels that pass data from one port to another according to some messaging scheme. Whereas with object-oriented design, components interact primarily by transferring control through method calls, in actor-oriented design, they interact by sending messages through channels. The use of channels to mediate communication implies that actors interact only with the channels that they are connected to and not directly with other actors.

Like actors, a model may also define an external interface; this interface is called its *hierarchical abstraction*. This interface consists of external ports and external parameters, which are distinct from the ports and parameters of the individual actors in the model. The external ports of a model can be connected by channels to other external ports of the model or to the ports of actors that compose the model. External parameters of a model can be used to determine the values of the parameters of actors inside the model.

Taken together, the concepts of models, actors, ports, parameters and channels describe the *abstract syntax* of actor-oriented design. This syntax can be represented concretely in several ways, such as graphically, as in figure 4, in XML as in figure 1.3, or in a program designed to a specific API (as in SystemC). Ptolemy II offers all three alternatives.

It is important to realize that the syntactic structure of an actor-oriented design says little about the semantics. The semantics is largely orthogonal to the syntax, and is determined by a model of computation. The model of computation might give operational rules for executing a model. These rules determine when actors perform internal computation, update their internal state, and perform external communication. The model of computation also defines the nature of communication between components.

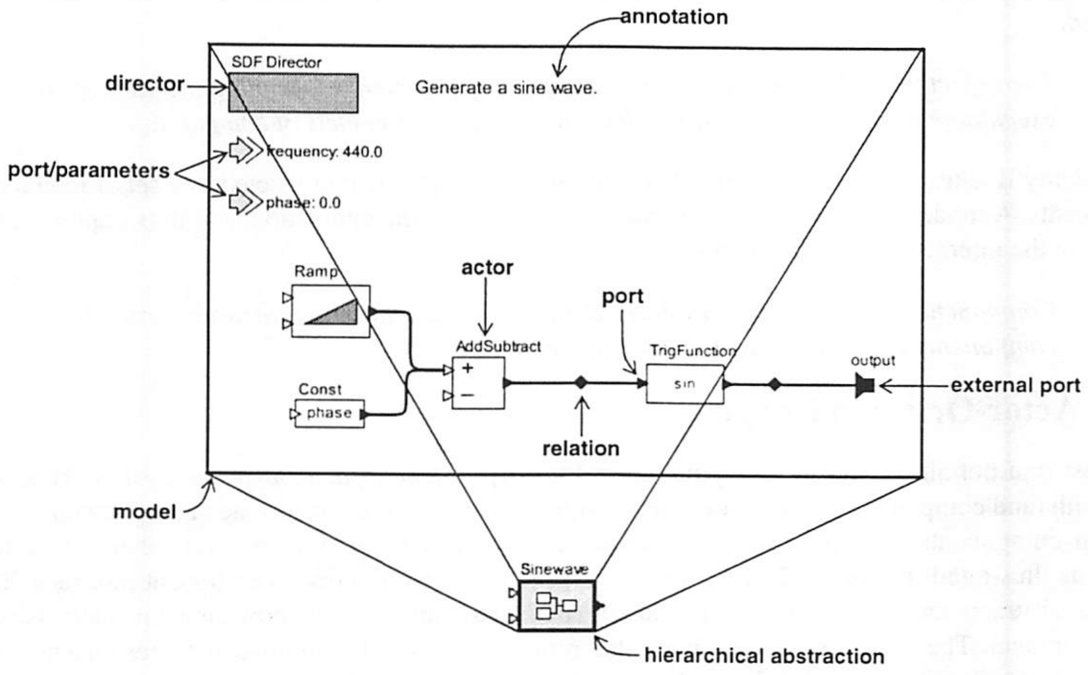


FIGURE 1.2. Illustration of an actor-oriented model (above) and its hierarchical abstraction (below).

Our notion of actor-oriented modeling is related to the work of Gul Agha and others. The term actor was introduced in the 1970's by Carl Hewitt of MIT to describe the concept of autonomous reasoning agents [48]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [1-5]. Agha's actors each have an independent thread of control and communicate via asynchronous message passing. We have further developed the term to embrace a larger family of models of concurrency that are often more constrained than general message passing. Our actors are still conceptually concurrent, but unlike Agha's actors, they need not have their own thread of control. Moreover, although communication is still through some form of message passing, it need not be strictly asynchronous.

Actor-oriented modeling has been around for some time, and is in widespread use through such programs as Simulink, from The Mathworks, LabView, from National Instruments, and many others. It is gaining broader legitimacy through the efforts of OMG in UML-2 [101], for example, some of which has its roots in the actor-oriented framework ROOM [116]. Many research projects are based on some form of actor-oriented modeling, but the Ptolemy Project is unique in the breadth of exploration of semantic alternatives and in the commitment made to a particular model of computation within a domain.

1.2.3 Architecture Design

Architecture description languages (ADLs), such as Wright [6] and Rapide [85], focus on formalisms for describing the rich sorts of component interactions that commonly arise in software architecture. Ptolemy II, by contrast, might be called an *architecture design language*, because its objective is not so much to describe existing interactions, but rather to promote coherent software architecture by imposing some structure on those interactions. Thus, while an ADL might focus on the compatibility of a sender and receiver in two distinct components, we would focus on a pattern of interactions among a set of components. Instead of, for example, verifying that a particular protocol in a single port-to-port interaction does not deadlock [6], we would focus on whether an assemblage of components can deadlock.

It is arguable that our approach is less modular, because components must be designed to the framework. Typical ADLs can describe pre-existing components, whereas in Ptolemy II, such pre-

```
<class name="Sinewave">
   <property name="samplingFrequency" value="8000.0"/>
   <property name="frequency" value="440.0"/>
   <property name="phase" value="0.0"/>
   <property name="SDF Director" class="ptolemy.domains.sdf.kernel.SDFDirector"/>
   <port name="output"><property name="output"/>
   <entity name="Ramp" class="ptolemy.actor.lib.Ramp">
      <property name="init" value="phase"/>
      <property name="step" value="frequency*2*PI/samplingFrequency"/>
   </entity>
   <entity name="TrigFunction" class="ptolemy.actor.lib.TrigFunction">
      <property name="function" value="sin" class="ptolemy.kernel.util.StringAttribute"/>
   </entity>
   <relation name="relation"/>
   <relation name="relation2"/>
   <link port="output" relation="relation2"/>
   <link port="Ramp.output" relation="relation"/>
   <link port="TrigFunction.input" relation="relation"/>
   <link port="TrigFunction.output" relation="relation2"/>
</class>
```

FIGURE 1.3. An XML representation of a simplified sinewave source.

existing components would have to wrapped in Ptolemy II actors. Moreover, designing components to a particular interface may limit their reusability, and in [illegible] interface may not ma[illegible] their needs well. All of these are valid points, and indeed a major pa[illegible]r research effort is to ameliorate these limitations. The net effect, we believe, is an approach that is much more powerful than ADLs.

First, we design components to be *domain polymorphic*, meaning that they can interact with other components within a wide variety of domains. In other words, instead of coming up with an ADL that can describe a number of different interaction mechanisms, we have come up with an architecture where components can be easily designed to interact in a number of ways. We argue that this makes the components more reusable, not less, because disciplined interaction within a well-defined semantics is possible. By contrast, with pre-existing components that have rigid interfaces, the best we can hope for is ad-hoc synthesis of adapters between incompatible interfaces, something that is likely to lead to designs that are very difficult to understand and to verify. Whereas ADLs draw an analogy between compatibility of interfaces and type checking [6], we use a technique much more powerful than type checking alone, namely polymorphism [68].

Second, to avoid the problem that a particular interaction mechanism may not fit the needs of a component well, we provide a rich set of interaction mechanisms embodied in the Ptolemy II domains. The domains force component designers to think about the overall pattern of interactions, and trade off uniformity for expressiveness. Where expressiveness is paramount, the ability of Ptolemy II to hierarchically mix domains offers essentially the same richness of more ad-hoc designs, but with much more discipline. By contrast, a non-trivial component designed without such structure is likely to use a *melange*, or ad-hoc mixture of interaction mechanisms, making it difficult to embed it within a comprehensible system.

Third, whereas an ADL might choose a particular model of computation to provide it with a formal structure, such as CSP for Wright [6], we have developed a more abstract formal framework that describes models of computation at a meta level [71]. This means that we do not have to perform awkward translations to describe one model of computation in terms of another. For example, stream based communication via FIFO channels are awkward in Wright [6].

We make these ideas concrete by describing the models of computation implemented in the Ptolemy II domains.

1.3 Models of Computation

There is a rich variety of models of computation that deal with concurrency and time in different ways. Each gives an interaction mechanism for components. The utility of a model of computation stems from the modeling properties that apply to all similar models. For many models of computation these properties are derived through formal mathematics. Depending on the model of computation, the model may be determinate [55], statically schedulable [72], or time safe [47]. Because of its modeling properties, a model of computation represents a style of modeling that is useful in any circumstance where those properties are desirable. In other words, models of computation form design patterns of component interaction, in the same sense that Gamma, et al. describe design patterns in object oriented languages [35].

For a particular application, an appropriate model of computation does not impose unnecessary constraints, and at the same time is constrained enough to result in useful derived properties. For example, by restricting the design space to synchronous designs, Scenic [74] enables cycle-driven simulation [41], which greatly improves execution efficiency over more general discrete-event models of computation (such as that found in VHDL). However, for applications with multirate behavior, synchronous design can be constraining. In such cases, a less constrained model of computation, such as synchronous dataflow [72] or Kahn process networks [55] may be more appropriate. One drawback of this relaxation of synchronous design constraints is that buffering becomes more difficult to analyze. On the other hand, techniques exist for synchronous dataflow that allow co-optimization of memory usage and execution latency [118] that would otherwise be difficult to apply to a multirate system. Selecting an appropriate model of computation for a particular application is often difficult, but this is a problem we should embrace instead of avoiding.

In this section, we describe models of computation that are implemented in Ptolemy II domains. Our focus has been on models of computation that are most useful for embedded systems. All of these can lend a semantics to the same bubble-and-arc, or block-and-arrow diagram shown in figure 1.4. Ptolemy II models are (clustered, or hierarchical) graphs of the form of figure 1.4, where the nodes are *entities* and the arcs are *relations*. For most domains, the entities are *actors* (entities with functionality) and the relations connecting them represent communication between actors.

1.3.1 Component Interaction - CI

The component interaction (CI) domain, created by Xiaojun Liu and Yang Zhao, models systems that blend data-driven and demand-driven styles of computation. As an example, the interaction between a web server and a browser is mostly demand-driven. When the user clicks on a link in the browser, it pulls the corresponding page from the web server. A stock-quote service can use a data-driven style of computation. The server generates events when stock prices change. The data drive the clients to update their displayed information. Such push/pull interaction between a data producer and consumer is common in distributed systems, and has been included in middleware services, most notably in the CORBA event service. These services motivated the design of this domain to study the interaction models in distributed systems, such as stock-quote services, traffic or weather information systems. Other applications include database systems, file systems, and the Click modular router [57].

An actor in a CI model can be active, which means it possesses its own thread of execution. For example, an interrupt source of an embedded system can be modeled as an active source actor. Such a source generates events asynchronously with respect to the software execution on the embedded processor. CI models can be used to simulate and study how the embedded software handles the asynchronous events, such as external interrupts and asynchronous I/O.

1.3.2 Communicating Sequential Processes - CSP

In the CSP domain (communicating sequential processes), created by Neil Smyth [117], actors

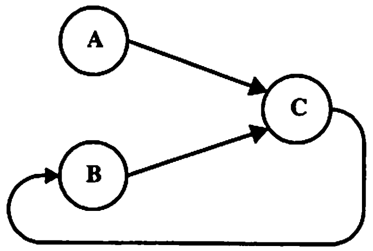


FIGURE 1.4. A single *syntax* (bubble-and-arc or block-and-arrow diagram) can have a number of possible *semantics* (interpretations).

represent concurrently executing processes, implemented as Java threads. These processes communicate by atomic, instantaneous actions called *rendezvous* (or sometimes, *synchronous message passing*). If two processes are to communicate, and one reaches the point first at which it is ready to communicate, then it stalls until the other process is ready to communicate. "Atomic" means that the two processes are simultaneously involved in the exchange, and that the exchange is initiated and completed in a single uninterruptable step. Examples of rendezvous models include Hoare's *communicating sequential processes* (CSP) [51] and Milner's *calculus of communicating systems* (CCS) [90]. This model of computation has been realized in a number of concurrent programming languages, including Lotos and Occam.

Rendezvous models are particularly well-matched to applications where resource sharing is a key element, such as client-server database models and multitasking or multiplexing of hardware resources. A key feature of rendezvous-based models is their ability to cleanly model nondeterminate interactions. The CSP domain implements both conditional send and conditional receive. It also includes an experimental timed extension.

1.3.3 Continuous Time - CT

In the CT domain (continuous time), created Jie Liu [81], actors represent components that interact via continuous-time signals. Actors typically specify algebraic or differential relations between inputs and outputs. The job of the director in the domain is to find a fixed-point, i.e., a set of continuous-time functions that satisfy all the relations.

The CT domain includes an extensible set of differential equation solvers. The domain, therefore, is useful for modeling physical systems with linear or nonlinear algebraic/differential equation descriptions, such as analog circuits and many mechanical systems. Its model of computation is similar to that used in Simulink, Saber, and VHDL-AMS, and is closely related to that in Spice circuit simulators.

Mixed Signal Models. Embedded systems frequently contain components that are best modeled using differential equations, such as MEMS and other mechanical components, analog circuits, and microwave circuits. These components, however, interact with an electronic system that may serve as a controller or a recipient of sensor data. This electronic system may be digital. Joint modeling of a continuous subsystem with digital electronics is known as *mixed signal modeling* [82]. The CT domain is designed to interoperate with other Ptolemy domains, such as DE, to achieve mixed signal modeling. To support such modeling, the CT domain models of discrete events as Dirac delta functions. It also includes the ability to precisely detect threshold crossings to produce discrete events.

Modal Models and Hybrid Systems. Physical systems often have simple models that are only valid over a certain regime of operation. Outside that regime, another model may be appropriate. A *modal model* is one that switches between these simple models when the system transitions between regimes. The CT domain interoperates with the FSM domain to create modal models. Such modal models are often called *hybrid systems.*

1.3.4 Discrete-Events - DE

In the discrete-event (DE) domain, created by Lukito Muliadi [94], the actors communicate via sequences of events placed in time, along a real time line. An *event* consists of a *value* and *time stamp.* Actors can either be processes that react to events (implemented as Java threads) or functions that fire when new events are supplied. This model of computation is popular for specifying digital hardware and for simulating telecommunications systems, and has been realized in a large number of simulation environments, simulation languages, and hardware description languages, including VHDL and Verilog.

DE models are excellent descriptions of concurrent hardware, although increasingly the globally consistent notion of time is problematic. In particular, it over-specifies (or over-models) systems where maintaining such a globally consistent notion is difficult, including large VLSI chips with high clock rates. Every event is placed precisely on a globally consistent time line.

The DE domain implements a fairly sophisticated discrete-event simulator. DE simulators in general need to maintain a global queue of pending events sorted by time stamp (this is called a *priority queue*). This can be fairly expensive, since inserting new events into the list requires searching for the right position at which to insert it. The DE domain uses a calendar queue data structure [17] for the global event queue. A calendar queue may be thought of as a hashtable that uses quantized time as a hashing function. As such, both enqueue and dequeue operations can be done in time that is independent of the number of events in the queue.

In addition, the DE domain gives deterministic semantics to simultaneous events, unlike most competing discrete-event simulators. This means that for any two events with the same time stamp, the order in which they are processed can be inferred from the structure of the model. This is done by analyzing the graph structure of the model for data precedences so that in the event of simultaneous time stamps, events can be sorted according to a secondary criterion given by their precedence relationships. VHDL, for example, uses delta time to accomplish the same objective.

1.3.5 Distributed Discrete Events - DDE

The distributed discrete-event (DDE) domain, created by John Davis [26], can be viewed either as a variant of DE or as a variant of PN (described below). Still highly experimental, it addresses a key problem with discrete-event modeling, namely that the global event queue imposes a central point of control on a model, greatly limiting the ability to distribute a model over a network. Distributing models might be necessary either to preserve intellectual property, to conserve network bandwidth, or to exploit parallel computing resources.

The DDE domain maintains a local notion of time on each connection between actors, instead of a single globally consistent notion of time. Each actor is a process, implemented as a Java thread, that can advance its local time to the minimum of the local times on each of its input connections. The domain systematizes the transmission of null events, which in effect provide guarantees that no event will be supplied with a time stamp less than some specified value.

1.3.6 Discrete Time - DT

The discrete-time (DT) domain, written by Chamberlain Fong [32], extends the SDF domain (described below) with a notion of time between tokens. Communication between actors takes the form of a sequence of tokens where the time between tokens is uniform. Multirate models, where distinct connections have distinct time intervals between tokens, are also supported. There is considerable subtlety in this domain when multirate components are used. The semantics is defined so that component behavior is always causal, in that outputs whose values depend on inputs are never produced at times prior to those of the inputs.

1.3.7 Finite-State Machines - FSM

The finite-state machine (FSM) domain, written by Xiaojun Liu, is radically different from the other Ptolemy II domains. The entities in this domain represent not actors but rather *state*, and the connections represent *transitions* between states. Execution is a strictly ordered sequence of state transitions. The FSM domain leverages the built-in expression language in Ptolemy II to evaluate *guards*, which determine when state transitions can be taken.

FSM models are excellent for expressing control logic and for building modal models (models with distinct modes of operation, where behavior is different in each mode). FSM models are amenable to in-depth formal analysis, and thus can be used to avoid surprising behavior.

**Charts.* FSM models have some key weaknesses. First, at a very fundamental level, they are not as expressive as the other models of computation described here. They are not sufficiently rich to describe all partial recursive functions. However, this weakness is acceptable in light of the formal analysis that becomes possible. Many questions about designs are decidable for FSMs and undecidable for other models of computation. A second key weakness is that the number of states can get very large even in the face of only modest complexity. This makes the models unwieldy.

Both problems can often be solved by using FSMs in combination with concurrent models of computation. This was first noted by David Harel, who introduced that Statecharts formalism. Statecharts combine a loose version of synchronous-reactive modeling (described below) with FSMs [42]. FSMs have also been combined with differential equations, yielding the so-called *hybrid systems* model of computation [45].

The FSM domain in Ptolemy II can be hierarchically combined with other domains. We call the resulting formalism "*charts" (pronounced "starcharts") where the star represents a wildcard [38]. Since most other domains represent concurrent computations, *charts model concurrent finite state machines with a variety of concurrency semantics. When combined with CT, they yield hybrid systems and modal models. When combined with SR (described below), they yield something close to Statecharts. When combined with process networks, they resemble SDL [115].

1.3.8 Process Networks - PN

In the process networks (PN) domain, created by Mudit Goel [39], processes communicate by sending messages through channels that can buffer the messages. The sender of the message need not wait for the receiver to be ready to receive the message. This style of communication is often called asynchronous message passing. There are several variants of this technique, but the PN domain specifically implements one that ensures determinate computation, namely Kahn process networks [55].

In the PN model of computation, the arcs represent sequences of data values (tokens), and the entities represent functions that map input sequences into output sequences. Certain technical restrictions on these functions are necessary to ensure determinacy, meaning that the sequences are fully specified. In particular, the function implemented by an entity must be *prefix monotonic*. The PN domain realizes a subclass of such functions, first described by Kahn and MacQueen [56], where *blocking reads* ensure monotonicity.

PN models are loosely coupled, and hence relatively easy to parallelize or distribute. They can be implemented efficiently in both software and hardware, and hence leave implementation options open. A key weakness of PN models is that they are awkward for specifying control logic, although much of this awkwardness may be ameliorated by combining them with FSM.

The PN domain in Ptolemy II has a highly experimental timed extension. This adds to the blocking reads a method for stalling processes until time advances. We anticipate that this timed extension will make interoperation with timed domains much more practical.

1.3.9 Synchronous Dataflow - SDF

The synchronous dataflow (SDF) domain, created by Steve Neuendorffer, handles regular computations that operate on streams. Dataflow models, popular in signal processing, are a special case of process networks (for the complete explanation of this, see [70]). Dataflow models construct processes of a process network as sequences of atomic actor *firings*. Synchronous dataflow (SDF) is a particularly restricted special case with the extremely useful property that deadlock and boundedness are decidable. Moreover, the schedule of firings, parallel or sequential, is computable statically, making SDF an extremely useful specification formalism for embedded real-time software and for hardware.

Certain generalizations sometimes yield to similar analysis. Boolean dataflow (BDF) models sometimes yield to deadlock and boundedness analysis, although fundamentally these questions are undecidable. Dynamic dataflow (DDF) uses only run-time analysis, and thus makes no attempt to statically answer questions about deadlock and boundedness. Neither a BDF nor DDF domain has yet been written in Ptolemy II. Process networks (PN) serves in the interim to handle computations that do not match the restrictions of SDF.

1.3.10 Giotto

The Giotto domain, created by Christoph Meyr Kirsch, realizes a model of computation developed by Tom Henzinger, Christoph Kirsch, Ben Horowitz and Haiyang Zheng [44]. This domain has a time-triggered flavor, where each actor is invoked periodically with a specified period. The domain is designed to work with the FSM domain to realize modal models. It is intended for hard-real-time systems, where resource allocation is precomputed.

1.3.11 Synchronous/Reactive - SR

In the synchronous/reactive (SR) domain, written by Paul Whitaker [121] implements a model of computation [11] where the arcs represent data values that are aligned with global clock ticks. Thus, they are discrete signals, but unlike discrete time, a signal need not have a value at every clock tick. The entities represent relations between input and output values at each tick, and are usually partial functions with certain technical restrictions to ensure determinacy. Examples of languages that use the SR model of computation include Esterel [13], Signal [12], Lustre [23], and Argos [86].

SR models are excellent for applications with concurrent and complex control logic. Because of the tight synchronization, safety-critical real-time applications are a good match. However, also because of the tight synchronization, some applications are overspecified in the SR model, limiting the implementation alternatives. Moreover, in most realizations, modularity is compromised by the need to seek a global fixed point at each clock tick. The SR domain implementation in Ptolemy II is similar to the SR implementation in Ptolemy Classic by Stephen Edwards[28].

1.3.12 Timed Multitasking - TM

The timed multitasking (TM) domain, created by Jie Liu [80], supports the design of concurrent real-time software. It assumes an underlying priority-driven preemptive scheduler, such as that typically found in a real-time operating systems (RTOS). But the behavior of models is more deterministic than that obtained by more ad hoc uses of an RTOS.

In TM, each actor executes (conceptually) as a concurrent task. It is a timed domain, meaning that there is a notion of "model time" that advances monotonically and uniformly. Each actor has a specified execution time *T*, and it delays the production of the outputs until it has had access to the CPU for that specified amount of time (in model time, which may or may not match real time). Actors execute when they receive new inputs, so the execution is event driven. Conceptually, the actor begins execution at some time t, and its output is produced at time $t + T + P$, where T is the declared execution time, and P is the amount of time where the actor is suspended due to being preempted by a higher priority actor. At any given model time t, the task with the highest priority that has received inputs but not yet produced its outputs has the CPU. All other tasks are suspended.

TM offers a way to design real-time systems that is more deterministic than ad hoc uses of an RTOS. In particular, typically, a task produces outputs at a time that depends on the actual execution time of the task, rather than on some declared parameter. This means that consumers of that data may or may not see updates to the data, depending on when their execution occurs relative to the actual execution time. Thus, the computational results that are produced depend on the actual execution time. TM avoids this by declaring the time that elapses before production of the outputs. By maintaining model time correctly, TM ensures that the data computation is deterministic, irrespective of actual execution time.

1.4 Choosing Models of Computation

The rich variety of concurrent models of computation outlined in the previous section can be daunting to a designer faced with having to select them. Most designers today do not face this choice because they get exposed to only one or two. This is changing, however, as the level of abstraction and domain-specificity of design software both rise. We expect that sophisticated and highly visual user interfaces will be needed to enable designers to cope with this heterogeneity.

An essential difference between concurrent models of computation is their modeling of time. Some are very explicit by taking time to be a real number that advances uniformly, and placing events on a time line or evolving continuous signals along the time line. Others are more abstract and take time to be discrete. Others are still more abstract and take time to be merely a constraint imposed by causality. This latter interpretation results in time that is partially ordered, and explains much of the expressiveness in process networks and rendezvous-based models of computation. Partially ordered time provides a mathematical framework for formally analyzing and comparing models of computation [71].

A grand unified approach to modeling would seek a concurrent model of computation that serves all purposes. This could be accomplished by creating a *melange*, a mixture of all of the above, but such a mixture would be extremely complex and difficult to use, and synthesis and simulation tools would be difficult to design.

Another alternative would be to choose one concurrent model of computation, say the rendezvous model, and show that all the others are subsumed as special cases. This is relatively easy to do, in theory. It is the premise of Wright [6] and Metropolis (http://www.gigascale.org/metropolis/), for example. Most of these models of computation are sufficiently expressive to be able to subsume most of the others. However, this fails to acknowledge the strengths and weaknesses of each model of computation. Wright, for example, uses rendezvous, which is very good at resource management, but very awkward for loosely coupled data-oriented computations. Asynchronous message passing is the reverse, where resource management is awkward, but data-oriented computations are natural[1]. Thus, to design interesting systems, designers need to use heterogeneous models.

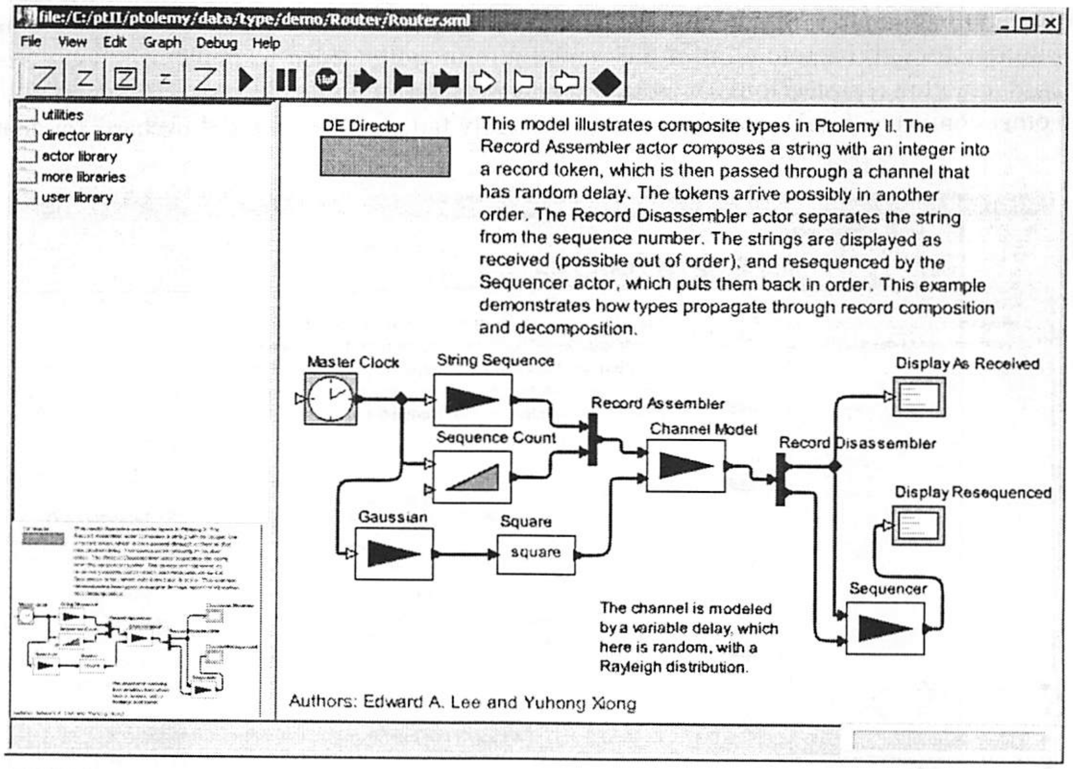
1.5 Visual Syntaxes

Visual depictions of systems have always held a strong human appeal, making them extremely effective in conveying information about a design. Many of the domains of interest in the Ptolemy project use such depictions to completely and formally specify models.

One of the principles of the Ptolemy project is that visual depictions of systems can help to offset the increased complexity that is introduced by heterogeneous modeling.

These visual depictions offer an alternative *syntax* to associate with the semantics of a model of computation. Visual syntaxes can be every bit as precise and complete as textual syntaxes, particularly when they are judiciously combined with textual syntaxes.

Figures 1.5 and 1.6 show two different visual renditions of Ptolemy II models. Both renditions are constructed in Vergil, the visual editor framework in Ptolemy II designed by Steve Neuendorffer. In figure 1.5, a Ptolemy II model is shown as a block diagram, which is an appropriate rendition for many discrete event models. In this particular example, records are constructed at the left by composing strings with integers representing a sequence number. The records are launched into a network that



This model illustrates composite types in Ptolemy II. The Record Assembler actor composes a string with an integer into a record token, which is then passed through a channel that has random delay. The tokens arrive possibly in another order. The Record Disassembler actor separates the string from the sequence number. The strings are displayed as received (possible out of order), and resequenced by the Sequencer actor, which puts them back in order. This example demonstrates how types propagate through record composition and decomposition.

The channel is modeled by a variable delay, which here is random, with a Rayleigh distribution.

Authors: Edward A. Lee and Yuhong Xiong

FIGURE 1.5. Visual rendition of a Ptolemy II model as a block diagram in Vergil (in the DE domain).

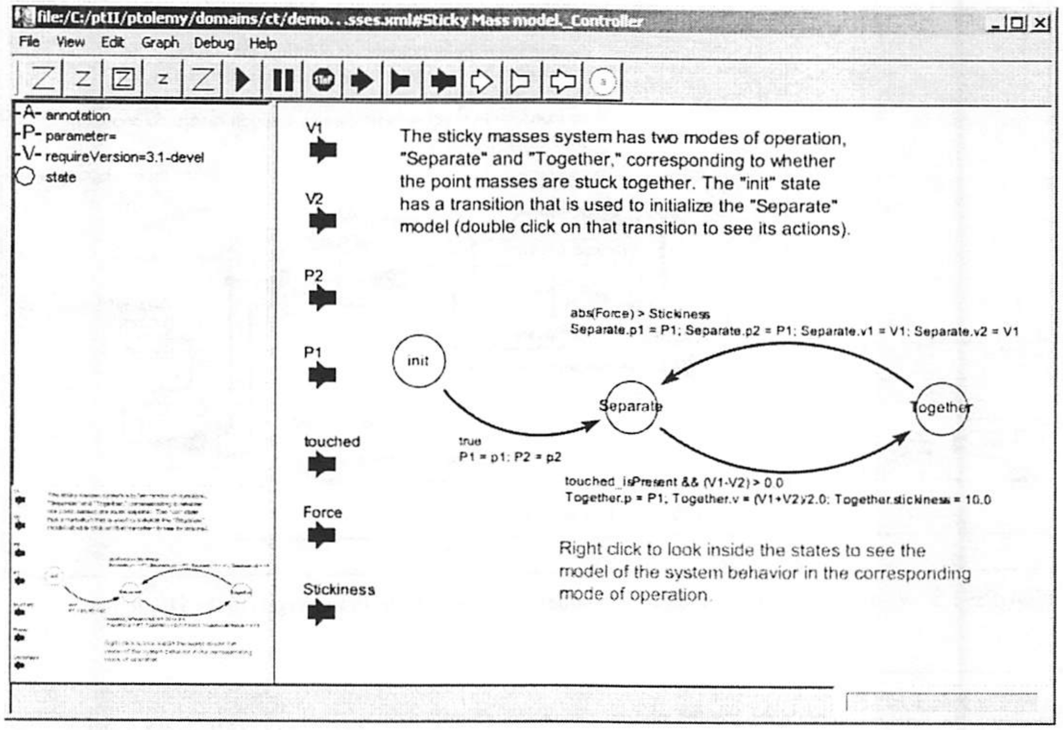
1. Consider the difference between the telephone (rendezvous) and email (asynchronous message passing). If you are trying to schedule a meeting between four busy people, getting them all on a conference call would lead to a quick resolution of the meeting schedule. Scheduling the meeting by email could take several days, and may in fact never converge. Other sorts of communication, however, are far more efficient by email.

introduces random delay. The records may arrive at the right out of order, but the Sequence actor is used to re-order them using the sequence number.

Figure 1.6 also shows a visual rendition of a Ptolemy II model, but now, the components are represented by circles, and the connections between components are represented by labeled arcs. This visual syntax is a familiar way to represent finite state machines (FSMs). Each circle represents a state of the model, and the arcs represent transitions between states. The particular example in the figure comes from a hybrid system model, where the two states, Separate and Together, represent two different modes of operation of a continuous-time system. The arcs are labeled with two lines, the first of which is a *guard*, and the second of which is an *action*. The guard is a boolean-valued textual expression that specifies when the transition should be taken, and the action is a sequence of commands that are executed when the transition is taken.

The visual renditions in figures 1.5 and 1.6 are both constructed using the same underlying infrastructure, Vergil, built by Stephen Neuendorffer. Vergil, in turn, in built on top of a GUI package called Diva, developed by John Reekie and Michael Shilman at Berkeley. Diva, in turn, is built on top of Swing and Java 2D, which are part of the Java platform from Sun Microsystems. In Vergil, a visual editor is constructed as an assembly of components in a Ptolemy II model. Thus, the system is configurable and customizable, and a great deal of infrastructure can be shared between the two distinct visual editors of figures 1.5 and 1.6.

Visual representations of models have a mixed history. In circuit design, schematic diagrams used to be routinely used to capture all of the essential information needed to implement some systems. Schematics are often replaced today by text in hardware description languages such as VHDL or Verilog. In other contexts, visual representations have largely failed, for example flowcharts for capturing


file:/C:/ptII/ptolemy/domains/ct/demo...sses.xml#Sticky Mass model_Controller
File View Edit Graph Debug Help
A- annotation
P- parameter=
V- requireVersion=3.1-devel
state
V1
V2
P2
P1
touched
Force
Stickiness
init
Separate
Together

The sticky masses system has two modes of operation, "Separate" and "Together," corresponding to whether the point masses are stuck together. The "init" state has a transition that is used to initialize the "Separate" model (double click on that transition to see its actions).

```
abs(Force) > Stickiness
Separate.p1 = P1; Separate.p2 = P1; Separate.v1 = V1; Separate.v2 = V1

true
P1 = p1; P2 = p2

touched_isPresent && (V1-V2) > 0.0
Together.p = P1; Together.v = (V1+V2)/2.0; Together.stickiness = 10.0
```

Right click to look inside the states to see the model of the system behavior in the corresponding mode of operation.

FIGURE 1.6. Visual rendition of a Ptolemy II model as a state transition diagram in Vergil (FSM domain).

the behavior of software. Recently, a number of innovative visual formalisms have been garnering support, including visual dataflow, hierarchical concurrent finite state machines, and object models. The UML visual language for object modeling has been receiving a great deal of attention. The static structure diagrams of UML, in fact, are used fairly extensively in the design of Ptolemy II itself (see appendix A of this chapter). Moreover, the Statecharts diagrams of UML are very similar to a hierarchical composition of the FSM and SR domains in Ptolemy II.

A subset of visual languages that are recognizable as "block diagrams" represent concurrent systems. There are many possible concurrency semantics (and many possible models of computation) associated with such diagrams. Formalizing the semantics is essential if these diagrams are to be used for system specification and design. Ptolemy II supports exploration of the possible concurrency semantics. A principle of the project is that the strengths and weaknesses of these alternatives make them complementary rather than competitive. Thus, interoperability of diverse models is essential.

1.6 Ptolemy II Architecture

Ptolemy II offers a unified infrastructure for implementations of a number of models of computation. The overall architecture consists of a set of packages that provide generic support for all models of computation and a set of packages that provide more specialized support for particular models of computation. Examples of the former include packages that contain math libraries, graph algorithms, an interpreted expression language, signal plotters, and interfaces to media capabilities such as audio. Examples of the latter include packages that support clustered graph representations of models, packages that support executable models, and *domains*, which are packages that implement a particular model of computation.

Ptolemy II is modular, with a careful package structure that supports a layered approach. The *core packages* support the data model, or *abstract syntax*, of Ptolemy II designs. They also provide the *abstract semantics* that allows domains to interoperate with maximum information hiding. The *UI packages* provide support for our XML file format, called MoML, and a visual interface for constructing models graphically. The *library packages* provide actor libraries that are *domain polymorphic*, meaning that they can operate in a variety of domains. And finally, the *domain packages* provide domains, each of which implements a model of computation, and some of which provide their own, domain-specific actor libraries.

1.6.1 Core Packages

The core packages are shown in figure 1.7. This is a UML package diagram. The name of each package is in the tab at the top of each box. Subpackages are contained within their parent package. Dependencies between packages are shown by dotted lines with arrow heads. For example, *actor* depends on *kernel* which depends on *kernel.util*. *Actor* also depends on *data* and *graph*. The role of each package is explained below.

actor	This package supports executable entities that receive and send data through ports. It includes both untyped and typed actors. For typed actors, it implements a sophisticated type system that supports polymorphism. It includes the base class Director that is extended in domains to control the execution of a model.
actor.lib	This subpackage and its subpackages contain domain polymorphic actors. The actor.lib package is discussed further in section 1.6.4.
actor.parameters	

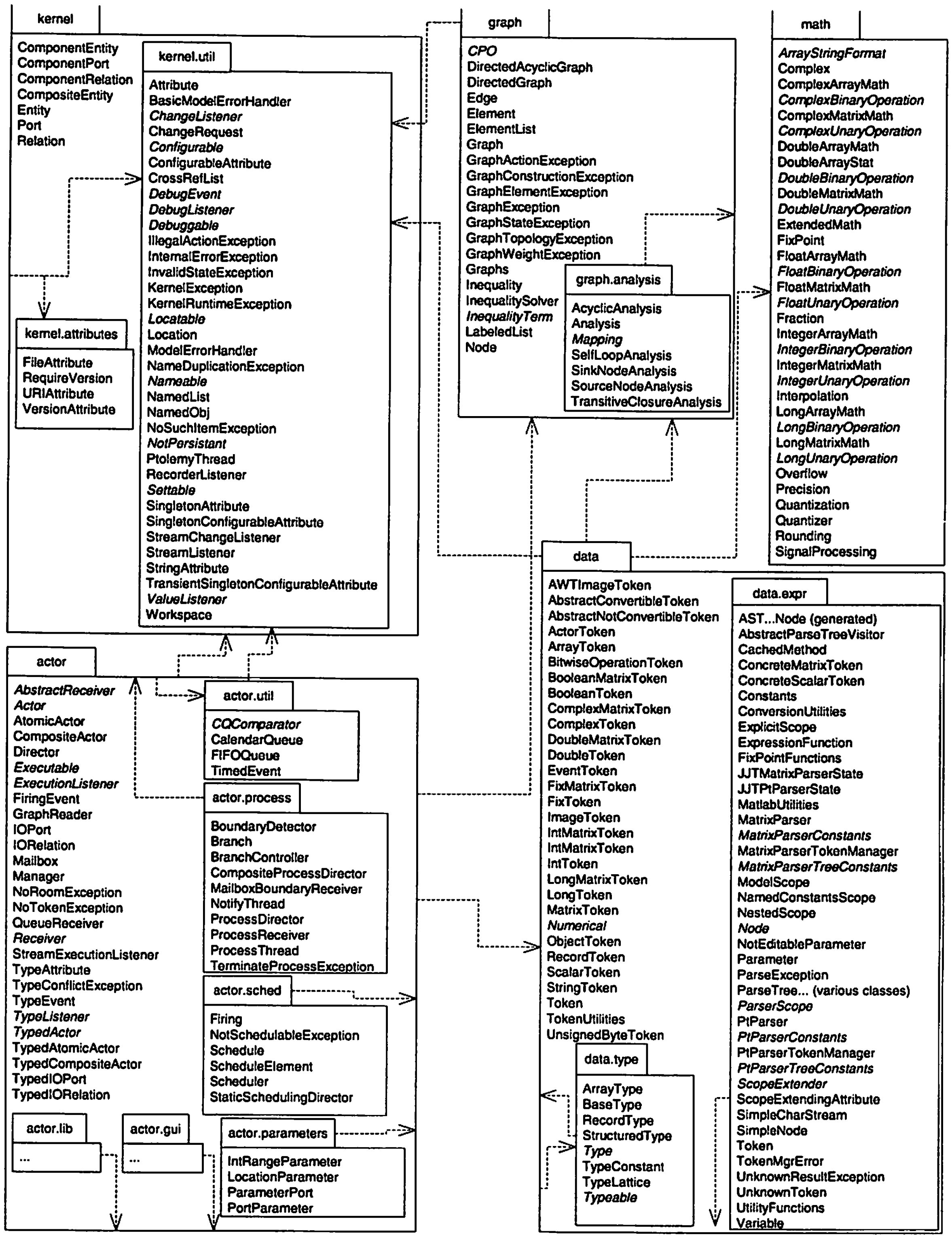


FIGURE 1.7. The core packages shown here support the data model (abstract syntax) and the actor model, (abstract semantics) of Ptolemy II designs.

	This subpackage provides specialized parameters for specifying locations, ranges of values, etc.
actor.process	This subpackage provides infrastructure for domains where actors are processes implemented on top of Java threads.
actor.sched	This subpackage provides infrastructure for domains where actors are statically scheduled by the director, or where there is static analysis of the topology of a model associated with scheduling.
actor.util	This subpackage contains utilities that support directors in various domains. Specifically, it contains a simple FIFO Queue and a sophisticated priority queue called a calendar queue.
copernicus	This subpackage contains the "actor specialization" infrastructure (Java code generation).
data	This package provides classes that encapsulate and manipulate data that is transported between actors in Ptolemy models. The key class is the Token class, which defines a set of polymorphic methods for operating on tokens, such as add(), subtract(), etc.
data.expr	This class supports an extensible expression language and an interpreter for that language. Parameters can have values specified by expressions. These expressions may refer to other parameters. Dependencies between parameters are handled transparently, as in a spreadsheet, where updating the value of one will result in the update of all those that depend on it.
data.type	This package contains classes and interfaces for the type system.
graph	This package provides algorithms for manipulating and analyzing mathematical graphs. This package is expected to supply a growing library of algorithms. These algorithms support scheduling and analysis of Ptolemy II models.
kernel	This package provides the software architecture for the Ptolemy II data model, or *abstract syntax*. This abstract syntax has the structure of clustered graphs. The classes in this package support *entities* with *ports*, and *relations* that connect the ports. Clustering is where a collection of entities is encapsulated in a single *composite entity*, and a subset of the ports of the inside entities are exposed as ports of the composite entity.
kernel.attributes	This subpackage of the kernel package provides specialized attributes such as FileAttribute, which is used in actors to specify a file or URL.
kernel.util	This subpackage of the kernel package provides a collection of utility classes that do not depend on the kernel package. It is separated into a subpackage so that these utility classes can be used without the kernel. The utilities include a collection of exceptions, classes supporting named objects with attributes, lists of named objects, a specialized cross-reference list class, and a thread class that helps Ptolemy keep track of executing threads.
math	This package encapsulates 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.
matlab	This package contains the MATLAB interface.

util This package contains various Ptolemy-independent utilities, such as string utilities and XML utilities.

1.6.2 Overview of Key Classes

Some of the key classes in Ptolemy II are shown in figure 1.8. This is a UML static structure diagram (see appendix A of this chapter). The key syntactic elements are boxes, which represent classes, the hollow arrow, which indicates generalization (or subclassing), and other lines, which indicate associations. Some lines have a small diamond, which indicates aggregation. The details of these classes will be discussed in subsequent chapters.

Instances of all of the classes shown can have names; they all implement the Nameable interface. Most of the classes generalize NamedObj, which in addition to being nameable can have a list of attributes associated with it. Attributes themselves are instances of NamedObj.

Entity, Port, and Relation are three key classes that extend NamedObj. These classes define the

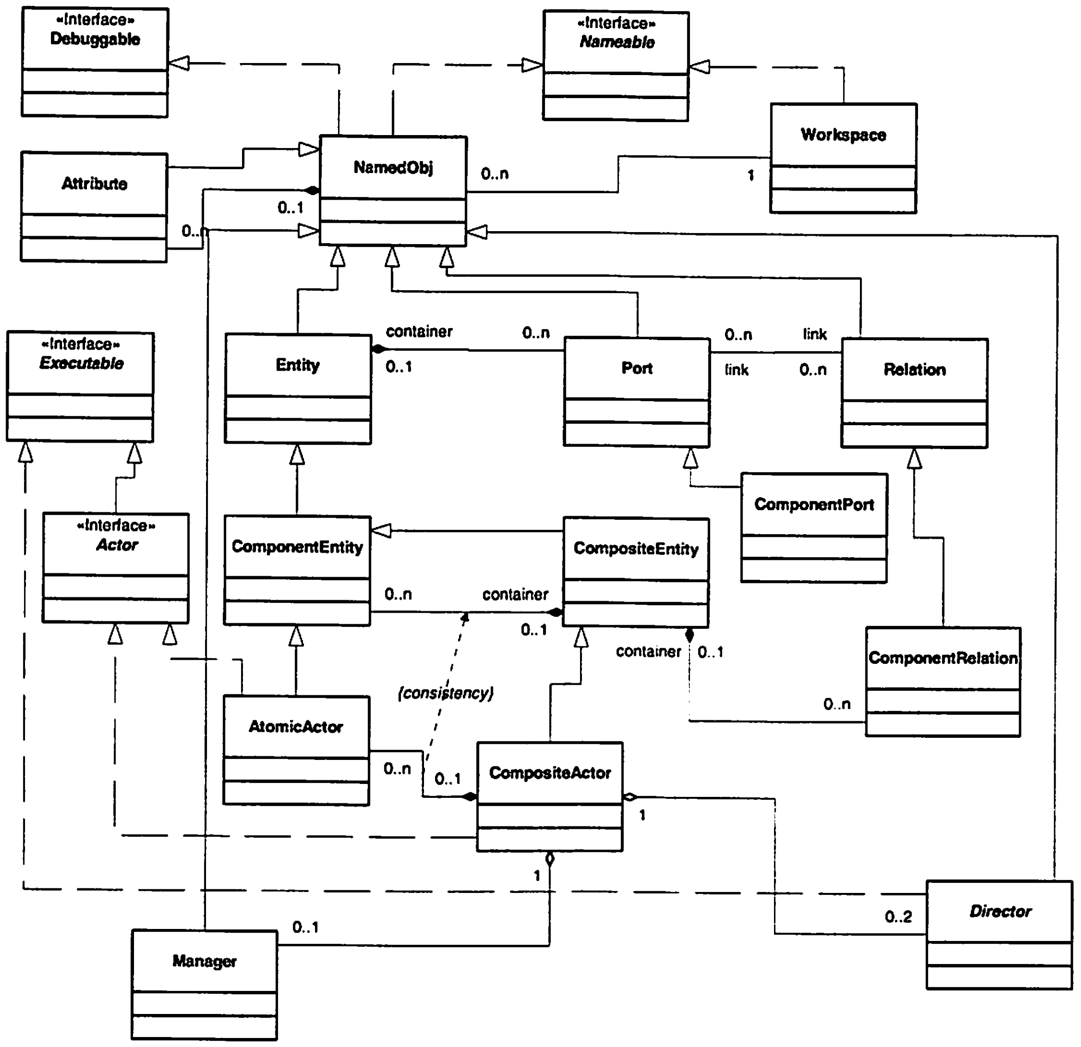


FIGURE 1.8. Some of the key classes in Ptolemy II. These are defined in the *kernel*, *kernel.util*, and *actor* packages. They define the Ptolemy II abstract syntax and abstract semantics.

primitives of the abstract syntax supported by Ptolemy II. They are fully explained in the kernel chapter. ComponentPort, ComponentRelation, and ComponentEntity extend these classes by adding support for clustered graphs. CompositeEntity extends ComponentEntity and represents an aggregation of instances of ComponentEntity and ComponentRelation.

The Executable interface, explained in the actors chapter, defines objects that can be executed. The Actor interface extends this with capability for transporting data through ports. AtomicActor and CompositeActor are concrete classes that implement this interface. The Executable and Actor interfaces are key to the Ptolemy II abstract semantics.

An executable Ptolemy II model consists of a top-level CompositeActor with an instance of Director and an instance of Manager associated with it. The manager provides overall control of the execution (starting, stopping, pausing). The director implements a semantics of a model of computation to govern the execution of actors contained by the CompositeActor.

Director is the base class for directors that implement models of computation. Each such director is associated with a domain. We have defined in Ptolemy II directors that implement continuous-time modeling (ODE solvers), process networks, synchronous dataflow, discrete-event modeling, and communicating sequential processes.

1.6.3 Domains

The domains in Ptolemy II are subpackages of the ptolemy.domains package. The more mature and frequently used domains are shown in figure 1.9. The experimental domains and less commonly used domains are not shown, but the examples in figure 1.9 are illustrative of their structure. These packages generally contain a kernel subpackage, which defines classes that extend those in the actor or kernel packages of Ptolemy II. The lib subpackage, when it exists, includes domain-specific actors.

1.6.4 Library Packages

Most domains extend classes in the actor package to give a specific semantic interpretation to an interconnection of actors. It is possible, and strongly encouraged, to define actors in such a way that they can operate in multiple domains. Such actors are said to be *domain polymorphic*. Actor that are domain polymorphic are organized in the packages shown in figure 1.10. These packages are briefly described below:

actor.lib	This package is the main library of polymorphic actors.
actor.lib.comm	This is a library of actors for modeling communications systems.
actor.lib.conversions	This package provides domain polymorphic actors that convert data between different types.
actor.lib.gui	This package is a library of polymorphic actors with user interface components, such as plotters.
actor.lib.hoc	This package is a library of higher-order components, which are components that construct portions of a model.
actor.lib.image	This package is a library of image processing actors.

actor.lib.io This package provides file I/O.

actor.lib.io.comm

This package provides an actor that communicate via the serial ports. This actor works only under Windows.

actor.lib.jai This is a library of image processing actors based on the *Java advanced imaging* API.

actor.lib.javasound

This package provides sound actors.

actor.lib.jmf This is a library of image processing actors based on the *Java media framework*

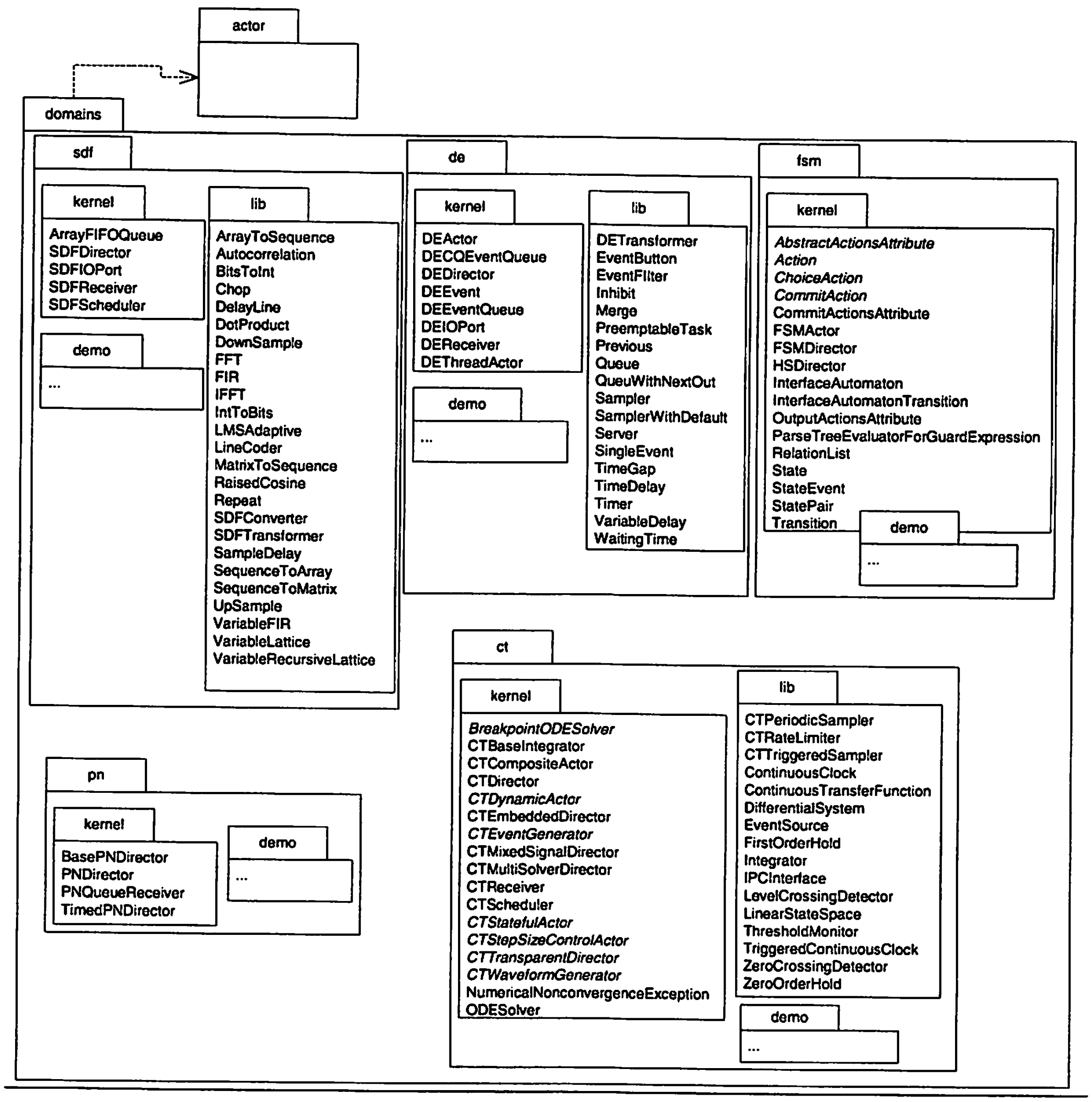


FIGURE 1.9. Package structure of common Ptolemy II domains.

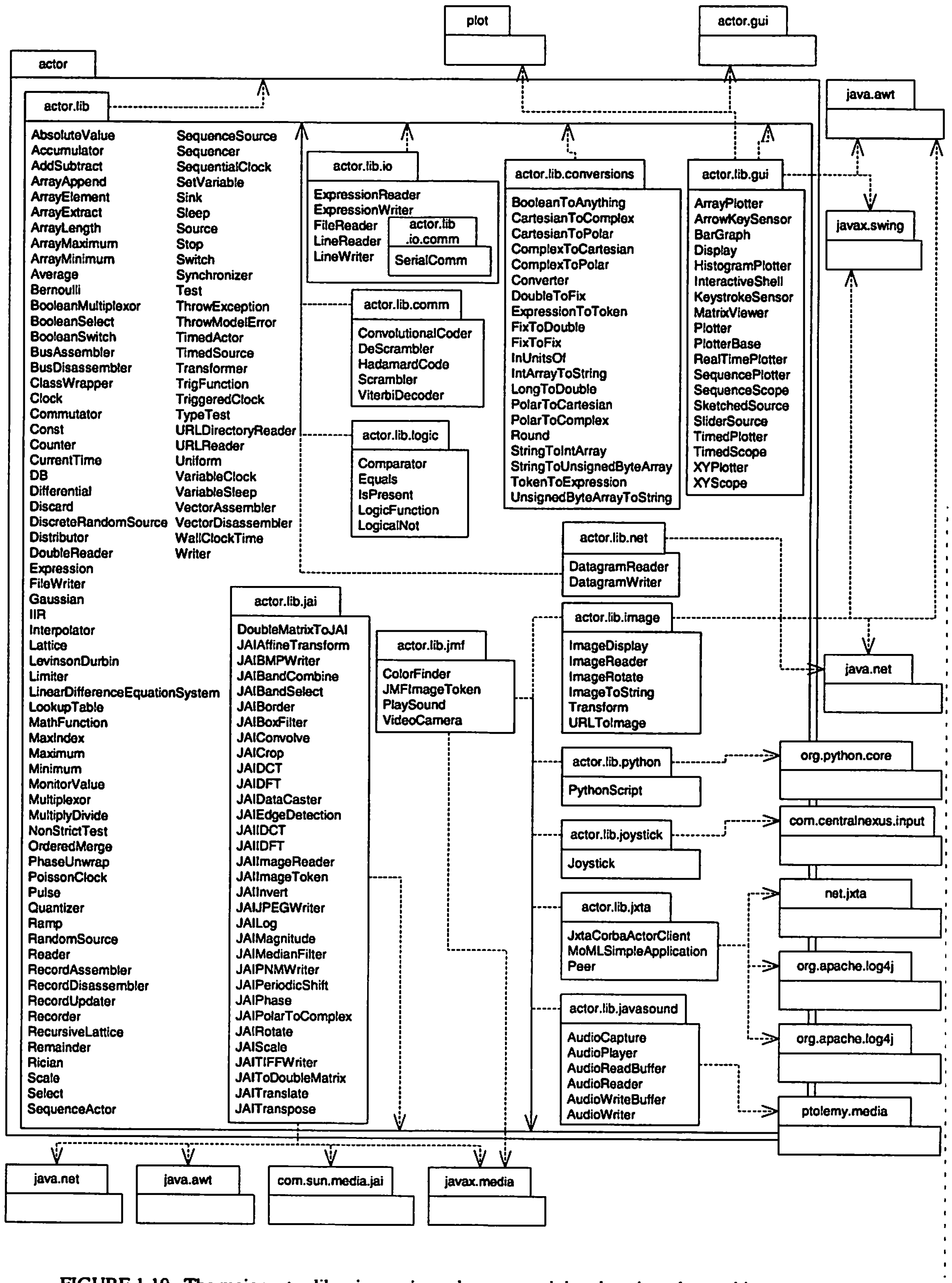


FIGURE 1.10. The major actor libraries are in packages containing domain-polymorphic actors.

	API.
actor.lib.joystick	This package provides an example actor that communicates with a particular I/O device, a joystick.
actor.lib.jxta	This is a library of experimental actors supporting the JXTA discovery mechanism from Sun Microsystems.
actor.lib.logic	This package provides actors that perform logical functions like AND, OR and NOT.
actor.lib.net	This package provides actors that communicate using datagrams.
actor.lib.python	This package provides an actor whose operation can be specified in Python.

1.6.5 User Interface Packages

The UI packages provide support for our XML file format, called MoML, and a visual interface for constructing models graphically, called Vergil. These packages are organized as shown in figures 1.11 and 1.12. The intent of each package is described below:

actor.gui	This package contains the configuration infrastructure, which supports modular construction of user interfaces that are themselves Ptolemy II models.
actor.gui.style	This package contains classes that decorate attributes to serve as hints to a user interface about how to present these attributes to the user.
gui	This package contains generically useful user interface components.
media	This package encapsulates a set of classes supporting audio and image processing.
moml	This package contains classes support our XML modeling markup language (MoML), which is used to describe Ptolemy II models.
moml.filter	This package provides backward compatibility between Ptolemy release. We hope to replace it with an XSL based solution in a future release.
plot	This package and its packages provides two-dimensional signal plotting widgets.
vergil	This package and its packages contains the Ptolemy II graphical user interface. It builds on Diva, a toolkit that extends Java 2D. For more information about Diva, see http://www.gigascale.org/diva

1.6.6 Capabilities

Ptolemy II is a third generation system. Its immediate predecessor, Ptolemy Classic, still has active users and developers, particularly through a commercial product that is based partly on it, Agilent's ADS. Ptolemy II has a somewhat different emphasis, and through its use of Java, concurrency, and integration with the network, is aggressively experimental. Some of the major capabilities in Ptolemy II that we believe to be new technology in modeling and design environments include:

- *Higher level concurrent design in Java^{TM}*. Java support for concurrent design is very low level, based on threads and monitors. Maintaining safety and liveness can be quite difficult [60]. Ptolemy II includes a number of domains that support design of concurrent systems at a much higher level

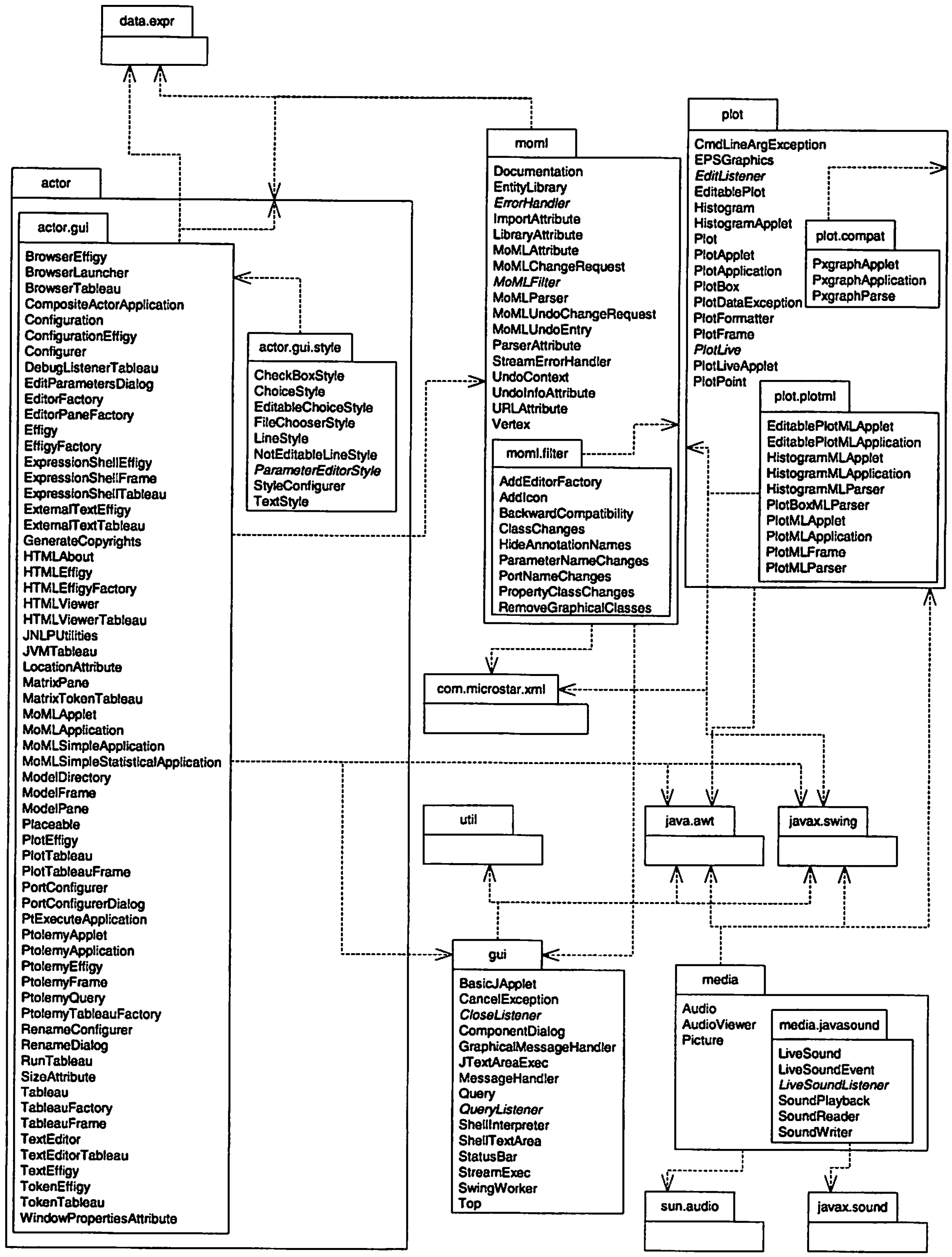


FIGURE 1.11. Packages in Ptolemy II that support the user interfaces, including the MoML XML schema, plotters and other display infrastructure, and support for windows and application configurations.

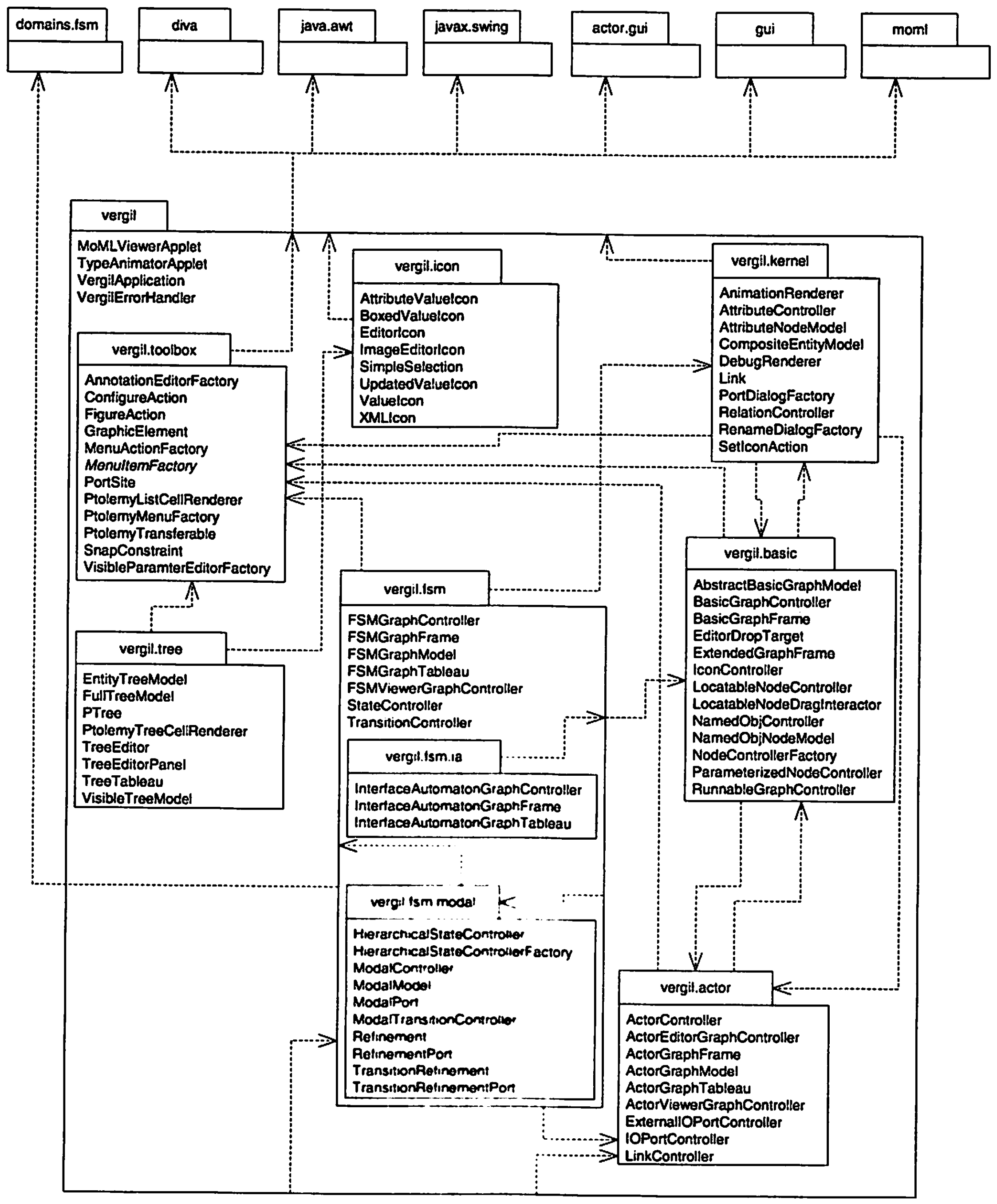


FIGURE 1.12. Packages in Ptolemy II provide the Vergil visual editor.

of abstraction, at the level of their software architecture. Some of these domains use Java threads as an underlying mechanism, while others offer an alternative to Java threads that is much more efficient, scalable, and understandable.

- *Better modularization through the use of packages.* Ptolemy II is divided into packages that can be used independently and distributed on the net, or drawn on demand from a server. This breaks with tradition in design software, where tools are usually embedded in huge integrated systems with interdependent parts.
- *Complete separation of the abstract syntax from the semantics.* Ptolemy designs are structured as clustered graphs. Ptolemy II defines a clean and thorough abstract syntax for such clustered graphs, and separates into distinct packages the infrastructure supporting such graphs from mechanisms that attach semantics (such as dataflow, analog circuits, finite-state machines, etc.) to the graphs.
- *Improved heterogeneity via a well-defined abstract semantics.* Ptolemy Classic provided a wormhole mechanism for hierarchically coupling heterogeneous models of computation. This mechanism is improved in Ptolemy II through the use of opaque composite actors, which provide better support for models of computation that are very different from dataflow, the best supported model in Ptolemy Classic. These include hierarchical concurrent finite-state machines and continuous-time modeling techniques.
- *Thread-safe concurrent execution.* Ptolemy models are typically concurrent, but in the past, support for concurrent execution of a Ptolemy model has been primitive. Ptolemy II supports concurrency throughout, allowing for instance for a model to mutate (modify its clustered graph structure) while the user interface simultaneously modifies the structure in different ways. Consistency is maintained through the use of monitors and read/write semaphores [51] built upon the lower level synchronization primitives of Java.
- *A software architecture based on object modeling.* Since Ptolemy Classic was constructed, software engineering has seen the emergence of sophisticated object modeling [89][110][113] and design pattern [35] concepts. We have applied these concepts to the design of Ptolemy II, and they have resulted in a more consistent, cleaner, and more robust design. We have also applied a simplified software engineering process that includes systematic design and code reviews [107].
- *A truly polymorphic type system.* Ptolemy Classic supported rudimentary polymorphism through the "anytype" particle. Even with such limited polymorphism, type resolution proved challenging, and the implementation is ad-hoc and fragile. Ptolemy II has a more modern type system based on a partial order of types and monotonic type refinement functions associated with functional blocks. Type resolution consists of finding a fixed point, using algorithms inspired by the type system in ML [92]. The type system is described in [124] and [125].
- *Domain-polymorphic actors.* In Ptolemy Classic, actor libraries were separated by domain. Through the notion of subdomains, actors could operate in more than one domain. In Ptolemy II, this idea is taken much further. Actors with intrinsically polymorphic functionality can be written to operate in a much larger set of domains. The mechanism they use to communicate with other actors depends on the domain in which they are used. This is managed through a concept that we call a *process level type system.*
- *Extensible XML-based file formats.* XML is an emerging standard for representation of information that focuses on the logical relationships between pieces of information. Human-readable representations are generated with the help of style sheets. Ptolemy II will use XML as its primary format for persistent design data.

- *Distributed models.* Ptolemy II has (still preliminary) infrastructure supporting distributed modeling using CORBA, Java RMI, or lower-level networking primitives. Ptolemy II has (still preliminary) support for migrating software components.
- *Component specialization.* Ptolemy II has an evolving code generation mechanism that is very different from that in Ptolemy Classic. In Ptolemy Classic, each component has to have a definition in the target language, and the code generator merely stitches together these components. In Ptolemy II, components are defined in Java, and the Java definition is parsed. An API for performing optimization transformations on the abstract syntax tree is defined, and then compiler back ends can be used to generate target code. A preliminary implementation of this approach is described in [98], [119] and [120].
- *Fully integrated expression language.* The Ptolemy II expression language is a higher-order, richly expressive language that is fully integrated with actor-oriented modeling. The type system inference mechanism propagates through expressions, parameters, and actor ports seamlessly.

1.6.7 Future Capabilities

Capabilities that we anticipate making available in the future include:

- *Integrated verification tools.* Modern verification tools based on model checking [46] could be integrated with Ptolemy II at least to the extent that finite state machine models can be checked. We believe that the separation of control logic from concurrency will greatly facilitate verification, since only much smaller cross-sections of the system behavior will be offered to the verification tools.
- *Reflection of dynamics.* Java supports reflection of static structure, but not of dynamic properties of process-based objects. For example, the data layout required to communicate with an object is available through the reflection package, but the communication protocol is not. We plan to extend the notion of reflection to reflect such dynamic properties of objects.
- *Meta modeling.* The domains in Ptolemy II are constructed based on an intuitive understanding of a useful class of modeling techniques, and then the support infrastructure for specifying and executing models in the domain are built by hand by writing Java code. Others have built tools that have the potential of improving on this situation by *meta modeling*. In Dome (from Honeywell) and GME (from Vanderbilt), for example, a modeling strategy itself is modeled, and user interfaces supporting that modeling strategy are synthesized from that model. We can view the current component-based architecture of Vergil as a starting point in this direction. In the future, we expect to see much more use of Ptolemy II itself to define and construct Ptolemy II domains and their user interfaces.

1.7 Acknowledgements

There have been many participants in the Ptolemy Project over the years. So many, that we may (inadvertently) omit some names here. With apologies to those people, we list the contributors. The principal authors of version 3.0 are

- Christopher Hylands,
- Edward Lee,
- Jie Liu,
- Xiaojun Liu,

- Stephen Neuendorffer,
- Yuhong Xiong,
- Yang Zhao, and
- Haiyang Zheng.

Major contributors to earlier versions of Ptolemy II include:

- John Reekie,
- Chamberlain Fong,
- John Davis, II,
- Mudit Goel,
- Bilung Lee,
- Lukito Muliadi,
- Neil Smyth, and
- Jeff Tsay.

Other contributors include Vincent Arnould, Frederic Boulanger, Adam Cataldo, Chris Chang, Albert Chen, Elaine Cheong, Brieuc Desoutter, Pedro Domecq, Johan Eker, Geroncio Galicia, Ben Horowitz, Heloise Hse, Jörn Janneck, Zoltan Kemenczy, Bart Kienhuis, Christoph Meyer Kirsch, Sanjeev Kohli, Robert Kroeger, David Lee, Michael Leung, John Li, Andrew Mihal, Eleftherios Matsikoudis, Aleksandar Necakov, Sarah Packman, Shankar Rao, Sonia Sachs, Michael Shilman, Sean Simmons, Mandeep Singh, Peter N. Steinmetz, Dick Stevens, Mary Stewart, Ned Stoffel, Manda Sutijono, Neil Turner, Guillaume Vibert, Kees Vissers, Brian K. Vogel, Yuke Wang, Xavier Warzee, Scott Weber, Paul Whitaker, Winthrop Williams, Ed Willink, Michael Wirthlin, William Wu, Paul Yang, James Yeh, Nick Zamora, and Rachel Zhou.

The Ptolemy Project has received financial support the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation, Chess (the Center for Hybrid and Embedded Software Systems), the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of California MICRO program, and the following companies: Agilent, Atmel, Cadence, Hewlett-Packard, Hitachi, National Semiconductor, and Philips.

Appendix A: UML — Unified Modeling Language

UML (the unified modeling language) [33][106] defines a suite of visual syntaxes for describing various aspects of software architecture. We make heavy use of two of these visual syntaxes, package diagrams and static structure diagrams. These syntaxes are summarized here. As with most descriptive syntaxes, any use of the syntax involves certain stylistic choices. These stylistic choices are not part of UML, but nonetheless can be important to understanding the diagrams. We explain the style that we use here.

A.1 Package Diagrams

Figures 1.7 and 1.9 show UML *package diagrams*, which have a simple syntax. A package is given as a box with a tab, with the tab containing the name of the package. Subpackages are enclosed in the box of the parent package, and package dependencies are indicated with arrows. A package dependency occurs when a Java file in a package includes a class in another package (using import in Java).

A.2 Static Structure Diagrams

Figure 1.8 is a different kind of UML diagram, called a *static structure diagram* or *class diagram*. It represents the relationships between classes, including inheritance relationships, containment relationships, and cross references. These relationships are called an *object model*, and represent many essential features about the design.

A.2.1 Classes

A simplified static structure diagram for some Ptolemy II classes is shown in figure 1.13. In this diagram, each class is shown in a box. The class name is at the top of each box, its *attributes* are below that, and its methods below that. Thus, each box is divided into three segments separated by horizontal lines. The attributes are members of the Java classes, which may be public, package friendly, protected, or private. Private members are prefixed by a minus sign "-", as for example the _container attribute of Port. Although private members are not visible directly to users of the class, they may nonetheless be a useful part of the object model because they indicate the state information contained by an instance of the class. Public members have a leading "+" and protected methods a leading "#" in a UML diagram. There are no public or protected members shown in figure 1.13. The type of a member is indicated after a colon, so for example, the _container method of Port is of type Entity.

Methods, which are shown below attributes, also have a leading "+" for public, "#" for protected, and "-" for private. Our object models do not show private methods, since they are not inherited and are not visible in the interface to the object. Figure 1.13 shows a number of public methods and one protected method, _link() in Port. The return value of a method is given after a colon, so for example, getContainer() of Port returns an Entity.

Although not usually included in UML diagrams, our diagrams show class constructors. They are listed first among the methods and have names that are the same as the name of the class. No return type is shown. For completeness, our object models typically show all public and protected methods of these classes, although a proper object model might only show those relevant to the issues being discussed. Figure 1.13 does not show all methods, so that we can simplify the discussion of UML. Our diagrams do not include deprecated methods or methods that are present in parent classes.

Arguments to a method or constructor are shown in parentheses, with the types after a colon, so for example, ComponentEntity shows a single constructor that takes two arguments, one of type CompositeEntity and the other of type String.

A.2.2 Inheritance

Subclasses are indicated by lines with white triangles (or outlined arrow heads). The class on the side of the arrow head is the *superclass* or *base class*. The class on the other end is the *subclass* or *derived class*. The derived class is said to *specialize* the base class, or conversely, the base class to *generalize* the derived class. The derived class *inherits* all the methods shown in the base class and may *override* or some of them. In our object models, we do not explicitly show methods that override those defined in a base class or are inherited from a base class. For example, in figure 1.13, ComponentEntity has all the methods of Entity and NamedObj, and may override some of those methods, but only

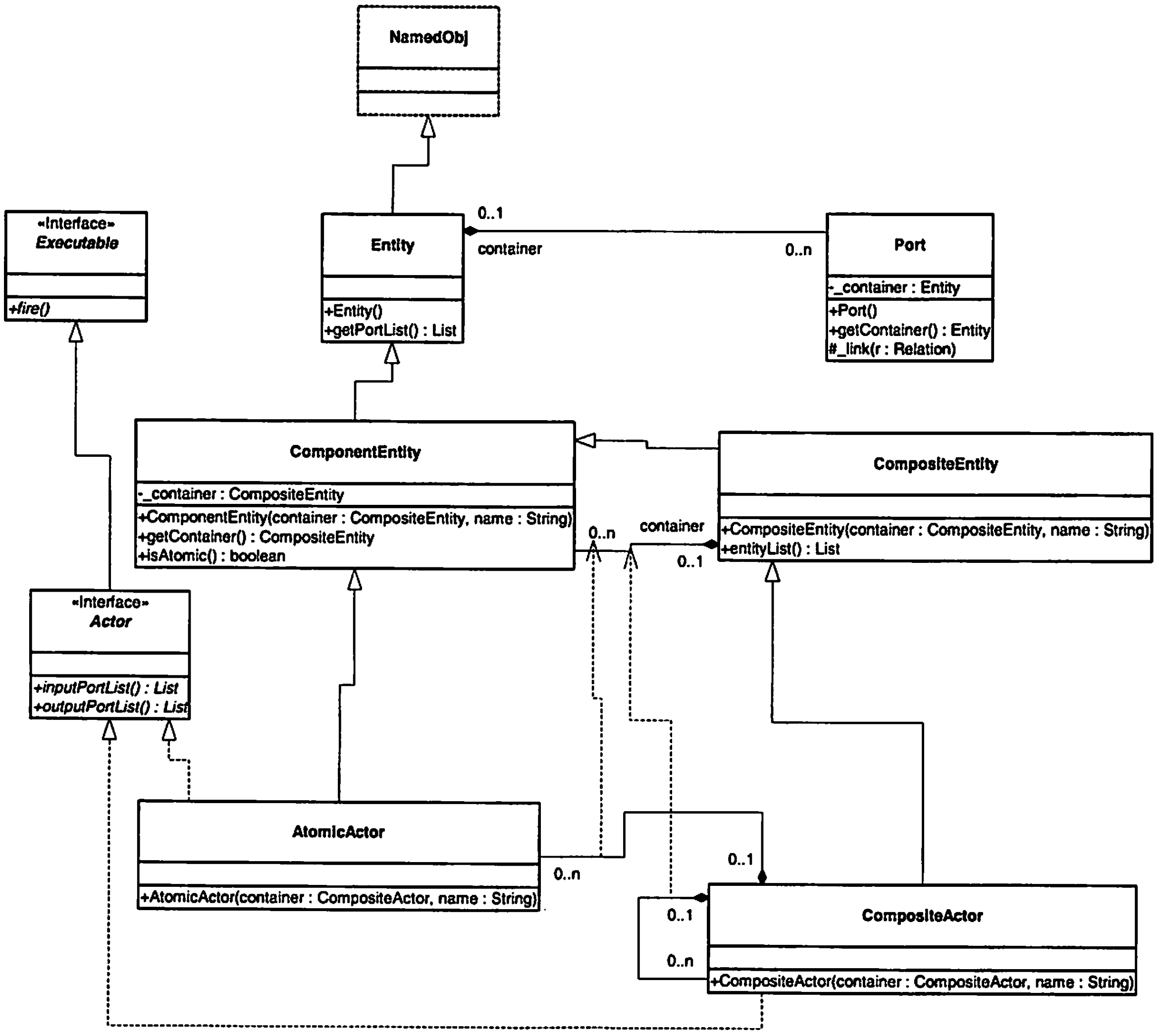


FIGURE 1.13. Simplified static structure diagram for some Ptolemy II classes. This diagram illustrates features of UML syntax that we use.

shows the one method it adds. Thus, the complete set of methods of a class is cumulative. Every class has its own methods plus those of all its superclasses.

An exception to this is constructors. In Java, constructors are not inherited. Thus, in our class diagrams, the only constructors available for a class are those shown in the box defining the class. Figure 1.13 does not show all the constructors of these classes, for simplicity.

Classes shown in boxes outlined with dashed lines, such as NamedObj in figure 1.13, are fully described elsewhere. This is not standard UML notation, but it gives us a convenient way to partition diagrams. Often, these classes belong to another package.

A.2.3 Interfaces

Figure 1.13 also shows two examples of *interfaces*, Executable and Actor. An interface is indicated by the label "<<Interface>>" and by italics in the name. An interface defines a set of methods without providing an implementation for them. It cannot be instantiated, and therefore has no constructors. When a class *implements* an interface, the object model shows the relationship with a dotted-line with an arrow. Any *concrete class* (one that can be instantiated) that implements an interface must provide implementations of all its methods. In our object models, we do not show those methods explicitly in the concrete class, just like inherited methods, but their presence is implicit in the relationship to the interface.

One interface can extend another. For example, in figure 1.13, Actor extends Executable. This means that any concrete class that implements Actor must implement the methods of Actor and Executable.

We will occasionally show *abstract classes*, which are like interfaces in that they cannot be instantiated, but unlike interfaces in that they may provide default implementations for some methods and may even have private members. Abstract classes are indicated by italics in the class name. There are no abstract classes in figure 1.13.

A.2.4 Associations

Inheritance and implementation are types of *associations* between entities in the object model. Associations of other types are indicated by other lines, often annotated with ranges like "0..n" or with diamonds on one end or the other.

Aggregations are shown as associations with diamonds. For example, an Entity is an aggregation of any number (0..n) instances of Port. More strongly, we say that a Port is *contained* by 0 or 1 instances of Entity. By containment, we mean that a port can only be contained by a single Entity. In a weaker form of aggregation, more than one aggregate may refer to the same component. The stronger form of aggregation (containment) is indicated by the filled diamond, while the weaker form is indicated by the unfilled diamond. There are no unfilled diamonds in figure 1.13. In fact, they are fairly rare in Ptolemy II, since many of its architectural features depend on containment relationships, where an object can have at most one container.

The relationship between ComponentEntity and CompositeEntity is particularly interesting. An instance of CompositeEntity can contain any number of instances of ComponentEntity, but CompositeEntity is derived from ComponentEntity. Thus, a CompositeEntity can contain any number of instances of either ComponentEntity or CompositeEntity. This is the classic Composite design pattern [35], which supports arbitrarily deeply nested containment hierarchies.

In figure 1.13, a CompositeActor is an aggregation of AtomicActors and CompositeActors. These two aggregation relations are derived from the aggregation relationship between ComponentEntity and CompositeEntity. This derived association is indicated with a dashed line with an open arrowhead.

Appendix B: Ptolemy II Naming Conventions

We have made an effort to be consistent about naming of classes, methods and members. This appendix describes our policy.

B.1 Classes

Class names are capitalized with internal word boundaries also capitalized (as in "CompositeEntity"). Most names are made up of complete words ("CompositeEntity" rather than "CompEnt")[1]. Interface names suggest their potential (as in "Executable," which means "can be executed").

Despite having packages to divide up the namespace, we attempt nonetheless to keep class names unique. This helps avoid confusion and bugs that may arise from having Java import statements in the wrong order. In many cases, a domain includes a specialized version of some more generic class. In this case, we create a unique name by prefixing the generic name with the domain name. For example, while Director is a base class in the actor package, DEDirector is a derived class in the DE domain.

For the most part, we try to avoid prefixing actor names with the domain name. e.g., we define Delay rather than DEDelay. Occasionally however, the domain prefix is useful to distinguish two versions of some similar functionality, both of which might be useful in a domain. For example, the DE domain can use actors derived from Transformer or from DETransformer, where the latter is specialized to DE.

B.2 Members

Member names are not capitalized, although internal word boundaries usually are (e.g. "declaredType"). If the member is private or protected, then its name begins with a leading underscore (e.g. "_declaredType").

B.3 Methods

Method names are similar to member names, in that they are not capitalized, except on internal word boundaries. Private and protected methods have a leading underscore. In text referring to methods, the method name is followed by open and close parentheses, as in "getName()." Usually, no arguments are given, even if the method takes arguments.

Method names that are plural, such as insideRelations(), usually return an enumeration (or sometimes an array, or an iterator). Methods that return Lists are usually of the form portList().

1. There are some (perhaps regrettable) exceptions to this, such as NamedObj.

2

Using Vergil

Authors: *Edward A. Lee*
Steve Neuendorffer

2.1 Introduction

There are many ways to use Ptolemy II. It can be used as a framework for assembling software components, as a modeling and simulation tool, as a block-diagram editor, as a system-level rapid prototyping application, as a toolkit supporting research in component-based design, or as a toolkit for building Java applications. This chapter introduces its use as a modeling and simulation tool.

In this chapter, we describe how to graphically construct models using Vergil, a graphical user interface (GUI) for Ptolemy II. Figure 2.1 shows a simple Ptolemy II model in Vergil, showing the graph editor, one of several editors available in Vergil. Keep in mind as you read this document that graphical entry of models is only one of several possible entry mechanisms available in Ptolemy II. Moreover, only some of the execution engines (called *domains*) are described here. A major emphasis of Ptolemy II is to provide a framework for the construction of modeling and design tools, so the specific modeling and design tools described here should be viewed as representative of our efforts.

2.2 Quick Start

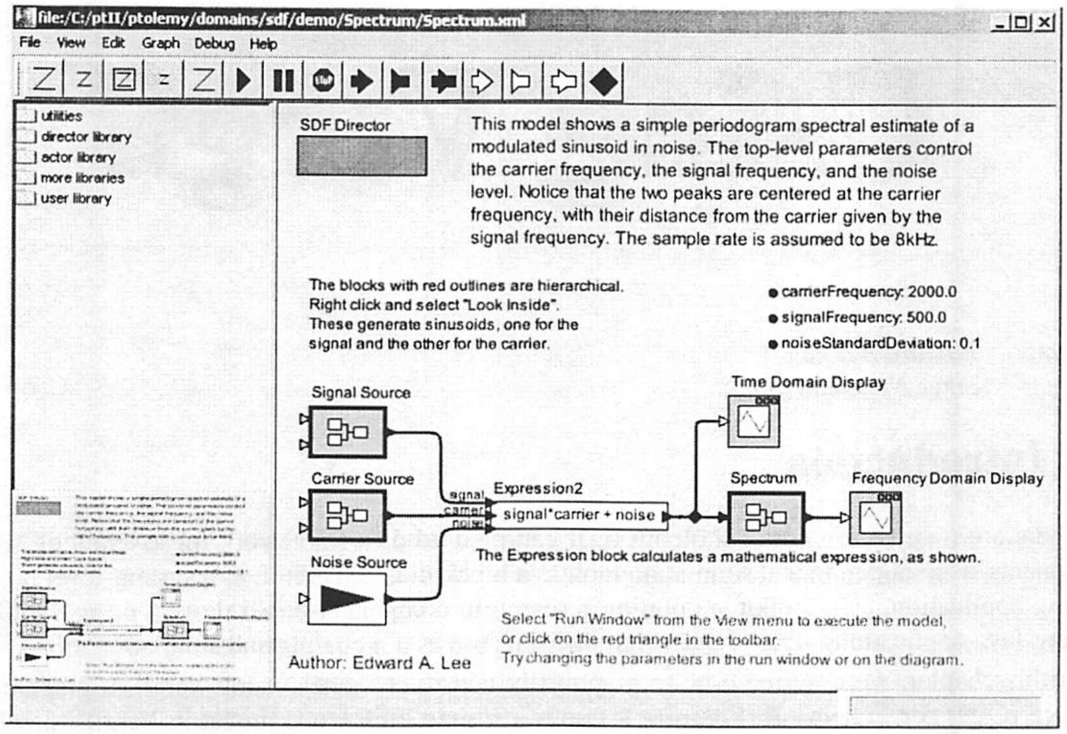
This section shows how to start Vergil, how to execute and explore pre-built models, and how to construct your own models.

2.2.1 Starting Vergil

First start Vergil. From the command line, enter "vergil", or select Vergil or Ptolemy II in the Start menu, or click on a Web Start link on a web page supporting the web edition. You should see an initial welcome window that looks like the one in figure 2.2. Feel free to explore the links in this window. The "Quick tour" link takes you to the page shown in figure 2.3.

2.2.2 Executing a Pre-Built Model: A Signal Processing Example

The very first example on the quick tour page is the model shown in figure 2.1, which creates a sinusoidal signal, multiplies it by a sinusoidal carrier, and adds noise. You can execute this model in either of two ways. First, you can select Run Window in the View menu, and then click on Go. The result is shown in figure 2.4. The upper plot shows the spectrum of the time-domain signal shown in the lower plot. Note the four peaks, which indicate the modulated sinusoid. In the run window you can



This model shows a simple periodogram spectral estimate of a modulated sinusoid in noise. The top-level parameters control the carrier frequency, the signal frequency, and the noise level. Notice that the two peaks are centered at the carrier frequency, with their distance from the carrier given by the signal frequency. The sample rate is assumed to be 8kHz.

The blocks with red outlines are hierarchical. Right click and select "Look Inside". These generate sinusoids, one for the signal and the other for the carrier.

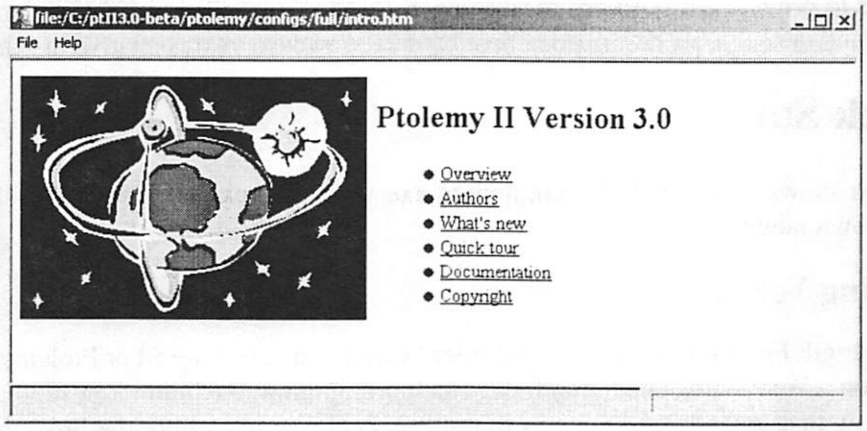
- carrierFrequency: 2000.0
- signalFrequency: 500.0
- noiseStandardDeviation: 0.1

The Expression block calculates a mathematical expression, as shown.

Select "Run Window" from the View menu to execute the model, or click on the red triangle in the toolbar.
Try changing the parameters in the run window or on the diagram.

Author: Edward A. Lee

FIGURE 2.1. Example of a Vergil window.



Ptolemy II Version 3.0

- Overview
- Authors
- What's new
- Quick tour
- Documentation
- Copyright

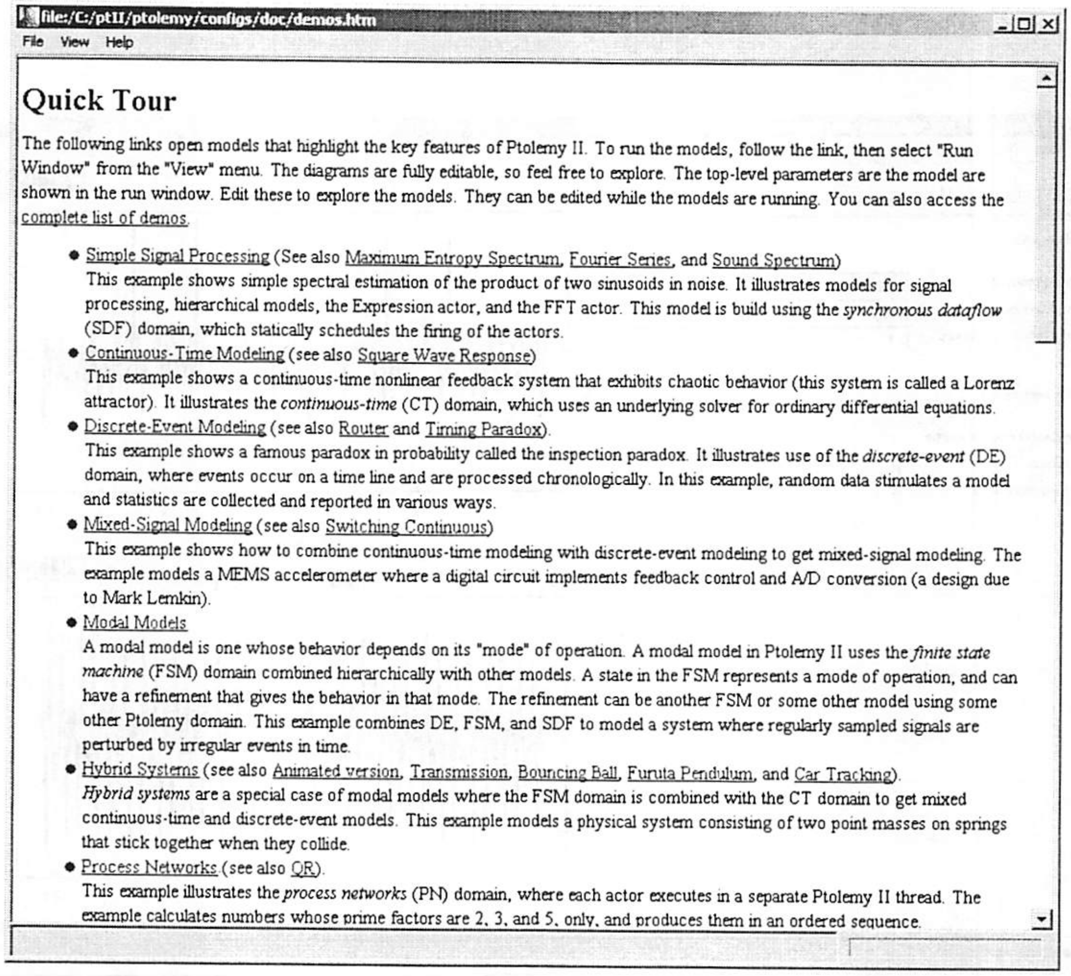
FIGURE 2.2. Initial welcome window.

adjust the frequencies of the signal and the carrier as well as the amount of noise. These can also be adjusted in the block diagram in figure 2.1 by double clicking on the bulleted parameters near the upper right of the window.

The second alternative for running the model is to click on the run button in the toolbar, which is indicated by a red triangle pointing to the right. If you use this alternative, then the two signal plots are displayed in their own windows.

You can study the way the model is constructed in figure 2.1. Note the Expression actor in the middle, whose icon indicates the expression being calculated: "`signal*carrier + noise`". The identifiers in this expression, `signal`, `carrier`, and `noise` refer to the input ports by name. The names of these ports are shown in the diagram. The Expression actor is a very flexible actor in the Ptolemy II actor library. It can have any number of input ports, with arbitrary names, and uses a rich and expressive expression language to specify the value of the output as a function of the inputs (and parameters of the containing model, if desired).

Three of the actors in figure 2.1 are *composite actors*, which means that their implementation is itself given as a block diagram. Composite actors are indicated visually by the red outline. You can look inside to reveal the implementation, as shown in figure 2.5, which shows the implementation of



Quick Tour

The following links open models that highlight the key features of Ptolemy II. To run the models, follow the link, then select "Run Window" from the "View" menu. The diagrams are fully editable, so feel free to explore. The top-level parameters are the model are shown in the run window. Edit these to explore the models. They can be edited while the models are running. You can also access the complete list of demos.

- Simple Signal Processing (See also Maximum Entropy Spectrum, Fourier Series, and Sound Spectrum)
  This example shows simple spectral estimation of the product of two sinusoids in noise. It illustrates models for signal processing, hierarchical models, the Expression actor, and the FFT actor. This model is build using the *synchronous dataflow* (SDF) domain, which statically schedules the firing of the actors.
- Continuous-Time Modeling (see also Square Wave Response)
  This example shows a continuous-time nonlinear feedback system that exhibits chaotic behavior (this system is called a Lorenz attractor). It illustrates the *continuous-time* (CT) domain, which uses an underlying solver for ordinary differential equations.
- Discrete-Event Modeling (see also Router and Timing Paradox).
  This example shows a famous paradox in probability called the inspection paradox. It illustrates use of the *discrete-event* (DE) domain, where events occur on a time line and are processed chronologically. In this example, random data stimulates a model and statistics are collected and reported in various ways.
- Mixed-Signal Modeling (see also Switching Continuous)
  This example shows how to combine continuous-time modeling with discrete-event modeling to get mixed-signal modeling. The example models a MEMS accelerometer where a digital circuit implements feedback control and A/D conversion (a design due to Mark Lemkin).
- Modal Models
  A modal model is one whose behavior depends on its "mode" of operation. A modal model in Ptolemy II uses the *finite state machine* (FSM) domain combined hierarchically with other models. A state in the FSM represents a mode of operation, and can have a refinement that gives the behavior in that mode. The refinement can be another FSM or some other model using some other Ptolemy domain. This example combines DE, FSM, and SDF to model a system where regularly sampled signals are perturbed by irregular events in time.
- Hybrid Systems (see also Animated version, Transmission, Bouncing Ball, Furuta Pendulum, and Car Tracking).
  Hybrid systems are a special case of modal models where the FSM domain is combined with the CT domain to get mixed continuous-time and discrete-event models. This example models a physical system consisting of two point masses on springs that stick together when they collide.
- Process Networks (see also OR).
  This example illustrates the *process networks* (PN) domain, where each actor executes in a separate Ptolemy II thread. The example calculates numbers whose prime factors are 2, 3, and 5, only, and produces them in an ordered sequence.

FIGURE 2.3. The quick-tour page.

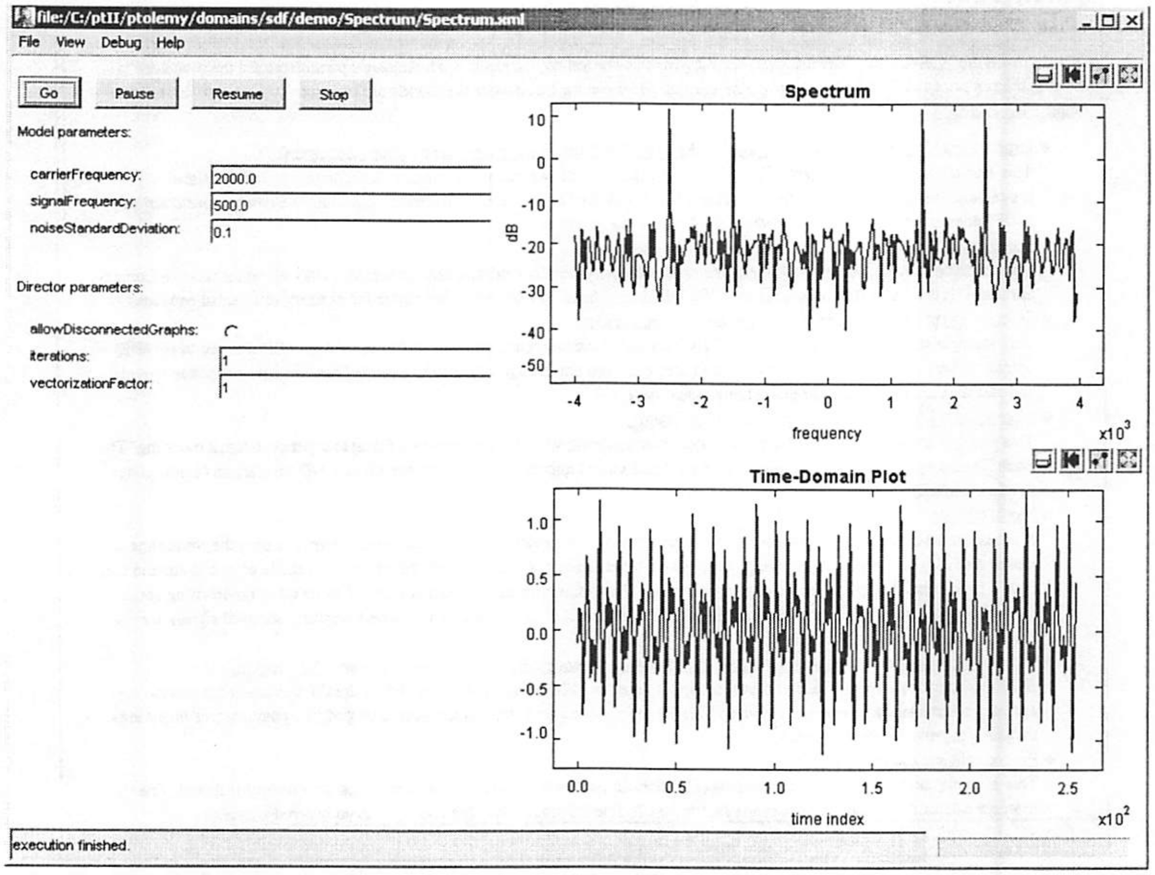
the Signal Source in figure 2.1. It is evident from the block diagram how a sinusoidal signal is generated.

2.2.3 Executing a Pre-Built Model: A Continuous-Time Example

A key principle of the Ptolemy II system is that the model of computation that defines the meaning of a block diagram is not built-in, but is rather specified by the *director* component that is included in the model. The box labeled "SDF Director" in figures 2.1 and 2.5 specifies that these block diagrams have *synchronous dataflow* semantics, which is explained further below. The second example in the quick tour of figure 2.3, by contrast, has continuous-time semantics (the one labeled "Continuous-Time Modeling"). The example is the well-known *Lorenz attractor*, a non-linear feedback system that exhibits chaotic behavior.

The Lorenz attractor model, shown in figure 2.1, is a block diagram representation of a set of non-linear ordinary differential equations. The blocks with integration signs in their icons are integrators. At any given time *t*, their output is given by

$$x(t) = x(t_0) + \int_{t_0}^{t} \dot{x}(\tau) d\tau, \tag{1}$$



Model parameters:

carrierFrequency: 2000.0
signalFrequency: 500.0
noiseStandardDeviation: 0.1

Director parameters:

allowDisconnectedGraphs:
iterations: 1
vectorizationFactor: 1

execution finished.

FIGURE 2.4. The run window for the model shown in figure 2.1.

where $x(t_0)$ is the initial state of the integrator, t_0 is the start time of the model, and $\dot{x}$ is the input signal. Note that since the output is the integral of the input, then at any given time, the input is the derivative of the output,

$$\dot{x}(t) = \frac{d}{dt}x(t) \,. \tag{2}$$

Thus, the system describes either an integral equation or a differential equation, depending on which of these two forms you use.

Let the output of the top integrator in figure 2.1 be x_1, the output of the middle integrator be x_2, and the output of the bottom integrator be x_3. Then the equations described by figure 2.1 are

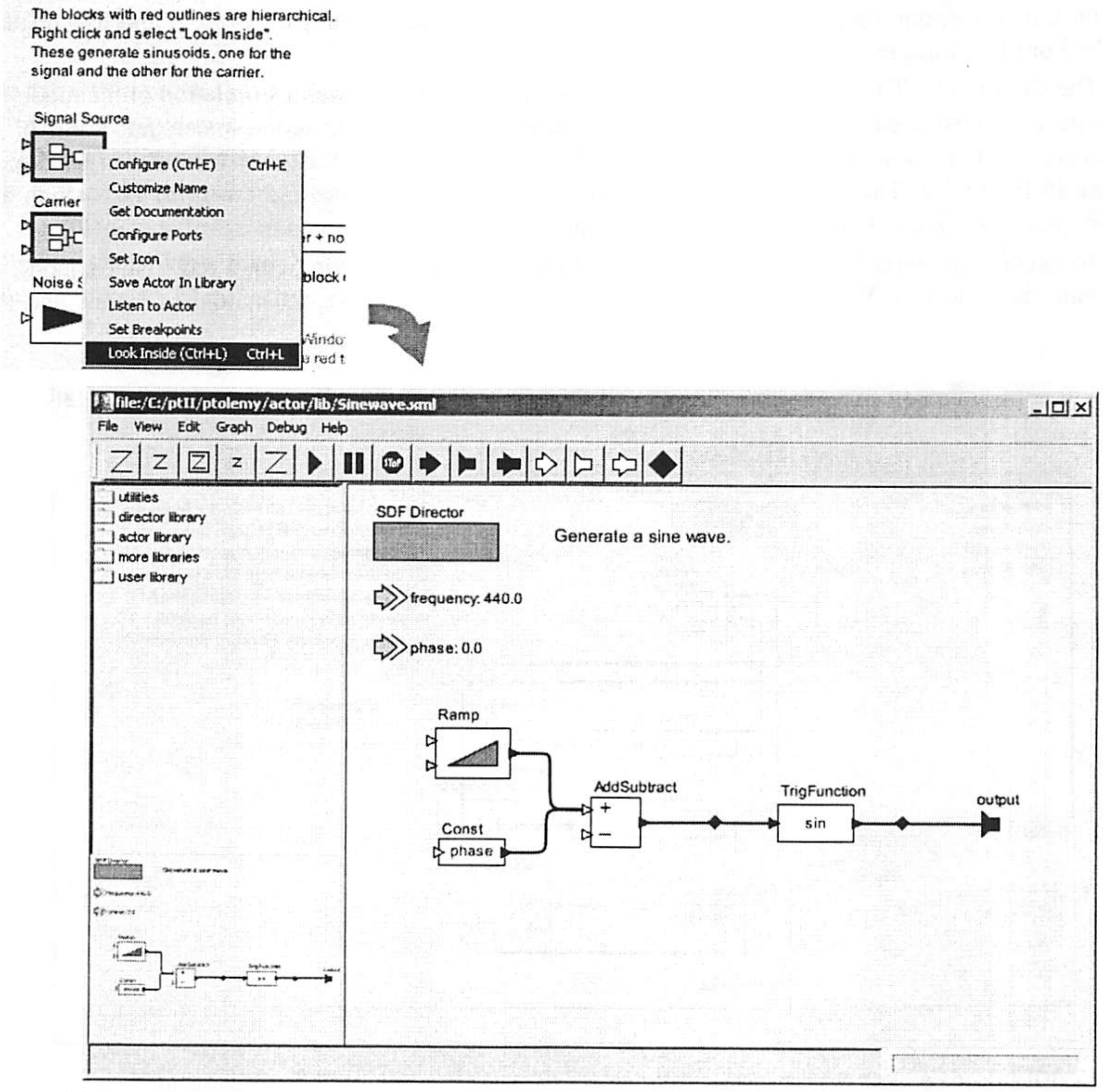


FIGURE 2.5. Look inside composite actors to reveal their implementation.

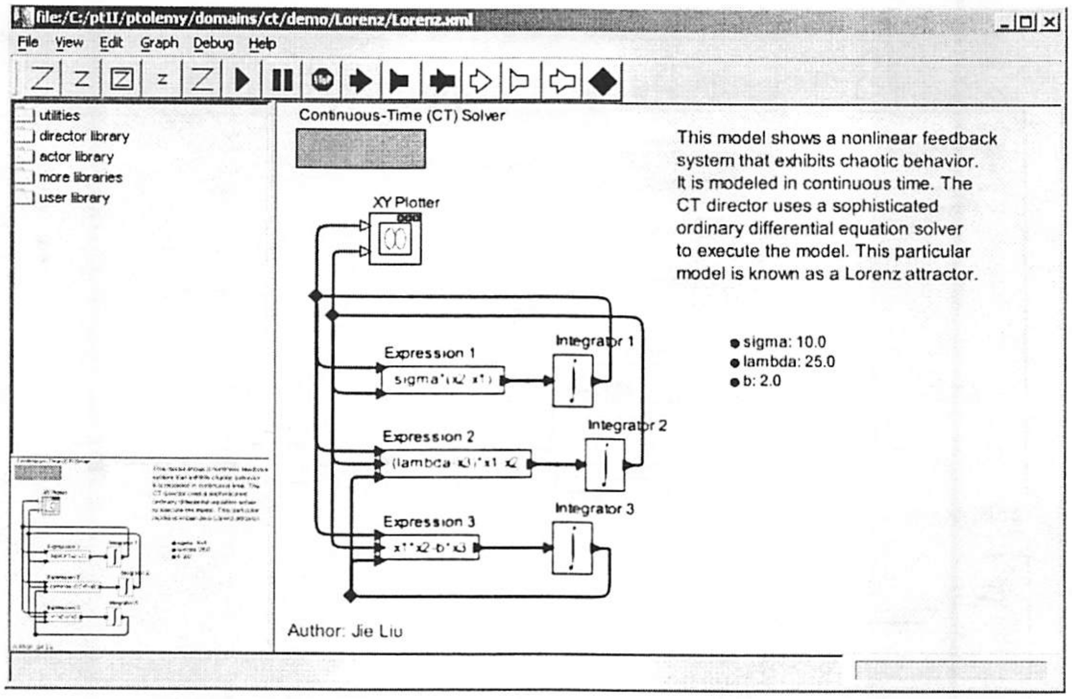
$$\begin{aligned}
\dot{x}_1(t) &= \sigma(x_2(t) - x_1(t)) \\
\dot{x}_2(t) &= (\lambda - x_3(t))x_1(t) - x_2(t)\,. \\
\dot{x}_3(t) &= x_1(t)x_2(t) - bx_3(t)
\end{aligned} \tag{3}$$

For each equation, the expression on the right 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 names of the input ports are not shown in the diagram, but if you linger over them with the mouse cursor, the name will pop up in a tooltip. 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 on the right.

The integrators each also have initial values, which you can examine and change by double clicking on the corresponding integrator icon. These define the initial values of x_1, x_2, and x_3, respectively. For this example, all three are set to 1.0.

The Continuous-Time (CT) Solver, shown at the upper right, manages a simulation of the model. It contains a sophisticated ODE solver, and to use it effectively, you will need to understand some of its parameters. The parameters are accessed by double clicking on solver box, which results in the dialog shown in figure 2.7. The simplest of these parameters are the *startTime* and the *stopTime*, which are self-explanatory. They define the region of the time line over which a simulation will execute.

To execute the model, you can click on the run button in the toolbar (with a red triangle icon), or you can open the Run Window in the View menu. In the former case, the model executes, and the



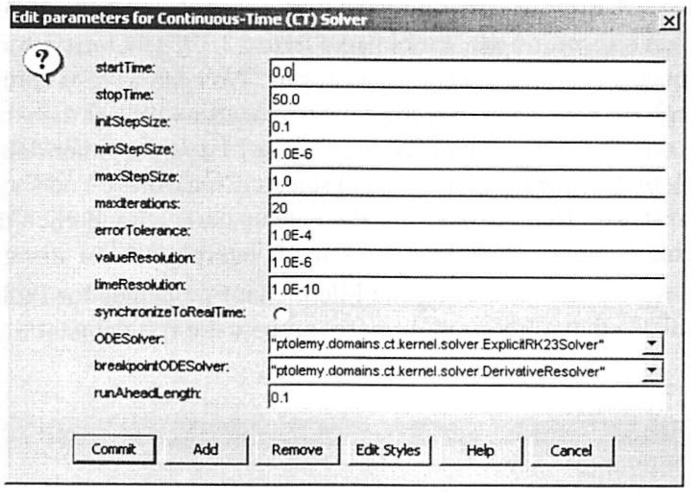
This model shows a nonlinear feedback system that exhibits chaotic behavior. It is modeled in continuous time. The CT director uses a sophisticated ordinary differential equation solver to execute the model. This particular model is known as a Lorenz attractor.

● sigma: 10.0
● lambda: 25.0
● b: 2.0

FIGURE 2.6. A block diagram representation of a set of nonlinear ordinary differential equations.

results are plotted in their own window, as shown in figure 2.8. What is plotted is $x_1(t)$ vs. $x_2(t)$ for values of t in between *startTime* and *stopTime*.

Like the Lorenz model, a typical continuous-time model contains integrators in feedback loops, or more elaborate blocks that realize linear and non-linear dynamical systems given abstract mathematical representations of them (such as Laplace transforms). In the next section, we will explore how to build a model from scratch.


Edit parameters for Continuous-Time (CT) Solver

startTime:	0.0
stopTime:	50.0
initStepSize:	0.1
minStepSize:	1.0E-6
maxStepSize:	1.0
maxIterations:	20
errorTolerance:	1.0E-4
valueResolution:	1.0E-6
timeResolution:	1.0E-10
synchronizeToRealTime:	☐
ODESolver:	"ptolemy.domains.ct.kernel.solver.ExplicitRK23Solver"
breakpointODESolver:	"ptolemy.domains.ct.kernel.solver.DerivativeResolver"
runAheadLength:	0.1

FIGURE 2.7. Dialog box showing solver parameters for the model in figure 2.1.

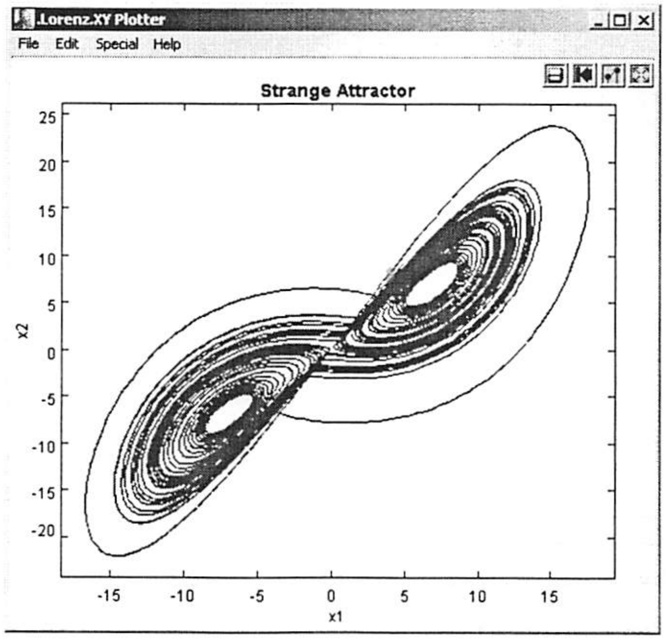


FIGURE 2.8. Result of running the Lorenz model using the run button in the toolbar.

2.2.4 Creating a New Model

Create a new model by selecting File->New->Graph Editor in the welcome window. You should see something like the window shown in figure 2.9. Ignoring the menus and toolbar for a moment, on the left is a palette of objects that can be dragged onto the page on the right. To begin with, the page on the right is blank. Open the *actor library* in the palette, and go into the *sources* library. Find the *Const* actor under *generic sources* and drag an instance over onto the blank page. Then go into the *sinks* library (*generic sinks* sublibrary) and drag a *Display* actor onto the page. Each of these actors can be dragged around on the page. However, we would like to connect one to the other. To do this, drag a connection from the output port on the right of the *Const* actor to the input port of the *Display* actor. Lastly, open the *director library* and drag an *SDF Director* onto the page. The director gives a meaning (semantics) to the graph, but for now we don't have to be concerned about exactly what that is.

Now you should have something that looks like figure 2.10. The *Const* actor is going to create our string, and the *Display* actor is going to print it out for us. We need to take care of one small detail to make it look like figure 2.10: we need to tell the *Const* actor that we want the string "Hello World". To do this we need to edit one of the parameters of the *Const*. To do this, either double click on the *Const* actor icon, or right click on the *Const* actor icon and select "Configure". You should see the dialog box in figure 2.11. Enter the string "Hello World" for the value parameter and click the Commit button. Be sure to include the double quotes, so that the expression is interpreted as a string.

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 you open that file.

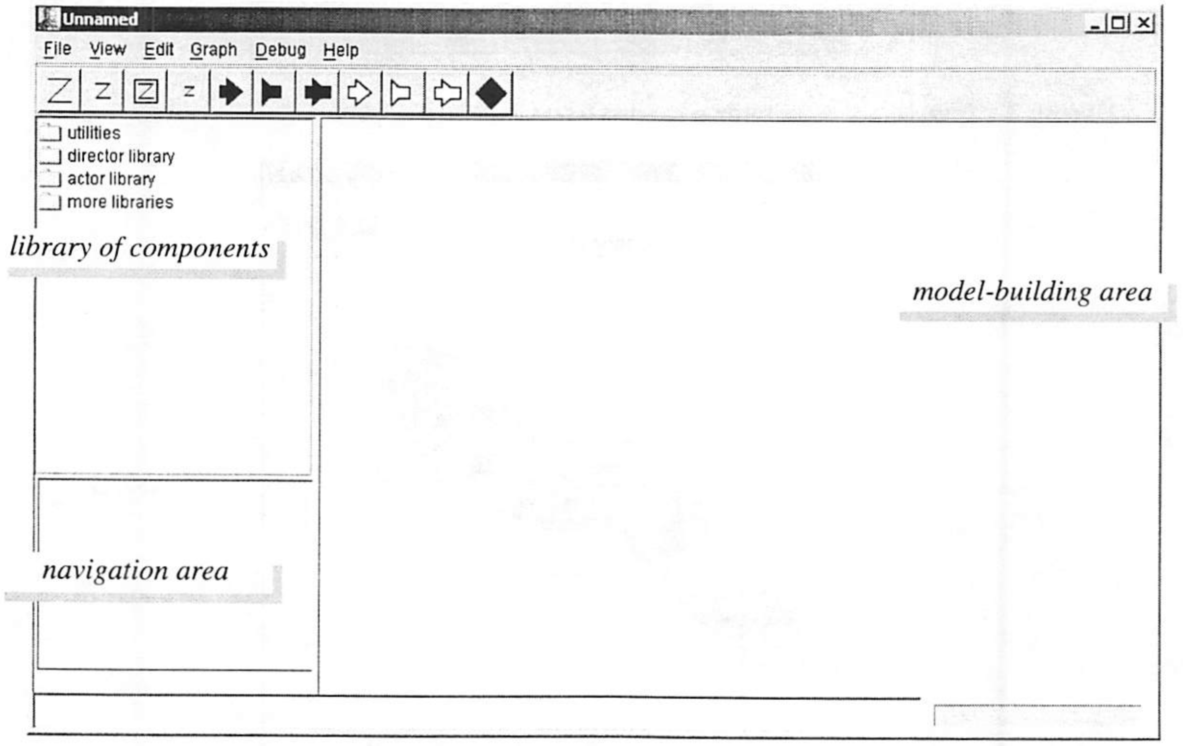


FIGURE 2.9. An empty Vergil Graph Editor.

2.2.5 Running the Model

To run the example, go to the View menu and select the Run Window. If you click the "Go" button, you will see a large number of strings in the display at the right. To stop the execution, click the "Stop" button. To see only one string, change the *iterations* parameter of the SDF Director to 1, which can be done in the run window, or in the graph editor in the same way you edited the parameter of the *Const* actor before. The run window is shown in figure 2.12.

2.2.6 Making Connections

The model constructed above contained only two actors and one connection between them. If you

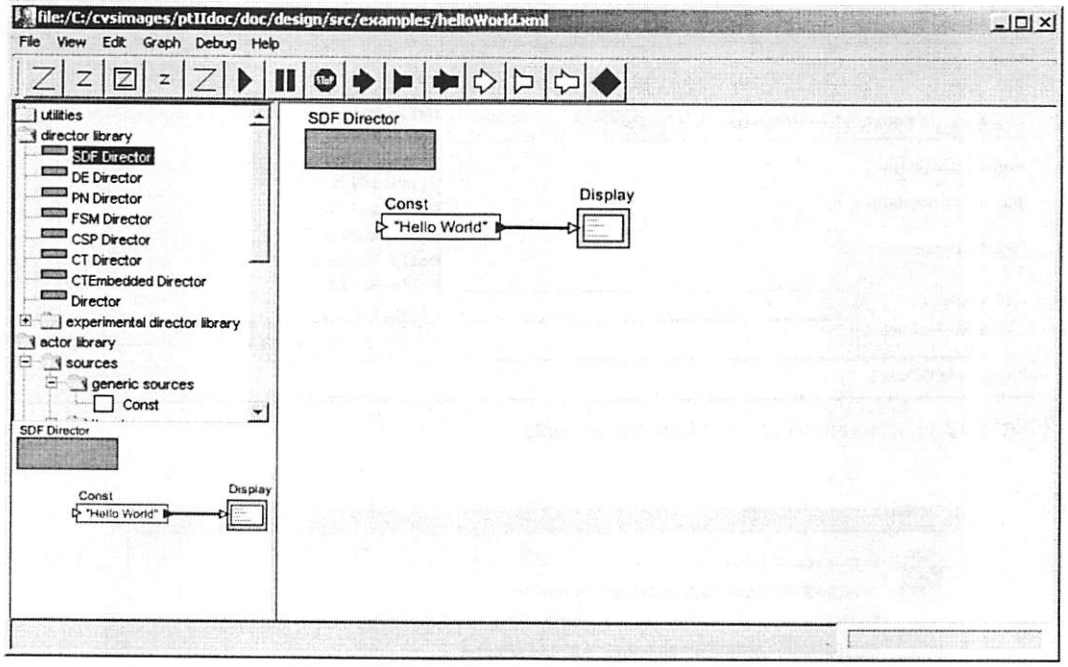


FIGURE 2.10. The Hello World example.

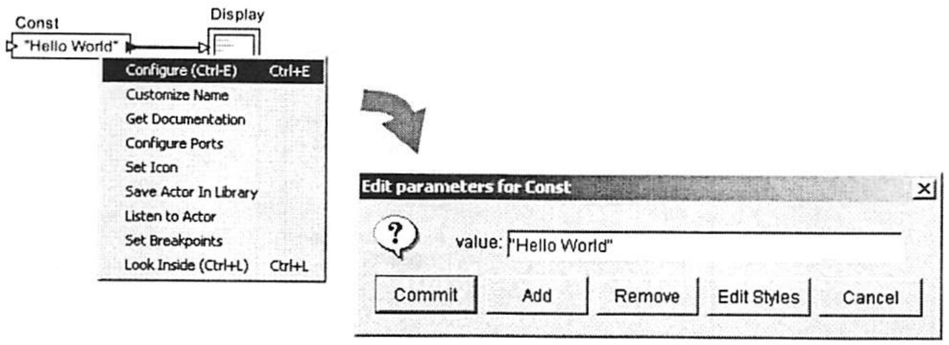
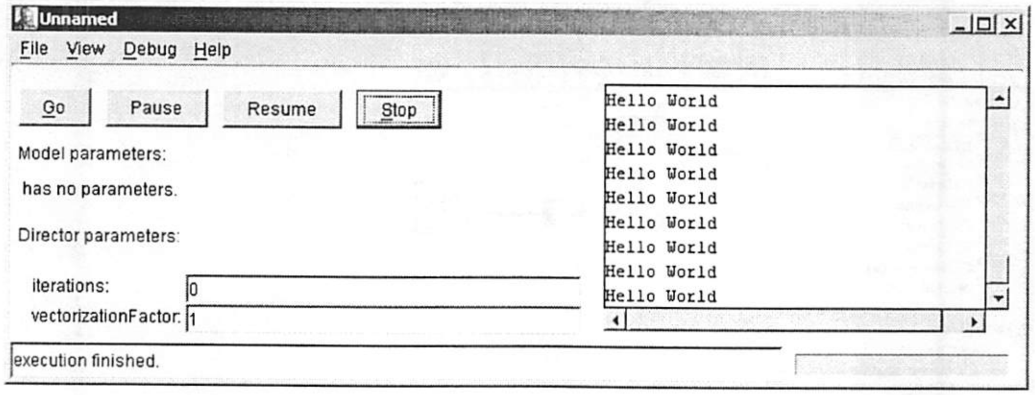


FIGURE 2.11. The Const parameter editor.

move either actor (by clicking and dragging), you will see that the connection is 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 *SDFDirector*, a *Ramp* actor (found in the *sources*) library, a *Display* actor, and a *SequencePlotter* actor, found in the *sinks* library, as shown in figure 2.13. Suppose we wish to route the output of the *Ramp* to both the *Display* and the *Sequence-Plotter*. If we simply attempt to make the connections, we get the exception shown in figure 2.13. Don't panic! Exceptions are normal and common. The key line in this exception report is the last one, which says

```
Attempt to link more than one relation to a single port.
```

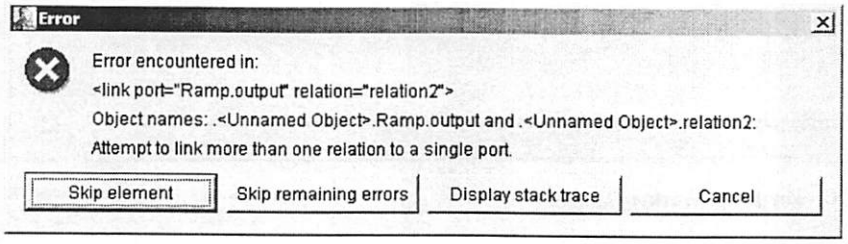


Model parameters:
has no parameters.
Director parameters:
iterations: 0
vectorizationFactor: 1

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

execution finished.

FIGURE 2.12. Execution of the Hello World example.



Error encountered in:
<link port="Ramp.output" relation="relation2">
Object names: .<Unnamed Object>.Ramp.output and .<Unnamed Object>.relation2:
Attempt to link more than one relation to a single port.

FIGURE 2.13. Exception that occurs if you attempt to simply wire the output of the *Ramp* in figure 2.14 to the inputs of the other two actors.

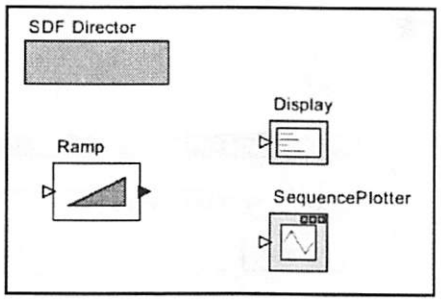


FIGURE 2.14. Three unconnected actors in a model.

The line above that gives the names of the objects involved, which are

```
    Object names:  .<Unnamed Object>.Ramp.output and .<Unnamed
                                              Object>.relation2
```

In Ptolemy II models, all objects have a dotted name. The dots separate elements in the hierarchy. Thus, ".<Unnamed Object>.Ramp.output" is an object named "output" contained by an object named "Ramp", which is contained by an unnamed object (the model itself). The model has no name because we have not assigned one (it acquires a name when we save it).

Why did this exception occur? Ptolemy II supports two distinct flavors of ports, indicated in the diagrams by a filled triangle or an unfilled triangle. The output port of the *Ramp* actor is a *single port*, indicated by a filled triangle, which means that it can only support a single connection. The input port of the *Display* and *SequencePlotter* actors are *multiports*, indicated by unfilled triangles, which means that they can support multiple connections. Each connection is treated as a separate *channel*, which is a path from an output port to an input port (via relations) that can transport a single stream of tokens.

So how do we get the output of the *Ramp* to the other two actors? We need an explicit *relation* in the diagram. A relation is represented in the diagram by a black diamond, as shown in figure 2.15. It can be created by either control-clicking on the background or by clicking on the button in the toolbar with the black diamond on it.

Making a connection to a relation can be tricky, since if you just click and drag on the relation, the relation gets selected and moved. To make a connection, hold the control button while clicking and dragging on the relation.

In the model shown in figure 2.15, 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 to it, a connection to a relation. Relations can also be used to control the routing of wires in the diagram. However, as of the 3.0 release of Ptolemy II, a connection can only have a single relation on it, so the degree to which routing can be controlled is limited.

To explore multiports, try putting some other signal source in the diagram and connecting it to the *SequencePlotter* or to the *Display*. If you explore this fully, you will discover that the *SequencePlotter* can only accept inputs of type *double*, or some type that can be losslessly converted to *double*, such as *int*. These data type issues are explored next.

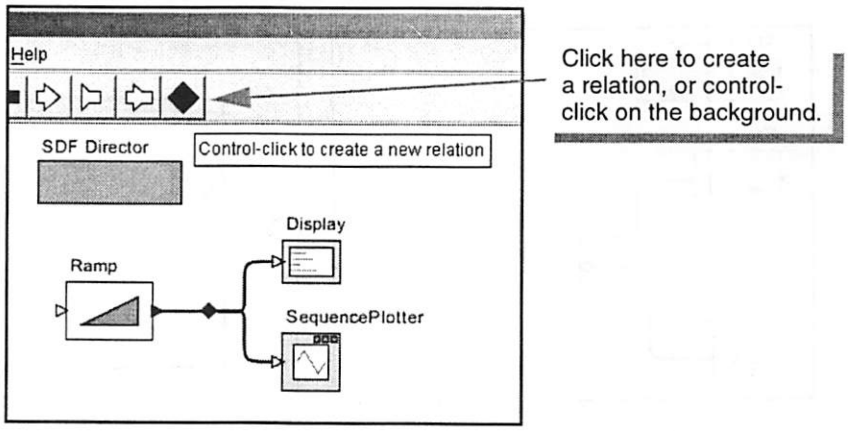

Help
Click here to create a relation, or control-click on the background.
SDF Director
Control-click to create a new relation
Display
Ramp
SequencePlotter

FIGURE 2.15. A relation can be used to broadcast an output from a single port.

2.3 Tokens and Data Types

In the example of figure 2.10, the *Const* actor creates a sequence of values on its output port. The values are encapsulated as *tokens*, and sent to the *Display* actor, which consumes them and displays them in the run window.

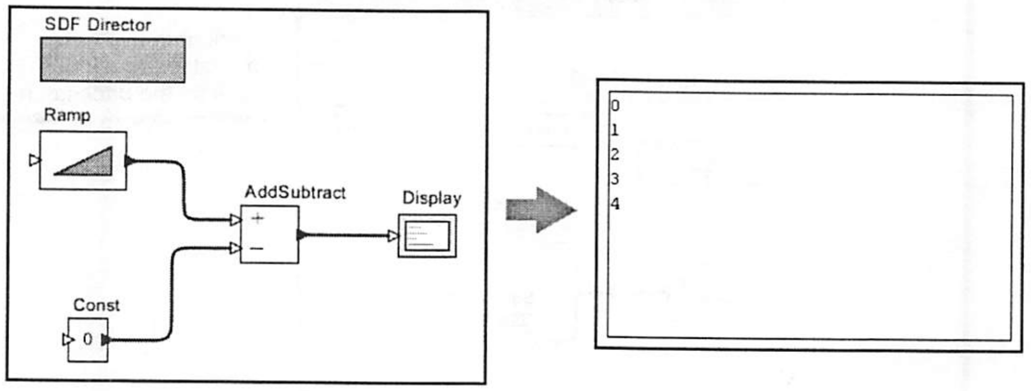
The tokens produced by the *Const* actor can have any value that can be expressed in the Ptolemy II *expression language*. We will say more about the expression language in chapter 3, "Expressions", but for now, try giving the value 1 (the integer with value one), or 1.0 (the floating-point number with value one), or {1.0} (An array containing a one), 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 expressions.

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*, meaning that they can operate on or produce data with multiple types. The behavior may even be different for different types. Multiplying matrices, for example, is not the same as multiplying integers, but both are accomplished by the *MultiplyDivide* actor in the *math library*. Ptolemy II includes a sophisticated type system that allows this to be done efficiently and safely.

To explore data types a bit further, try creating the model in figure 2.16. The *Ramp* actor is listed under *sources*, *sequence sources*, and the *AddSubtract* actor is listed under *math*. Set the *value* parameter of the constant to be 0 and the *iterations* parameter of the director to 5. Running the model should result in 5 numbers between 0 and 4, as shown in the figure. These are the values produced by the *Ramp*, which are having the value of the *Const* actor subtracted from them. Experiment with changing the value of the *Const* actor and see how it changes the 5 numbers at the output.

Now for the real test: 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 2.17. Do not worry; exceptions are a normal part of constructing (and debugging) models. In this case, the exception window is telling you that you have tried to subtract a string value from an integer value, which doesn't make much sense at all (following Java, adding strings *is* allowed). This is an example of a type error.

Exceptions can be a very useful debugging tool, particularly if you are developing your own components in Java. To illustrate how to use them, click on the Display Stack Trace button in the exception window of figure 2.17. You should see the stack trace shown in figure 2.18. This window displays the execution sequence that resulted in the exception. For example, the line



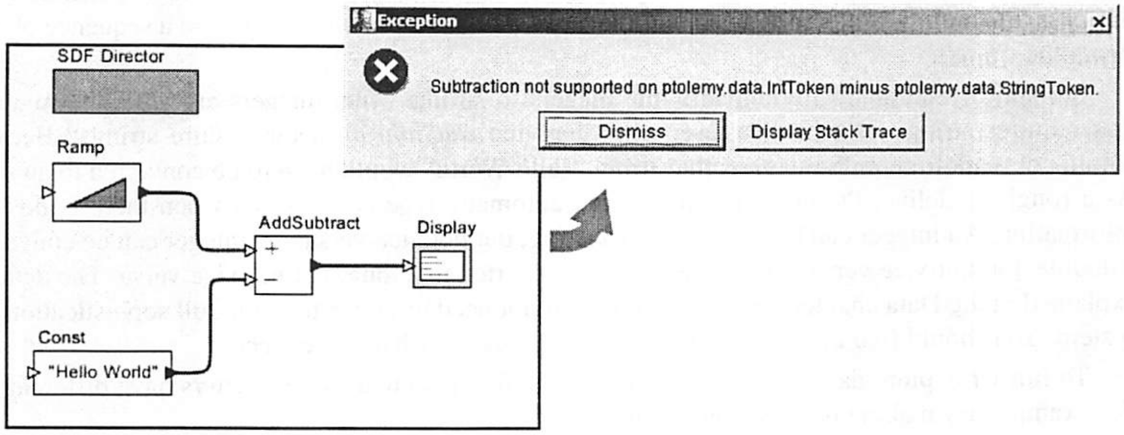
```
0
1
2
3
4
```

FIGURE 2.16. Another example, used to explore data types in Ptolemy II.

```
    at ptolemy.data.IntToken.subtract(IntToken.java:547)
```

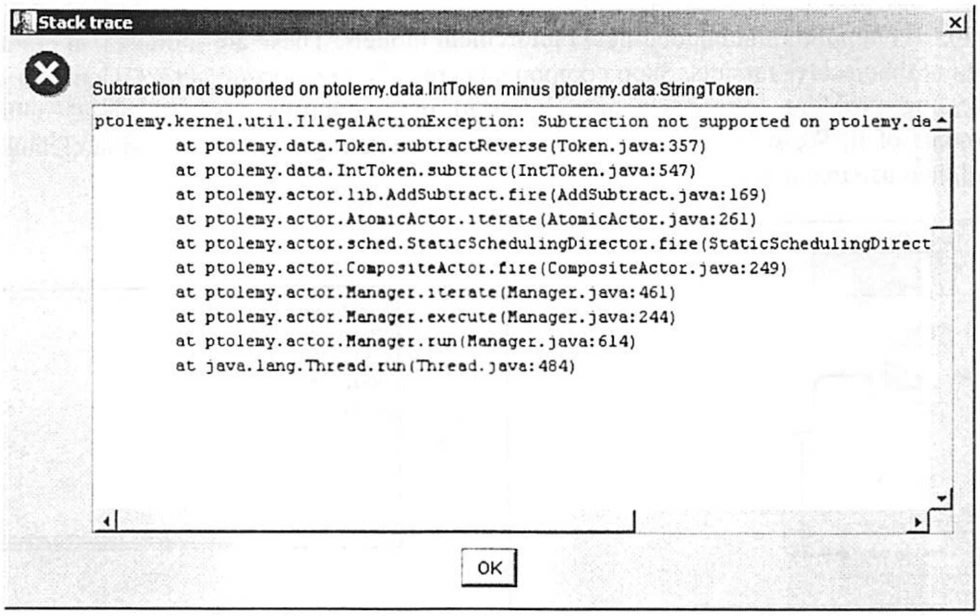
indicates that the exception occurred within the subtract() method of the class ptolemy.data.IntToken, at line 547 of the source file IntToken.java. Since Ptolemy II is distributed with source code (except in the Windows installer version and the Web Start Web Edition), this can be very useful information. 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 IntToken.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



Subtraction not supported on ptolemy.data.IntToken minus ptolemy.data.StringToken.

FIGURE 2.17. An example that triggers an exception when you attempt to execute it. Strings cannot be subtracted from integers.



Subtraction not supported on ptolemy.data.IntToken minus ptolemy.data.StringToken.

```
ptolemy.kernel.util.IllegalActionException: Subtraction not supported on ptolemy.da
        at ptolemy.data.Token.subtractReverse(Token.java:357)
        at ptolemy.data.IntToken.subtract(IntToken.java:547)
        at ptolemy.actor.lib.AddSubtract.fire(AddSubtract.java:169)
        at ptolemy.actor.AtomicActor.iterate(AtomicActor.java:261)
        at ptolemy.actor.sched.StaticSchedulingDirector.fire(StaticSchedulingDirect
        at ptolemy.actor.CompositeActor.fire(CompositeActor.java:249)
        at ptolemy.actor.Manager.iterate(Manager.java:461)
        at ptolemy.actor.Manager.execute(Manager.java:244)
        at ptolemy.actor.Manager.run(Manager.java:614)
        at java.lang.Thread.run(Thread.java:484)
```

FIGURE 2.18. Stack trace for the exception shown in figure 2.17.

replaced by slashes; in this case, it is at $PTII/ptolemy/data/IntToken.java (the slashes might be back-slashes under Windows).

Let's try a small change to the model to get something that does not trigger an exception. Disconnect the *Const* from the lower port of the *AddSubtract* actor and connect it instead to the upper port, as shown in figure 2.19. You can do this by selecting the connection and deleting it (using the delete key), then 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 it is a *multiport*, meaning that you can make more than one connection to it. Now when you run the model you should see strings like "0HelloWorld", as shown in the figure.

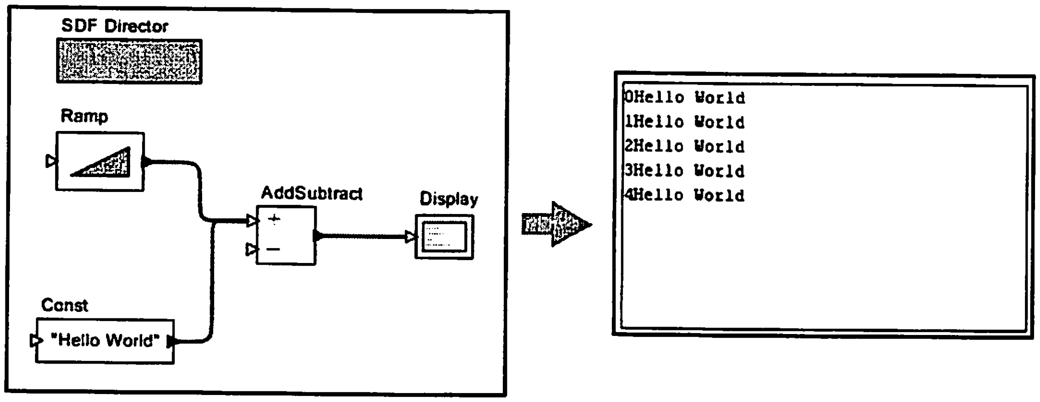
There are two interesting things going on here. The first is that, as in Java, strings are added by concatenating them. The second is that the integers from the *Ramp* are converted to strings and concatenated with the string "Hello World". All the connections to a multiport must have the same type. In this case, the multiport has a sequence of integers coming in (from the *Ramp*) and a sequence of strings (from the *Const*).

Ptolemy II automatically converts the integers to strings when integers are provided to an actor that requires strings. But in this case, why does the *AddSubtract* actor require strings? Because it would not work to require integers; the string "Hello World" would have to be converted to an integer. As a rough guideline, Ptolemy II will perform automatic type conversions when there is no loss of information. An integer can be converted to a string, but not vice versa. An integer can be converted to a double, but not vice versa. An integer can be converted to a long, but not vice versa. The details are explained in the Data chapter, but many users will not need to understand the full sophistication of the system. You should find that most of the time it will just do what you expect.

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

2.4 Hierarchy

Ptolemy II supports (and encourages) hierarchical models. These are models that contain components that are themselves models. Such components are called *composite actors*. Consider a small signal processing problem, where we are interested in recovering a signal based only on noisy measurements of it. We will create a composite actor modeling a communication channel that adds noise, and then use that actor in a model.



```
0Hello World
1Hello World
2Hello World
3Hello World
4Hello World
```

FIGURE 2.19. Addition of a string to an integer.

2.4.1 Creating a Composite Actor

First open a new graph editor and drag in a *Typed Composite Actor* from the *utilities* library. This actor is going to add noise to our measurements. First, using the context menu (obtained by right clicking over the composite actor), select "Customize Name", and give the composite a better name, like "Channel", as shown in figure 2.20. Then, using the context menu again, select "Look Inside" on the actor. You should get a blank graph editor, as shown in figure 2.21. The original graph editor is still open. To see it, move the new graph editor window by dragging the title bar of the window.

2.4.2 Adding Ports to a Composite Actor

First we have to add some ports to the composite actor. There are several ways to do this, but clicking on the port buttons in the toolbar is probably the easiest. You can explore the ports in the toolbar by lingering with the mouse over each button in the toolbar. A tool tip pops up that explains the button.

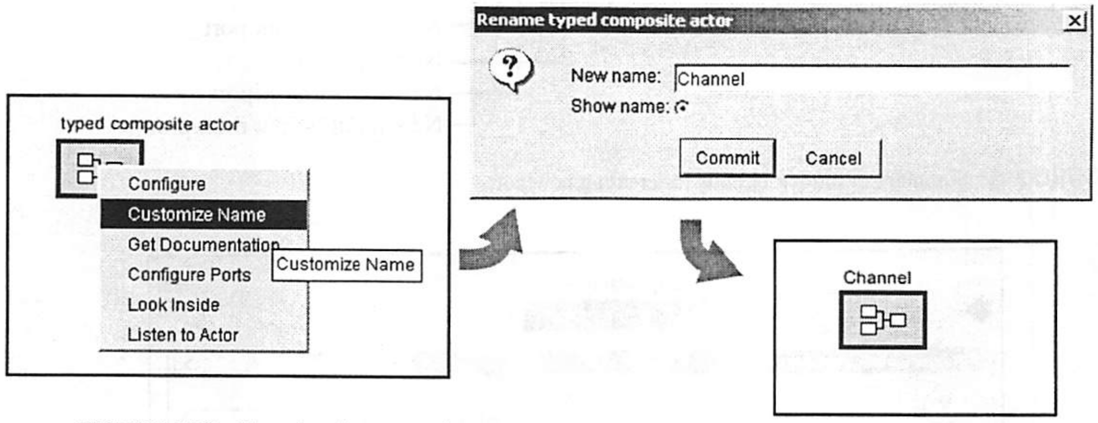


FIGURE 2.20. Changing the name of an actor.

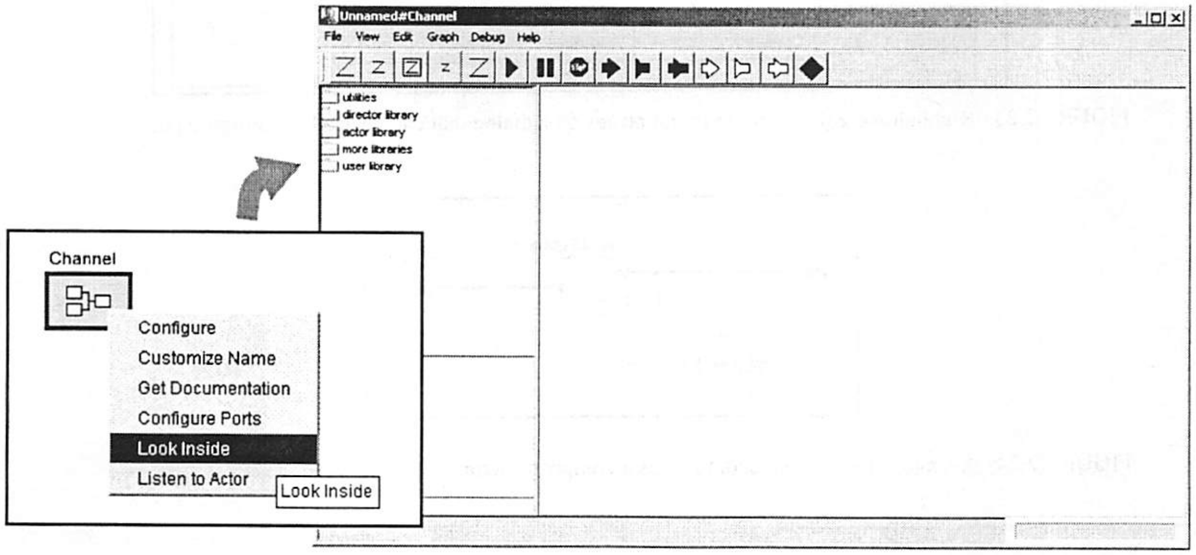


FIGURE 2.21. Looking inside a composite actor.

The buttons are summarized in figure 2.22. Create an input port and an output port and rename them *input* and *output* right by clicking on the ports and selecting "Customize Name". Note that, as shown in figure 2.23, you can also right click on the background of the composite actor and select *Configure Ports* to change whether a port is an input, an output, or a multiport. The resulting dialog also allows you to set the type of the port, although much of the time you will not need to do this, since the type inference mechanism in Ptolemy II will figure it out from the connections.

Then using these ports, create the diagram shown in figure 2.24[1]. The *Gaussian* actor creates values from a Gaussian distributed random variable, and is found in the *random* library. Now if you close

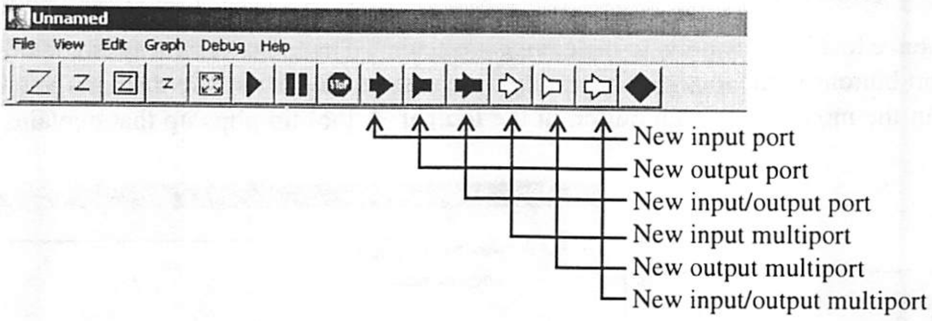
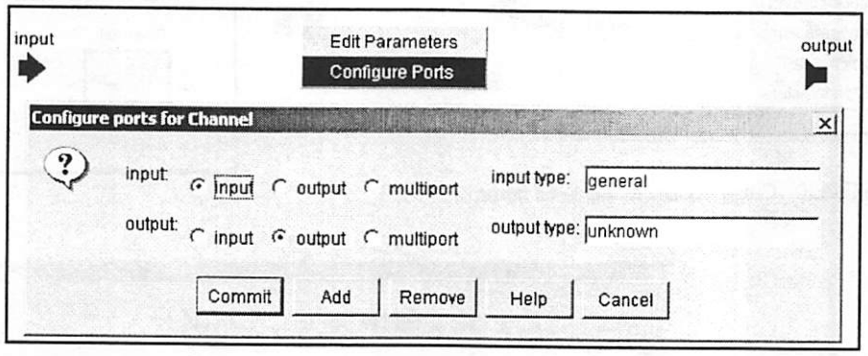


FIGURE 2.22. Summary of toolbar buttons for creating new ports.



input: [✓ input] output multiport input type: general

output: input [✓ output] multiport output type: unknown

FIGURE 2.23. Right clicking on the background brings up a dialog that can be used to configure ports.

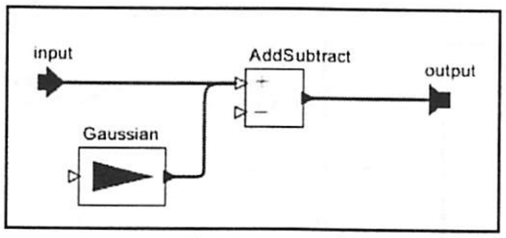


FIGURE 2.24. A simple channel model defined as a composite actor.

1. **Hint:** to create a connection starting on one of the external ports, hold down the control key when dragging.

this editor and return to the previous one, you should be able to easily create the model shown in figure 2.25. The *Sinewave* actor is listed under *sources*, and the *SequencePlotter* actor is found in *sinks*. Notice that the *Sinewave* actor is also a hierarchical model, as suggested by its red outline (try looking inside). If you execute this model (you will probably want to set the iterations to something reasonable, like 100), you should see something like figure 2.26.

2.4.3 Setting the Types of Ports

In the above example, we never needed to define the types of any ports. The types were inferred from the connections. Indeed, this is usually the case in Ptolemy II, but occasionally, you will need to set the types of the ports. Notice in figure 2.23 that there is a position in the dialog box that configures ports for specifying the type. Thus, to specify that a port has type *boolean*, you could enter *boolean* into the dialog box. There are other commonly used types: *complex*, *double*, *fixedpoint*, *general*, *int*, *long*, *matrix*, *object*, *scalar*, *string*, and *unknown*. Let's take a more complicated case. How would you specify that the type of a port is a double matrix? Easy:

```
    [double]
```

This expression actually creates a 1 by 1 matrix containing a double (the value of which is irrelevant). It thus serves as a prototype to specify a double matrix type. Similarly, we can specify an array of com-

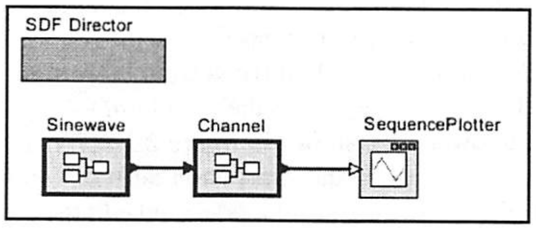
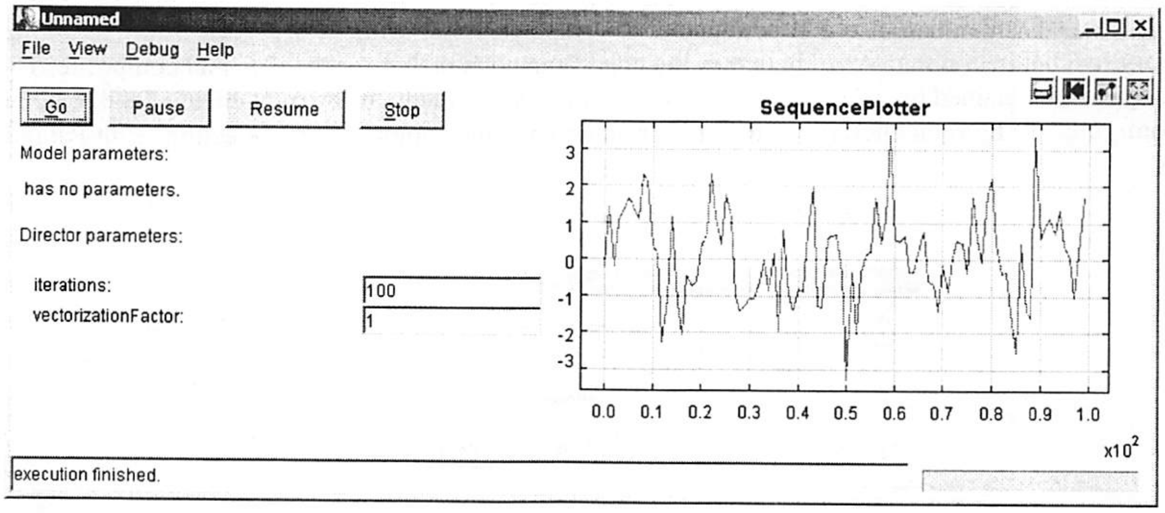


FIGURE 2.25. A simple signal processing example that adds noise to a sinusoidal signal.



Model parameters:
has no parameters.
Director parameters:
iterations: 100
vectorizationFactor: 1

execution finished.

FIGURE 2.26. The output of the simple signal processing model in figure 2.25.

plex numbers as

```
    {complex}
```

In the Ptolemy II expression language, square braces are used for matrices, and curly braces are used for arrays. What about a record containing a string named "name" and an integer named "address"? Easy:

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

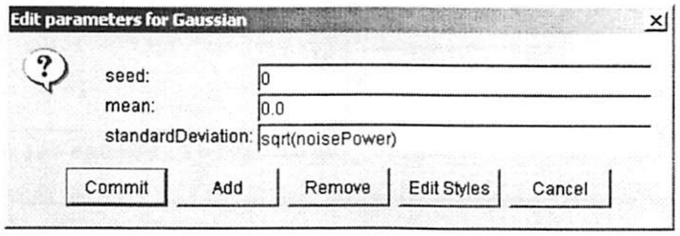
2.5 Annotations and Parameterization

In this section, we will enhance the model in figure 2.25 in a number of ways.

First, notice from figure 2.26 that the noise overwhelms the sinusoid, making it barely visible. A useful channel model would have a parameter that sets the level of the noise. Look inside the channel model, and add a parameter by dragging one in from the *utilities* library, as shown in figure 2.27. Right click on the parameter to change its name to "noisePower". (In order to be able to use this parameter in expressions, the name cannot have any spaces in it.) Also, right click or double click on the parameter to change its default value to 0.1.

Now we can use this parameter. First, let's use it to set the amount of noise. The *Gaussian* actor has a parameter called *standardDeviation*. In this case, the power of the noise is equal to the variance of the Gaussian, not the standard deviation. If you recall from basic statistics, the standard deviation is equal to the square root of the variance. Change the *standardDeviation* parameter of the *Gaussian* actor so its value is "sqrt(noisePower)", as shown in figure 2.28. This is an expression that references the *noisePower* parameter. We will explain the expression language in the next chapter. But first, let check our improved model. Return to the top-level model, and edit the parameters of the *Channel* actor (by either double clicking or right clicking and selecting "Configure"). Change the noise power from the default 0.1 to 0.01. Run the model. You should now get a relatively clean sinusoid like that shown in figure 2.29.

Note that you can also add parameters to a composite actor without dragging from the *utilities* library by clicking on the "Add" button in the edit parameters dialog for the *Channel* composite. This dialog can be obtained by either double clicking on the *Channel* icon, or by right clicking and selecting "Configure", or by right clicking on the background inside the composite and selecting "Edit Parameters".


Edit parameters for Gaussian

seed: 0
mean: 0.0
standardDeviation: sqrt(noisePower)

FIGURE 2.28. The standard deviation of the *Gaussian* actor is set to the square root of the noise power.

There are several other useful enhancements you could make to this model. Try dragging an *annotation* from the *utilities* library and creating a title on the diagram. Also, try setting the title of the plot by clicking on the second button from the right in the row of buttons at the top right of the plot. This button produces the tool tip "Set the plot format" and bring up the format control window.

2.6 Navigating Larger Models

Sometimes, a model gets large enough that it is not convenient to view it all at once. There are four toolbar buttons, shown in figure 2.27 that help. These buttons permit zooming in and out. The "Zoom reset" button restores the zoom factor to the "normal" one, and the "Zoom fit" calculates the zoom factor so that the entire model is visible in the editor window.

In addition, it is possible to pan over a model. Consider the window shown in figure 2.31. Here, we have zoomed in so that icons are larger than the default. The *pan window* at the lower left shows the

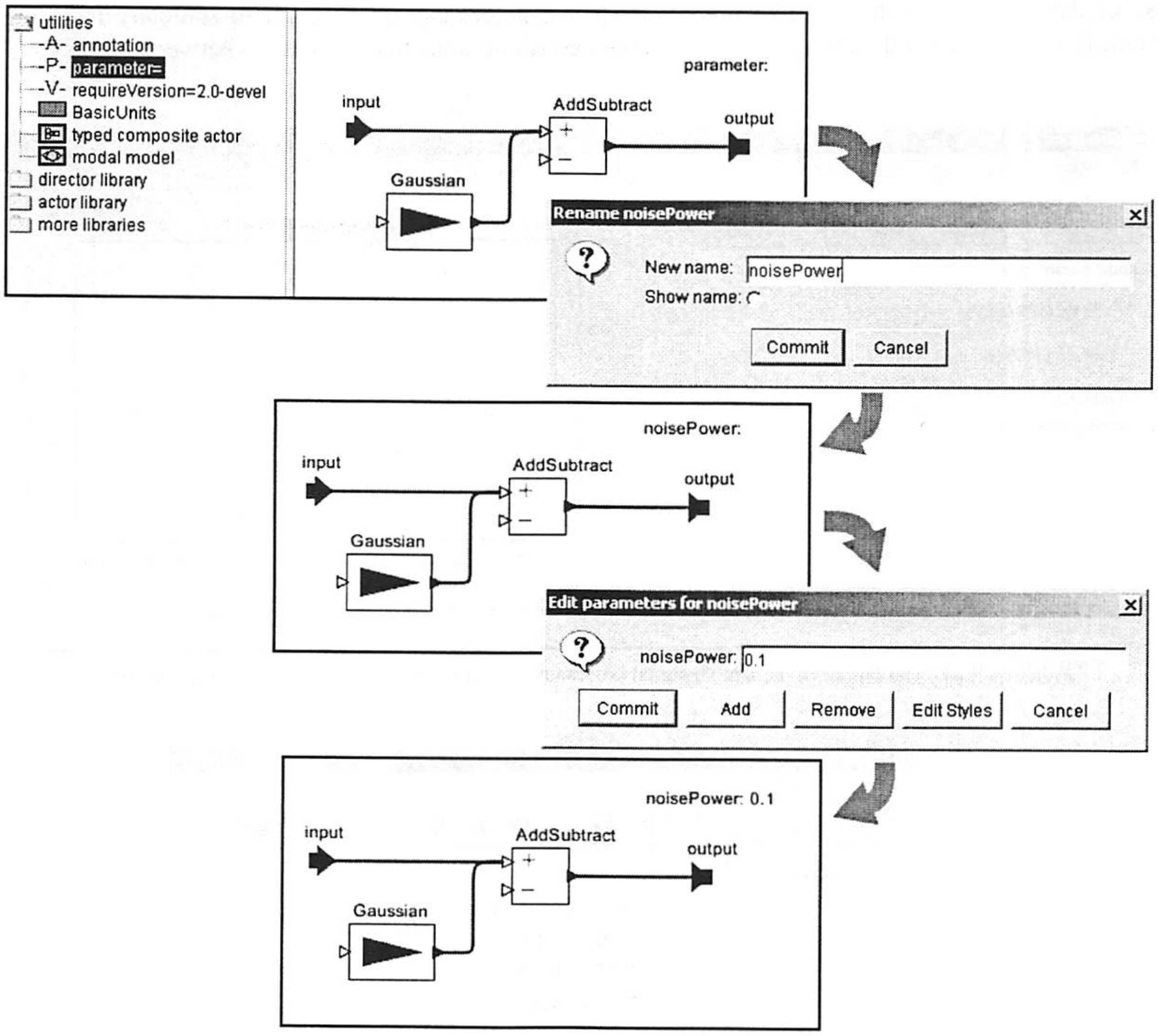


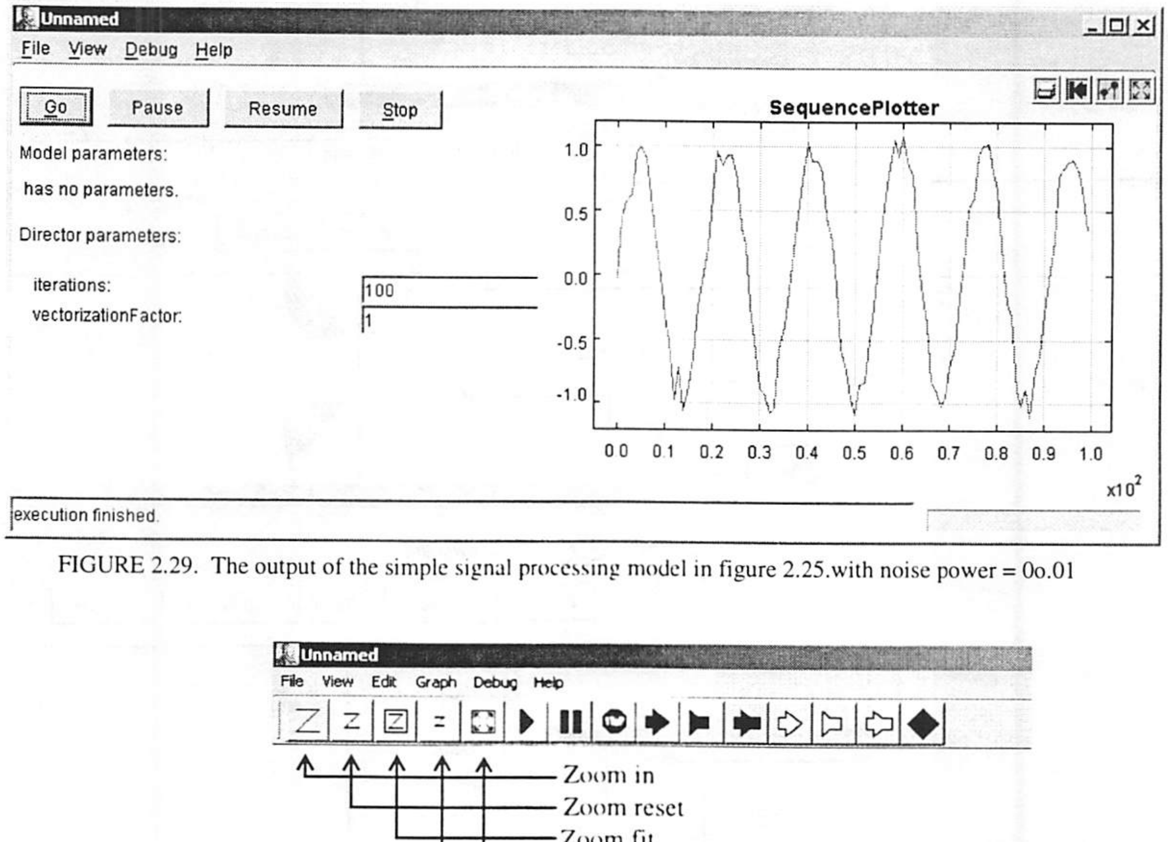
FIGURE 2.27. Adding a parameter to the channel model.

entire model, with a red box showing the visible portion of the model. By clicking and dragging in the pan window, it is easy to navigate around the entire model. Clicking on the "Zoom fit" button in the toolbar results in the editor area showing the entire model, just as the pan window does.

2.7 Domains

A key innovation in Ptolemy II is that, unlike other design and modeling environments, there are several available *models of computation* that define the meaning of a diagram. In the above examples, we directed you to drag in an *SDF Director* without justifying why. A director in Ptolemy II gives meaning (semantics) to a diagram. It specifies what a connection means, and how the diagram should be executed. In Ptolemy II terminology, the director realizes a *domain*. Thus, when you construct a model with an SDF director, you have constructed a model "in the SDF domain."

The SDF director is fairly easy to understand. "SDF" stands for "synchronous dataflow." In dataflow models, actors are invoked (fired) when their input data is available. SDF is particularly simple case of dataflow where the order of invocation of the actors can be determined statically from the model. It does not depend on the data that is processed (the tokens that are passed between actors).



Model parameters:

has no parameters.

Director parameters:

iterations: 100

vectorizationFactor: 1

execution finished.

FIGURE 2.29. The output of the simple signal processing model in figure 2.25.with noise power = 0o.01

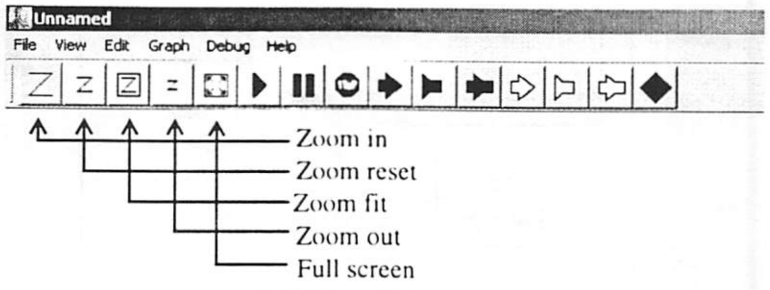


FIGURE 2.30. Summary of toolbar buttons for zooming and fitting.

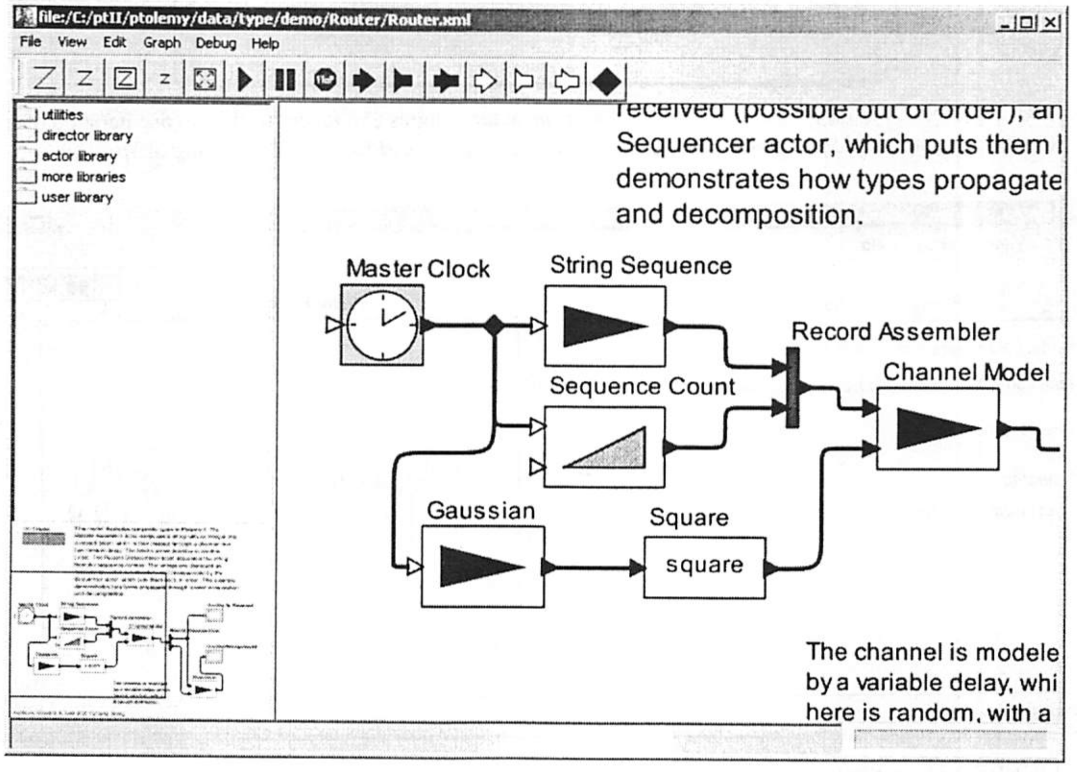
But there are other models of computation available in Ptolemy II. It can be difficult to determine which one to use without having experience with several. Moreover, you will find that although most actors in the library do *something* in any domain in which you use them, they do not always do something useful. It is important to understand the domain you are working with and the actors you are using. Here, we give a very brief introduction to some of the domains. But we begin first by explaining some of the subtleties in SDF.

2.7.1 SDF and Multirate Systems

So far we have been dealing with relatively simple systems. They are simple in the sense that each actor produces and consumes one token from each port at a time. In this case, the SDF director simply ensures that an actor fires after the actors whose output values it depends on. The total number of output values that are created by each actor is determined by the number of iterations, but in this simple case only one token would be produced per iteration.

It turns out that the SDF scheduler is actually much more sophisticated. It is capable of scheduling the execution of actors with arbitrary prespecified data rates. Not all actors produce and consume just a single sample each time they are fired. Some require several input token before they can be fired, and produce several tokens when they are fired.

One such actor is a spectral estimation actor. figure 2.32 shows a system that computes the spectrum of the same noisy sine wave that we constructed in figure 2.25. The *Spectrum* actor has a single parameter, which gives the *order* of the FFT used to calculate the spectrum. Figure 2.33 shows the output of the model with *order* set to 8 and the number of *iterations* set to 1. **Note that there are 256 out-**



received (possible out of order), an
Sequencer actor, which puts them i
demonstrates how types propagate
and decomposition.

The channel is modele
by a variable delay, whi
here is random, with a

FIGURE 2.31. The pan window at the lower left has a red box representing the visible are of the model in the main editor window. This red box can be moved around to view different parts of the model.

put samples output from the *Spectrum* actor. This is because the *Spectrum* actor requires 2^8, or 256 input samples to fire, and produces 2^8, or 256 output samples when it fires. Thus, one iteration of the model produces 256 samples. The *Spectrum* actor makes this a *multirate* model, because the firing rates of the actors are not all identical.

It is common in SDF to construct models that require exactly one iteration to produce a useful result. In some multirate models, it can be complicated to determine how many firings of each actor occur per iteration of the model. See the SDF chapter for details.

A second subtlety with SDF models is that if there is a feedback loop, as in figure 2.34, then the loop must have at least one instance of the *SampleDelay* actor in it (found in the *flow control* library). Without this actor, the loop will deadlock. The *SampleDelay* actor produces initial tokens on its output, before the model begins firing. The initial tokens produced are given by a the *initialOutputs* parameter, which specifies an array of tokens. These initial tokens enable downstream actors and break the circular dependencies that would result otherwise from a feedback loop.

A final issue to consider with the SDF domain is time. Notice that in all the examples above we have suggested using the *SequencePlotter* actor, not the *TimedPlotter* actor, which is in the same *sinks* library. This is because the SDF domain does not include in its semantics a notion of time. Time does

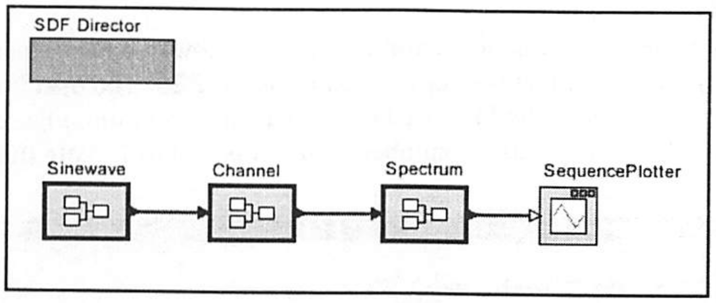
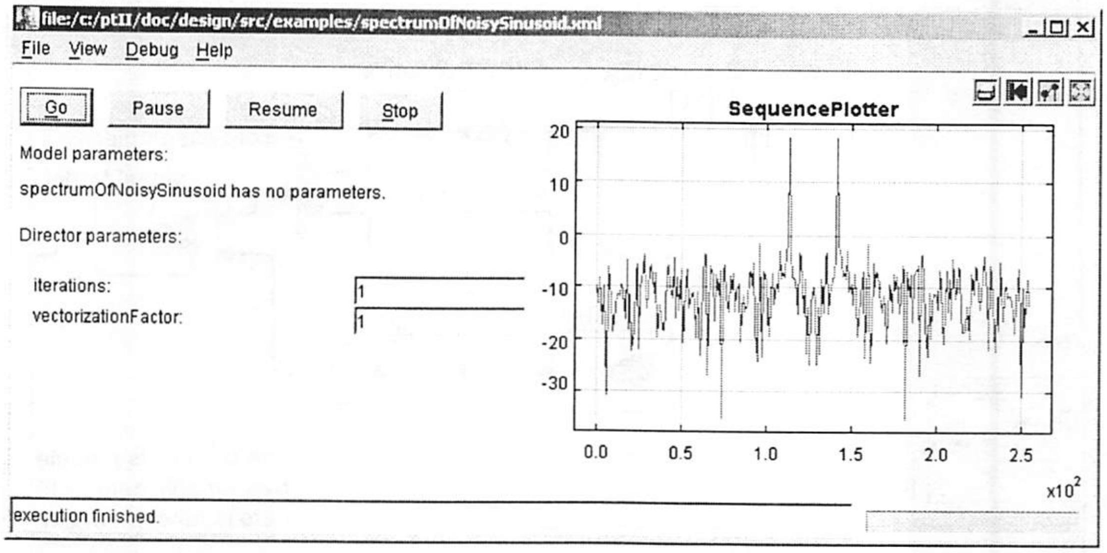


FIGURE 2.32. A multirate SDF model. The *Spectrum* actor requires 256 tokens to fire, so one iteration of this model results in 256 firings of *Sinewave, Channel,* and *SequencePlotter*, and one firing of *Spectrum*.



Model parameters:

spectrumOfNoisySinusoid has no parameters.

Director parameters:

iterations: 1
vectorizationFactor: 1

execution finished.

FIGURE 2.33. A single iteration of the SDF model in figure 2.32 produces 256 output tokens.

not advance as an SDF model executes, so the *TimedPlotter* actor would produce very uninteresting results, where the horizontal axis value would always be zero. The *SequencePlotter* actor uses the index in the sequence for the horizontal axis. The first token received is plotted at horizontal position 0, the second at 1, the third at 2, etc. The next domain we consider, DE, includes much stronger notion of time, and it is almost always more appropriate in the DE domain to use the *TimedPlotter* actor.

2.7.2 Discrete-Event Systems

In discrete-event (DE) systems, the 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 its time stamp is a double-precision floating-point number. This is different from dataflow, where a signal consists of a sequence of tokens, and there is no time significance in the signal.

A DE model executes chronologically, processing the oldest events first. Time advances as events are processed. There is potential confusion, however, between *model time*, the time that evolves in the model, and *real time*, the time that elapses in the real world while the model executes (also called *wall-clock time*). Model time may advance more rapidly than real time or more slowly. The DE director has a parameter, *synchronizeToRealTime*, that, when set to true, attempts to synchronize the two notions of time. It does this by delaying execution of the model, if necessary, allowing real time to catch up with model time.

Consider the DE model shown in figure 2.35. This model includes a *PoissonClock* actor, a *CurrentTime* actor, and a *WallClockTime* actor, all found in the *sources* library. The *PoissonClock* actor generates a sequence of events with random times, where the time between events is exponentially distributed. Such an event sequence is known as a Poisson process. The value of the events produced by the *PoissonClock* actor is a constant, but the value of that constant is ignored in this model. Instead, these events trigger the *CurrentTime* and *WallClockTime* actors. The *CurrentTime* actor outputs an event with the same time stamp as the input, but whose value is the current model time (equal to the time stamp of the input). The *WallClockTime* actor an event with the same time stamp as the input, but whose value is the current real time, in seconds since initialization of the model.

The plot in figure 2.35 shows an execution. Note that model time has advanced approximately 10 seconds, but real time has advanced almost not at all. In this model, model time advances much more rapidly than real time. If you build this model, and set the *synchronizeToRealTime* parameter of the director to true, then you will find that the two plots coincide almost perfectly.

A significant subtlety in using the DE domain is in how simultaneous events are handled. Simultaneous events are simply events with the same time stamp. We have stated that events are processed in chronological order, but if two events have the same time stamp, then there is some ambiguity. Which one should be processed first? If the two events are on the same signal, then the answer is simple: pro-

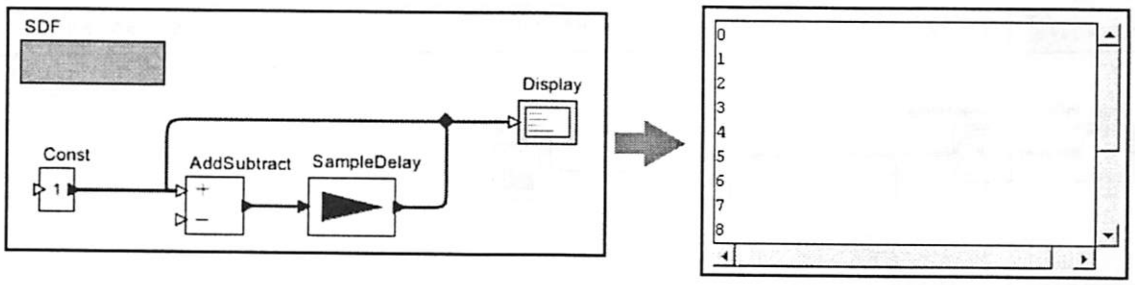


FIGURE 2.34. An SDF model with a feedback loop must have at least one instance of the *SampleDelay* actor in it.

cess first the one that was produced first. However, if the two events are on different signals, then the answer is not so clear.

Consider the model shown in figure 2.36, which produces a histogram of the interarrival times of events from the *PoissonClock* actor. In this model, we calculate the difference between the current

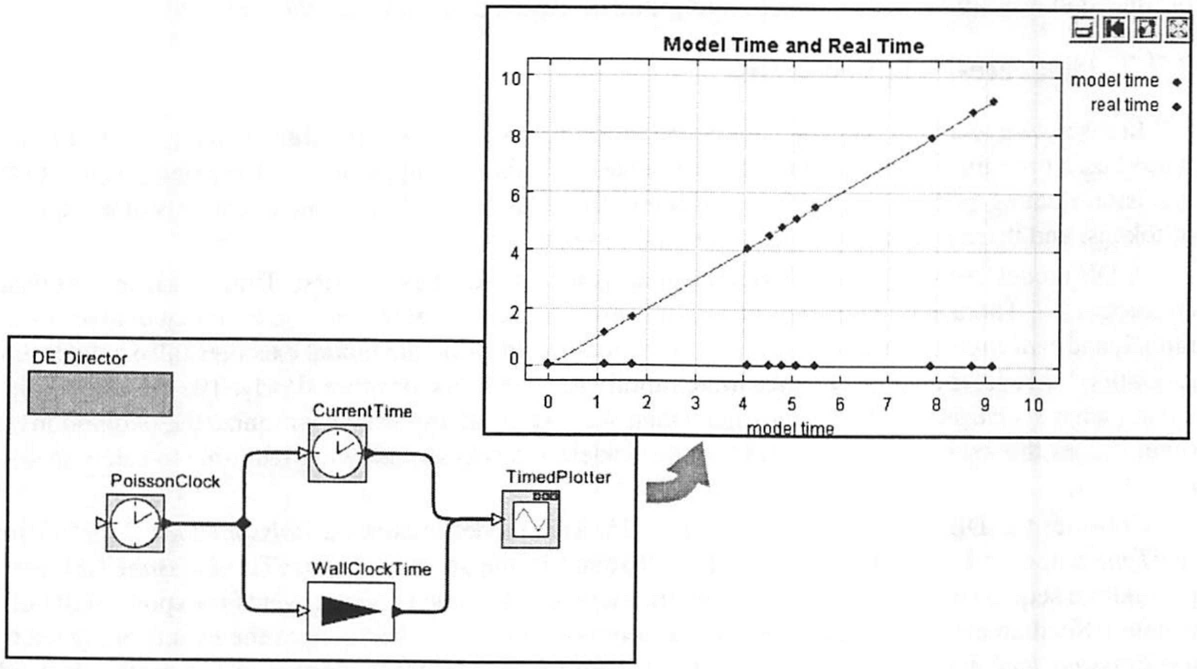


FIGURE 2.35. Model time vs. real time (wall clock time).

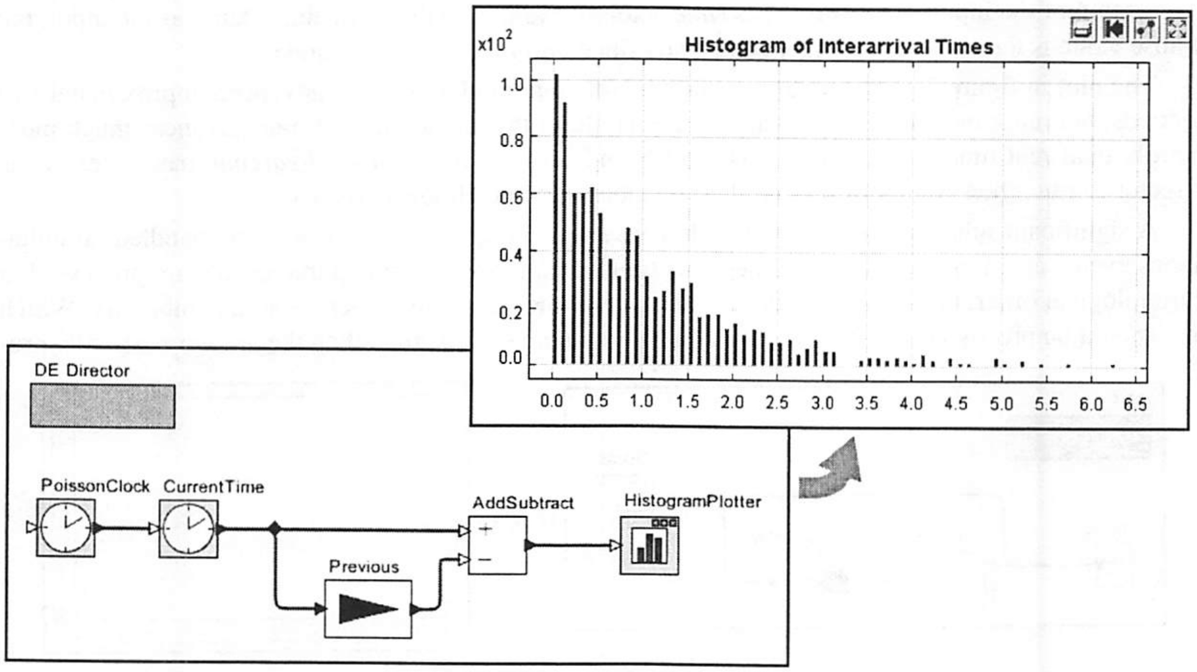


FIGURE 2.36. Histogram of interarrival times, illustrating handling of simultaneous events.

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). Thus, when the *PoissonClock* actor produces an output, there will be two simultaneous events, one at the input to the *plus* port of the *AddSubtract* actor, and one at the input of the *Previous* actor. Should the director fire the *AddSubtract* actor or the *Previous* actor? Either seems OK if it is to respect chronological order, but it seems intuitive that the *Previous* actor should be fired first.

It is helpful to know how the *AddSubtract* actor works. When it fires, it adds all available tokens on the *plus* port, and subtracts all available tokens on the *minus* port. If the *AddSubtract* actor fires before the *Previous* actor, then the only available token will be the one on the *plus* port, and the expected subtraction will not occur. Intuitively, we would expect the director to invoke the *Previous* actor before the *AddSubtract* actor so that the subtraction occurs.

How does the director deliver on the intuition that the *Previous* actor should be fired first? Before executing the model, the DE director constructs a *topological sort* of the model. A topological sort is simply a list of the actors in data-precedence order. For the model in figure 2.36, there is only one allowable topological sort:

- *PoissonClock, CurrentTime, Previous, AddSubtract, HistogramPlotter*

In this list, *AddSubtract* is after *Previous*. So the when they have simultaneous events, the DE director fires *Previous* first.

Thus, the DE director, by analyzing the structure of the model, usually delivers the intuitive behavior, where actors that produce data are fired before actors that consume their results, even in the presence of simultaneous events.

There remains one key subtlety. If the model has a directed loop, then a topological sort is not possible. In the DE domain, every feedback loop is required to have at least one actor in it that introduces a time delay, such as the *TimedDelay* actor, which can be found in the *domain specific* library under *discrete-event* (this library is shown on the left in figure 2.37). Consider for example the model shown in figure 2.37. That model has a *Clock* actor, which is set to produce 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.

Occasionally, you will need to put a *TimedDelay* actor in a feedback loop with a delay of 0.0. This is particularly true if you are building complex models that mix domains, and there is a delay inside a composite actor that the DE director cannot recognize as a delay. The *TimedDelay* actor with a delay of 0.0 can be thought of as a way to let the director know that there is a time delay in the preceding actor, without specifying the amount of the time delay.

2.7.3 Continuous-Time Systems

The continuous-time domain (CT) is another relatively mature domain with semantics considerably different from either DE or SDF. In CT, the signals sent along connections between actors are usually continuous-time signals. A CT example is described above in section 2.2.3.

The CT domain can also handle discrete events. These events are usually related to a continuous-time signal, for example representing a zero-crossing of the continuous-time signal. The CT director is quite sophisticated in its handling of such mixed signal systems.

2.8 Hybrid Systems and Modal Models

Hybrid systems are models that combine continuous dynamics with discrete mode changes. They are created in Ptolemy II by creating a *modal model*, found in the *utilities* library. We start by examining a pre-built modal model, and conclude by illustrating how to construct one. Modal models can be constructed with other domains besides CT, but this section will concentrate on CT. Feel free to examine other examples of modal models given in the quick tour, figure 2.3.

2.8.1 Examining a Pre-Built Model

Consider the bouncing ball example, which can be found under "Hybrid Systems" in figure 2.3. The top-level contents of this model is shown in figure 2.38. It contains only two actors, a *Ball Model* and a *TimedPlotter*. The *Ball Model* is an instance of the *modal model* found in the *utilities* library, but renamed. If you execute the model, you should see a plot like that in the figure. The continuous dynamics correspond to the times when the ball is in the air, and the discrete events correspond to the times when the ball hits the surface and bounces.

If you look inside the *Ball Model*, you will see something like figure 2.39. Figure 2.39 shows a state-machine editor, which has a slightly different toolbar and a significantly different library at the

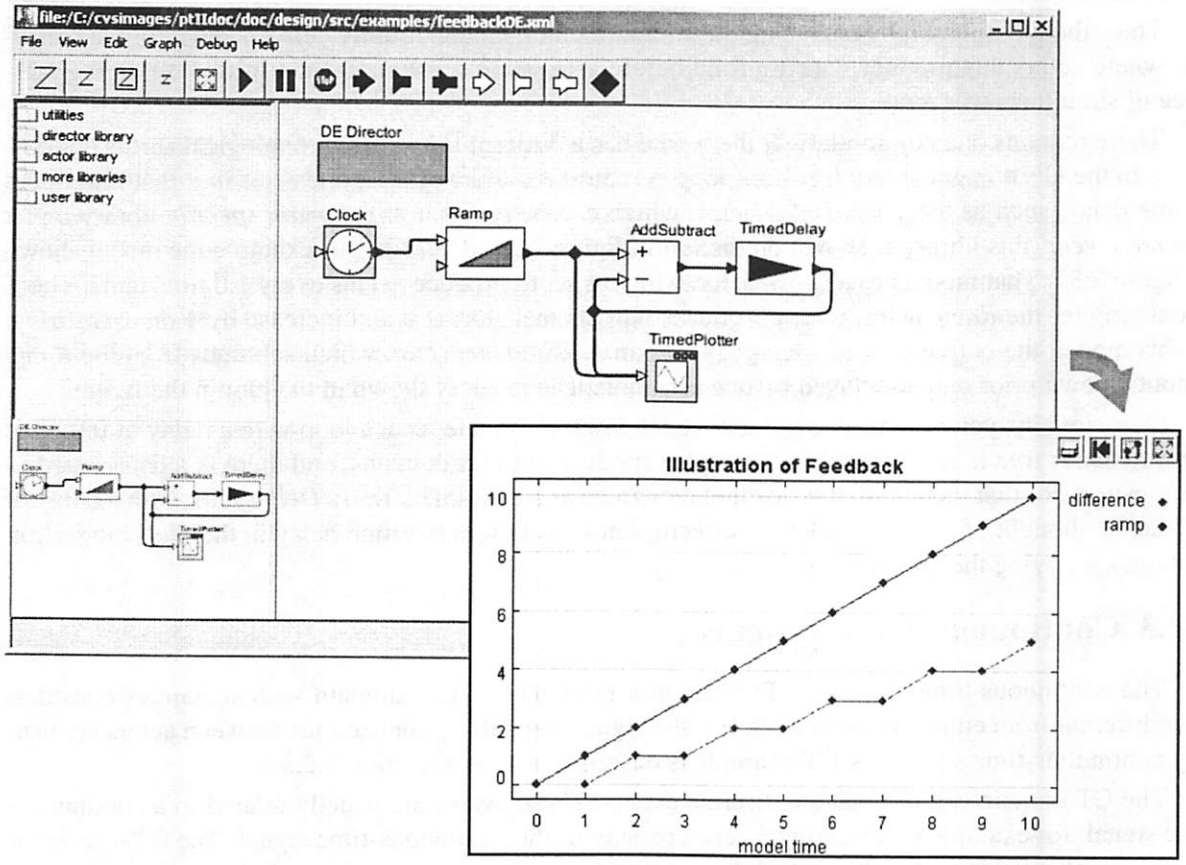


FIGURE 2.37. Discrete-event model with feedback, which requires a delay actor such as *TimedDelay*. Notice the library of domain-specific actors at the left.

left. The circles in figure 2.39 are states, and the arcs between circles are *transitions* between states. A modal model is one that has *modes*, which represent regimes of operation. Each mode in a modal model is represented by a state in a finite-state machine.

The state machine in figure 2.39 has three states, named *init*, *free*, and *stop*. The *init* state is the initial state, which is set as shown in figure 2.40. The *free* state represents the mode of operation where the ball is in free fall, and the *stop* state represents the mode where the ball has stopped bouncing.

At any time during the execution of the model, the modal model is in one of these three states. When the model begins executing, it is in the *init* state. During the time a modal model is in a state, the

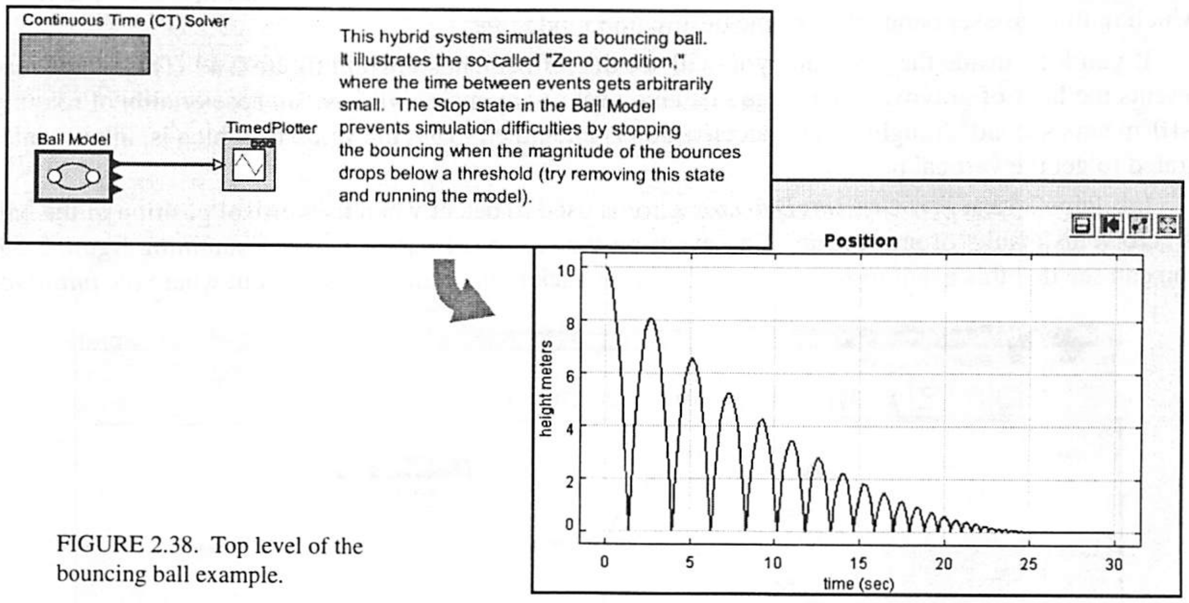
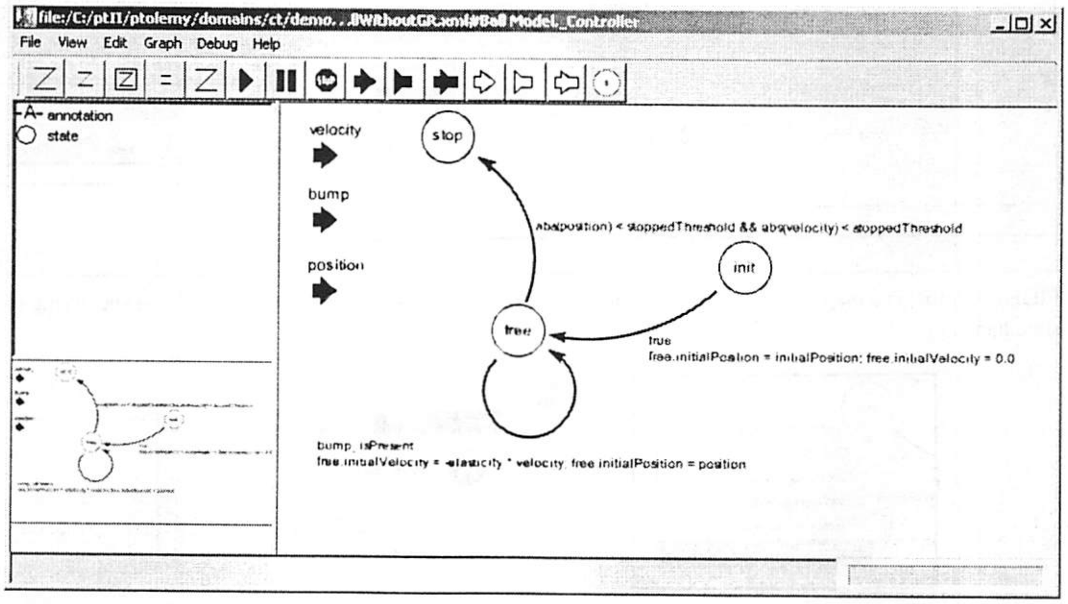

This hybrid system simulates a bouncing ball. It illustrates the so-called "Zeno condition," where the time between events gets arbitrarily small. The Stop state in the Ball Model prevents simulation difficulties by stopping the bouncing when the magnitude of the bounces drops below a threshold (try removing this state and running the model).

FIGURE 2.38. Top level of the bouncing ball example.


file:/C:/ptII/ptolemy/domains/ct/demo...BWithoutGR.xml#Ball Model_Controller
File
View
Edit
Graph
Debug
Help
annotation
state
velocity
bump
position
stop
init
free

abs(position) < stoppedThreshold && abs(velocity) < stoppedThreshold

true
free.initialPosition = initialPosition; free.initialVelocity = 0.0

bump_isPresent
free.initialVelocity = -elasticity * velocity; free.initialPosition = position

FIGURE 2.39. Inside the *Ball Model* of figure 2.38.

behavior of the modal model is specified by the *refinement* of the state. The refinement can be examined by looking inside the state. As shown in figure 2.41, the *init* state has no refinement.

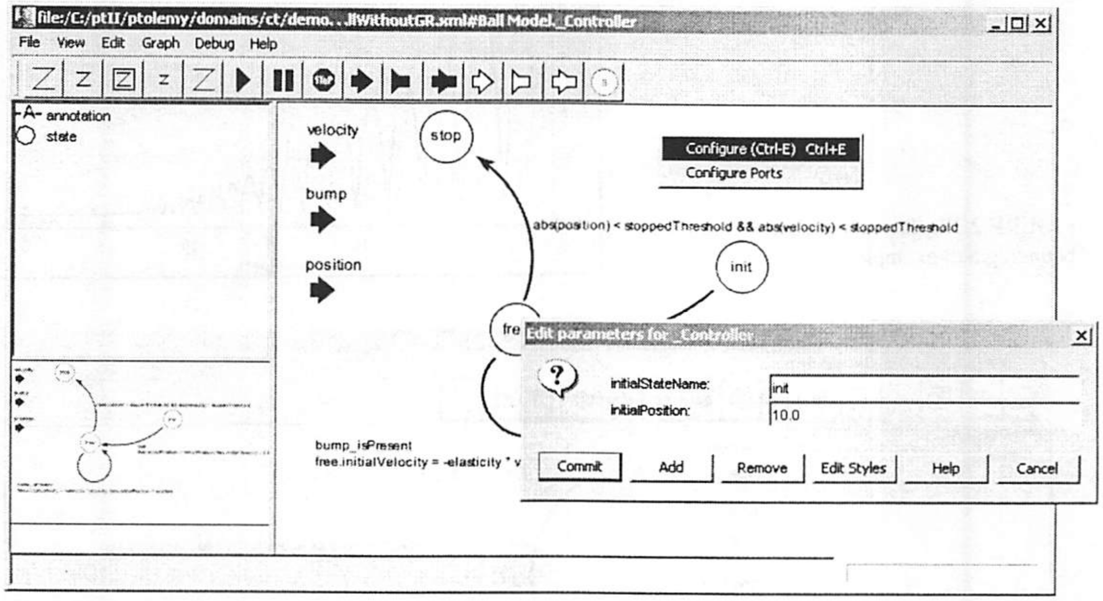
Consider the transition from *init* to *free*. It is labeled as follows:

```
true
free.initialPosition = initialPosition; free.initialVelocity = 0.0
```

The first line is a *guard*, which is predicate that determines when the transition is enabled. In this case, the transition is always enabled, since the predicate has value *true*. Thus, the first thing this model will do is take this transition and change modes to *free*. The second line specifies a sequence of *actions*, which in this case set parameters of the destination mode *free*.

If you look inside the *free* state, you will see the refinement shown in figure 2.42. This model represents the laws of gravity, which state that an object of any mass will have an acceleration of roughly -10 meters/second^2 (roughly). The acceleration is integrated to get the velocity. which is, in turn, integrated to get the vertical position.

In figure 2.42, a *ZeroCrossingDetector* actor is used to detect when the vertical position of the ball is zero. This results in production of an event on the (discrete) output *bump*. Examining figure 2.39, you can see that this event triggers a state transition back to the same *free* state, but where the *initialVe-*

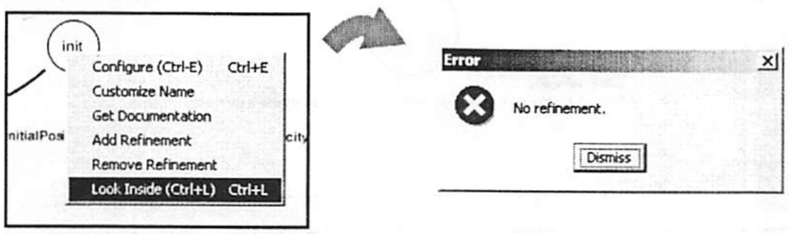

file:/C:/ptII/ptolemy/domains/ct/demo...llWithoutGR.xml#Ball Model._Controller
File View Edit Graph Debug Help
annotation
state
velocity
bump
position
stop
init
fre
Configure (Ctrl-E) Ctrl+E
Configure Ports
Edit parameters for _Controller
Commit
Add
Remove
Edit Styles
Help
Cancel

abs(position) < stoppedThreshold && abs(velocity) < stoppedThreshold

bump_isPresent
free.initialVelocity = -elasticity * v

initialStateName: init
initialPosition: 10.0

FIGURE 2.40. The initial state of a state machine is set by right clicking on the background and specifying the state name.



No refinement.

FIGURE 2.41. A state may or may not have a refinement, which specified the behavior of the model while the model is in that state. In this case, *init* has no refinement.

locity parameter is changed to reverse the sign and attenuate it by the *elasticity*. This results in the ball bouncing, and losing energy, as shown by the plot in figure 2.38.

As you can see from figure 2.39, 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. This results in the model producing no further output.

2.8.2 Numerical Precision and Zeno Conditions

The bouncing ball model of figures 2.38 and 2.39 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. If you remove the *stop* state from the FSM, and re-run the model, you get the result shown in figure 2.43. The ball, in effect, 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 some fundamental pitfalls with hybrid system modeling. The event detected by the *ZeroCrossingDetector* 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 time at that time. However, when the numbers get small enough, numerical errors take over, and the event is missed.

A related phenomenon is called the Zeno phenomenon. In the case of the bouncing ball, the time between bounces gets smaller as the simulation progresses. Since the simulator is attempting to capture every bounce event with a time step, we could encounter the problem where the number of time steps

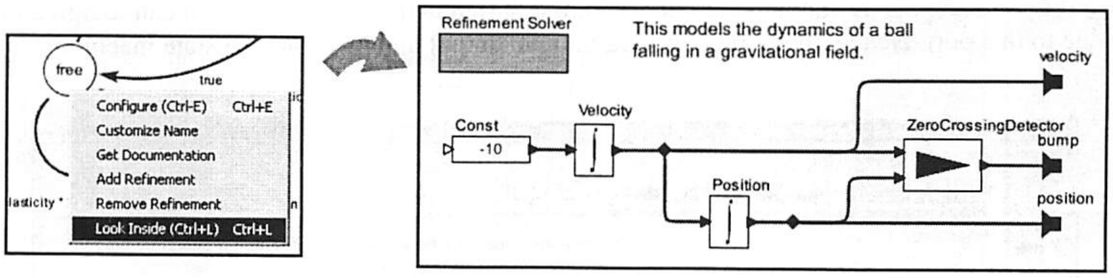


FIGURE 2.42. The refinement of the *free* state, shown here, is a continuous-model representing the laws of gravity.

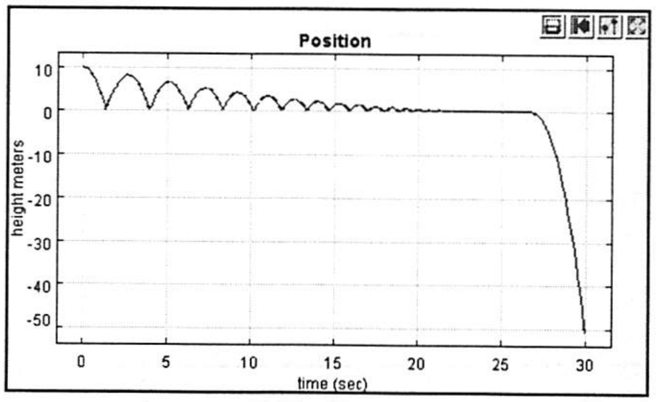


FIGURE 2.43. Result of running the bouncing ball model without the *stop* state.

becomes infinite over a finite time interval. This makes it impossible for time to advance. In fact, in theory, the bouncing ball example exhibits this Zeno phenomenon. However, numerical precision errors take over, since the simulator cannot possibly keep decreasing the magnitude of the time increments.

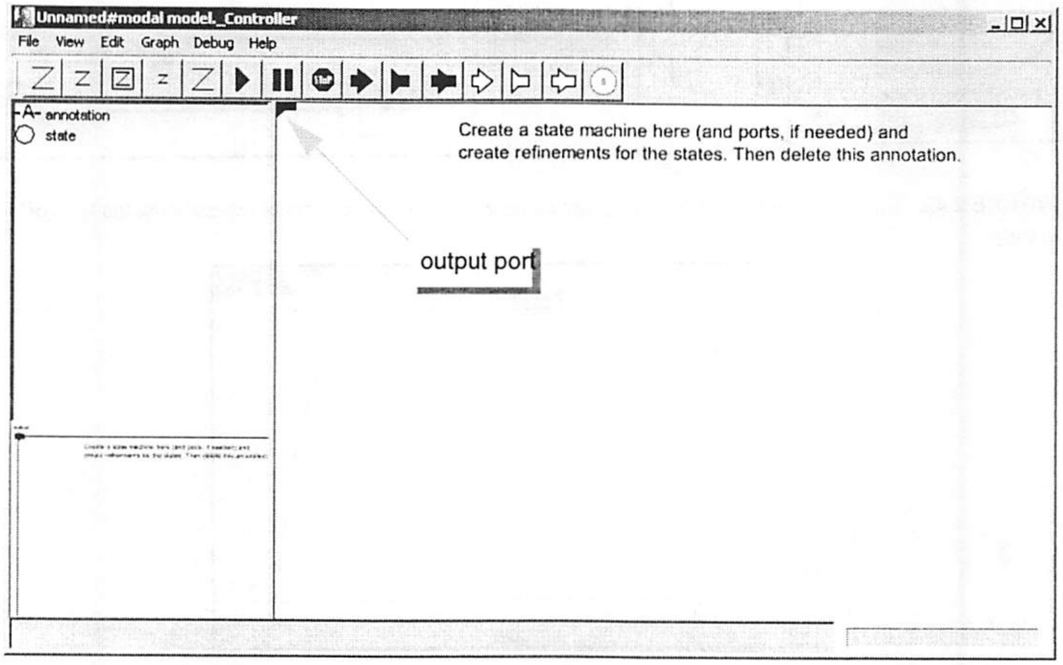
The lesson is that some caution needs to be exercised when relying on the results of a simulation of a hybrid system. Use your judgement.

2.8.3 Constructing Modal Models

A modal model is a component in a larger continuous-time (or other kind of) model. You can create a modal model by dragging one in from the *utilities* library. By default, it has no ports. To make it useful, you will need to add ports. The mechanism for doing that is identical to adding ports to a composite model, and is explained in section 2.4.2. Figure 2.38 shows a top-level continuous-time model with a single modal model that has been renamed *Ball Model*. Three output ports have been added to that modal model, but only the top one is used. It gives the vertical distance of the ball from the surface on which it bounces.

If you create a new modal model by dragging it in from the *utilities* library, create an output port and name it *output*, and then look inside, you will get an FSM editor like that shown in figure 2.44. Note that the output port is (regrettably) located at the upper left, and is only partially visible. The annotation text suggests delete once you no longer need it. You may want to move the port to a more reasonable location (where it is visible).

The output port that you created is in fact indicated in the state machine as being both an output and input port. The reason for this is that guards in the state machine can refer to output values that are produced on this port by refinements. In addition, the output actions of a transition can assign an output value to this port. Hence, the port is, in fact, both an output and input for the state machine.


Unnamed#modal model._Controller
File View Edit Graph Debug Help
annotation
state
output port

Create a state machine here (and ports, if needed) and create refinements for the states. Then delete this annotation.

FIGURE 2.44. Inside of a new modal model that has had a single output port added.

To create a finite-state machine like that in figure 2.39, drag in states (white circles), or click on the state icon in the toolbar. You can rename these states by right clicking on them and selecting "Customize Name". Choose names that are pertinent to your application. In figure 2.39, there is an *init* state for initialization, a *free* state for when the ball is in the air, and a *stop* state for when the ball is no longer bouncing. You must specify the initial state of the FSM by right clicking on the background of the FSM Editor, selecting "Edit Parameters", and specifying an initial state name, as shown in figure 2.40. In that figure, the initial state is named *init*.

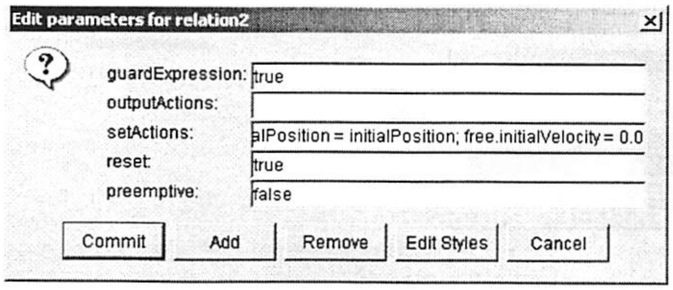
Creating Transitions. To create transitions, you must hold the control button on the keyboard while clicking and dragging from one state to the next (a transition can also go back to the same state). The handles on the transition can be used to customize its curvature and orientation. Double clicking on the transition (or right clicking and selecting "Configure") allows you to configure the transition. The dialog for the transition from *init* to *free* is shown in figure 2.45. In that dialog, we see the following:

- The guard expression is *true*, so this transition is always enabled. The transition will be taken as soon as the model begins executing. A guard expression can be any boolean-valued expression that depends on the inputs, parameters, or even the outputs of any refinement of the current state (see below). Thus, this transition is used to initialize the model.
- The output actions are empty, meaning that when this transition is taken, no output is specified. This parameter can have a list of assignments of values to output ports, separated by semicolons. Those values will be assigned to output ports when the transition is taken.
- The set actions field contains the following statements:

  ```
  free.initialPosition = initialPosition; free.initialVelocity = 0.0
  ```

  The "free" in these expressions refers to the mode refinement in the *free* state. Thus, *free.initialPosition* is a parameter of that mode refinement. Here, its value is assigned to the value of the parameter *initialPosition*. The parameter *free.initialVelocity* is set to zero.
- The *reset* parameter is set to *true*, meaning that the destination mode refinement will be initialized when the transition is taken.
- The *preemptive* parameter is set to *false*. In this case, it makes no difference, since the *init* state has no refinement. Normally, if a transition out of a state is enabled and *preemptive* is *true*, then the transition will be taken without first executing the refinement. Thus, the refinement will not affect the outputs of the modal model.

A state may have several outgoing transitions. However, it is up to the model builder to ensure that at


Edit parameters for relation2

guardExpression:	true
outputActions:	
setActions:	alPosition = initialPosition; free.initialVelocity = 0.0
reset:	true
preemptive:	false

FIGURE 2.45. Transition dialog for the transition from *init* to *free* in figure 2.39.

no time does more than one guard on these transitions evaluate to true. In other words, Ptolemy II does not allow nondeterministic state machines, and will throw an exception if it encounters one.

Creating Refinements. Both states and transitions can have *refinements.* To create a refinement, right click on the state or transition, and select "Add Refinement." You will see a dialog like that in figure 2.46. As shown in the figure, you will be offered the alternatives of a "Default Refinement" or a "State Machine Refinement." The first of these provides a block diagram model as the refinement. The second provides another finite state machine as the refinement. In the former case (the default), a blank refinement model will open, as shown in the figure. As before, the output port will appear in an inconvenient location. You will almost certainly want to move it to a more convenient location. You will have to create a director in the refinement. The modal model will not operate without a director in the refinement.

You can also create refinements for transitions, but these have somewhat different behavior. They will execute exactly once when the transition is taken. For this reason, only certain directors make sense in such refinements. The most commonly useful is the SDF director. Such refinements are typically used to perform arithmetic computations that are too elaborate to be conveniently specified as an action on the transition.

Once you have created a refinement, you can look inside a state or transition. For the bouncing ball example, the refinement of the *free* state is shown in figure 2.42. This model exhibits certain key properties of refinements:

- Refinements must contain directors. In this case, the CTEmbeddedDirector is used. When a continuous-time model is used inside a mode, this director must be used instead of the default CTDirector (see the CT domain documentation for details).
- The refinement has the same ports as the modal model, and can read input value and specify output values. When the state machine is in the state of which this is the refinement, this model will be executed to read the inputs and produce the outputs.

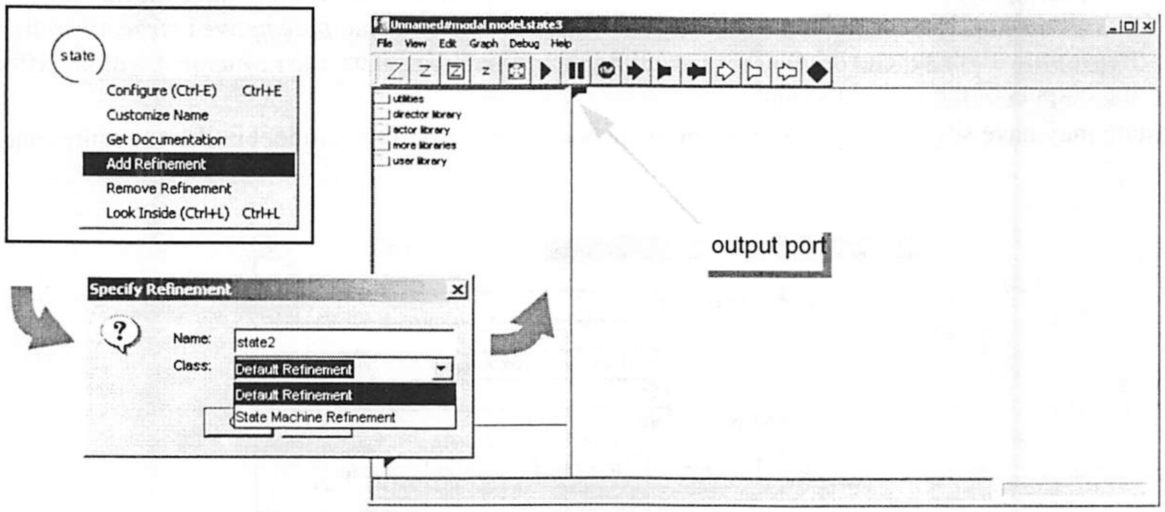


FIGURE 2.46. Adding a refinement to a state.

2.8.4 Execution Semantics

The behavior of a refinement is simple. When the modal model is executed, the following sequence of events occurs:

- For any transitions out of the current state for which *preemptive* is *true*, the guard is evaluated. If exactly one such guard evaluates to *true*, then that transition is chosen. The *output actions* of the transition are executed, and the *refinements* of the transition (if any) are executed, followed by the *set actions*.
- If no preemptive transition evaluated to true, then the refinement of the current state, if there is one, is evaluated at the current time step.
- Once the refinement has been evaluated (and it has possibly updated its output values), the guard expressions on all the outgoing transitions of the current state are evaluated. If none is true, the execution is complete. If one is true, then that transition is taken. If more than one is true, then an exception is thrown (the state machine is nondeterministic). What it means for the transition to be "taken" is that its *output actions* are executed, its *refinements* (if any) are executed, and its *set actions* are executed.
- If *reset* is true on a transition that is taken, then the refinement of the destination mode (if there is one) is initialized.

There is a subtle distinction between the *output actions* and the *set actions*. The intent of these two fields on the transition is that *output actions* are used to define the values of output ports, while *set actions* are used to define state variables in the refinements of the destination modes. The reason that these two actions are separated is that while solving a continuous-time system of equations, the solver may speculatively execute models at certain time steps before it is sure what the next time step will be. The *output actions* make no permanent changes to the state of the system, and hence can be executed during this speculative phase. The *set actions*, however, make permanent changes to the state variables of the destination refinements, and hence are not executed during the speculative phase.

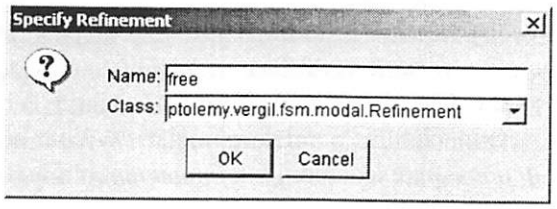
2.9 Using the Plotter

Several of the plots shown above have flaws that can be fixed using the features of the plotter. For instance, the plot shown in figure 2.33 has the default (uninformative) title, the axes are not labeled, and the horizontal axis ranges from 0 to 255[1], because in one iteration, the *Spectrum* actor produces 256 output tokens. These outputs represent frequency bins that range between $-\pi$ and π radians per second.

1. **Hint:** Notice the "x10^2" at the bottom right, which indicates that the label "2.5" stands for "250".

The *SequencePlotter* actor has some pertinent parameters, shown in figure 2.47. 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 2.48.

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 2.49, filled in with values that result in the plot shown in figure 2.50. Most of these are self-explanatory, but the following pointers may be useful:



Name: free
Class: ptolemy.vergil.fsm.modal.Refinement

FIGURE 2.47. Dialog for creating a refinement of a state.

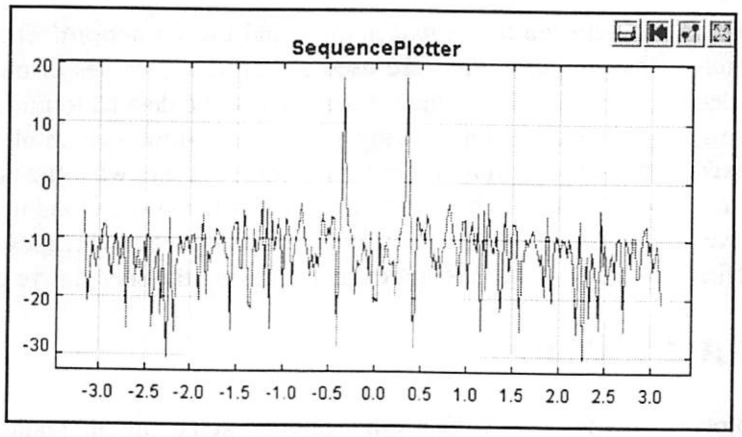
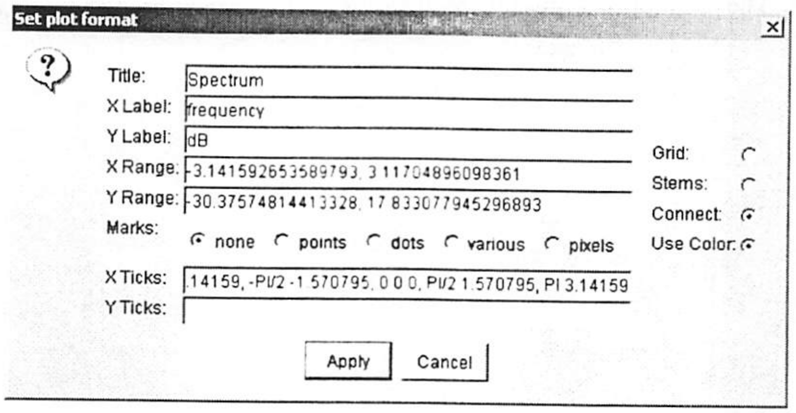


FIGURE 2.48. Better labeled plot, where the horizontal axis now properly represents the frequency values.



Title: Spectrum
X Label: frequency
Y Label: dB
X Range: -3.141592653589793, 3.117048960983661
Y Range: -30.375748144133 28, 17.833077945296893
Marks: [✓ none] points dots various pixels
X Ticks: .14159, -PI/2 -1.570795, 0 0.0, PI/2 1.570795, PI 3.14159
Y Ticks:
Grid:
Stems:
[✓ Connect:]
[✓ Use Color:]

FIGURE 2.49. Format control window for a plot.

- 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"
- 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.

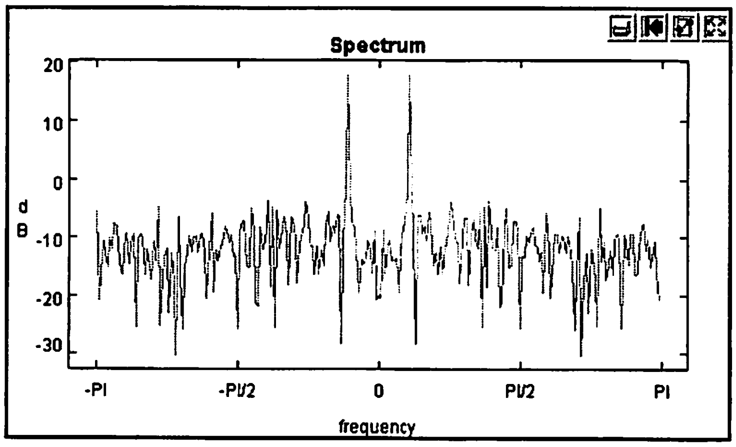


FIGURE 2.50. Still better labeled plot.

3

Expressions

Authors: *Edward A. Lee*
Xiaojun Liu
Steve Neuendorffer
Neil Smyth
Yuhong Xiong

3.1 Introduction

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(2π (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.

3.1.1 Expression Evaluator

Vergil provides an interactive *expression evaluator*, which is accessed through the File:New menu. This operates like an interactive command shell, and is shown in figure 3.1. 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.

3.2 Simple Arithmetic Expressions

3.2.1 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, MaxInt, MinInt, MaxLong, MinLong, MaxDouble, MinDouble. For example,

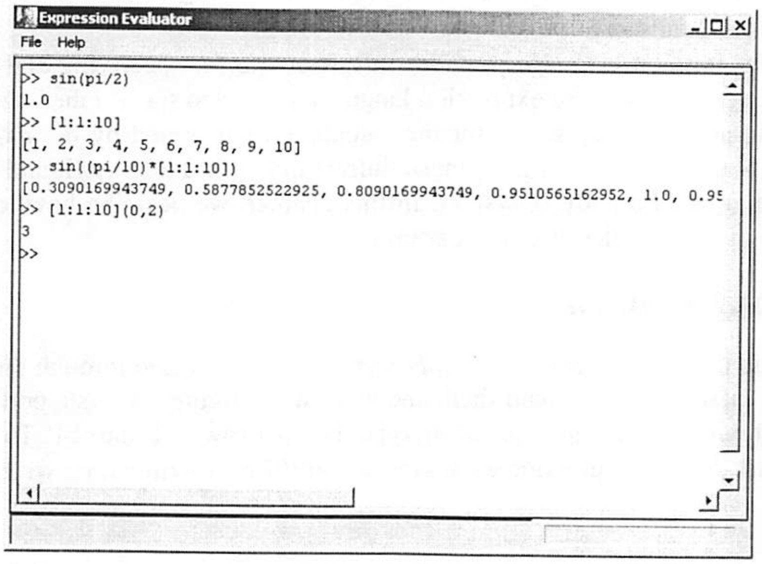
```
PI/2.0
```

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 the square root of –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 without decimal points followed by the character "l" (el) or "L" are of type *long*. 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 not quite the same as the Java byte, which has a value between -128 and 127.

Numbers of type *int*, *long*, 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.

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



```
>> sin(pi/2)
1.0
>> [1:1:10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>> sin((pi/10)*[1:1:10])
[0.3090169943749, 0.5877852522925, 0.8090169943749, 0.9510565162952, 1.0, 0.95
>> [1:1:10](0,2)
3
>>
```

FIGURE 3.1. Expression evaluator, which is accessed through the File:New menu.

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 quotes, "...", is interpreted as a string constant. The following built-in string-valued constants are defined:

TABLE 1: String-valued constants defined in the expression language.

Variable name	Meaning	Property name	Example under Windows
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

The value of these variables is the value of the Java virtual machine property, such as *user.home*. The properties *user.dir* and *user.home* are standard in Java. Their values are platform dependent; see the documentation for the java.lang.System.getProperties() method for details. 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. Vergil will display all the Java properties if you invoke JVM Properties in the View 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.

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 something like:

```
{CWD="C:\ptII\ptolemy\data\expr", E=2.718281828459, HOME="C:\Documents
and Settings\eal", Infinity=Infinity, MaxDouble=1.7976931348623E308,
MaxInt=2147483647, MaxLong=9223372036854775807L,
MaxUnsignedByte=255ub, MinDouble=4.9E-324, MinInt=-2147483648,
MinLong=-9223372036854775808L, MinUnsignedByte=0ub, NaN=NaN,
NegativeInfinity=-Infinity, PI=3.1415926535898, PTII="c:\ptII",
PositiveInfinity=Infinity, boolean=false, complex=0.0 + 0.0i,
double=0.0, e=2.718281828459, false=false, fixedpoint=fix(0.0,2,1),
general=present, i=0.0 + 1.0i, int=0, j=0.0 + 1.0i, long=0L, matrix=[],
object=object(null),pi=3.1415926535898, scalar=present, string="",
true=true, unknown=present, unsignedByte=0ub}
```

3.2.2 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" is 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 "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.

3.2.3 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 an *int* or an *unsignedByte*.

The *unsignedByte*, *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 if 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 5 on page 96). 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 page 90). 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, *int* 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.
```

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 integer types and *fixedpoint*, overflow results in wraparound. For instance, while the value of `MaxInt` is 2147483647, 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*, *int* and *long* (but not *fixedpoint*, *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.

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*, *int*, and *long*.

3.2.4 Comments

In expressions, anything inside `/*...*/` is ignored, so you can insert comments.

3.3 Uses of Expressions

3.3.1 Parameters

The values of most parameters of actors can be given as expressions[1]. 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 3.9. Adding parameters to actors is straightforward, as explained in the previous chapter.

3.3.2 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

1. 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.

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.

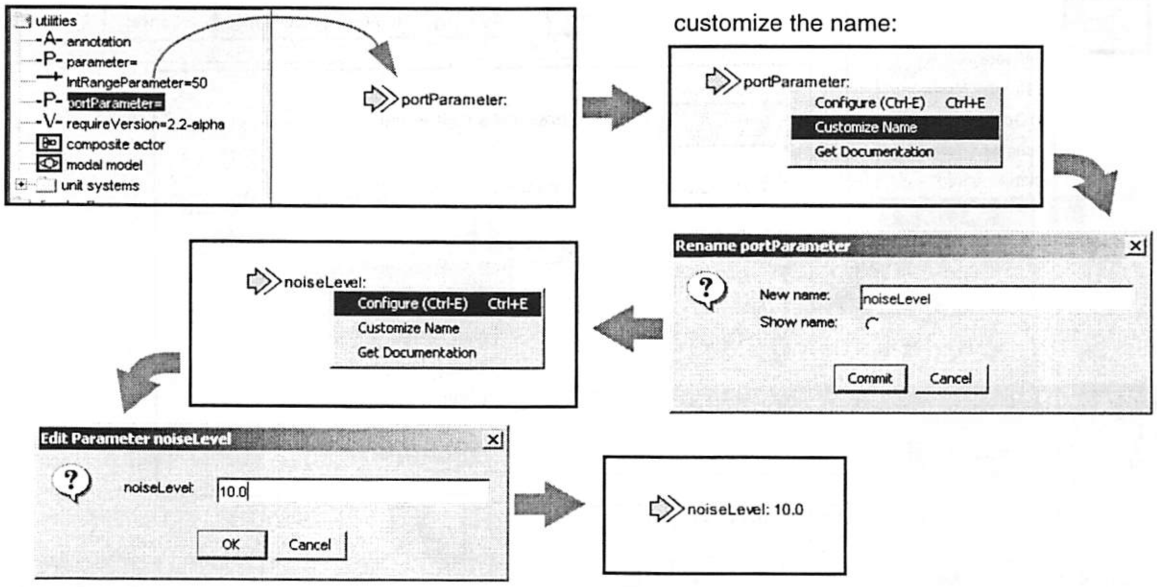
A PortParameter might be contained by an atomic actor or a composite actor. To put one in a composite actor, drag it into a model from the *utilities* library, as shown in figure 3.2. The resulting icon is actually a combination of two icons, one representing the port, and the other representing the parameter. These can be moved separately, but doing so might create confusion, so we recommend selecting both by clicking and dragging over the pair and moving both together.

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 Name," as shown in figure 3.2. In the figure, the name is set to "noiseLevel." Then set the default value by either double clicking or selecting "Configure." 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* library in Vergil. It is shown in figure 3.3. 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.

3.3.3 Expression Actor

The *Expression* actor is a particularly useful actor found in the *math* library. By default, it has one output and no inputs, as shown in Figure 3.4(a). The first step in using it is to add ports, as shown in (b) and (c), resulting in a new icon as shown in (d). Note: In (c) when you click on Add, you will be prompted for a Name (pick one) and a Class. Leave the Class entry blank and click OK. You then specify an expression using the port names, as shown in (e), resulting in the icon shown in (f).



New name: noiseLevel

noiseLevel: 10.0

FIGURE 3.2. 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.

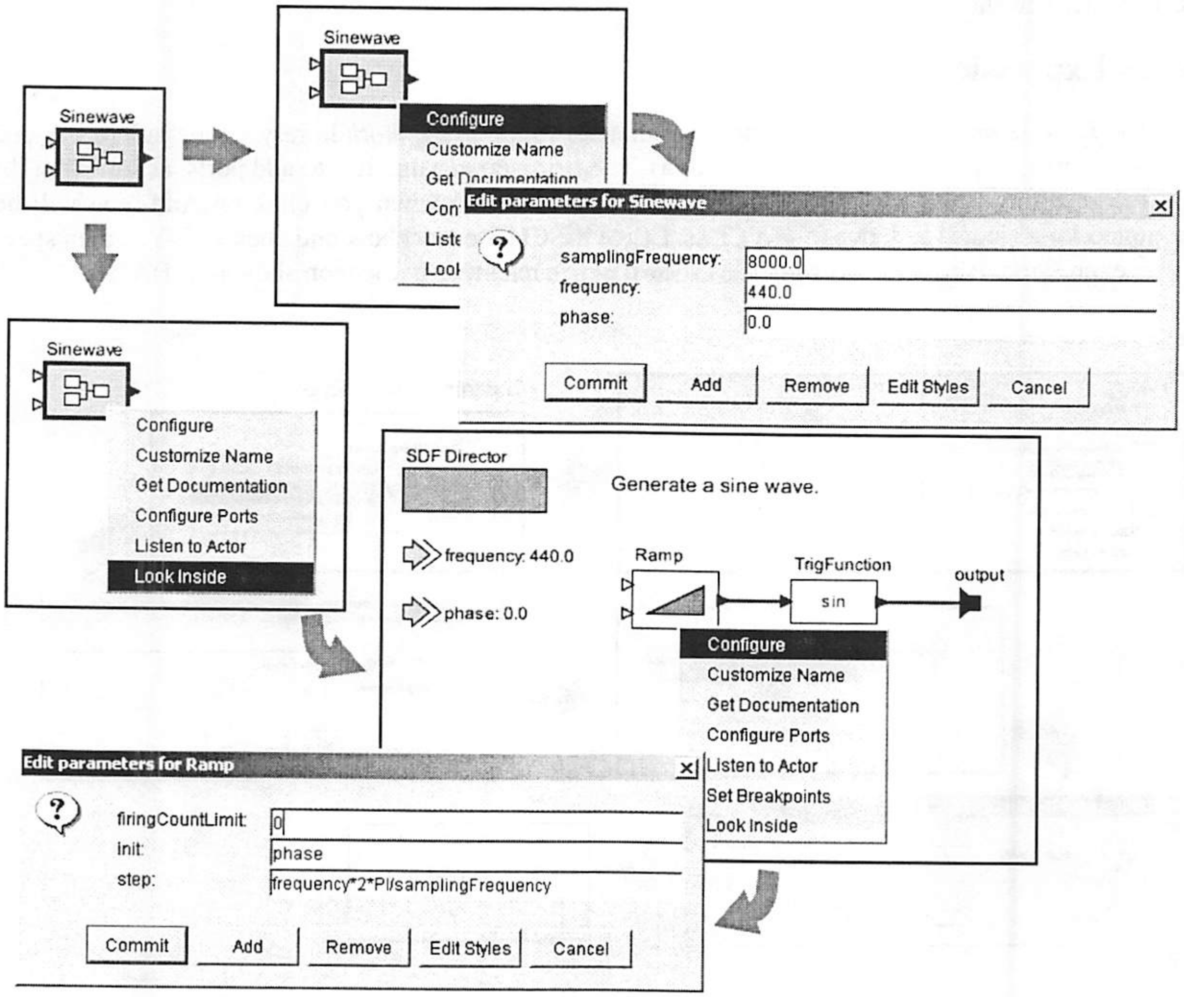
3.3.4 State Machines

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.

3.4 Composite Data Types

3.4.1 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 *string*. The types are denoted "`{int}`" and "`{string}`" respectively. An array is an ordered list of tokens of any type, with the only constraint being that the elements all have



samplingFrequency: 8000.0
frequency: 440.0
phase: 0.0

Generate a sine wave.

frequency: 440.0
phase: 0.0

firingCountLimit: 0
init: phase
step: frequency*2*PI/samplingFrequency

FIGURE 3.3. Sinewave actor, showing its port parameters, and their use at the lower level of the hierarchy.

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}
```

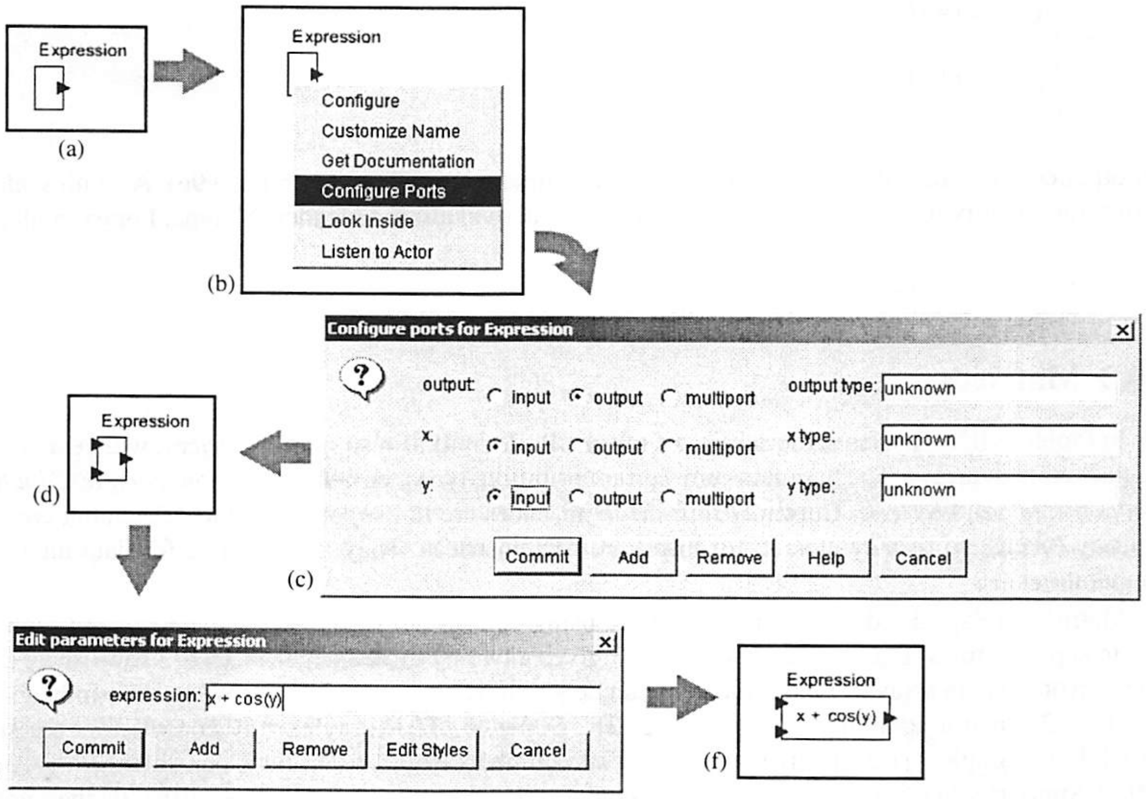
has value

```
{1.0, 2.3}
```

Its type is `{double}`. 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
```

which yields 2.3. Note that indexing begins at 0. Of course, if *name* is the name of a variable in scope



output: ( ) input (•) output ( ) multiport — output type: unknown
x: (•) input ( ) output ( ) multiport — x type: unknown
y: (•) input ( ) output ( ) multiport — y type: unknown

expression: x + cos(y)

FIGURE 3.4. Illustration of the *Expression* actor.

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}
```

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 5 on page 96). 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
```

3.4.2 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 *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).

Reference to elements of matrices have the form "*matrix*(*n*, *m*)" or "*name*(*n*, *m*)" where *name* 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, as seen in the expression evaluator (see figure 3.1),

```
>> [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 6 on page 98). 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 6 on page 98). A matrix can be raised to an *int* 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 5 on page 96). 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
>>
```

3.4.3 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 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).

Fields may be accessed using the period operator. For example,

```
{a=1,b=2}.a
```

yields 1. You can optionally write this as if it were a method call:

```
{a=1,b=2}.a()
```

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. Hence

```
>> {a=1, b=2}=={b=2, a=1}
true
```

Moreover, 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}
```

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
>>
```

3.5 Invoking Methods

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 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()
```

3.6 Defining Functions

The expression language supports definition of functions. 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
>>
```

Functions can be passed as arguments to certain "higher-order functions" that have been defined (see table Table 9 on page 102). 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}
```

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. Consider the model shown in figure 3.5. 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.

A more elaborate use is shown in figure 3.6. 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 adds the output of the Ramp to the square of the output of the Ramp.

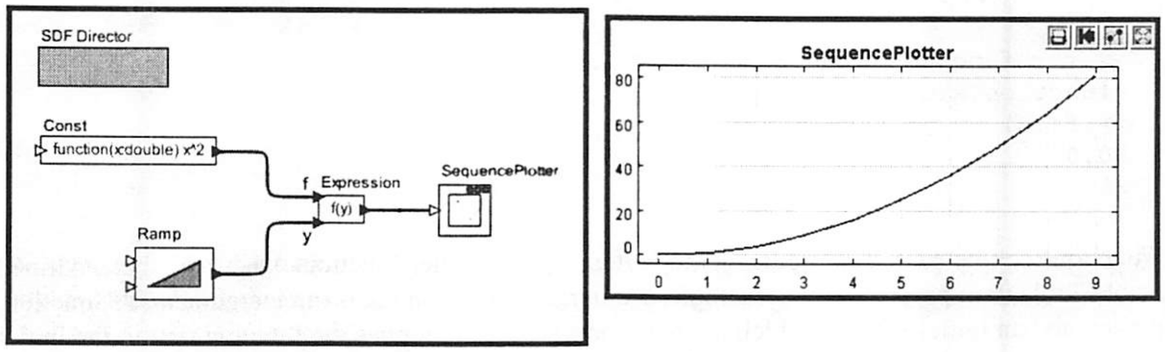


FIGURE 3.5. Example of a function being passed from one actor to another.

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:gen-
eral, 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.

3.7 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 2 on page 88, which together provide a rich set[1]. The functions currently available are shown in the tables in the appendix, which also show the argument types and return types.

In most cases, a function that operates on scalar arguments can also operate on arrays and matrices.

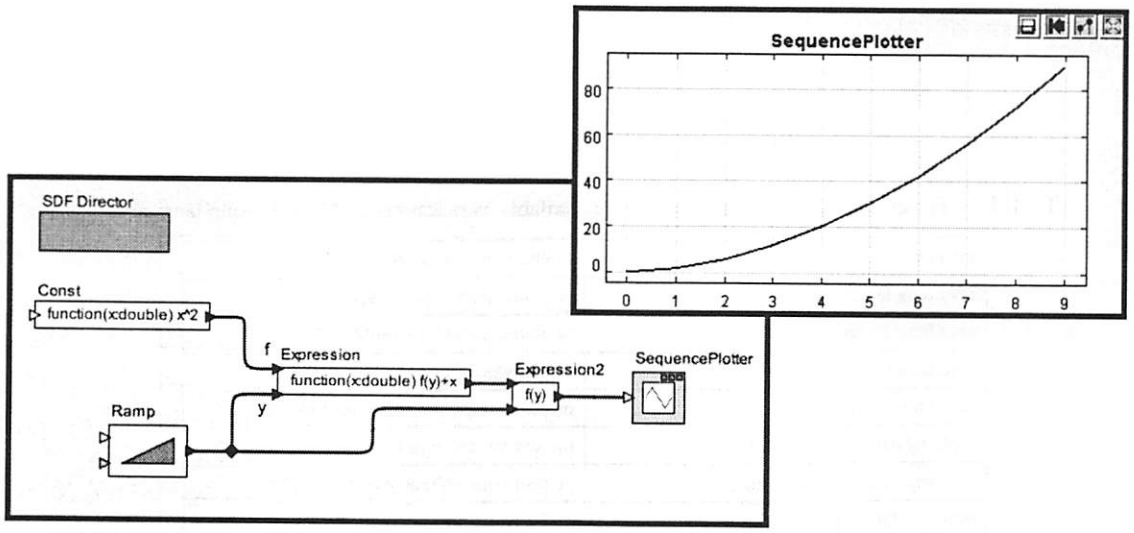


FIGURE 3.6. More elaborate example with functions passed between actors.

1. Moreover, the set of available 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).

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* 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 4 on page 95 shows trigonometric functions. Note that these operate on *double* or *complex*, and hence on *int* and *unsignedByte*, which can be losslessly converted to *double.* The result will always be double. For example,

```
>> cos(0)
1.0
```

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 5 on page 96 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* 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,

TABLE 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	

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 6 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 matricesTable 7 shows utility functions for evaluating expressions given as strings or representing numbers as strings. Of these, the eval() function is the most flexible (see page 89).

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 property user.dir), the 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 functions random() and gaussian() shown in Table 5 on page 96 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 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.

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 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

$$X_k = s_k \sum_{n=0}^{N-1} x_n \cos\left((2n+1)k\frac{\pi}{2D}\right) \tag{4}$$

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 3: Normalization options for the DCT function

Name	Third argument	Normalization
Normalized	0	$s_k = \begin{cases} 1/\sqrt{2}; & k=0 \\ 1; & \text{otherwise} \end{cases}$
Unnormalized	1	$s_k = 1$
Orthonormal	2	$s_k = \begin{cases} 1/\sqrt{D}; & k=0 \\ \sqrt{2/D}; & \text{otherwise} \end{cases}$

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

$$x_n = \sum_{k=0}^{N-1} s_k X_k \cos\left((2n+1)k\frac{\pi}{2D}\right). \tag{5}$$

3.8 Fixed Point Numbers

Ptolemy II includes a preliminary 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.

3.9 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 3.7. This unit system is called BasicUnits; the units it defines can be examined by double clicking on its icon, or by invoking Configure, as shown in figure 3.8. 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".

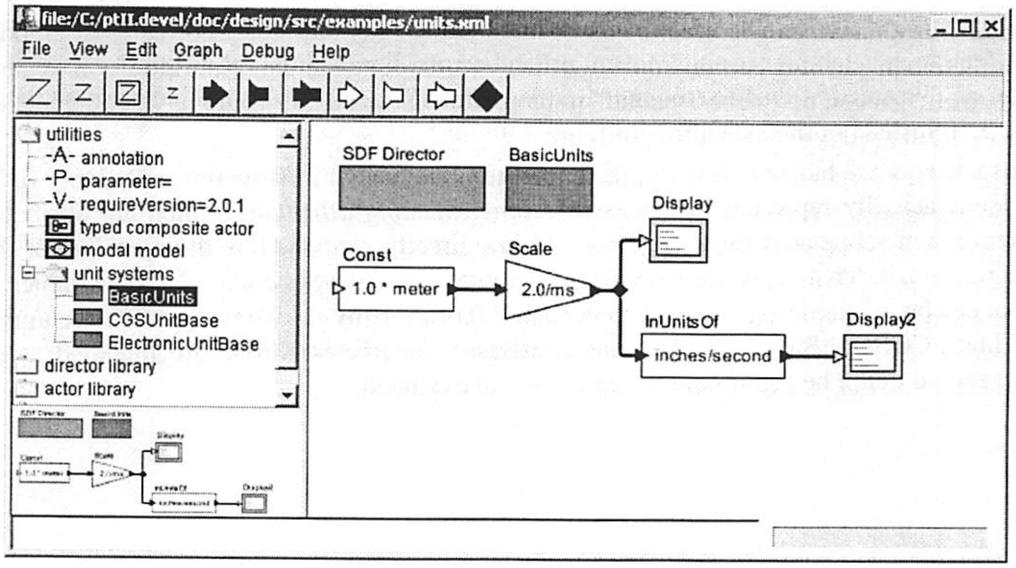
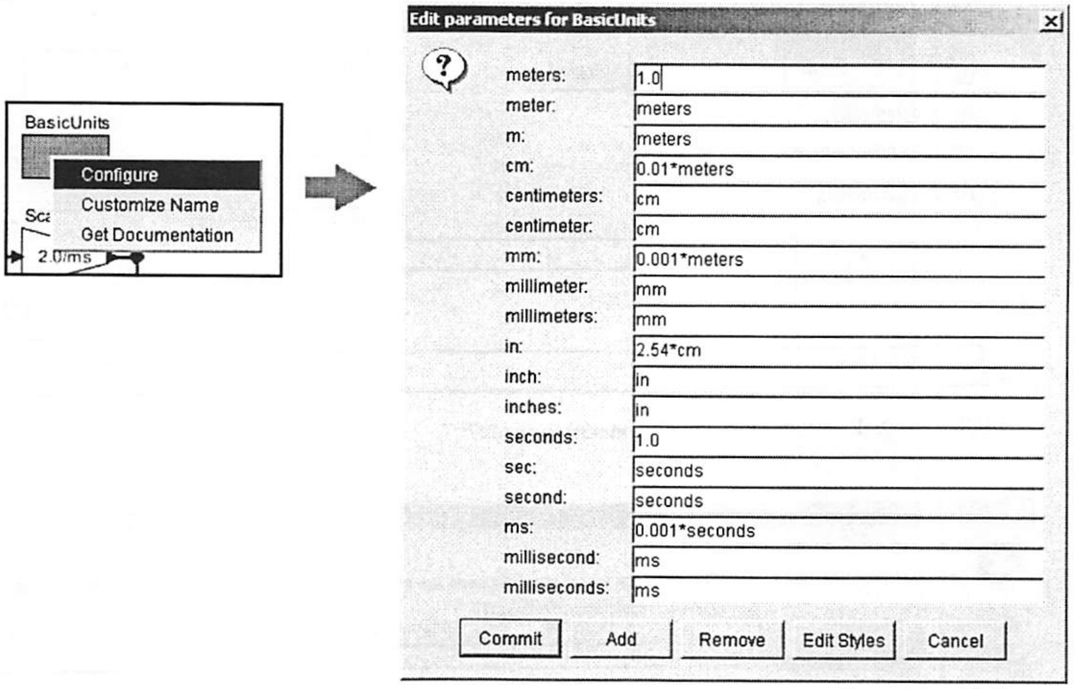


FIGURE 3.7. Example of a model that includes a unit system.



meters:	1.0
meter:	meters
m:	meters
cm:	0.01*meters
centimeters:	cm
centimeter:	cm
mm:	0.001*meters
millimeter:	mm
millimeters:	mm
in:	2.54*cm
inch:	in
inches:	in
seconds:	1.0
sec:	seconds
second:	seconds
ms:	0.001*seconds
millisecond:	ms
milliseconds:	ms

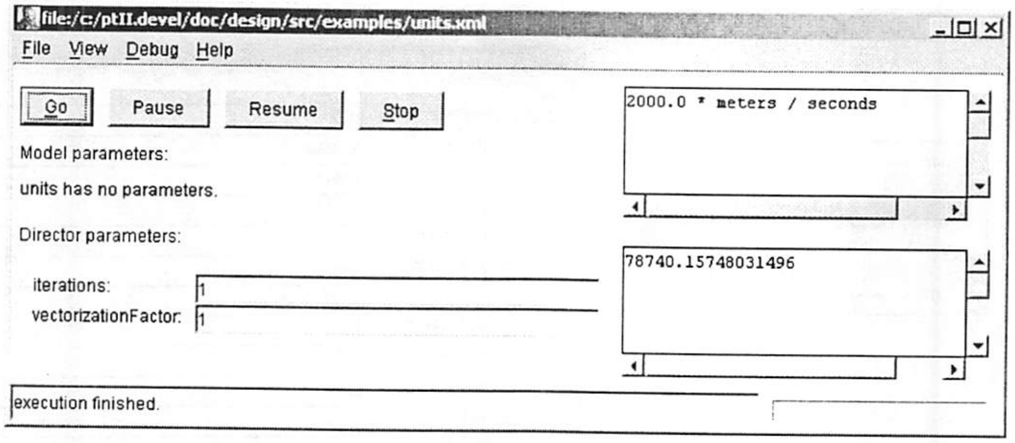
FIGURE 3.8. Units defined in a units system can be examined by invoking Configure on its icon.

In the example in figure 3.7, 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 3.9, top display. The canonical units for length are meters, and for time are seconds.

In figure 3.7, we also take the result and feed it to the *InUnitsOf* actor, which performs 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 3.7 we had entered "seconds/inch" instead of "inches/second" in the InUnitsOf actor, we would have gotten the exception in figure 3.10 instead of the execution in figure 3.9.

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. In version 2.0 of Ptolemy II, 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.



Model parameters:

units has no parameters.

Director parameters:

iterations: 1

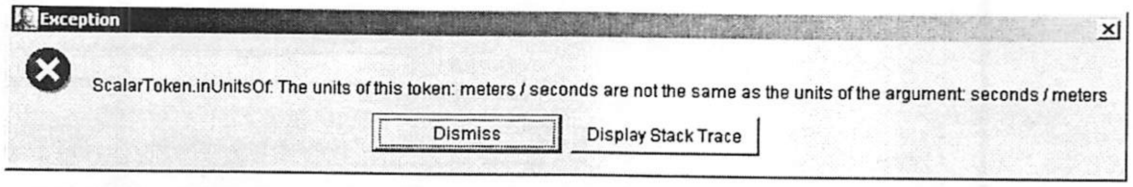
vectorizationFactor: 1

```
2000.0 * meters / seconds
```

```
78740.15748031496
```

execution finished.

FIGURE 3.9. Result of running the model in figure 3.7.



ScalarToken.inUnitsOf: The units of this token: meters / seconds are not the same as the units of the argument: seconds / meters

FIGURE 3.10. Example of an exception resulting from a units mismatch.

Appendix A. Tables of Functions

In this appendix, we tabulate the functions available in the expression language. Further explanation of many of these functions is given in section section 3.7 above.

A.1 Trigonometric Functions

TABLE 4: Trigonometric functions.

function	argument type(s)	return type	description
acos	*double* in the range [-1.0, 1.0] or *complex*	*double* in the range [0.0, pi] or NaN if out of range or *complex*	arc cosine *complex* case: $\mathrm{acos}(z) = -i\log(z + i\mathrm{sqrt}(1 - z^2))$
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: $\mathrm{asin}(z) = -i\log(iz + \mathrm{sqrt}(1 - z^2))$
atan	*double* or *complex*	*double* in the range [-pi/2, pi/2] or *complex*	arc tangent *complex* case: $\mathrm{atan}(z) = -\frac{i}{2}\log\left(\frac{i-z}{i+z}\right)$
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: $\mathrm{acosh}(z) = \log(z + \mathrm{sqrt}(z^2 - 1))$
asinh	*double* or *complex*	*double* or *complex*	hyperbolic arc sine *complex* case: $\mathrm{asinh}(z) = \log(z + \mathrm{sqrt}(z^2 + 1))$
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: $\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$

A.2 Basic Mathematical Functions

TABLE 5: Basic mathematical functions

function	argument type(s)	return type	description
abs	*double* or *int* or *long* or *complex*	*double* or *int* or *long* (*complex* returns double)	absolute value complex case: $\text{abs}(a + ib) = \lvert z \rvert = \sqrt{a^2 + b^2}$
angle	*complex*	*double* in the range [-pi, pi]	angle or argument of the complex number: $\angle z$
ceil	*double*	*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, infinity] 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 {*double*} or [*double*]	one or more Gaussian random variables with the specified mean and standard deviation (see page 89).
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: $\log(z) = \log(\text{abs}(z) + i\text{angle}(z))$
log10	*double*	*double*	log base 10
log2	*double*	*double*	log base 2
max	*double, double* or *int, int* or *long, long* or *unsignedByte, unsignedByte* or {*double*} or {*int*} or {*long*} or {*unsignedByte*}	*double* or *int* or *long* or *unsignedByte*	maximum
min	*double, double* or *int, int* or *long, long* or *unsignedByte, unsignedByte* or {*double*} or {*int*} or {*long*} or {*unsignedByte*}	*double* or *int* or *long* or *unsignedByte*	minimum

TABLE 5: Basic mathematical functions

function	argument type(s)	return type	description
neighborhood	*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.
pow	*double, double* or *complex, complex*	double or complex	first argument to the power of the second
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 page 89)
real	*complex*	*double*	real part
remainder	*double, double*	*double*	remainder after division, according to the IEEE 754 floating-point standard (see page 90).
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: $\mathrm{sqrt}(z) = \sqrt{\lvert z\rvert}\left(\cos\left(\frac{\angle z}{2}\right) + i\sin\left(\frac{\angle z}{2}\right)\right)$
toDegrees	*double*	*double*	convert radians to degrees
toRadians	*double*	*double*	convert degrees to radians

3.10 Matrix, Array, and Record Functions.

TABLE 6: Functions that take or return matrices, arrays, or records.

function	argument type(s)	return type	description
arrayToMatrix	{*type*}, int, int	[*type*]	Create a matrix from the specified array with the specified number of rows and columns
conjugateTranspose	[*complex*]	[*complex*]	Return the conjugate transpose of the specified matrix.
createSequence	*type*, *type*, int	{*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* or	[*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	{*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
hilbert	*int*	[*double*]	Return a square Hilbert matrix, where $A_{ij} = 1/(i+j+1)$. A Hilbert matrix is nearly, but not quite singular.
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*]	{*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.
orthogonalizeColumns	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthogonal columns.
orthogonalizeRows	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthogonal rows.
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*	{*type*}	Create an array by repeating the specified token the specified number of times.
sum	{*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.

TABLE 6: Functions that take or return matrices, arrays, or records.

function	argument type(s)	return type	description
transpose	[*type*]	[*type*]	Return the transpose of the specified matrix.
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.

A.3 Functions for Evaluating Expressions

TABLE 7: Utility functions for evaluating expressions

function	argument type(s)	return type	description
eval	*string*	any type	evaluate the specified expression (see page 89).
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* or *long, int*	*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 page 89).

A.4 Signal Processing Functions

TABLE 8: Functions performing signal processing operations

function	argument type(s)	return type	description
convolve	*{double}*, *{double}* or *{complex}*, *{complex}*	*{double}* or *{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	*{double}* or *{double}*, *int* or *{double}*, *int*, *int*	*{double}*	Return the discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see page 90).
downsample	*{double}*, *int* or *{double}*, *int*, *int*	*{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	*{double}* or *{complex}* or *{double}*, *int* *{complex}*, *int*	*{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.
generateBartlettWindow	*int*	*{double}*	Return a 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: $w(n) = \begin{cases} 2\frac{n}{M}; & \text{if } 0 \le n \le \frac{M}{2} \\ 2-2\frac{n}{M}; & \text{if } \frac{M}{2} \le n \le M \end{cases}$
generateBlackmanWindow	*int*	*{double}*	Return a Blackman window with the specified length. For length $M+1$, the formula is: $w(n) = 0.42 + 0.5\cos(2\pi n/M) + 0.08\cos(4\pi n/M)$
generateBlackmanHarrisWindow	*int*	*{double}*	Return a Blackman-Harris window with the specified length. For length $M+1$, the formula is: $w(n) = 0.35875 + 0.48829\cos(2\pi n/M) + 0.14128\cos(4\pi n/M) + 0.01168\cos(6\pi n/M)$
generateGaussianCurve	*double*, *double*, *int*	*{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).
generateHammingWindow	*int*	*{double}*	Return a Hamming window with the specified length. For length $M+1$, the formula is: $w(n) = 0.54 - 0.46\cos(2\pi n/M)$
generateHanningWindow	*int*	*{double}*	Return a Hanning window with the specified length. For length $M+1$, the formula is: $w(n) = 0.5 - 0.5\cos(2\pi n/M)$

TABLE 8: Functions performing signal processing operations

function	argument type(s)	return type	description
generatePolynomialCurve	*{double}, double, double, int*	*{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.
generateRaisedCosinePulse	*double, double, int*	*{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: $h(t) = \frac{\sin(\pi t/T)}{\pi t/T} \times \frac{\cos(x\pi t/T)}{1-(2xt/T)^2}$, 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 excessBandwidth of 0.0, this pulse is a sinc pulse.
generateRectangularWindow	*int*	*{double}*	Return an array filled with 1.0 of the specified length. This is a rectangular window.
IDCT	*{double}* or *{double}, int* or *{double}, int, int*	*{double}*	Return the inverse discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see page 90).
IFFT	*{double}* or *{complex}* or *{double}, int* *{complex}, int*	*{complex}*	Return the 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.
nextPowerOfTwo	*double*	*int*	Return the next power of two larger than or equal to the argument.
poleZeroToFrequency	*{complex}, {complex}, complex, int*	*{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.

TABLE 8: Functions performing signal processing operations

function	argument type(s)	return type	description
toDecibels	*double*	*double*	Return $20 \times \log_{10}(z)$, where z is the argument.
unwrap	*{double}*	*{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	*{double}, int*	*{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 $n > 0$.

A.5 I/O Functions and Other Miscellaneous Functions

TABLE 9: Miscellaneous functions.

function	argument type(s)	return type	description
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.
freeMemory	none	*long*	Return the approximate number of bytes available for future memory allocation.
iterate	*function, int, type*	*{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, {type}*	*{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.
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.
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.

4

Actor Libraries

Authors: *Elaine Cheong*
Christopher Hylands
Edward A. Lee
Steve Neuendorffer
Yuhong Xiong

Contributors: *Chamberlain Fong*
Mudit Goel
Bart Kienhuis
Edward A. Lee
Michael Leung
Jie Liu
Xiaojun Liu
Sarah Packman
Shankar Rao
Michael Shilman
Jeff Tsay
Brian K. Vogel
Paul Whitaker

4.1 Overview

Ptolemy II focuses on component-based design. In this design approach, components are aggregated and connected to construct a model. One of the advantages of component-based design is that reuse of components becomes possible. Polymorphism is one of the key tenets of object-oriented design. It refers to the ability of a component to adapt in a controlled way to the type of data being supplied. For example, an addition operation is realized differently when adding vectors than when adding scalars. In Ptolemy II, use of polymorphism maximizes the potential for reuse of components.

We call this classical form of polymorphism *data polymorphism*, because objects are polymorphic with respect to data types. A second form of polymorphism, introduced in Ptolemy II, is *domain polymorphism*, where a component adapts in a controlled way to the protocols that are used to exchange data between components. For example, an addition operation can accept input data delivered by any of a number of mechanisms, including discrete events, rendezvous, and asynchronous message passing.

Ptolemy II includes libraries of polymorphic actors that use both kinds of polymorphism to maximize reusability. Actors from these libraries can be used in a broad range of domains, where the domain provides the communication protocol between actors. In addition, most of these actors are data polymorphic, meaning that they can operate on a broad range of data types. In general, writing data and domain polymorphic actors is considerably more difficult than writing more specialized actors. This chapter discusses some of the issues.

4.2 Actor Classes

Figure 4.1 shows a UML static structure diagram for the key classes in the ptolemy.actor.lib package (see appendix A of chapter 1 for an introduction to UML). All the classes in figure 4.1 extend TypedAtomicActor, except TimedActor and SequenceActor, which are interfaces. TypedAtomicActor is in the ptolemy.actor package, and is described in more detail in volume 2, on software architecture. For our purposes here, it is sufficient to know that TypedAtomicActor provides a base class for actors with ports where the ports carry typed data (encapsulated in objects called *tokens*).

None of the classes in figure 4.1 have any methods, except those inherited from the base classes (which are not shown). The classes in figure 4.1 do, however, have public members, most of which are instances of TypedIOPort or Parameter. By convention, actors in Ptolemy II expose their ports and parameters as public members, and much of the functionality of an actor is accessed through its ports and parameters.

Many of the actors are *transformers*, which extend the Transformer base class. These actors read input data, modify it in some way, and produce output data. Some other actors that also have this character, such as AddSubtract, MultiplyDivide, and Expression, do not extend Transformer because they have somewhat unconventional port names. These actors are represented in figure 4.1 by the box labeled "... Other Actors ...".

The stacked boxes labeled "... Transformers ..." and "... Other Actors ..." in figure 4.1 are not standard UML. They are used here to refer to a set of actors that are listed below. There are too many actors to show them individually in the static structure diagram. The diagram would lose its utility because of the resulting clutter.

Most of the library actors can be used in any domain. Some domains, however, can only execute a subset of the actors in this library. For example, the CT (continuous time) domain, which solves ordinary differential equations, may present data to actors that represent arbitrarily spaced samples of a continuous-time signal. For such signals, the data presented to an actor cannot be assumed by the actor to be a sequence, since the domain determines how closely spaced the samples are. For example, the SampleDelay actor would produce unpredictable results, since the spacing of samples is likely to be uneven over time.

The TimedActor and SequenceActor interfaces are intended to declare assumptions that the actor makes about the inputs. They are empty interfaces (i.e., they contain no methods), and hence they are used only as markers. An actor that implements SequenceActor declares that it assumes its inputs are sequences of distinct data values, and that it will produce sequences of distinct data values as outputs. In particular, the input must not be a continuous-time waveform. Thus, any actor that will not work properly in the CT domain should declare that it implements this interface[1]. Most actors do not implement SequenceActor, because they do not care whether the input is a sequence.

An actor that implements the TimedActor interface declares that the current time in a model execution affects its behavior. Currently, all domains can execute actors that implement TimedActor, because all directors provide a notion of current time. However, the results may not be what is expected. The SDF (synchronous dataflow) domain, for example, does not advance current time. Thus,

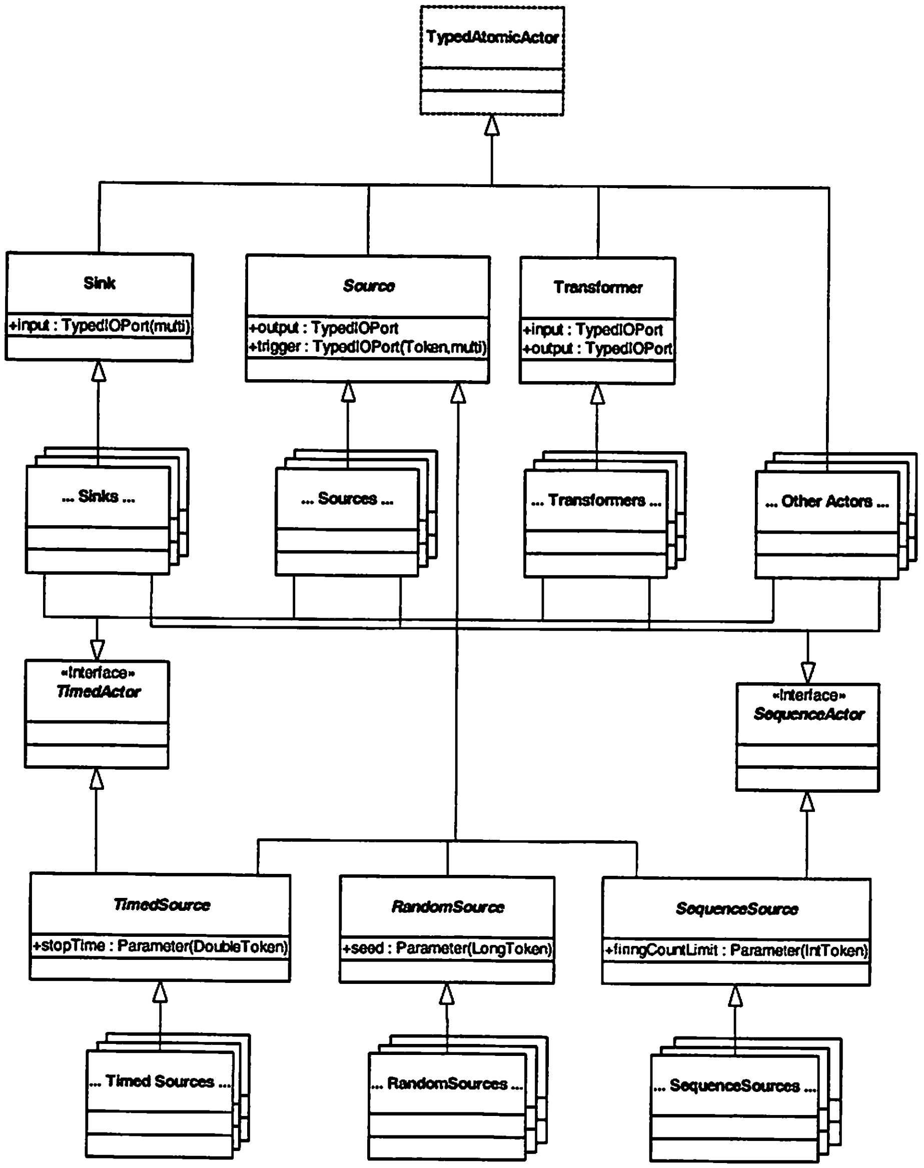


FIGURE 4.1. Key actor base classes and interfaces.

1. Unfortunately, a scan of the current actor library (as of version 3.0) will reveal that we have not been very rigorous about this, and many actors that make a sequential assumption about the input fail to implement this interface. We are working on a more rigorous way of making this distinction, based on the concept of behavioral types.

if SDF is the top-level domain, the current time will always be zero, which is likely to lead to some confusion with timed actors.

4.3 Actor Summaries

In this section, we summarize the actors that are provided in the default Vergil actor library, shown at the left-hand side of the window in figure 4.2. Note that this library is organized for user convenience, and the organization does not exactly match the package structure. Here, we give brief descriptions of each actor to give a high-level view of what actors are available in the library. Refer to the class documentation for a complete description of these actors (in Vergil, you can right-click on an icon and select "Get Documentation" to get the detailed documentation for an actor).

It is useful to know some general patterns of behavior:

- Unless otherwise stated, actors will read at most one input token from each input channel of each input port, and will produce at most one output token. No output token is produced unless there are input tokens.
- Unless otherwise stated, actors can operate in all domains except the FSM (finite state machine) domain, where components are instances of the State class. Additionally, actors that implement the SequenceActor or TimedActor interfaces may be rejected by some domains.

4.3.1 Sources

A source actor is a source of tokens. Most source actors extend the Source base class, as shown in figure 4.1. Such actors have a *trigger* input port, which in some domains serves to stimulate an output. In the DE (discrete event) domain, for example, an input at the *trigger* port causes the current value of the source to be produced at the time stamp of the trigger input. The *trigger* port is a multiport, mean-

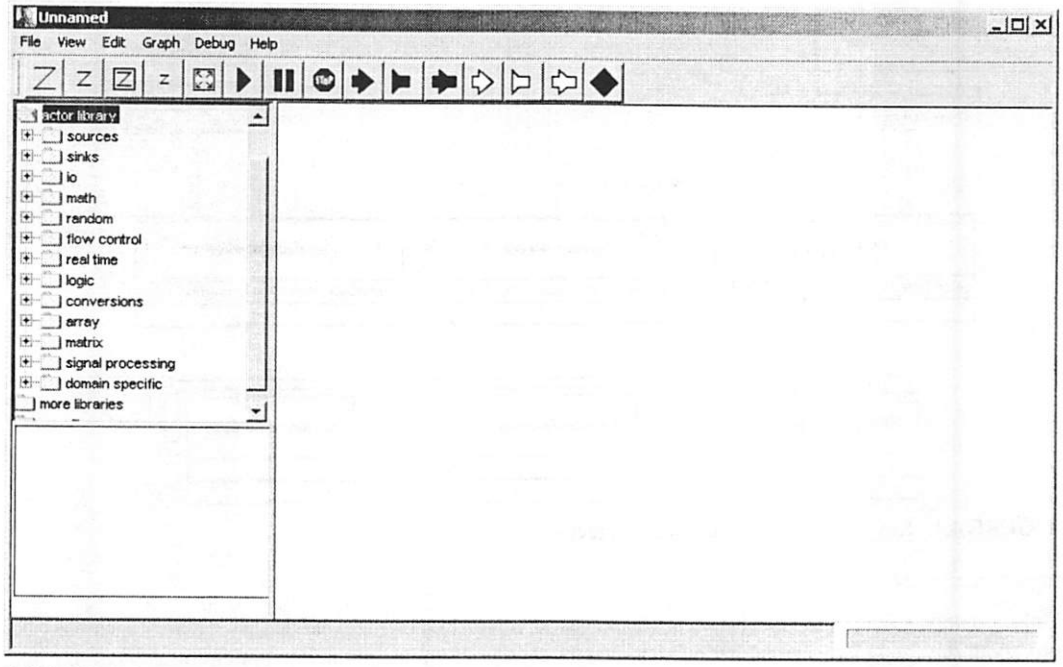


FIGURE 4.2. The default Vergil actor library is shown at the left, expanded to the first level.

ing that multiple channels can be connected to it. The *trigger* port can also be left unconnected in domains that will invoke the actor automatically (SDF, DT, PN, ...). There is no need for a trigger in these domains.

Some source actors use the fireAt() method of the director to request that the actor be fired at particular times in the future. In domains that do not ignore fireAt(), such as DE, such actors will fire repeatedly even if there is no trigger input. In the DE domain, the fireAt() method schedules an event in the future to refire the actor.

Source actors that extend TimedSource have a parameter called *stopTime*. When the current time of the model reaches this time, then the actor requests of the director that this actor not be invoked again. Thus, *stopTime* can be used to generate a finite source signal. By default, the *stopTime* parameter has value 0.0, which indicates unbounded execution.

Source actors that extend SequenceSource have a parameter called *firingCountLimit*. When the number of iterations of the actor reaches this limit, then the actor requests of the director that this actor not be invoked again. Thus, *firingCountLimit* can be used to generate a finite source signal. By default, the *firingCountLimit* parameter has value 0, which indicates unbounded execution.

In some domains, such as SDF and DT, it makes no sense to stop the execution of a single actor. The statically constructed schedule would be disrupted. In these domains, when the specified *stopTime* or *firingCountLimit* is reached, the execution of the entire model will stop.

Some of the most useful actors are *Clock*, which is used extensively in DE models to trigger regularly timed events; *Ramp*, which produces a counting sequence; *Const*, which produces a constant; and *Pulse*, which produces an arbitrary sequence. In Vergil, the source library is divided into *generic sources*, *timed sources*, and *sequence sources*. The first group includes only one source, *Const*, which is agnostic about whether its output is interpreted as a timed output or a sequence output. The other two groups contain actors for which the output is either timed or is logically a sequence.

Generic Sources

Const (extends *Source*): Produce a constant output with value given by **value**.

Timed Sources

Clock (extends *TimedSource*): Produce samples of a piecewise constant signal with the specified values. The transitions between values occur with the specified period and offsets within the period. This actor uses fireAt() to schedule firings when time matches the transition times, and thus will at least produce outputs at these times. To generate a continuous-time clock, you will likely want to use ContinuousClock instead; that version produces two outputs at the transition times, one with the old value and one with the new.

CurrentTime (extends *TimedSource*): Produce an output token with value equal to the current time (the model time when the actor is fired).

PoissonClock (extends *TimedSource*): Produce samples of a piecewise constant signal with the specified values. The transitions between values occur according to a Poisson process (which has random interarrival times with an exponential distribution). This actor uses fireAt() to schedule firings when time matches the transition times, and thus will at least produce outputs at these times.

TimedSinewave (composite actor) Output samples of a sinusoidal waveform taken at *current time* (when the actor is fired). Note that to generate a continuous-time sine wave in the CT domain you probably want to use *ContinuousSinewave* instead.

TriggeredClock (extends *Clock*): This actor is an extension of Clock with a *start* and *stop* input. A token at the *start* input will start the clock. A token at the *stop* input will stop the clock, if it is still running. To generate a continuous-time clock, you will likely want to use *TriggeredContinuousClock* instead; that version produces *two* outputs at the transition times, one with the old value and one with the new.

VariableClock (extends *Clock*): An extension of Clock with an input to dynamically control the period. NOTE: This actor will likely be replaced at some point by a version of Clock with a *period* PortParameter.

Sequence Sources

InteractiveShell (extends *TypedAtomicActor*): This actor creates a command shell on the screen, sending commands that are typed by the user to its output port, and reporting strings received at its input by displaying them. Each time it fires, it reads the input, displays it, then displays a command prompt (which by default is ">>"), and waits for a command to be typed. The command is terminated by an enter or return character, which then results in the command being produced on the output

Interpolator (extends *SequenceSource*): Produce an output sequence by interpolating a specified set of values. This can be used to generate complex, smooth waveforms.

Pulse (extends *SequenceSource*): Produce a sequence of values at specified iteration indexes. The sequence repeats itself when the *repeat* parameter is set to true. This is similar to the Clock actor, but it is not timed. Whenever it is fired, it progresses to the next value in the *values* array, irrespective of the current time.

Ramp (extends *SequenceSource*): Produce a sequence that begins with the value given by *init* and is incremented by *step* after each iteration. The types of *init* and *step* are required to support addition.

Sinewave (composite actor): Output successive samples of a sinusoidal waveform. This is a sequence actor. The timed and continuous versions are *TimedSinewave* and *ContinuousSinewave* respectively.

SketchedSource (implements *SequenceActor*): Output a signal that has been sketched by the user on the screen.

4.3.2 Sinks

Sink actors are the ultimate destinations for tokens. Sink actors have no outputs, and include actors that display data in plots, textual form, or tables.

Many of these actors are shown in figure 4.3, which shows a UML static structure diagram. Several of these sinks have both time-based and sequence-based versions. TimedPlotter, for example, displays a plot of its input data as a function of time. SequencePlotter, by contrast, ignores current time, and uses for the horizontal axis the position of an input token in a sequence of inputs. XYPlotter, on the other hand, uses neither time nor sequence number, and therefore implements neither TimedActor nor SequenceActor. All three are derived from Plotter, a base class with a public member, *plot*, which implements the plot. This base class has a *fillOnWrapup* parameter, which has a boolean value. If the

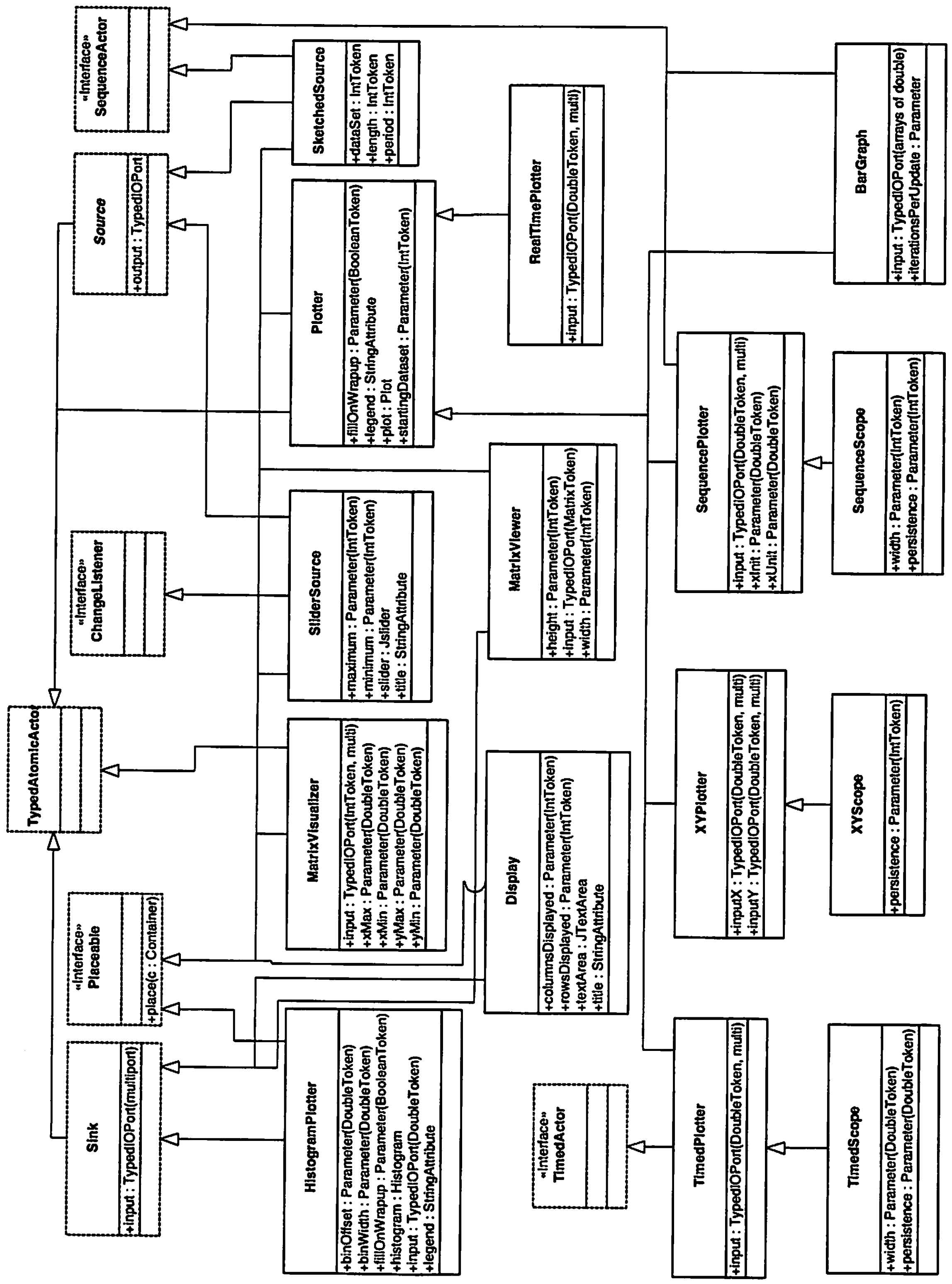

«Interface» SequenceActor
SketchedSource
+dataSet : IntToken
+length : IntToken
+period : IntToken
Source
+output : TypedIOPort
RealTimePlotter
+input : TypedIOPort(DoubleToken, multi)
BarGraph
+input : TypedIOPort(arrays of double)
+iterationsPerUpdate : Parameter
Plotter
+fillOnWrapup : Parameter(BooleanToken)
+legend : StringAttribute
+plot : Plot
+startingDataset : Parameter(IntToken)
SequencePlotter
+input : TypedIOPort(DoubleToken, multi)
+xInit : Parameter(DoubleToken)
+xUnit : Parameter(DoubleToken)
SequenceScope
+width : Parameter(IntToken)
+persistence : Parameter(IntToken)
«Interface» ChangeListener
SliderSource
+maximum : Parameter(IntToken)
+minimum : Parameter(IntToken)
+slider : Jslider
+title : StringAttribute
MatrixViewer
+height : Parameter(IntToken)
+input : TypedIOPort(MatrixToken)
+width : Parameter(IntToken)
TypedAtomicActor
MatrixVisualizer
+input : TypedIOPort(IntToken, multi)
+xMax : Parameter(DoubleToken)
+xMin : Parameter(DoubleToken)
+yMax : Parameter(DoubleToken)
+yMin : Parameter(DoubleToken)
Display
+columnsDisplayed : Parameter(IntToken)
+rowsDisplayed : Parameter(IntToken)
+textArea : JTextArea
+title : StringAttribute
XYPlotter
+inputX : TypedIOPort(DoubleToken, multi)
+inputY : TypedIOPort(DoubleToken, multi)
XYScope
+persistence : Parameter(IntToken)
«Interface» Placeable
+place(c : Container)
Sink
+input : TypedIOPort(multiport)
HistogramPlotter
+binOffset : Parameter(DoubleToken)
+binWidth : Parameter(DoubleToken)
+fillOnWrapup : Parameter(BooleanToken)
+histogram : Histogram
+input : TypedIOPort(DoubleToken)
+legend : StringAttribute
«Interface» TimedActor
TimedPlotter
+input : TypedIOPort(DoubleToken, multi)
TimedScope
+width : Parameter(DoubleToken)
+persistence : Parameter(DoubleToken)

FIGURE 4.3. Organization of actors in the ptolemy.actor.lib.gui package.

value is true (the default), then at the conclusion of the execution of the model, the axes of the plot will be adjusted to just fit the observed data.

All of the sink actors implement the Placeable interface. Actors that implement this interface have graphical widgets that a user of the actor may wish to place on the screen. Vergil constructs a display panel by placing such actors. More customized placement can be achieved by calling the place() method of the Placeable interface in custom Java code.

In Vergil, the sinks library is divided into *generic sinks*, *timed sinks*, and *sequence sinks*. The first group includes sinks that are agnostic about whether their inputs are interpreted as a timed events or as sequence inputs. The other two groups contain actors for which the input is either timed or is logically a sequence.

Generic Sinks

Discard (extends *Sink*): Consume and discard input tokens.

Display (extends *Sink*): Display input tokens in a text area on the screen.

MonitorValue (extends *Sink*): Display input tokens in the icon of the actor in the block diagram. The *value* parameter specifies what to display before any inputs are provided.

Recorder (extends *Sink*): Record all input tokens for later querying (by Java code). This actor is useful for Java code that executes models and then wishes to query for results.

SetVariable (extends *TypedAtomicActor*): Set the value of a variable contained by the container. The change to the value of the variable takes hold at the end of the current iteration. This helps ensure that users of value of the variable will see changes to the value deterministically (independent of the schedule of execution of the actors).

XYPlotter (extends *Plotter*): Display a plot of the data on each *inputY* channel vs. the data on the corresponding *inputX* channel.

XYScope (extends *XYPlotter*): Display a plot of the data on each *inputY* channel vs. the data on the corresponding *inputX* channel with finite persistence.

Timed Sinks

TimedPlotter (extends *Plotter*): Plot inputs as a function of time.

TimedScope (extends *TimedPlotter*): Plot inputs as a function of time in an oscilloscope style.

Sequence Sinks

ArrayPlotter (extends *Plotter*): Plot a sequence of arrays of doubles.

BarGraph (extends *ArrayPlotter*): Plot bar graphs, given arrays of doubles as inputs.

HistogramPlotter (extends *PlotterBase*): Display a histogram of the data on each input channel.

SequencePlotter (extends *Plotter*): Plot the input tokens vs. their index number.

SequenceScope (extends *SequencePlotter*): Plot sequences that are potentially infinitely long in an oscilloscope style.

4.3.3 I/O

The "io" library (see figure 4.2) consists of actors that read and write to the file system or network. Note that the "comm" library under "more libraries" includes a Windows only *SerialComm* actor that communicates with serial and parallel ports.

ArrowKeySensor (extends *TypedAtomicActor*): Pop up a frame that senses arrow keystrokes and produces outputs accordingly.

DatagramReader (extends TypedAtomicActor): Read datagram packets from the network socket specified by localSocketNumber and produce them on the output.

DatagramWriter (extends *TypedAtomicActor*): Send input data received on *data* port as a UDP datagram packet to the network address specified by *remoteAddress* and *remoteSocketNumber.*

ExpressionReader (extends *LineReader*): Read a file or URL, one line at a time, evaluate each line as an expression, and output the token resulting from the evaluation.

ExpressionWriter (extends *LineWriter*): Read input tokens and write them, one line at a time, to a specified file.

LineReader (extends *Source*): Read a file or URL, one line at a time, and output each line as a string token.

LineWriter (extends *LineWriter*): Read input string-valued tokens and write them, one line at a time, to a specified file.

URLDirectoryReader (extends *URLReader*): If the URL names a directory, it outputs the name of each file or subdirectory contained in the directory, and if the URL names a file, then it outputs the name of that file.

4.3.4 Math

The math library (see figure 4.2) consists mostly of transformer actors, each of which calculates some mathematical function. Some of these actors operate on type "scalar", meaning all numerical data types (complex, double, int, long, and fix)[1].

AbsoluteValue (extends *Transformer*): Produce an output on each firing with a value that is equal to the absolute value of the input.

AddSubtract (extends *TypedAtomicActor*): Add tokens on the *plus* input channels and subtract tokens on the *minus* input channels.

Accumulator (extends *Transformer*): Output the initial value plus the sum of all the inputs since the last time a true token was received at the *reset* port.

1. In future releases, these actors may operate also on arrays and matrices.

Average (extends *Transformer*): Output the average of the inputs since the last time a true token was received at the *reset* port. The *reset* input may be left disconnected in most domains.

Counter (extends *TypedAtomicActor*): An up-down counter of received tokens.

Differential (extends *Transformer*): Output the difference between successive inputs.

DotProduct (extends *TypedAtomicActor*): Output the dot product of two input arrays.

Expression (extends *TypedAtomicActor*): On each firing, evaluate the *expression* parameter, whose value is set by an expression that may include references to any input ports that have been added to the actor. The expression language is described in the Expressions chapter, with the addition that the expressions can refer to the values of inputs, and to the current time by the variable name "time," and to the current iteration count by the variable named "iteration." To add input ports to the actor in Vergil, right click on its icon and select "Configure Ports," and then select "Add."

Limiter (extends *Transformer*): Produce an output token on each firing with a value that is equal to the input if the input lies between *top* and *bottom*. Otherwise, if the input is greater than *top*, output *top*. If the input is less than *bottom*, output *bottom*.

LookupTable (extends *Transformer*): Output the value in the array of tokens specified by the *table* parameter at the index specified by the *input* port.

MathFunction (extends *TypedAtomicActor*): Produce an output token with a value that is a function of the input(s). The function is specified by the *function* attribute, where valid functions are *exp*, *log*, *modulo*, *sign*, *square*, and *sqrt*.

Maximum (extends *TypedAtomicActor*): Broadcast an output token on each firing on *maximumValue* with a value that is the maximum of the values on the input channels. The index of this maximum is broadcast on *channelNumber*.

Minimum (extends *TypedAtomicActor*): Broadcast an output token on each firing on *minimumValue* with a value that is the minimum of the values on the input channels. The index of this minimum is broadcast on *channelNumber*.

MultiplyDivide (extends *TypedAtomicActor*): Multiply tokens on the *multiply* input channels, and divide by tokens on the *divide* input channels.

Quantizer (extends *Transformer*): Produce an output token with the value in *levels* that is closest to the input value.

Remainder (extends *Transformer*): Produce an output token with the value that is the remainder after dividing the token on the *input* port by the *divisor*.

Scale (extends *Transformer*): Produce an output that is the product of the *input* and the *factor*.

TrigFunction (extends *Transformer*): Produce an output token with a value that is a function of the input. The function is specified by the *function* attribute, where valid functions are *acos*, *asin*, *atan*, *cos*, *sin*, and *tan*.

4.3.5 Random

The random library (see figure 4.2) consists of actors that generate random data.

Bernoulli (extends *RandomSource*): Produce a random sequence of booleans (a source of coin flips).

DiscreteRandomSource (extends *RandomSource*): Produce tokens with the given probability mass function.

Gaussian (extends *RandomSource*): Produce a random sequence with a Gaussian distribution.

Rician (extends *RandomSource*): Produce a random sequence with a Rician or Rayleigh distribution.

Uniform (extends *RandomSource*): Produce a random sequence with a uniform distribution.

4.3.6 Flow Control

The flow control actors route tokens or otherwise affect the flow of control. The output of some of these actors are controlled via a *control* or *select* port. The flow control directory of the Vergil actor library contains a subdirectory named "boolean flow control". Actors in this subdirectory are variants of actors in the "flow control" directory that have boolean *select* or *control* ports.

Aggregators

BusAssembler (extends *TypedAtomicActor*): Aggregate all input relation channels into one output "bus" relation.

BusDisassembler (extends *TypedAtomicActor*): Split the input bus relation into individual (possibly bus) output port relations.

Commutator (extends *Transformer*): Interleave the data on the input channels into a single sequence on the output.

Distributor (extends *Transformer*): Distribute the data on the input sequence into multiple sequences on the output channels.

Multiplexor (extends *Transformer*): Produce as output the token on the channel of *input* specified by the *select* input. Exactly one token is consumed from each channel of *input* in each firing.

RecordAssembler (extends *TypedAtomicActor*): Produce an output token that results from combining a token from each of the input ports (which must be added by the user). To add input ports to the actor in Vergil, right click on its icon and select "Configure Ports," and then select "Add." The name of each field in the record is the name of the port that supplies the field.

RecordDisassembler (extends *TypedAtomicActor*): Produce output tokens on the output ports (which must be added by the user) that result from separating the record on the input port. To add output ports to the actor in Vergil, right click on its icon and select "Configure Ports," and then select "Add." The name of each field extracted from the record is the name of the output port to which the value of the field is sent.

RecordUpdater (extends *TypedAtomicActor*): Produce an output token that results from the union of the record read from the *input* port and the values supplied by the other input ports. The user must create the other input ports. Input ports with the same name as a field in the original input record are used to update the corresponding field in the output token.

Select (extends *Transformer*): Produce as output the token on the channel of *input* specified by the *control* input. Tokens on channels that are not selected are not consumed.

Switch (extends *Transformer*): Produce the token read from the *input* port on the channel of *output* specified by the *control* input.

Synchronizer (extends *Transformer*): Wait until at least one token exists on each channel of *input*, then consume exactly one token from each input channel and output each token on its corresponding output channel.

VectorAssembler (extends *Transformer*): On each firing, read exactly one token from each channel of the *input* port and assemble the tokens into a double matrix with one column.

VectorDisassembler (extends *Transformer*): On each firing, read one column vector (i.e. a double matrix token with one column) from the *input* port and send out individual doubles to each channel of the *output* port.

Boolean Flow Control

BooleanMultiplexor (extends *TypedAtomicActor*): Produce as output the token from either *trueInput* or *falseInput* as specified by the *select* input. Exactly one token from each input port is consumed.

BooleanSelect (extends *TypedAtomicActor*): Produce as output the token from either *trueInput* or *falseInput* as specified by the *control* input. Tokens from the port that is not selected are not consumed.

BooleanSwitch (extends *TypedAtomicActor*): Produce the token read from the *input* port on either the *trueOutput* or the *falseOutput* port, as specified by the *control* input port.

Sequence Control

Chop (extends *SDFTransformer*): Chop an input sequence and construct from it a new output sequence. This actor can be used, for example, for zero-padding, overlapping windows, delay lines, etc.

Repeat (extends *SDFTransformer*): Repeat each input sample (a block of tokens) a specified number of times.

SampleDelay (extends *SDFTransformer*): Produce a set of initial tokens during the initialize() method, and subsequently pass input tokens to the output. Used to break dependency cycles in directed loops of SDF models.

Sequencer (extends *Transformer*): Put tokens in order according to their numbers in a sequence.

Execution Control

Stop (extends *Sink*): Stop execution of a model when it receives a *true* token on any input channel.

ThrowException (extends *Sink*): Throw an *IllegalActionException* when it receives a *true* token on any input channel.

ThrowModelError (extends *Sink*): Throw a model error when it receives a *true* token on any input channel. A model error is an exception that is passed up the containment hierarchy rather than being immediately thrown as an exception.

4.3.7 Real Time

The behavior of the real time actors is affected by the amount of elapsed real time.

RealTimePlotter (extends *Plotter*): Plot input data as a function of elapsed real time.

Sleep (extends *Transformer*): Produce as output the tokens received on input after an amount of real time specified by the *sleepTime* parameter.

VariableSleep (extends *Transformer*): Produce as output the tokens received on input after an amount of real time specified by the *sleepTime* input. NOTE: This will likely be replaced by a version of *Sleep* with a PortParameter.

WallClockTime (extends *Source*): Output the elapsed real time in seconds. This actor also appears in the "real time" directory of the Vergil actor library.

4.3.8 Logic

The logic actors perform logical operations on inputs.

Comparator (extends *TypedAtomicActor*): Produce an output token with a value that is a comparison of the input. The comparison is specified by the *comparison* attribute, where valid comparisons are >, >=, <, <=, and ==.

Equals (extends *Transformer*): Consume at most one token from each channel of *input*, and produce an output token with value *true* if these tokens are equal in value, and false otherwise.

IsPresent (extends *Transformer*): Consume at most one token from each channel of *input*, and output a boolean on the corresponding output channel (if there is one). The value of the boolean is true if the input is present and false otherwise.

LogicalNot (extends *Transformer*): Produce an output token which is the logical negation of the input token.

LogicFunction (extends *Transformer*): Produce an output token with a value that is a logical function of the tokens on the channels of *input*. The function is specified by the *function* attribute, where valid functions are *and*, *or*, *xor*, *nand*, *nor*, and *xnor*.

4.3.9 Conversions

Ptolemy II has a sophisticated type system that allows actors to be polymorphic (to operate on multiple data types). Typically, actors express type constraints between their ports and their parameters. When actors are connected, these type constraints are resolved to determine the type of each port. Conversions between types are automatic if they result in no loss of data. However, sometimes, a model builder may wish to force a particular conversion. The actors in the conversions library support this.

BooleanToAnything (extends *Converter*): Convert a Boolean input token to any data type.

BitsToInt (extends *SDFConverter*): Convert 32 successive binary inputs into a two's complement integer.

CartesianToComplex (extends *TypedAtomicActor*): Convert two tokens representing the real and imaginary of a complex number into their complex representation.

CartesianToPolar (extends *TypedAtomicActor*): Convert a Cartesian pair (a token on the *x* input and a token on the *y* input) to two tokens representing its polar form (which are output on *angle* and *magnitude*).

ComplexToCartesian (extends *TypedAtomicActor*): Convert a token representing a complex number into its Cartesian components (which are output on *real* and *imag*).

ComplexToPolar (extends *TypedAtomicActor*): Convert a token representing a complex number into two tokens representing its polar form (which are output on *angle* and *magnitude*).

DoubleToFix (extends *Converter*): Convert a double into a fix point number with a specific precision, using a specific quantization strategy.

ExpressionToToken (extends *Converter*): Read a string expression from the input port and outputs the token resulting from the evaluation.

FixToDouble (extends *Converter*): Convert a fix point into a double, by first setting the precision of the fix point to the supplied precision, using a specific quantization strategy.

FixToFix (extends *Converter*): Convert a fix point into another fix point with possibly a different precision, using a specific quantizer and overflow strategy.

IntToBits (extends *SDFConverter*): Convert an input integer into 32 successive binary outputs.

InUnitsOf (extends *Transformer*): Convert input tokens to specified units by dividing the input by the value of the *units* parameter. This actor is designed to be used with a *unit system*, which must be included in the model (note that some Ptolemy II applications do not include unit systems).

LongToDouble (extends *Converter*): Convert an input of type *long* to an output of type *double*.

PolarToCartesian (extends *TypedAtomicActor*): Converts two tokens representing a polar coordinate (a token on *angle* and a token on *magnitude*) to two tokens representing their Cartesian form (which are output on *x* and *y*).

PolarToComplex (extends *TypedAtomicActor*): Converts two tokens representing polar coordinates (a token on *angle* and a token on *magnitude*) to a token representing their complex form.

Round (extends *TypedAtomicActor*): Produce an output token with a value that is a rounded version of the input. The rounding method is specified by the *function* attribute, where valid functions are *ceil*, *floor*, *round*, and *truncate*.

StringToUnsignedByteArray (extends *Converter*): Convert an input of type *string* to an array of type *unsignedByte*.

TokenToExpression (extends *Converter*): Read a string expression from the input port and output the token resulting from the evaluation.

UnsignedByteArrayToString (extends *Converter*): Convert an input that is an array of bytes into a string.

4.3.10 Array

The array library supports manipulations of arrays, which are ordered collections of tokens of arbitrary type.

ArrayAppend (extends *Transformer*): Append arrays on the input channels to produce a single output array.

ArrayElement (extends *Transformer*): Extract an element from an array and produce it on the output.

ArrayExtract (extends *Transformer*): Extract a subarray from an array and produce it on the output.

ArrayLength (extends *Transformer*): Output the length of the input array.

ArrayMaximum (extends *Transformer*): Extract the maximum element from an array.

ArrayMinimum (extends *Transformer*): Extract the minimum element from an array.

ArrayToSequence (extends *SDFTransformer*): Extract all elements from an *input* array and produce them sequentially on the output port.

SequenceToArray (extends *SDFTransformer*): Collect a sequence of inputs into an array and produce the array on the output port.

4.3.11 Matrix

The matrix library supports matrix manipulations. Currently this library is very small; if you need matrix operations that are not in this library, then very likely they are available in the expression language (see the Expression chapter). You can access these using the *Expression* actor.

MatrixToSequence (extends *SDFTransformer*): Unbundle a matrix into a sequence of output tokens. On each firing, this actor writes the elements of the array to the output as a sequence of output tokens.

MatrixViewer (extends *Sink*): Display the contents of a matrix input.

SequenceToMatrix (extends *SDFTransformer*): Bundle a specified number of input tokens into a matrix. On each firing, this actor reads *rows* times *columns* input tokens and writes one output matrix token with the specified number of rows and columns.

4.3.12 Signal Processing

The signal processing library is divided into sublibraries.

Audio

The audio library provides actors that can read and write audio files, can capture data from an audio input such as a CD or microphone, and can play audio data through the speakers of the computers. It uses the javasound library, which is part of the Sun Microsystems' Java 2 Standard Edition (J2SE) version 1.3.0 and higher. The AudioCapture and AudioPlayer actors are unusual in that they have coupled parameter values. Changing the parameters of one results in the parameters of the other being changed. Also, as of this writing, they have the restriction that only one of each may be used in a model at a time, and that if there are two models that use them, then those two models may not be executed simultaneously.

AudioCapture (extends *Source*): Capture audio from the audio input port of the computer, or from its microphone, and produce the samples at the output.

AudioReader (extends *Source*): Read audio from a URL, and produce the samples at the output.

AudioPlayer (extends *Sink*): Play audio samples on the audio output port of the computer, or from its speakers.

AudioWriter (extends *Sink*): Write audio data to a file.

Communications

The communications library collects actors that support modeling and design of digital communication systems.

ConvolutionalCoder (extends *Transformer*): Encode an input sequence of bits using a convolutional code.

DeScrambler (extends *Transformer*): Descramble the input bit sequence using a feedback shift register.

HadamardCode (extends *Source*): Produce a Hadamard codeword by selecting a row from a Hadamard matrix.

LineCoder (extends *SDFTransformer*): Read a sequence of booleans (of length *wordLength*) and interpret them as a binary index into the *table*, from which a token is extracted and sent to the output.

LMSAdaptive (extends *FIR*): Filter the input with an adaptive filter, and update the coefficients of the filter using the input error signal according to the LMS (least mean-square) algorithm.

RaisedCosine (extends *FIR*): An FIR filter with a raised cosine frequency response. This is typically used in a communication systems as a pulse shaper or a matched filter.

Scrambler (extends *Transformer*): Scramble the input bit sequence using a feedback shift register.

ViterbiDecoder (extends *Transformer*): Decode inputs using (hard or soft) Viterbi decoding.

Filtering

DelayLine (extends *SDFTransformer*): In each firing, output the *n* most recent input tokens collected into an array, where *n* is the length of *initialValues*. In the beginning, before there are *n* most recent tokens, use the tokens from *initialValues*.

DownSample (extends *SDFTransformer*): Read *factor* inputs and produce only one of them on the output.

FIR (extends *SDFTransformer*): Produce an output token with a value that is the input filtered by an FIR filter with coefficients given by *taps*.

IIR (extends *Transformer*): Produce an output token with a value that is the input filtered by an IIR filter using a direct form II implementation.

Lattice (extends *Transformer*): Produce an output token with a value that is the input filtered by an FIR lattice filter with coefficients given by *reflectionCoefficients*.

LinearDifferenceEquationSystem (extends *Transformer*): Linear system given by an [A, b, c, d] state-space model.

LMSAdaptive (extends *FIR*): Filter the input with an adaptive filter, and update the coefficients of the filter using the input error signal according to the LMS (least mean-square) algorithm.

RecursiveLattice (extends *Transformer*): Produce an output token with a value that is the input filtered by a recursive lattice filter with coefficients given by reflectionCoefficients.

UpSample (extends *SDFTransformer*): Read one input token and produce *factor* outputs, with all but one of the outputs being a zero of the same type as the input.

VariableFIR (extends *FIR*): Filter the input sequence with an FIR filter with coefficients given on the *newTaps* input port. The *blockSize* parameter specifies the number of successive inputs that are processed for each set of taps provided on *newTaps*.

VariableLattice (extends *Lattice*): Filter the input sequence with an FIR lattice filter with coefficients given on the *newCoefficients* input port. The *blockSize* parameter specifies the number of successive inputs that are processed for each set of taps provided on *newCoefficients*.

VariableRecursiveLattice (extends *Lattice*): Filter the input sequence with a recursive lattice filter with coefficients given on the *newCoefficients* input port. The *blockSize* parameter specifies the number of successive inputs that are processed for each set of taps provided on *newCoefficients*.

Image Processing

A preliminary image processing library is provided with the 3.0 release, but it is at a sufficiently early stage of development that we do not document here. See the on-line documentation.

Spectrum

DB (extends *Transformer*): Produce a token that is the value in decibels ($k*\log_{10}(z)$) of the token received, where *k* is 10 if *inputIsPower* is true, and 20 otherwise. The output is never less than *min* (it is clipped if necessary).

FFT (extends *SDFTransformer*): A fast Fourier transform of size 2^{order}.

IFFT (extends *SDFTransformer*): An inverse fast Fourier transform of size 2^{order}.

LevinsonDurbin (extends *SDFTransformer*): Calculate the linear predictor coefficients (for both an FIR and Lattice filter) for the specified autocorrelation input.

MaximumEntropySpectrum (composite actor): A fancy spectrum estimator that uses the Levinson-Durbin algorithm to calculate linear predictor coefficients, and then uses those as a parametric model for the random process.

Periodogram (composite actor): A spectrum estimator calculates a periodogram.

PhaseUnwrap (extends *Transformer*): A simple phase unwrapper.

SmoothedSpectrum (composite actor): A spectrum estimator called the Blackman-Tukey algorithm, which estimates an autocorrelation function by averaging products of the input samples, and then calculates the FFT of that estimate.

Spectrum (composite actor): A simple spectrum estimator that calculates the FFT of the input. For a random process, this is called the periodogram spectral estimate.

Statistical

A small number of statistical analysis actors are provided.

Autocorrelation (extends *SDFTransformer*): Estimate the autocorrelation by averaging products of the input samples.

PowerEstimate (extends *Transformer*): Estimate the power of the input signal.

4.3.13 Domain Specific

Several sublibraries contain actors that are primarily useful only with corresponding directors.

Continuous Time

The continuous-time library contains a set of actors designed specifically for use in the CT domain. The continuous time directory of the Vergil actor library contains subdirectories named "event generators and "waveform generators".

ContinuousClock: Generate a piecewise-constant signal with instantaneous transitions between levels.

TriggeredContinuousClock: Generate a piecewise-constant signal with instantaneous transitions between levels, where two input ports are provided to start and stop the clock.

ContinuousSinewave: Generate a continuous-time sinusoidal signal.

CTCompositeActor: Composite actor to use when a continuous-time model is created within a continuous-time model.

Continuous Time: Dynamics

The actors in this sublibrary have continuous-time dynamics (i.e., they involve integrators, and hence must coordinate with the differential equation solver).

Integrator: Integrate the input signal over time to produce the output signal. That is, the input is the derivative of the output with respect to time. This actor can be used to close feedback loops in CT to define interesting differential equation systems.

LaplaceTransferFunction: Filter the input with the specified rational Laplace transform transfer function. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.

LinearStateSpace: Filter the input with a linear system. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.

DifferentialSystem: Filter the input with the specified system, which can nonlinear, and is specified using the expression language. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.

RateLimiter: Limit the first derivative of the input signal, and produce the result as an output sequence.

Continuous Time: To Discrete

The actors in this sublibrary produce discrete event signals, which are signals that only have values at discrete points in time.

EventSource: Output a set of events at discrete set of time points.

LevelCrossingDetector: A event detector that converts continuous signals to discrete events when the continuous signal crosses a level threshold.

PeriodicSampler: Sample the input signal with the specified rate, producing discrete output events.

TriggeredSampler: Sample the input signal at times where the trigger input has a discrete input events.

ThresholdMonitor: Output *true* if the input value is in the interval [*a*, *b*], which is centered at *thresholdCenter* and has width *thresholdWidth*. This actor controls the integration step size so that the input does not cross the threshold without producing at least one *true* output.

ZeroCrossingDetector: When the *trigger* is zero (within the specified *errorTolerance*), then output the value from the *input* port as a discrete event. This actor controls the integration step size to accurately resolve the time at which the zero crossing occurs.

Continuous Time: To Continuous

The actors in this sublibrary convert discrete event signals into continuous-time signals.

ZeroOrderHold: Convert discrete events at the input to a continuous-time signal at the output by holding the value of the discrete event until the next discrete event arrives.

FirstOrderHold: Convert discrete events at the input to a continuous-time signal at the output by projecting the value with the derivative.

4.3.14 Discrete Event

A library of actors is provided to particularly support discrete-event models. In discrete-event models, signals consist of events placed in time, where time is a double. Events are processed in chronological order.

EventButton: Output a token in response to the click of a button.

EventFilter: An actor that filters a stream of Boolean Tokens. Every true input token that it receives is reproduced on the output port. False tokens are discarded. This is usually used to properly trigger other discrete event actors (such as inhibit and select) based on boolean values.

Inhibit: Output a received input token, unless the inhibit port receives a token.

Merge: Merge input events into a single signal.

PreemptableTask: Simulate a preemptable task.

Previous: On each iteration, this actor produces the token received on the previous iteration. On the first iteration, it produces the token given by the *initialValue* parameter, if such a value has been set.

Queue: This actor implements an event queue. When a token is received on the input port, it is stored in the queue. When the trigger port receives a token, the oldest element in the queue is output. If there is no element in the queue when a token is received on the trigger port, then no output is produced.

QueueWithNextOut: This actor is like the *Queue* actor above. An additional output port, *nextOut*, has been added which allows the model to know what's next to come out. This new output produces a token whenever the queue has been empty and a new token is queued. It also produces an output whenever a token is taken from the queue and at least one token remains. Otherwise, no output token is produced at *nextOut*. The token produced is the oldest token remaining in the queue.

Sampler: On each *trigger* input, produce at the output the most recently seen input.

SamplerWithDefault: Output the most recent input token when the *trigger* port receives a token. If no token has been received on the *input* port when a token is received on the *trigger* port, then the value of the *initialValue* parameter is produced.

Server: Delay input events until they have been "served" for the specified amount of time.

SingleEvent: Produce a single event with the specified time and value.

TimedDelay: Delay input events by the specified amount.

TimeGap: Produce at the output the amount of time between input events.

Timer: Given an input time value, produce *value* on the output that amount of time in the future.

VariableDelay: Delay input events by the specified amount.

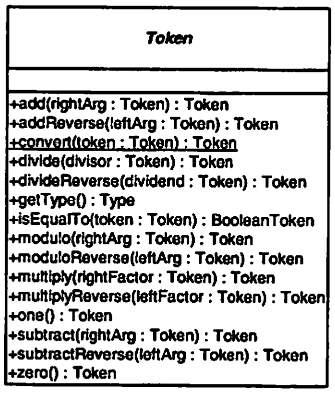
WaitingTime: Measure the amount of time that one event (arriving on *waiter*) has to wait for an event to arrive on *waitee*. There is an output event for every event that arrives on *waiter*, where the value of that output is the time spent waiting, and the time of the output is time of the arriving *waitee* event.

4.4 Data Polymorphism

A data polymorphic actor is one that can operate on any of a number of input data types. For example, AddSubtract can accept any type of input. Addition and subtraction are possible on any type of token because they are defined in the base class Token.

Figure 4.4 shows the methods defined in the base class Token. All data exchanged between actors in Ptolemy is wrapped in an instance of Token (or more precisely, in an instance of a class derived from Token). Notice that add() and subtract() are methods of this base class. This makes it easy to implement a data polymorphic adder.

It is instructive to examine the code in an actor that performs data polymorphic operations. The fire() method of the AddSubtract actor is shown in figure 4.5. In this code, we first iterate through the channels of *plus* input. The first token read (by the get() method) is assigned to sum. Subsequently, the polymorphic add() method of that token is used to add additional tokens. The second iteration, over the channels at the *minus* input port, is slightly trickier. If no tokens were read from the *plus* input, then the variable sum is initialized by calling the polymorphic zero() method of the first token read at the *minus* port. The zero() method returns whatever a zero value is for the token in question.



Token
+add(rightArg : Token) : Token
+addReverse(leftArg : Token) : Token
+convert(token : Token) : Token
+divide(divisor : Token) : Token
+divideReverse(dividend : Token) : Token
+getType() : Type
+isEqualTo(token : Token) : BooleanToken
+modulo(rightArg : Token) : Token
+moduloReverse(leftArg : Token) : Token
+multiply(rightFactor : Token) : Token
+multiplyReverse(leftFactor : Token) : Token
+one() : Token
+subtract(rightArg : Token) : Token
+subtractReverse(leftArg : Token) : Token
+zero() : Token

FIGURE 4.4. The Token class defines a polymorphic interface that includes basic arithmetic operations.

```
public void fire() throws IllegalActionException {
    Token sum = null;
    for (int i = 0; i < plus.getWidth(); i++) {
        if (plus.hasToken(i)) {
            if (sum == null) {
                sum = plus.get(i);
            } else {
                sum = sum.add(plus.get(i));
            }
        }
    }
    for (int i = 0; i < minus.getWidth(); i++) {
        if (minus.hasToken(i)) {
            Token in = minus.get(i);
            if (sum == null) {
                sum = in.zero();
            }
            sum = sum.subtract(in);
        }
    }
    if (sum != null) {
        output.send(0, sum);
    }
}
```

FIGURE 4.5. The fire() method of the AddSubtract shows the use of polymorphic add() and subtract() methods in the Token class (see figure 4.4).

Not all classes derived from Token override all its methods. For example, StringToken overrides add() but not subtract(). Adding strings means simply concatenating them, but it is hard to assign a reasonable meaning to subtraction. Thus, if AddSubtract is used on strings, then the *minus* port must not ever receive tokens. It may be simply left disconnected, in which case minus.getWidth() returns zero. If the subtract() method of a StringToken is called, then a runtime exception will be thrown.

4.5 Domain Polymorphism

Most actors access their ports as shown in figure 4.5, using the hasToken(), get(), and send() methods. Those methods are polymorphic, in that their exact behavior depends on the domain. For example, get() in the CSP domain causes a rendezvous to occur, which means that the calling thread is suspended until another thread sends data to the same port (using, for example, the send() method on one of its ports). Correspondingly, a call to send() causes the calling thread to suspend until some other thread calls a corresponding get(). In the PN domain, by contrast, send() returns immediately (if there is room in the channel buffers), and only get() causes the calling thread to suspend.

Each domain has slightly different behavior associated with hasToken(), get(), send() and other methods of ports. The actor, however, does not really care. The fire() method shown in figure 4.5 will work for any reasonable implementation of these methods. Thus, the AddSubtract actor is domain polymorphic.

Domains also have different behavior with respect to when the fire() method is invoked. In process-oriented domains, such as CSP and PN, a thread is created for each actor, and an infinite loop is created to repeatedly invoke the fire() method. Moreover, in these domains, hasToken() always returns *true*, since you can call get() on a port and it will not return until there is data available. In the DE domain, the fire() method is invoked only when there are new inputs that happen to be the oldest ones in the systems, and hasToken() returns *true* only if there is new data on the input port. The design of actors for multiple domains is covered in the Designing Actors chapter.

5

Designing Actors

Authors: *Christopher Hylands*
Edward A. Lee
Jie Liu
Xiaojun Liu
Steve Neuendorffer
Yuhong Xiong
Winthrop Williams

5.1 Overview

Ptolemy is about component-based design. The domains define the semantics of the interaction between components. This chapter explains the common, domain-independent principles in the design of components that are actors. Actors are components with input and output that at least conceptually operate concurrently with other actors.

As explained in the previous chapter, some actors are designed to be domain polymorphic, meaning that they can operate in multiple domains. Others are domain specific. Refer to the domain chapters in part 3 for domain-specific information relevant to the design of actors. This chapter explains how to design actors so that they are maximally domain polymorphic. As also explained in the previous chapter, many actors are also data polymorphic. This means that they can operate on a wide variety of token types. Domain and data polymorphism help to maximize the reusability of actors and to minimize the amount of 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. Figure 4.1 shows a UML static structure diagram of the Ptolemy actor library. Three base classes, Source, Sink, and Transformer, exist to ensure consistent naming of ports and to avoid duplicating code associated with those ports. Since most actors in the library extend these base classes, users of the library can guess that an input port is named "input" and an output port is named "output," and they will probably be right. Using base classes avoids input ports named "in" or "inputSignal" or something else. This sort of consistency helps to promote re-use of actors because it makes them easier to use. Thus, we recommend using a reasonably deep class hierarchy to promote consistency.

5.2 Anatomy of an Actor

Each actor consists of a source code file written in Java. Sources are compiled to Java byte code as directed by the makefile in their directory. Thus, when creating a new actor, it is necessary to add its name to the local makefile. Vergil, described fully in its own chapter, is the graphical design tool commonly used to compose actors and other components into a complete program, a "Ptolemy model." To facilitate using an actor in Vergil, it must appear in one of the actor libraries. This permits it to be dragged from the library pallet onto the design canvas. The libraries are XML files. Many of the actor libraries are in the $PTII/ptolemy/actor/lib directory.

The basic structure of an actor is shown in figure 5.1. In that figure, keywords in bold are features of Ptolemy II that are briefly described here and described in more detail in the chapters of part 2. Italic text would be substituted with something pertinent in an actual actor definition.

We will go over this structure in detail in this chapter. The source code for existing Ptolemy II actors, located mostly in $PTII/ptolemy/actor/lib, should also be viewed as a key reference.

5.2.1 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 5.1 shows a single port *portName* that is an instance of TypedIOPort, declared in the line

```
public TypedIOPort portName;
```

Most ports in actors are instances of TypedIOPort, unless they require domain-specific services, in which case they may be instances of a domain-specific subclass, such as DEIOPort. The port is actually created in the constructor by the line

```
portName = new TypedIOPort(this, "portName", true, false);
```

The first argument to the constructor is the container of the port, this actor. The second is the name of the port, which can be any string, but by convention, is the same as the name of the public member. The third argument specifies whether the port is an input (it is in this example), and the fourth argument specifies whether it is an output (it is not in this example). There is no difficulty with having a port that is both an input and an output, but it is rarely useful to have one that is neither.

Multiports and Single Ports. A port can be a single port or a multiport. By default, it is a single port. It can be declared to be a multiport with a statement like

```
portName.setMultiport(true);
```

All ports have a *width*, which corresponds to the number of channels the port represents. 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.

```
/** Javadoc comment for the actor class. */
public class ActorClassName extends BaseClass implements MarkerInterface {

    /** Javadoc comment for constructor. */
    public ActorClassName(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        // Create and configure ports, e.g. ...
        portName = new TypedIOPort(this, "portName", true, false);
        // Create and configure parameters, e.g. ...
        parameterName = new Parameter(this, "parameterName");
        parameterName.setTypeEquals(BaseType.DOUBLE);
    }

    ///////////////////////////////////////////////////////////////////
    ////                    ports and parameters                   ////

    /** Javadoc comment for port. */
     public TypedIOPort portName;

    /** Javadoc comment for parameter. */
    public Parameter parameterName;

    ///////////////////////////////////////////////////////////////////
    ////                      public methods                       ////

    /** Javadoc comment for fire method. */
    public void fire() {
        super.fire();
        ... read inputs and produce outputs ...
    }

    /** Javadoc comment for initialize method. */
    public void initialize() {
        super.initialize();
        ... initialize local variables ...
    }

    /** Javadoc comment for prefire method. */
    public boolean prefire() {
        ... determine whether firing should proceed and return false if not ...
        return super.prefire();
    }

    /** Javadoc comment for preinitialize method. */
    public void preinitialize() {
        super.preinitialize();
        ... set port types and/or scheduling information ...
    }

    /** Javadoc comment for postfire method. */
    public boolean postfire() {
        ... update persistent state ...
        ... determine whether firing should continue to next iteration and return false if not ...
        return super.postfire();
    }

    /** Javadoc comment for wrapup method. */
    public void wrapup() {
        super.wrapup();
        ... display final results ...
    }
}
```

FIGURE 5.1. Anatomy of an actor.

Reading and Writing. Data (encapsulated in a *token*) can be sent to a particular channel of an output port with the syntax

```
portName.send(channelNumber, token);
```

where *channelNumber* is the number of the channel (beginning with 0 for the first channel). The width of the port, 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 complaining.

A token can be sent to all output channels of a port (or none if there are none) 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 contained value (if you know its type) by

```
double variableName = ((DoubleToken)portName.get(0)).doubleValue();
```

You can query an input port to see whether such 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 most current domains, the answer is always true. Note that the get(), hasRoom() and hasToken() methods throw IllegalActionException if the channel is out of range, but send() just silently returns.

Type Constraints. Ptolemy II includes a sophisticated type system, described fully in the Type System chapter. This type system supports specification of type constraints in the form of inequalities between types. These inequalities can be easily understood as representing the possibility of lossless conversion. Type a is less than type b if an instance of a can be losslessly converted to an instance of b. For example, IntToken is less than DoubleToken, which is less than ComplexToken. However, LongToken is not less than DoubleToken, and DoubleToken is not less than LongToken, so these two types are said to be *incomparable*.

Suppose that you wish 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);
```

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 the other order,

```
parameterName.setTypeSameAs(portName);
```

which obviously means the same thing, or

```
parameterName.setTypeAtLeast(portName);
```

which is not quite the same.

Another common type constraint is an *absolute type constraint*, which fixes the type of the port (i.e. making it monomorphic rather than polymorphic),

```
portName.setTypeEquals(BaseType.DOUBLE);
```

The above line declares that the port can only handle doubles. 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, type constraints are automatically created that declares its type to be less than 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 be unconstrained. Conversely, if there are output ports with no constraints, but no input ports lacking constraints, then those output ports will be unconstrained. Of course, you can declare a port to be unconstrained by saying

```
portName.setTypeAtMost(BaseType.GENERAL);
```

For full details of the type system, see the Type System chapter in volume 2.

Examples. To be concrete, consider first the code segment shown in figure 5.2, from the Transformer class in the ptolemy.actor.lib package. This actor is a base class for actors with one input and one output. The code shows two ports, one that is an input and one that is an output. By convention, the Javadoc[1] comments indicate type constraints on the ports, if any. If the ports are multiports, then the Javadoc comment will indicate that. Otherwise, they are assumed to be single ports. Derived classes may change this, making the ports into multiports, in which case they should document this fact in the class comment. Derived classes may also set the type constraints on the ports.

An extension of Transformer is shown in figure 5.3, the SimplerScale actor, which is a simplified version of the Scale actor which is defined in $PTII/ptolemy/actor/lib/Scale.java. 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.

Notice in figure 5.3 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.

```
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 already has 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 5.2. Code segment showing the port definitions in the Transformer class.

1. Javadoc is a program that generates HTML documentation from Java files based on comments enclosed in "/** ... */".

```
import ptolemy.actor.lib.Transformer;
import ptolemy.data.IntToken;
import ptolemy.data.expr.Parameter;
import ptolemy.data.Token;
import ptolemy.kernel.util.*;
import ptolemy.kernel.CompositeEntity;

public class SimplerScale extends Transformer {
   ...
   public SimplerScale(CompositeEntity container, String name)
           throws NameDuplicationException, IllegalActionException  {
      super(container, name);
      factor = new Parameter(this, "factor", new IntToken(1));

      // set the type constraints.
      output.setTypeAtLeast(input);
      output.setTypeAtLeast(factor);
   }

    ///////////////////////////////////////////////////////////////////
    ////                      ports and parameters                 ////

    /** The factor.
     *  This parameter can contain any token that supports multiplication.
     *  The default value of this parameter is the IntToken 1.
     */
    public Parameter factor;

    ///////////////////////////////////////////////////////////////////
    ////                         public methods                    ////

    /** Clone the actor into the specified workspace. This calls the
     *  base class and then sets the type constraints.
     *  @param workspace The workspace for the new object.
     *  @return A new actor.
     *  @exception CloneNotSupportedException If a derived class has
     *   an attribute that cannot be cloned.
     */
    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;
    }

    /** Compute the product of the input and the <i>factor</i>.
     *  If there is no input, then produce no output.
     *  @exception IllegalActionException If there is no director.
     */
    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 5.3. Code segment from the SimplerScale actor, showing the handling of ports and parameters.

5.2.2 Parameters

Like ports, parameters are public members of actors by convention. Figure 5.3 shows a parameter *factor* that is an instance of Parameter, declared in the line

```
public Parameter factor;
```

and created in the line

```
factor = new Parameter(this, "factor", new IntToken(1));
```

The third argument to the constructor, which is optional, is a default value for the parameter. In this example, the *factor* parameter defaults to the integer one. Alternatively, the default value of the parameter can be set via an expression, as in

```
factor = new Parameter(this, "factor");
factor.setExpression("2*PI");
```

As with ports, you can specify type constraints on parameters. The most common type constraint is to fix the type, using

```
parameterName.setTypeEquals(BaseType.DOUBLE);
```

In fact, exactly the same relative or absolute type constraints that one can specify for ports can be specified for parameters as well. But in addition, arbitrary constraints on parameter values are possible, not just type constraints.

An actor is notified when a parameter value changes by having its attributeChanged() method called. Consider the example shown in figure 5.4, 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.

The attributeChanged() method is passed the parameter that changed. (Typically it is being changed by the user via the Configure dialog.) If this is *meanTime*, then this method checks to make sure that the specified value is positive, and if not, it throws an exception. This exception is presented to the user in a new dialog box. It shows up when the user attempts to commit a non-positive value. The new dialog requests that the user choose a new value or cancel the change.

A change in a parameter value sometimes has broader repercussions than just the local actor. It may, for example, affect the schedule of execution of actors. An actor can call the invalidateSchedule() method of the director, which informs the director that any statically computed schedule (if there is one) is no longer valid. This would be used, for example, if the parameter affects the number of tokens produced or consumed when an actor fires.

When the type of a parameter changes, the attributeTypeChanged() method in the actor containing that parameter will be called. The default implementation of this method in TypedAtomicActor is to invalidate type resolution. So parameter type change will cause type resolution to be performed in the model. This default implementation is suitable for most actors. In fact, most of the actors in the actor library do not override this method. However, if for some reason, an actor does not wish to redo type

```
public class PoissonClock extends TimedSource {

   public Parameter meanTime;
   public Parameter values;

   public PoissonClock(CompositeEntity container, String name)
           throws NameDuplicationException, IllegalActionException  {
      super(container, name);
      meanTime = new Parameter(this, "meanTime", new DoubleToken(1.0));
      meanTime.setTypeEquals(BaseType.DOUBLE);
      ...
   }

   /** If the argument is the meanTime parameter, check that it is
    *  positive.
    *  @exception IllegalActionException If the meanTime value is
    *   not 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.  meanTime given: "
                       + mean);
           }
       } else if (attribute == values) {
           ArrayToken val = (ArrayToken)(values.getToken());
           _length = val.length();
       } else {
           super.attributeChanged(attribute);
       }
   }
   ...
}
```

FIGURE 5.4. Code segment from the PoissonClock actor, showing the attributeChanged() method.

resolution upon parameter type change, the attributeTypeChanged() method can be overridden. But this is rarely necessary.

5.2.3 Constructors

We have seen already that the major task of the constructor is to create and configure ports and parameters. In addition, you may have noticed that it calls

```
super(container, name);
```

and that it declares that it throws NameDuplicationException and IllegalActionException. The latter is the most widely used exception, and many methods in actors declare that they can throw it. The former is thrown if the specified container already contains an actor with the specified name. For more details about exceptions, see the Kernel chapter.

5.2.4 Cloning

All actors are cloneable. A clone of an actor needs to be a new instance of the same class, with the same parameter values, but without any connections to other actors.

Consider the clone() method in figure 5.5, taken from the SimplerScale actor. This method begins with:

```
SimplerScale newObject = (SimplerScale)super.clone(workspace);
```

The convention in Ptolemy II is that each clone method begins the same way, so that cloning works its way up the inheritance tree until it ultimately uses the clone() method of the Java Object class. That method performs what is called a "shallow copy," which is not sufficient for our purposes. In particular, members of the class that are references to other objects, including public members such as ports and parameters, are copied by copying the references. The NamedObj and TypedAtomicActor base classes implement a "deep copy" so that all the contained objects are cloned, and public members reference the proper cloned objects[2].

Although the base classes neatly handle most aspects of the clone operation, there are subtleties involved with cloning type constraints. Absolute type constraints on ports and parameters are carried automatically into the clone, so clone() methods should never call setTypeEquals(). However, relative type constraints are not cloned automatically because of the difficulty of ensuring that the other object being referred to in a relative constraint is the intended one. Thus, in figure 5.5, the clone() method

```
public class SimplerScale extends Transformer {
   ...
   public SimplerScale(CompositeEntity container, String name)
           throws NameDuplicationException, IllegalActionException  {
      super(container, name);
      output.setTypeAtLeast(input);
      output.setTypeAtLeast(factor);
   }

   ///////////////////////////////////////////////////////////////////
   ////                   ports and parameters                    ////

   /** The factor.  The default value of this parameter is the integer 1. */
   public Parameter factor;

   ///////////////////////////////////////////////////////////////////
   ////                       public methods                      ////

   /** Clone the actor into the specified workspace. This calls the
    *  base class and then sets the type constraints.
    *  @param workspace The workspace for the new object.
    *  @return A new actor.
    *  @exception CloneNotSupportedException If a derived class has
    *   has an attribute that cannot be cloned.
    */
   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;
   }
   ...
}
```

FIGURE 5.5. Code segment from the SimplerScale actor, showing the clone() method.

2. Be aware that the implementation of the deep copy relies on a strict naming convention. Public members that reference ports and parameters must have the same name as the object that they are referencing in order to be properly cloned.

repeats the relative type constraints that were specified in the constructor:

```
newObject.output.setTypeAtLeast(newObject.input);
newObject.output.setTypeAtLeast(newObject.factor);
```

Note that at no time during cloning is any constructor invoked, so it is necessary to repeat in the clone() method any initialization in the constructor. For example, the clone() method in the Expression actor sets the values of a few private variables:

```
newObject._iterationCount = 1;
newObject._time = (Variable)newObject.getAttribute("time");
newObject._iteration =
   (Variable)newObject.getAttribute("iteration");
```

5.3 Action Methods

Figure 5.1 shows a set of public methods called the *action methods* because they specify the action performed by the actor. By convention, these are given in alphabetical order in Ptolemy II Java files, but we will discuss them here in the order that they are invoked. The first to be invoked is the preinitialize() method, which is invoked exactly once before any other action method is invoked. The preinitialize() method is often used to set type constraints. After the preinitialize() method is called, type resolution happens and all the type constraints are resolved. The initialize() method is invoked next, and is typically used to initialize state variables in the actor, which generally depends on type resolution.

After the initialize() method, the actor experiences some number of *iterations*, where an iteration is defined to be exactly one invocation of prefire(), some number of invocations of fire(), and at most one invocation of postfire().

5.3.1 Initialization

The initialize() method of the Average actor is shown in figure 5.6. This data- and domain-polymorphic actor computes the average of tokens that have arrived. To do so, it keeps a running sum in a private variable _sum, and a running count of the number of tokens it has seen in a private variable

```
public class Average extends Transformer {
   ...
   public void initialize() throws IllegalActionException {
      super.initialize();
      _count = 0;
      _sum = null;
   }
   ...

   ///////////////////////////////////////////////////////////////////
   ////                      private members                      ////

   private Token _sum;
   private int _count = 0;
}
```

FIGURE 5.6. Code segment from the Average actor, showing the initialize() method.

_count. Both of these variables are initialized in the initialize() method. Notice that the actor also calls super.initialize(), allowing the base class to perform any initialization it expects to perform. This is essential because one of the base classes initializes the ports. An actor will almost certainly fail to run properly if super.initialize() is not called.

Note that the initialization of the Average actor does not affect, or depend on, type resolution. This means that the code to initialize this actor can be placed either in the preinitialize() method, or in the initialize() method. However, in some cases an actor may require part of its initialization to happen before type resolution, in the preinitialize() method, or part after type resolution, in the initialize() method. For example, an actor may need to dynamically create type constraints before each execution[3]. Such an actor must create its type constraints in preinitialize(). On the other hand, an actor may wish to produce (send or broadcast) an initial output token once at the beginning of an execution of a model. This production can only happen during initialize(), because data transport through ports depends on type resolution.

5.3.2 Prefire

The prefire() method is the only method that is invoked exactly once per iteration[4]. It returns a boolean that indicates to the director whether the actor wishes for firing to proceed. The fire() method of an actor should never be called until after its prefire() method has returned true. The most common use of this method is to test a condition to see whether the actor is ready to fire.

Consider for example an actor that reads from *trueInput* if a private boolean variable _state is *true*, and otherwise reads from *falseInput*. The prefire() method might look like this:

```
public boolean prefire() throws IllegalActionException {
   if (_state) {
      return trueInput.hasToken(0);
   } else {
      return falseInput.hasToken(0);
   }
}
```

It is good practice to check the superclass in case it has some reason to decline to be fired. The above example becomes:

```
public boolean prefire() throws IllegalActionException {
   if (_state) {
      return trueInput.hasToken(0) && super.prefire();
   } else {
      return falseInput.hasToken(0) && super.prefire();
   }
}
```

3. The need for this is relatively rare, but important. Examples include higher-order functions, which are actors that replace themselves with other subsystems, and certain actors whose ports are not created at the time they are constructed, but rather are added later. In most cases, the type constraints of an actor do not change and are simply specified in the constructor.
4. Some domains invoke the fire() method only once per iteration, but others will invoke it multiple times (searching for global convergence to a solution, for example).

The prefire() method can also be used to perform an operation that will happen exactly once per iteration. Consider the prefire method of the Bernoulli actor in figure 5.7:

```
public boolean prefire() throws IllegalActionException {
   if (_random.nextDouble() <
        ((DoubleToken)(trueProbability.getToken())).doubleValue()) {
     _current = true;
   } else {
     _current = false;
   }
   return super.prefire();
}
```

This method selects a new boolean value that will correspond to the token sent during each firing in that iteration.

5.3.3 Fire

The fire() method is the main point of execution and is generally responsible for reading inputs and producing outputs. It may also read the current parameter values, and the output may depend on them. Things to remember when writing fire() methods are:

- To get data polymorphism, use the methods of the Token class for arithmetic whenever possible (see the Data Package chapter). Consider for example the Average actor, shown in figure 5.8. Notice the use of the add() and divide() methods of the Token class to achieve data polymorphism.

```
public class Bernoulli extends RandomSource {

   public Bernoulli(CompositeEntity container, String name)
           throws NameDuplicationException, IllegalActionException  {
      super(container, name);

      output.setTypeEquals(BaseType.BOOLEAN);

      trueProbability = new Parameter(this, "trueProbability", new DoubleToken(0.5));
      trueProbability.setTypeEquals(BaseType.DOUBLE);
   }

   public Parameter trueProbability;

   public void fire() {
      super.fire();
      output.send(0, new BooleanToken(_current));
   }

   public boolean prefire() throws IllegalActionException {
      if (_random.nextDouble() < ((DoubleToken)(trueProbability.getToken())).doubleValue()) {
         _current = true;
      } else {
         _current = false;
      }
      return super.prefire();
   }

   private boolean _current;
}
```

FIGURE 5.7. Code for the Bernoulli actor, which is not data polymorphic.

```
public class Average extends Transformer {

   ... constructor ...

    ///////////////////////////////////////////////////////////////////
    ////                    ports and parameters                   ////

    public TypedIOPort reset;

    ///////////////////////////////////////////////////////////////////
    ////                       public methods                      ////

   ... clone method ...

    public void fire() throws IllegalActionException {
        _latestSum = _sum;
        _latestCount = _count + 1;
        // Check whether to reset.
        for (int i = 0; i < reset.getWidth(); i++) {
            if (reset.hasToken(i)) {
                BooleanToken r = (BooleanToken)reset.get(0);
                if b(r.booleanValue()) {
                    // Being reset at this firing.
                    _latestSum = null;
                    _latestCount = 1;
                }
            }
        }
        if (input.hasToken(0)) {
            Token in = input.get(0);
            if (_latestSum == null) {
                _latestSum = in;
            } else {
                _latestSum = _latestSum.add(in);
            }
            Token out = _latestSum.divide(new IntToken(_latestCount));
            output.send(0, out);
        }
    }

    public void initialize() throws IllegalActionException {
        super.initialize();
        _count = 0;
        _sum = null;
    }

    public boolean postfire() throws IllegalActionException {
        _sum = _latestSum;
        _count = _latestCount;
        return super.postfire();
    }

    ///////////////////////////////////////////////////////////////////
    ////                       private members                     ////

    private Token _sum;
    private Token _latestSum;
    private int _count = 0;
    private int _latestCount;
}
```

FIGURE 5.8. Code segment from the Average actor, showing the action methods.

- When data polymorphism is not practical or not desired, then it is usually easiest to use the setTypeEquals() to define the type of input ports. The type system will assure that you can safely cast the tokens that you read to the type of the port. Consider again the Average actor shown in figure 5.8. This actor declares the type of its *reset* input port to be BaseType.BOOLEAN. In the fire() method, the input token is read and cast to a BooleanToken. The type system ensures that no cast error will occur. The same can be done with a parameter, as with the Bernoulli actor shown in figure 5.7.
- A domain-polymorphic actor cannot assume that there is data at all the input ports. Most domain-polymorphic actors will read at most one input token from each port, and if there are sufficient inputs, produce exactly one token on each output port.
- Some domains invoke the fire() method multiple times, working towards a converged solution. Thus, each invocation of fire() can be thought of as doing a tentative computation with tentative inputs and producing tentative outputs. Thus, the fire() method should not update persistent state. Instead, that should be done in the postfire() method, as discussed in the next section.

5.3.4 Postfire

The postfire() method has two tasks:

- updating persistent state, and
- determining whether the execution of an actor is complete.

Consider the fire() and postfire() methods of the Average actor in figure 5.8. Notice that the persistent state variables `_sum` and `_count` are not updated in fire(). Instead, they are shadowed by `_latestSum` and `_latestCount`, and updated in postfire().

The return value of postfire() is a boolean that indicates to the director whether execution of the actor is complete. By convention, the director should avoid iterating further an actor after its postfire() method returns false. In other words, the director won't call prefire(), fire(), or postfire() again during this execution of the model.

Consider the two examples shown in figure 5.9. These are base classes for source actors. SequenceSource is a base class for actors that output sequences. Its key feature is a parameter *firingCountLimit*, which specifies a limit on the number of iterations of the actor. When this limit is reached, the postfire() method returns false. Thus, this parameter can be used to define sources of finite sequences.

TimedSource is similar, except that instead of specifying a limit on the number of iterations, it specifies a limit on the current model time. When that limit is reached, the postfire() method returns false.

5.3.5 Wrapup

The wrapup() method is used typically for displaying final results. It is invoked exactly once at the end of an execution, including when an exception occurs that stops execution (as opposed to an exception occurring in, say, attributeChanged(), which does not stop execution). However, when an actor is removed from a model during execution, the wrapup() method is not called.

An actor may lock a resource (which it intends to release in wrapup() for example), or its designer may have another reason to ensure that wrapup() always is called, even when the actor is removed from an executing model. This can be achieved by overriding the setContainer() method. In this case, the actor would have a setContainer() method which might look like this:

```
public class SequenceSource extends Source implements SequenceActor {

    public SequenceSource(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        firingCountLimit = new Parameter(this, "firingCountLimit", new IntToken(0));
        firingCountLimit.setTypeEquals(BaseType.INT);
    }

    public Parameter firingCountLimit;

    ...
    public void attributeChanged(Attribute attribute)
            throws IllegalActionException {
        if (attribute == firingCountLimit) {
            _firingCountLimit =
                ((IntToken)firingCountLimit.getToken()).intValue();
        }
    }

    public boolean postfire() throws IllegalActionException {
        if (_firingCountLimit != 0) {
            _iterationCount++;
            if (_iterationCount == _firingCountLimit) {
                return false;
            }
        }
        return true;
    }

    protected int _firingCountLimit;
    protected int _iterationCount = 0;
}


public class TimedSource extends Source implements TimedActor {

    public TimedSource(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        stopTime = new Parameter(this, "stopTime", new DoubleToken(0.0));
        stopTime.setTypeEquals(BaseType.DOUBLE);
        ...
    }

    public Parameter stopTime;

    ...

    public boolean postfire() throws IllegalActionException {
        double time = ((DoubleToken)stopTime.getToken()).doubleValue();
        if (time > 0.0 && getDirector().getCurrentTime() >= time) {
            return false;
        }
        return true;
    }
}
```

FIGURE 5.9. Code segments from the SequenceSource and TimedSource base classes.

```
public void setContainer(CompositeEntity container)
        throws IllegalActionException, NameDuplicationException {
    if (container != getContainer()) {
      wrapup();
    }
  super.setContainer(container);
}
```

When overriding the setContainer() method in this way, it is best to make wrapup() idempotent (implying that it can be invoked many times without causing harm), because future implementations of the director might automatically unlock resources of removed actors, or call wrapup() on removed actors.

5.4 Coupled Port and Parameter

Often, in the design of an actor, it is hard to decide whether a quantity should be given by a port or by a parameter. Fortunately, you can design an actor to offer both options. An example of such an actor is shown in figure 5.10. This actor starts with an initial value, given by the *init* parameter, and increments it each time by the value of *step*. The value of *step* is given by either a parameter named *step* or a port named *step*. To use the parameter exclusively, the model builder simply leaves the port unconnected. If the port is connected, then the parameter provides the default value, used before anything arrives on the port. But after something arrives on the port, that is used.

When the model containing a Ramp actor is saved, only the parameter value is stored. No data that arrives on the port is stored. Thus, the default value given by the parameter is persistent, while the values that arrive on the port are transient.

To set up this arrangement, the Ramp actor creates an instance of the class PortParameter in its constructor, as shown in figure 5.10. This is a parameter that, when created, creates a coupled port. There is no need to explicitly create the port. The Ramp actor creates an instance of Parameter to specify the *init* value, since it makes less sense for the value of *init* to be provided via a port.

Referring to figure 5.10, the constructor, after creating *init* and *step*, sets up type constraints. These specify that the type of the output is at least as general as the types of *init* and *step*. The PortParameter class takes care of an additional constraint, which is that the type of the *step* parameter must match the type of the *step* port. The clone() method duplicates the type constraints that are given explicitly.

In the attributeChanged() method, the actor resets its state if *init* is the parameter that changed. This will work with an instance of Parameter, but it is not recommended for an instance of PortParameter. The reason is that each time you call getToken() on an instance of PortParameter, the method checks to see whether there is an input on the port, and consumes it if there is. Actors are expected to consume inputs in their action methods, fire() and postfire(), not in attributeChanged(). Some domains, like SDF, will be confused by the consumption of the token too early.

In the Ramp actor in figure 5.10, the fire() method simply outputs the current state, whereas the postfire() method calls getToken() on *step* and adds its value to the state. This follows the Ptolemy II convention that the state of the actor is not modified in fire(), but only in postfire(). Thus, this actor can be used in domains with fixed-point semantics, such as SR.

When using a PortParameter in an actor, care must be exercised to call update() exactly once per firing prior to calling getToken(). Each time update() is called, a new token will be consumed from the associated port (if the port is connected and has a token). If this is called multiple times in an iteration, it may result in consuming tokens that were intended for subsequent iterations. Thus, for example, update() should not be called in fire() and then again in postfire(). Moreover, in some domains (such as DE), it is essential that if a token is provided on a port, that it is consumed. In DE, the actor will be repeatedly fired until the token is consumed. Thus, it is an error to not call update() once per iteration.

It is important that the actor call getToken() exactly once in either the fire() method or in the postfire() method. In particular, it should not call it in both, because this could result in consumption of two tokens from the input port, inappropriately. Moreover, it should always call it, even if it has no use for the value. Otherwise, in the DE domain, the actor will be repeatedly fired if an input event is provided

```
public class Ramp extends SequenceSource {

    public Ramp(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        init = new Parameter(this, "init", new IntToken(0));
        step = new PortParameter(this, "step", new IntToken(1));

      // set the type constraints.
      output.setTypeAtLeast(init);
      output.setTypeAtLeast(step);
   }

   public Parameter init;
   public PortParameter step;

   public void attributeChanged(Attribute attribute) throws IllegalActionException {
      if (attribute == init) {
         _stateToken = init.getToken();
      } else {
         super.attributeChanged(attribute);
      }
   }

   public Object clone(Workspace workspace) throws CloneNotSupportedException {
      Ramp newObject = (Ramp)super.clone(workspace);
      newObject.output.setTypeAtLeast(newObject.init);
      newObject.output.setTypeAtLeast(newObject.step);
      ...
      return newObject;
   }

   public void fire() throws IllegalActionException {
      super.fire();
      output.send(0, _stateToken);
   }

   public void initialize() throws IllegalActionException {
      super.initialize();
      _stateToken = init.getToken();
   }

   ...

   public boolean postfire() throws IllegalActionException {
      step.update();
      _stateToken = _stateToken.add(step.getToken());
      return super.postfire();
   }

   private Token _stateToken = null;
```

FIGURE 5.10. Code segments from the Ramp actor.

on the port but not consumed. Time cannot advance until that event is processed.

The way that the PortParameter class works is as follows. On each call to getToken(), the *step* instance of PortParameter first checks to see whether an input has arrived at the associated *step* port since the last setExpression() or setToken(), and if so, returns a token read from that port. Also, any call to get() on the associated port will result in the value of this parameter being updated, although normally an actor writer will not call get() on the port.

5.5 Iterate Method

Some domains (such as SDF) will always invoke prefire(), fire(), and postfire() in sequence, one after another, so there is no benefit from having their functionality separated into three methods. Moreover, in SDF this sequence of method invocations may be repeated a large number of times. An actor designer can improve execution efficiency by providing an iterate() method. Figure 5.11 shows the iterate() method of the Ramp actor. Its behavior is equivalent to invoking prefire(), and that returns true, then invoking fire() and postfire() in sequence. Moreover the iterate() method takes an integer argument, which specifies how many times this sequence of operations should be repeated. The return values are NOT_READY, STOP_ITERATING, or COMPLETED, which are constants defined in the Execut-

```
public class Ramp extends SequenceSource {

public Ramp(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
       super(container, name);
      ...
      _resultArray = new Token[1];
   }

   ...
   public Object clone(Workspace workspace) throws CloneNotSupportedException {
      ...
      _resultArray = new Token[1];
      return newObject;
   }

   public int iterate(int count) throws IllegalActionException {
      // Check whether we need to reallocate the output token array.
      if (count > _resultArray.length) {
         _resultArray = new Token[count];
      }
      for (int i = 0; i < count; i++) {
         _resultArray[i] = _stateToken;
         step.update();
         _stateToken = _stateToken.add(step.getToken());
      }
      output.send(0, _resultArray, count);
      if (_firingCountLimit != 0) {
         _iterationCount += count;
         if (_iterationCount >= _firingCountLimit) {
            return STOP_ITERATING;
         }
      }
      return COMPLETED;
   }

   ...

   private Token[] _resultArray;
}
```

FIGURE 5.11. More code segments from the Ramp actor of figure 5.10, showing the iterate() method.

able interface of the actor package. Returning NOT_READY is appropriate when prefire() would have returned false. Returning STOP_ITERATING is appropriate when postfire() would have returned false. Otherwise, the proper return value is COMPLETED.

5.6 Time

An actor whose behavior depends on current model time should implement the TimedActor interface. This is a marker interface (with no methods). Implementing this interface alerts the director that the actor depends on time. Domains that have no meaningful notion of time can reject such actors.

An actor can access current model time using:

```
double currentTime = getDirector().getCurrentTime();
```

Notice that although the director has a public method setCurrentTime(), an actor should never use it. Typically, only another enclosing director will call this method.

An actor can request an invocation at a future time using the fireAt() or fireAtCurrentTime() method of the director. These method returns immediately (for a correctly implemented director). The fireAt() method takes two arguments, an actor and a time. The fireAtCurrentTime() method takes only one argument, an actor. The director is responsible for performing one iteration of the specified actor at the specified time. These methods can be used to get a source actor started, and to keep it operating. In the actor's initialize() method, it can call fireAt() with a zero time. Then in each invocation of postfire(), it calls fireAt() again. Notice that the call should be in postfire() not in fire() because a request for a future firing is persistent state.

Note that while fireAt() can safely be called by any of the actors action methods, code which executes asynchronously from the director should avoid calling fireAt(). Examples of such code include the private thread within the DatagramReader actor and the serialEvent() callback method of the SerialComm actor. Because these process hardware events, which can occur at any time, they instead use the fireAtCurrentTime() method. When fireAt() was used (before fireAtCurrentTime() was written) exceptions were occasionally thrown as model time advanced just as a firing was being requested at the previous (formerly current) model time.

5.7 Icons

An actor designer can specify a custom icon when defining the actor. The Ramp actor, for instance, specifies the icon shown in figure 5.12 using the code shown in the constructor in figure 5.13. It uses a

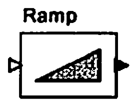


FIGURE 5.12. The Ramp icon.

convenience method, `_attachText()`, which is a protected method defined in the base class NamedObj. This method creates an attribute named "_iconDescription" with a textual value, where in this case, the textual value is:

```
<svg>
```

```
public class Ramp extends SequenceSource {

public Ramp(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
       ...
      _attachText("_iconDescription", "<svg>\n"
                + "<rect x=\"-30\" y=\"-20\" "
                + "width=\"60\" height=\"40\" "
                + "style=\"fill:white\"/>\n"
                + "<polygon points=\"-20,10 20,-10 20,10\" "
                + "style=\"fill:grey\"/>\n"
                + "</svg>\n");
   }

   ...
}
```

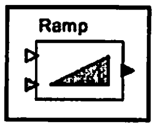


FIGURE 5.13. The Ramp actor defines a custom icon as shown.

```
    <rect x="-30" y="-20" width="60" height="40" style="fill:white"/>
    <polygon points="-20,10 20,-10 20,10" style="fill:grey"/>
</svg>
```

This is XML, using the schema SVG (scalable vector graphics). The Ptolemy II visual editor (Vergil) is built on top of a graphics package called Diva, which has limited support for SVG. As of this writing, the SVG elements that are supported are shown in figure 5.14. The positions in SVG are given by real numbers, where the values are increasing to the right and down from the origin, which is the nominal center of the figure. The Ramp icon contains a white rectangle and a polygon that forms a triangle.

Most of the elements in figure 5.14 support style attributes, as summarized in the table. A style attribute has value *keyword*:*value*. It can also have multiple *keyword*:*value* pairs, separated by semicolons. For example, the keywords currently supported by the *rect* element are "fill", "stroke" and "stroke-width." The "fill" gives the color of the body of the figure (for figures for which this makes sense), while the "stroke" gives the color of the outline. The supported colors are black, blue, cyan, darkgray, gray, green, lightgray, magenta, orange, pink, red, white, and yellow, plus any color supported by the Java Color class getColor() method. The "stroke-width" is a real number giving the thickness of the outline line, where the default is 1.0.

The image element, although tempting, is problematic in the current implementation. Images are very slow to load. It is not recommended.

5.8 Code Format

Ptolemy software follows fairly rigorous conventions for code formatting. Although many of these conventions are arbitrary, the resulting consistency makes reading the code much easier, once you get used to the conventions. We recommend that if you extend Ptolemy II in any way, that you follow these conventions. To be included in future versions of Ptolemy II, the code *must* follow the conventions.

A template that follows these rules can be found in $(PTII)/doc/coding/templates. There are also templates for other common files. In general, consult the template or highly rated (green) code if you have questions that are not covered here.

Several useful tools are provided in the directories under `$PTII/util/` to help enforce the standards. `lisp/ptjavastyle.el` is a lisp module for GNU Emacs that has appropriate indenting rules. `testsuite/jindent` is a shell script that uses Emacs and the above module to properly indent many files at once. `testsuite/ptspell` is a script that checks Java code and prints out an alphabetical list of unrecognized spellings. It properly handles namesWithEmbeddedCapitalization and has a list of author names. `testsuite/chkjava` is a shell script for checking various other potentially bad things in Java code, such as debugging code, and FIXME's.

5.8.1 Indentation

Nested statements should be indented by 4 characters, as in:

```
if (container != null) {
    Manager manager = container.getManager();
    if (manager != null) {
```

SVG element	Attributes
rect	*x*: horizontal position of the upper left corner *y*: vertical position of the upper left corner *width*: the width of the rectangle *height*: the height of the rectangle *style*: fill, stroke, stroke-width
circle	*cx*: horizontal position of the center of the circle *cy*: vertical position of the center of the circle *r*: radius of the circle *style*: fill, stroke, stroke-width
ellipse	*cx*: horizontal position of the center of the ellipse *cy*: vertical position of the center of the ellipse *rx*: horizontal radius of the ellipse *ry*: vertical radius of the ellipse *style*: fill, stroke, stroke-width
line	*x1*: horizontal position of the start of the line *y1*: vertical position of the start of the line *x2*: horizontal position of the end of the line *y2*: vertical position of the end of the line *style*: stroke, stroke-width
polyline	*points*: List of x,y pairs of points, vertices of line segments, delimited by commas or spaces *style*: stroke, stroke-width
polygon	*points*: List of x,y pairs of points, vertices of the polygon, delimited by commas or spaces *style*: fill, stroke, stroke-width
text	*x*: horizontal position of the text *y*: vertical position of the text *style*: font-family, font-size, fill
image	*x*: horizontal position of the image *y*: vertical position of the image *width*: the width of the image *height*: the height of the image *xlink:href*: A URL for the image

FIGURE 5.14. SVG subset currently supported by Diva, useful for creating custom icons.

```
            manager.requestChange(change);
        }
    }
```

Closing brackets should be on a line by themselves, aligned with the beginning of the line that contains the open bracket. Tabs are 8 space characters, not a Tab character. The reason for this is that code becomes unreadable when the Tab character is interpreted differently by different programs. Do not override this in your text editor[5]. Long lines should be broken up into many small lines. The easiest places to break long lines are usually just before operators, with the operator appearing on the next line. Long strings can be broken up using the + operator in Java, with the + starting the next line. Continuation lines are indented by 8 characters, as in the throws clause of the constructor in figure 5.1.

5.8.2 Spaces

Use a space after each comma:

```
Right: foo(a, b);
Wrong: foo(a,b);
```

Use spaces around operators such as plus, minus, multiply, divide or equals signs, and after semicolons. Java keywords such as if, for and while should have a space before the opening parenthesis:

```
Right: a = b + 1;
Wrong: a=b+1;
Right: for (i = 0; i < 10; i += 2)
Wrong: for(i=0 ;i<10;i+=2)
```

5.8.3 Comments

Comments should be complete sentences and complete thoughts, capitalized at the beginning and with a period at the end. Spelling and grammar should be correct. Comments should include honest information about the limitations of the object definition.

Comments for base class methods that are intended to be overridden should include information about what the method generally does, along with a description of how the base class implements it. Comments in derived classes for methods that override the base class should copy the general description from the base class, and then document the particular implementation. In general comments with FIXME's and implementation details should be used liberally in the code, but never in the interface description. (The interface description is the sum of all the Javadoc comments. These are the comments that will be visible in Vergil via the Get Documentation context menu choice.) If something is important to know when using the actor, put it in one of the Javadoc comments. Otherwise, put the comment elsewhere.

5.8.4 Names

In general, the names of classes, methods and members should consist of complete words separated using internal capitalization[6]. Class names, and only class names have their first letter capitalized, as in AtomicActor. Method and member names are not capitalized, except at internal word

5. For information about how to set this in Eclipse, see $PTII/doc/coding/eclipse.htm
6. Yes, there are exceptions (NamedObj, CrossRefList, IOPort). Many discussions dealt with these names, and we still regret not making them complete words.

boundaries, as in getContainer(). Protected or private members and methods are preceded by a leading underscore "_" as in _protectedMethod().

Static final constants should be in uppercase, with words separated by underscores, as in INFINITE_CAPACITY. A leading underscore should be used if the constant is protected or private.

Package names should be short and not capitalized, as in "de" for the discrete-event domain.

In Java, there is no limit to name sizes (as it should be). Do not hesitate to use long names.

5.8.5 Exceptions

A number of exceptions are provided in the ptolemy.kernel.util package. Use these exceptions when possible because they provide convenient constructor arguments of type Nameable that identify the source of the exception by name in a consistent way.

A key decision you need to make is whether to use a compile-time exception or a run-time exception. A run-time exception is one that implements the RuntimeException interface. Run-time exceptions are more convenient in that they do not need to be explicitly declared by methods that throw them. However, this can have the effect of masking problems in the code.

The convention we follow is that a run-time exception is acceptable only if the cause of the exception can be tested for prior to calling the method. This is called a *testable precondition*. For example, if a particular method will fail if the argument is negative, and this fact is documented, then the method can throw a run-time exception if the argument is negative. On the other hand, consider a method that takes a string argument and evaluates it as an expression. The expression may be malformed, in which case an exception will be thrown. Can this be a run-time exception? No, because to determine whether the expression is malformed, you really need to invoke the evaluator. Making this a compile-time exception forces the caller to explicitly deal with the exception, or to declare that it too throws the same exception. In general, we prefer to use compile-time exceptions wherever possible.

When throwing an exception, the detail message should be a complete sentence that includes a string that fully describes what caused the exception. For example

```
throw IllegalActionException(this,
        "Cannot append an object of type: "
        + obj.getClass().getName() + "because "
        + "it does not implement Cloneable.");
```

Note that the exception not only gives a way to identify the objects that caused the exception, but also why the exception occurred. There is no need to include in the message an identification of the "this" object passed as the first argument to the exception constructor. That object will be identified when the exception is reported to the user.

Sometimes, a piece of code will throw an Exception that is caught and then rethrown as a Ptolemy Exception. The Ptolemy Kernel exceptions usually have a constructor that takes a *cause* argument, which is the exception that caused the exception that is being constructed. When the exception is displayed, the stack trace of the initial exception that was passed in via the cause argument will also be displayed. For example, failure to open a file might throw an IOException, which will be caught and rethrown as an IllegalActionException:

```
try {
  // Code that throws an IOException
} catch (IOException ex)
```

```
        throw new IllegalActionException(this, ex,
            "Failed to open '" + fileName + "'");
    }
```

5.8.6 Javadoc

Javadoc is a program distributed with Java that generates HTML documentation files from Java source code files. Javadoc comments begin with "/**" and end with "*/". The comment immediately preceding a method, member, or class documents that method, member, or class. Ptolemy II classes include Javadoc documentation for all classes and all public and protected members and methods. Pay special attention to the first sentence of each Javadoc comment. This first sentence is used as a summary in the Javadocs. Private members and methods need not be documented by Javadoc comments. Documentation can include embedded HTML formatting. For example, by convention, in actor documentation, we set in italics the names of the ports and parameters using the syntax

```
/** In this actor, inputs are read from the <i>input</i> port ... */
```

By convention, method names are set in the default font, but followed by empty parentheses, as in

```
/** The fire() method is called when ... */
```

The parentheses are empty even if the method takes arguments. The arguments are not shown. If the method is overloaded (has several versions with different argument sets), then the text of the documentation needs to distinguish which version is being used.

It is common in the Java community to use the following style for documenting methods:

```
/** Sets the expression of this variable.
 *  @param expression The expression for this variable.
 */
public void setExpression(String expression) {
    ...
}
```

We use instead the imperative tense, as in

```
/** Set the expression of this variable.
 *  @param expression The expression for this variable.
 */
public void setExpression(String expression) {
    ...
}
```

The reason we do this is that our sentence is a well-formed, grammatical English sentence, while the usual convention is not (it is missing the subject). Moreover, calling a method is a command "do this," so it seems reasonable that the documentation say "Do this." The use of imperative tense has a large impact on how interfaces are documented, especially when using the Listener design pattern. For instance, the java.awt.event.ItemListener interface has the method:

```
/**
 * Invoked when an item has been selected or deselected.
 * The code written for this method performs the operations
 * that need to occur when an item is selected (or deselected).
 */
void itemStateChanged(ItemEvent e);
```

A naive attempt to rewrite this in imperative tense might result in:

```
/**
 * Notify this object that an item has been selected or deselected.
 */
void itemStateChanged(ItemEvent e);
```

However, this sentence does not capture what the method does. The method may be called *in order to* notify the listener, but the *listener* does not "notify this object". The correct way to concisely document this method in imperative tense (and with meaningful names) is:

```
/**
 * React to the selection or deselection of an item.
 */
void itemStateChanged(ItemEvent event);
```

The annotation for the arguments (the @param statement) is not a complete sentence, since it is usually presented in tabular format. However, we do capitalize it and end it with a period.

Exceptions that are thrown by a method need to be identified in the Javadoc comment. An @exception tag should read like this:

```
* @exception MyException If such and such occurs.
```

Notice that the body always starts with "If", not "Thrown if", or anything else. Just look at the Javadoc output to see why this occurs. In the case of an interface or base class that does not throw the exception, use the following:

```
* @exception MyException Not thrown in this base class. Derived
*  classes may throw it if such and such happens.
```

The exception still has to be declared so that derived classes can throw it, so it needs to be documented as well.

The Javadoc program gives extensive diagnostics when run on a source file. Our policy is to format the comments until there are no Javadoc warnings.

5.8.7 Code Organization

The basic file structure that we use follows the outline in figure 5.1, preceded by a one-line description of the file and a copyright notice. The key points to note about this organization are:

- The file is divided into sections with highly visible delimiters. The sections contain constructors, ports and parameters (and other public members, if there are any), public methods, protected methods, protected members, private methods, and private members, in that order. Note in particular that although it is customary in the Java community to list private members at the beginning of a class definition, we put them at the end. They are not part of the public interface, and thus should not be the first thing you see.
- Within each section, methods appear in alphabetical order, in order to easily search for a particular method. If you wish to group methods together, try to name them so that they have a common prefix. Static methods are generally mixed with non-static methods.

6

MoML

Authors: *Edward A. Lee*
Steve Neuendorffer
Contributor: *Christopher Hylands*

6.1 Introduction

Ptolemy II models might be *simulations* (executable models of some other system) or *implementations* (the system itself). They might be classical computer programs (applications), or any of a number of network-integrated programs (applets, servlets, or CORBA services, for example).

Models can be specified in a number of ways. You can write Java code that instantiates components, parameterizes them, and interconnects them. Or you can use Vergil (see the Vergil chapter above) to graphically construct models. Vergil stores models in ASCII files using an XML schema called MoML. MoML (which stands for Modeling Markup Language) is the primary persistent file format for Ptolemy II models. It is also the primary mechanism for constructing models whose definition and execution is distributed over the network.

This chapter explains MoML. Most users will not need to edit MoML files directly. Use Vergil instead. Occasionally, however, it is useful to examine and/or edit MoML files directly.

MoML is a modeling markup schema in the Extensible Markup Language (XML). It is intended for specifying interconnections of parameterized components. A MoML file can be executed as an application using any of the following commands,

```
ptolemy filename.xml
ptexecute filename.xml
vergil filename.xml
moml configuration.xml filename.xml
```

These commands are defined in the directory `$PTII/bin`, which must be in your path[1], where `$PTII` is the location of the Ptolemy II installation. In all cases, the filename can be replaced by a URL. The `ptolemy` command assumes that the file defines an executable Ptolemy II model, and opens a control panel to execute it. The `ptexecute` command executes it without a control panel. The `vergil` command opens a graphical editor to edit and execute the model. The `moml` command uses the specified configuration file (a MoML file containing a Ptolemy II configuration) to invoke some set of customized views or editors on the model. The filename extension can be ".xml" or ".moml" for MoML files. And the same XML file can be used in an applet[2].

To get a quick start, try entering the following into a file called `test.xml` (This file is also available as $PTII/ptolemy/moml/demo/test.xml):

```
<?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="test" class="ptolemy.actor.TypedCompositeActor">
  <property name="director"
            class="ptolemy.domains.sdf.kernel.SDFDirector"/>
  <entity name="ramp" class="ptolemy.actor.lib.Ramp"/>
  <entity name="plot" class="ptolemy.actor.lib.gui.SequencePlotter"/>
  <relation name="r" class="ptolemy.actor.TypedIORelation"/>
  <link port="ramp.output" relation="r"/>
  <link port="plot.input" relation="r"/>
</entity>
```

This code defines a model in a top-level entity called "test". By convention, we use the same name for the top-level model and the file in which it resides. The top-level model is an instance of the Ptolemy II class `ptolemy.actor.TypedCompositeActor`. It contains a director, two entities, a relation, and two links. The model is depicted in figure 6.1, where the director is not shown. It can be run using the command

```
ptolemy test.xml
```

You should get a window looking like that in figure 6.2. Enter "10" in the iterations box and hit the "Go" button to execute the model for 10 iterations (leaving the default "0" in the iterations box exe-

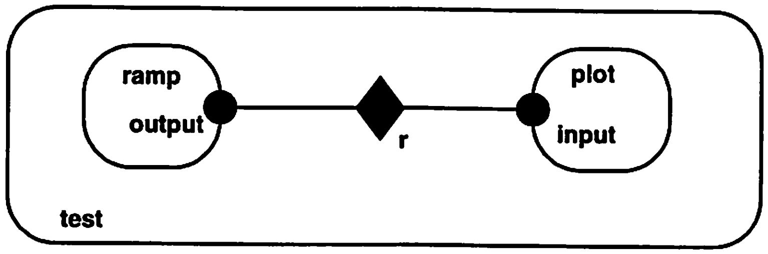


FIGURE 6.1. Simple example in the file $PTII/ptolemy/moml/demo/test.xml.

1. These commands are executed this way on Unix systems and on Windows systems with Cygwin installed. On other configurations, the equivalent commands are invoked in some other way.
2. An *applet* is a Java program that is downloaded from a web server by a browser and executed in the client's computer (usually within a plug-in for the browser).

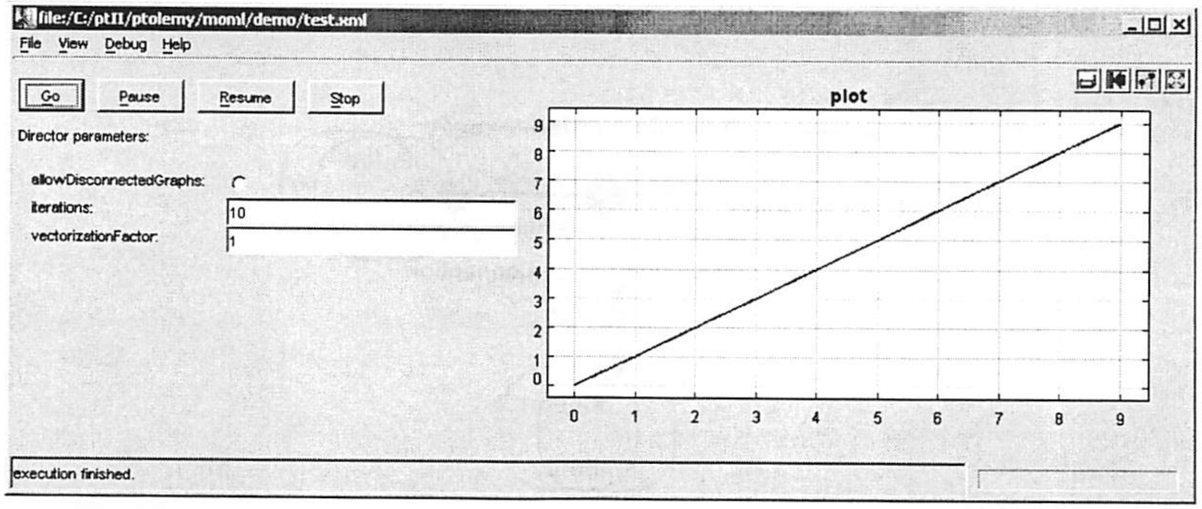
cutes it forever, until you hit the "Stop" button).

The structure of the above MoML text is explained in detail in this chapter. A more interesting example is given in the appendix to this chapter. You may wish to refer to that example as you read about the details. The next chapter explains how to bypass MoML and write applets directly. The chapter after that describes the actors libraries that are included in the current Ptolemy II version.

6.2 MoML Principles

The key features of MoML include:

- *Web integration.* MoML is an XML schema. XML, the popular *extensible markup language*[122], provides a standard syntax and a standard way of defining the content within that syntax. The syntax is a subset of SGML[123], and is similar to HTML. It is intended for use on the Internet, and is intended for precisely this sort of specialization into schemas. File references are via URIs (in practice, URLs), both relative and absolute, so MoML is equally comfortable working in applets and applications.
- *Implementation independence.* MoML is designed to work with a variety of tools. A modeling tool that reads MoML files is expected to provide a class loader in some form. Given the name of a class, and possibly a URL for the class definition, the class loader must be able to instantiate it. Classes might be defined in MoML or in some base language such as Java. In Java, the class loader could be that built in to the JVM. In C++ or other languages, the class loader would have to be implemented by the modeling tool. Ptolemy II can be viewed as a reference implementation of a MoML tool that uses Java as its base language.
- *Extensibility.* Components can be parameterized in two ways. First, they can have named properties with string values. Second, they can be associated with an external configuration file that can be in any format understood by the component. Typically, the configuration will be in some other XML schema, such as PlotML or SVG (scalable vector graphics).
- *Classes and inheritance.* Components can be defined in MoML as classes which can then be



Director parameters:
allowDisconnectedGraphs:
iterations: 10
vectorizationFactor: 1

execution finished.

FIGURE 6.2. Simple example of a Ptolemy II model execution control window.

instantiated in a model. Components can extend other components through an object-oriented inheritance mechanism.

- *Semantics independence.* MoML defines no semantics for an interconnection of components. It represents only the hierarchical containment relationships between entities with properties, their ports, and the connections between their ports. In Ptolemy II, the meaning of a connection (the semantics of the model) is defined by the director for the model, which is a property of the top-level entity. The director defines the semantics of the interconnection. MoML knows nothing about directors except that they are instances of classes that can be loaded by the class loader and assigned as properties.

6.2.1 Clustered Graphs

A model is given as a clustered graph, which is an *abstract syntax* for representing netlists, state transition diagrams, block diagrams, etc. An abstract syntax is a conceptual data organization. A particular clustered graph configuration is called a *topology*. A topology is a collection of *entities* and *relations*. Furthermore, entities have *ports* and relations connect the ports. We consistently use the term *connection* to denote the association between connected ports (or their entities), and the term *link* to denote the association between ports and relations. Thus, a connection consists of a relation and two or more links.

The concept of an abstract syntax can be contrasted with a *concrete syntax*, which is a persistent, readable representation of the data. For example, EDIF is a concrete syntax for representing netlists. MoML is a concrete syntax for the clustered graph abstract syntax. Furthermore, we use the visual notation shown in figure 6.3, where entities are depicted as rounded boxes, relations as diamonds, and entities as filled circles.

The use of ports and hierarchy distinguishes our topologies from mathematical graphs. In a mathematical graph, an entity would be a vertex, and an arc would be a connection between entities. A vertex could be represented in our schema using entities that always contain exactly one port. In a directed graph, the connections are divided into two subsets, one consisting of incoming arcs, and the other of outgoing arcs. The vertices in such a graph could be represented by entities that contain two ports, one for incoming arcs and one for outgoing arcs. Thus, in mathematical graphs, entities always have one or two ports, depending on whether the graph is directed. Our schema generalizes this by permitting an entity to have any number of ports, thus dividing its connections into an arbitrary number of subsets.

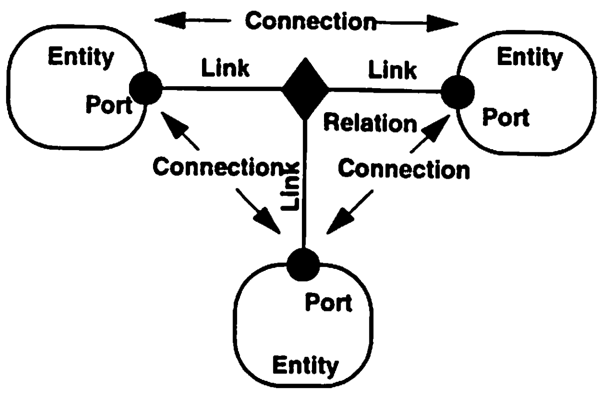


FIGURE 6.3. Visual notation and terminology.

A second difference between our graphs and mathematical graphs is that our relations are multiway associations, whereas an arc in a graph is a two-way association. A third difference is that mathematical graphs normally have no notion of hierarchy (clustering).

Relations are intended to serve a mediators, in the sense of the Mediator design pattern[35]. "Mediator promotes loose coupling by keeping objects from referring to each other explicitly..." For example, a relation could be used to direct messages passed between entities. Or it could denote a transition between states in a finite state machine, where the states are represented as entities. Or it could mediate rendezvous between processes represented as entities. Or it could mediate method calls between loosely associated objects, as for example in remote method invocation over a network.

6.2.2 Abstraction

Composite entities (clusters) are entities that can contain a topology (entities and relations). Clustering is illustrated by the example in figure 6.4. A port contained by a composite entity has inside as well as outside links. Such a port serves to expose ports in the contained entities as ports of the composite. This is the converse of the "hiding" operator often found in process algebras. Ports within an entity are hidden by default, and must be explicitly exposed to be visible (linkable) from outside the entity[3]. The composite entity with ports thus provides an abstraction of the contents of the composite.

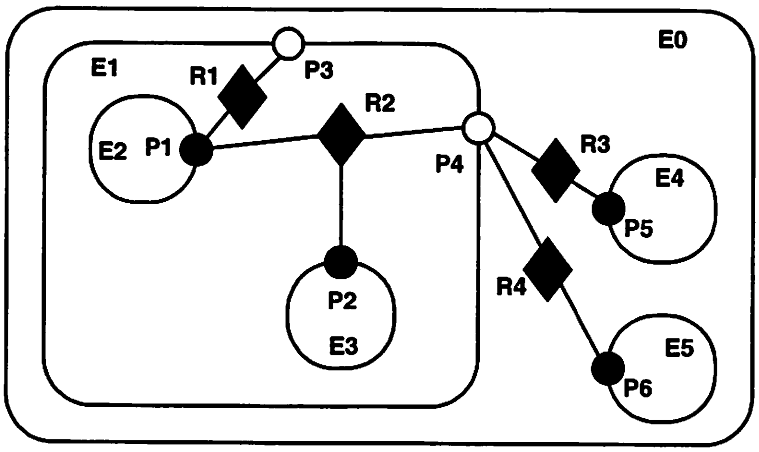


FIGURE 6.4. Ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

3. Unless level-crossing links are allowed. MoML supports these, but they are discouraged.

6.3 Specification of a Model

In this section, we describe the XML elements that are used to define MoML models.

6.3.1 Data Organization

As with all XML files, MoML files have two parts, one defining the MoML language and one containing the model data. The first part is called the *document type definition*, or DTD. This dual specification of content and structure is a key XML innovation. The DTD for MoML is given in figure 6.5. If you are adept at reading these, it is a complete specification of the schema. However, since it is not particularly easy to read, we explain its key features here.

Every MoML file must either contain or refer to a DTD. The simplest way to do this is with the following file structure:

```
<?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="modelname" class="classname">
  model definition ...
</entity>
```

Here, "`model definition`" is a set of XML elements that specify a clustered graph. The syntax for these elements is described in subsequent sections. The first line above is required in any XML file. It asserts the version of XML that this file is based on (1.0) and states that the file includes external references (in this case, to the DTD). The second and third lines declare the document type (model) and provide references to the DTD.

The references to the DTD above refer to a "public" DTD. The name of the DTD is

```
-//UC Berkeley//DTD MoML 1//EN
```

which follows the standard naming convention of public DTDs. The leading dash "-" indicates that this is not a DTD approved by any standards body. The first field, surrounded by double slashes, is the name of the "owner" of the DTD, "`UC Berkeley`." The next field is the name of the DTD, "`DTD MoML 1`" where the "`1`" indicates version 1 of the MoML DTD. The final field, "`EN`" indicates that the language assumed by the DTD is English. The Ptolemy II MoML parser requires that the public DTD be given exactly as shown, or it will not recognize the file as MoML.

In addition to the name of the DTD, the `DOCTYPE` element includes a URL pointing to a copy of the DTD on the web. If a particular MoML tool does not have access to a local copy of the DTD, then it finds it at this web site.

The "entity" element may be replaced by a "class" element, as in:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE class PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<class name="modelname" class="classname">
```

```
<!ELEMENT class (class | configure | deleteEntity | deletePort | deleteRelation | director |
  doc | entity | group | import | input | link | port | property | relation | rename |
  rendition | unlink)*>
<!ATTLIST class name CDATA #REQUIRED
              extends CDATA #IMPLIED
              source CDATA #IMPLIED>

<!ELEMENT configure (#PCDATA)>
<!ATTLIST configure source CDATA #IMPLIED>

<!ELEMENT deleteEntity EMPTY>
<!ATTLIST deleteEntity name CDATA #REQUIRED>

<!ELEMENT deletePort EMPTY>
<!ATTLIST deletePort name CDATA #REQUIRED>

<!ELEMENT deleteProperty EMPTY>
<!ATTLIST deleteProperty name CDATA #REQUIRED>

<!ELEMENT deleteRelation EMPTY>
<!ATTLIST deleteRelation name CDATA #REQUIRED>

<!ELEMENT doc (#PCDATA)>
<!ATTLIST doc name CDATA #IMPLIED>

<!ELEMENT entity (class | configure | deleteEntity | deletePort | deleteRelation | director |
  doc | entity | group | import | input | link | port | property | relation | rename |
  rendition | unlink)*>
<!ATTLIST entity name CDATA #REQUIRED
               class CDATA #IMPLIED
               source CDATA #IMPLIED>

<!ELEMENT group ANY>
<!ATTLIST group name CDATA #IMPLIED>

<!ELEMENT input EMPTY>
<!ATTLIST input source CDATA #REQUIRED>

<!ELEMENT link EMPTY>
<!ATTLIST link insertAt CDATA #IMPLIED
               insertInsideAt CDATA #IMPLIED
               port CDATA #REQUIRED
               relation CDATA #IMPLIED
               vertex CDATA #IMPLIED>
<!ELEMENT port (configure | doc | property | rename)*>
<!ATTLIST port class CDATA #IMPLIED
               name CDATA #REQUIRED>
<!ELEMENT property (configure | doc | property | rename)*>
<!ATTLIST property class CDATA #IMPLIED
                   name CDATA #REQUIRED
                   value CDATA #IMPLIED>
<!ELEMENT relation (configure | doc | property | rename | vertex)*>
<!ATTLIST relation name CDATA #REQUIRED
                   class CDATA #IMPLIED>
<!ELEMENT rename EMPTY>
<!ATTLIST rename name CDATA #REQUIRED>
<!ELEMENT unlink EMPTY>
<!ATTLIST unlink index CDATA #IMPLIED
               insideIndex CDATA #IMPLIED
               port CDATA #REQUIRED
               relation CDATA #IMPLIED>
<!ELEMENT vertex (configure | doc | location | property | rename)*>
<!ATTLIST vertex name CDATA #REQUIRED
                 pathTo CDATA #IMPLIED
                 value CDATA #IMPLIED>
```

FIGURE 6.5. MoML version 1.2 DTD.

```
    class definition ...
</class>
```

We will say more about class definitions below.

6.3.2 Overview of XML

An XML document consists of the header tags "`<?xml ... ?>`" and "`<!DOCTYPE ... >`" followed by exactly one *element*. The element has the structure:

```
start tag
body
end tag
```

where the start tag has the form

```
<elementName attributes>
```

and the end tag has the form

```
</elementName>
```

The body, if present, can contain additional elements as well as arbitrary text. If the body is not present, then the element is said to be *empty*; it can optionally be written using the shorthand:

```
<elementName attributes/>
```

where the body and end tag are omitted.

The attributes are given as follows:

```
<elementName attributeName="attributeValue" .../>
```

Which attributes are legal in an element is defined by the DTD. The quotation marks delimit the value of the attributes, so if the attribute value needs to contain quotation marks, then they must be given using the special XML entity "`&quot;`" as in the following example:

```
<elementName attributeName="&quot;foo&quot;"/>
```

The value of the attribute will be

```
"foo"
```

(with the quotation marks).

In XML "`&quot;`" is called an *entity*, creating possible confusion with our use of entity in Ptolemy II. In XML, an entity is a named storage unit of data. Thus, "`&quot;`" references an entity called "`quot`" that stores a double quote character.

6.3.3 Names and Classes

Most MoML elements have *name* and *class* attributes. The name is a handle for the object being defined or referenced by the element. In MoML, the same syntax is used to reference a pre-existing object as to create a new object. If a new object is being created, then the class attribute (usually) must be given. If a pre-existing object is being referenced, or if the MoML reader has a built-in default class for the element, then the class attribute is optional. If the class attribute is given, then the pre-existing object must be an instance of the specified class.

A name is either absolute or relative. Absolute names begin with a period "." and consist of a series of name fields separated by periods, as in ".x.y.z". Each name field can have alphanumeric characters, spaces, or the underscore "_" character. The first field is the name of the top-level model or class object. The second field is the name of an object immediately contained by that top-level.

Any name that does not begin with a period is relative to the current context, the object defined or referenced by an enclosing element. The first field of such a name refers to or defines an object immediately contained by that object. For example, inside of an object with absolute name ".x" the name "y.z" refers to an object with absolute name ".x.y.z".

A name is required to be unique within its container. That is, in any given model, the absolute names of all the objects must be unique. There can be two objects named "z", but they must not be both contained by ".x.y".

Not much more will be said about classes. Particular implementations of MoML can use this field as necessary to specify different variations of the basic syntactic objects. The class names that are used in the Ptolemy II implementation of MoML are always fully qualified Java class names. In addition, in Ptolemy II a MoML file can be referenced as a class in the same way

6.3.4 Top-Level Entities

A very simple MoML file looks like this:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="modelname" class="classname">
</entity>
```

The *entity* element has name and class attributes, and defines a Ptolemy II model. This value of the class attribute must be a class that instantiable by the MoML tool. For example, in Ptolemy II, we can define a model with:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="ptIImodel" class="ptolemy.actor.TypedCompositeActor">
</entity>
```

Here, `ptolemy.actor.TypedCompositeActor` is a class that a Java class loader can find and that the MoML parser can instantiate. In Ptolemy II, it is a container class for clustered graphs representing executable models or libraries of instantiable model classes. A model can be an instance of ptolemy.kernel.util.NamedObj or any derived class, although most useful models will be instances of ptolemy.kernel.CompositeEntity or a derived class. TypedCompositeActor, as in the above example, is derived from CompositeEntity.

6.3.5 Entity Element

A model typically contains entities, as in the following Ptolemy II example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="ptIImodel" class="ptolemy.actor.TypedCompositeActor">
  <entity name="source" class="ptolemy.actor.lib.Ramp"/>
  <entity name="sink" class="ptolemy.actor.lib.SequencePlotter"/>
</entity>
```

Notice the common XML shorthand here of writing "<entity ... />" rather than "<entity ...></entity>." Of course, the shorthand only works if there is nothing in the body of the entity element.

An entity can contain other entities, as shown in this example:

```
<entity name="ptIImodel" class="ptolemy.actor.TypedCompositeActor">
  <entity name="container" class="ptolemy.actor.TypedCompositeActor">
    <entity name="source" class="ptolemy.actor.lib.Ramp"/>
  </entity>
</entity>
```

An entity must specify a class unless the entity already exists in the containing entity or model. The name of the entity reflects the container hierarchy. Thus, in the above example, the *source* entity has the full name ".ptIImodel.container.source".

The definition of an entity can be distributed in the MoML file. Once created, it can be referred to again by name as follows:

```
<entity name="top" class="classname">
  <entity name="x" class="classname"/>
  ...
  <entity name="x">
    <property name="y">
  </entity>
</entity>
```

The property element (see section 6.3.6 below) is added to the pre-existing entity with name "x" when the second entity element is encountered.

In principle, MoML supports multiple containment, as in the following:

```
<entity name="top" class="classname">
  <entity name="x" class="classname"/>
  ...
```

```
        <entity name="y" class="classname">
            <entity name=".top.x"/>
        </entity>
    </entity>
```

Here, the element named "x" appears both in "top" and in ".top.y", i.e. the same instance appears in two different places. Thus, it would have two full names, ".top.x" and ".top.y.x". However, Ptolemy II does not support this, as it implements a strict container relationship, where an object can have only one container. Thus, attempting to parse the above MoML will result in an exception being thrown.

6.3.6 Properties

Entities (and some other elements) can be parameterized. There are two mechanisms. The simplest one is to use the *property* element:

```
<entity name="source" class="ptolemy.actor.lib.Ramp">
    <property name="init"
              value="5"
              class="ptolemy.data.expr.Parameter"/>
</entity>
```

The property element has a name, at minimum (the value and class are optional). It is common for the enclosing class to already contain properties, in which case the property element is used only to set the value. For example:

```
<entity name="source" class="ptolemy.actor.lib.Ramp">
    <property name="init" value="5"/>
</entity>
```

In the above, the enclosing object (*source*, an instance of `ptolemy.actor.lib.Ramp`) must already contain a property with the name *init*. This is typically how library components are parameterized. In Ptolemy II, the value of a property may be an expression, as in "`PI/50`". The expression may refer to other properties of the containing entity or of its container. Note that the expression language is not part of MoML, but is rather part of Ptolemy II. In MoML, a property value is simply an uninterpreted string. It is up to a MoML tool, such as Ptolemy II, to interpret that string.

A property can be declared without a class and without a pre-existing property if it is a *pure property*, one with only a name and no value. For example:

```
<entity name="source" class="ptolemy.actor.lib.Ramp">
    <property name="abc"/>
</entity>
```

A property can also contain a property, as in

```
<property name="x" value="5">
    <property name="y" value="10"/>
</property>
```

A second, much more flexible mechanism is provided for parameterizing entities. The *configure* element can be used to specify a relative or absolute URL pointing to a file that configures the entity, or it can be used to include the configuration information in line. That information need not be MoML information. It need not even be XML, and can even be binary encoded data (although binary data cannot be in line; it must be in an external file). For example,

```
<entity name="sink" class="ptolemy.actor.lib.SequencePlotter">
  <configure source="url"/>
</entity>
```

Here, *url* can give the name of a file containing data, or a URL for a remote file. (For the SequencePlotter actor, that external data will have PlotML syntax; PlotML is another XML schema for configuring plotters.) Configure information can also be given in the body of the MoML file as follows:

```
<entity name="sink" class="ptolemy.actor.lib.SequencePlotter">
  <configure>
    configure information
  </configure>
</entity>
```

With the above syntax, the configure information must be textual data. It can contain XML markup with only one restriction: if the tag "`</configure>`" appears in the textual data, then it must be preceeded by a matching "`<configure>`". That is, any configure elements in the markup must have balanced start and end tags.[4]

You can give both a source attribute and in-line configuration information, as in the following:

```
<entity name="sink" class="ptolemy.actor.lib.SequencePlotter">
  <configure source="url">
    configure information
  </configure>
</entity>
```

In this case, the file data will be passed to the application first, followed by the in-line configuration data.

In Ptolemy II, the configure element is supported by any class that implements the Configurable interface. That interface defines a configure() method that accepts an input stream. Both external file data and in-line data are provided to the class as a character stream by calling this method.

There is a subtle limitation with using markup within the configure element. If any of the elements within the configure element match MoML elements, then the MoML DTD will be applied to assign default values, if any, to their attributes. Thus, this mechanism works best if the markup within the configure element is not using an XML schema that happens to have element names that match those in MoML. Alternatively, if it does use MoML element names, then those elements are used with their MoML meaning. This limitation can be fixed using XML namespaces, something we will eventually

4. XML allow markup to be included in arbitrary data as long as it appears within either a processing instruction or a CDATA body. However, for reasons that would only baffle anyone familiar with modern programming languages, processing instructions and CDATA bodies cannot be nested within one another. The MoML configure element can be nested, so it offers a much more flexible mechanism than the standard ones in XML.

implement.

6.3.7 Doc Element

Some elements can be documented using the *doc* element. For example,

```
<entity name="source" class="ptolemy.actor.lib.Ramp">
  <property name="init" value="5">
    <doc>Initialize the ramp above the default because... </doc>
  </property>
  <doc>
  This actor produces an increasing sequence beginning with 5.
  </doc>
</entity>
```

With the above syntax, the documentation information must be textual data. It can include markup, as in the following example, which uses XHTML[5] formatting within the doc element:

```
<entity name="source" class="ptolemy.actor.lib.Ramp">
  <doc><H1>Using HTML</H1>Text with <I>markup</I>.</doc>
</entity>
```

An alternative method is to use an XML processing instruction as follows:

```
<entity name="source" class="ptolemy.actor.lib.Ramp">
  <doc><?xhtml <H1>Using HTML</H1>Text with <I>markup</I>.?></doc>
</entity>
```

This requires that any utility that uses the documentation information be able to handle the `xhtml` processing instruction, but it makes it very clear that the contents are XHTML. However, for reasons we do not understand, XML does not allow processing instructions to be nested, so this technique has its limitations.

More than one doc element can be included in an element. To do this, give each doc element a name, as follows:

```
<entity name="entityname" class="classname">
  <doc name="docname">
    doc contents
  </doc>
</entity>
```

The name must not conflict with any preexisting property. If a doc element or a property with the specified name exists, then it is removed and replaced with the property. If no name is given, then the doc element is assigned the name "_doc".

5. XHTML is HTML with additional constraints so that it conforms with XML syntax rules. In particular, every start tag must be matched by an end tag, something that ordinary HTML does not require (but fortunately, does allow).

A common convention, used in Ptolemy II, is to add doc elements with the name "tooltip" to define a tooltip for GUI views of the component. A tooltip is a small window with short documentation that pops up when the mouse lingers on the graphical component.

Note that the same limitation of using markup within configure elements also applies to doc elements.

6.3.8 Ports

An entity can declare a port:

```
<entity name="A" class="classname">
  <port name="out"/>
</entity>
```

In the above example, no class is given for the port. If a port with the specified name already exists in the class for entity A, then that port is the one referenced. Otherwise, a new port is created in Ptolemy II by calling the newPort() method of the container. Alternatively, we can specify a class name, as in

```
<entity name="A" class="classname">
  <port name="out" class="classname"/>
</entity>
```

In this case, a port will be created if one does not already exist. If it does already exist, then its class is checked for consistency with the declared class (the pre-existing port must be an instance of the declared class). In Ptolemy II, the typical classname for a port would be

```
ptolemy.actor.TypedIOPort
```

In Ptolemy II, the container of a port is required to be an instance of ptolemy.kernel.Entity or a derived class.

It is often useful to declare a port to be an input, an output, or both. To do this, enclose in the port a property named "input" or "output" or both, as in the following example:

```
<port name="out" class="ptolemy.actor.IOPort">
  <property name="output"/>
</port>
```

This is an example of a pure property. Optionally, the property can be given a boolean value, as in

```
<port name="out" class="ptolemy.actor.IOPort">
  <property name="output" value="true"/>
</port>
```

The value can be either "true" or "false", where the latter will define the port to not be an output. A port can be defined to be both an input and an output, as follows

```
<port name="out" class="ptolemy.actor.IOPort">
```

```
        <property name="output" value="true"/>
        <property name="input" value="true"/>
    </port>
```

It is also sometimes necessary to declare that a port is a multiport. To do this, enclose in the port a property named "multiport" as in the following example:

```
    <port name="out" class="ptolemy.actor.IOPort">
        <property name="multiport"/>
    </port>
```

The enclosing port must be an instance of IOPort (or a derived class such as TypedIOPort), or else the property is treated as an ordinary property. As with the input and output attribute, the multiport property can be given a boolean value, as in

```
    <port name="out" class="ptolemy.actor.IOPort">
        <property name="multiport" value="true"/>
    </port>
```

If a port is an instance of TypedIOPort (for library actors, most are), then you can set the type of the port in MoML as follows:

```
    <port name="out" class="ptolemy.actor.IOPort">
        <property name="type"
                  value="double"
                  class="ptolemy.actor.TypeAttribute"/>
    </port>
```

This is occasionally useful when you need to constrain the types beyond what the built-in type system takes care of. The names of the built-in types are (currently) boolean, booleanMatrix, complex, complexMatrix, double, doubleMatrix, fix, fixMatrix, int, intMatrix, long, longMatrix, unsignedByte, unsignedByteMatrix, object, string, and general. These are defined in the class ptolemy.data.type.BaseType.

6.3.9 Relations and Links

To connect entities, you create relations and links. The following example describes the topology shown in figure 6.6:

```
<entity name="top" class="classname">
    <entity name="A" class="classname">
        <port name="out"/>
    </entity>
    <entity name="B" class="classname">
        <port name="out"/>
    </entity>
    <entity name="C" class="classname">
        <port name="in">
```

```
            <property name="multiport"/>
        </port>
    </entity>
    <relation name="r1" class="classname"/>
    <relation name="r2" class="classname"/>
    <link port="A.out" relation="r1"/>
    <link port="B.out" relation="r2"/>
    <link port="C.in" relation="r1"/>
    <link port="C.in" relation="r2"/>
</entity>
```

In Ptolemy II, the typical classname for a relation would be `ptolemy.actor.TypedIORelation`. The classname may be omitted, in which case the newRelation() method of the container is used to create a new relation. The container is required to be an instance of ptolemy.kernel.CompositeEntity, or a derived class. As usual, the class attribute may be omitted if the relation already exists in the containing entity.

Notice that this example has two distinct links to `C.in` from two different relations. The order of these links may be important to a MoML tool, so any MoML tool must preserve the order in which they are specified, as Ptolemy II does. We say that C has two links, indexed 0 and 1.

The `link` element can explicitly give the index number at which to insert the link. For example, we could have achieved the same effect above by saying

```
<link port="C.in" relation="r1" insertAt="0"/>
<link port="C.in" relation="r2" insertAt="1"/>
```

Whenever the insertAt option is not specified, the link is always appended to the end of the list of links.

When the insertAt option is specified, the link is inserted at that position, so any pre-existing links with larger indices will have their index numbers incremented. For example, if we do

```
<link port="C.in" relation="r1" insertAt="0"/>
<link port="C.in" relation="r2" insertAt="1"/>
<link port="C.in" relation="r3" insertAt="1"/>
```

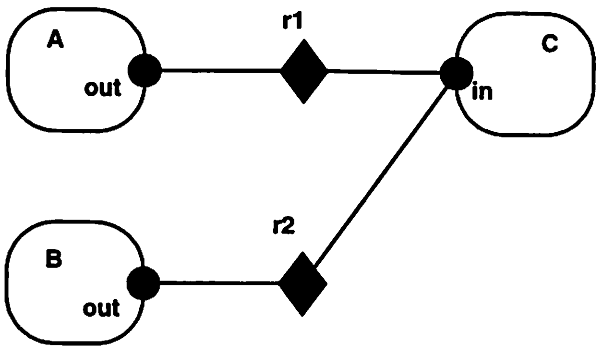


FIGURE 6.6. Example topology.

then there will be a link to r1 with index 0, a link to r2 with index 2 (note! not 1), and a link to r3 with index 1.

If the specified index is beyond the existing number of links, then null links (i.e. links to nothing) are created to fill in. So for example, if the first link we create is given by

```
<link port="C.in" relation="r2" insertAt="1"/>
```

then the port will have *two* links, not one, but the first one will be an empty link. If we then say

```
<link port="C.in" relation="r2"/>
```

then the port will have *three* links, with the first one being empty. If we then say

```
<link port="C.in" relation="r2" insertAt="0"/>
```

then there will be *four* links, with the *second* one being empty.

Normally, it is not necessary in MoML to specify whether a link occurs on the inside of a port or on the outside. This can be determined automatically by identifying the relation.For example, in figure 5.4, port P4 is linked on the inside to relation R2 and on the outside to relations R3 and R4.

However, close examination of the DTD reveals that the relation attribute is optional. If the relation attribute is not present, then a null link is inserted. However, if you do not specify a relation, then there is no way to determine whether an inside null link or an outside null link was intended. MoML defines the default to be an outside null link. To specify an inside null link, use the insertInsideAt attribute. For example, to insert a null link on the inside of P4 in figure 6.4 prior to the link to R2, use:

```
<entity name="E0.E1">
   <link port="P4.in" insertInsideAt="0"/>
</entity>
```

Note that the index number is not the same thing as the channel number in Ptolemy II. In Ptolemy II, a relation may have a width greater than one, so a single link may represent more than one channel (actually, it could even represent zero channels if that relation is not linked to another ports).

6.3.10 Classes

So far, entities have been instances of externally defined classes accessed via a class loader. They can also be instances of classes defined in MoML. To define a class in MoML, use the *class* element, as in the following example:[6]

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE class PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
   "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<class name="Gen" extends="ptolemy.actor.TypedCompositeActor">
```

6. This is a simplified version of the Sinewave class, whose complete definition is given in the appendix.

```
    <entity name="ramp" class="ptolemy.actor.lib.Ramp">
      <port name="output"/>
      <property name="step" value="2*PI/50"/>
    </entity>
    <entity name="sine" class="ptolemy.actor.lib.TrigFunction">
      <port name="input"/>
      <port name="output"/>
    </entity>
    <port name="output" class="ptolemy.actor.TypedIOPort"/>
    <relation name="r1" class="ptolemy.actor.TypedIORelation"/>
    <relation name="r2" class="ptolemy.actor.TypedIORelation"/>
    <link port="ramp.output" relation="r1"/>
    <link port="sine.input" relation="r1"/>
    <link port="sine.output" relation="r2"/>
    <link port="output" relation="r2"/>
</class>
```

The class element may be the top-level element in a file, in which case the DOCTYPE should be declared as "class" as done above. It can also be nested within a model. The above example specifies the topology shown in figure 6.7. Once defined, it can be instantiated as if it were a class loaded by the class loader:

```
<entity name="instancename" class="classname"/>
```

or

```
<entity name="instancename" class="classname" source="url"/>
```

The first form can be used if the class definition can be found from the *classname*. There are two ways that this could happen. First, the *classname* might be an absolute name for a class defined within the same top level entity that this entity element is in. Second, the *classname* might be sufficient to find the class definition in a file, much the way Java classes are found. For example, if the classname is `ptolemy.actor.lib.Sinewave` and the class is defined in the file `$PTII/ptolemy/actor/lib/Sinewave.xml`, then there is no need to use the second form to specify the URL where the class is defined. Specifically, the CLASSPATH[7] is searched for a file matching the classname. By convention, the file defining the class has the same name as the class, with the extension "`.xml`" or "`.moml`".

In the first of these techniques, the class name follows the same convention as entity names, except

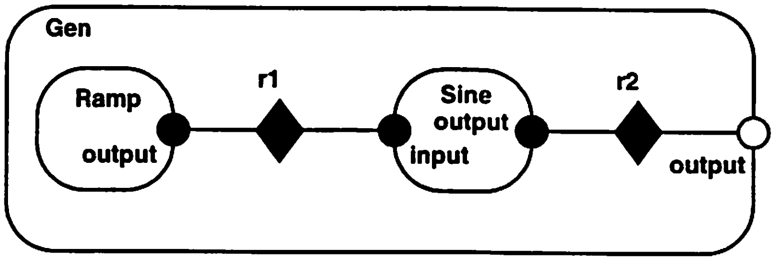


FIGURE 6.7. Sine wave generator topology.

that a classname referring to a class defined within the same MoML top-level must be absolute. In fact, a class *is* an entity with the additional feature that one can create new instances of it with the entity element. Consider for example,

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="top" extends="ptolemy.kernel.CompositeEntity">
  <class name="Gen" extends="ptolemy.actor.TypedCompositeActor">
    class definition ...
  </class>
  <entity name="derived" class=".top.Gen"/>
</entity>
```

Here, the entity `derived` is an instance of `.top.Gen`, which is defined within the same MoML top level. The absolute class name is "`.top.Gen`".

The ability to give a URL as the source of a class definition is very powerful. It means that a model may be build from component libraries that are defined worldwide. There is no need to localize these. Of course, referencing a URL means the usual risks that the link will become invalid. It is our hope that reliable and trusted sources of components will emerge who will not allow this to happen.

The `Gen` class given at the beginning of this subsection generates a sine wave with a period of 50 samples. It is not all that useful without being parameterized. Let us extend it and add properties:[8]

```
<class name="Sinegen" extends="Gen">
  <property name="samplingFrequency"
          value="8000.0"
          class="ptolemy.data.expr.Parameter">
    <doc>The sampling frequency in Hertz.</doc>
  </property>
  <property name="frequency"
          value="440.0"
          class="ptolemy.data.expr.Parameter">
    <doc>The frequency in Hertz.</doc>
  </property>
  <property name="ramp.step"
          value="frequency*2*PI/samplingFrequency">
    <doc>Formula for the step size.</doc>
  </property>
  <property name="ramp.init"
          value="phase">
  </property>
</class>
```

This class extends Gen by adding two properties, and then sets the properties of the component entities

7. CLASSPATH is an environment variable that Java uses to find Java classes. The Ptolemy II implementation of MoML simply leverages this so that MoML classes can also be found if they are on the CLASSPATH.
8. This is still not quite as elaborate as the Sinewave class defined in the appendix, which is why we give it a slightly different name, Sinegen.

to have values that are expressions.

6.3.11 Inheritance

MoML supports inheritance by permitting you to extend existing classes. For example, consider the following MoML file:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
  "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="top" class="ptolemy.kernel.CompositeEntity">
  <class name="base" extends="ptolemy.kernel.CompositeEntity">
    <entity name="e1" class="ptolemy.kernel.ComponentEntity">
    </entity>
  </class>
  <class name="derived" extends=".top.base">
    <entity name="e2" class="ptolemy.kernel.ComponentEntity"/>
  </class>
  <entity name="instance" extends=".top.derived"/>
</entity>
```

Here, the "derived" class extends the "base" class by adding another entity to it, and "instance" is an instance of derived. The class "derived" can also give a source attribute, which gives a URL for the source definition.

6.3.12 Directors

Recall that a clustered graph in MoML has no semantics. However, a particular model has semantics. It may be a dataflow graph, a state machine, a process network, or something else. To give it semantics, Ptolemy II requires the specification of a director associated with a model, an entity, or a class. The director is a property of the model. The following example gives discrete-event semantics to a Ptolemy II model:

```
<entity name="top" class="ptolemy.actor.TypedCompositeActor">
  <property name="director"
            class="ptolemy.domains.de.kernel.DEDirector">
    <property name="stopTime" value="100.0"/>
  </director>
  ...
</entity>
```

This example also sets a property of the director. The name of the director is not important, except that it cannot collide with the name of any other property in the model.

6.3.13 Input Element

It is possible to insert MoML from another file or URL into a particular point in your model. For example:

```
<entity name="top" class="...">
```

```
        <entity name="a" class="...">
            <input source="url"/>
        </entity>
    </entity>
```

This takes the contents of the URL specified in the source attribute of the input element and places them inside the entity named "a". The base of the current document (the one containing the import statement) is used to interpret a relative URL, or if the current document has no base, then the current working directory is used, or if that fails, the current CLASSPATH.

6.3.14 Annotations for Visual Rendering

The abstract syntax of MoML, clustered graphs, is amenable to visual renditions as bubble and arc diagrams or as block diagrams. To support tools that display and/or edit MoML files visually, MoML allows a relation to have multiple vertices that form a path. Links can then be made to individual vertices. Consider the following example:

```
<relation name="r" class="ptolemy.actor.TypedIORelation">
  <vertex name="v1" class="classname" value="location"/>
  <vertex name="v2" class="classname" value="location" pathTo="v1"/>
</relation>
<link port="A.out" relation="r" vertex="v1"/>
<link port="B.in" relation="r" vertex="v1"/>
<link port="C.in" relation="r" vertex="v2"/>
```

This assumes that there are three entities named *A*, *B*, and *C*. The relation is annotated with a set of vertices, v1 and v2, which will normally be rendered as graphical objects. The vertices are linked together with paths, which in a simple visual tool might be straight lines, or in a more sophisticated tool might be autorouted paths. In the above example, v1 and v2 are linked by a path. The link elements specify not just a relation, but also a vertex within that relation. This tells the visual rendering tool to draw a path from the specified port to the specified vertex.

Figure 6.8 illustrates how the above fragment might be rendered. The square boxes are icons for the three entities. They have ports with arrowheads suggesting direction. There is a single relation, which shows up visually only as a set of lines and two vertices. The vertices are shown as small diamonds.

A vertex is exactly like a property, except that it has an additional attribute, pathTo, used to link vertices, and it can be referenced in a link element. Like any other property, it has a class attribute, which specifies the class implementing the vertex. In Ptolemy II, the class for a vertex is typically ptolemy.moml.Vertex. Like other properties, a vertex can have a value. This value will typically spec-

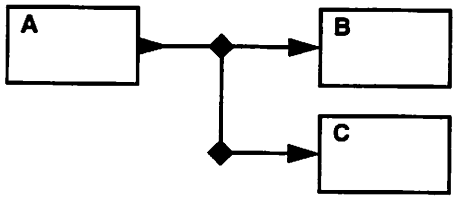


FIGURE 6.8. Example showing how MoML might be visually rendered.

ify a location for a visual rendition. For example, in Ptolemy II, the first vertex above might be given as

```
<vertex name="v1"
        class="ptolemy.moml.Vertex"
        value="184.0, 93.0"/>
```

This indicates that the vertex should be rendered at the location 184.0, 93.0.

Ptolemy II uses ordinary MoML properties to specify other visual aspects of a model. First, an entity can contain a location property, which is a hint to a visual renderer, as follows:

```
<entity name="ramp" class="ptolemy.actor.lib.Ramp">
  <property name="location"
            class="ptolemy.moml.Location"
            value="50.0, 50.0"/>
</entity>
```

This suggests to the visual renderer that the Ramp actor should be drawn at location 50.0, 50.0.

Ptolemy II also supports a powerful and extensible mechanism for specifying the visual rendition of an entity. Consider the following example:

```
<entity name="ramp" class="ptolemy.actor.lib.Ramp">
  <property name="location"
            class="ptolemy.moml.Location"
            value="50.0, 50.0"/>
  <property name="iconDescription"
            class="ptolemy.kernel.util.SingletonAttribute">
     <configure><svg>
       <rect x="0" y="0" width="80" height="20"
          style="fill:green;stroke:black;stroke-width:5"/>
     </svg></configure>
  </property>
</entity>
```

The SingletonAttribute class is used to attach an XML description of the rendition, which in this case is a wide box filled with green. The XML schema used to define the icon is SVG (scalable vector graphics), which can be found at http://www.w3.org/TR/SVG/.[9]

The rendering of the icon is done by another property of class XMLIcon, which need not be explicitly specified because the visual renderer will create it if it isn't present. However, it is possible to create totally customized renditions by defining classes derived from XMLIcon, and attaching them to entities as properties. This is beyond the scope of this chapter.

9. Currently, the Diva graphics infrastructure, which is used by Vergil to render these icons, only supports a small subset of SVG. Eventually, we hope it will support the full specification.

6.4 Incremental Parsing

MoML may be used as a command language to modify existing models, as well as being used to specify complete models. This technique is known as *incremental parsing*.

6.4.1 Adding Entities

Consider for example the simple model created as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="top" class="ptolemy.actor.TypedCompositeActor">
    ... contents of the model ...
</entity>
```

Later, the following MoML element can be used to add an entity to the model:

```
<entity name=".top">
    <entity name="inside" class="ptolemy.actor.TypedCompositeActor"/>
</entity>
```

The name of the outer entity "`.top`" is the name of the top-level model created by the first segment of MoML. (Recall that the leading period means that the name is absolute.) The line

```
<entity name=".top">
```

defines the context for evaluation of the element

```
    <entity name="inside" class="ptolemy.actor.TypedCompositeActor"/>
```

Any entity constructed in a previous parsing phase can be specified as the context for evaluation of a new MoML element.

Of course, the MoML parser must have a reference to the context in order to later parse this incremental element. This is accomplished by either using the same parser, which keeps track of the top-level entity in the last model it parsed, or by calling the setTopLevel() or setContext() methods of the parser, passing as an argument the model.

6.4.2 Using Absolute Names

Above, we have used the fact that an entity element can refer to a pre-existing element by name. That name can be relative to the context in which the entity element exists, or it can be absolute. If it is absolute, then it must nonetheless be properly contained by the enclosing entity. The following example is incorrect, and will trigger an exception:

```
<entity name="top" class="ptolemy.actor.TypedCompositeActor">
  <entity name="a" class="ptolemy.actor.TypedCompositeActor"/>
  <entity name="b" class="ptolemy.actor.TypedCompositeActor">
```

```
        <entity name=".top.a"/>
      </entity>
    </entity>
```

The "`.top.a`" cannot be specified within "b" because it is already contained within "`top`."

6.4.3 Adding Ports, Relations, and Links

A port or relation can be added to an entity that has been previously constructed by the parser. For example, assuming that `.top.inside` has been constructed as before, we can add a port to it with the following MoML segment:

```
<entity name=".top.inside">
    <port name="input" class="ptolemy.actor.TypedIOPort"/>
</entity>
```

A relation and link can then be added as follows:

```
<entity name=".top">
    <relation name="r" class="ptolemy.actor.TypedIORelation"/>
    <link port="inside.input" relation="r"/>
</entity>
```

6.4.4 Changing Port Configurations

A port that is an input can be converted to an output with the following MoML segment:

```
<port name="portname">
    <property name="input" value="false"/>
    <property name="output" value="true"/>
</port>
```

A port can be made into a multiport as follows:

```
<port name="portname">
    <property name="multiport" value="true"/>
</port>
```

6.4.5 Deleting Entities, Relations, and Ports

An entity that has been previously constructed by a parser can be deleted by evaluating MoML. For example, assuming that `.top.inside` has been constructed as before, we can delete it with the following MoML segment:

```
<entity name=".top">
    <deleteEntity name="inside"/>
</entity>
```

Any links to ports of the entity will also be deleted. Similarly, relations can be deleted using the `deleteRelation` element, and ports can be deleted using the `deletePort` element.

6.4.6 Renaming Objects

A previously existing entity can be renamed using the `rename` element, as follows:

```
<entity name="entityName">
    <rename name="newName"/>
</entity>
```

The new name is required to not have any periods in it. It consists of alphanumeric characters, the underscore, and spaces.

6.4.7 Changing Documentation, Properties, and Directors

Documentation is attached to entities using the doc element (see section 6.3.7). A doc element can optionally be given a name; if no name is given, then the name is implicitly "_doc". To replace a doc element, just give a new doc element with the same name. To remove a doc element, give a doc element with the same name and an empty body, as in

```
<doc name="docname"></doc>
```

or

```
<doc name="docname"/>
```

Properties can have their value changed using the property element (see section 6.3.6) with a new value, for example:

```
<property name="propertyname" value="propertyvalue"/>
```

A property can be deleted using the deleteProperty element

```
<deleteProperty name="propertyname"/>
```

Since a director is a property, this same mechanism can be used to remove a director.

6.4.8 Removing Links

To remove individual links, use the `unlink` element. This element has three forms. The first is

```
<unlink port="portname" relation="relationname"/>
```

This unlinks a port from the specified relation. If the port is linked more than once to the specified relation, then all links to this relation are removed. It makes no difference whether the link is an inside link or an outside link, since this can be determined from the containers of the port and the relation.

The second and third forms are

```
<unlink port="portname" index="linknumber"/>
<unlink port="portname" insideIndex="linknumber"/>
```

These both remove a link by index number. The first is used for an outside link, and the second for an inside link. The valid indices range from 0 to one less than the number of links that the port has. If the port is not a multiport, then there is at most one valid index, number 0. If an invalid index is given then the element is ignored. Note that the indexes of links above that of the removed link will be decremented by one.

The unlink element can also be used to remove null links. For example, if we have created a link with

```
<link port="portname" relation="r" insertAt="1"/>
```

where there was previously no link on this port, then this leaves a null link (not linked to anything) with index 0 (see section 6.3.9), and of course a link to relation `r` with index 1. The null link can be removed with

```
<unlink port="portname" insideIndex="0"/>
```

which leaves the link to `r` as the sole link, having index 0.

Note that the index is not the same thing as the channel number. A relation may have a width greater than one, so a single link may represent more than one channel (actually, it could even represent zero channels if that relation is not linked to other suitable ports).

6.4.9 Grouping Elements

Occasionally, you may wish to incrementally parse a set of elements. For example, in the Ptolemy II implementation, the parser has a method for setting the context, so you could set the context to a CompositeEntity and then create several entities by parsing the following MoML:

```
<entity name="firstEntity" class="classname"/>
<entity name="firstEntity" class="classname"/>
<entity name="firstEntity" class="classname"/>
```

However, the XML parser will fail to parse this because it requires that there be a single top-level element. The group element is provided for this purpose:

```
<group>
    <entity name="firstEntity" class="classname"/>
    <entity name="firstEntity" class="classname"/>
    <entity name="firstEntity" class="classname"/>
</group>
```

This element is ignored by the parser, in that it does not define a new container for the enclosed entities. It simply aggregates them, leaving the context the same as it is for the group element itself.

The group element may be given a name attribute, in which case it defines a *namespace*. All named objects (such as entities) that are immediately inside the group will have their names modified by prepending them with the name of the group and a colon. For example,

```
<group name="a">
    <entity name="b" class="classname">
        <entity name="c" class="classname"/>
    </entity>
</group>
```

The entity "b" will actually be named "a:b". The entity "c" will not be affected by the group name. Its full name, however, will be "a:b.c".

If the namespace given is "auto" then the group tag has a particular special effect. Each element contained immediately within the group that has a name will be assigned a new unique name within the container based on the specified name. Hence, if the specified name is "foo", but the container already contains an object named "foo", then a new object will be created with name "foo2" or "foo3". This feature of the group element seems rather bizarre, but it proves convenient when using MoML to cut and paste. In order to paste a group of objects into a container, those objects have to be assigned names that do not collide with names of objects already in the container. The following MoML will have that effect:

```
<group name="auto">
    <entity name="b" class="classname">
        <entity name="c" class="classname"/>
    </entity>
</group>
```

In this example, automatic naming is only applied to objects *immediately* contained by the group. Thus, the entity with name "b" may in fact be created with name "b2" (if there is already a "b"), but the entity with name "c" will have name "c".

6.5 Parsing MoML

MoML is intended to be a generic modeling markup language, not one that is specialized to Ptolemy II. As such, Ptolemy II may be viewed as a reference implementation of a MoML tool. In Ptolemy II, MoML is supported primarily by the moml package.

The moml package contains the classes shown in figure 6.9 (see appendix A of chapter 1 for UML syntax). The basis for the MoML parser is the parser distributed by Microstar. The parse() methods of the MoMLParser class read MoML data and construct a Ptolemy II model. They return the top-level model. The same parser can then be used to incrementally parse MoML segments to modify that model.

The EntityLibrary class takes particular advantage of MoML. This class extends CompositeEntity, and is designed to contain a library of entities. But it is carefully designed to avoid instantiating those entities until there is some request for them. Instead, it maintains a MoML representation of the library. This allows for arbitrarily large libraries without the overhead of instantiating components in the library that might not be needed.

Incrementa' parsing is when a MoML parser is used to modify a pre-existing model (see section 6.4). A MoML parser that was used to create the pre-existing model can be used to modify it. If there is no such parser, then it is necessary to call the `setToplevel()` method of MoMLParser to associate the parser with the pre-existing model.

Incremental parsing should (usually) be done using a change request. A change request is an active object that makes a modification to a Ptolemy model. They are queued with a composite entity container by calling its requestChange() method. This ensures that the mutation is executed only when it is safe to modify the structure of the model. The class MoMLChangeRequest (see figure 6.9) can be used for this purpose. Simply create an instance of this class, providing the constructor with a string containing the MoML code that specifies the change.

The exportMoML() methods of Ptolemy II objects can be used to produce a MoML file given a model. Thus, MoML can be used as the persistent file format for Ptolemy II models

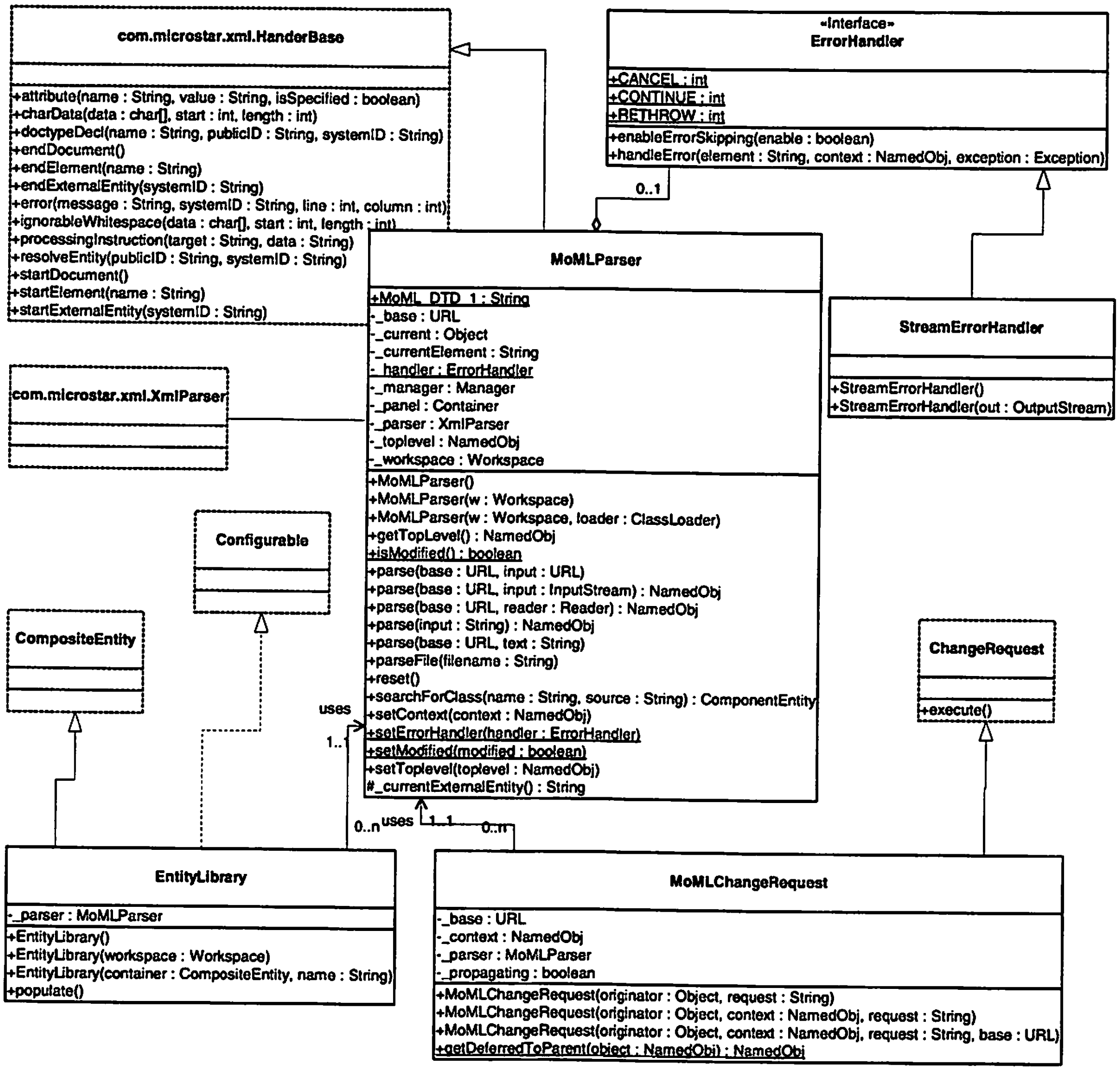

com.microstar.xml.HandlerBase
+attribute(name : String, value : String, isSpecified : boolean)
+charData(data : char[], start : int, length : int)
+doctypeDecl(name : String, publicID : String, systemID : String)
+endDocument()
+endElement(name : String)
+endExternalEntity(systemID : String)
+error(message : String, systemID : String, line : int, column : int)
+ignorableWhitespace(data : char[], start : int, length : int)
+processingInstruction(target : String, data : String)
+resolveEntity(publicID : String, systemID : String)
+startDocument()
+startElement(name : String)
+startExternalEntity(systemID : String)
«Interface»
ErrorHandler
+CANCEL : int
+CONTINUE : int
+RETHROW : int
+enableErrorSkipping(enable : boolean)
+handleError(element : String, context : NamedObj, exception : Exception)
0..1
MoMLParser
+MoML_DTD_1 : String
-_base : URL
-_current : Object
-_currentElement : String
-_handler : ErrorHandler
-_manager : Manager
-_panel : Container
-_parser : XmlParser
-_toplevel : NamedObj
-_workspace : Workspace
+MoMLParser()
+MoMLParser(w : Workspace)
+MoMLParser(w : Workspace, loader : ClassLoader)
+getTopLevel() : NamedObj
+isModified() : boolean
+parse(base : URL, input : URL)
+parse(base : URL, input : InputStream) : NamedObj
+parse(base : URL, reader : Reader) : NamedObj
+parse(input : String) : NamedObj
+parse(base : URL, text : String)
+parseFile(filename : String)
+reset()
+searchForClass(name : String, source : String) : ComponentEntity
+setContext(context : NamedObj)
+setErrorHandler(handler : ErrorHandler)
+setModified(modified : boolean)
+setToplevel(toplevel : NamedObj)
#_currentExternalEntity() : String
StreamErrorHandler
+StreamErrorHandler()
+StreamErrorHandler(out : OutputStream)
com.microstar.xml.XmlParser
Configurable
CompositeEntity
ChangeRequest
+execute()
uses
1..1
0..n
uses
1..1
0..n
EntityLibrary
-_parser : MoMLParser
+EntityLibrary()
+EntityLibrary(workspace : Workspace)
+EntityLibrary(container : CompositeEntity, name : String)
+populate()
MoMLChangeRequest
-_base : URL
-_context : NamedObj
-_parser : MoMLParser
-_propagating : boolean
+MoMLChangeRequest(originator : Object, request : String)
+MoMLChangeRequest(originator : Object, context : NamedObj, request : String)
+MoMLChangeRequest(originator : Object, context : NamedObj, request : String, base : URL)
+getDeferredToParent(object : NamedObj) : NamedObj

FIGURE 6.9. Classes supporting MoML parsing in the moml package.

6.6 Exporting MoML

Almost any Ptolemy II object can export a MoML description of itself. The following methods of NamedObj (and derived classes) are particularly useful:

```
exportMoML(): String
exportMoML(output: Writer)
exportMoML(output: Writer, depth: int)
exportMoML(output: Writer, depth: int, name: String)
_exportMoMLContents(output: Writer, depth: int)
```

Since any object derived from NamedObj can export MoML, MoML becomes an effective persistent format for Ptolemy II models. Almost everything in Ptolemy II is derived from NamedObj. It is much more compact than serializing the objects, and the description is much more durable (since serialized objects are not guaranteed to load properly into future versions of the Java virtual machine).

There is one significant subtlety that occurs when an entity is instantiated from a class defined in MoML. Consider the example:

```
<entity name="top" class="ptolemy.kernel.CompositeEntity">
    <class name="master" extends="ptolemy.kernel.ComponentEntity">
        <port name="p" class="ptolemy.kernel.ComponentPort"/>
    </class>
    <entity name="derived" class=".top.master"/>
</entity>
```

This model defines one class and one entity that instantiates that class. When we export MoML for this top-level model, we get:

```
<entity name="top" class="ptolemy.kernel.CompositeEntity">
    <class name="master" extends="ptolemy.kernel.ComponentEntity">
        <port name="p" class="ptolemy.kernel.ComponentPort">
        </port>
    </class>
    <entity name="derived" class=".top.master">
    </entity>
</entity>
```

Aside from some minor differences in syntax, this is identical to our specification above. In particular, note that the entity "derived" does not describe its port "p" even though it certainly has such a port. That port is implied because the entity instantiates the class "`.top.master`".

Suppose that using incremental parsing we subsequently modify the model as follows:

```
<entity name=".top.derived">
    <port name="q" class="ptolemy.kernel.ComponentPort"/>
</entity>
```

That is, we add a port to the instantiated entity. Then the added port *is* exported when we export

MoML. That is, we get:

```
<entity name="top" class="ptolemy.kernel.CompositeEntity">
    <class name="master" extends="ptolemy.kernel.ComponentEntity">
        <port name="p" class="ptolemy.kernel.ComponentPort">
        </port>
    </class>
    <entity name="derived" class=".top.master">
        <port name="q" class="ptolemy.kernel.ComponentPort">
        </port>
    </entity>
</entity>
```

This is what we would expect. The entity is based on the specified class, but actually extends it with additional features. Those features are persistent.

Properties are treated more simply. They are always described when MoML is exported, regardless of whether they are defined in the class on which an entity is based. The reason for this is that properties are usually modified in instances, for example by giving them new values.

There is an additional subtlety. If a topology is modified by making direct kernel calls, then exportMoML() will normally export the modified topology. However, if a derived component is modified by direct kernel calls, then exportMoML() will fail to catch the changes. In fact, only if the changes are made by evaluating MoML will the modifications be exported. This actually can prove to be convenient. It means that if a model mutates during execution, and is later saved, that a user interface can ensure that only the original model, before mutations, is saved.

6.7 Special Attributes

The moml package also includes a set of attribute classes that decorate the objects in a model with MoML-specific information, as shown in figure 6.10. These classes are used to decorate a Ptolemy II object with additional information that is relevant to a GUI or other user interface. For example, the Location class is used to specify the location of visual rendition of a component in a visual editor. A Vertex decorates a relation with one of several visual handles to which connections can be made. A MoMLAttribute decorates an object with a property that can describe itself with arbitrary MoML.

6.8 Acknowledgements

Many thanks to Ed Willink of Racal Research Ltd. and Simon North of Synopsys for many helpful suggestions, only some of which have made it into this version of MoML. Also, thanks to Tom Henzinger, Alberto Sangiovanni-Vincentelli, and Kees Vissers for helping clarify issues of abstract syntax.

Appendix B: Example

Figures 6.11 and 6.12 show a simple Ptolemy II model in the SDF domain. Figure 6.13 shows the execution window for this model. This model generates two sinusoidal waveforms and multiplies them together. This appendix gives the complete MoML code. The MoML code is divided into two files. The first of these defined a component, a sinewave generator. The second creates two instances of this sinewave generator and multiplies their outputs. The code listings are (hopefully) self-explanatory.

B.1 Sinewave Generator

The Sinewave component is defined in the file $PTII/ptolemy/actor/lib/Sinewave.xml, which is listed below. This file defines a MoML class, which can then be referenced by the class name ptolemy.actor.lib.Sinewave. The Vergil rendition of this model is shown in figure 6.11.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE class PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<class name="Sinewave" extends="ptolemy.actor.TypedCompositeActor">
    <property name="_createdBy" class="ptolemy.kernel.attributes.VersionAttribute" value="3.0-beta">
    </property>
    <doc>This composite actor generates a sine wave.</doc>
    <property name="samplingFrequency" class="ptolemy.data.expr.Parameter" value="8000.0">
        <doc>The sampling frequency, in the same units as the frequency.</doc>
    </property>
```

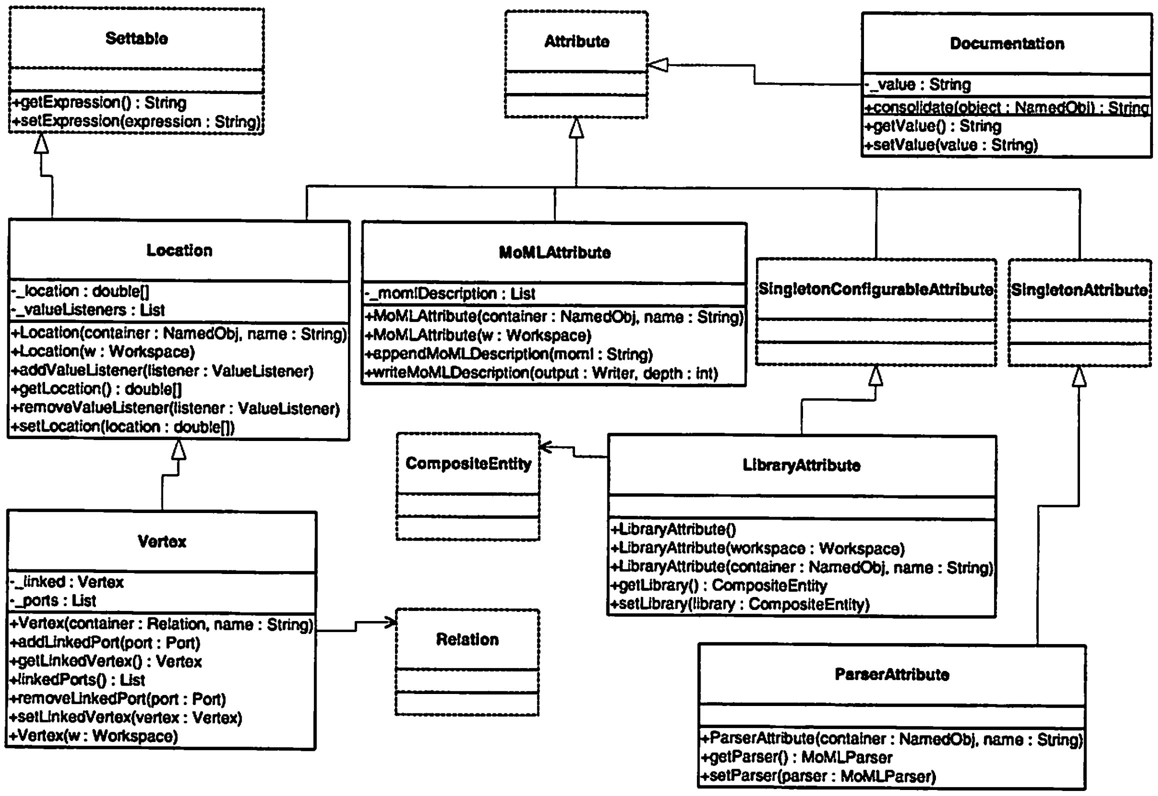

Settable
+getExpression() : String
+setExpression(expression : String)
Attribute
Documentation
-_value : String
+consolidate(object : NamedObj) : String
+getValue() : String
+setValue(value : String)
Location
-_location : double[]
-_valueListeners : List
+Location(container : NamedObj, name : String)
+Location(w : Workspace)
+addValueListener(listener : ValueListener)
+getLocation() : double[]
+removeValueListener(listener : ValueListener)
+setLocation(location : double[])
MoMLAttribute
-_momlDescription : List
+MoMLAttribute(container : NamedObj, name : String)
+MoMLAttribute(w : Workspace)
+appendMoMLDescription(moml : String)
+writeMoMLDescription(output : Writer, depth : int)
SingletonConfigurableAttribute
SingletonAttribute
CompositeEntity
LibraryAttribute
+LibraryAttribute()
+LibraryAttribute(workspace : Workspace)
+LibraryAttribute(container : NamedObj, name : String)
+getLibrary() : CompositeEntity
+setLibrary(library : CompositeEntity)
Vertex
-_linked : Vertex
-_ports : List
+Vertex(container : Relation, name : String)
+addLinkedPort(port : Port)
+getLinkedVertex() : Vertex
+linkedPorts() : List
+removeLinkedPort(port : Port)
+setLinkedVertex(vertex : Vertex)
+Vertex(w : Workspace)
Relation
ParserAttribute
+ParserAttribute(container : NamedObj, name : String)
+getParser() : MoMLParser
+setParser(parser : MoMLParser)

FIGURE 6.10. Attributes in the moml package.

```
    <property name="_vergilSize" class="ptolemy.actor.gui.SizeAttribute" value="[596, 450]">
    </property>
    <property name="_vergilLocation" class="ptolemy.actor.gui.LocationAttribute" value="[104, 102]">
    </property>
    <property name="annotation" class="ptolemy.kernel.util.Attribute">
        <property name="_hideName" class="ptolemy.kernel.util.SingletonAttribute">
        </property>
        <property name="_iconDescription" class="ptolemy.kernel.util.SingletonConfigurableAttribute">
            <configure><svg><text x="20" y="20" style="font-size:14; font-family:SansSerif;
fill:blue">Generate a sine wave.</text></svg></configure>
        </property>
        <property name="_smallIconDescription" class="ptolemy.kernel.util.SingletonConfigurableAt-
tribute">
            <configure>
      <svg>
        <text x="20" style="font-size:14; font-family:SansSerif; fill:blue" y="20">-A-</text>
      </svg>
    </configure>
        </property>
        <property name="_controllerFactory" class="ptolemy.vergil.basic.NodeControllerFactory">
        </property>
        <property name="_editorFactory" class="ptolemy.vergil.toolbox.AnnotationEditorFactory">
        </property>
        <property name="_location" class="ptolemy.kernel.util.Location" value="145.0, 25.0">
        </property>
    </property>
    <property name="SDF Director" class="ptolemy.domains.sdf.kernel.SDFDirector">
        <property name="Scheduler" class="ptolemy.domains.sdf.kernel.SDFScheduler">
        </property>
        <property name="allowDisconnectedGraphs" class="ptolemy.data.expr.Parameter" value="false">
        </property>
        <property name="iterations" class="ptolemy.data.expr.Parameter" value="0">
        </property>
        <property name="vectorizationFactor" class="ptolemy.data.expr.Parameter" value="1">
        </property>
        <property name="_location" class="ptolemy.kernel.util.Location" value="70.0, 45.0">
```

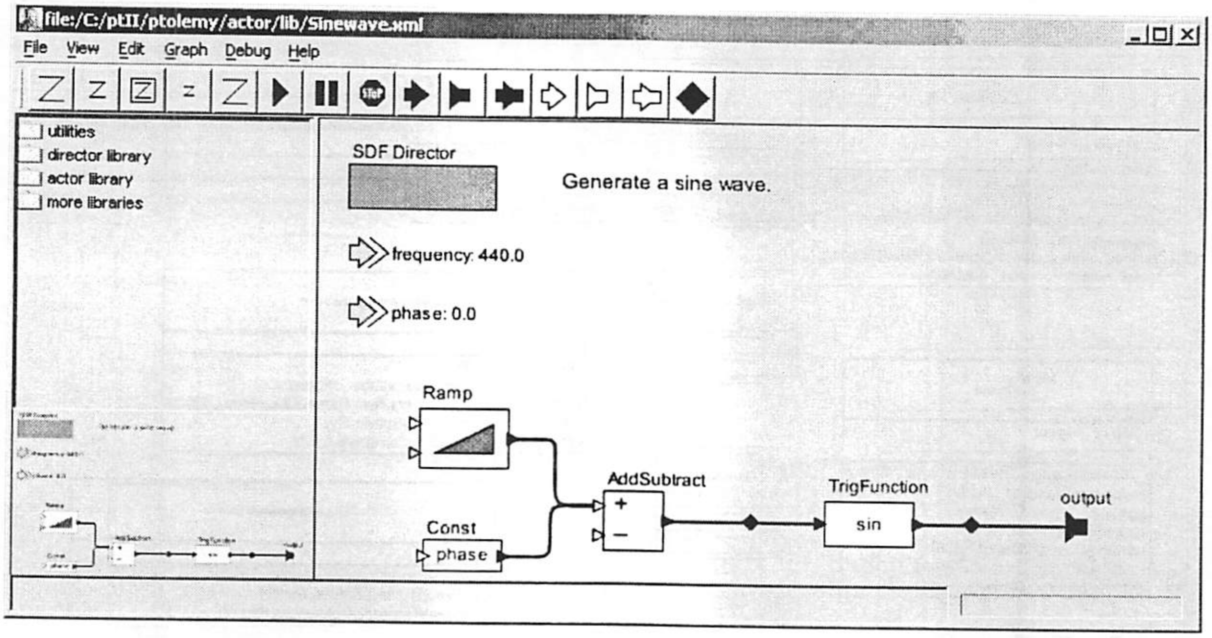


FIGURE 6.11. Rendition of the Sinewave class in Vergil 1.0.

```
            </property>
        </property>
        <property name="frequency" class="ptolemy.actor.parameters.PortParameter" value="440.0">
            <property name="_hideName" class="ptolemy.kernel.util.SingletonAttribute">
            </property>
            <property name="_iconDescription" class="ptolemy.kernel.util.SingletonConfigurableAttribute">
                <configure>
          <svg>
        <polyline points="-15,-15, -3,-5, -16,5" style="stroke:black"></polyline>
          </svg>
        </configure>
            </property>
            <property name="_icon" class="ptolemy.vergil.icon.ValueIcon">
            </property>
            <property name="_smallIconDescription" class="ptolemy.kernel.util.SingletonConfigurableAt-
tribute">
                <configure>
          <svg>
            <text x="20" style="font-size:14; font-family:SansSerif; fill:green" y="20">-P-</text>
          </svg>
        </configure>
            </property>
            <property name="_editorFactory" class="ptolemy.vergil.toolbox.VisibleParameterEditorFactory">
            </property>
            <property name="_location" class="ptolemy.kernel.util.Location" value="50.0, 95.0">
            </property>
        </property>
        <property name="phase" class="ptolemy.actor.parameters.PortParameter" value="0.0">
            <property name="_hideName" class="ptolemy.kernel.util.SingletonAttribute">
            </property>
            <property name="_iconDescription" class="ptolemy.kernel.util.SingletonConfigurableAttribute">
                <configure>
          <svg>
        <polyline points="-15,-15, -3,-5, -16,5" style="stroke:black"></polyline>
          </svg>
        </configure>
            </property>
            <property name="_icon" class="ptolemy.vergil.icon.ValueIcon">
            </property>
            <property name="_smallIconDescription" class="ptolemy.kernel.util.SingletonConfigurableAt-
tribute">
                <configure>
          <svg>
            <text x="20" style="font-size:14; font-family:SansSerif; fill:green" y="20">-P-</text>
          </svg>
        </configure>
            </property>
            <property name="_editorFactory" class="ptolemy.vergil.toolbox.VisibleParameterEditorFactory">
            </property>
            <property name="_location" class="ptolemy.kernel.util.Location" value="50.0, 135.0">
            </property>
        </property>
        <property name="_windowProperties" class="ptolemy.actor.gui.WindowPropertiesAttribute"
value="{bounds={108, 103, 811, 561}}">
        </property>
        <port name="frequency" class="ptolemy.actor.parameters.ParameterPort">
            <property name="input"/>
            <property name="_location" class="ptolemy.kernel.util.Location" value="30.0, 90.0">
            </property>
        </port>
        <port name="phase" class="ptolemy.actor.parameters.ParameterPort">
            <property name="input"/>
            <property name="_location" class="ptolemy.kernel.util.Location" value="30.0, 130.0">
            </property>
        </port>
        <port name="output" class="ptolemy.actor.TypedIOPort">
            <property name="output"/>
            <doc>Sinusoidal waveform output.</doc>
        <property name="_location" class="ptolemy.kernel.util.Location" value="515.0, 270.0">
        </property>
    </port>
    <entity name="Ramp" class="ptolemy.actor.lib.Ramp">
        <property name="firingCountLimit" class="ptolemy.data.expr.Parameter" value="0">
        </property>
        <property name="init" class="ptolemy.data.expr.Parameter" value="0">
        </property>
        <property name="step" class="ptolemy.actor.parameters.PortParameter" value="frequency*2*PI/sam-
plingFrequency">
        </property>
        <property name="_location" class="ptolemy.kernel.util.Location" value="100.0, 215.0">
        </property>
        <port name="output" class="ptolemy.actor.TypedIOPort">
            <property name="output"/>
        </port>
        <port name="trigger" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="multiport"/>
        </port>
        <port name="step" class="ptolemy.actor.parameters.ParameterPort">
            <property name="input"/>
        </port>
    </entity>
    <entity name="TrigFunction" class="ptolemy.actor.lib.TrigFunction">
        <property name="function" class="ptolemy.kernel.util.StringAttribute" value="sin">
            <property name="style" class="ptolemy.actor.gui.style.ChoiceStyle">
                <property name="acos" class="ptolemy.kernel.util.StringAttribute" value="acos">
                </property>
                <property name="asin" class="ptolemy.kernel.util.StringAttribute" value="asin">
                </property>
                <property name="atan" class="ptolemy.kernel.util.StringAttribute" value="atan">
                </property>
                <property name="cos" class="ptolemy.kernel.util.StringAttribute" value="cos">
                </property>
                <property name="sin" class="ptolemy.kernel.util.StringAttribute" value="sin">
                </property>
                <property name="tan" class="ptolemy.kernel.util.StringAttribute" value="tan">
                </property>
            </property>
        </property>
        <property name="_icon" class="ptolemy.vergil.icon.AttributeValueIcon">
            <property name="attributeName" class="ptolemy.kernel.util.StringAttribute" value="function">
            </property>
            <property name="displayWidth" class="ptolemy.data.expr.Parameter" value="6">
            </property>
        </property>
        <property name="_location" class="ptolemy.kernel.util.Location" value="375.0, 270.0">
        </property>
        <port name="input" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
        </port>
        <port name="output" class="ptolemy.actor.TypedIOPort">
            <property name="output"/>
        </port>
    </entity>
    <entity name="Const" class="ptolemy.actor.lib.Const">
        <property name="value" class="ptolemy.data.expr.Parameter" value="phase">
        </property>
        <doc>Create a constant sequence</doc>
        <property name="_icon" class="ptolemy.vergil.icon.BoxedValueIcon">
            <property name="attributeName" class="ptolemy.kernel.util.StringAttribute" value="value">
            </property>
            <property name="displayWidth" class="ptolemy.data.expr.Parameter" value="40">
            </property>
        </property>
        <property name="_location" class="ptolemy.kernel.util.Location" value="100.0, 295.0">
        </property>
        <port name="output" class="ptolemy.actor.TypedIOPort">
            <property name="output"/>
        </port>
        <port name="trigger" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="multiport"/>
        </port>
    </entity>
    <entity name="AddSubtract" class="ptolemy.actor.lib.AddSubtract">
        <property name="_location" class="ptolemy.kernel.util.Location" value="215.0, 270.0">
        </property>
        <port name="plus" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="multiport"/>
        </port>
        <port name="minus" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="multiport"/>
        </port>
        <port name="output" class="ptolemy.actor.TypedIOPort">
            <property name="output"/>
        </port>
    </entity>
    <relation name="relation3" class="ptolemy.actor.TypedIORelation">
        <vertex name="vertex1" value="445.0, 270.0">
        </vertex>
    </relation>
    <relation name="relation4" class="ptolemy.actor.TypedIORelation">
        <vertex name="vertex1" value="295.0, 270.0">
        </vertex>
    </relation>
    <relation name="relation" class="ptolemy.actor.TypedIORelation">
    </relation>
    <relation name="relation2" class="ptolemy.actor.TypedIORelation">
    </relation>
    <link port="output" relation="relation3"/>
    <link port="Ramp.output" relation="relation"/>
    <link port="TrigFunction.input" relation="relation4"/>
    <link port="TrigFunction.output" relation="relation3"/>
    <link port="Const.output" relation="relation2"/>
    <link port="AddSubtract.plus" relation="relation"/>
    <link port="AddSubtract.plus" relation="relation2"/>
    <link port="AddSubtract.output" relation="relation4"/>
</class>
```

B.2 Modulation

The top-level is defined in the file $PTII/ptolemy/moml/demo/modulation.xml, which is listed below. The Vergil rendition of this model is shown in figure 6.12, and its execution is shown in figure 6.13.

```
<?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="modulation" class="ptolemy.actor.TypedCompositeActor">
    <property name="_createdBy" class="ptolemy.kernel.attributes.VersionAttribute" value="3.0-beta">
    </property>
    <doc>Multiply a low-frequency sine wave (the signal)
by a higher frequency one (the carrier).</doc>
    <property name="frequency1" class="ptolemy.data.expr.Parameter" value="PI*0.2">
        <doc>Frequency of the carrier</doc>
    </property>
    <property name="frequency2" class="ptolemy.data.expr.Parameter" value="PI*0.02">
        <doc>Frequency of the sinusoidal signal</doc>
    </property>
    <property name="director" class="ptolemy.domains.sdf.kernel.SDFDirector">
        <property name="Scheduler" class="ptolemy.domains.sdf.kernel.SDFScheduler">
        </property>
        <property name="allowDisconnectedGraphs" class="ptolemy.data.expr.Parameter" value="false">
        </property>
        <property name="iterations" class="ptolemy.data.expr.Parameter" value="100">
            <doc>Number of iterations in an execution.</doc>
        </property>
        <property name="vectorizationFactor" class="ptolemy.data.expr.Parameter" value="1">
        </property>
        <property name="_location" class="ptolemy.kernel.util.Location" value="65.0, 35.0">
        </property>
    </property>
    <property name="_vergilSize" class="ptolemy.actor.gui.SizeAttribute" value="[430, 295]">
    </property>
    <property name="_vergilLocation" class="ptolemy.actor.gui.LocationAttribute" value="[175, 147]">
    </property>
    <property name="_windowProperties" class="ptolemy.actor.gui.WindowPropertiesAttribute"
value="{bounds={153, 24, 645, 411}}">
    </property>
    <entity name="carrier" class="ptolemy.actor.lib.Sinewave">
        <property name="_createdBy" class="ptolemy.kernel.attributes.VersionAttribute" value="2.1-devel-
2">
        </property>
```

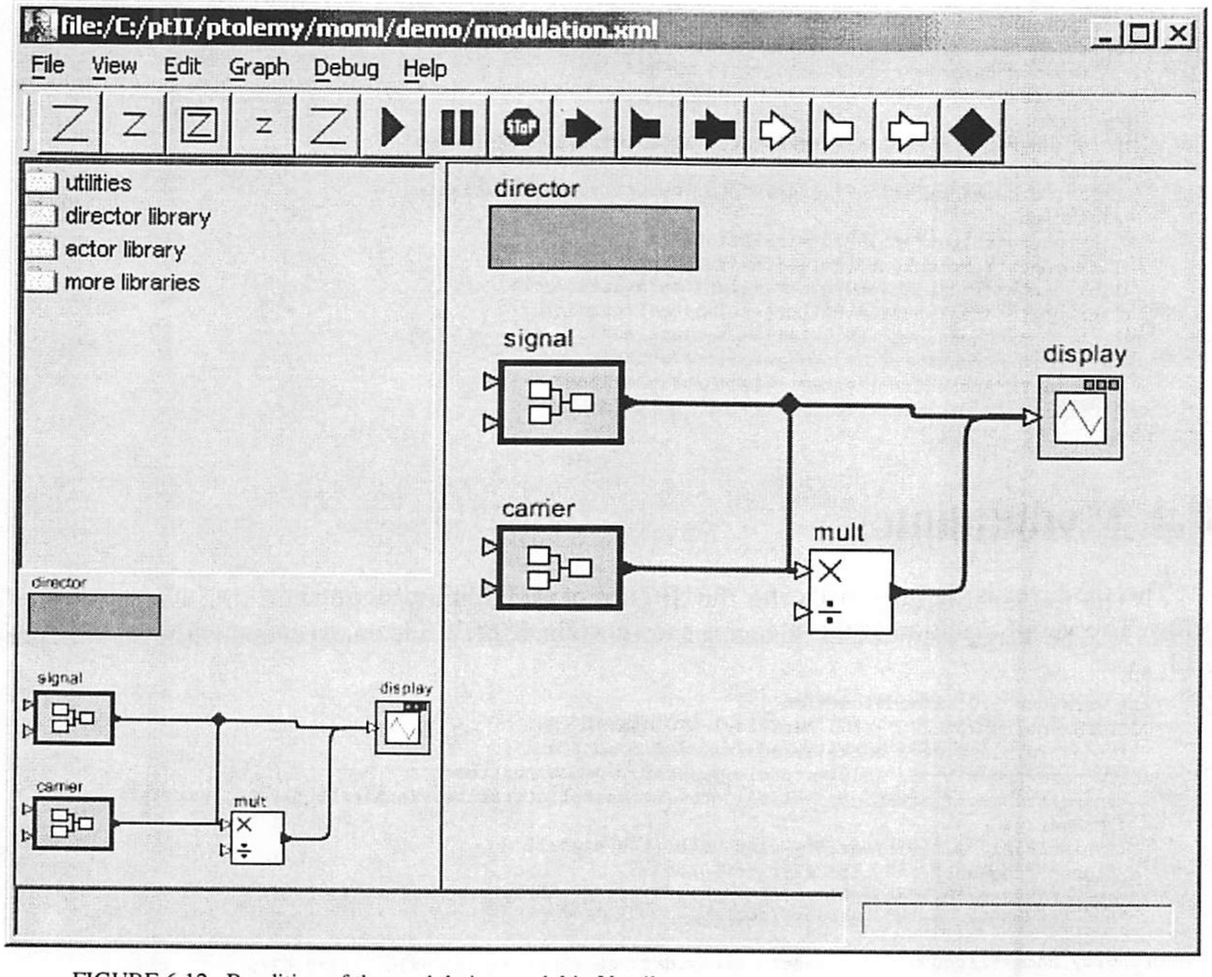
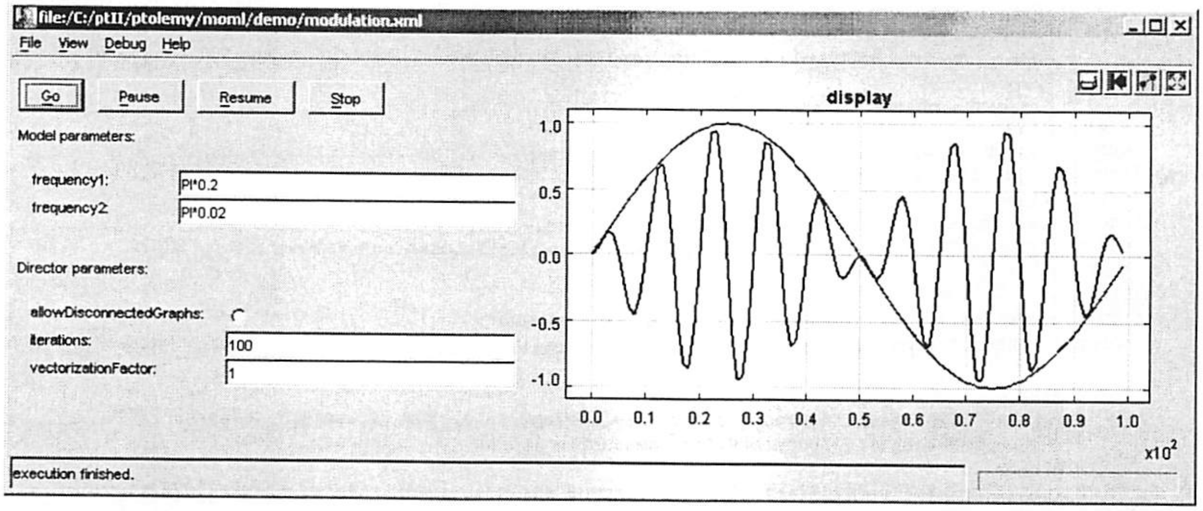


FIGURE 6.12. Rendition of the modulation model in Vergil.

```
        <property name="samplingFrequency" class="ptolemy.data.expr.Parameter" value="2*PI">
            <doc>The sampling frequency, in the same units as the frequency.</doc>
        </property>
        <property name="_vergilSize" class="ptolemy.actor.gui.SizeAttribute" value="[600, 450]">
        </property>
        <property name="_vergilLocation" class="ptolemy.actor.gui.LocationAttribute" value="[104, 102]">
        </property>
        <property name="annotation" class="ptolemy.kernel.util.Attribute">
            <property name="_hideName" class="ptolemy.kernel.util.SingletonAttribute">
            </property>
            <property name="_iconDescription" class="ptolemy.kernel.util.SingletonConfigurableAttribute">
                <configure><svg><text x="20" y="20" style="font-size:14; font-family:SansSerif;
fill:blue">Generate a sine wave.</text></svg></configure>
            </property>
            <property name="_smallIconDescription" class="ptolemy.kernel.util.SingletonConfigurableAt-
tribute">
                <configure>
      <svg>
        <text x="20" style="font-size:14; font-family:SansSerif; fill:blue" y="20">-A-</text>
      </svg>
    </configure>
            </property>
            <property name="_controllerFactory" class="ptolemy.vergil.basic.NodeControllerFactory">
            </property>
            <property name="_editorFactory" class="ptolemy.vergil.toolbox.AnnotationEditorFactory">
            </property>
            <property name="_location" class="ptolemy.kernel.util.Location" value="25.0, 20.0">
            </property>
        </property>
        <property name="SDF Director" class="ptolemy.domains.sdf.kernel.SDFDirector">
            <property name="Scheduler" class="ptolemy.domains.sdf.kernel.SDFScheduler">
            </property>
            <property name="allowDisconnectedGraphs" class="ptolemy.data.expr.Parameter" value="false">
            </property>
            <property name="iterations" class="ptolemy.data.expr.Parameter" value="0">
            </property>
            <property name="vectorizationFactor" class="ptolemy.data.expr.Parameter" value="1">
            </property>
            <property name="_location" class="ptolemy.kernel.util.Location" value="70.0, 45.0">
            </property>
        </property>
        <property name="frequency" class="ptolemy.actor.parameters.PortParameter" value="frequency1">
```



Model parameters:

frequency1: PI*0.2

frequency2: PI*0.02

Director parameters:

allowDisconnectedGraphs:

iterations: 100

vectorizationFactor: 1

execution finished.

FIGURE 6.13. Execution window for the modulation model.

```
            <property name="_hideName" class="ptolemy.kernel.util.SingletonAttribute">
            </property>
            <property name="_iconDescription" class="ptolemy.kernel.util.SingletonConfigurableAttribute">
                <configure>
        <svg>
    <polyline points="-15,-15, -3,-5, -16,5" style="stroke:black"></polyline>
        </svg>
     </configure>
            </property>
            <property name="_icon" class="ptolemy.vergil.icon.ValueIcon">
            </property>
            <property name="_smallIconDescription" class="ptolemy.kernel.util.SingletonConfigurableAt-
tribute">
                <configure>
        <svg>
          <text x="20" style="font-size:14; font-family:SansSerif; fill:green" y="20">-P-</text>
        </svg>
     </configure>
            </property>
            <property name="_editorFactory" class="ptolemy.vergil.toolbox.VisibleParameterEditorFactory">
            </property>
            <property name="_location" class="ptolemy.kernel.util.Location" value="50.0, 95.0">
            </property>
            <doc>The frequency of the sinusoid, in the same units as the sampling frequency.</doc>
        </property>
        <property name="phase" class="ptolemy.actor.parameters.PortParameter" value="0.0">
            <property name="_hideName" class="ptolemy.kernel.util.SingletonAttribute">
            </property>
            <property name="_iconDescription" class="ptolemy.kernel.util.SingletonConfigurableAttribute">
                <configure>
        <svg>
    <polyline points="-15,-15, -3,-5, -16,5" style="stroke:black"></polyline>
        </svg>
     </configure>
            </property>
            <property name="_icon" class="ptolemy.vergil.icon.ValueIcon">
            </property>
            <property name="_smallIconDescription" class="ptolemy.kernel.util.SingletonConfigurableAt-
tribute">
                <configure>
        <svg>
          <text x="20" style="font-size:14; font-family:SansSerif; fill:green" y="20">-P-</text>
        </svg>
     </configure>
            </property>
            <property name="_editorFactory" class="ptolemy.vergil.toolbox.VisibleParameterEditorFactory">
            </property>
            <property name="_location" class="ptolemy.kernel.util.Location" value="50.0, 135.0">
            </property>
            <doc>The phase, in radians.</doc>
        </property>
        <property name="_windowProperties" class="ptolemy.actor.gui.WindowPropertiesAttribute"
value="{bounds={108, 103, 811, 561}}">
        </property>
        <doc>This composite actor generates a sine wave.</doc>
        <property name="_location" class="ptolemy.kernel.util.Location" value="120.0, 230.0">
        </property>
    </entity>
    <entity name="signal" class="ptolemy.actor.lib.Sinewave">
        <property name="_createdBy" class="ptolemy.kernel.attributes.VersionAttribute" value="2.1-devel-
2">
        </property>
        <property name="samplingFrequency" class="ptolemy.data.expr.Parameter" value="2*PI">
            <doc>The sampling frequency, in the same units as the frequency.</doc>
        </property>
        <property name="_vergilSize" class="ptolemy.actor.gui.SizeAttribute" value="[600, 450]">
        </property>
        <property name="_vergilLocation" class="ptolemy.actor.gui.LocationAttribute" value="[104, 102]">
        </property>
        <property name="annotation" class="ptolemy.kernel.util.Attribute">
            <property name="_hideName" class="ptolemy.kernel.util.SingletonAttribute">
            </property>
            <property name="_iconDescription" class="ptolemy.kernel.util.SingletonConfigurableAttribute">
                <configure><svg><text x="20" y="20" style="font-size:14; font-family:SansSerif;
fill:blue">Generate a sine wave.</text></svg></configure>
            </property>
            <property name="_smallIconDescription" class="ptolemy.kernel.util.SingletonConfigurableAt-
tribute">
                <configure>
      <svg>
        <text x="20" style="font-size:14; font-family:SansSerif; fill:blue" y="20">-A-</text>
      </svg>
    </configure>
            </property>
            <property name="_controllerFactory" class="ptolemy.vergil.basic.NodeControllerFactory">
            </property>
            <property name="_editorFactory" class="ptolemy.vergil.toolbox.AnnotationEditorFactory">
            </property>
            <property name="_location" class="ptolemy.kernel.util.Location" value="25.0, 20.0">
            </property>
        </property>
        <property name="SDF Director" class="ptolemy.domains.sdf.kernel.SDFDirector">
            <property name="Scheduler" class="ptolemy.domains.sdf.kernel.SDFScheduler">
            </property>
            <property name="allowDisconnectedGraphs" class="ptolemy.data.expr.Parameter" value="false">
            </property>
            <property name="iterations" class="ptolemy.data.expr.Parameter" value="0">
            </property>
            <property name="vectorizationFactor" class="ptolemy.data.expr.Parameter" value="1">
            </property>
            <property name="_location" class="ptolemy.kernel.util.Location" value="70.0, 45.0">
            </property>
        </property>
        <property name="frequency" class="ptolemy.actor.parameters.PortParameter" value="frequency2">
            <property name="_hideName" class="ptolemy.kernel.util.SingletonAttribute">
            </property>
            <property name="_iconDescription" class="ptolemy.kernel.util.SingletonConfigurableAttribute">
                <configure>
      <svg>
    <polyline points="-15,-15, -3,-5, -16,5" style="stroke:black"></polyline>
      </svg>
    </configure>
            </property>
            <property name="_icon" class="ptolemy.vergil.icon.ValueIcon">
            </property>
            <property name="_smallIconDescription" class="ptolemy.kernel.util.SingletonConfigurableAt-
tribute">
                <configure>
      <svg>
        <text x="20" style="font-size:14; font-family:SansSerif; fill:green" y="20">-P-</text>
      </svg>
    </configure>
            </property>
            <property name="_editorFactory" class="ptolemy.vergil.toolbox.VisibleParameterEditorFactory">
            </property>
            <property name="_location" class="ptolemy.kernel.util.Location" value="50.0, 95.0">
            </property>
            <doc>The frequency of the sinusoid, in the same units as the sampling frequency.</doc>
        </property>
        <property name="phase" class="ptolemy.actor.parameters.PortParameter" value="0.0">
            <property name="_hideName" class="ptolemy.kernel.util.SingletonAttribute">
            </property>
            <property name="_iconDescription" class="ptolemy.kernel.util.SingletonConfigurableAttribute">
                <configure>
      <svg>
    <polyline points="-15,-15, -3,-5, -16,5" style="stroke:black"></polyline>
```

```
      </svg>
    </configure>
            </property>
            <property name="_icon" class="ptolemy.vergil.icon.ValueIcon">
            </property>
            <property name="_smallIconDescription" class="ptolemy.kernel.util.SingletonConfigurableAt-
tribute">
                <configure>
      <svg>
        <text x="20" style="font-size:14; font-family:SansSerif; fill:green" y="20">-P-</text>
      </svg>
    </configure>
            </property>
            <property name="_editorFactory" class="ptolemy.vergil.toolbox.VisibleParameterEditorFactory">
            </property>
            <property name="_location" class="ptolemy.kernel.util.Location" value="50.0, 135.0">
            </property>
            <doc>The phase, in radians.</doc>
        </property>
        <property name="_windowProperties" class="ptolemy.actor.gui.WindowPropertiesAttribute"
value="{bounds={108, 103, 811, 561}}">
        </property>
        <doc>This composite actor generates a sine wave.</doc>
        <property name="_location" class="ptolemy.kernel.util.Location" value="120.0, 95.0">
        </property>
    </entity>
    <entity name="mult" class="ptolemy.actor.lib.MultiplyDivide">
        <property name="_location" class="ptolemy.kernel.util.Location" value="260.0, 180.0">
        </property>
        <port name="multiply" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="multiport"/>
        </port>
        <port name="divide" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="multiport"/>
        </port>
        <port name="output" class="ptolemy.actor.TypedIOPort">
            <property name="output"/>
        </port>
    </entity>
    <entity name="display" class="ptolemy.actor.lib.gui.SequencePlotter">
        <property name="fillOnWrapup" class="ptolemy.data.expr.Parameter" value="true">
        </property>
        <property name="legend" class="ptolemy.kernel.util.StringAttribute">
        </property>
        <property name="_windowProperties" class="ptolemy.actor.gui.WindowPropertiesAttribute">
        </property>
        <property name="_plotSize" class="ptolemy.actor.gui.SizeAttribute">
        </property>
        <property name="startingDataset" class="ptolemy.data.expr.Parameter" value="0">
        </property>
        <property name="xInit" class="ptolemy.data.expr.Parameter" value="0.0">
        </property>
        <property name="xUnit" class="ptolemy.data.expr.Parameter" value="1.0">
        </property>
        <property name="_location" class="ptolemy.kernel.util.Location" value="385.0, 95.0">
        </property>
        <port name="input" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="multiport"/>
        </port>
    </entity>
    <relation name="r1" class="ptolemy.actor.TypedIORelation">
    </relation>
    <relation name="r2" class="ptolemy.actor.TypedIORelation">
        <vertex name="vertex0" value="195.0, 95.0">
        </vertex>
    </relation>
    <relation name="r3" class="ptolemy.actor.TypedIORelation">
    </relation>
    <link port="carrier.output" relation="r1"/>
    <link port="signal.output" relation="r2"/>
    <link port="mult.multiply" relation="r1"/>
    <link port="mult.multiply" relation="r2"/>
    <link port="mult.output" relation="r3"/>
    <link port="display.input" relation="r2"/>
    <link port="display.input" relation="r3"/>
</entity>
```

7

Custom Applets

Authors: *Edward A. Lee*
Christopher Hylands

7.1 Introduction

Ptolemy II models can be embedded in applets. In most cases, the MoMLApplet class can be used. For the MoMLApplet class, the model is given by a MoML file, which can be created using Vergil. The URL for the MoML file is given by the *modelURL* applet parameter in the HTML file.

Occasionally, however, it is useful to create an applet that exercises more control over the display and user interaction, or constructs or manipulates Ptolemy II models in ways that cannot be done in MoML. In such cases, the PtolemyApplet class can be useful. The MoMLApplet class is derived from PtolemyApplet, as shown in figure 7.1 (see appendix A of chapter 1 for UML syntax). Developers may either use PtolemyApplet directly or extend it to provide a more sophisticated user interface or a more elaborate method for model construction or manipulation.

The PtolemyApplet class provides four applet parameters:

- *background*: The background color, typically given as a hex number of the form "*#rrggbb*" where *rr* gives the red component, *gg* gives the green component, and *bb* gives the blue component.
- *controls*: This gives a comma-separated list of any subset of the words "buttons", "topParameters", and "directorParameters" (case insensitive), or the word "none". If this parameter is not given, then it is equivalent to giving "buttons", and only the control buttons mentioned above will be displayed. If the parameter is given, and its value is "none", then no controls are placed on the screen. If the word "topParameters" is included in the comma-separated list, then controls for the top-level parameters of the model are placed on the screen, below the buttons. If the word "directorParameters" is included, then controls for the director parameters are also included.
- *modelClass*: The fully qualified class name of a Java class that extends NamedObj. This class defines the model.
- *orientation*: This can have value "horizontal", "vertical", or "controls_only" (case insensitive). If it is "vertical", then the controls are placed above the visual elements of the Placeable actors. This

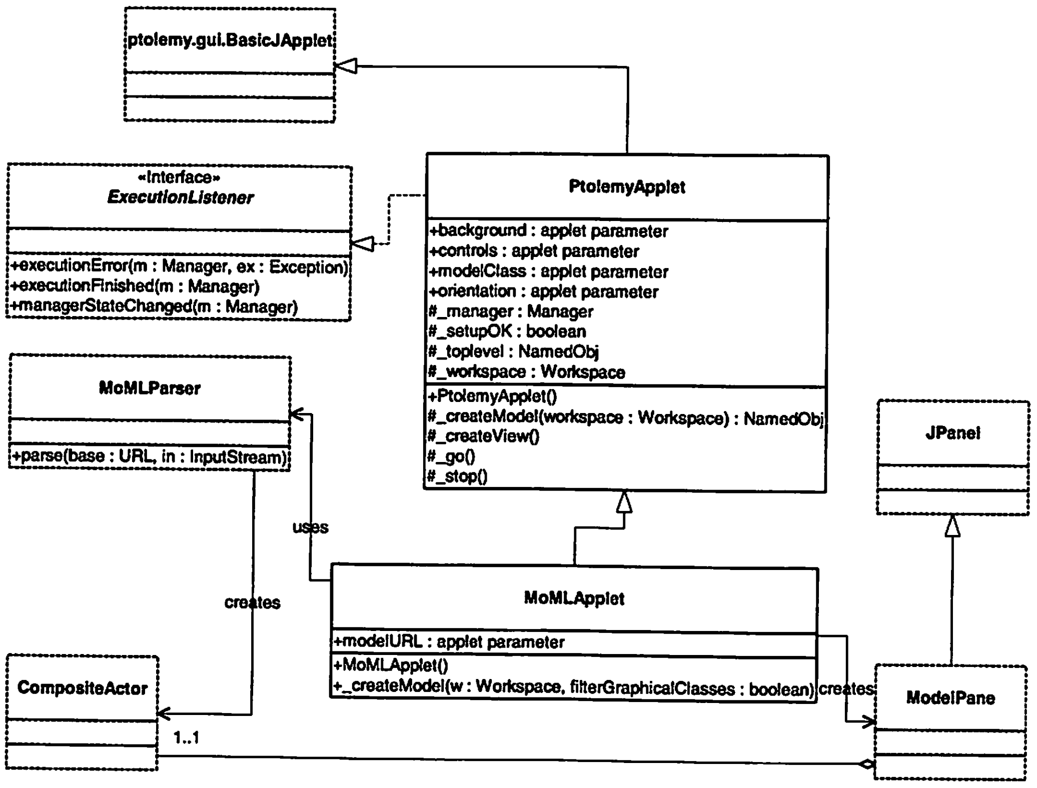

ptolemy.gui.BasicJApplet
«Interface»
ExecutionListener
+executionError(m : Manager, ex : Exception)
+executionFinished(m : Manager)
+managerStateChanged(m : Manager)
PtolemyApplet
+background : applet parameter
+controls : applet parameter
+modelClass : applet parameter
+orientation : applet parameter
#_manager : Manager
#_setupOK : boolean
#_toplevel : NamedObj
#_workspace : Workspace
+PtolemyApplet()
#_createModel(workspace : Workspace) : NamedObj
#_createView()
#_go()
#_stop()
MoMLParser
+parse(base : URL, in : InputStream)
uses
creates
JPanel
MoMLApplet
+modelURL : applet parameter
+MoMLApplet()
+_createModel(w : Workspace, filterGraphicalClasses : boolean)
creates
CompositeActor
1..1
ModelPane

FIGURE 7.1. UML static structure diagram for PtolemyApplet, a convenience class for constructing applets. PtolemyApplet is in the ptolemy.actor.gui package.

is the default. If it is "horizontal", then the controls are placed to the left of the visual elements. If it is "controls_only" then no visual elements are placed.

The use of these applet parameters is explained in more detail below.

7.2 HTML Files Containing Applets

An applet is a Java class that can be referenced by an HTML file and accessed either locally or over the web and run in a secure manner on the local machine in a web browser. Unfortunately, many browsers available today are shipped with an earlier version of Java that does not provide features that Ptolemy II requires. The work around is to use Sun's Java Plug-In, which invokes the 1.4 version of the Java Runtime Environment (JRE), instead of the default Java runtime that is shipped with the browser. The Java Plug-in is installed when the JRE or the Java Development Kit (JDK) is installed. Unfortunately, using the Java Plug-in makes the applet HTML more complex. There are two choices:

1. Use fairly complex JavaScript to determine which browser is running and then to properly select one of three different ways to invoke the Java Plug-in. This method works on the most different types of platforms and browsers. The JavaScript is so complex, that rather than reproduce it here, please see one of the demonstration html files such as $PTII/ptolemy/domains/sdf/demo/Butterfly/Butterfly.htm. Sun provides a free tool called HTMLConverter that will automatically generate the html code, see the Java Plug-in home page at http://java.sun.com/products/plugin/.

2. Use the much simpler <applet> ...</applet> tag to invoke the Java Plug-in. This method works on many platforms and browsers, but requires a more recent version of the Java Plug-in, and will not work under Netscape Communicator 4.x. However, all is not lost for Netscape Communicator 4.x users, since the `appletviewer` command that is included with the Java Developmen kit will display applets written using the simpler format.

   For details about the above two choices, see http://java.sun.com/products/plugin/versions.html.

Sample HTML for the <applet> . . . </applet> style of custom applet is shown in figure 7.2. An HTML file containing the segment shown in figure 7.2 can be found in $PTII/doc/tutorial/TutorialApplet1.htm, where $PTII is the home directory of the Ptolemy II installation. Also in that directory are a number of sample Java files for applets, each named TutorialApplet*n*.java, where *n* is an integer starting with 1. These files contain a series of applet definitions, each with increasing sophistication, that are discussed below. Each applet has a corresponding TutorialApplet*n*.htm file.

Since our example applets are in a directory $PTII/doc/tutorial, the codebase for the applet is "../.." in figure 7.2, which is the directory $PTII. This permits the applets to refer to any class in the Ptolemy II tree.

There are some parameters in the HTML in figure 7.2 that you may want to change. The width and the height, for example, specify the amount of space on the screen that the browser gives to the applet.

7.3 Defining a Model in a Java File

PtolemyApplet supports two techniques for instantiating models:

1. The model can be defined as a Java class that extends NamedObj, with the class name given by the *modelClass* applet parameter in the HTML file.
2. The model can be defined as a Java class that extends PtolemyApplet and overrides the protected method _createModel() to create the model, and optionally overrides the _createView() method to create the visual display for the model.

The first of these is simpler, so we begin by explaining this technique.

7.3.1 A Model Class as a Composite Actor

If the model is defined in a Java class that extends NamedObj, then we can use the *modelClass* applet parameter to pass the class name to PtolemyApplet and invoke the PtolemyApplet code from the applet. PtolemyApplet will then construct our model and provide the basic functionality we need.

```
<APPLET
 code = "ptolemy/actor/gui/PtolemyApplet"
 codebase = "../.."
 width = "800"
 height = "300"
 >
 <PARAM NAME = "modelClass" VALUE = "doc.tutorial.TutorialApplet1" > \
No Java Plug-in support for applet, see
 <a href="http://java.sun.com/products/plugin/"><code>http://java.sun.com/products/plugin/</code></a>
</APPLET>
```

FIGURE 7.2. An HTML segment that invokes the Java 1.4 Plug-in under both most browser, except Netscape 4.x. This text can be found in $PTII/doc/tutorial/TutorialApplet1.htm.

In figure 7.3 is a listing of an extremely simple applet that runs in the discrete-event (DE) domain. The first line declares that the applet is in a package called "doc.tutorial," which matches the directory name relative to the codebase specified in the HTML file. In the next several lines, the applet imports the following classes from Ptolemy II:

- `TypedCompositeActor`: Our model extends TypedCompositeActor, which itself eventually extends NamedObj. This is the typical top-level container class for models in most Ptolemy II domains.
- `PtolemyApplet`: This base class creates a top-level composite actor called _toplevel, a manager called _manager, and a workspace called _workspace (all protected members of the class, shown in figure 7.1). We will see shortly how to use these.
- `Clock`: This is an actor that generates a clock signal, which by default is a sequence of events placed one time unit apart and alternating in value between 1 and 0.
- `TimedPlotter`: This is an actor that plots functions of time.
- `DEDirector`: The discrete-event domain director that manages execution of the model.
- `IllegalActionException`: This exception thrown on an attempt to perform an action that would result in an inconsistent or contradictory data structure if it were allowed to complete.
- `NameDuplicationException`: This exception is thrown on an attempt to add a named object to a collection that requires unique names, and finding that there already is an object by that name in the collection.
- `Workspace`: An object for synchronization and version tracking of groups of objects.

Next, the construct:

```
package doc.tutorial;
import ptolemy.actor.TypedCompositeActor;
import ptolemy.actor.gui.PtolemyApplet;
import ptolemy.actor.lib.Clock;
import ptolemy.actor.lib.gui.TimedPlotter;
import ptolemy.domains.de.kernel.DEDirector;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;
import ptolemy.kernel.util.Workspace;

public class TutorialApplet1 extends TypedCompositeActor {
    public TutorialApplet1(Workspace workspace)
          throws IllegalActionException, NameDuplicationException {
        super(workspace);

        // Create the director.
        DEDirector director = new DEDirector(this, "director");
        setDirector(director);
        director.stopTime.setExpression("10.0");

        // Create two actors.
        Clock clock = new Clock(this,"clock");
        TimedPlotter plotter = new TimedPlotter(this,"plotter");

        // Connect them.
        connect(clock.output, plotter.input);
    }
}
```

FIGURE 7.3. An extremely simple applet that runs in the DE domain. This text can be found in $PTII/tutorial/TutorialApplet1.java.

```
public class TutorialApplet1 extends TypedCompositeActor {...}
```

defines a class called TutorialApplet1 that extends TypedCompositeActor. The new class provides a constructor that takes one argument, the Workspace into which to place the model:

```
public TutorialApplet1(Workspace workspace)
        throws IllegalActionException, NameDuplicationException {...}
```

The body of the constructor first invokes the constructor in the base class with:

```
super(workspace);
```

It then creates a DE director.

```
DEDirector director = new DEDirector(this, "director");
```

The director implements the discrete-event model of computation, which controls when the component actors are invoked and how they communicate. The next line tells the model to use the director:

```
setDirector(director);
```

The next line sets a director parameter that controls the duration of an execution of the model:

```
director.stopTime.setExpression("10.0");
```

If we don't set the stop time, then the model will run forever, or until the user hits the stop button. The next few lines create an instance of Clock and an instance of TimedPlotter, and connect them together:

```
// Create two actors.
Clock clock = new Clock(this,"clock");
TimedPlotter plotter = new TimedPlotter(this,"plotter");

// Connect them.
connect(clock.output, plotter.input);
```

The constructors for Clock and TimedPlotter take two arguments, the container (a composite actor), and an arbitrary name (which must be unique within the container). This example uses the variable `this`, which refers to the class we are creating, a TypedCompositeActor, as a container. The connection is accomplished by the connect() method of the composite actor, which takes two ports as arguments. Instances of Clock have one output port, `output`, which is a public member, and instances of TimedPlotter have one input port, `input`, which is also a public member.

7.3.2 Compiling

To compile this class definition, you must tell the Java compiler where to find the Ptolemy classes by using the -classpath command line argument. For example, in bash or a similar shell, assuming the environment variable PTII is set to the location of the Ptolemy II installation:

```
bash-2.05b$ cd $PTII/doc/tutorial
bash-2.05b$ javac -classpath ../.. TutorialApplet1.java
```

(The part before the "$" is the prompt issued by bash). Java requires that classes are defined in files that have the same name as the class. The Ptolemy II style convention is to extend this notion and have HTML files have the same name as the model they use, so the HTML file that runs the model in TutorialApplet1.java is named TutorialApplet1.htm.

You should now be able to run the applet with the command:

```
bash-2.05b$ appletviewer TutorialApplet1.htm
```

The result of running the applet is a new window which should look like that shown in figure 7.4. The following applet parameters are useful to customize the display:

- *controls*: This gives a comma-separated list of any subset of the words "buttons", "topParameters", and "directorParameters" (case insensitive), or the word "none". If this parameter is not given, then it is equivalent to giving "buttons", and only the control buttons mentioned above will be displayed. If the parameter is given, and its value is "none", then no controls are placed on the screen. If the word "topParameters" is included in the comma-separated list, then controls for the top-level parameters of the model are placed on the screen, below the buttons. If the word "directorParameters" is included, then controls for the director parameters are also included.
- *orientation*: This can have value "horizontal", "vertical", or "controls_only" (case insensitive). If it is "vertical", then the controls are placed above the visual elements of the Placeable actors. This is the default. If it is "horizontal", then the controls are placed to the left of the visual elements. If it is "controls_only" then no visual elements are placed.

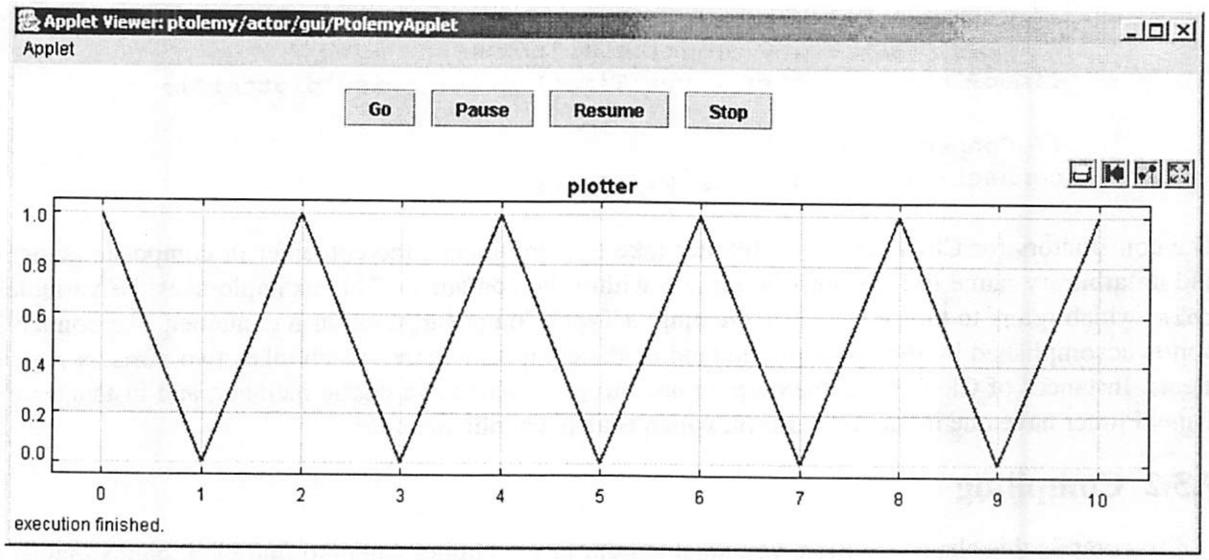


FIGURE 7.4. Result of running the (all too simple) applet of figure 7.3.

For example, if the HTML includes the following lines within the APPLET element:

```
<PARAM NAME="controls" VALUE="buttons, directorParameters">
<PARAM NAME="orientation" VALUE="horizontal">
```

then the result of execution looks like figure 7.5. The layout is now horizontal, with the controls to the left of the displays instead of on top, and the director parameters have been made available to the applet user.

7.3.3 Executing the Model in an Application

A model created as above can also be executed as an application, in addition to running it as a mode. Any class that extends CompositeActor, the base class for TypedCompositeActor, can be executed using the CompositeActorApplication class, shown in figure 7.6. The command is simply:

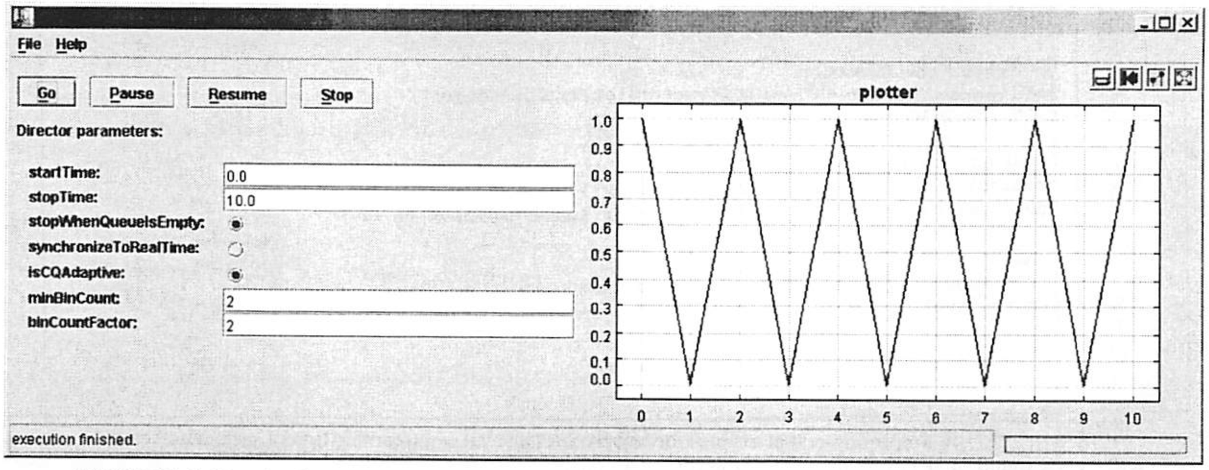
```
bash-2.05b$ cd $PTII/doc/tutorial
bash-2.05b$ java -classpath ../.. \
  ptolemy.actor.gui.CompositeActorApplication \
  -class doc.tutorial.TutorialApplet1
```

The result will look like figure 7.5. This ability to use the same class definition in both an applet and an application is convenient.

7.3.4 Extending PtolemyApplet

Another way to use PtolemyApplet is to define the model as a Java class that extends it and overrides the protected method _createModel() to create a model and optionally overrides the _createView() protected method to create a custom display. Extending PtolemyApplet gives the developer the opportunity to control the look and feel of the applet in as much detail as necessary, including creating completely customized displays and controls.

In figure 7.7 we define the same applet by extending PtolemyApplet instead of extending Typed-



Director parameters:

startTime: 0.0
stopTime: 10.0
stopWhenQueueIsEmpty: ●
synchronizeToRealTime: ○
isCQAdaptive: ●
minBinCount: 2
binCountFactor: 2

execution finished.

FIGURE 7.5. Result of the model as an application.

CompositeActor. This class overrides the _createView() method, which takes a Workspace as an argument and returns a NamedObj. Note that since we are no longer extending TypedCompositeActor, we need to instantiate a TypedCompositeActor named toplevel and use it where we used "this" in the previous example. Otherwise, the code is very similar to that in figure 7.3.

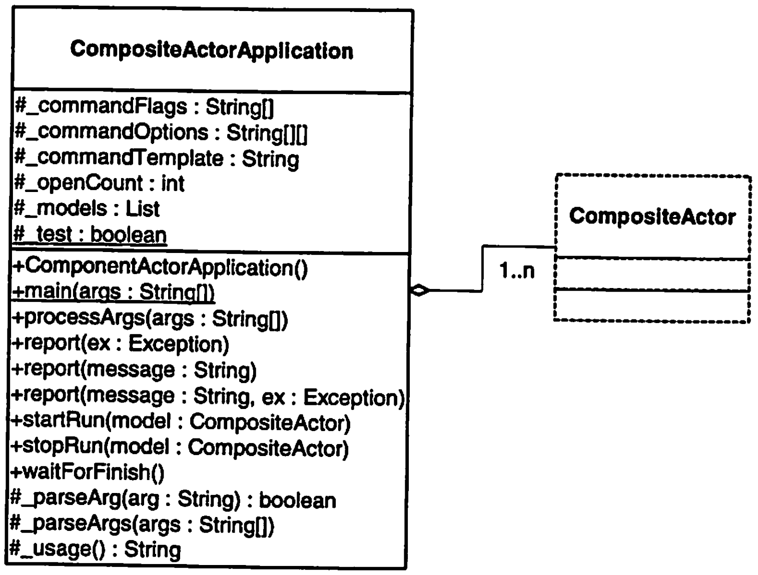

CompositeActorApplication
#_commandFlags : String[]
#_commandOptions : String[][]
#_commandTemplate : String
#_openCount : int
#_models : List
#_test : boolean
+ComponentActorApplication()
+main(args : String[])
+processArgs(args : String[])
+report(ex : Exception)
+report(message : String)
+report(message : String, ex : Exception)
+startRun(model : CompositeActor)
+stopRun(model : CompositeActor)
+waitForFinish()
#_parseArg(arg : String) : boolean
#_parseArgs(args : String[])
#_usage() : String
1..n
CompositeActor

FIGURE 7.6. Any class that extends CompositeActor can be executed using the CompositeActorApplication class.

```
package doc.tutorial;
import ptolemy.actor.TypedCompositeActor;
import ptolemy.actor.gui.PtolemyApplet;
import ptolemy.actor.lib.Clock;
import ptolemy.actor.lib.gui.TimedPlotter;
import ptolemy.domains.de.kernel.DEDirector;
import ptolemy.kernel.util.NamedObj;
import ptolemy.kernel.util.Workspace;

public class TutorialApplet2 extends PtolemyApplet {
    public NamedObj _createModel(Workspace workspace)
          throws Exception {
        TypedCompositeActor toplevel = new TypedCompositeActor(workspace);

        // Create the director.
        DEDirector director = new DEDirector(toplevel, "director");
        director.stopTime.setExpression("10.0");

        // Create two actors.
        Clock clock = new Clock(toplevel,"clock");
        TimedPlotter plotter = new TimedPlotter(toplevel,"plotter");

        // Connect them.
        toplevel.connect(clock.output, plotter.input);
        return toplevel;
    }
}
```

FIGURE 7.7. A simple applet that extends PtolemyApplet instead of extending TypedCompositeActor. This text can be found in $PTII/doc/tutorial/TutorialApplet2.java.

We can improve this applet by giving the user more specialized control over its execution.

7.3.5 Using Model Parameters

Typically, a model has a set of parameters that you wish for the user to be able to control in the applet. Suppose for example that in the above applet you wish for the user to be able to control the stop time of the director and the period of the clock actor. You can modify the Java code in figure 7.3 as shown in figure 7.8. This code uses the Parameter class to define two top-level parameters. The following lines create the top-level parameters:

```
Parameter stopTime = new Parameter(this, "stopTime");
Parameter clockPeriod = new Parameter(this, "clockPeriod");
```

The default values of these two parameters are set by the following lines:

```
package doc.tutorial;
import ptolemy.actor.TypedCompositeActor;
import ptolemy.actor.gui.PtolemyApplet;
import ptolemy.actor.lib.Clock;
import ptolemy.actor.lib.gui.TimedPlotter;
import ptolemy.data.expr.Parameter;
import ptolemy.domains.de.kernel.DEDirector;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;
import ptolemy.kernel.util.Workspace;

public class TutorialApplet3 extends TypedCompositeActor {
    public TutorialApplet3(Workspace workspace)
         throws IllegalActionException, NameDuplicationException {
      super(workspace);

      // Create model parameters
      Parameter stopTime = new Parameter(this, "stopTime");
      Parameter clockPeriod = new Parameter(this, "clockPeriod");

      // Give the model parameters default values.
      stopTime.setExpression("10.0");
      clockPeriod.setExpression("2.0");

      // Create the director
      DEDirector director = new DEDirector(this, "director");
      setDirector(director);

      // Create two actors.
      Clock clock = new Clock(this,"clock");
      TimedPlotter plotter = new TimedPlotter(this,"plotter");

      // Set the user controlled parameters.
      director.stopTime.setExpression("stopTime");
      clock.period.setExpression("clockPeriod");

      // Connect the actors.
      connect(clock.output, plotter.input);
    }
}
```

FIGURE 7.8. Code that adds model parameters control to the applet. This code can be found in $PTII/doc/tutorial/TutorialApplet3.java.

```
        stopTime.setExpression("10.0");
        clockPeriod.setExpression("2.0");
```

Finally, the values of the director and Clock actor parameters are coupled to these top-level parameters by the lines

```
        director.stopTime.setExpression("stopTime");
        clock.period.setExpression("clockPeriod");
```

The expressions being set here can be much more elaborate. The expression language is documented in the Data Package chapter. Here, the expressions each contain a single variable reference, referring to the top-level parameters by name.

In order for the top-level parameters to appear in the controls of an applet, we must configure the HTML file as shown in figure 7.9. The line

```
    <PARAM NAME="controls" VALUE="buttons, topParameters">
```

accomplish the objective. The result of invoking the appletviewer on the HTML file in figure 7.9 is shown in figure 7.10.

7.3.6 Adding Custom Actors

The intent of Ptolemy II is to have a reasonably rich set of actors in the actor libraries. However, it is anticipated that model builders will often need to define their own, custom actors. This is relatively easy to do, as discussed in the Designing Actors chapter. By convention, when a specialized actor is created for a particular applet or application, we store that actor in the same directory with the applet or application, rather than in the actor libraries. The actor libraries are for generic, reusable actors.

7.3.7 Using Jar Files

A jar file is a Java Archive File that contains multiple .class files. Applets that are being downloaded over the net will start up more quickly if all the relevant Java .class files are collected together into one or more jar files. This dramatically reduces the number of HTTP transactions.

Models in the Ptolemy II demo directories typically use three separate jar files:

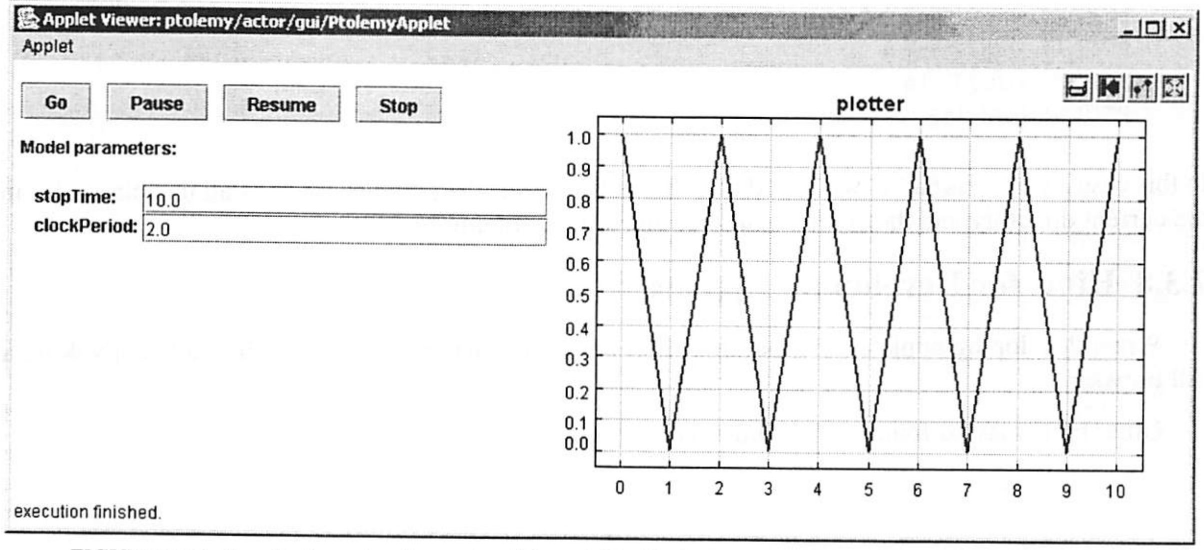
```
<APPLET
 code = "ptolemy/actor/gui/PtolemyApplet"
 codebase = "../.."
 width = "800"
 height = "300"
 >
 <PARAM NAME = "modelClass" VALUE = "doc.tutorial.TutorialApplet3" > \
 <PARAM NAME = "controls" VALUE = "buttons, topParameters" > \
 <PARAM NAME = "orientation" VALUE = "horizontal" > \

No Java Plug-in support for applet, see
 <a href="http://java.sun.com/products/plugin/"><code>http://java.sun.com/products/plugin/</code></a>
</APPLET>
```

FIGURE 7.9. The HTML that displays model parameters for the applet user to control. This file can be found in $PTII/doc/tutorial/TutorialApplet3.htm

- ptolemy/ptsupport.jar — A jar file containing classes from ptolemy.kernel, ptolemy.actor and other packages, see $PTII/ptolemy/makefile for a complete list;
- ptolemy/domains/*domain*/*domain*.jar — A domain specific jar file such as de.jar, where *domain* is replaced by a domain name;
- ptolemy/domains/*domain*/demo/*Demo*/*Demo*.jar — A model-specific jar file. Models with sophisticated GUIs that use Listeners can result in multiple .class files per .java file, so having a jar file can help download speeds.

The third jar file is not needed if the model resides in a single .class file. To use jar files, you must modify the HTML shown in figure 7.2 to read as shown in figure 7.11.

An important downside of using jar files is that during Java development, one must regenerate the jar files each time a Java file is recompiled. If you are developing an applet, you may want to avoid using jar files, or only include jar files that are from packages that are not actively being developed.



Model parameters:

stopTime: 10.0
clockPeriod: 2.0

execution finished.

FIGURE 7.10. Result of running the applet of figure 7.8 with horizontal layout, and including the top-level parameters.

```
<APPLET
 code = "ptolemy/actor/gui/PtolemyApplet"
 codebase = "../.."
 width = "800"
 height = "300"
 archive="ptolemy/ptsupport.jar, ptolemy/domains/de/de.jar"
 >
 <PARAM NAME = "modelClass" VALUE = "doc.tutorial.TutorialApplet3" >
 <PARAM NAME = "controls" VALUE = "buttons, topParameters" >
 <PARAM NAME = "orientation" VALUE = "horizontal" >
No Java Plug-in support for applet, see
 <a href="http://java.sun.com/products/plugin/"><code>http://java.sun.com/products/plugin/</code></a>
</APPLET>
```

FIGURE 7.11. An HTML segment that modifies that of figure 7.2 to use jar files. This text can be found in $PTII/doc/tutorial/tutorialApplet4.htm.

How Jar files are built. To know which jar files in the Ptolemy II tree you might need for your applet, you need to know how the jar files are constructed. The short story is that every package has a jar file that includes subpackages. Since the package structure mirrors the directory structure, it is easy to peruse the Ptolemy II tree (rooted at $PTII) and look for jar files. There are a few exceptions; for example, domain jar files, such as de.jar, do not include the demos, even though the demos are in a subpackage of the domain package.

The longer story is that the `make install` rule in Ptolemy II makefiles builds various jar files that contain the Ptolemy II .class files. In general, make install builds a jar file in each directory that contains more than one .class file. If a directory contains subdirectories that in turn contain jar files, then the subdirectory jar files are expanded and included in the upper level jar file. For example, the $PTII/ptolemy/kernel/makefile contains:

```
# Used to build jar files
PTPACKAGE = ptolemy.kernel
PTDIST = $(PTPACKAGE)$(PTVERSION)
PTCLASSJAR =
# Include the .class files from these jars in PTCLASSALLJAR
PTCLASSALLJARS = \
  util/util.jar
PTCLASSALLJAR = kernel.jar
```

In this case `make install` will build a jar file called `kernel.jar` that contains all the .class files in the current directory and the contents of $PTII/ptolemy/kernel/util/util.jar.

7.3.8 Hints for Developing Applets

When developing applets, you may find it easier to test using appletviewer instead of invoking a full browser.

Other hints may be found in $PTII/doc/coding/applets.htm

References

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] G. Agha, "Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems," in *Formal Methods for Open Object-based Distributed Systems*, IFIP Transactions, E. Najm and J.-B. Stefani, Eds., Chapman & Hall, 1997.

[3] G. Agha, "Concurrent object-oriented programming," *Communications of the ACM*, 33(9):125–140, Sept. 1990.

[4] G. Agha, S. Frolund, W. Kim, R. Panwar, A. Patterson, and D. Sturman, "Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3–14, May 1993.

[5] G. Agha, I. A. Mason, S. F.Smith, and C. L. Talcott, "A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[6] R. Allen and D. Garlan, "Formalizing Architectural Connection," in *Proc. of the 16th International Conference on Software Engineering* (ICSE 94), May 1994, pp. 71-80, IEEE Computer Society Press.

[7] G. R. Andrews, *Concurrent Programming — Principles and Practice*, Addison-Wesley, 1991.

[8] R. L. Bagrodia, "Parallel Languages for Discrete Event Simulation Models," *IEEE Computational Science & Engineering*, vol. 5, no. 2, April-June 1998, pp 27-38.

[9] R. Bagrodia, R. Meyer, *et al.*, "Parsec: A Parallel Simulation Environment for Complex Systems," *IEEE Computer*, vol. 31, no. 10, October 1998, pp 77-85.

[10] M. von der Beeck, "A Comparison of Statecharts Variants," in *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, LNCS 863, pp. 128-148, Springer-Verlag, 1994.

[11] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1270-1282.

[12] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Tr. on Automatic Control*, Vol. 35, No. 5, pp. 525-546, May 1990.

[13] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, 19(2):87-152, 1992.

[14] S. Bhatt, R. M. Fujimoto, A. Ogielski, and K. Perumalla, "Parallel Simulation Techniques for Large-Scale Networks," *IEEE Communications Magazine*, Vol. 36, No. 8, August 1998, pp. 42-47.

[15] S. S. Bhattacharyya, P. K. Murthy and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwell, Mass, 1996.

[16] J. Bier, E. Goei, W. Ho, P. Lapsley, M. O'Reilly, G. Sih and E. A. Lee, "Gabriel: A Design Environment for DSP," *IEEE Micro Magazine*, October 1990, vol. 10, no. 5, pp. 28-45.

[17] Randy Brown, "CalendarQueue: A Fast Priority Queue Implementation for The Simulation Event Set Problem", Communications of the ACM, October 1998, Volume 31, Number 10.

[18] V. Bryant, "Metric Spaces," Cambridge University Press, 1985.

[19] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int. Journal of Computer Simulation*, special issue on "Simulation Software Development," vol. 4, pp. 155-182, April, 1994. (http://ptolemy.eecs.berkeley.edu/publications/papers/94/JEurSim).

[20] A. Burns, *Programming in OCCAM 2*, Addison-Wesley, 1988.

[21] James C. Candy, "A Use of Limit Cycle Oscillations to Obtain Robust Analog-to-Digital Converters," *IEEE Tr. on Communications*, Vol. COM-22, No. 3, pp. 298-305, March 1974.

[22] L. Cardelli, *Type Systems*, Handbook of Computer Science and Engineering, CRC Press, 1997.

[23] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A Declarative Language for Programming Synchronous Systems," *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, Munich, Germany, January, 1987.

[24] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, no. 11, November 1981, pp. 198-205.

[25] B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[26] John Davis II, "Order and Containment in Concurrent System Design," **Ph.D. thesis**, Memorandum UCB/ERL M00/47, Electronics Research Laboratory, University of California, Berkeley, September 8, 2000.(http://ptolemy.eecs.berkeley.edu/publications/papers/00/concsys/)

[27] S. A. Edwards and E. A. Lee, "The Semantics and Execution of a Synchronous Block-Diagram Language," **to appear** in *Science of Computer Programming*, 2003.

[28] S. A. Edwards, "The Specification and Execution of Heterogeneous Synchronous Reactive Systems," **Ph.D. thesis**, University of California, Berkeley, May 1997. Available as UCB/ERL M97/31. (http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/)

[29] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, "Taming Heterogeneity-the Ptolemy Approach," *Proceedings of the IEEE*, V. 91, No 1, January 2003.

[30] P. H. J. van Eijk, C. A. Vissers, M. Diaz, *The formal description technique LOTOS*, Elsevier Science, B.V., 1989. (http://wwwtios.cs.utwente.nl/lotos)

[31] P. A. Fishwick, *Simulation Model Design and Execution: Building Digital Worlds*, Prentice Hall, 1995.

[32] C. Fong, "Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II," Master's Report, Memorandum UCB/ERL M01/9, Electronics Research Laboratory, University of California, Berkeley, January 2001.(http://ptolemy.eecs.berkeley.edu/publications/papers/00/dt/)

[33] M. Fowler and K. Scott, *UML Distilled*, Addison-Wesley, 1997.

[34] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, no. 10, October 1990, pp 30-53.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading MA, 1995.

[36] C. W. Gear, "Numerical Initial Value Problems in Ordinary Differential Equations," Prentice Hall Inc. 1971.

[37] A. J. C. van Gemund, *"Performance Prediction of Parallel Processing Systems: The PAMELA Methodology,"* Proc. 7th Int. Conf. on Supercomputing, pages 418-327, Tokyo, July 1993.

[38] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," April 13, 1998 (revised from Memorandum UCB/ERL M97/57, Electronics Research Laboratory, University of California, Berkeley, CA 94720, August 1997). (http://ptolemy.eecs.berkeley.edu/publications/papers/98/starcharts)

[39] M. Goel, *Process Networks in Ptolemy II*, MS Report, ERL Technical Report UCB/ERL No. M98/69, University of California, Berkeley, CA 94720, December 16, 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/PNinPtolemyII)

[40] M. Grand, *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML*, John Wiley & Sons, 1998.

[41] C. Hansen, "Hardware logic simulation by compilation," In *Proceedings of the Design Automation Conference* (DAC). SIGDA, ACM, 1988.

[42] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.*, vol 8, pp. 231-274, 1987.

[43] P. G Harrison, "A Higher-Order Approach to Parallel Algorithms," *The Computer Journal*, Vol. 35, No. 6, 1992.

[44] T. A. Henzinger, B. Horowitz and C. M. Kirsch, "Giotto: A Time-Triggered Language for Embedded Programming," EMSOFT 2001. Tahoe City, CA, Springer-Verlag,

[45] T. A. Henzinger, "The theory of hybrid automata," in *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1996, pp. 278-292, invited tutorial.

[46] T.A. Henzinger, and O. Kupferman, and S. Qadeer, "From *pre*historic to *post*modern symbolic model checking," in *CAV 98: Computer-aided Verification*, pp. 195-206, eds. A.J. Hu and M.Y. Vardi, Lecture Notes in Computer Science 1427, Springer-Verlag, 1998.

[47] T. A. Henzinger and C. M. Kirsch, "The Embedded Machine: Predictable, portable real-time code," In *Proceedings of Conference on Programming Language Design and Implementation* (PLDI). SIGPLAN, ACM, June 2002.

[48] C. Hewitt, "Viewing control structures as patterns of passing messages," *Journal of Artificial Intelligence*, 8(3):323–363, June 1977.

[49] M. G. Hinchey and S. A. Jarvis, *Concurrent Systems: Formal Developments in CSP*, McGraw-Hill, 1995.

[50] C. W. Ho, A. E. Ruehli, and P. A. Brennan, "The Modified Nodal Approach to Network Analysis," IEEE Tran. on Circuits and Systems, Vol. CAS-22, No. 6, 1975, pp. 504-509.

[51] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[52] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[53] IEEE DASC 1076.1 Working Group, "VHDL-A Design Objective Document, version 2.3," http://www.vhdl.org/analog/ftp_files/requirements/DOD_v2.3.txt

[54] D. Jefferson, Brian Beckman, et al, "Distributed Simulation and the Time Warp Operating System," UCLA Computer Science Department: 870042, 1987.

[55] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.

[56] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[57] E. Kohler, *The Click Modular Router*, Ph.D. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 2001.

[58] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.

[59] P. Laramie, R.S. Stevens, and M.Wan, "Kahn process networks in Java," ee290n class project report, Univ. of California at Berkeley, 1996.

[60] D. Lea, *Concurrent Programming in Java*^{TM}, Addison-Wesley, Reading, MA, 1997.

[61] B. Lee and E. A. Lee, "Interaction of Finite State Machines with Concurrency Models," *Proc. of Thirty Second Annual Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/InteractionFSM/)

[62] B. Lee and E. A. Lee, "Hierarchical Concurrent Finite State Machines in Ptolemy," *Proc. of International Conference on Application of Concurrency to System Design*, p. 34-40, Fukushima, Japan, March 1998 (http://ptolemy.eecs.berkeley.edu/publications/papers/98/HCFSMinPtolemy/)

[63] E. A. Lee, S. Neuendorffer and M. J. Wirthlin, "Actor-Oriented Design of Embedded Hardware and Software Systems," **invited paper, to appear in** *Journal of Circuits, Systems, and Computers*, 2003.

[64] E. A. Lee, "Embedded Software," in *Advances in Computers* (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

[65] E. A. Lee and T. M. Parks, "Dataflow Process Networks," in *Readings in Hardware/Software Co-Design*, G. De Micheli, R. Ernst, and W. Wolf, eds., Morgan Kaufmann, San Francisco, 2002 (reprinted from 70).

[66] E. A. Lee, "What's Ahead for Embedded Software?" *IEEE Computer*, September 2000, pp. 18-26.

[67] E. A. Lee, "Modeling Concurrent Real-time Processes Using Discrete Events," Invited paper to *Annals of Software Engineering*, Special Volume on Real-Time Software Engineering, Volume 7, 1999, pp 25-45. Also UCB/ERL Memorandum M98/7, March 4th 1998.(http://ptolemy.eecs.berkeley.edu/publications/papers/98/realtime)

[68] E. A. Lee and Y. Xiong, "System-Level Types for Component-Based Design," *First Workshop on Embedded Software*, EMSOFT 2001, Lake Tahoe, CA, USA, Oct. 8-10, 2001. (also Technical Memorandum UCB/ERL M00/8, Electronics Research Lab, University of California, Berkeley, CA 94720, USA, February 29, 2000. http://ptolemy.eecs.berkeley.edu/publications/papers/01/systemLevelType/).

[69] E. A. Lee, "Computing for Embedded Systems," **invited paper**, *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 21-23, 2001.

[70] E. A. Lee and T. M. Parks, "Dataflow Process Networks,", *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May, 1995. (http://ptolemy.eecs.berkeley.edu/publications/papers/95/processNets)

[71] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation,", *IEEE Transactions on CAD*, Vol 17, No. 12, December 1998 (Revised from ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997). (http://ptolemy.eecs.berkeley.edu/publications/papers/97/denotational/)

[72] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, January, 1987.

[73] M. A. Lemkin, *Micro Accelerometer Design with Digital Feedback Control*, Ph.D. dissertation, University of California, Berkeley, Fall 1997.

[74] S. Y. Liao, S. Tjiang, and R. Gupta, "An efficient implementation of reactivity for modeling hardware in the Scenic design environment," In *Proceedings of the 34th Design Automation Conference* (DAC'1997). SIGDA, ACM, 1997.

[75] J. Liu, J. Eker, J. W. Janneck and E. A. Lee, "Realistic Simulations of Embedded Control Systems," *International Federation of Automatic Control, 15th IFAC World Congress*, Barcelona, Spain, July 21-26, 2002.

[76] J. Liu, X. Liu, and E. A. Lee, "Modeling Distributed Hybrid Systems in Ptolemy II," **invited embedded tutorial** in *American Control Conference*, Arlington, VA, June 25-27, 2001.

[77] J. Liu, S. Jefferson, and E. A. Lee, "Motivating Hierarchical Run-Time Models in Measurement and Control Systems," *American Control Conference*, Arlington, VA, pp. 3457-3462, June 25-27, 2001.

[78] J. Liu and E. A. Lee, "A Component-Based Approach to Modeling and Simulating Mixed-Signal and Hybrid Systems," **to appear** in *ACM Trans. on Modeling and Computer Simulation*, special issue on computer automated multi-paradigm modeling, 2003.

[79] J. Liu and E. A. Lee, "On the Causality of Mixed-Signal and Hybrid Models," *6th International Workshop on Hybrid Systems: Computation and Control* (HSCC '03), April 3-5, Prague, Czech Republic, 2003.

[80] J. Liu, "Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems," **Ph.D. thesis**, Technical Memorandum UCB/ERL M01/41, University of California, Berkeley, CA 94720, December 20th, 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/responsibleFrameworks/)

[81] J. Liu, *Continuous Time and Mixed-Signal Simulation in Ptolemy II*, MS Report, UCB/ERL Memorandum M98/74, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/MixedSignalinPtII/)

[82] J. Liu and E. A. Lee, "Component-based Hierarchical Modeling of Systems with Continuous and Discrete Dynamics," *Proc. of the 2000 IEEE International Conference on Control Applications and IEEE Symposium on Computer-Aided Control System Design* (CCA/CACSD'00), Anchorage, AK, September 25-27, 2000. pp. 95-100

[83] J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S. Sastry, and E. A. Lee, "A Hierarchical Hybrid System and Its Simulation", 1999 38th IEEE Conference on Decision and Control (CDC'99), Phoenix, Arizona.

[84] X. Liu, J. Liu, J. Eker, and E. A. Lee, "Heterogeneous Modeling and Design of Control Systems," **to appear** in *Software-Enabled Control: Information Technology for Dynamical Systems*, T. Samad and G. Balas (eds.), New York City: IEEE Press, 2003.

[85] D. C. Luckham and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, 21(9), pp. 717-734, September, 1995.

[86] F. Maraninchi, "The Argos Language: Graphical Representation of Automata and Description of Reactive Systems," in *Proc. of the IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.

[87] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 1993.

[88] K. Mehlhorn and Stefan Naher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1997.

[89] B. Meyer, *Object Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.

[90] R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[91] R. Milner, *"A Calculus of Communicating Systems"*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[92] R. Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences 17, pp. 384-375, 1978.

[93] J. Misra, "Distributed Discrete-Event Simulation," *Computing Surveys*, vol. 18, no. 1, March 1986, pp. 39-65.

[94] L. Muliadi, "Discrete Event Modeling in Ptolemy II," MS Report, Dept. of EECS, University of California, Berkeley, CA 94720, May 1999. (http://ptolemy.eecs.berkeley.edu/publications/papers/99/deModeling/)

[95] P. K. Murthy and E. A. Lee, "Multidimensional Synchronous Dataflow," *IEEE Transactions on Signal Processing*, volume 50, no. 8, pp. 2064 -2079, August 2002.

[96] L. W. Nagal, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," ERL Memo No. ERL-M520, Electronics Research Laboratory, University of California, Berkeley, CA 94720.

[97] NASA Office of Safety and Mission Assurance, *Software Formal Inspections Guidebook*, August 1993 (http://satc.gsfc.nasa.gov/fi/gdb/fitext.txt).

[98] S. Neuendorffer, "Automatic Specialization of Actor-Oriented Models in Ptolemy II," Master's Report, Technical Memorandum UCB/ERL M02/41, University of California, Berkeley, CA 94720, December 25, 2002.(http://ptolemy.eecs.berkeley.edu/papers/02/actorSpecialization)

[99] A. R. Newton and A. L. Sangiovanni-Vincentelli, "Relaxation-Based Electrical Simulation," *IEEE Tr. on Electronic Devices*, Vol. ed-30, No. 9, Sept. 1983.

[100]S. Oaks and H. Wong, *Java Threads*, O'Reilly, 1997.

[101]OMG, *Unified Modeling Language: Superstructure*, version 2.0, 3rd revised submission to RFP ad/00-09-02, April 10, 2003

[102]J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.

[103]J. K. Ousterhout, *Scripting: Higher Level Programming for the 21 Century*, IEEE Computer magazine, March 1998.

[104]T. M. Parks, *Bounded Scheduling of Process Networks*, Technical Report UCB/ERL-95-105. **Ph.D. Dissertation**. EECS Department, University of California. Berkeley, CA 94720, December 1995. (http://ptolemy.eecs.berkeley.edu/publications/papers/95/parksThesis/)

[105]J. K. Peacock, J. W. Wong and E. G. Manning, "Distributed Simulation Using a Network of Processors," *Computer Networks*, vol. 3, no. 1, February 1979, pp. 44-56.

[106]Rational Software Corporation, *UML Notation Guide*, Version 1.1, September 1997, http://www.rational.com/

[107]J. Reekie, S. Neuendorffer, C. Hylands and E. A. Lee, "Software Practice in the Ptolemy Project," Technical Report Series, GSRC-TR-1999-01, Gigascale Silicon Research Center, University of California, Berkeley, CA 94720, April 1999.(http://ptolemy.eecs.berkeley.edu/publications/papers/99/sftwareprac/)

[108]J. Rehof and T. Mogensen, "Tractable Constraints in Finite Semilattices," *Third International Static Analysis Symposium*, pp. 285-301, Volume 1145 of Lecture Notes in Computer Science, Springer, Sept., 1996.

[109]C. Rettig, "Automatic Units Tracking," *Embedded System Programming*, March, 2001.

[110]A. J. Riel, *Object Oriented Design Heuristics*, Addison Wesley, 1996.

[111]R. C. Rosenberg and D.C. Karnopp, *Introduction to Physical System Dynamics*, McGraw-Hill, NY, 1983.

[112]J. Rowson and A. Sangiovanni-Vincentelli, "Interface Based Design," *Proc. of DAC '97.*

[113]J. Rumbaugh, et al. *Object-Oriented Modeling and Design* Prentice Hall, 1991.

[114]J. Rumbaugh, *OMT Insights*, SIGS Books, 1996.

[115]S. Saracco, J. R. W. Smith, and R. Reed, *Telecommunications Systems Engineering Using SDL*, North-Holland - Elsevier, 1989.

[116]B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, NY 1994.

[117]N. Smyth, *Communicating Sequential Processes Domain in Ptolemy II*, MS Report, UCB/ERL Memorandum M98/70, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/CSPinPtolemyII/)

[118]J. Teich, E. Zitzler, and S. Bhattacharyya, "3D exploration of software schedules for DSP algorithms," In *Proceedings of International Symposium on Hardware/Software Codesign* (CODES). SIGDA, ACM, May 1999.

[119]J. Tsay, "A Code Generation Framework for Ptolemy II," ERL Technical Report UCB/ERL No. M00/25, Dept. EECS, University of California, Berkeley, CA 94720, May 19, 2000. (http://ptolemy.eecs.berkeley.edu/publications/papers/00/codegen).

[120]J. Tsay, C. Hylands and E. A. Lee, "A Code Generation Framework for Java Component-Based Designs," *CASES '00*, November 17-19, 2000, San Jose, CA.

[121]P. Whitaker, "The Simulation of Synchronous Reactive Systems In Ptolemy II," Master's Report, Memorandum UCB/ERL M01/20, Electronics Research Laboratory, University of California, Berkeley, May 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/sr/)

[122]World Wide Web Consortium, *XML 1.0 Recommendation*, October 2000, http://www.w3.org/XML/

[123]World Wide Web Consortium, *Overview of SGML Resources*, August 2000, http://www.w3.org/MarkUp/SGML/

[124]Y. Xiong and E. A. Lee, "An Extensible Type System for Component-Based Design," *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. LNCS 1785.

[125]Y. Xiong, "An Extensible Type System for Component-Based Design," **Ph.D. thesis**, Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1, 2002. (http://ptolemy.eecs.berkeley.edu/papers/02/typeSystem).

Glossary

abstract syntax	A conceptual data organization. cf. *concrete syntax.*
action methods	The methods initialize(), prefire(), fire(), postfire(), and wrapup() in the Executable interface.
actor	An executable entity. This was called a *block* in Ptolemy Classic.
anytype	The Ptolemy Classic name for *undeclared type*.
applet	A Java program that is downloaded from a web server by a browser and executed in the client's computer (usually within a plug-in for the browser). An applet has restricted access to local resources for security reasons. cf. application.
application	A Java program that is executed as an ordinary program on a host computer. Unlike an applet, an application can have full access to local resources such as the file system. cf. applet.
atomic actor	A primitive actor. That is, one that is not a composite actor. This was called a *star* in Ptolemy Classic.
attribute	A named property associated with a named object in Ptolemy II. Also, in XML, a modifier to an element.
block	The Ptolemy Classic name for an *actor.*
browser	A program that renders HTML and accesses the worldwide web using the HTTP protocol.
channel	A path from an output port to an input port (via relations) that can transport a single stream of tokens.
clustered graph	A graph with hierarchy. Ptolemy II topologies are clustered graphs.
code generation	Translation of a model into efficient, standalone software for execution autonomously from the design environment. Code generation was a major emphasis of Ptolemy Classic.
composite actor	An actor that is internally composed of other actors and relations. This was called a *galaxy* in Ptolemy Classic.
concrete syntax	A persistent representation of a data organization. cf. *abstract syntax.*
connection	A path from one port to another via relations and possibly transparent ports. A connection consists of one or more *relations* and two or more *links.*
container	An object that logically owns another. A Ptolemy II object can have at most one container.
dangling relation	A relation with only input ports or only output ports linked to it.
data polymorphism	Ability to operate with more than one token type.
deep traversals	Traversals of a clustered graph that see through transparent cluster boundaries (transparent composite entities and ports).

disconnected port	A port with no relation linked to it.
director	An object that controls the execution of a model or an opaque composite entity according to some *model of computation.*
domain	An implementation of a model of computation in Ptolemy II and Ptolemy Classic.
domain polymorphism	Ability to operate under more than one model of computation.
element	In XML, a portion of a document consisting of a begin tag, a body, and an end tag.
entity	A node in a Ptolemy II clustered graph. Also, in XML, a named text segment.
event	In the DE domain, an event is a token with a time stamp.
execution	One invocation of initialize(), followed by any number of *iterations*, followed by one invocation of wrapup().
executive director	From the perspective of an actor inside an opaque composite actor, the director of the container of the opaque composite actor.
galaxy	The Ptolemy Classic name for a *composite actor.*
immutable property	A property of an object that is set up when the object is constructed and that cannot be changed during the lifetime of the object.
iteration	One invocation of prefire(), followed by any number of invocations of fire(), followed by one invocation of postfire().
link	An association between a port and a relation.
manager	The top-level controller for the execution of a model.
model	A complete Ptolemy II application. This was called a *universe* in Ptolemy Classic.
model of computation	The rules that govern the interaction, communication, and control flow of a set of components.
MoML	Modeling markup language, an XML dialect for specifying component-based designs such those in Ptolemy II.
multiport	A port that can send or receive tokens over more than one channel.
opaque	For a composite entity or a port, an attribute that indicates that the inside should not be visible from the outside. That is, deep traversals of the topology do not see through an opaque boundary.
opaque composite actor	A composite actor with a local director. Such an actor appears to the outside domain to be atomic, but internally is composed of an interconnection of other actors. This was called a *wormhole* in Ptolemy Classic.
package	A collection of classes that forms a logical unit and occupies one directory in the source code tree.
parameter	An *attribute* with a value. This was called a *state* in Ptolemy Classic.
particle	The Ptolemy Classic name for a *token.*
port	A named interface of an entity to which connections be made.
Ptolemy Classic	A C++ software system for construction of concurrent models and implementation through code generation.

Ptolemy II	A Java software system for construction and execution of concurrent models.
Ptolemy Project	A research project at Berkeley that investigates modeling, simulation, and design of concurrent, networked, embedded systems.
relation	An object representing an interconnection between entities.
resolved type	A type for a port that is consistent with the type constraints of the actor and any port it is connected to. It is the result of type resolution.
servlet	A Java program that is executed on a web server and that produces results viewed remotely on a web browser.
star	The Ptolemy Classic name for an *atomic actor.*
state	The Ptolemy Classic name for a *parameter.*
subpackage	A package that is logically related to a parent package and occupies a subdirectory within the parent package in the source code tree.
tag	In XML, a portion of markup having the syntax *<tagname>*.
token	A unit of data that is communicated by actors. This was called a *particle* in Ptolemy Classic.
topology	The structure of interconnections between entities (via relations) in a Ptolemy II model. See *clustered graph.*
transparent	For an entity or port, not opaque. That is, deep traversals of the topology pass right through its boundaries.
transparent composite actor	A composite actor with no local director.
transparent port	The port of a transparent composite entity. Deep traversals of the topology see right through such a port.
type constraints	The declared constraints on the token types that an actor can work with.
type resolution	The process of reconciling type constraints prior to running a model.
undeclared type	Capable of working with any type of token. This was called *anytype* in Ptolemy Classic.
universe	The Ptolemy Classic name for a *model.*
width of a port	The sum of the widths of the relations linked to it, or zero if there are none.
width of a relation	The number of channels supported by the relation.
wormhole	The Ptolemy Classic name for an *opaque composite actor.*

Index