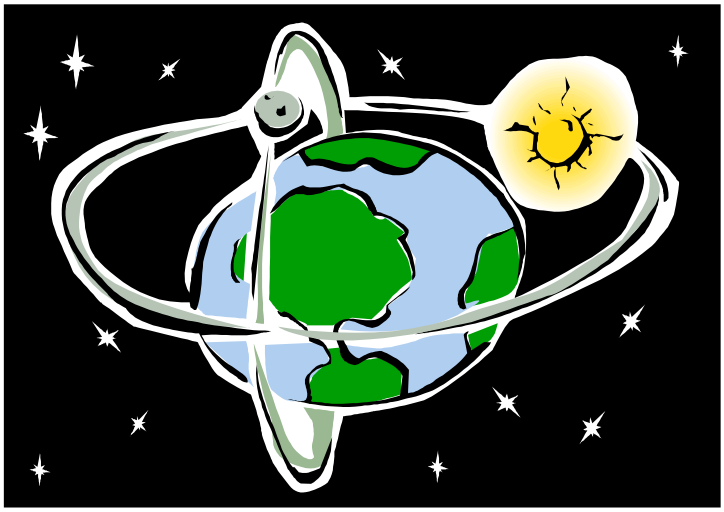
-PTOLEMY II -



HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

John Davis, II
Mudit Goel
Christopher Hylands
Bart Kienhuis
Edward A. Lee
Jie Liu
Xiaojun Liu
Lukito Muliadi
Steve Neuendorffer
John Reekie
Neil Smyth
Jeff Tsay
Yuhong Xiong

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

Memorandum UCB/ERL M99/40
Document Version 0.3.1
July 19, 1999

This project is supported by the Defense Advanced Research Projects Agency (DARPA), the Microelectronics Advanced Research Corporation (MARCO) , the State of California MICRO program, and the following companies: Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, Motorola, NEC, and Philips.

Contents

Part 1: Using Ptolemy II

1. Introduction 1-1

1.1. Modeling and Design 1-1
1.2. Architecture Design 1-3
1.3. Models of Computation 1-4
1.3.1. Communicating Sequential Processes - CSP 1-4
1.3.2. Continuous Time - CT 1-4
1.3.3. Discrete-Events - DE 1-5
1.3.4. Distributed Discrete Events - DDE 1-6
1.3.5. Discrete Time - DT 1-6
1.3.6. Finite-State Machines - FSM 1-6
1.3.7. Process Networks - PN 1-7
1.3.8. Synchronous Dataflow - SDF 1-7
1.3.9. Synchronous/Reactive - SR 1-7
1.4. Choosing Models of Computation 1-8
1.5. Visual Syntaxes 1-8
1.6. Ptolemy II 1-9
1.6.1. Package Structure 1-9
1.6.2. Overview of Key Classes 1-12
1.6.3. Domains 1-13
1.6.4. Capabilities 1-13
1.6.5. Future Capabilities 1-15
Appendix: UML — Unified Modeling Language 1-17
Package Diagrams 1-17
Static Structure Diagrams 1-17
Appendix: Ptolemy II Naming Conventions 1-20
Classes 1-20
Members 1-20
Methods 1-20

2. Building Models 2-1

2.1. Introduction 2-1
2.2. Applets 2-1
2.2.1. HTML Files Containing Applets 2-2
2.2.2. Creating Models 2-4
2.2.3. Compiling 2-5
2.2.4. Reporting Errors 2-6
2.2.5. Graphical Elements 2-6
2.2.6. Controlling Execution Time 2-7
2.2.7. Controlling Model Parameters 2-9

2.2.8. Adding Custom Actors 2-13
2.2.9. Using Jar Files 2-13
2.2.10. Script for Creating Demo Applets 2-15
2.2.11. Hints for Developing Applets 2-16
Appendix: Inspection Paradox Example 2-17
Description of the Problem 2-17
Observations 2-17
Code Listing 2-18
3. Actor Libraries 3-1
3.1. Overview 3-1
3.2. Library Organization 3-1
3.2.1. Actor.Lib 3-2
3.2.2. Actor.GUI 3-3
3.3. Data Polymorphism 3-3
3.4. Domain Polymorphism 3-5
3.4.1. Iterations 3-6
3.4.2. Domains with Fixed Point Semantics 3-6
3.4.3. Actors with State 3-7
3.5. Descriptions of Libraries 3-7
3.5.1. Functional Actors 3-9
3.5.2. Polymorphic Sources 3-10
3.5.3. Polymorphic Sinks and Displays 3-12
3.5.4. Expression Actor 3-13
3.5.5. Other Actors 3-14
4. Designing Actors 4-1
4.1. Overview 4-1
4.2. Anatomy of an Actor 4-2
4.2.1. Ports 4-2
4.2.2. Parameters 4-6
4.2.3. Constructors 4-8
4.2.4. Cloning 4-9
4.3. Action Methods 4-11
4.3.1. Initialize 4-11
4.3.2. Prefire 4-12
4.3.3. Fire 4-13
4.3.4. Postfire 4-15
4.3.5. Wrapup 4-15
4.4. Time 4-15
4.5. Code Format 4-16
4.5.1. Indentation 4-16
4.5.2. Spaces 4-17
4.5.3. Comments 4-17
4.5.4. Names 4-17
4.5.5. Exceptions 4-17
4.5.6. Javadoc 4-18
4.5.7. Code Organization 4-19

Part 2: Software Architecture

5. The Kernel 5-1

5.1. Abstract Syntax 5-1

5.2. Non-Hierarchical Topologies 5-2

5.2.1. Links 5-2

5.2.2. Consistency 5-3

5.3. Support Classes 5-5

5.3.1. Containers 5-5

5.3.2. Name and Full Name 5-5

5.3.3. Workspace 5-5

5.3.4. Attributes 5-6

5.3.5. List Classes 5-6

5.4. Clustered Graphs 5-6

5.4.1. Abstraction 5-8

5.4.2. Level-Crossing Connections 5-9

5.4.3. Tunneling Entities 5-9

5.4.4. Description 5-11

5.4.5. Cloning 5-11

5.4.6. An Elaborate Example 5-11

5.5. Opaque Composite Entities 5-12

5.6. Concurrency 5-13

5.6.1. Limitations of Monitors 5-15

5.6.2. Read and Write Access Permissions for Workspace 5-17

5.6.3. Making a Workspace Read Only 5-19

5.7. Mutations 5-19

5.7.1. Change Requests 5-19

5.7.2. Managers and Listeners 5-21

5.8. Exceptions 5-21

5.8.1. Base Class 5-21

5.8.2. Less Severe Exceptions 5-21

5.8.3. More Severe Exceptions 5-22

6. Actor Package 6-1

6.1. Concurrent Computation 6-1

6.2. Message Passing 6-2

6.2.1. Data Transport 6-2

6.2.2. Example 6-5

6.2.3. Transparent Ports 6-6

6.2.4. Data Transfer in Various Models of Computation 6-8

6.2.5. Discussion of the Data Transfer Mechanism 6-11

6.3. Execution 6-11

6.3.1. Director 6-13

6.3.2. Manager 6-17

6.3.3. ExecutionListener 6-18

6.3.4. Mutations 6-18

6.3.5. Opaque Composite Actors 6-18

6.3.6. Scheduler and Process Support 6-20

7. Data Package 7-1
7.1. Introduction 7-1
7.2. Data Encapsulation 7-1
7.3. Polymorphism 7-3
7.3.1. Polymorphic Arithmetic Operators 7-3
7.3.2. Lossless Type Conversion 7-4
7.3.3. Limitations 7-6
7.4. Variables and Parameters 7-6
7.4.1. Values 7-8
7.4.2. Types 7-8
7.4.3. Dependencies 7-10
7.5. Expressions 7-10
7.5.1. The Ptolemy II Expression Language 7-10
7.5.2. Functions 7-12
7.5.3. Limitations 7-12
Appendix: Expression Evaluation 7-13
Generating the parse tree 7-13
Evaluating the parse tree 7-14
8. Graph Package 8-1
8.1. Introduction 8-1
8.2. Classes and Interfaces in the Graph Package 8-2
8.2.1. Graph 8-3
8.2.2. Directed Graphs 8-3
8.2.3. Directed Acyclic Graphs and CPO 8-4
8.2.4. Inequality Terms, Inequalities, and the Inequality Solver 8-4
8.3. Example Use 8-5
8.3.1. Generating A Schedule for A Composite Actor 8-5
8.3.2. Forming and Solving Constraints over a CPO 8-6
9. Type System 9-1
9.1. Introduction 9-1
9.2. Formulation 9-3
9.2.1. Type Constraints 9-3
9.2.2. Run-time Type Checking and Lossless Type Conversion 9-5
9.3. Implementation Classes 9-6
9.3.1. Static Type Checking and Type Resolution 9-6
9.3.2. Run-time Type Checking and Type Conversion 9-9
9.4. Examples 9-9
9.4.1. Polymorphic Downsampler 9-9
9.4.2. Fork Connection 9-10
9.4.3. A Sampler System 9-10
Appendix: The Type Resolution Algorithm 9-12
10. Plot Package 10-1
10.1. Overview 10-1
10.2. User Interface 10-1
10.3. File Format 10-3
10.3.1. Commands Configuring the Axes 10-3

10.3.2. Commands for Plotting Data 10-4
10.4.Exporting 10-6
10.5.Limitations 10-6
Part 3: Domains
11. CT Domain 11-1
11.1.Introduction 11-1
11.1.1. Basic Terminology 11-1
11.1.2. Time 11-3
11.1.3. Fixed-Point Behavior 11-3
11.1.4. Discontinuity 11-3
11.2.System Specification 11-4
11.2.1. An Example 11-5
11.3.CT Actors 11-7
11.3.1. Integrator and ODE Solvers 11-7
11.3.2. Actor Library 11-9
11.3.3. Domain Polymorphic Actors 11-10
11.4.CT Directors 11-10
11.4.1. CT Director Parameters 11-11
11.4.2. CTSingleSolverDirector 11-11
11.4.3. CTMultiSolverDirector 11-12
11.4.4. CTMixedSignalDirector 11-12
11.4.5. CTEmbeddedDirector 11-13
11.5.CT Domain Demos 11-13
11.5.1. Lorenz System 11-13
11.5.2. Micro Accelerator with Digital Feedback. 11-14
11.5.3. Thermostat System 11-15
11.6.Implementations 11-16
11.7.Technical Details 11-18
11.7.1. Scheduling 11-18
11.7.2. Controlling Step Sizes 11-20
11.7.3. Interaction with other domains 11-21
Appendix: Brief Mathematical Background 11-22
12. DE Domain 12-1
12.1.Introduction 12-1
12.1.1. Model Time 12-1
12.1.2. Iteration 12-2
12.1.3. Getting a Model Started 12-2
12.1.4. Stopping Execution 12-2
12.2.Overview of The Software Architecture 12-3
12.3.The DE Actor Library 12-3
12.4.Mutations 12-3
12.5.Writing DE Actors 12-5
12.5.1. General Guidelines 12-5
12.5.2. Simultaneous Events 12-7
12.5.3. Examples 12-8
12.5.4. Thread Actors 12-11

12.6.Composing DE with Other Domains 12-13
12.6.1. DE inside Another Domain 12-14
12.6.2. Another Domain inside DE 12-15
13. SDF Domain 13-1
13.1.Overview 13-1
13.1.1. Properties 13-1
13.1.2. Scheduling 13-2
13.2.Kernel 13-3
13.2.1. SDF Director 13-3
13.2.2. Scheduling 13-5
13.2.3. SDF ports and receivers 13-6
13.2.4. ArrayFIFOQueue 13-6
13.2.5. SDFAtomicActor 13-6
14. CSP Domain 14-1
14.1.Introduction 14-1
14.2.CSP Communication Semantics 14-2
14.2.1. Atomic Communication: Rendezvous 14-2
14.2.2. Choice: Nondeterministic Rendezvous 14-2
14.2.3. Deadlock 14-4
14.2.4. Time 14-4
14.2.5. Differences from Original CSP Model as Proposed by Hoare 14-5
14.3.Example CSP Applications 14-5
14.3.1. Dining Philosophers 14-6
14.3.2. Hardware Bus Contention 14-7
14.3.3. Sieve of Eratosthenes 14-7
14.3.4. An M/M/1 Queue 14-7
14.4.Building CSP Applications 14-9
14.4.1. Rendezvous 14-9
14.4.2. Conditional Communication Constructs 14-9
14.4.3. Time 14-10
14.5.The CSP Software Architecture 14-11
14.5.1. Class Structure 14-11
14.5.2. Starting the model 14-13
14.5.3. Detecting deadlocks: 14-13
14.5.4. Terminating the model 14-14
14.5.5. Pausing/Resuming the Model 14-15
14.6.Technical Details 14-15
14.6.1. Brief Introduction to Threads in Java 14-15
14.6.2. Rendezvous Algorithm 14-17
14.6.3. Conditional Communication Algorithm 14-19
14.6.4. Modification of Rendezvous Algorithm 14-21
15. DDE Domain 15-1
15.1.Introduction 15-1
15.2.DDE Semantics 15-1
15.2.1. Enabling Communication: Advancing Time 15-2
15.2.2. Maintaining Communication: Null Tokens 15-3
15.2.3. Alternative Distributed Discrete Event Methods 15-5

15.3.Example DDE Applications 15-5
15.4.Building DDE Applications 15-6
15.4.1. DDEActor 15-6
15.4.2. DDEIOPort 15-6
15.4.3. Feedback Topologies 15-7
15.5.The DDE Software Architecture 15-7
15.5.1. Local Time Management 15-8
15.5.2. Detecting Deadlock 15-9
15.5.3. Ending Execution 15-9
15.6.Technical Details 15-10
15.6.1. Synchronization Hierarchy 15-10
16. PN Domain 16-1
16.1.Introduction 16-1
16.2.Process Network Semantics 16-2
16.2.1. Asynchronous Communication 16-2
16.2.2. Bounded Memory Execution 16-2
16.2.3. Time 16-2
16.2.4. Mutations 16-3
16.3.The PN Software Architecture 16-3
16.3.1. PN Domain 16-3
16.3.2. The Execution Sequence 16-4
16.3.3. Detecting deadlocks: 16-7
16.3.4. Terminating the model: 16-8
16.3.5. Mutations of a Graph 16-8
16.4.Technical Details 16-9
16.4.1. Mutual Exclusion using Monitors 16-9
16.4.2. Hierarchy of Locks 16-10
16.4.3. Undetected Deadlocks 16-11
References R-1
Glossary G-1
Index I-1

PART 1:

USING PTOLEMY II

The chapters in this part describe 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 (and some that are just planned). It describes the package structure of the software, and includes as an appendix a brief tutorial on UML notation, which is used throughout this document to explain the structure of the software. The second chapter includes a tutorial on constructing applets. The third chapter gives an overview of domain-polymorphic actor libraries. Model builders will also want to refer to the domain chapter for the particular domain(s) they are using, since domain-specific actors are described there.

1

Introduction

Author: Edward A. Lee

1.1 Modeling and Design

The Ptolemy project studies heterogeneous modeling, simulation, and design of concurrent systems. The focus is on *embedded systems*, particularly those that mix technologies, including for example analog and digital electronics, hardware and software, and electronics and mechanical devices (including MEMS, microelectromechanical systems). The focus is also on systems that are complex in the sense that they mix widely different operations, such as signal processing, feedback control, 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.

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 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 the interaction of components.

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

1.2 Architecture Design

Architecture description languages (ADLs), such as Wright [3] and Rapide [53], 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 [3], 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-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 [3], we use a technique much more powerful than type checking alone, namely polymorphism.

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 emboddied 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 embedded 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 [3], we have developed a more abstract formal framework that describes models of computation at a meta level [49]. 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 [3].

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. In this section, we describe models of computation that are implemented in Ptolemy II domains, plus a couple of additional ones that are planned. 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 1.1. Ptolemy II models are (clustered, or hierarchical) graphs of the form of figure 1.1, 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 Communicating Sequential Processes - CSP

In the CSP domain (communicating sequential processes), created by Neil Smyth [78], actors 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) [36] and Milner’s *calculus of communicating systems* (CCS) [57]. 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.

1.3.2 Continuous Time - CT

In the CT domain (continuous time), created Jie Liu [51], 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.

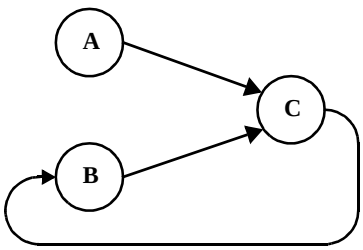


FIGURE 1.1. A single *syntax* (bubble-and-arc or block-and-arrow diagram) can have a number of possible *semantics* (interpretations).

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.

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*. 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.

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.

1.3.3 Discrete-Events - DE

In the discrete-event (DE) domain, created by Lukito Muliadi [61], 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 [11] 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.

1.3.4 Distributed Discrete Events - DDE

The distributed discrete-event (DDE) domain, created by John Davis, 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.

1.3.5 Discrete Time - DT

The discrete-time (DT) domain, which has not been written yet, will extend 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, will be supported.

1.3.6 Finite-State Machines - FSM

The finite-state machine (FSM) domain, written by Xiaojun Liu (but not yet released), 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 control logic in embedded systems, particularly safety-critical systems. FSM models are amenable to in-depth formal analysis, and thus can be used to avoid surprising behavior.

FSM models have a number of 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.

The latter problem 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 [30]. FSMs have also been combined with differential equations, yielding the so-called *hybrid systems* model of computation [32].

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 [28]. 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 [77].

1.3.7 Process Networks - PN

In the process networks (PN) domain, created by Mudit Goel [29], 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 [40].

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 [41], 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.

1.3.8 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 [48]). 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 interrim to handle computations that do not match the restrictions of SDF.

1.3.9 Synchronous/Reactive - SR

In the synchronous/reactive (SR) model of computation [7], 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 [9], Signal [8], Lustre [17], and Argos [54].

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. An SR domain has not yet been implemented in Ptolemy II, although the methods used by Stephen Edwards in Ptolemy Classic can be adapted to this purpose [20].

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 [49].

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, for example [3]. 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. Rendezvous 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 electronic 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.

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.

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.

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 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, and in fact is used fairly extensively in the design of Ptolemy II itself (see appendix A of this chapter).

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

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.

1.6.1 Package Structure

The package structure is shown in figure 1.2. 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.event* which 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 for domain-specific classes that control the execution of a model.

actor.gui This subpackage is a library of polymorphic actors with user interface components, plus some convenience base classes for applets and applications.

actor.lib This subpackage is a library of polymorphic actors.

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

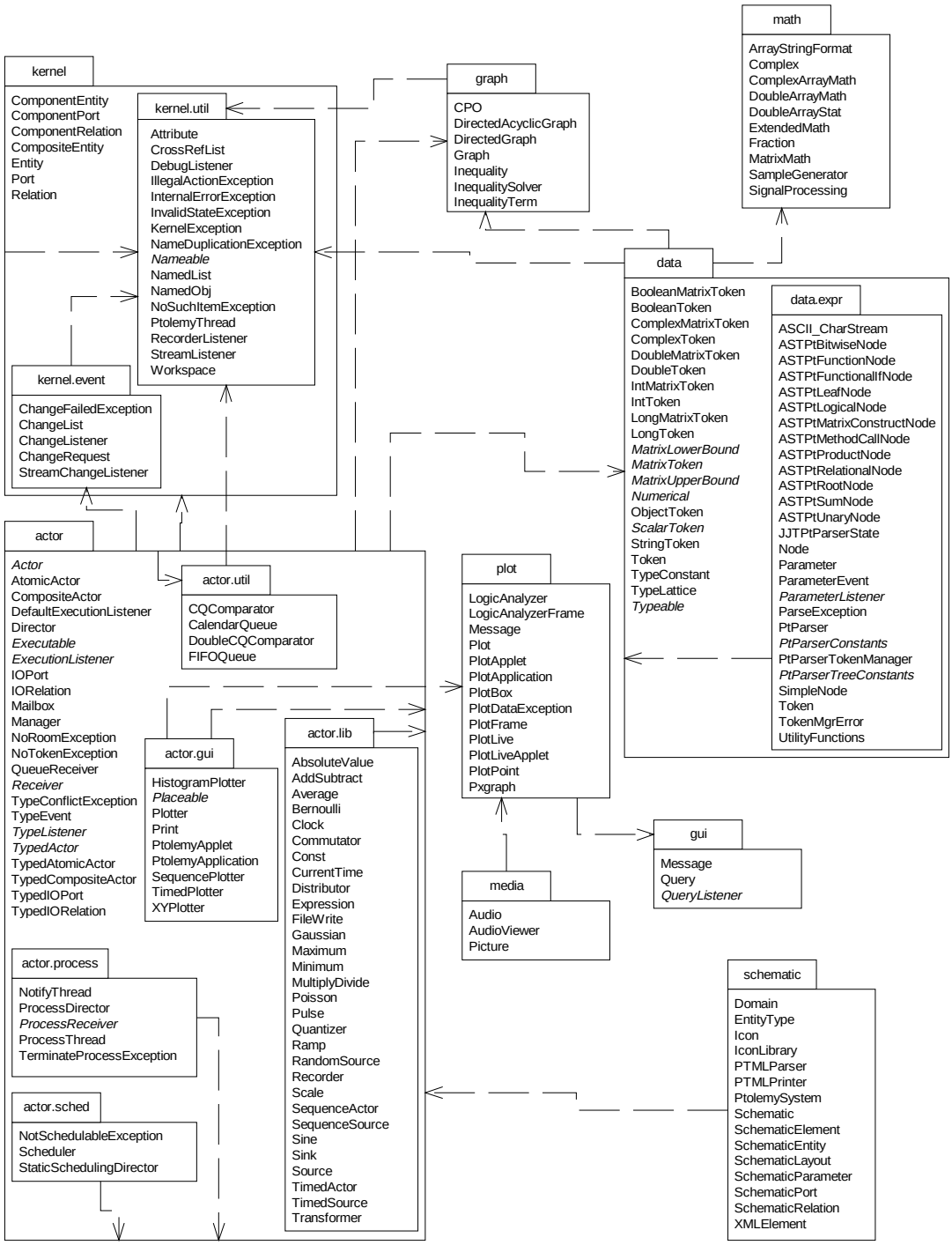


FIGURE 1.2. The package structure of Ptolemy II, without the domains.

	scheduled by the director.
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.
data	This package provides classes that encapsulate and manipulate data that is transported between actors in Ptolemy models.
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.
domains	This package contains one subpackage for each Ptolemy II domain.
graph	This package provides algorithms for manipulating and analyzing mathematical graphs. Mathematical graphs are simpler than Ptolemy II clustered graphs in that there is no hierarchy, and arcs link exactly two nodes. This package is expected to supply a growing library of algorithms.
gui	This package contains generically useful user interface components.
kernel	This package provides the software architecture for the key abstract syntax, 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 cluster entity.
kernel.event	This package contains classes and interfaces that support controlled mutations of clustered graphs. Mutations are modifications in the topology, and in general, they are permitted to occur during the execution of a model. But in certain domains, where maintaining determinacy is imperative, the director may wish to exercise tight control over precisely when mutations are performed. This package supports queueing of mutation requests for later execution. It uses a publish-and-subscribe design pattern.
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 and a class supporting fractions.
media	This package encapsulates a set of classes supporting audio and image processing.
plot	This package provides two-dimensional signal plotting widgets.
schematic	This package provides a top-level interface to Ptolemy II. A GUI can use the classes in this package to gain access to Ptolemy II models.

1.6.2 Overview of Key Classes

Some of the key classes in Ptolemy II are shown in figure 1.3. 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, and other lines, which indicate association. 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 primitives of the abstract syntax supported by Ptolemy II. They will be fully explained in the kernel chapter. ComponentPort, ComponentRelation, and ComponentEntity extend these classes by adding

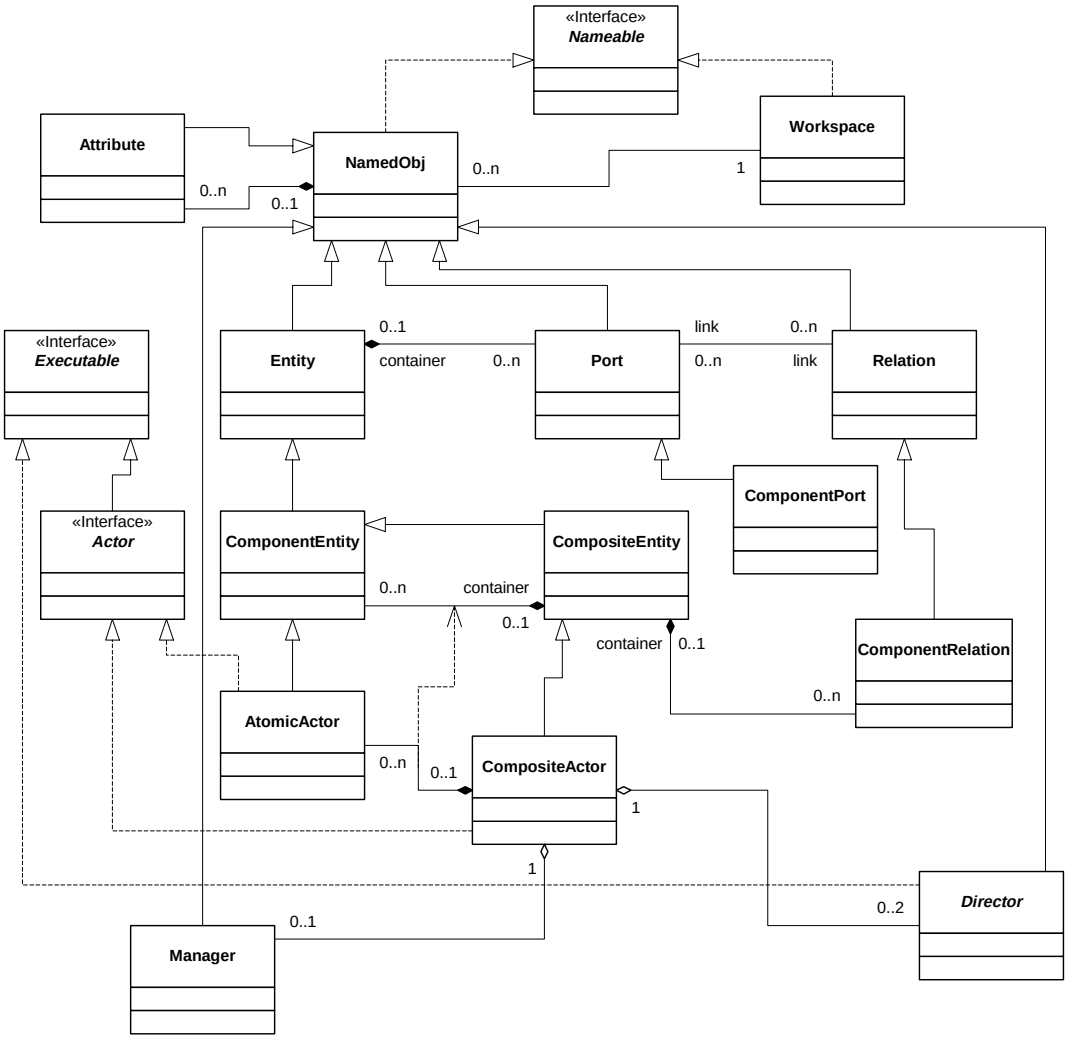


FIGURE 1.3. Some of the key classes in Ptolemy II. These are defined in the *kernel*, *kernel.util*, and *actor* packages.

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.

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, as shown in figure 1.4. These packages generally contain a kernel subpackage, which defines classes that extend classes in the actor or kernel packages of Ptolemy II. The gui subpackage contains a domain-specific applet class, which provides facilities for easily creating applets that use that domain. The lib subpackage, when it exists, includes domain-specific actors.

1.6.4 Capabilities

Ptolemy II is a second generation system. Its predecessor, Ptolemy Classic, still has many active users and developers, and may continue to evolve for some time. 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™*. Java support for concurrent design is very low level, based on threads and monitors. Maintaining safety and liveness can be quite difficult [43]. Ptolemy II includes a number of domains that support design of concurrent systems at a much higher level 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 and scalable.
- *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.* Ptolemy Classic provided a wormhole mechanism for hierarchically coupling heterogeneous models of computation. This mechanism is improved in Ptolemy II through

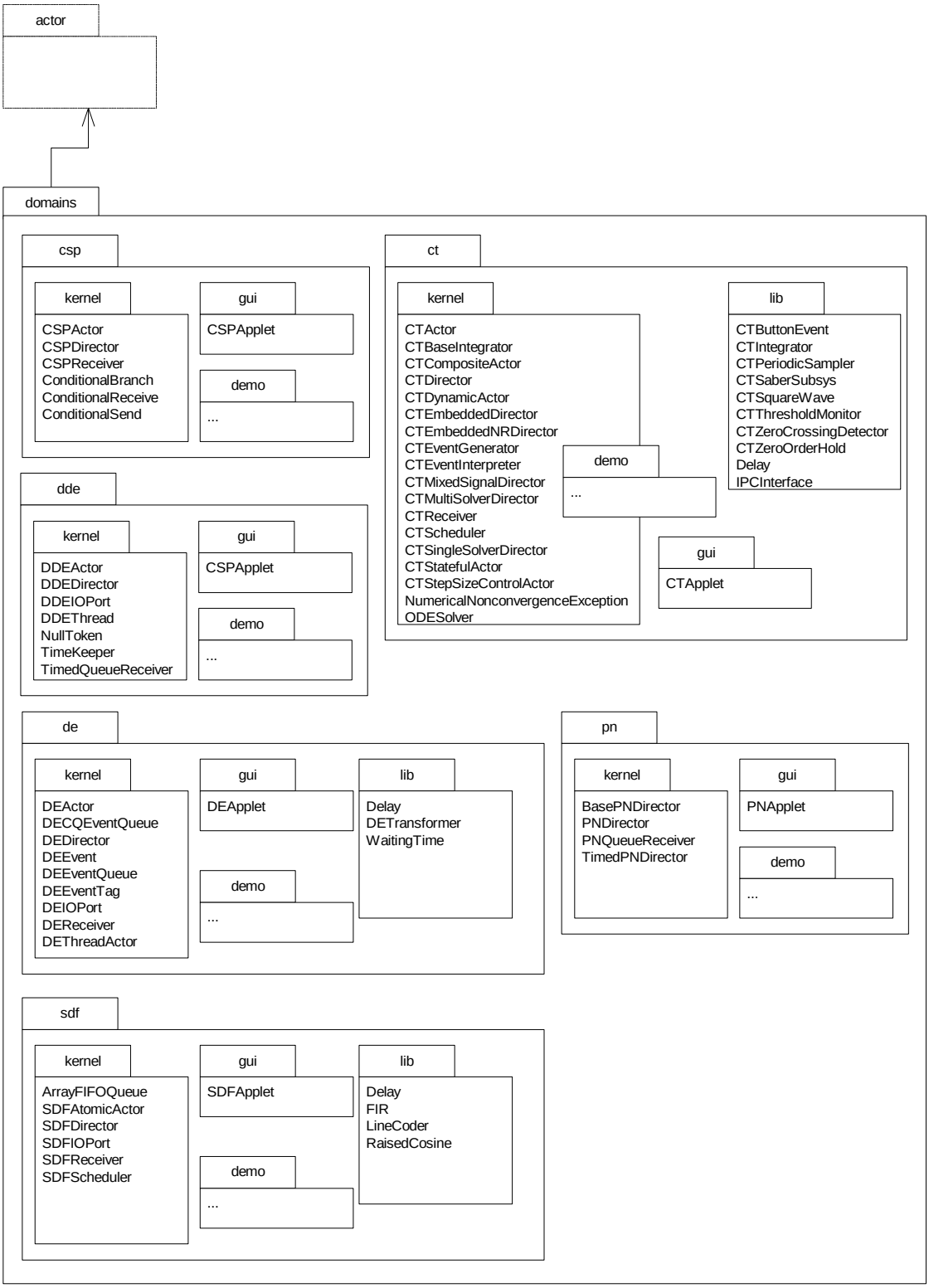


FIGURE 1.4. Package structure of Ptolemy II domains.

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 [36] 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 [56][73][75] and design pattern [25] 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 [71].
- *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 [59].
- *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*.

1.6.5 Future Capabilities

Capabilities that we anticipate making available in the future include:

- *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.
- *Interoperability through software components*. Ptolemy II will use distributed software component technology such as CORBA, Java RMI, or DCOM, in a number of ways. Components (actors) in a Ptolemy II model will be implementable on a remote server. Also, components may be parameterized where parameter values are supplied by a server (this mechanism supports *reduced-order modeling*, where the model is provided by the server). Ptolemy II models will be exported via a server. And finally, Ptolemy II will support migrating software components.
- *Embedded software synthesis*. Pertinent Ptolemy II domains will be tuned to run on a Java virtual machine on an embedded CPU. Hardware, firmware, and configurable hardware components will expose abstractions to this Java software that obey the model of computation of the pertinent domain. Java's native code interface will be used to define a stub for the embedded hardware components so that they are indistinguishable from any other Java thread within the model of computation. Domains that seem particularly well suited to this approach include PN and CSP.

- *Embedded hardware synthesis*. Ptolemy Classic had only very weak mechanisms for migrating designs from idealized floating-point simulations through fixed-point simulations to embedded software, FPGA, and hardware designs. Ptolemy II will leverage polymorphism, allowing libraries to be constructed where compatibility across implementation technologies is assured [74].
- *Integrated verification tools.* Modern verification tools based on model checking [33] 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.

Appendix A: UML — Unified Modeling Language

UML (the unified modeling language) [23][70] 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 here the style that we use.

A.1 Package Diagrams

Figures 1.2 and 1.4 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 1.3 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 1.5. 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 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 1.5. 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 1.5 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 1.5 does not show all methods, so that we can simplify the discussion of UML.

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 1.5, ComponentEntity has all the methods of Entity and NamedObj, and may override some of those methods, but only 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.

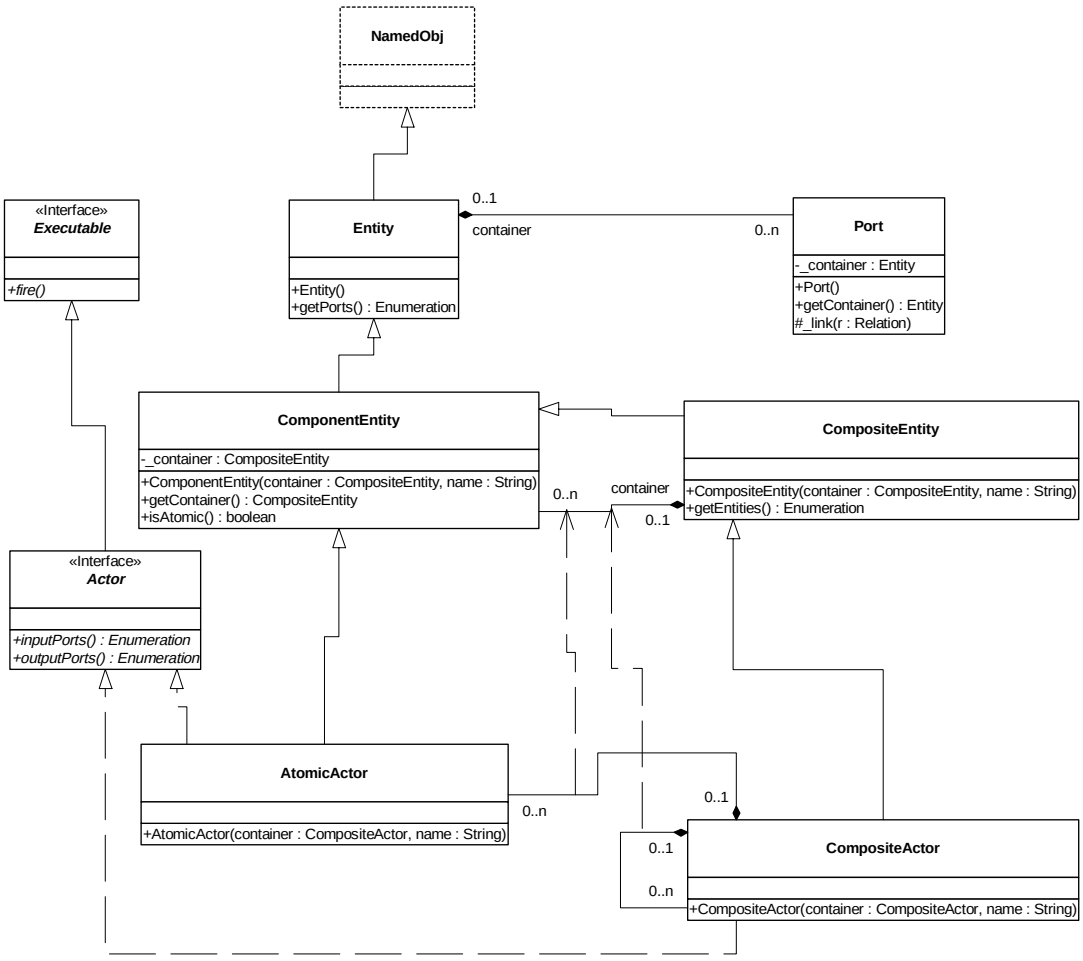


FIGURE 1.5. Simplified static structure diagram for some Ptolemy II classes. This diagram illustrates features of UML syntax that we use.

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 1.5 does not show all the constructors of these classes, for simplicity.

Classes shown in boxes outlined with dashed lines, such as NamedObj in figure 1.5, 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 1.5 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 1.5, 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 1.5.

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 1.5. 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 [25], which supports arbitrarily deeply nested containment hierarchies.

In figure 1.5, 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 getPorts(), usually return an enumeration (or sometimes an array, or an iterator).

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

2

Building Models

Author: Edward A. Lee
Contributor: Christopher Hylands

2.1 Introduction

Ptolemy II models can be specified in a number of ways and used in a number of ways. They 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). At the current writing, applets have the best developed infrastructure, so we describe here how to construct them. It is possible today to construct other network-integrated architectures, but there is less built-in Ptolemy II support at this time.

Eventually, you will be able to construct applets and other Ptolemy II architectures using XML files. There will also be graphical editors for building these XML files using, for example, block diagrams. However, at this time, you must write Java code to construct a Ptolemy II model.

2.2 Applets

Ptolemy II models can be embedded in applets[1]. For convenience, each domain includes a base class *XX*Applet, where *XX* is replaced by the domain name. This section uses a DE domain applet to illustrate the basic concepts, so the base class is DEApplet. Refer to subsequent chapters and to the code documentation for more complete information about the classes and methods being used. DEApplet is derived from PtolemyApplet, as shown in figure 2.1 (see appendix A of chapter 1 for UML syntax).

1. 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).

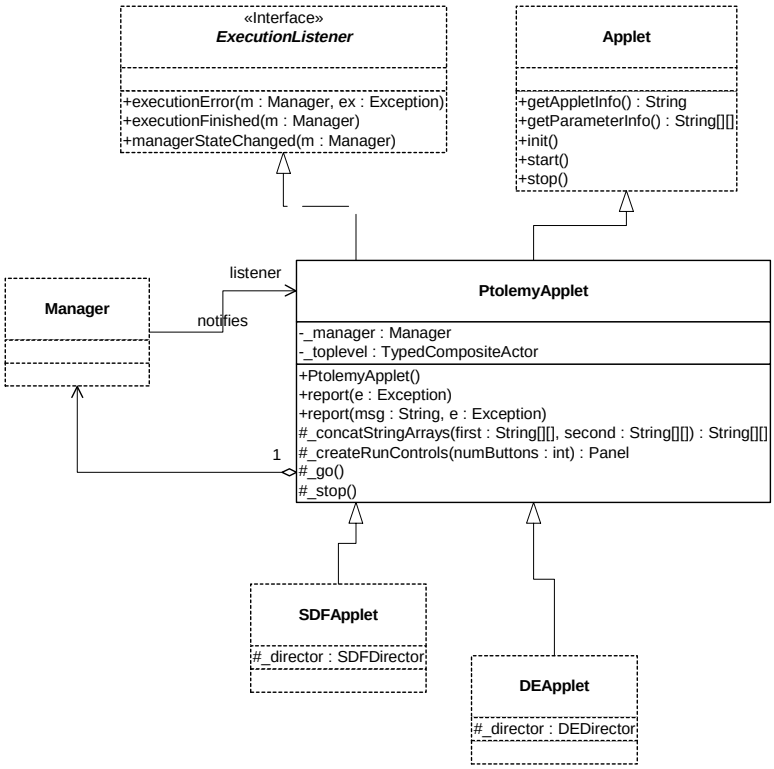


FIGURE 2.1. UML static structure diagram for PtolemyApplet, a convenience base class for constructing applets. PtolemyApplet is in the ptolemy.actor.gui package.

2.2.1 HTML Files Containing Applets

An applet is a Java class that can be referenced by an HTML file and accessed over the web. Unfortunately, most browsers available today do not have built-in support for the (relatively recent) version of Java that Ptolemy II is based on (version 1.2.1). The work around is to use the Java Plug-In, which invokes Sun's Java Runtime Environment (JRE), instead of the browser's default Java runtime. The table below lists platform and plug-in availability.I

Table 17: Plug-in Availability

Platform	Availability
Windows 95, 98, NT	JDK1.2.1 Plug-in installed as part of the JDK or JRE. See http://www.javasoft.com/products/plugin/
Irix	Plug-in available, see http://www.sgi.com/Products/Evaluation/#java_plugin
Solaris 2.6, 2.7	Custom version of Netscape Navigator available, see http://www.sun.com/solaris/netscape
Linux	As of 6/13/99, JDK 1.2.1 Plug-in not yet available, see http://www.blackdown.org/. Use the appletviewer program
Solaris 2.5.1 and all other platforms with JDK 1.2.1	Use the appletviewer program to run the applets.

Platforms that do not have the Java Plug-in 1.2.1, but do have the Java Development Kit (JDK) 1.2.1 can use Sun's appletviewer command to run applets locally. Each demonstration directory has a `make demo` rule that sets the `CLASSPATH` appropriately and then calls appletviewer. We discuss how to run `appletviewer` by hand in "Compiling" on page 2-5. However, the appletviewer does not render the HTML text in the web page, so you get only a subset of the information.

Regrettably, the way the plug-in is invoked depends on the browser being used. One approach to creating applets that use the plug-in is to write them as if they were to use the native Java runtime environment of the browser, and then invoke Sun's Plug-In HTML Converter, available on their web site. The converter did not work for me, however, at least under Windows NT (its file browser failed to list HTML files, or any other file, for that matter, in most directories). Thus, I recommend writing the HTML files directly rather than using the converter.

Sample HTML is shown in is shown in figure 2.2. The incredible ugliness and awkwardness of this text is hopefully transitory, while browser vendors agree on how to properly support plug-ins (I hope it is transitory, or plug-ins will not be a long term solution). You should be able to essentially copy what we have, making very few modifications, and get things to work.

The code in figure 2.2, amazingly, relies on the fact that Microsoft's Internet Explorer understands some HTML tags that Netscape Navigator does not, and vice versa. IE uses the `<OBJECT>` and `</OBJECT>` tags to specify plug-ins. The "classid" is a magic string that is always the same (it identifies the Java version 1.2 run-time environment, in a not very user-friendly way).

The `<COMMENT>` and `</COMMENT>` tag is understood only by Internet Explorer, so it is used to hide from IE the `<EMBED>` and `</EMBED>` tags, which Netscape uses to support plug-ins. The Sun appletviewer, a utility program for invoking applets without rendering the surrounding HTML, also uses this tag.

In the case of the `<OBJECT>` and `</OBJECT>` tags, the specified "codebase" is the place to find the plug-in if it is not already installed. The codebase for the applet itself (the root of the tree to search for classes) is given by a `<PARAM>` tag with name "codebase." This is confusing, especially given that for the `<EMBED>` and `</EMBED>` tags, the plug-in location is given by the pluginspage parameter, and codebase refers to the code base for the applet. We can only hope that some day there will be standard-

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
   width="700"
   height="300"
   codebase="http://java.sun.com/products/plugin/1.2/jinstall-12-win32.cab#Version=1,2,0,0">
<PARAM NAME="code" VALUE="doc.tutorial.TutorialApplet.class">
<PARAM NAME="codebase" VALUE="../..">
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.2"
   width="700"
   height="300"
   code="doc/tutorial/TutorialApplet.class"
   codebase="../.."
   pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html">
</COMMENT>
<NOEMBED>
No JDK 1.2 support for applet!
</NOEMBED>
</EMBED>
</OBJECT>
```

FIGURE 2.2. An HTML segment that invokes the Java 1.2 Plug-in under both Netscape and Internet Explorer (it is regrettable how complex this is). This text can be found in $PTII/doc/tutorial/tutorial.htm.

ization of HTML.

In the case of the <OBJECT> and </OBJECT> tags, if there were any additional applet parameters, they would be given in <PARAM> tags. For the <EMBED> and </EMBED> tags, applet parameters are given as parameters of the <EMBED> tag.

An HTML file containing the segment shown in figure 2.2 can be found in $PTII/doc/tutorial/tutorial.htm, where $PTII is the home directory of the Ptolemy II installation. For example, on my machine (running Windows NT and using bash as a shell), I have Ptolemy II installed in d:\ptII, and using the system control panel, I have the environment variable PTII set to d:/ptII (I'm not sure whether it matters that the slashes go different ways). 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. To compile and use each file, it must be copied into TutorialApplet.java (without the *n*).

Since our example applets are in a directory $PTII/doc/tutorial, the codebase for the applet is "../.." in figure 2.2, 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 2.2 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. Unfortunately, they are specified twice in the file.

Fortunately, getting the Java code right is easy compared to getting the HTML right.

2.2.2 Creating Models

In figure 2.3 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 three lines, the applet imports three classes from Ptolemy II:

- `DEApplet`: A base class for DE applets that is provided for convenience. This base class creates a top-level composite actor called `_toplevel`, a manager called `_manager`, and a director called `_director` (all protected members of the class, shown in figure 2.1). 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

```
package doc.tutorial;
import ptolemy.domains.de.gui.DEApplet;
import ptolemy.actor.lib.Clock;
import ptolemy.actor.gui.TimedPlotter;

public class TutorialApplet extends DEApplet {
    public void init() {
        super.init();
        try {
            Clock clock = new Clock(_toplevel,"clock");
            TimedPlotter plotter = new TimedPlotter(_toplevel,"plotter");
            _toplevel.connect(clock.output, plotter.input);
        } catch (Exception ex) {}
    }
}
```

FIGURE 2.3. An extremely simple applet that runs in the DE domain. This text can be found in $PTII/tutorial/TutorialApplet1.java. It should be copied to TutorialApplet.java before compiling. This text can be found in $PTII/doc/tutorial/TutorialApplet1.java.

placed one time unit apart and alternating in value between 1 and 0.

- `TimedPlotter`: This is an actor that plots functions of time.

Next, the construct

```
public class TutorialApplet extends DEApplet { ... }
```

defines a class called TutorialApplet that extends DEApplet. The new class overrides the init() method of the base class with the following body:

```
super.init();
try {
   Clock clock = new Clock(_toplevel,"clock");
   TimedPlotter plotter = new TimedPlotter(_toplevel,"plotter");
   _toplevel.connect(clock.output, plotter.input);
} catch (Exception ex) {}
```

This first invokes the base class, then creates an instance of Clock and an instance of TimedPlotter, and connects them together.

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 `_toplevel`, provided by the base class, 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. We will discuss the try . . . catch statement below.

2.2.3 Compiling

To compile this class definition, you must first copy TutorialApplet1.java into TutorialApplet.java (Java requires that file names match the names of the classes they contain). Then set the CLASSPATH environment variable to point to the root of the Ptolemy II tree. For example, in bash, assuming the variable PTII is set,

```
bash-2.02$ cd $PTII/doc/tutorial
bash-2.02$ cp TutorialApplet1.java TutorialApplet.java
bash-2.02$ CLASSPATH=$PTII
bash-2.02$ export CLASSPATH
bash-2.02$ javac TutorialApplet.java
```

(The part before the “$” is the prompt issued by bash). You should now be able to run the applet with the command:

```
bash-2.02$ appletviewer tutorial.htm
```

The result of running the applet is a new window which should look like that shown in figure 2.4. This is not (yet) a very interesting applet. Let us improve on it.

2.2.4 Reporting Errors

The code in figure 2.3 has a try ... catch statement that does something that is almost never a good idea: it discards exceptions. If an error were to occur during construction of the model, this statement would mask the error, and the applet would silently fail.

The base class PtolemyApplet, fortunately, provides a report() method (see figure 2.1) for reporting errors in a uniform way. The modified code is shown in figure 2.5.

2.2.5 Graphical Elements

The applet, as written so far, has the annoying feature that it opens a new window to display the plotted results, as shown in figure 2.4. Most applets will want to display their results in the browser window, as part of the text of a web page.

The TimedPlotter actor, and most other Ptolemy II components with graphical elements, implements the Placeable interface. This interface has a single method, setPanel(), which can be used to specify the panel into which the object should be placed. If this method is not called before the object is mapped to the screen, then the object will create its own frame into which to place itself. This is

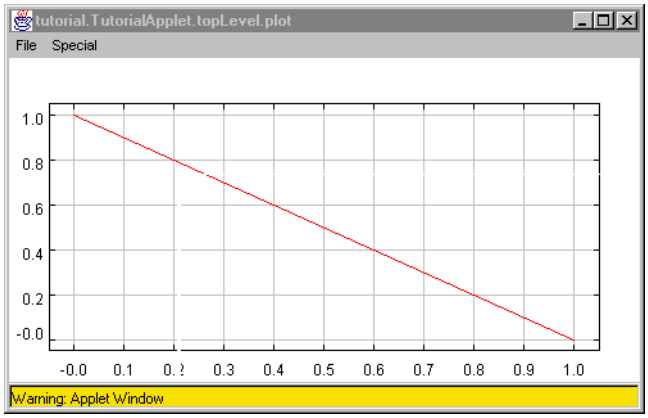


FIGURE 2.4. Result of running the (all too simple) applet of figure 2.3.

```
package doc.tutorial;
import ptolemy.domains.de.gui.DEApplet;
import ptolemy.actor.lib.Clock;
import ptolemy.actor.gui.TimedPlotter;

public class TutorialApplet extends DEApplet {
    public void init() {
        super.init();
        try {
            Clock clock = new Clock(_toplevel,"clock");
            TimedPlotter plotter = new TimedPlotter(_toplevel,"plotter");
            _toplevel.connect(clock.output, plotter.input);
        } catch (Exception ex) {
            report("Error constructing model.", ex);
        }
    }
}
```

FIGURE 2.5. An improved applet that properly reports errors in model construction. This text can be found in $PTII/doc/tutorial/TutorialApplet2.java.

what happened with our applet, resulting in the frame shown in figure 2.4.

Modified code that places the plot in the applet window itself (an instance of Applet is a Panel) is shown in figure 2.6. Note that the size of the plot is now also specified to match the size of the applet, using the statement:

```
plotter.plot.setSize(700,300);
```

This line refers to a public member of the plotter object, called "plot," which is an instance of the class Plot, from the ptolemy.plot package. That class has a method, setSize(), that can be used to control the size of the plot.

The resulting applet looks like that shown in figure 2.7. In this case, the default layout manager of the Applet class is allowed to control where the plot is placed. Arbitrarily fine control can be exercised, however, by placing a new instance of Panel in the applet using any suitable layout manager and then specifying that panel for the plotter using its setPanel() method. This is done in the next section.

2.2.6 Controlling Execution Time

By default, the director in the DE domain executes a model for one time unit. This does not give a a very satisfactory plot in figure 2.7. To change the amount of time to 30, include in the body of the init() method the following line:

```
_director.setStopTime(30.0);
```

Alternatively, you can set the stop time in the HTML file by setting an applet parameter called "stopTime." To set this applet parameter to 100, add the line

```
<PARAM NAME="stopTime" VALUE="100">
```

to the `<OBJECT>` ... `</OBJECT>` body in figure 2.2, and the parameter

```
stopTime="100"
```

```
package doc.tutorial;
import ptolemy.domains.de.gui.DEApplet;
import ptolemy.actor.lib.Clock;
import ptolemy.actor.gui.TimedPlotter;

public class TutorialApplet extends DEApplet {
    public void init() {
        super.init();
        try {
          Clock clock = new Clock(_toplevel,"clock");
          TimedPlotter plotter = new TimedPlotter(_toplevel,"plotter");
          plotter.setPanel(this);
          plotter.plot.setSize(700,300);
          _toplevel.connect(clock.output, plotter.input);
        } catch (Exception ex) {
          report("Error constructing model.", ex);
        }
    }
}
```

FIGURE 2.6. A modified applet that places the resulting plot in the browser window itself, as shown in figure 2.7. This text can be found in $PTII/doc/tutorial/TutorialApplet3.java.

to the <EMBED> tag. Regrettably, both must be added or the applet will behave differently in IE and Netscape.

You can instruct the applet to put controls on the screen that allow the applet user to control the model execution time with the following line (see figure 2.1):

```
add(_createRunControls(2));
```

The argument specifies how many run control buttons to create. A value of “1” or larger requests a “go” button, while “2” or larger requests both a “go” and a “stop” button. We must also modify the size of the plotter, as shown in figure 2.8. The size is set so that 50 pixels in the vertical direction are allowed for the run controls. The resulting page, with an entered execution time of 30, is shown in figure 2.9.

The default stop time in the entry box can be controlled by setting an applet parameter “defaultStopTime.” To set this applet parameter so that the default is 100, add the line

```
<PARAM NAME="defaultStopTime" VALUE="100">
```

to the <OBJECT> ... </OBJECT> body in figure 2.2, and the parameter

```
defaultStopTime="100"
```

to the <EMBED> tag.

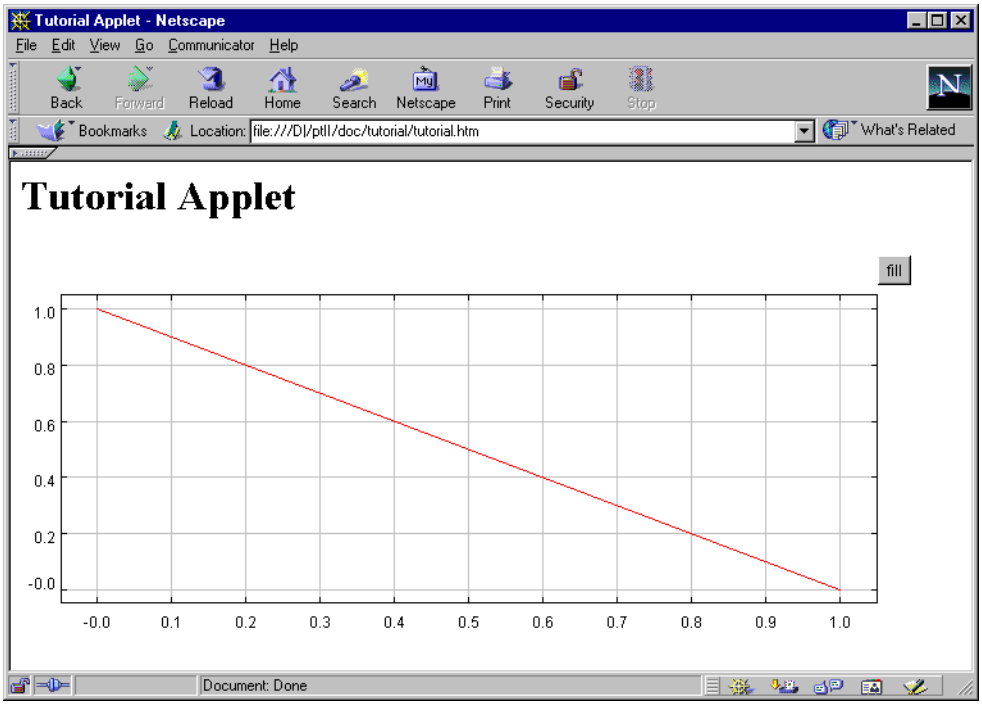


FIGURE 2.7. Applet with embedded plot as displayed in Netscape Navigator.

In the SDF domain, the corresponding parameters are “iterations” and “defaultIterations.” For other domains, see the documentation for the *XXX*Applet class, where *XXX* is the domain name.

2.2.7 Controlling Model Parameters

Most actors in Ptolemy II have parameters that control their behavior. The Clock actor, for exam-

```
package doc.tutorial;
import ptolemy.domains.de.gui.DEApplet;
import ptolemy.actor.lib.Clock;
import ptolemy.actor.gui.TimedPlotter;
import java.awt.Panel;
import java.awt.Dimension;

public class TutorialApplet extends DEApplet {
    public void init() {
        super.init();
        try {
            Clock clock = new Clock(_toplevel,"clock");
            TimedPlotter plotter = new TimedPlotter(_toplevel,"plotter");
            plotter.setPanel(this);
           plotter.plot.setSize(700, 250);
            _toplevel.connect(clock.output, plotter.input);
            add(_createRunControls(2));
        } catch (Exception ex) {
            report("Error constructing model.", ex);
        }
    }
}
```

FIGURE 2.8. Code that adds execution time controls to the applet and resizes the plotter display to make room for them. This code can be found in $PTII/doc/tutorial/TutorialApplet4.java.

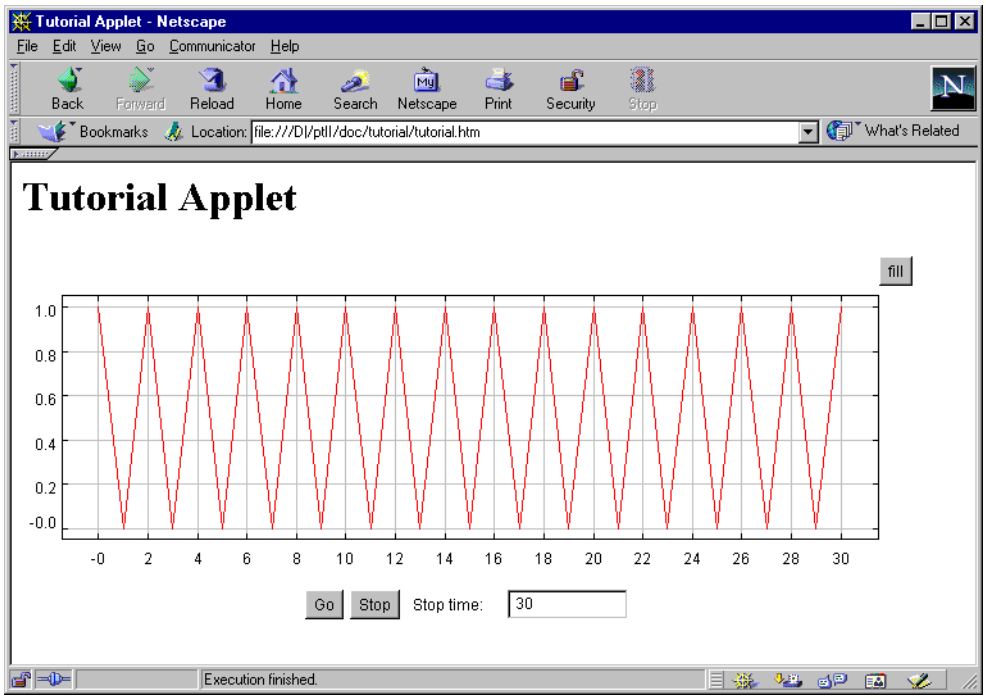


FIGURE 2.9. Browser view of the applet in figure 2.8.

ple, has a *period* parameter. Parameters are generally public members, instances of the class Parameter. To change the period of the clock from the default 2.0 to, say, 1.0, add the line to the init() method of the applet:

```
clock.period.setExpression("1.0");
```

More interestingly, you may wish to offer this control to the applet user. This can be done using built-in Java classes to create an entry box in the applet, or more easily by using an instance of the Query class in the ptolemy.gui package.

The code shown in figure 2.10 results in the browser display shown in figure 2.11. The important changes are shown in bold. First, the class now has a private member _query that is an instance of Query. In addition, a reference to the Clock actor is now stored in a private member _clock instead of in a local variable in the init() method. The following lines have been added to the init() method:

```
_query.setBackground(getBackground());
_query.addLine("period", "Period", "2.0");
_query.addQueryListener(this);
add(_query);
```

```
package doc.tutorial;
import ptolemy.domains.de.gui.DEApplet;
import ptolemy.actor.lib.Clock;
import ptolemy.actor.gui.TimedPlotter;
import ptolemy.gui.Query;
import ptolemy.gui.QueryListener;
import java.awt.Panel;
import java.awt.Dimension;

public class TutorialApplet extends DEApplet implements QueryListener {
    private Query _query = new Query();
    private Clock _clock;
    public void init() {
        super.init();
        try {
            _clock = new Clock(_toplevel,"clock");
            TimedPlotter plotter = new TimedPlotter(_toplevel,"plotter");
            plotter.setPanel(this);
            plotter.plot.setSize(700, 250);
            _toplevel.connect(_clock.output, plotter.input);
            add(_createRunControls(2));
            _query.setBackground(getBackground());
            _query.addLine("period", "Period", "2.0");
            _query.addQueryListener(this);
            add(_query);
        } catch (Exception ex) {
            report("Error constructing model.", ex);
        }
    }
    public void changed(String name) {
        _clock.period.setExpression(_query.stringValue("period"));
        try {
            _go();
        } catch (Exception ex) {
            report("Error executing model.", ex);
        }
    }
}
```

FIGURE 2.10. Code that adds a parameter control to the applet. This code can be found in $PTII/doc/tutorial/TutorialApplet5.java.

The first of these sets the background color of the query widget to match the background color of the applet. The second adds a single-line entry box to the query, with name “period,” label “Period,” and default entry “2.0.” The name is an arbitrary string that can be used elsewhere to refer to this particular entry in the query. In this applet, the query has only one entry, so giving it a name seems unnecessary. But in general, a query can have any number of entries, so unique names are necessary.

The third line above informs the query that this applet is a listener, meaning that it should be notified when any entry in the query changes. To be a listener, it must implement the QueryListener interface, which requires that a method changed() be implemented. That method is defined as:

```
public void changed(String name) {
   _clock.period.setExpression(_query.stringValue("period"));
   try {
      _go();
   } catch (Exception ex) {
      report("Error executing model.", ex);
   }
}
```

This method is called by the query object whenever an entry in the query changes. The argument is the name of the entry that changed. In this applet, there is only one entry, so we can ignore the argument. We use the stringValue() method of the Query class to obtain the text of the entry, and the setExpression() method of the *period* parameter to set its value. In addition, we invoke the protected method _go() of the parent class, DEApplet, to execute the model. Thus, whenever the applet user changes the

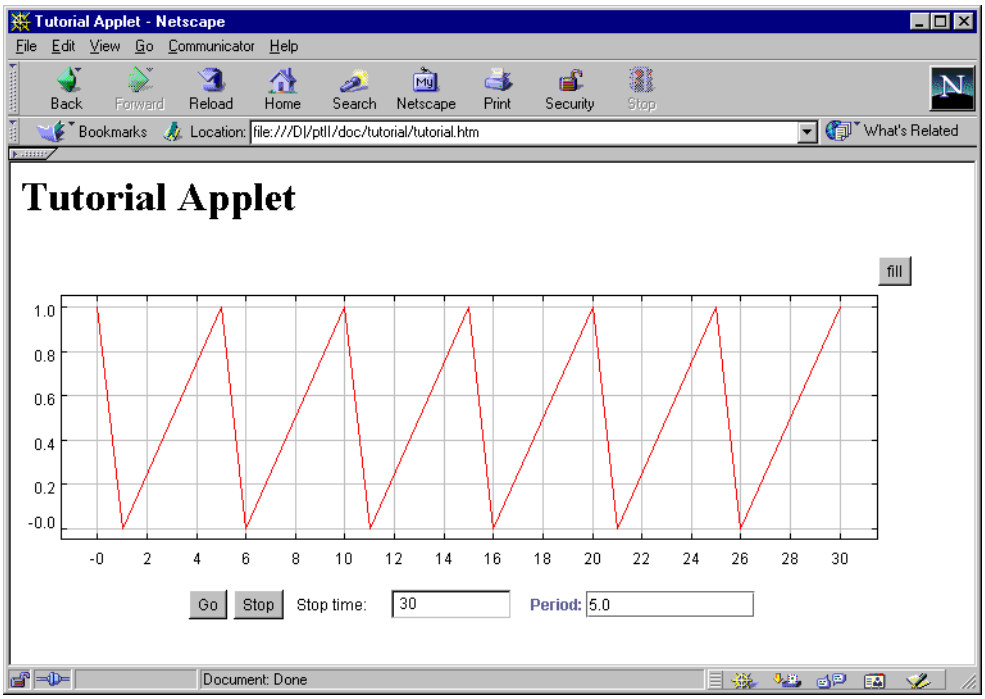


FIGURE 2.11. Browser view of the applet in figure 2.10.

period parameter, the model automatically executes again.

Notice that the method name setExpression() suggests that the value need not be a number, but rather can be an expression. Indeed, this is the case. The expression language that is supported is defined in the Data Package chapter. In short, ordinary arithmetic operations on constants are supported, as are all the methods of the Java Math class (sin(), cos(), etc.), and some pre-defined constants (pi, e, i, ...). In addition, expressions can symbolically refer to other parameters (by name) contained in the same container, or contained in the container of the container. Thus, for example, if you have several actors that you want to have the same parameter value, define a parameter "x" in the container (composite actor) and set the parameters in the actors to have value "x".

The plot display that is generated by this applet can be improved considerably. The TimedPlotter actor is somewhat special in that it contains a public member that is not a Parameter, but is rather an instance of Plot (see the Plot chapter). An instance of Plot can be configured in a large number of ways. In figure 2.12, we have added the following lines to the applet init() method:

```
package doc.tutorial;
import ptolemy.domains.de.gui.DEApplet;
import ptolemy.actor.lib.Clock;
import ptolemy.actor.gui.TimedPlotter;
import ptolemy.gui.Query;
import ptolemy.gui.QueryListener;
import java.awt.Panel;
import java.awt.Dimension;

public class TutorialApplet extends DEApplet implements QueryListener {
    private Query _query = new Query();
    private Clock _clock;
    public void init() {
        super.init();
        try {
            _clock = new Clock(_toplevel,"clock");
            TimedPlotter plotter = new TimedPlotter(_toplevel,"plotter");
            plotter.setPanel(this);
            plotter.plot.setTitle("clock signal");
            plotter.plot.setXLabel("time");
            plotter.plot.setImpulses(true);
            plotter.plot.setConnected(false);
            plotter.plot.setMarksStyle("dots");
            plotter.plot.setSize(700, 250);
            _toplevel.connect(_clock.output, plotter.input);
            add(_createRunControls(2));
            _query.setBackground(getBackground());
            _query.addLine("period", "Period", "2.0");
            _query.addQueryListener(this);
            add(_query);
        } catch (Exception ex) {
            report("Error constructing model.", ex);
        }
    }
    public void changed(String name) {
        _clock.period.setExpression(_query.stringValue("period"));
        try {
            _go();
        } catch (Exception ex) {
            report("Error executing model.", ex);
        }
    }
}
```

FIGURE 2.12. An applet that extends that in figure 2.10 by configuring the plotter. This code can be found in $PTII/doc/tutorial/TutorialApplet6.java.

```
plotter.plot.setTitle("clock signal");
plotter.plot.setXLabel("time");
plotter.plot.setImpulses(true);
plotter.plot.setConnected(false);
plotter.plot.setMarksStyle("dots");
```

The first line defines a title, and the second defines a label for the horizontal axis. The third requests a "stem plot" style, where a line is drawn vertically from the value to the origin. The fourth requests that events not be connected by lines, and the last requests that large dot be placed on each event. The resulting applet, as viewed by Sun's appletviewer, is shown in figure 2.13. A more detailed example is shown in the appendix.

2.2.8 Adding Custom Actors

The intent in 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 following 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.

2.2.9 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 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:

- 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 sophis-

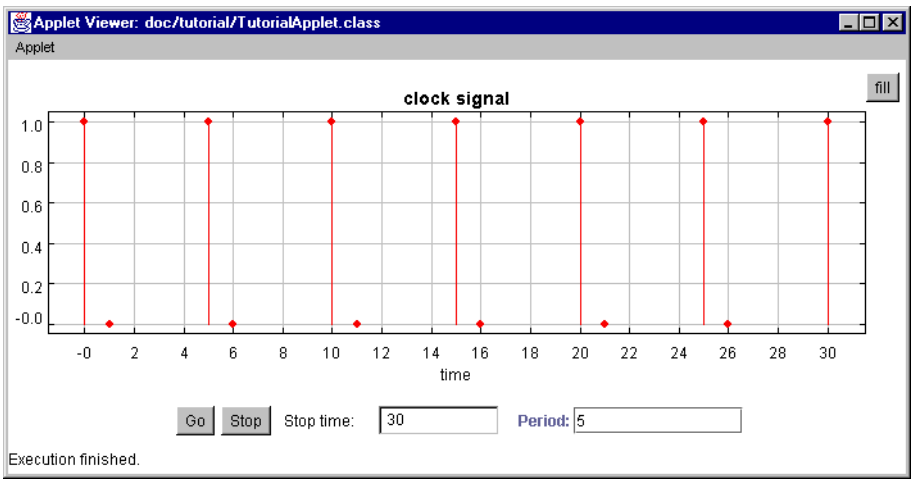


FIGURE 2.13. View of the applet in figure 2.12, as displayed by Sun's appletviewer.

ticated 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 2.2 to read as shown in figure 2.14.

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.

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
```

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
   width="700"
   height="300"
   codebase="http://java.sun.com/products/plugin/1.2/jinstall-12-win32.cab#Version=1,2,0,0">
<PARAM NAME="code" VALUE="doc.tutorial.TutorialApplet.class">
<PARAM NAME="codebase" VALUE="../..">
<PARAM NAME="archive" VALUE="
   ptolemy/ptsupport.jar,
   ptolemy/domains/de/de.jar">
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.2"
   width="700"
   height="300"
   code="doc/tutorial/TutorialApplet.class"
   codebase="../.."
   archive="
       ptolemy/ptsupport.jar,
       ptolemy/domains/de/de.jar"
   pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html">
</COMMENT>
<NOEMBED>
No JDK 1.2 support for applet!
</NOEMBED>
</EMBED>
</OBJECT>
```

FIGURE 2.14. An HTML segment that modifies that of figure 2.2 to use jar files. This text can be found in $PTII/doc/tutorial/tutorialJar.htm.

```
PTCLASSALLJARS = \
  event/event.jar \
  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 ptolemy/kernel/event/event.jar and ptolemy/kernel/util/util.jar.

2.2.10 Script for Creating Demo Applets

Most applet authors start with the same set of operations: creating an HTML file and a Java file defining a class that extends one of the domain-specific applet classes. To help get started, we have created $PTII/bin/ptmkmodel, a shell script that creates stub files for a model in the demo directory. This script is fairly specialized, supporting in particular the construction of Ptolemy II demos. For example, it assumes the applet will reside in the demo of the directory of the appropriate domain. Nonetheless, the script does illustrate several useful naming conventions, and may help some other authors avoid repetitive operations.

The script is invoked with two arguments, the *domain* and the *name* of the model. The *domain* must be a directory in $PTII/ptolemy/domains, and the *name* of the model must be a directory that does not yet exist in $PTII/ptolemy/domains/*domain*/demo/*name*. The Ptolemy naming convention for a model is that the model name starts with a capital letter and has embedded capital letters at the start of each word. The script creates the model directory and creates several files in the directory.

For example, to create a DE model called Ramp, one would run

```
$PTII/bin/ptmkmodel de Ramp
```

Which would create the following files:

- $PTII/ptolemy/domains/de/demo/Ramp/RampApplet.java — Java code defining the applet;
- $PTII/ptolemy/domains/de/demo/Ramp/Ramp.htm — HTML that sets the CLASSPATH and loads the applet; and
- $PTII/ptolemy/domains/de/demo/Ramp/makefile — A makefile with rules to build and run the applet.

To build and then run the applet, one would type:

```
cd $PTII/ptolemy/domains/de/demo/Ramp
make
make demo
```

Note that ptmkmodel has a number of limitations.

- The model that is created only works under the DE domain. For other domains, you will need to substitute in your own actors.
- The models are created in the demo directory under the domain.

You may wish to use this script as a starting point for your own, site-specific script.

2.2.11 Hints for Developing Applets

Unfortunately, Java plug-in technology is fairly immature. In the current version, we have encountered a number of problems, and have found workarounds that we can share.

Processes Linger After the Browser Quits. Sometimes, after running an applet under a browser, a process remains live even after the browser has been exited. This appears to be a bug with the just-in-time (JIT) compiler included in the plug-in. You can configure the plug-in to disable the JIT. On a Windows installation, the plug-in comes with a control panel (look under Programs in the Start menu). Select the "advanced" tab, and disable the JIT. This will noticeably slow down your applets, but you will not have to kill lingering processes.

Difficulty Reloading Applets. While developing applets, it is helpful to be able to reload the applet after modifying the Java code. In Netscape, you must use Shift-reload, and in IE, Control-reload. Unfortunately, this does not always cause the applet to be reloaded. A workaround is described on the Sun website, http://java.sun.com/products/plugin/1.2/docs/controlpanel.html, which says:

> *"Cache JARs in memory. If this option is checked, the applet classes already loaded will be cached and reused when applet is reloaded. This allows much more efficient memory use. You should leave this option unchecked if you are debugging an applet or are always interested in loading the latest applet classes. The default setting is to cache JARs in memory. Warning: turning off this option makes it more likely that you will get OutOfMemoryErrors, as this reduces sharing of memory."*

This option is under the "basic" tab of the same control panel described in the previous hint.

Appendix C: Inspection Paradox Example

In this appendix, we show the code for a more detailed and more interesting applet in the DE domain. This applet illustrates the "inspection paradox," which briefly stated, says that the average amount of time you wait for Poisson arrivals is twice what you might intuitively expect.

C.1 Description of the Problem

The inspection paradox concerns Poisson arrivals of events. The metaphor used in this applet is that of busses and passengers. Passengers arrive according to a Poisson process. Busses arrive either at regular intervals or according to a Poisson process. The user selects from these two options by clicking the appropriate on-screen control. The user can also control the mean interarrival time of both busses and passengers.

The inspection paradox concerns the average time that a passenger waits for a bus (more precisely, the expected value). If the busses arrive at regular intervals with interarrival time equal to *T*, then the expected waiting time is *T*/2, which is perfectly intuitive. Counter intuitively, however, if the busses arrive according to a Poisson process with mean interarrival time equal to *T*, the expected waiting time is *T*, not *T*/2. These expected waiting times are approximated in this applet by the average waiting time. The applet also shows that actual arrival times for both passengers and busses, and the waiting time of each passenger.

The intuition that resolves the paradox is as follows. If the busses are arriving according to a Poisson process, then some intervals between busses are larger than other intervals. A particular passenger is more likely to arrive at the bus stop during one of these larger intervals than during one of the smaller intervals. Thus, the expected waiting time is larger if the bus arrival times are irregular.

This paradox is called the *inspection paradox* because the passengers are viewed as inspecting the Poisson process of bus arrivals.

The applet code is listed below, and is included in the demo directory of the DE domain. A browser window displaying the applet is shown in figure 2.15. In that figure, there are two plots, one showing the events in the model, namely arrivals of busses and passengers plus the waiting time of passengers as they board the busses. The lower plot shows a histogram of waiting times, which has an approximately exponential shape.

C.2 Observations

There are a number of interesting features of this applet. It includes an instance of Query, at the upper left, with four entries. The first two are entry boxes similar to that in figure 2.10. The third is a different style of entry called a check box. The fourth is a display-only entry that shows final results from execution of the model.

The mechanism used to obtain and display final results is interesting. First, notice below that the applet uses an instance of the Average actor to calculate the average waiting time of a passenger. The output of this actor is fed an instance of Recorder, an actor that simply stores the data it is given for later querying. The final value of the average waiting time is obtained in the executionFinished() method, which is invoked when the execution of the model is completed.

A second interesting feature of this applet is that in the _go() method, depending on the state of the check box query, the model may be modified structurally before it is executed. Subsequent chapters

will explain precisely the meaning of the methods that are used to accomplish this modification.

C.3 Code Listing

```
package ptolemy.domains.de.demo.Inspection;

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.util.Enumeration;

import ptolemy.kernel.*;
import ptolemy.kernel.util.*;
import ptolemy.data.*;
import ptolemy.actor.Manager;
import ptolemy.actor.lib.*;
import ptolemy.actor.gui.*;
import ptolemy.gui.Query;
import ptolemy.gui.QueryListener;
import ptolemy.domains.de.kernel.*;
```

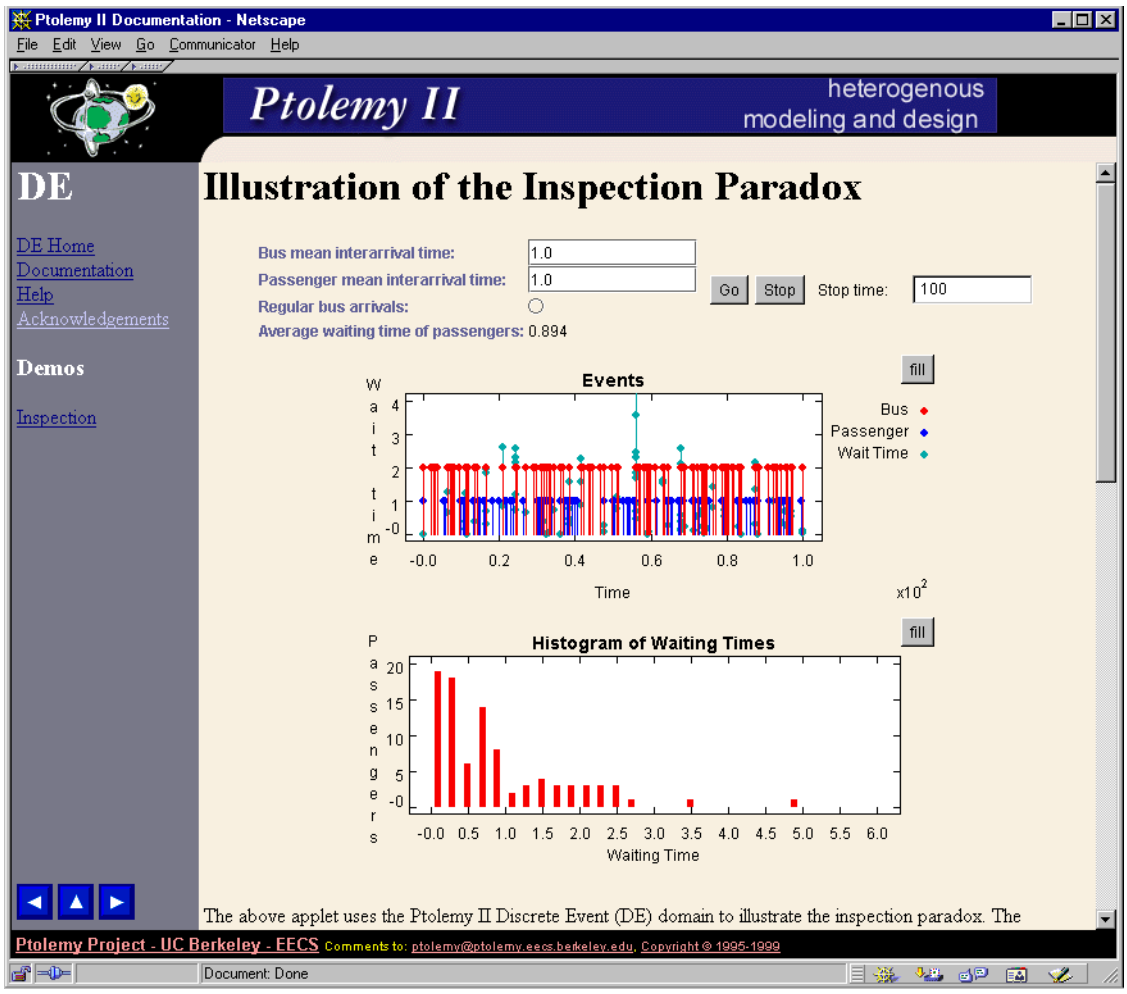


FIGURE 2.15. View of the inspection paradox applet described in the appendix.

```
import ptolemy.domains.de.lib.*;
import ptolemy.domains.de.gui.DEApplet;
import ptolemy.plot.*;

/** An applet that uses Ptolemy II DE domain to illustrate the inspection paradox.
@author Edward A. Lee and Lukito Muliadi
*/
public class InspectionApplet extends DEApplet implements QueryListener {

    ////////////////////////////////////////////////////////////////
    ////                         public methods                 ////

    /** If the argument is the string "regular", then set the
     *  variable that controls whether bus arrivals will be regular
     *  or Poisson.  If the argument is anything else, update the
     *  parameters of the model from the values in the query boxes.
     *  @param name The name of the entry that changed.
     */
    public void changed(String name) {
        if (name == "regular") {
            _regular = _query.booleanValue("regular");
        }
        try {
            if (_regular) {
                _regularBus.period.setToken(new DoubleToken(_query.doubleValue("busmean")));
            } else {
                _poissonBus.meanTime.setToken(new DoubleToken(_query.doubleValue("busmean")));
                _passenger1.meanTime.setToken(new DoubleToken(_query.doubleValue("passmean")));
            }
            _go();
        } catch (IllegalActionException ex) {
            throw new InternalErrorException(ex.toString());
        }
    }

    /** Override the base class to display the recorded average.
     */
    public void executionFinished(Manager manager) {
        super.executionFinished(manager);
        _query.setDisplay("average", _recorder.getLatest(0));
    }

    /** Initialize the applet.
     */
    public void init() {
        super.init();
        try {
            _query = new Query();
            _query.setBackground(_getBackground());
            _query.addLine("busmean", "Bus mean interarrival time", "1.0");
            _query.addLine("passmean", "Passenger mean interarrival time", "1.0");
            _query.addCheckBox("regular", "Regular bus arrivals", false);
            _query.addDisplay("average", "Average waiting time of passengers", "");
            add(_query);
            _query.addQueryListener(this);

            // The 2 argument requests a go and stop button.
            add(_createRunControls(2));

            if (_regular) {
                // Create regular bus source.
                _regularBus = new Clock(_toplevel, "regularBus");
                // Create Poisson bus source, but then remove from the container,
                // since it won't be used unless the user toggles the button.
                _poissonBus = new Poisson(_toplevel,"poissonBus");
                _poissonBus.setContainer(null);
            } else {
                // Create Poisson bus source.
```

```
            _poissonBus = new Poisson(_toplevel, "poissonBus");
            // Create regular bus source, but then remove from the container,
            // since it won't be used unless the user toggles the button.
            _regularBus = new Clock(_toplevel,"regularBus");
            _regularBus.setContainer(null);
        }
        // Set default parameters for both sources.
        _regularBus.values.setExpression("[2]");
        _regularBus.period.setToken(new DoubleToken(1.0));
        _regularBus.offsets.setExpression("[0.0]");
        _poissonBus.values.setExpression("[2]");
        _poissonBus.meanTime.setToken(new DoubleToken(1.0));

        // Create and configure the passenger source.
        _passenger1 = new Poisson(_toplevel, "passenger");
        _passenger1.values.setExpression("[1]");
        _passenger1.meanTime.setToken(new DoubleToken(1.0));

        // Waiting time.
        _wait = new WaitingTime(_toplevel, "waitingTime");

        // Average actor.
        Average average = new Average(_toplevel, "average");

        // Record the average.
        _recorder = new Recorder(_toplevel, "recorder");

        // Create and configure a plotter.
        _eventplot = new TimedPlotter(_toplevel, "plot");
        _eventplot.setPanel(this);
        _eventplot.plot.setGrid(false);
        _eventplot.plot.setTitle("Events");
        _eventplot.plot.addLegend(0, "Bus");
        _eventplot.plot.addLegend(1, "Passenger");
        _eventplot.plot.addLegend(2, "Wait Time");
        _eventplot.plot.setXLabel("Time");
        _eventplot.plot.setYLabel("Wait time");
        _eventplot.plot.setXRange(0.0, _getStopTime());
        _eventplot.plot.setYRange(0.0, 4.0);
        _eventplot.plot.setSize(450, 200);
        _eventplot.plot.setConnected(false);
        _eventplot.plot.setImpulses(true);
        _eventplot.plot.setMarksStyle("dots");
        _eventplot.fillOnWrapup.setToken(new BooleanToken(false));

        // Create and configure a histogram.
        _histplot = new HistogramPlotter(_toplevel, "histplot");
        _histplot.setPanel(this);
        _histplot.histogram.setGrid(false);
        _histplot.histogram.setTitle("Histogram of Waiting Times");
        _histplot.histogram.setXLabel("Waiting Time");
        _histplot.histogram.setYLabel("Passengers");
        _histplot.histogram.setXRange(0.0, 6.0);
        _histplot.histogram.setYRange(0.0, 20.0);
        _histplot.histogram.setSize(450, 200);
        _histplot.histogram.setBinWidth(0.2);
        _histplot.fillOnWrapup.setToken(new BooleanToken(false));

        // Connections, except the bus source, which is postponed.
        _busRelation = _toplevel.connect(_wait.waitee, _eventplot.input);
        ComponentRelation rel2 = _toplevel.connect(_passenger1.output, _eventplot.input);
        _wait.waiter.link(rel2);
        ComponentRelation rel3 = _toplevel.connect(_wait.output, _eventplot.input);
        _histplot.input.link(rel3);
        average.input.link(rel3);
        _toplevel.connect(average.output, _recorder.input);
        _initCompleted = true;
    } catch (Exception ex) {
```

```
            report("Setup failed:", ex);
        }
    }

    ////////////////////////////////////////////////////////////////
    ////                         protected methods                ////

    /** Execute the model.  This overrides the base class to read the
     *  values in the query box first and set parameters.
     *  @exception IllegalActionException If the topology changes or the
     *   model or parameter changes on the actors throw it.
     */
    protected void _go() throws IllegalActionException {
        // If an exception occurred during initialization, then we don't
        // want to run here.  The model is probably not complete.
        if (!_initCompleted) return;

        // If the manager is not idle then either a run is in progress
        // or the model has been corrupted.  In either case, we do not
        // want to run.
        if (_manager.getState() != _manager.IDLE) return;

        // Depending on the state of the radio button, we may want
        // either regularly spaced bus arrivals, or Poisson arrivals.
        // Here, we alter the model topology to implement one or the other.
        if (_regular) {
            try {
                _poissonBus.setContainer(null);
                _regularBus.setContainer(_toplevel);
            } catch (NameDuplicationException ex) {
                throw new InternalErrorException(ex.toString());
            }
            _regularBus.period.setToken
                    (new DoubleToken(_query.doubleValue("busmean")));
            _regularBus.output.link(_busRelation);
        } else {
            try {
                _regularBus.setContainer(null);
                _poissonBus.setContainer(_toplevel);
            } catch (NameDuplicationException ex) {
                throw new InternalErrorException(ex.toString());
            }
            _poissonBus.meanTime.setToken(new DoubleToken(_query.doubleValue("busmean")));
            _passenger1.meanTime.setToken(new DoubleToken(_query.doubleValue("passmean")));
            _poissonBus.output.link(_busRelation);
        }

        // The the X range of the plotter to show the full run.
        // The method being called is a protected member of DEApplet.
        _eventplot.plot.setXRange(0.0, _getStopTime());

        // Clear the average display.
        _query.setDisplay("average", "");

        // The superclass sets the stop time of the director based on
        // the value in the entry box on the screen.  Then it starts
        // execution of the model in its own thread, leaving the user
        // interface of this applet live.
        super._go();
    }

    ////////////////////////////////////////////////////////////////
    ////                         private variables                ////

    // Actors in the model.
    private Query _query;
    private Poisson _poissonBus;
    private Clock _regularBus;
```

```
    private Poisson _passenger1;
    private TimedPlotter _eventplot;
    private HistogramPlotter _histplot;
    private WaitingTime _wait;

    // An indicator of whether regular or Poisson bus arrivals are desired.
    private boolean _regular = false;

    // The relation to which links are made and unmade in response to
    // changes in the radio button state that selects regular or Poisson
    // bus arrivals.
    private ComponentRelation _busRelation;

    // Flag to prevent spurious exception being thrown by _go() method.
    // If this flag is not true, the _go() method will not execute the model.
    private boolean _initCompleted = false;

    // The observer of the average.
    private Recorder _recorder;
}
```

3

Actor Libraries

Authors: Edward A. Lee
Yuhong Xiong

3.1 Overview

Ptolemy II is about component-based design. Components are aggregated and interconnected to construct a model. One of the advantages of such an approach to design is that re-use of components becomes possible. In Ptolemy II, re-use potential is maximized through the use of polymorphism. 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 for vectors vs. scalars.

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, or asynchronous message passing.

Ptolemy includes libraries of polymorphic actors using both kinds of polymorphism to maximize re-usability. 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.

3.2 Library Organization

Two key libraries of domain-polymorphic actors are provided by Ptolemy II. The actors with graphical user interface (GUI) functions are collected in the ptolemy.actor.gui package, and the rest are in ptolemy.actor.lib. Domain-specific actors are in ptolemy.domains.*x*.lib, where "*x*" is the domain

name.

3.2.1 Actor.Lib

Figure 3.1 shows a UML static structure diagram for the ptolemy.actor.lib package (see appendix A of chapter 1 for an introduction to UML). All the classes in figure 3.1 extend TypedAtomicActor, except TimedActor and SequenceActor, which are interfaces. TypedAtomicActor is in the ptolemy.actor package, and is described in more detail in the Actor chapter. For our purposes here, it is sufficient to know that it provides a base class for actors with ports where the ports carry typed data (encapsulated in objects called *tokens*).

The grey boxes in figure 3.1 are not standard UML. They are used here to aggregate actors with common inheritance, or to refer to a set of actors (sources and sinks) that are enumerated later in this chapter.

None of the classes in figure 3.1 have any methods, except those inherited from the base classes, which are not shown. By convention, actors in Ptolemy II expose their ports and parameters as public

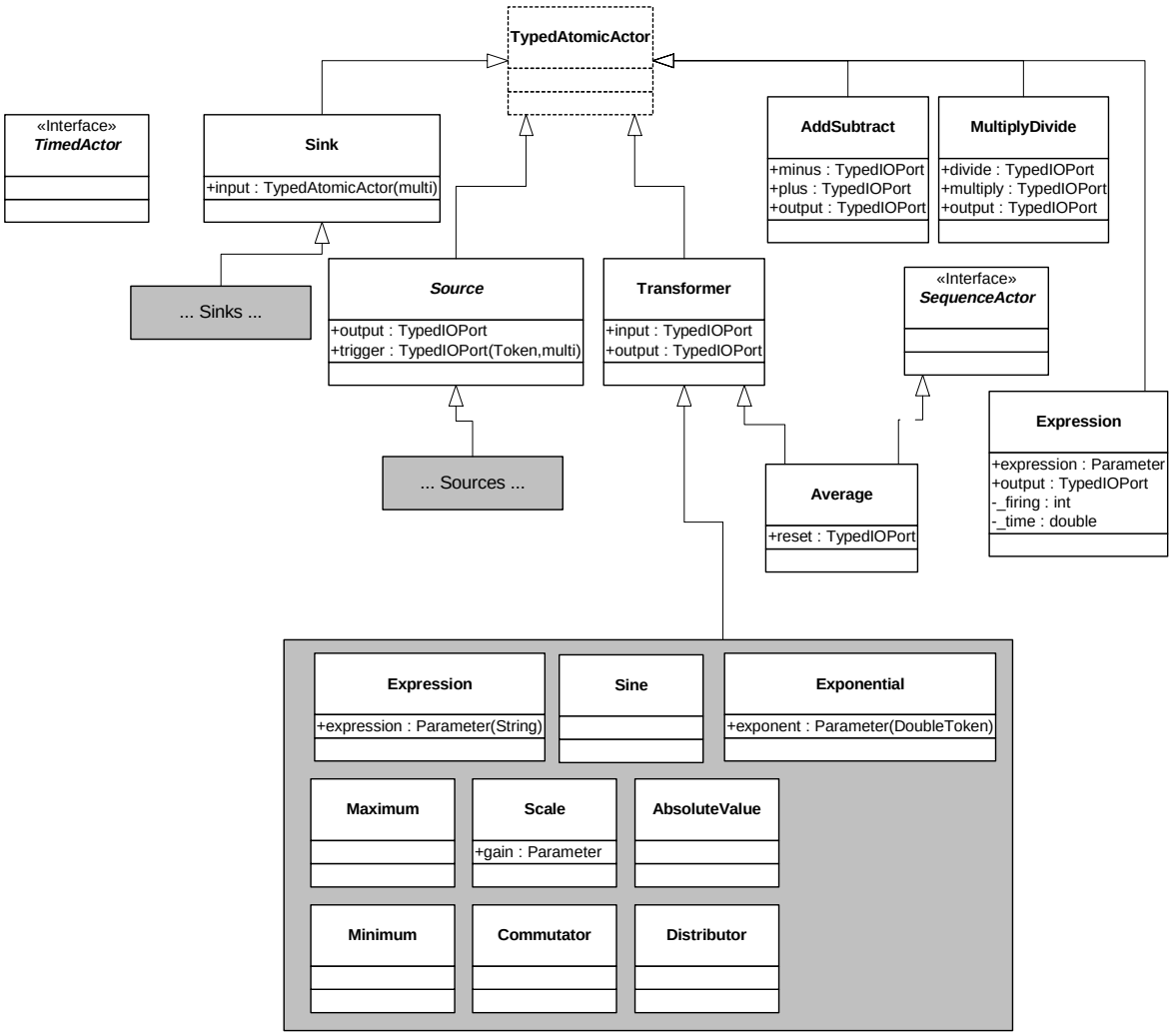


FIGURE 3.1. Organization of actors in the ptolemy.actor.lib package.

members, and much of the functionality of an actor is accessed through the ports and parameters.

Many of the actors in this package are *transformers*, which extend the Transformer class. These actors read input data, modify it in some way, and produce output data. AddSubtract, MultiplyDivide, and Expression are also transformers, but they do not extend Transformer because they have somewhat unconventional port names (or in the case of Expression, no pre-defined ports at all).

The interfaces TimedActor and SequenceActor play a very important role in this library. They are empty interfaces (no methods), and hence are used only as markers. An actor that implements SequenceActor declares that its inputs are assumed to be sequences of distinct data values, and that the outputs it produces will be sequences of distinct data values. Thus, for example, the Average actor computes a running average of the input tokens. By contrast, Sine does not implement SequenceActor, because it does not care whether the input is a sequence. In particular, the order in which inputs are presented is irrelevant, and whether a particular input is presented more than once is irrelevant. Implementing the TimedActor interface declares that the current time in a model execution affects the behavior of the actor.

Some domains can only execute a subset of the actors in this library. In particular, some domains cannot handle actors that implement SequenceActor. For example, the CT domain (continuous time), which solves ordinary differential equations, may present data to actors that represents arbitrarily closely spaced samples of a continuous-time signal. Thus, the data presented to an actor cannot be viewed as a sequence, since the domain determines how closely spaced the samples are. For example, the Average actor would produce unpredictable results, since the spacing of samples is likely to be uneven over time. Thus, it is up to the director in the CT domain to reject actors that implement SequenceActor.

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 domain (synchronous dataflow), for example, does not advance current time. Thus, if it is the top-level domain, current time will always be zero, which is likely to lead to some confusion with timed actors.

3.2.2 Actor.GUI

Figure 3.2 shows a UML static structure diagram for the ptolemy.actor.gui package. These actors all have graphical user interface (GUI) functions. The TimedPlotter, for example, which was used in the previous chapter, 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, by contrast, uses neither time nor the sequence number, and therefore implements neither TimedActor nor SequenceActor. All three are derived from Plotter, an abstract base class with a public member, *plot*, which implements the plot.

This package includes the Placeable interface, discussed in the previous chapter. Actors that implement this interface have graphical widgets that a user of the actor may wish to place on the screen. The setPanel() method is used to specify where to place the graphical element.

3.3 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 3.3 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.

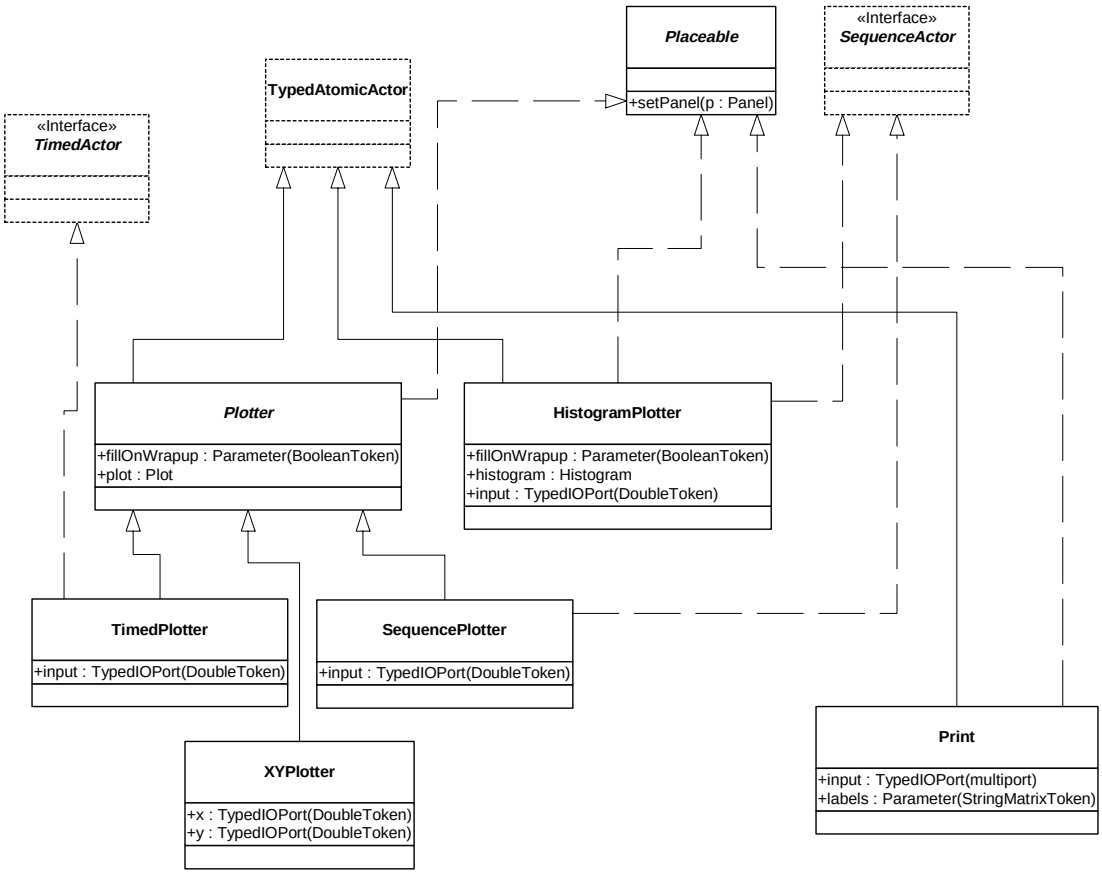


FIGURE 3.2. Organization of actors in the ptolemy.actor.gui package.

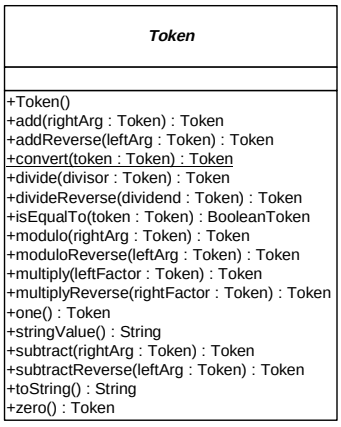


FIGURE 3.3. The Token class defines a polymorphic interface that includes basic arithmetic operations.

The fire method of the AddSubtract actor is shown in figure 3.4. 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.

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 had better never 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.

3.4 Domain Polymorphism

Most actors access their ports as shown in figure 3.4, using the hasToken(), get(), and broadcast() 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 broadcast() method on one of its ports). Correspondingly, a call to broadcast() causes the calling thread to suspend until some other thread calls a corresponding get(). In the PN domain, by contrast, broadcast() 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(), broadcast() and other methods of ports. The actor, however, does not really care. The fire() method shown in figure 3.4

```
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.broadcast(sum);
   }
}
```

FIGURE 3.4. The fire() method of the AddSubtract shows the use of polymorphic add() and subtract() methods in the Token class (see figure 3.3).

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 simplest domain polymorphic actors are *stateless*, meaning that they store no data from one firing to the next. Such actors might also be called *functional*, since the output is a function of the input (only). For such actors, it is evident that when the fire() method is invoked is not important. The AddSubtract actor, for example, simply operates on whatever inputs are available. To understand how actors with state behave, we need to explain how actors are invoked.

3.4.1 Iterations

Invocation of actors in all domains follows a particular sequence. First, the initialize() method is invoked, exactly once. This method is invoked prior to type resolution, so the types of the ports may not have yet been determined. At the end of the execution, the wrapup() method is invoked, again exactly once (unless an exception occurs, in which case it may not be invoked at all).

An *iteration* of an actor is defined to be one invocation of prefire(), any number of invocations of fire(), and one invocation of postfire(). An *execution* is defined to be one invocation of initialize(), followed by any number of iterations, followed by one invocation of wrapup(). The methods initialize(), prefire(), fire(), postfire(), and wrapup() are called the *action methods*.

For more details, see the Actor Package chapter.

3.4.2 Domains with Fixed Point Semantics

The reason for allowing an iteration to consist of any number of invocations of the fire() method is that some domains have *fixed-point semantics*. This means that the tokens produced at the output converge during an iteration to a final, correct value. Early in an iteration the output values may be approximations, or may be absent. The semantics of the domain is that only the last outputs produced in the iteration are correct.

For example, directors in the CT domain, which models continuous-time systems, begin an iteration by estimating the time step that the iteration should take to get reasonable accuracy. They invoke the prefire() method of all actors, then the fire() method. They then query the actors to determine whether the results were sufficiently accurate. Domain-polymorphic actors are always content with the result, but some domain-specific actors may respond with “no, the time step was too big.” The director then needs to recalculate the time step, and re-invoke the fire() methods of all the actors. Once all actors respond that the results are acceptable, the postfire() methods of all the actors are invoked.

The notion that the actors can specify whether the result is acceptable means that an iteration concludes when all actors are content (the solution has converged). That is, given the observed inputs, the output produced by each actor is correct. This is called a *fixed point* by analogy with the following mathematical equation:

$$f(x) = x. \tag{1}$$

A solution *x* to this equation is called a fixed point of the function *f*. A candidate solution *x* is "accept able" to the function if when presented as an argument to the function, the result of the function is equal to *x*. In general, *x* may be a vector. The analogy, therefore, is that *x* is a vector of values of all signals at the conclusion of an iteration, and *f* is the collective effect of all the actors. The values in the vector *x* are acceptable if all actors find their current outputs consistent with their current inputs.

3.4.3 Actors with State

The fact that the fire() method may be invoked with approximate inputs, and may be permitted to produce approximate outputs has fairly profound implications on the design of domain-polymorphic actors. In particular, if the actor maintains state information, then it should not update that state until the postfire() method, after convergence has been reached.

It is instructive to examine the fire() and postfire() methods of the Average actor, shown in figure 3.5. This actor extends Transformer, which provides ports named *input* and *output*. It adds an additional port called *reset*. Its state is stored as private variables, _sum and _count. The _sum variable is the sum of all inputs it has seen. Clearly, it should sum only input values that are the final values of an iteration, not approximate or tentative values. The _count variable counts the number of iterations, which in general may be fewer than the number of invocations of the fire() method.

The general strategy here is to create two shadow variables, `_latestSum` and `_latestCount`, and update these in the fire() method. The postfire() method then simply copies the values of these variables into the permanent state variables, _sum and _count.

Examining the fire() method, we see that the first thing it does is to set _latestSum and _latestCount equal to the permanent state variables. Thus, if the fire() method has been previously invoked in this iteration, the results of that invocation are now discarded.

The fire() method then checks to see whether there is a *reset* input, and if there is, it sets the shadow variables accordingly, as if this were the first input being seen. It then obtains an input, and uses the polymorphic add() method of the input token to add the input to the shadow sum. Finally, it uses the polymorphic divide() method of the input token to divide by the total number of inputs seen since the last reset.

3.5 Descriptions of Libraries

Here we briefly describe the actors in the ptolemy.actor.lib and ptolemy.actor.gui libraries. This should be viewed as a summary only. Refer to the class documentation for a complete description of these actors. The summary is useful, however, because these actors are carefully designed to be extensively re-usable. Some general terms that may be useful in interpreting the descriptions are:

lub: Least upper bound, referring particularly to data types. For typical data polymorphic actors, the output data type is the lub of the input data types. This means that each input data type can be losslessly converted to the type of the output. In some cases, the output data type also depends on the type of parameters. See the Type System chapter for more detail.

multiport: A port that links to any number of channels. Ports described below are multiports only if they say so explicitly. Multiports can be left disconnected in all domains, in which case no inputs are read. Multiports resolve to a single data type, so all channels must have the same data type.

```
public class Average extends Transformer {

   public TypedIOPort reset;

   private Token _sum;
   private Token _latestSum;
   private int _count = 0;
   private int _latestCount;

   public void initialize() throws IllegalActionException {
        super.initialize();
        _count = 0;
        _sum = null;
   }

   public void fire() throws IllegalActionException {
        try {
            _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(r.booleanValue()) {
                        // Being reset at this firing.
                        _latestSum = null;
                        _latestCount = 1;
                    }
                }
            }
            Token in = input.get(0);
            if (_latestSum == null) {
                _latestSum = in;
            } else {
                _latestSum = _latestSum.add(in);
            }
            Token out = _latestSum.divide(new IntToken(_latestCount));
            output.broadcast(out);
        } catch (IllegalActionException ex) {
            // Should not be thrown because this is an output port.
            throw new InternalErrorException(ex.getMessage());
        }
   }

   public boolean postfire() throws IllegalActionException {
        _sum = _latestSum;
        _count = _latestCount;
        return super.postfire();
    }

   ...
}
```

FIGURE 3.5. The fire() and postfire() methods of the Average actor show how state is updated only in post-fire().

It is also 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 implement neither SequenceActor nor TimedActor.

3.5.1 Functional Actors

The functional actors in the ptolemy.actor.lib are shown in figure 3.1. They are summarized here.

AbsoluteValue

Ports: *input* (DoubleToken), *output* (DoubleToken).

Produce an output token on each firing with a value that is equal to the absolute value of the input.

AddSubtract

Ports: *plus* (multiport, polymorphic), *minus* (multiport, polymorphic), *output* (lub(*plus*, *minus*)).

Add tokens on the *plus* input and subtract tokens on the *minus* input.

Maximum

Ports: *input* (multiport, DoubleToken), *output* (DoubleToken).

Produce an output token on each firing with a value that is the maximum of the input values.

Minimum

Ports: *input* (multiport, DoubleToken), *output* (DoubleToken).

Produce an output token on each firing with a value that is the minimum of the input values.

MultiplyDivide

Ports: *multiply* (multiport, polymorphic), *divide* (multiport, polymorphic), *output* (lub(*multiply*, *divide*)).

Multiply tokens on the *multiply* input and divide by tokens on the *divide* input.

Quantizer

Ports: *input* (DoubleToken), *output* (DoubleToken).
Parameters: *levels* (DoubleMatrixToken).

Produce an output token with the value in *levels* that is closest to the input value.

Scale

Ports: *input* (polymorphic), *output* (lub(*input*, *gain*)).
Parameters: *gain* (polymorphic).

Produce an output that is the product of the *input* and the *gain*.

Sine

Ports: *input* (DoubleToken), *output* (DoubleToken).
Parameters: *amplitude* (DoubleToken), *omega* (DoubleToken), *phase* (DoubleToken).

Produce an output token with value equal to *amplitude* $\times$ sin((*omega* $\times$ *input*) + *phase*). Note that this can be used to compute a cosine by setting *phase* to -PI/2.

3.5.2 Polymorphic Sources

Source actors are shown in figure 3.6. All of these actors have a *trigger* input, which is a multiport specifically so that it can be left disconnected in all domains (if disconnected, it has width zero). The trigger input can be used to force an output in domains where the firing of an actor is driven by input data. In DE, for example, it causes the current value of the source to be produced at the time stamp of the trigger input. In SDF and PN, if the *trigger* input is not connected, then the actor simply fires often enough to supply the destination actors with tokens.

Those sources that implement TimedActor have a parameter *stopTime*. When the current time of the model reaches this time, then postfire() returns false, requesting of the director that this actor not be invoked again. This can be used to generate a finite source signal. By default, this parameter has value 0.0, which indicates unbounded execution.

Some of these source actors use the fireAt() method of the director to request firing at particular times. Such actors will fire repeatedly even if there is no trigger input, even in domains (like DE) that fire actors only in response to input events. The fireAt() method schedules an event in the future to refire the actor.

Those sources that implement SequenceActor have a parameter *firingCountLimit*. When the number of iterations of the actor reaches this limit, then postfire() returns false, requesting of the director that this actor not be invoked again. This can be used to generate a finite source signal. By default, this

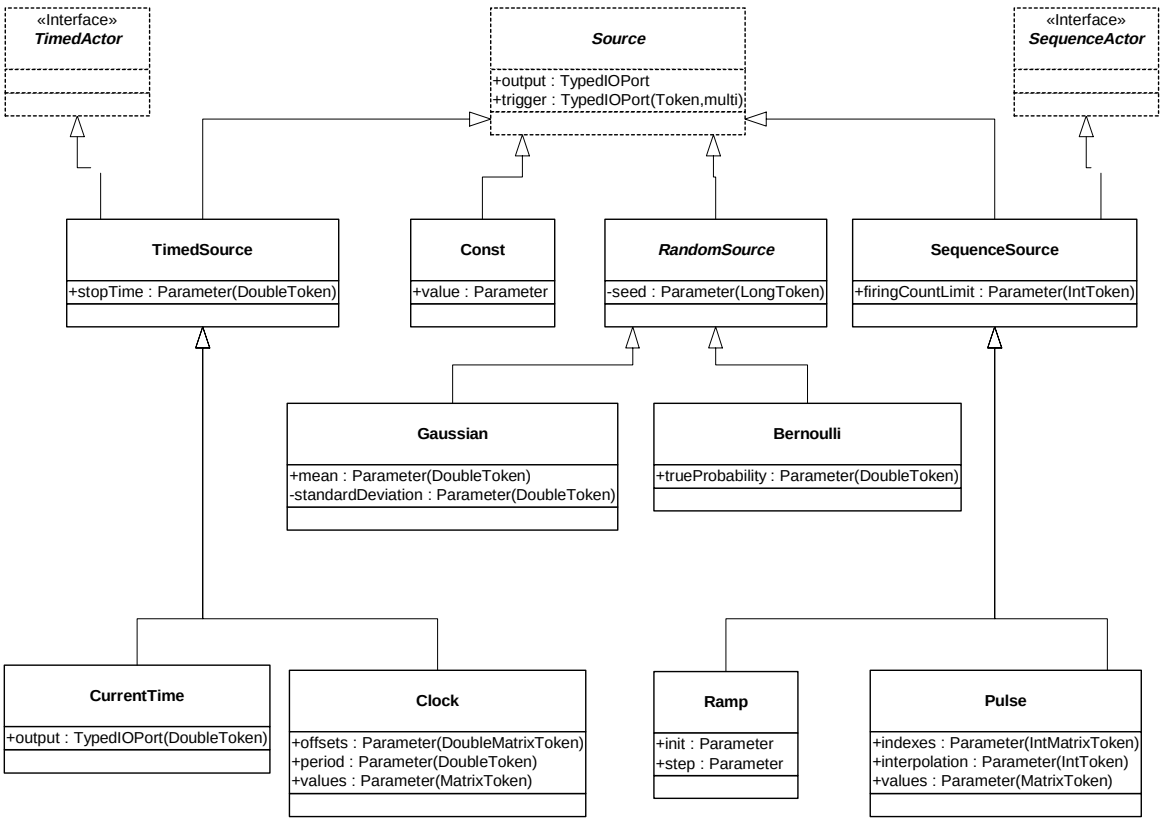


FIGURE 3.6. Source actors in the ptolemy.actor.lib package.

parameter has value 0, which indicates unbounded execution.

Bernoulli
Ports: *trigger* (input multiport, Token), *output* (BooleanToken).
Parameters: *trueProbability* (DoubleToken), *seed* (LongToken).

Produce a random sequence of booleans (a source of coin flips).

Clock (implements TimedActor)
Ports: *trigger* (input multiport, Token), *output* (lub(elements of *values*)).
Parameters: *offsets* (DoubleMatrixToken), *period* (DoubleToken), *values* (MatrixToken), *stopTime* (DoubleToken).

Produce a piecewise-constant, periodic signal (or at minimum, a sequence of events corresponding to transitions in this signal). This actor uses fireAt() to schedule firings when time matches the transition times.

Const
Ports: *trigger* (input multiport, Token), *output* (type of *value*).
Parameters: *value* (polymorphic).

Produce a constant output with value given by *value*.

CurrentTime (implements TimedActor)
Ports: *trigger* (input multiport, Token), *output* (DoubleToken).
Parameters: *stopTime* (DoubleToken).

Produce an output token with value equal to the current time.

Gaussian
Ports: *output* (DoubleToken).
Parameters: *mean* (DoubleToken), *standardDeviation* (DoubleToken), *seed* (LongToken).

Produce a random sequence where each output is the outcome of a Gaussian random variable.

Poisson
Ports: *trigger* (input multiport, Token), *output* (lub(elements of *values*)).
Parameters: *meanTime* (DoubleToken), *values* (MatrixToken), *stopTime* (DoubleToken).

Produce a piecewise-constant signal where transitions occur according to a Poisson process (or at minimum, a sequence of events corresponding to transitions in this signal). This actor uses fireAt() to schedule firings at time intervals determined by independent and identically distributed exponential random variables with mean *meanTime*.

Pulse (implements SequenceActor)
Ports: *trigger* (input multiport, Token), *output* (lub(elements of *values*)).
Parameters: *indexes* (IntMatrixToken), *values* (MatrixToken), *firingCountLimit* (IntToken)

Produce a sequence of values at specified iterations.

Ramp (implements SequenceActor)

Ports: *trigger* (input multiport, Token), *output* (lub(*init*, *step*)).
Parameters: *init* (polymorphic), *step* (polymorphic), *firingCountLimit* (IntToken)

Produce a sequence that begins with the value given by *init* and is incremented by *step* after each iteration.

3.5.3 Polymorphic Sinks and Displays

The following actors are in the ptolemy.actor.lib package (see figure 3.7) if they have no graphical component, and in ptolemy.actor.gui (see figure 3.8) otherwise. Several of the plotters have a *fillOnWrapup* parameter, which has a boolean value. If the 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.

FileWrite

Ports: *input* (multiport, Token).
Parameters: *filename* (StringToken).

Write the string representation of input tokens to the specified file or to standard out.

HistogramPlotter

Ports: *input* (multiport, DoubleToken).
Parameters: *fillOnWrapup* (BooleanToken).

Display a histogram of the data on each input channel.

Print

Ports: *input* (multiport, Token).

Display a string representation of the data on each input channel in a text area on the screen.

Recorder

Ports: *input* (multiport, Token).

Record the inputs for later querying. This actor is used extensively in the Ptolemy II test suites, and can also be used to collect data for display at the conclusion of an execution.

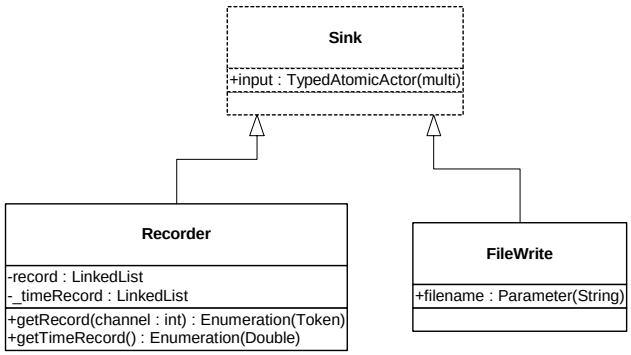


FIGURE 3.7. Sink actors in the ptolemy.actor.lib package.

SequencePlotter

Ports: *input* (multiport, DoubleToken).
Parameters: *fillOnWrapup* (BooleanToken).

Display a plot of the data on each input channel vs. the iteration count for the actor.

TimedPlotter

Ports: *input* (multiport, DoubleToken).
Parameters: *fillOnWrapup* (BooleanToken).

Display a plot of the data on each input channel vs. the current time when the data is consumed.

XYPlotter

Ports: *inputX* (multiport, DoubleToken), *inputY* (multiport, DoubleToken).
Parameters: *fillOnWrapup* (BooleanToken).

Display a plot of the data on each *inputX* channel vs. the data on the corresponding *inputY* channel.

3.5.4 Expression Actor

One particularly powerful actor is the Expression actor, which can have any number of input ports, and produces an output that is an arbitrary expression involving the inputs. The expression just refers

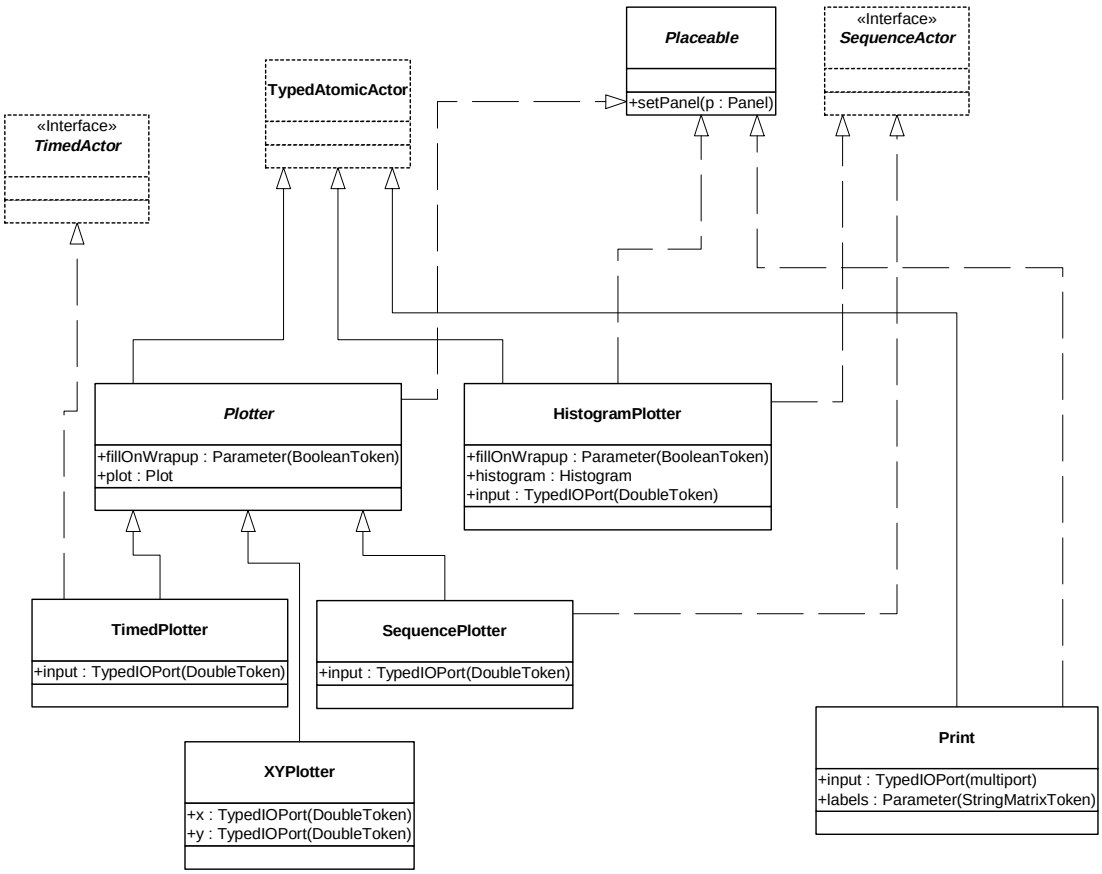


FIGURE 3.8. Display actors in the ptolemy.actor.gui package.

to the inputs by the name of the port. The expression language is described in the Data Package chapter. Notice that by default, there are no input ports. The user of this actor must create ports (instances of TypedIOPort) and add them to this actor by calling their setContainer() method. The expression can also refer to current time by the variable “time” and to the current iteration count by the variable “iteration.”

Expression

Ports: *output* (polymorphic).
Parameters: *expression* (polymorphic).

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.

3.5.5 Other Actors

Average (implements SequenceActor)

Ports: *input* (polymorphic), *output* (type of *input*), *reset* (multiport, BooleanToken).

Produce on each firing the average of all the inputs received since the last *true* on *reset*. The *reset* input may be left disconnected.

Commutator (implements SequenceActor)

Ports: *input* (multiport, polymorphic), *output* (type of *input*).

Interleave the data on the input channels into a single sequence on the output.

Distributor (implements SequenceActor)

Ports: *input* (polymorphic), *output* (multiport, type of *input*).

Distribute the data on the input sequence into a multiple sequences on the output channels.

4

Designing Actors

Author: Edward A. Lee

4.1 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 various domains. Others are domain specific. Refer to the domain chapter 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. This minimizes the amount of duplicated code between domains.

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. This chapter explains how to design actors so that they are maximally data polymorphic. This minimizes the amount of duplicated code in actor libraries.

Code duplication can be also be avoided using object-oriented inheritance. Inheritance can also be used to enforce consistency across a set of classes. Figure 3.1, which shows a UML static-structure diagram for an actor library, shows three base classes, Source, Sink, and Transformer, which exist to ensure consistent naming of ports and to avoid duplicating code associated with those ports (although there is not much code there). The fact that these three classes are the base classes for many of the actors in the library means that a user of the library can guess that an input port is named “input” and an output port is named “output,” and he will probably be right. It is unlikely that an input port would be 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.

4.2 Anatomy of an Actor

The basic structure of an actor is shown in figure 4.1. 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. Italics is used to indicate text that would be substituted with something else 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 resource.

4.2.1 Ports

By convention, ports are public members of actors. They mediate input and output. Figure 4.1 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 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*. If the port is not connected, the width is zero. If the port is a single port, the width can be zero or one. If the port is a multiport, the width can be larger than one. A multiport mediates communication over any number of *channels*.

Reading and Writing. Data (encapsulated in a *token*) can be sent to a particular channel of an output multiport 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 with the syntax

```
int width = portName.getWidth();
```

A token can be sent to all channels (or none if there are none) with the syntax

```
/** Javadoc comment for the class. */
public class ClassName extends BaseClass implements MarkerInterface {

   /** Javadoc comment for constructor. */
   public ClassName(TypedCompositeActor 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(DoubleToken.class);
   }

   ////////////////////////////////////////////////////////////////
   ////                     ports and parameters                 ////

   /** Javadoc comment for port. */
    public TypedIOPort portName;

   /** Javadoc comment for parameter. */
   public Parameter parameterName;

   ////////////////////////////////////////////////////////////////
   ////                         public methods                   ////

   /** Javadoc comment for method. */
    public Object clone(Workspace ws) throws CloneNotSupportedException {
      ClassName newobj = (ClassName)super.clone(ws);
      newobj.portName = (TypedIOPort)newobj.getPort("portName");
      newobj.parameterName = (Parameter)newobj.getAttribute("parameterName");
      return newobj;
   }

   /** 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 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 4.1. Anatomy of an actor.

```
portName.broadcast(token);
```

A token can be read from a channel with the syntax

```
Token token = portName.get(channelNumber);
```

You can query an input port to see whether such a get() will succeed (whether a token is available or can be made available) with the syntax

```
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.

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 unimorphic rather than polymorphic),

```
portName.setTypeEquals(DoubleToken.class);
```

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(ComplexToken.class);
```

which declares that any type that can be losslessly converted to ComplexToken is acceptable.

If no type constraints are given for any ports of an actor, then by default, the output ports are constrained to have at least the type(s) of the input ports. If *any* type constraints are given for any ports in the actor, then this default is not applied for any of the other ports. Thus, if you specify any type constraints, you should specify all of them. For full details of the type system, see the Type System chapter.

Examples. To be concrete, consider first the code segment shown in figure 4.2, 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

```
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(TypedCompositeActor 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 4.2. Code segment showing the port definitions in the Transformer class.

class comment. Derived classes may also change the type constraints of a port.

An extension of Transformer is shown in figure 4.3, the Scale actor. 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 *gain* parameter. In the constructor, the output type is constrained to be at least as general as both the input and the *gain* parameter.

Notice in figure 4.3 how the fire() method uses hasToken() to ensure that no output is produced if there is no input. 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 gain actor can operate on any token type that supports multiplication, including all the numeric types and matrices.

4.2.2 Parameters

Like ports, by convention, parameters are public members of actors. Figure 4.3 shows a parameter *gain* that is an instance of Parameter, declared in the line

```
public Parameter gain;
```

```
public class Scale extends Transformer {
   ...
   public Scale(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException  {
      super(container, name);
      gain = new Parameter(this, "gain", new IntToken(1));

      // set the type constraints.
      output.setTypeAtLeast(input);
      output.setTypeAtLeast(gain);
   }

   ////////////////////////////////////////////////////////////////
   ////                     ports and parameters                 ////

   /** The gain.  The default value of this parameter is the integer 1. */
   public Parameter gain;

   ////////////////////////////////////////////////////////////////
   ////                         public methods                   ////

   /** Compute the product of the input and the <i>gain</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 gainToken = gain.getToken();
           Token result = gainToken.multiply(in);
           output.broadcast(result);
       }
   }
}
```

FIGURE 4.3. Code segment from the Scale actor, showing the handling of ports and parameters.

1. Javadoc is a program that generates HTML documentation from Java files based on comments enclosed in "/** ... */".

and created in the line

```
gain = new Parameter(this, "gain", new IntToken(1));
```

The third argument to the constructor, which is optional, is a default value for the parameter. In this example, the *gain* parameter defaults to the integer one.

As with ports, you can specify type constraints on parameters. The most common type constraint is to fix the type, using

```
parameterName.setTypeEquals(DoubleToken.class);
```

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 4.4, taken from the Poisson 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. If this is *meanTime*, then this method checks to make sure that the specified value is positive, and if not, it throws an exception.

A change in a parameter value sometimes has broader repercussions than just the local actor. It may, for example, impact the schedule of execution of actors. An actor can call the invalidateSched-

```
public class Poisson extends TimedSource {

   public Parameter meanTime;

   public Poisson(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException  {
      super(container, name);
      meanTime = new Parameter(this, "meanTime", new DoubleToken(1.0));
      meanTime.setTypeEquals(DoubleToken.class);
      ...
   }

   /** 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 {
            super.attributeChanged(attribute);
      }
   }
   ...
}
```

FIGURE 4.4. Code segment from the Poisson actor, showing the attributeChanged() method.

ule() 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.

> **Note**: It may be tempting to override attributeChanged() to read the new parameter value and store it in a local (private) variable for more convenient access by other methods. This is ill-advised, however, at least at the time of this writing. A parameter may have a value that is given as an expression that references other parameter values. For example, parameter *a* might have expression "*b* + *c*." The expression language in Ptolemy II is *lazy*, meaning that the expression is evaluated only when the value is requested. Thus, if you read the parameter value in attributeChanged(), the expression will be evaluated using the current values of *b* and *c*. However, if *b* changes, the value of *a* will be updated only when next read. Only at that point will the attributeChanged() method be called. Thus, the actor will likely never see the change in *b*.[1]

By default, actors do not allow type changes on parameters. Once a parameter is given a value, then the value can change but not the type. Thus, the statement

```
meanTime.setTypeEquals(DoubleToken.class);
```

in figure 4.4 is actually redundant, since the type was fixed in the previous line,

```
meanTime = new Parameter(this, "meanTime", new DoubleToken(1.0));
```

However, some actors may wish to allow type changes in their parameters. Consider again the Scale actor, which is data polymorphic. The *gain* parameter can be of any type that supports multiplication, so type changes should be allowed.

To allow type changes in an actor, you must override the attributeTypeChanged() method. This method is defined in the based class for actors NamedObj, and by default, throws an exception. The method is called whenever the type of a parameter is changed. Consider figure 4.5, taken from the Scale actor. The first task of this method is to not throw an exception. This means that type changes are allowed. However, recall from figure 4.3 that a type constraint was specified that relates the input type to the gain. If the type changes, then the resolved type of the input may no longer be valid. The code in figure 4.5 notifies the director of this fact by calling its invalidateResolvedTypes() method.

4.2.3 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);
```

1. This design is debatable. Lazy evaluation has the key advantage that expressions are not immediately evaluated, and thus can be set in any order. In other words, a parameter can be given an expression value before the parameters it depends on are defined. This greatly simplifies storing and retrieving models in files, for examples. In principle, it is possible to notify all parameters whose values are expressions when the value of the expression might change, but this has the side effect of causing expressions to be evaluated immediately most of the time.

and that it declares that it throws NameDuplicationException and IllegalActionException. The latter is the most widely used exception, with most methods of actors having the possibility of throwing 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.

4.2.4 Cloning

All actors have the possibility of being cloned. A user interface, for example, may support "copy and paste" of actors, so the clone() method is provided to make this easy. 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 4.6, taken from the Scale actor. This method begins with

```
Scale newobj = (Scale)super.clone(ws);
```

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. So for example, the public member *gain* in an instance of Scale is copied into a clone of this instance, but the member in the new actor will reference *the same instance of Parameter as the original actor.* This is obviously not what we want. It would mean, for example, that if you change the *gain* of the clone, then the *gain* of the original actor will also change.

Actors are derived from a base class called NamedObj (see "The Kernel" chapter), which has a clone() method that clones all the parameters (Parameter is derived from Attribute, and instance of NamedObj can contain any number of instances of Attribute). Thus, a clone of an instance of Scale contains its own *gain* parameter, but the public member *gain* still does not reference it. This situation is corrected using the following line:

```
newobj.gain = (Parameter)newobj.getAttribute("gain");
```

```
public class Scale extends Transformer {
   ...
    /** Notify the director when a type change in the parameter occurs.
     *  This will cause type resolution to be redone at the next opportunity.
     *  It is assumed that type changes in the parameter are implemented
     *  by the director's change request mechanism, so they are implemented
     *  when it is safe to redo type resolution.
     *  If there is no director, then do nothing.
     */
    public void attributeTypeChanged(Attribute attribute) {
        Director dir = getDirector();
        if (dir != null) {
            dir.invalidateResolvedTypes();
        }
    }
   ...
}
```

FIGURE 4.5. Code segment from the Scale actor, showing the attributeChanged() method.

This looks up the parameter by name in the new object and sets the public member to refer to the correct parameter instance.

How type constraints are handled by clone() is a little bit subtle. Absolute type constraints on ports and parameters are carried automatically into the clone. Relative type constraints are not, however, because of the difficulty of ensuring that the other object being referred to in the relative constraint is the intended one. Thus, in figure 4.6, the clone() method repeats the relative type constraints that were specified in the constructor:

```
newobj.output.setTypeAtLeast(newobj.input);
newobj.output.setTypeAtLeast(newobj.gain);
```

Note that at no time during cloning is any constructor invoked, so it is necessary to repeat in the clone() method any initialization that will be automatically cloned.

Notice that in figure 4.6 the clone() method does nothing about the ports. This is because the clone method of the superclass, Transformer, takes care of this. Its clone method is:

```
public Object clone(Workspace ws) throws CloneNotSupportedException {
   Transformer newobj = (Transformer)super.clone(ws);
```

```
public class Scale extends Transformer {
   ...
   public Scale(TypedCompositeActor container, String name)
           throws NameDuplicationException, IllegalActionException  {
      super(container, name);
      gain = new Parameter(this, "gain", new IntToken(1));
      output.setTypeAtLeast(input);
      output.setTypeAtLeast(gain);
   }

   ////////////////////////////////////////////////////////////////
   ////                     ports and parameters                 ////

   /** The gain.  The default value of this parameter is the integer 1. */
   public Parameter gain;

   ////////////////////////////////////////////////////////////////
   ////                         public methods                   ////

   /** Clone the actor into the specified workspace. This calls the
    *  base class and then sets the type constraints.
    *  @param ws 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 ws) throws CloneNotSupportedException {
      Scale newobj = (Scale)super.clone(ws);
      newobj.gain = (Parameter)newobj.getAttribute("gain");
      newobj.output.setTypeAtLeast(newobj.input);
      newobj.output.setTypeAtLeast(newobj.gain);
      return newobj;
   }
   ...
}
```

FIGURE 4.6. Code segment from the Scale actor, showing the clone() method.

```
        newobj.input = (TypedIOPort)newobj.getPort("input");
        newobj.output = (TypedIOPort)newobj.getPort("output");
        return newobj;
    }
```

4.3 Action Methods

Figure 4.1 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 initialize() method, which is invoked exactly once before any other action method is invoked. Typically, this is used to initialize state variables in the actor.

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().

4.3.1 Initialize

The initialize() method of the Average actor is shown in figure 4.7. 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 `_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. The actor will fail to run if you forget this step.

Occasionally, an actor wishes to produce an initial output token once at the beginning of an execution of a model. It is tempting to do that in the initialize() method. However, currently, this cannot be done because type resolution is done after the initialize() methods of all the actors have been run. This permits actors to take action in the initialize() method that may affect type resolution[1]. To get the same effect, we recommend the following pattern:

```
    private boolean firstIteration;
```

```
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 4.7. Code segment from the Average actor, showing the initialize() method.

```
public void initialize() throws IllegalActionException {
   super.initialize();
   firstIteration = true;
   ...
}
public boolean prefire() throws IllegalActionException {
   if (firstIteration) output.send(0, token);
   return super.prefire();
}
```

This seems tedious, but actors that need to do this are relatively rare. Nonetheless, we are contemplating extensions to the type system design to allow initial tokens to be produced in the initialize() method in the future (the type of the token will become a type constraint on the port).

4.3.2 Prefire

The prefire() method is invoked exactly once per iteration. It returns a boolean that indicates to the director whether the actor wishes for firing to proceed. Thus, there are two possible uses for prefire(). First, if you need a method that is guaranteed to be invoked once per iteration of the actor in all domains, this is it. 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). Second, if you wish to test an input to see whether you are ready to fire, then you can do that here.

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) {
      if(trueInput.hasToken()) return true;
   } else {
      if(falseInput.hasToken()) return true;
   }
   return false;
}
```

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) {
      if(trueInput.hasToken()) return super.prefire();
   } else {
      if(falseInput.hasToken()) return super.prefire();
   }
   return false;
}
```

1. 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, such as the Expression actor, but rather are added later.

4.3.3 Fire

Figure 4.3 shows a typical fire() method. The fire() method reads inputs and produces 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 4.8. Notice the use of the add() and divide() methods of the Token class to achieve data polymorphism.
- When data polymorphism is not practical or not desired, then if you use setTypeEquals() to define the type of input ports, then the type system assures that you can safely cast the tokens that you read. Consider the Sine actor shown in figure 4.9. This actor only operates on inputs and parameters of type DoubleToken, so it declares these types in the constructor. In the fire() method, the input token read and the current parameter values are all cast to DoubleToken. The type system ensures that no cast error will occur. This actor computes $amplitude \times \sin(omega \times input + phase)$.
- 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, iterating or converging towards a solution. Thus, each invocation 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.

```
public class Sine extends Transformer {

   public Sine(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException  {
      super(container, name);

      amplitude = new Parameter(this, "amplitude", new DoubleToken(1.0));
      omega = new Parameter(this, "omega", new DoubleToken(1.0));
      phase = new Parameter(this, "phase", new DoubleToken(0.0));
      input.setTypeEquals(DoubleToken.class);
      output.setTypeEquals(DoubleToken.class);
   }

   public Parameter amplitude;
   public Parameter omega;
   public Parameter phase;

   public void fire() throws IllegalActionException {
      if (input.hasToken(0)) {
           DoubleToken in = (DoubleToken)input.get(0);
           double A = ((DoubleToken)amplitude.getToken()).doubleValue();
           double w = ((DoubleToken)omega.getToken()).doubleValue();
           double p = ((DoubleToken)phase.getToken()).doubleValue();
           double result = A*Math.sin(w*in.doubleValue()+p);
           output.broadcast(new DoubleToken(result));
      }
   }
}
```

FIGURE 4.9. Code for the Sine 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(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.broadcast(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 4.8. Code for the Average actor, showing the action methods.

4.3.4 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 4.8. 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 that returns false. Consider the two examples shown in figure 4.10. These are base classes for source actors (those with no input ports).

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.

4.3.5 Wrapup

The wrapup() method is invoked exactly once at the end of an execution, unless an exception occurs during execution. It is used typically for displaying final results.

4.4 Time

Actors 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 with the syntax

```
double currentTime = getDirector().getCurrentTime();
```

Notice that although the director has a public method setCurrentTime(), an actor should never use it. An actor does not have sufficiently global visibility to be able to do this. Typically, only another enclosing director will call this method.

An actor can request an invocation at a future time using the fireAt() method of the director. This method returns immediately (for a correctly implemented director). It takes two arguments, an actor and a time. The director is responsible for iterating the specified actor at the specified time. This method can be used to get a source actor started, and to keep it operating. In its 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.

4.5 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.

4.5.1 Indentation

Nested statements should be indented 4 characters, as in:

```
public class SequenceSource extends Source implements SequenceActor {

    public SequenceSource(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        firingCountLimit = new Parameter(this, "firingCountLimit", new IntToken(0));
    }

    public Parameter firingCountLimit;

    ...

    public boolean postfire() throws IllegalActionException {
        _iterationCount++;
        if (_iterationCount == ((IntToken)firingCountLimit.getToken()).intValue()) {
            return false;
        }
        return true;
    }

    private int _iterationCount = 0;
}



public class TimedSource extends Source implements TimedActor {

    public TimedSource(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        stopTime = new Parameter(this, "stopTime", new DoubleToken(0.0));
    }

    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 4.10. Code for the Average actor, showing the Ptolemy coding convention standards.

```
if (container != null) {
    Manager manager = container.getManager();
    if (manager != null) {
        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 viewers. Do not override this in your text editor. Long lines should be broken up into many small lines. Long strings can be broken up using the + operator in Java. Continuation lines are indented by 8 characters, as in the throws clause of the constructor in figure 4.1.

4.5.2 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:

```
Right: a = b + 1;
Wrong: a=b+1;
Right: for (i = 0; i < 10; i += 2)
Wrong: for (i=0; i<10; i+=2)
```

4.5.3 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.

4.5.4 Names

Class names, and only class names, are capitalized, and also have internal capitalization at word boundaries, as in AtomicActor. Method and member names are not capitalized, except at internal word 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.

4.5.5 Exceptions

A number of exceptions are provided in the ptolemy.kernel.util package. Use these exceptions when possible because they provide convenient 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.

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() + ".");
```

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.

4.5.6 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 member, method, or class. Ptolemy II classes include Javadoc documentation for all classes and all public and protected members and methods. Private members and methods need not be documented. 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 expr The expression for this variable.
```

```
 */
public void setExpression(String expr) {
   ...
}
```

We use instead the imperative tense, as in

```
/** Set the expression of this variable.
 *  @param expr The expression for this variable.
 */
public void setExpression(String expr) {
   ...
}
```

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 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. Except for a huge number of rather meaningless messages that it insists on giving about serialization of objects that cannot be serialized, our policy is to format the comments until there are no Javadoc warnings.

4.5.7 Code Organization

The basic file structure that we use follows the outline in figure 4.1, 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. This helps find them. If you wish to group methods together, try to name them so that they have a common prefix.

PART 2:

SOFTWARE ARCHITECTURE

The chapters in this part describe the software architecture of Ptolemy II. The first chapter covers the kernel package, which provides a set of Java classes supporting clustered graph topologies for models. This provides a very general abstract syntax for component-based modeling, without assuming or imposing any semantics on the models. The actor package begins to add semantics by providing basic infrastructure for data transport between components. The data package provides classes to encapsulate the data that is transported. It also provides an extensible type system and an interpreted expression language. The graph package provides graph-theoretic algorithms that are used in the type system and by schedulers in the individual domains. The plot package provides a visual data plotting utility that is used in many of the applets and applications.

5

The Kernel

Author: Edward A. Lee
Contributors:
John Davis, II
Ron Galicia
Mudit Goel
Christopher Hylands
Jie Liu
Xiaojun Liu
Lukito Muliadi
Steve Neuendorffer
John Reekie
Neil Smyth

5.1 Abstract Syntax

The kernel defines a small set of Java classes that implement a data structure supporting a general form of uninterpreted clustered graphs, plus methods for accessing and manipulating such graphs. These graphs provide an abstract syntax for netlists, state transition diagrams, block diagrams, etc. An *abstract syntax* is a conceptual data organization. It can be contrasted with a *concrete syntax*, which is a syntax for a persistent, readable representation of the data, such as EDIF for netlists. A particular graph configuration is called a *topology.*

Although this idea of an uninterpreted abstract syntax is present in the original Ptolemy kernel [13], in fact the original Ptolemy kernel has more semantics than we would like. It is heavily biased towards dataflow, the model of computation used most heavily. Much of the effort involved in implementing models of computation that are very different from dataflow stems from having to work around certain assumptions in the kernel that, in retrospect, proved to be particular to dataflow.

A topology is a collection of *entities* and *relations*. We use the graphical notation shown in figure 5.1, where entities are depicted as rounded boxes and relations as diamonds. Entities have *ports*,

shown as filled circles, 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 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.

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 of Gamma, *et al*. [25]. “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.

5.2 Non-Hierarchical Topologies

The classes shown in figure 5.2 support non-hierarchical topologies, like that shown in figure 5.1. Figure 5.2 is a UML static structure diagram (see appendix A of chapter 1).

5.2.1 Links

An Entity contains any number of Ports; such an aggregation is indicated by the association with an unfilled diamond and the label “0..n” to show that the Entity can contain any number of Ports, and the label “0..1” to show that the Port is contained by at most one Entity. This association is uses the NamedList class shown at the bottom of figure 5.2 and defined fully in figure 5.3. There is exactly one

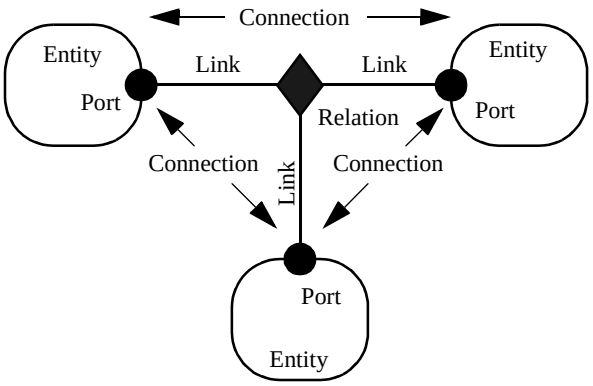


FIGURE 5.1. Visual notation and terminology.

instance of NamedList associated with Entity, and it aggregates the ports.

A Port is associated with any number of Relations (the association is called a *link*), and a Relation is associated with any number of Ports. Link associations use CrossRefList, shown in figure 5.3. There is exactly one instance of CrossRefList associated with each port and each relation. The links define a web of interconnected entities.

5.2.2 Consistency

A major concern in the choice of methods to provide and in their design is maintaining consistency. By *consistency* we mean that the following key properties are satisfied:

- Every link is symmetric and bidirectional. That is, if a port has a link to a relation, then the relation has a link back to that port.
- Every object that appears on a container's list of contained objects has a back reference to its container.

In particular, the design of these classes ensures that the _container attribute of a port refers to an entity that includes the port on its _portList. This is done by limiting the access to both attributes. The only way to specify that a port is contained by an entity is to call the setContainer() method of the port. That method guarantees consistency by first removing the port from any previous container's portList, then adding it to the new container's port list. A port is removed from an entity by calling setContainer() with a null argument.

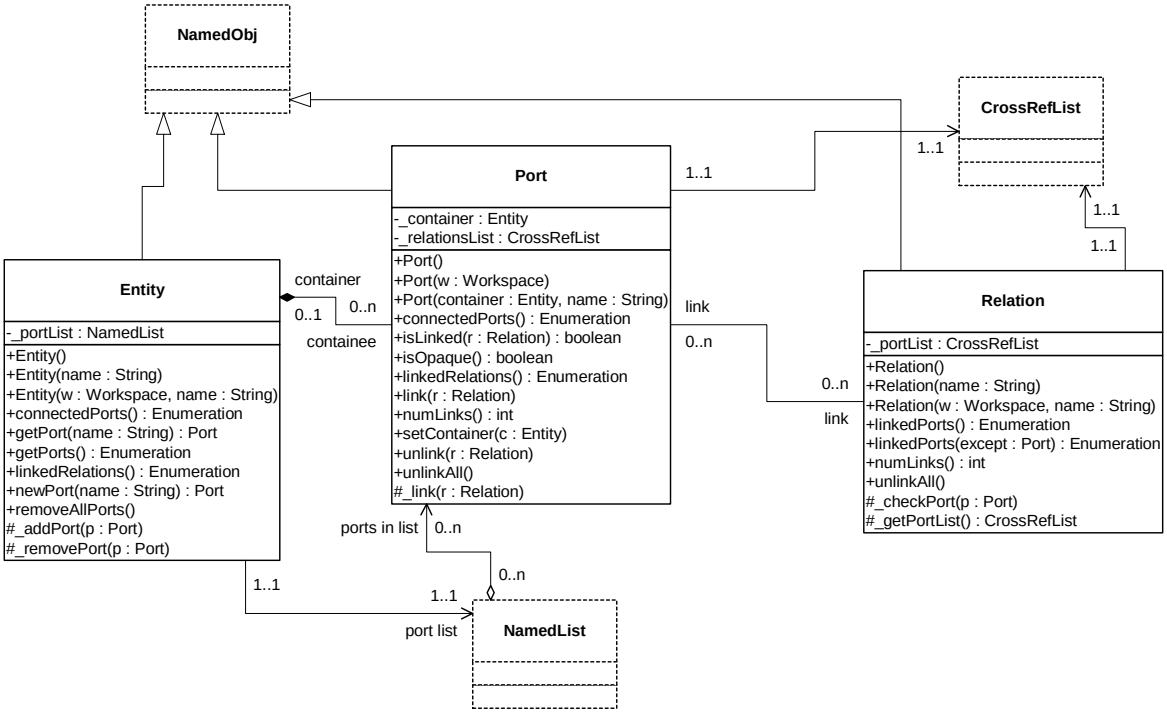


FIGURE 5.2. Key classes in the kernel package and their methods supporting basic (non-hierarchical) topologies. Methods that override those defined in a base class or implement those in an interface are not shown. The "+" indicates public visibility, "#" indicates protected, and "-" indicates private. Capitalized methods are constructors. The classes shown with dashed outlines are in the kernel.util subpackage.

A change in a containment association involves several distinct objects, and therefore must be atomic, in the sense that other threads must not be allowed to intervene and modify or access relevant attributes halfway through the process. This is ensured by synchronization on the workspace, as

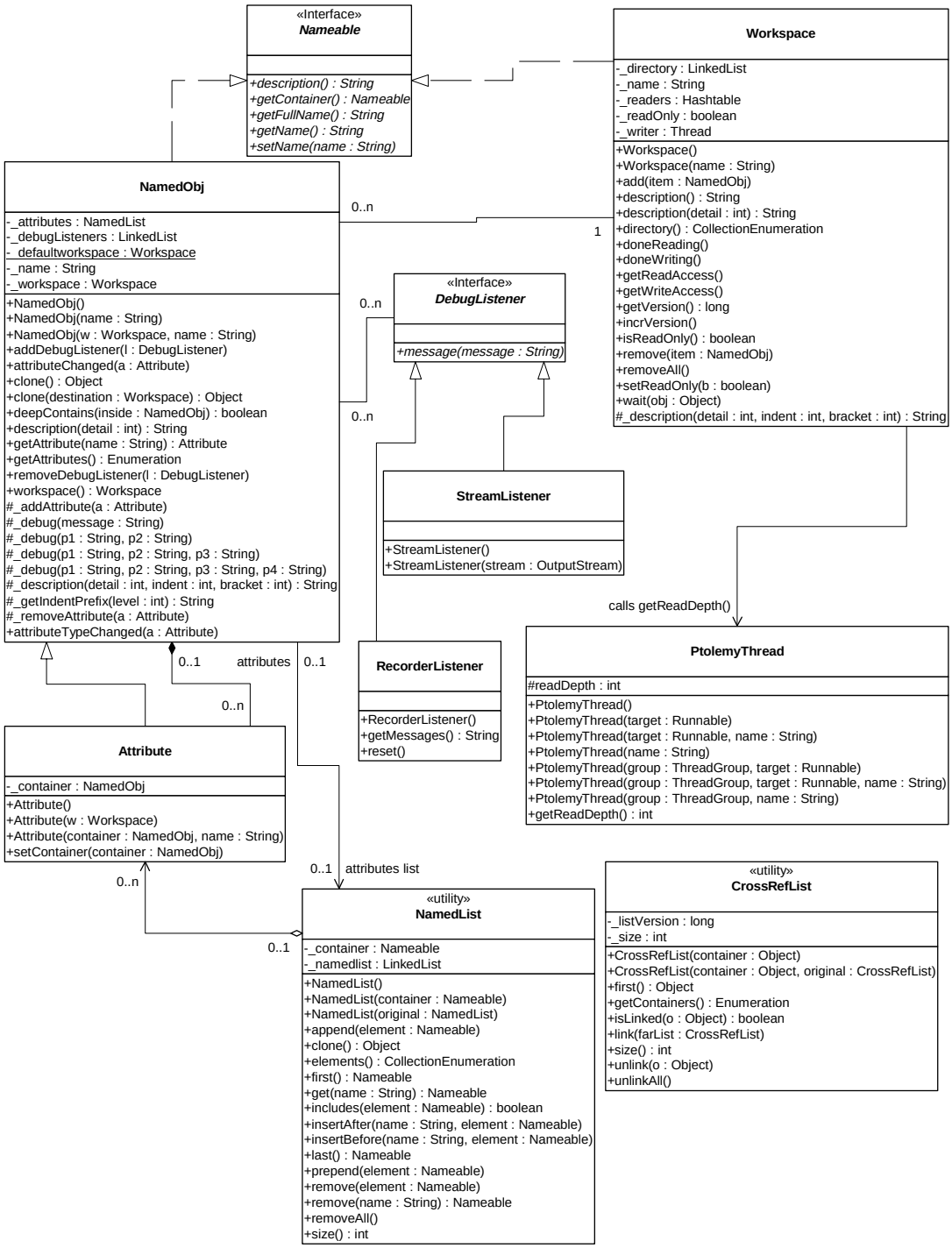


FIGURE 5.3. Support classes in the kernel.util package.

explained below in section 5.6. Moreover, if an exception is thrown at any point during the process of changing a containment association, any changes that have been made must be undone so that a consistent state is restored.

5.3 Support Classes

The kernel package has a subpackage called kernel.util that provides some underlying support classes, some of which are shown in figure 5.3. These classes define notions basic to Ptolemy II of containment, naming, and parameterization, and provide generic support for relevant data structures.

5.3.1 Containers

Although these classes do not provide support for constructing clustered graphs, they provide rudimentary support for *container* associations. An instance of these classes can have at most one container. That container is viewed as the owner of the object, and "managed ownership" [43] is used as a central tool in thread safety, as explained in section 5.6 below.

In the base classes shown in figure 5.2, only an instance of Port can have a non-null container. It is the only class with a setContainer() method. Instances of all other classes have no container, and their getContainer() method will return null. In the classes of figure 5.3, only Attribute has a setContainer() method.

Every object is associated with exactly one instance of Workspace, as shown in figure 5.3, but the workspace is not viewed as a container. The workspace is defined when an object is constructed, and no methods are provided to change it. It is said to be *immutable*, a critical property in its use for thread safety.

5.3.2 Name and Full Name

The Nameable interface supports hierarchy in the naming so that individual named objects in a hierarchy can be uniquely identified. By convention, the *full name* of an object is a concatenation of the full name of its container, if there is one, or the name of the workspace, if there is not, a period (“.”), and the name of the object. The full name is used extensively for error reporting.

NamedObj is a concrete class implementing the Nameable interface. It also serves as an aggregation of attributes, as explained below in section 5.3.4.

Names of objects are only required to be unique within a container. Thus, even the full name is not assured of being globally unique.

Here, names are a property of the instances themselves, rather than properties of an association between entities. As argued by Rumbaugh in [76], this is not always the right choice. Often, a name is more properly viewed as a property of an association. For example, a file name is a property of the association between a directory and a file. A file may have multiple names (through the use of symbolic links). Our design takes a stronger position on names, and views them as properties of the object, much as we view the name of a person as a property of the person (vs. their employee number, for example, which is a property of their association with an employer).

5.3.3 Workspace

Workspace is a concrete class that implements the Nameable interface, as shown in figure 5.3. All objects in a topology are associated with a workspace, and almost all operations that involve multiple

objects are only supported for objects in the same workspace. This constraint is exploited to ensure thread safety, as explained in section 5.6 below. The name of the workspace is always the first term in the full name. If the workspace has no name (a common situation), then the full name simply has a leading period.

5.3.4 Attributes

In almost all applications of Ptolemy II, entities, ports, and relations need to be parameterized. The base classes shown in figure 5.3 provide for these objects to have any number of instances of the Attribute class attached to them. Attribute is a NamedObj that can be contained by another NamedObj, and serves as a base class for parameters.

Attributes are added to a NamedObj by calling their setContainer() method and passing it a reference to the container. They are removed by calling setContainer() with a null argument. The NamedObj class provides the getAttribute() method, which takes an attribute name as an argument and returns the attribute, and the getAttributes() method, which returns an enumeration of all the attributes in the object.

By itself, an instance of the Attribute class carries only a name, which may not be sufficient to parameterize objects. A derived class called Parameter is defined in the data package.

5.3.5 List Classes

Figure 5.3 shows two list classes that are used extensively in Ptolemy II. NamedList implements an ordered list of objects with the Nameable interface. It is unlike a hash table in that it maintains an ordering of the entries that is independent of their names. It is unlike a vector or a linked list in that it supports accesses by name. It is used in figure 5.3 to maintain a list of attributes, and in figure 5.2 to maintain the list of ports contained by an entity.

The class CrossRefList is a bit more interesting. It mediates bidirectional links between objects that contain CrossRefLists, in this case, ports and relations. It provides a simple and efficient mechanism for constructing a web of objects, where each object maintains a list of the objects it is linked to. That list is an instance of CrossRefList. The class ensures consistency. That is, if one object in the web is linked to another, then the other is linked back to the one. CrossRefList also handles efficient modification of the cross references. In particular, if a link is removed from the list maintained by one object, the back reference in the remote object also has to be deleted. This is done in O(1) time. A more brute force solution would require searching the remote list for the back reference, increasing the time required and making it proportional to the number of links maintained by each object.

5.4 Clustered Graphs

The classes shown in figure 5.2 provide only partial support for hierarchy, through the concept of a container. Subclasses, shown in figure 5.4, extend these with more complete support for hierarchy. ComponentEntity, ComponentPort, and ComponentRelation are used whenever a clustered graph is used. All ports of a ComponentEntity are required to be instances of ComponentPort. CompositeEntity extends ComponentEntity with the capability of containing ComponentEntity and ComponentRelation objects. Thus, it contains a subgraph. The association between ComponentEntity and CompositeEntity is the classic Composite design pattern [25].

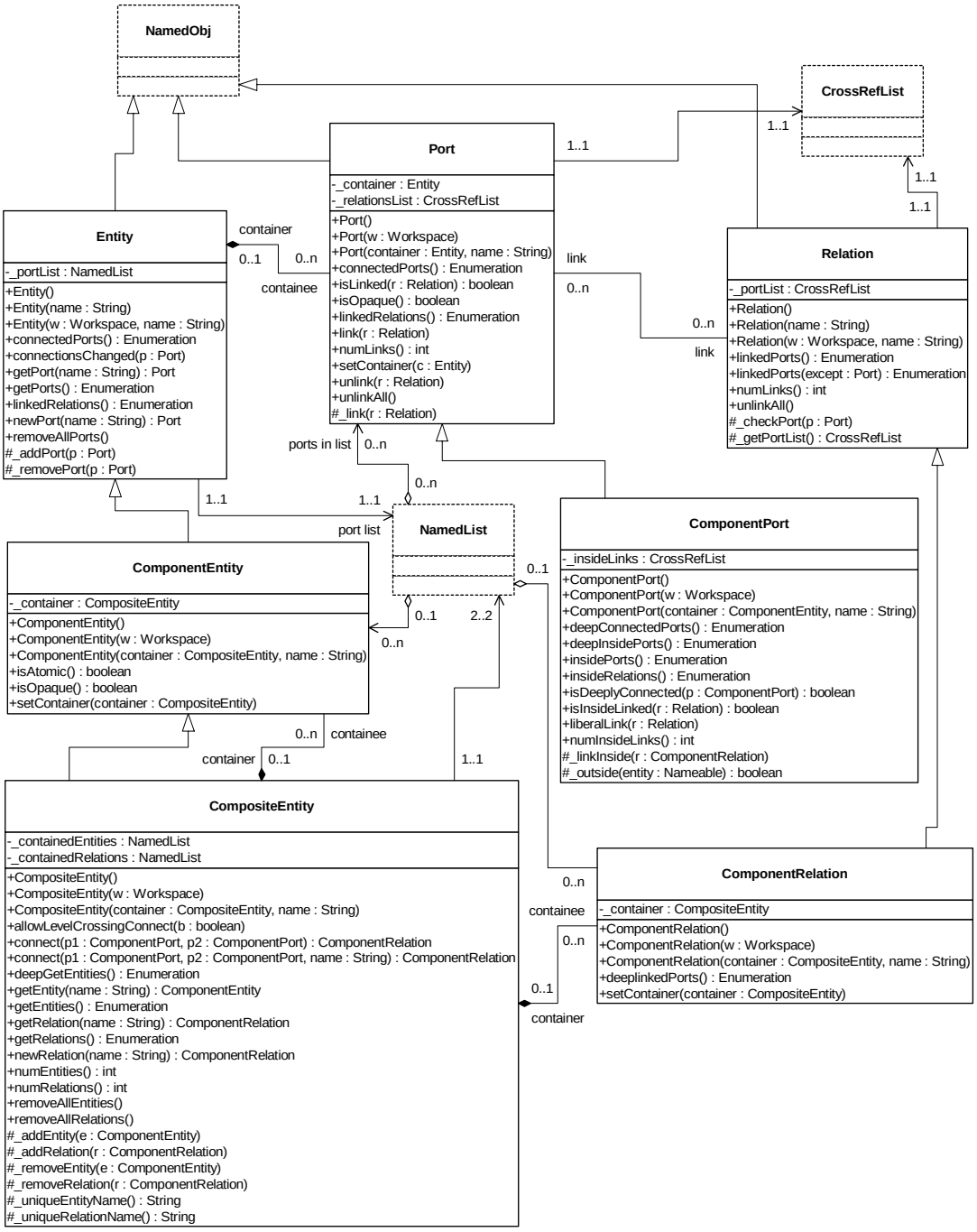


FIGURE 5.4. Key classes supporting clustered graphs.

5.4.1 Abstraction

Composite entities are non-atomic (isAtomic() return false). They can contain a graph (entities and relations). By default, a CompositeEntity is transparent (isOpaque() returns false). Conceptually, this means that its contents are visible from the outside. The hierarchy can be ignored (flattened) by algorithms operating on the topology. Some subclasses of CompositeEntity are opaque (see the Actor Package chapter for examples). This forces algorithms to respect the hierarchy, effectively hiding the contents of a composite and making it appear indistinguishable from atomic entities.

A ComponentPort contained by a CompositeEntity has inside as well as outside links. It maintains two lists of links, those to relations inside and those to relations outside. 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 [57]. Ports within an entity are hidden by default, and must be explicitly exposed to be visible (linkable) from outside the entity[1]. The composite entity with ports thus provides an abstraction of the contents of the composite.

A port of a composite entity may be opaque or transparent. It is defined to be *opaque* if its container is opaque. Conceptually, if it is opaque, then its inside links are not visible from the outside, and the outside links are not visible from the inside. If it is opaque, it appears from the outside to be indistinguishable from a port of an atomic entity.

The transparent port mechanism is illustrated by the example in figure 5.5[2]. Some of the ports in figure 5.5 are filled in white rather than black. These ports are said to be *transparent*. Transparent 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).

ComponentPort, ComponentRelation, and CompositeEntity have a set of methods with the prefix "deep," as shown in figure 5.4. These methods flatten the hierarchy by traversing it. Thus, for example,

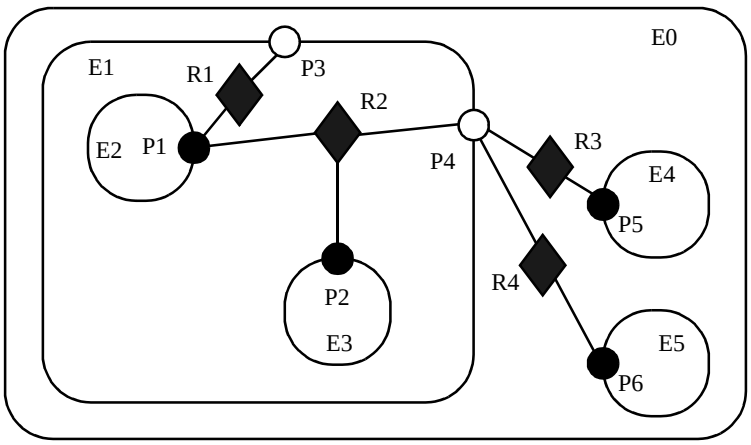


FIGURE 5.5. Transparent ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

1. Unless level-crossing links are allowed, which is discouraged.
2. In that figure, every object has been given a unique name. This is not necessary since names only need to be unique within a container. In this case, we could refer to P5 by its full name .E0.E4.P5, assuming the workspace has no name (the leading period indicates this). However, using unique names makes our explanations more readable.

the ports that are “deeply” connected to port P1 in figure 5.5 are P2, P5, and P6. No transparent port is included, so note that P3 is not included.

Deep traversals of a graph follow a simple rule. If a transparent port is encountered from inside, then the traversal continues with its outside links. If it is encountered from outside, then the traversal continues with its inside links. Thus, for example, the ports deeply connected to P5 are P1 and P2. Note that P6 is not included. Similarly, the deepGetEntities() method of CompositeEntity looks inside transparent entities, but not inside opaque entities.

Since deep traversals are more expensive than just checking adjacent objects, both ComponentPort and ComponentRelation cache them. To determine the validity of the cached list, the version of the workspace is used. As shown in figure 5.2, the Workspace class includes a getVersion() and incrVersion() method. All methods of objects within a workspace that modify the topology in any way are expected to increment the version count of the workspace. That way, when a deep access is performed by a ComponentPort, it can locally store the resulting list and the current version of the workspace. The next time the deep access is requested, it checks the version of the workspace. If it is still the same, then it returns the locally cached list. Otherwise, it reconstructs it.

For ComponentPort to support both inside links and outside links, it has to override the link() and unlink() methods. Given a relation as an argument, these methods can determine whether a link is an inside link or an outside link by checking the container of the relation. If that container is also the container of the port, then the link is an inside link.

5.4.2 Level-Crossing Connections

For a few applications, such as Statecharts [30], level-crossing links and connections are needed. The example shown in figure 5.6 has three level-crossing connections that are slightly different from one another. The links in these connections are created using the liberalLink() method of ComponentPort. The link() method prohibits such links, throwing an exception if they are attempted (most applications will prohibit level-crossing connections by using only the link() method).

An alternative that may be more convenient for a user interface is to use the connect() methods of CompositeEntity rather than the link() or liberalLink() method of ComponentPort. To allow level-crossing links using connect(), first call allowLevelCrossingConnect() with a *true* argument.

The simplest level-crossing connection in figure 5.6 is at the bottom, connecting P2 to P7 via the relation R5. The relation is contained by E1, but the connection would be essentially identical if it were contained by any other entity. Thus, the notion of composite entities containing relations is somewhat weaker when level-crossing connections are allowed.

The other two level-crossing connections in figure 5.6 are mediated by transparent ports. This sort of hybrid could come about in heterogeneous representations, where level-crossing connections are permitted in some parts but not in others. It is important, therefore, for the classes to support such hybrids.

To support such hybrids, we have to modify slightly the algorithm by which a port recognizes an inside link. Given a relation and a port, the link is an inside link if the relation is contained by an entity that is either the same as or is deeply contained (i.e. directly or indirectly contained) by the entity that contains the port. The deepContains() method of NamedObj supports this test.

5.4.3 Tunneling Entities

The transparent port mechanism we have described supports connections like that between P1 and

P5 in figure 5.7. That connection passes through the entity E2. The relation R2 is linked to the inside of each of P2 and P4, in addition to its link to the outside of P3. Thus, the ports deeply connected to P1 are P3 and P5, and those deeply connected to P3 are P1 and P5, and those deeply connected to P5 are P1 and P3.

A *tunneling entity* is one that contains a relation with links to the inside of more than one port. It may of course also contain more standard links, but the term "tunneling" suggests that at least some deep graph traversals will see right through it.

Support for tunneling entities is a major increment in capability over the previous Ptolemy kernel [13] (Ptolemy Classic). That infrastructure required an entity (which was called a *star*) to intervene in any connection through a composite entity (which was called a *galaxy*). Two significant limitations

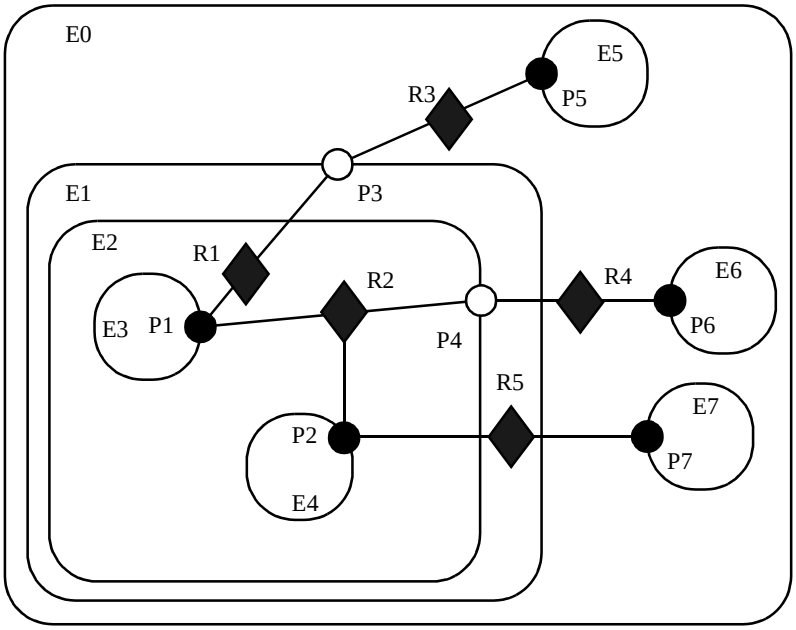


FIGURE 5.6. An example with level-crossing transitions.

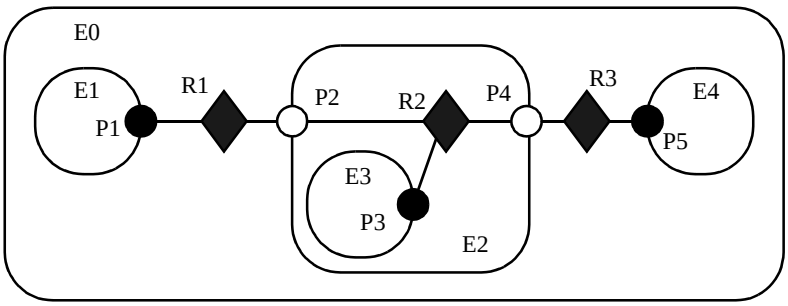


FIGURE 5.7. A tunneling entity contains a relation with inside links to more than one port.

resulted. The first was that compositionality was compromised. A connection could not be subsumed into a composite entity without fundamentally changing the structure of the application (by introducing a new intervening entity). The second was that implementation of higher-order functions that mutated the graph [48] was made much more complicated. These higher-order functions had to be careful to avoid mutations that created tunneling.

5.4.4 Description

The intent of Ptolemy II is that most applications will use graphical rather than textual syntaxes to visualize topologies. However, this is not always possible, and in any case, a graphical description may depict only the starting point of a topology that mutates. It can get difficult to understand an intricate topology.

The description() method in the Nameable interface (figure 5.3) provides a way to obtain detailed information about a topology in a human and machine readable format. This method is implemented by the NamedObj class, which also provides an alternative method that takes a *detail* argument. This argument can be used to control how much information is obtained.

An example is shown in figure 5.8, which describes the topology in figure 5.7. The general syntax for describing an object is "*classname* {*fullname*} *keyword* {*value*} *keyword* {*value*}". The value is often itself a description in exactly this form, or a list of descriptions in this form. For example, in figure 5.8, the keyword "attributes" is always followed by an empty value because no attributes have been set. The keyword "ports" precedes a list of contained ports, each a description. The keyword "entities" precedes a list of contained entities. The rest of the description should be evident.

5.4.5 Cloning

The kernel classes are all capable of being *cloned*, with some restrictions. Cloning means that an identical but entirely independent object is created. Thus, if the object being cloned contains other objects, then those objects are also cloned. If those objects are linked, then the links are replicated in the new objects. The clone() method in NamedObj provides the interface for doing this. Each subclass provides an implementation.

There is a key restriction to cloning. Because they break modularity, level-crossing links prevent cloning. With level-crossing links, a link does not clearly belong to any particular entity. An attempt to clone a composite that contains level-crossing links will trigger an exception.

5.4.6 An Elaborate Example

An elaborate example of a clustered graph is shown in figure 5.9. This example includes instances of all the capabilities we have discussed. The top-level entity is named "E0." All other entities in this example have containers. A Java class that implements this example is shown in figure 5.10. A script in the Tcl language [66] that constructs the same graph is shown in figure 5.11. This script uses Tcl Blend, an interface between Tcl and Java that is distributed by Scriptics.

The order in which links are constructed matters, in the sense that methods that return lists of objects preserve this order. The order implemented in both figures 5.10 and 5.11 is top-to-bottom and left-to-right in figure 5.9. A graphical syntax, however, does not generally have a particularly convenient way to completely control this order.

The results of various method accesses on the graph are shown in figure 5.12. This table can be studied to better understand the precise meaning of each of the methods.

5.5 Opaque Composite Entities

One of the major tenets of the Ptolemy project is that of modeling heterogeneous systems through the use of hierarchical heterogeneity. Information-hiding is a central part of this. In particular, transparent ports and entities compromise information hiding by exposing the internal topology of an entity. In some circumstances, this is inappropriate, for example when the entity internally operates under a different model of computation from its environment. The entity should be opaque in this case.

An entity can be opaque and composite at the same time. Ports are defined to be opaque if the entity containing them is opaque (isOpaque() returns true), so deep traversals of the topology do not cross these ports, even though the ports support inside and outside links. The actor package makes extensive use of such entities to support mixed modeling. That use is described in the Actor Package chapter. In the previous generation system, Ptolemy Classic, composite opaque entities were called

```
pt.kernel.CompositeEntity {.E0} attributes { } ports { } entities {
   pt.kernel.ComponentEntity {.E0.E1} attributes { } ports {
      pt.kernel.ComponentPort {.E0.E1.P1} attributes { } links {
         pt.kernel.ComponentRelation {.E0.R1} attributes { }
      } insidelinks { }
   }
   pt.kernel.CompositeEntity {.E0.E2} attributes { } ports {
      pt.kernel.ComponentPort {.E0.E2.P2} attributes { } links {
         pt.kernel.ComponentRelation {.E0.R1} attributes { }
      } insidelinks {
         pt.kernel.ComponentRelation {.E0.E2.R2} attributes { }
      }
      pt.kernel.ComponentPort {.E0.E2.P4} attributes { } links {
         pt.kernel.ComponentRelation {.E0.R3} attributes { }
      } insidelinks {
         pt.kernel.ComponentRelation {.E0.E2.R2} attributes { }
      }
   } entities {
      pt.kernel.ComponentEntity {.E0.E2.E3} attributes { } ports {
         pt.kernel.ComponentPort {.E0.E2.E3.P3} attributes { } links {
            pt.kernel.ComponentRelation {.E0.E2.R2} attributes { }
         } insidelinks { }
      }
   } relations {
      pt.kernel.ComponentRelation {.E0.E2.R2} attributes { } links {
         pt.kernel.ComponentPort {.E0.E2.P2} attributes { }
         pt.kernel.ComponentPort {.E0.E2.E3.P3} attributes { }
         pt.kernel.ComponentPort {.E0.E2.P4} attributes { }
      }
   }
   pt.kernel.ComponentEntity {.E0.E4} attributes { } ports {
      pt.kernel.ComponentPort {.E0.E4.P5} attributes { } links {
         pt.kernel.ComponentRelation {.E0.R3} attributes { }
      } insidelinks { }
   }
} relations {
   pt.kernel.ComponentRelation {.E0.R1} attributes { } links {
      pt.kernel.ComponentPort {.E0.E1.P1} attributes { }
      pt.kernel.ComponentPort {.E0.E2.P2} attributes { }
   }
   pt.kernel.ComponentRelation {.E0.R3} attributes { } links {
      pt.kernel.ComponentPort {.E0.E2.P4} attributes { }
      pt.kernel.ComponentPort {.E0.E4.P5} attributes { }
   }
}
```

FIGURE 5.8. An example of the syntax returned by the description() method.

wormholes.

5.6 Concurrency

We expect concurrency. Topologies often represent the structure of computations. Those computations themselves may be concurrent, and a user interface may be interacting with the topologies while they execute their computation. Moreover, using RMI or CORBA, Ptolemy II objects may interact with other objects concurrently over the network via RMI or CORBA.

Both computations within an entity and the user interface are capable of modifying the topology. Thus, extra care is needed to make sure that the topology remains consistent in the face of simultaneous modifications (we defined consistency in section 5.2.2).

Concurrency could easily corrupt a topology if a modification to a symmetric pair of references is interrupted by another thread that also tries to modify the pair. Inconsistency could result if, for example, one thread sets the reference to the container of an object while another thread adds the same object to a different container's list of contained objects.

Ptolemy II prevents such inconsistencies from occurring. Such enforced consistency is called

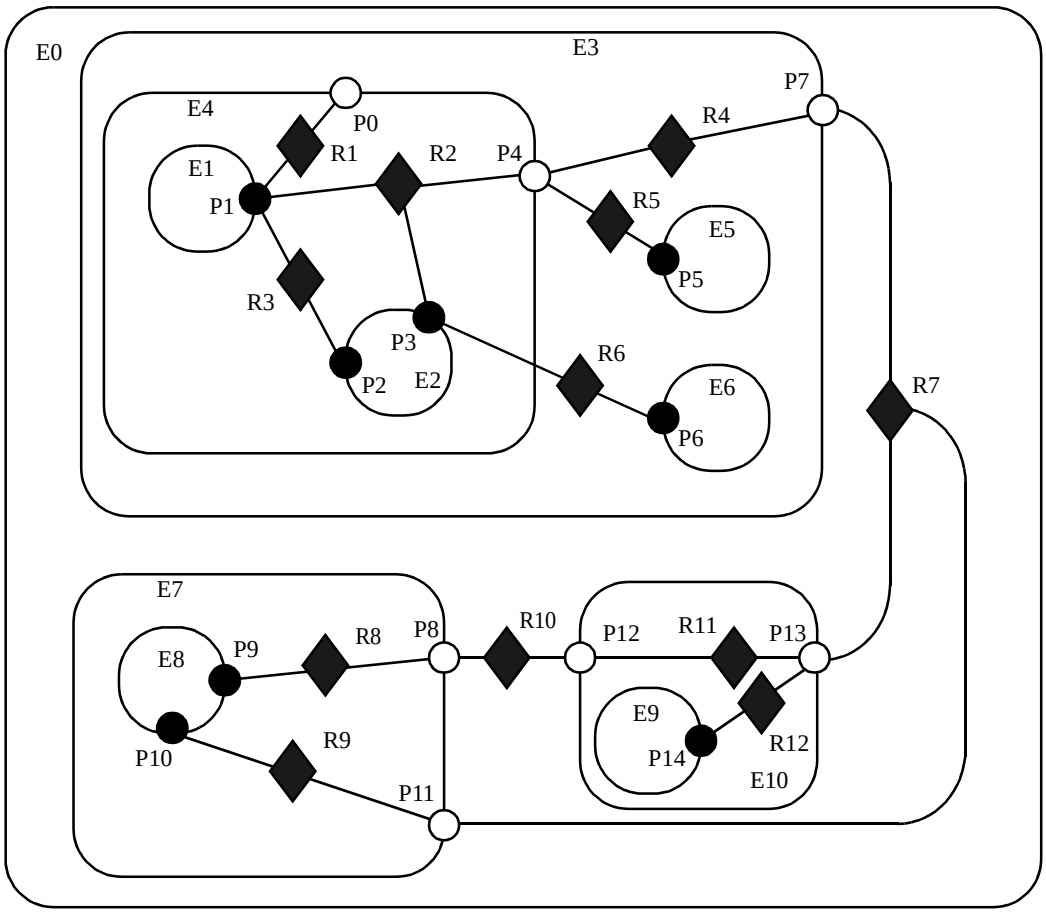


FIGURE 5.9. An example of a clustered graph.

```
public class ExampleSystem {
    private CompositeEntity e0, e3, e4, e7, e10;
    private ComponentEntity e1, e2, e5, e6, e8, e9;
    private ComponentPort p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p4;
    private ComponentRelation r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12;

    public ExampleSystem() throws IllegalActionException, NameDuplicationException {
        e0 = new CompositeEntity();
        e0.setName("E0");
        e3 = new CompositeEntity(e0, "E3");
        e4 = new CompositeEntity(e3, "E4");
        e7 = new CompositeEntity(e0, "E7");
        e10 = new CompositeEntity(e0, "E10");

        e1 = new ComponentEntity(e4, "E1");
        e2 = new ComponentEntity(e4, "E2");
        e5 = new ComponentEntity(e3, "E5");
        e6 = new ComponentEntity(e3, "E6");
        e8 = new ComponentEntity(e7, "E8");
        e9 = new ComponentEntity(e10, "E9");

        p0 = (ComponentPort) e4.newPort("P0");
        p1 = (ComponentPort) e1.newPort("P1");
        p2 = (ComponentPort) e2.newPort("P2");
        p3 = (ComponentPort) e2.newPort("P3");
        p4 = (ComponentPort) e4.newPort("P4");
        p5 = (ComponentPort) e5.newPort("P5");
        p6 = (ComponentPort) e5.newPort("P6");
        p7 = (ComponentPort) e3.newPort("P7");
        p8 = (ComponentPort) e7.newPort("P8");
        p9 = (ComponentPort) e8.newPort("P9");
        p10 = (ComponentPort) e8.newPort("P10");
        p11 = (ComponentPort) e7.newPort("P11");
        p12 = (ComponentPort) e10.newPort("P12");
        p13 = (ComponentPort) e10.newPort("P13");
        p14 = (ComponentPort) e9.newPort("P14");

        r1 = e4.connect(p1, p0, "R1");
        r2 = e4.connect(p1, p4, "R2");
        p3.link(r2);
        r3 = e4.connect(p1, p2, "R3");
        r4 = e3.connect(p4, p7, "R4");
        r5 = e3.connect(p4, p5, "R5");
        e3.allowLevelCrossingConnect(true);
        r6 = e3.connect(p3, p6, "R6");
        r7 = e0.connect(p7, p13, "R7");
        r8 = e7.connect(p9, p8, "R8");
        r9 = e7.connect(p10, p11, "R9");
        r10 = e0.connect(p8, p12, "R10");
        r11 = e10.connect(p12, p13, "R11");
        r12 = e10.connect(p14, p13, "R12");
        p11.link(r7);
    }
    ...
}
```

FIGURE 5.10. The same topology as in figure 5.9 implemented as a Java class.

thread safety.

5.6.1 Limitations of Monitors

Java threads provide a low-level mechanism called a *monitor* for controlling concurrent access to data structures. A monitor locks an object preventing other threads from accessing the object (a design pattern called *mutual exclusion*). However, the mechanism is fairly tricky to use correctly. It is non-trivial to avoid deadlock and race conditions. One of the major objectives of Ptolemy II is provide higher-level concurrency models that can be used with confidence by non experts.

Monitors are invoked in Java via the “synchronized” keyword. This keyword annotates a body of

```
# Create composite entities
  set e0 [java::new pt.kernel.CompositeEntity E0]
  set e3 [java::new pt.kernel.CompositeEntity $e0 E3]
  set e4 [java::new pt.kernel.CompositeEntity $e3 E4]
  set e7 [java::new pt.kernel.CompositeEntity $e0 E7]
  set e10 [java::new pt.kernel.CompositeEntity $e0 E10]

  # Create component entities.
  set e1 [java::new pt.kernel.ComponentEntity $e4 E1]
  set e2 [java::new pt.kernel.ComponentEntity $e4 E2]
  set e5 [java::new pt.kernel.ComponentEntity $e3 E5]
  set e6 [java::new pt.kernel.ComponentEntity $e3 E6]
  set e8 [java::new pt.kernel.ComponentEntity $e7 E8]
  set e9 [java::new pt.kernel.ComponentEntity $e10 E9]

  # Create ports.
  set p0 [$e4 newPort P0]
  set p1 [$e1 newPort P1]
  set p2 [$e2 newPort P2]
  set p3 [$e2 newPort P3]
  set p4 [$e4 newPort P4]
  set p5 [$e5 newPort P5]
  set p6 [$e6 newPort P6]
  set p7 [$e3 newPort P7]
  set p8 [$e7 newPort P8]
  set p9 [$e8 newPort P9]
  set p10 [$e8 newPort P10]
  set p11 [$e7 newPort P11]
  set p12 [$e10 newPort P12]
  set p13 [$e10 newPort P13]
  set p14 [$e9 newPort P14]

  # Create links
  set r1 [$e4 connect $p1 $p0 R1]
  set r2 [$e4 connect $p1 $p4 R2]
  $p3 link $r2
  set r3 [$e4 connect $p1 $p2 R3]
  set r4 [$e3 connect $p4 $p7 R4]
  set r5 [$e3 connect $p4 $p5 R5]
  $e3 allowLevelCrossingConnect true
  set r6 [$e3 connect $p3 $p6 R6]
  set r7 [$e0 connect $p7 $p13 R7]
  set r8 [$e7 connect $p9 $p8 R8]
  set r9 [$e7 connect $p10 $p11 R9]
  set r10 [$e0 connect $p8 $p12 R10]
  set r11 [$e10 connect $p12 $p13 R11]
  set r12 [$e10 connect $p14 $p13 R12]
  $p11 link $r7
```

FIGURE 5.11. The same topology as in figure 5.9 described by the Tcl Blend commands to create it.

code or a method, as shown in figure 5.13. It indicates that an exclusive lock should be obtained on a specific object before executing the body of code. If the keyword annotates a method, as in figure 5.13(a), then the method's object is locked (an instance of class A in the figure). The keyword can also be associated with an arbitrary body of code and can acquire a lock on an arbitrary object. In figure 5.13(b), the code body represented by ellipses (...) can be executed only after a lock has been acquired on object *obj*.

Modifications to a topology that run the risk of corrupting the consistency of the topology involve more than one object. Java does not directly provide any mechanism for simultaneously acquiring a lock on multiple objects. Acquiring the locks sequentially is not good enough because it introduces deadlock potential. I.e., one thread could acquire the lock on the first object block trying to acquire a lock on the second, while a second thread acquires a lock on the second object and blocks trying to acquire a lock on the first. Both methods block permanently, and the application is deadlocked. Neither thread can proceed.

One possible solution is to ensure that locks are always acquired in the same order [43]. For example, we could use the containment hierarchy and always acquired locks top-down in the hierarchy. Suppose for example that a body of code involves two objects *a* and *b*, where *a* contains *b* (directly or indirectly). In this case, "involved" means that it either modifies members of the objects or depends on their values. Then this body of code would be surrounded by:

```
synchronized(a) {
```

Table 1: Methods of ComponentRelation

Method Name	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
getLinkedPorts	P1 P0	P1 P4 P3	P1 P2	P4 P7	P4 P5	P3 P6	P7 P13 P11	P9 P8	P10 P11	P8 P12	P12 P13	P14 P13
deepGetLinkedPorts	P1	P1 P9 P14 P10 P5 P3	P1 P2	P1 P3 P9 P14 P10	P1 P3 P5	P3 P6	P1 P3 P9 P14 P10	P9 P1 P3 P10	P10 P1 P3 P9 P14	P9 P1 P3 P10	P9 P1 P3 P10	P14 P1 P3 P10

Table 2: Methods of ComponentPort

Method Name	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
getConnectedPorts		P0 P4 P3 P2	P1	P1 P4 P6	P7 P5	P4	P3	P13 P11	P12	P8	P11	P7 P13	P8	P7 P11	P13
deepGetConnectedPorts		P9 P14 P10 P5 P3 P2	P1	P1 P9 P14 P10 P5 P6	P9 P14 P10 P5	P1 P3	P3	P9 P14 P10	P1 P3 P10	P1 P3 P10	P1 P3 P9 P14	P1 P3 P9 P14	P9	P1 P3 P10	P1 P3 P10

FIGURE 5.12. Key methods applied to figure 5.9.

```
        synchronized (b) {
            ...
        }
    }
```

If all code that locks *a* and *b* respects this same order, then deadlock cannot occur. However, if the code involves two objects where one does not contain the other, then it is not obvious what ordering to use in acquiring the locks. Worse, a change might be initiated that reverses the containment hierarchy while another thread is in the process of acquiring locks on it. A lock must be acquired to read the containment structure before the containment structure can be used to acquire a lock! Some policy could certainly be defined, but the resulting code would be difficult to guarantee. Moreover, testing for deadlock conditions is notoriously difficult, so we implement a more conservative, and much simpler strategy.

5.6.2 Read and Write Access Permissions for Workspace

One way to guarantee thread safety without introducing the risk of deadlock is to give every object an immutable association with another object, which we call its *workspace*. *Immutable* means that the association is set up when the object is constructed, and then cannot be modified. When a change involves multiple objects, those objects must be associated with the same workspace. We can then acquire a lock on the workspace before making any changes or reading any state, preventing other threads from making changes at the same time.

Ptolemy II uses monitors only on instances of the class Workspace. As shown in figure 5.3, every

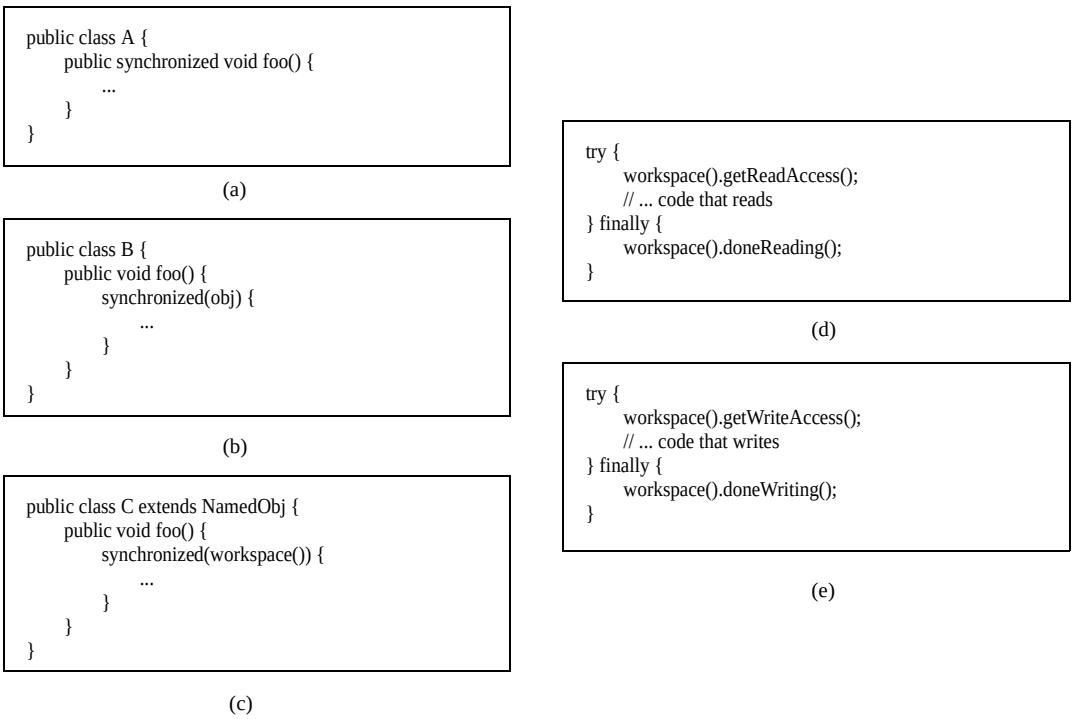


FIGURE 5.13. Using monitors for thread safety. The method used in Ptolemy II is in (d) and (e).

instance of NamedObj (or derived classes) is associated with a single instance of Workspace. Each body of code that alters or depends on the topology must acquire a lock on its workspace. Moreover, the workspace associated with an object is immutable. It is set in the constructor and never modified. This is enforced by a very simple mechanism: a reference to the workspace is stored in a private variable of the base class NamedObj, as shown in figure 5.3, and no methods are provided to modify it. Moreover, in instances of these kernel classes, a container and its containees must share the same workspace (derived classes may be more liberal in certain circumstances). This "managed ownership" [43] is our central strategy in thread safety.

As shown in figure 5.13(c), a conservative approach would be to acquire a monitor on the workspace for each body of code that reads or modified objects in the workspace. However, this approach is too conservative. Instead, Ptolemy II allows any number of readers to simultaneously access a workspace. Only one writer can access the workspace, however, and only if no readers are concurrently accessing the workspace.

The code for readers and writers is shown in figure 5.13(d) and (e). In (d), a reader first calls the getReadAccess() method of the Workspace class. That method does not return until it is safe to read data anywhere in the workspace. It is safe if there is no other thread concurrently holding (or requesting) a write lock on the workspace (the thread calling getReadAccess() may safely hold both a read and a write lock). When the user is finished reading the workspace data, it must call doneReading(). Failure to do so will result in no writer ever again gaining write access to the workspace. Because it is so important to call this method, it is enclosed in the finally clause of a try statement. That clause is executed even if an exception occurs in the body of the try statement.

The code for writers is shown in figure 5.13(e). The writer first calls the getWriteAccess() method of the Workspace class. That method does not return until it is safe to write into the workspace. It is safe if no other thread has read or write permission on the workspace. The calling thread, of course, may safely have both read and write permission at the same time. Once again, it is essential that doneWriting() be called after writing is complete.

This solution, while not as conservative as the single monitor of figure 5.13(c), is still conservative in that mutual exclusion is applied even on write actions that are independent of one another if they share the same workspace. This effectively serializes some modifications that might otherwise occur in parallel. However, there is no constraint in Ptolemy II on the number of workspaces used, so subclasses of these kernel classes could judiciously use additional workspaces to increase the parallelism. But they must do so carefully to avoid deadlock. Moreover, most of the methods in the kernel refuse to operate on multiple objects that are not in the same workspace, throwing an exception on any attempt to do so. Thus, derived classes that are more liberal will have to implement their own mechanisms supporting interaction across workspaces.

There is one significant subtlety regarding read and write permissions on the workspace. In a multithreaded application, normally, when a thread suspends (for example by calling wait()), if that thread holds read permission on the workspace, that permission is not relinquished during the time the thread is suspended. If another thread requires write permission to perform whatever action the first thread is waiting for, then deadlock will ensue. That thread cannot get write access until the first thread releases its read permission, and the first thread cannot continue until the second thread gets write access.

The way to avoid this situation is to use the wait() method of Workspace, passing as an argument the object on which you wish to wait (see Workspace methods in figure 5.3). That method first relinquishes all read permissions before calling wait on the target object. When wait() returns, notice that it is possible that the topology has changed, so callers should be sure to re-read any topology-dependent information. In general, this technique should be used whenever a thread suspends while it holds read

permissions.

5.6.3 Making a Workspace Read Only

Acquiring read and write access permissions on the workspace is not free, and it is performed so often in a typical application that it can significantly degrade performance. In some situations, an application may simply wish to prohibit all modifications to the topology for some period of time. This can be done by calling setReadOnly() on the workspace (see Workspace methods in figure 5.3). Once the workspace is read only, requests for read permission are routinely (and very quickly) granted, and requests for write permission trigger an exception. Thus, making a workspace read only can significantly improve performance, at the expense of denying changes to the topology.

5.7 Mutations

Often it is necessary to carefully constrain when changes can be made in a topology. For example, an application that uses the actor package to execute a model defined by a topology may require the topology to remain fixed during segments of the execution. During these segments, the workspace can be made read-only (see section 5.6.3), significantly improving performance.

A subpackage of the kernel, called the event package, provides support for carefully controlled mutations. The classes and interfaces in this package are shown in figure 5.14.

The typical usage pattern involves an originator that wishes to have a mutation performed, such as an actor (see the Actor Package chapter), and an object that accepts change requests, such as a director or manager, (again, see the Actor Package chapter). The originator creates an instance of the class ChangeRequest and enqueues that request by calling the requestChange() of the director or manager (a director typically delegates the request to the manager). When it is safe, the manager executes the change by calling execute() on each enqueued ChangeRequest. In addition, it informs any registered change listeners of the mutations so that they can react accordingly.

We have taken some liberties with the notation in figure 5.14. The originator implements the Nameable interface. Since this interface is not in the kernel.event package, it is shown with a dashed outline.

5.7.1 Change Requests

A manager processes a change request by calling its execute() method. If the request completes successfully, the director then notifies all change listeners attached to it. The ChangeRequest class is abstract. In a typical use, an originator will define an anonymous inner class, like this:

```
ChangeRequest change = new ChangeRequest(originator, "description") {
    public void execute() throws ChangeFailedException {
        try {
            ...
        } catch (IllegalActionException ex) {
            throw new ChangeFailedException(this, ex);
        }
    }
};
manager.requestChange(change);
```

The body of the execute() method could create entities, relations, ports, links, etc. For example, the code in the execute() method to create and link a new entity might look like this:

```
Entity newentity = new MyEntityClass(originator, "NewEntity");
relation.link(newentity.port);
```

When execute() is called, the entity named *newentity* will be created, added to *originator* (which is assumed to be an instance of CompositeEntity here) and linked to *relation*.

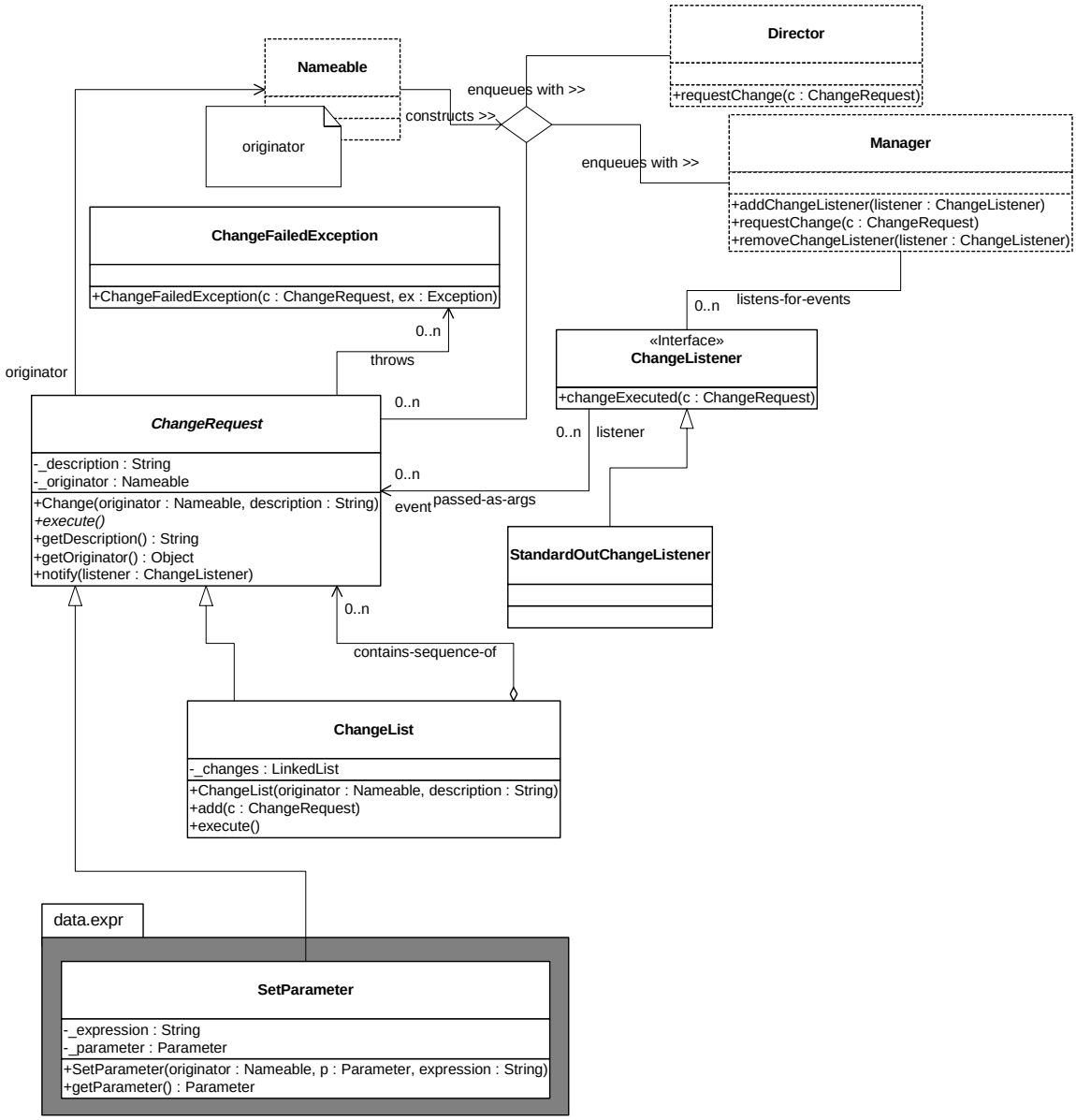


FIGURE 5.14. Classes and interfaces in kernel.event, which supports controlled topology mutations. An originator requests topology changes and a manager performs them at a safe time.

A set of concrete classes extending ChangeRequest are implemented in the actor.event package. Those are described in the Actor Package chapter. One such concrete derived class is shown in figure 5.14. The SetParameter class appears in the data.expr package (see the Data Package chapter). It provides a convenient mechanism for changing parameters of a model only at safe points during the execution of the model.

5.7.2 Managers and Listeners

The manager can safely perform topology changes, typically between iterations, even during an execution, but limiting the changes to occur between iterations. It provides addChangeListener() and removeChangeListener() methods, so that interested objects can register to be notified when topology changes occur. In addition, it provides a method that originators can use to queue requests. The director can also be used to queue requests, but it typically delegates the request to the manager.

A change listener is any object that implements the ChangeListener interface, and will typically include user interfaces and visualization components. The instance of ChangeRequest is passed to the listener, but unless it is one of the recognized concrete derived classes, then it is unlikely to be intelligible to the listener. At most the listener can obtain a description of the change. To get more details, it must query the topology.

5.8 Exceptions

Ptolemy II includes a set of exception classes that provide a uniform mechanism for reporting errors that takes advantage of the identification of named objects by full name. These exception are summarized in the class diagram in figure 5.15.

5.8.1 Base Class

KernelException. Not used directly. Provides common functionality for the kernel exceptions. In particular, it provides methods that take zero, one, or two Nameable objects plus an optional detail message (a String). The arguments provided are arranged in a default organization that is overridden in derived classes.

5.8.2 Less Severe Exceptions

These exceptions generally indicate that an operation failed to complete. These can result in a topology that is not what the caller expects, since the caller’s modifications to the topology did not succeed. However, they should *never* result in an inconsistent or contradictory topology.

IllegalActionException. Thrown on an attempt to perform an action that is disallowed. For example, the action would result in an inconsistent or contradictory data structure if it were allowed to complete. E.g., attempt to set the container of an object to be another object that cannot contain it because it is of the wrong class.

NameDuplicationException. 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.

NoSuchItemException. Thrown on access to an item that doesn't exist. E.g., attempt to remove a port by name and no such port exists.

5.8.3 More Severe Exceptions

The following exceptions should never trigger. If they trigger, it indicates a serious inconsistency in the topology and/or a bug in the code. At the very least, the topology being operated on should be abandoned and reconstructed from scratch. They are runtime exceptions, so they do not need to be explicitly declared to be thrown.

InvalidStateException. Some object or set of objects has a state that in theory is not permitted. E.g., a NamedObj has a null name. Or a topology has inconsistent or contradictory information in it, e.g. an entity contains a port that has a different entity as its container. Our design should make it impossible for this exception to ever occur, so occurrence is a bug. This exception is derived from the Java RuntimeException.

InternalErrorException. An unexpected error other than an inconsistent state has been encountered. Our design should make it impossible for this exception to ever occur, so occurrence is a bug. This

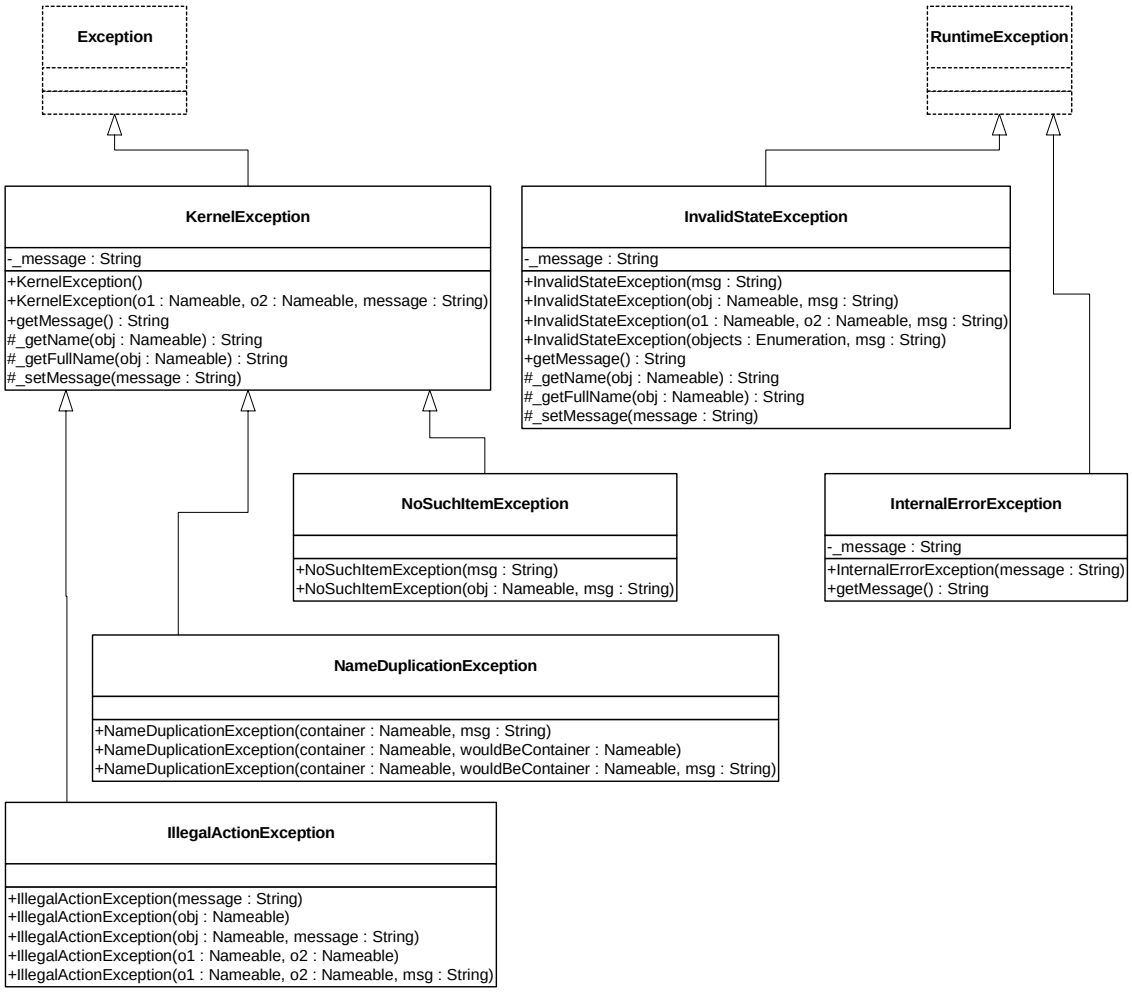


FIGURE 5.15. Summary of exceptions defined in the kernel.util package. These are used primarily through constructor calls. The form of the constructors is shown in the text. Exception and RuntimeException are

exception is derived from the Java RuntimeException.

6

Actor Package

Author: Edward A. Lee
Contributors:
Mudit Goel
Christopher Hylands
Jie Liu
Lukito Muliadi
Steve Neuendorffer
Neil Smyth
Yuhong Xiong

6.1 Concurrent Computation

In the kernel package, entities have no semantics. They are syntactic placeholders. In many of the uses of Ptolemy II, entities are executable. The actor package provides basic support for executable entities. It makes a minimal commitment to the semantics of these entities by avoiding specifying the order in which actors execute (or even whether the execute sequentially or concurrently), and by avoiding specifying the communication mechanism between actors. These properties are defined in the domains.

In most uses, these executable entities conceptually (if not actually) execute concurrently. The goal of the actor package is to provide a clean infrastructure for such concurrent execution that is neutral about the model of computation. It is intended to support dataflow, discrete-event, synchronous-reactive, continuous-time, communicating sequential processes, and process networks models of computation, at least. The detailed model of computation is then implemented in a set of derived classes called a *domain*. Each domain is a separate package.

Ptolemy II is an object-oriented application framework. *Actors* [1] extend the concept of objects to concurrent computation. Actors encapsulate a thread of control and have interfaces for interacting with other actors. They provide a framework for “open distributed object-oriented systems.” An actor can create other actors, send messages, and modify its own local state.

Inspired by this model, we group a certain set of classes that support computation within entities in the actor package. Our use of the term “actors,” however, is somewhat broader, in that it does not require an entity to be associated with a single thread of control, nor does it require the execution of threads associated with entities to be fair. Some subclasses, in other packages, impose such requirements, as we will see, but not all.

Agha’s actors [1] can only send messages to *acquaintances* — actors whose addresses it was given at creation time, or whose addresses it has received in a message, or actors it has created. Our equivalent constraint is that an actor can only send a message to an actor if it has (or can obtain) a reference to an input port of that actor. The usual mechanism for obtaining a reference to an input port uses the topology, probing for a port that it is connected to. Our relations, therefore, provide explicit management of acquaintance associations. Derived classes may provide additional implicit mechanisms. We define *actor* more loosely to refer to an entity that processes data that it receives through its ports, or that creates and sends data to other entities through its ports.

The actor package provides templates for two key support functions. These templates support message passing and the execution sequence (flow of control). They are *templates* in that no mechanism is actually provided for message passing or flow of control, but rather base classes are defined so that domains only need to override a few methods, and so that domains can interoperate.

6.2 Message Passing

The actor package provides templates for executable entities called *actors* that communicate with one another via message passing. Messages are encapsulated in *tokens* (see the Data Package chapter). Messages are sent via ports. IOPort is the key class supporting message transport, and is shown in figure 6.2. An IOPort can only be connected to other IOPort instances, and only via IORelations. The IORelation class is also shown in figure 6.2. TypedIOPort and TypedIORelation are subclasses that manage type resolution. These subclasses are used much more often, in order to benefit from the type system. This is described in detail in the Type System chapter.

An instance of IOPort can be an input, an output, or both. An *input port* (one that is capable of receiving messages) contains one or more instances of objects that implement the Receiver interface. Each of these receivers is capable of receiving messages from a distinct *channel*.

The type of receiver used depends on the communication protocol, which depends on the model of computation. The actor package includes two receivers, Mailbox and QueueReceiver. These are generic enough to be useful in several domains. The QueueReceiver class contains a FIFOQueue, the capacity of which can be controlled. It also provides a mechanism for tracking the history of tokens that are received by the receiver. The Mailbox class implements a FIFO (first in, first out) queue with capacity equal to one.

6.2.1 Data Transport

Data transport is depicted in figure 6.1. The originating actor E1 has an output port P1, indicated in the figure with an arrow in the direction of token flow. The destination actor E2 has an input port P2, indicated in the figure with another arrow. E1 calls the send() method of P1 to send a token *t* to a remote actor. The port obtains a reference to a remote receiver (via the IORelation) and calls the put() method of the receiver, passing it the token. The destination actor retrieves the token by calling the get() method of its input port, which in turn calls the get() method of the designated receiver.

Domains typically provide specialized receivers. These receivers override get() and put() to imple-

ment the communication protocol pertinent to that domain. A domain that uses asynchronous message passing, for example, can usually use the QueueReceiver shown in figure 6.2. A domain that uses synchronous message passing (rendezvous) has to provide a new receiver class.

In figure 6.1 there is only a single channel, indexed 0. The "0" argument of the send() and get() methods refer to this channel. A port can support more than one channel, however, as shown in figure 6.3. This can be represented by linking more than one relation to the port, or by linking a relation that has a width greater than one. A port that supports this is called a *multiport*. The channels are indexed $0, \ldots, N-1$, where N is the number of channels. An actor distinguishes between channels using this index in its send() and get() methods. By default, an IOPort is not a multiport, and thus supports only one channel. It is converted into a multiport by calling its setMultiport() method with a *true* argument. After conversion, it can support any number of channels.

Multiports are typically used by actors that communicate via an indeterminate number of channels. For example, a "distributor" or "demultiplexor" actor might divide an input stream into a number of output streams, where the number of output streams depends on the connections made to the actor. A *stream* is a sequence of tokens sent over a channel.

An IORelation, by default, represents a single channel. By calling its setWidth() method, however, it can be converted to a *bus*. A multiport may use a bus instead of multiple relations to distribute its data, as shown in figure 6.4. The *width of a relation* is the number of channels supported by the relation. If the relation is not a bus, then its width is one.

The *width of a port* is the sum of the widths of the relations linked to it. In figure 6.4, both the sending and receiving ports are multiports with width two. This is indicated by the "2" adjacent to each port. Note that the width of a port could be zero, if there are no relations linked to a port (such a port is said to be *disconnected*). Thus, a port may have width zero, even though a relation cannot. By convention, in Ptolemy II, if a token is sent from such a port, the token goes nowhere. Similarly, if a token is

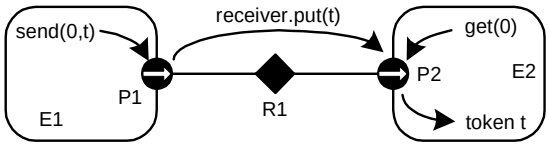


FIGURE 6.1. Message passing is mediated by the IOPort class. Its send() method obtains a reference to a remote receiver, and calls the put() method of the receiver, passing it the token *t*. The destination actor retrieves the token by calling the get() method of its input port.

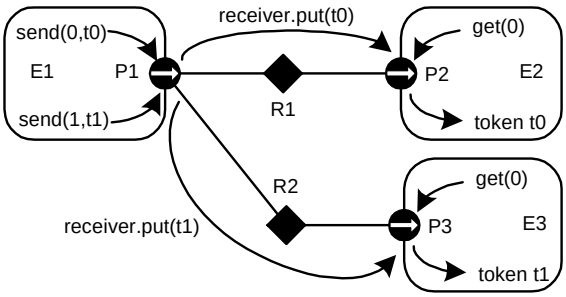


FIGURE 6.3. A port can support more than one channel, permitting an entity to send distinct data to distinct destinations via the same port. This feature is typically used when the number of destinations varies in different instances of the source actor.

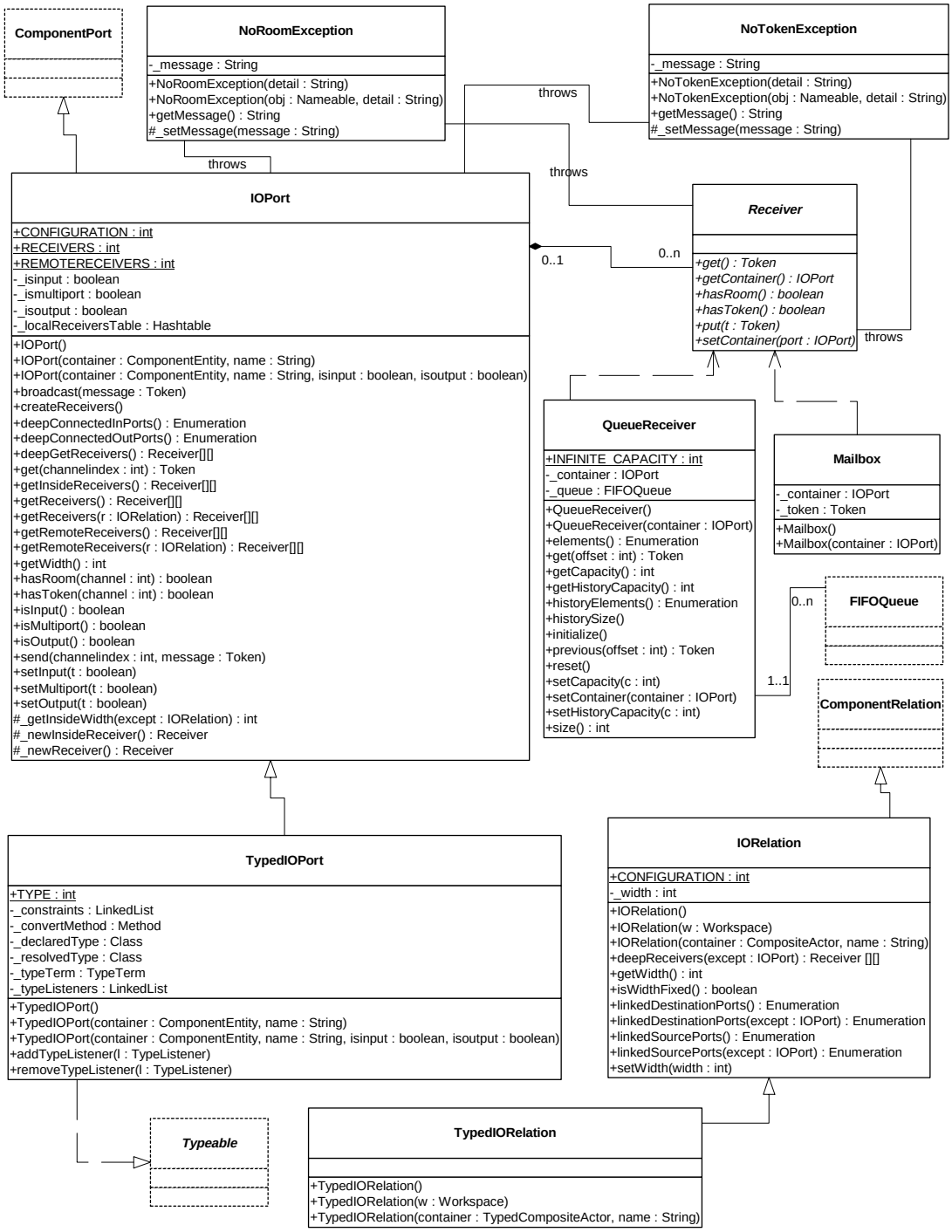


FIGURE 6.2. Port and receiver classes that provide infrastructure for message passing under various communication protocols.

sent via a relation that is not linked to any input ports, then the token goes nowhere. Such a relation is said to be *dangling*.

A given channel may reach multiple ports, as shown in figure 6.5. This is represented by a relation that is linked to multiple input ports. In the default implementation, in class IOPort, a reference to the token is sent to all destinations. Note that tokens are assumed to be immutable, so the recipients cannot modify the value. This is important because in most domains, it is not obvious in what order the recipients will see the token.

IOPort provides a broadcast() method for convenience. This method sends a specified token to all receivers linked to the port, regardless of the width of the port.

6.2.2 Example

An elaborate example showing all of the above features is shown in figure 6.6. In that example, we assume that links are constructed in top-to-bottom order. The arrows in the ports indicate the direction of the flow of tokens, and thus specify whether the port is an input, an output, or both. Multiports are indicated by adjacent numbers larger than one.

The top relation is a bus with width two, and the rest are not busses. The width of port *P1* is four. Its first two outputs (channels zero and one) go to *P4* and to the first two inputs of *P5*. The third output of *P1* goes nowhere. The fourth becomes the third input of *P5*, the first input of *P6*, and the only input of *P8*, which is both an input and an output port. Ports *P2* and *P8* send their outputs to the same set of destinations, except that *P8* does not send to itself. Port *P3* has width zero, so its send() method cannot be called without triggering an exception. Port *P6* has width two, but its second input channel has no output ports connected to it, so calling get(1) will trigger an exception that indicates that there is no data. Port *P7* has width zero so calling get() with any argument will trigger an exception.

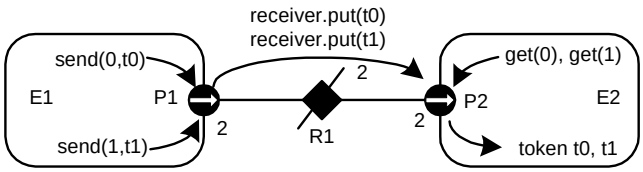


FIGURE 6.4. A bus is an IORelation that represents multiple channels. It is indicated by a relation with a slash through it, and the number adjacent to the bus is the width of the bus.

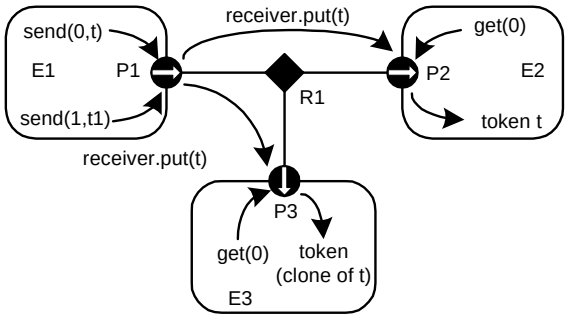


FIGURE 6.5. Channels may reach multiple destinations. This is represented by relations linking multiple input ports to an output port.

6.2.3 Transparent Ports

Recall that a port is transparent if its container is transparent (isOpaque() returns *false*). A CompositeActor is transparent unless it has a local director. Figure 6.7 shows an elaborate example where busses, input, and output ports are combined with transparent ports. The transparent ports are filled in white, and again arrows indicate the direction of token flow. The Tcl Blend code to construct this example is shown in figure 6.8.

By definition, a transparent port is an input if either

- it is connected on the inside to the outside of an input port, or
- it is connected on the inside to the inside of an output port.

That is, a transparent port is an input port if it can accept data (which it may then just pass through to a

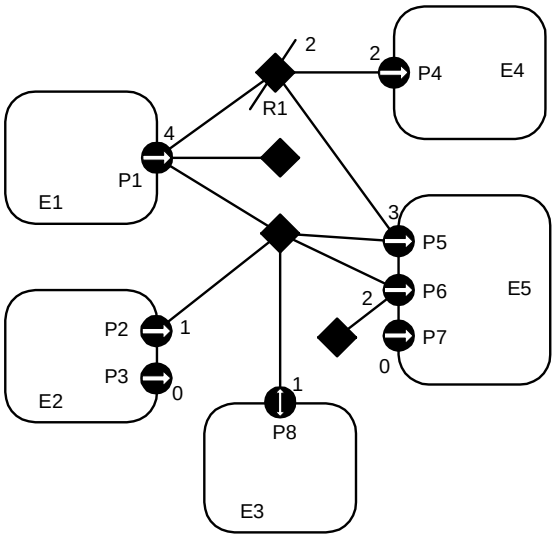


FIGURE 6.6. An elaborate example showing several features of the data transport mechanism.

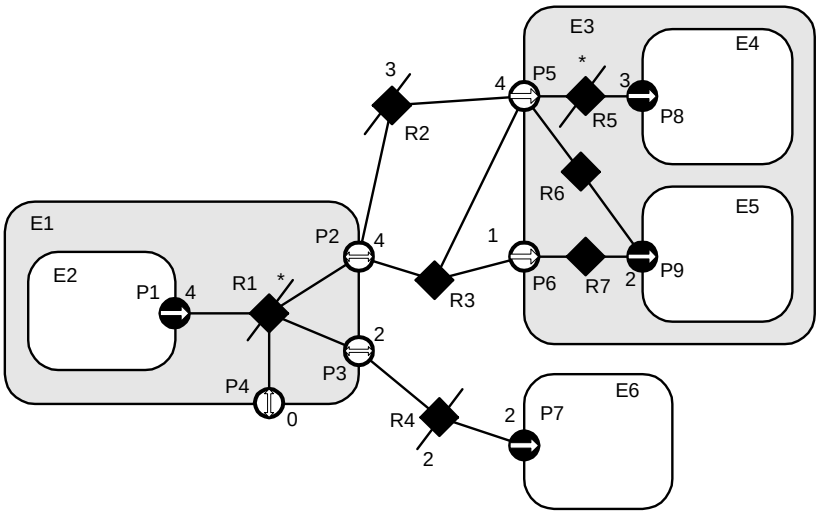


FIGURE 6.7. An example showing busses combined with input, output, and transparent ports.

transparent output port). Correspondingly, a transparent port is an output port if either

- it is connected on the inside to the outside of an output port, or
- it is connected on the inside to the inside of an input port.

Thus, assuming P1 is an output port and P7, P8, and P9 are input ports, then P2, P3, and P4 are both input and output ports, while P5 and P6 are input ports only.

Two of the relations that are inside composite entities (R1 and R5) are labeled as busses with a star (*) instead of a number. These are busses with unspecified width. The width is inferred from the topology. This is done by checking the ports that this relation is linked to from the inside and setting the width to the maximum of those port widths, minus the widths of other relations linked to those ports on the inside. Each such port is allowed to have at most one inside relation with an unspecified width, or an exception is thrown. If this inference yields a width of zero, then the width is defined to be one. Thus, R1 will have width 4 and R5 will have width 3 in this example. The width of a transparent port is the sum of the widths of the relations it is linked to on the outside (just like an ordinary port). Thus, P4 has width 0, P3 has width 2, and P2 has width 4. Recall that a port can have width 0, but a relation cannot have width less than one.

When data is sent from P1, four distinct channels can be used. All four will go through P2 and P5, the first three will reach P8, two copies of the fourth will reach P9, the first two will go through P3 to

```
set e0 [java::new ptolemy.actor.CompositeActor]
$e0 setDirector $director
$e0 setManager $manager

set e1 [java::new ptolemy.actor.CompositeActor $e0 E1]
set e2 [java::new ptolemy.actor.AtomicActor $e1 E2]
set e3 [java::new ptolemy.actor.CompositeActor $e0 E3]
set e4 [java::new ptolemy.actor.AtomicActor $e3 E4]
set e5 [java::new ptolemy.actor.AtomicActor $e3 E5]
set e6 [java::new ptolemy.actor.AtomicActor $e0 E6]

set p1 [java::new ptolemy.actor.IOPort $e2 P1 false true]
set p2 [java::new ptolemy.actor.IOPort $e1 P2]
set p3 [java::new ptolemy.actor.IOPort $e1 P3]
set p4 [java::new ptolemy.actor.IOPort $e1 P4]
set p5 [java::new ptolemy.actor.IOPort $e3 P5]
set p6 [java::new ptolemy.actor.IOPort $e3 P6]
set p7 [java::new ptolemy.actor.IOPort $e6 P7 true false]
set p8 [java::new ptolemy.actor.IOPort $e4 P8 true false]
set p9 [java::new ptolemy.actor.IOPort $e5 P9 true false]

set r1 [java::new ptolemy.actor.IORelation $e1 R1]
set r2 [java::new ptolemy.actor.IORelation $e0 R2]
set r3 [java::new ptolemy.actor.IORelation $e0 R3]
set r4 [java::new ptolemy.actor.IORelation $e0 R4]
set r5 [java::new ptolemy.actor.IORelation $e3 R5]
set r6 [java::new ptolemy.actor.IORelation $e3 R6]
set r7 [java::new ptolemy.actor.IORelation $e3 R7]

$p1 setMultiport true
$p2 setMultiport true
$p3 setMultiport true
$p4 setMultiport true
$p5 setMultiport true
$p7 setMultiport true
$p8 setMultiport true
$p9 setMultiport true
```

```
$r1 setWidth 0
$r2 setWidth 3
$r4 setWidth 2
$r5 setWidth 0

$p1 link $r1
$p2 link $r1
$p3 link $r1
$p4 link $r1
$p2 link $r2
$p5 link $r2
$p2 link $r3
$p5 link $r3
$p6 link $r3
$p3 link $r4
$p7 link $r4
$p5 link $r5
$p8 link $r5
$p5 link $r6
$p9 link $r6
$p6 link $r7
$p9 link $r7
```

FIGURE 6.8. Tcl Blend code to construct the example in figure 6.7.

P7, and none will go through P4.

By default, an IORelation is not a bus, so its width is one. To turn it into a bus with unspecified width, call setWidth() with a zero argument. Note that getWidth() will nonetheless never return zero (it returns at least one). To find out whether setWidth() has been called with a zero argument, call isWidthFixed() (see figure 6.2). If a bus with unspecified width is not linked on the inside to any transparent ports, then its width is one. It is not allowed for a transparent port to have more than one bus with unspecified width linked on the inside (an exception will be thrown on any attempt to construct such a topology). Note further that a bus with unspecified width is still a bus, and so can only be linked to multiports.

In general, bus widths inside and outside a transparent port need not agree. For example, if $M < N$ in figure 6.9, then first M channels from P1 reach P3, and the last $N - M$ channels are dangling. If $M > N$, then all N channels from P1 reach P3, but the last $M - N$ channels at P3 are dangling. Attempting to get a token from these channels will trigger an exception. Sending a token to these channels just results in loss of the token.

Note that data is not actually transported through the relations or transparent ports in Ptolemy II. Instead, each output port caches a list of the destination receivers (in the form of the two-dimensional array returned by getRemoteReceivers()), and sends data directly to them. The cache is invalidated whenever the topology changes, and only at that point will the topology be traversed again. This significantly improves the efficiency of data transport.

6.2.4 Data Transfer in Various Models of Computation

The receiver used by an input port determines the communication protocol. This is closely bound to the model of computation. The IOPort class creates a new receiver when necessary by calling its _newReceiver() protected method. That method delegates to the director returned by getDirector(), calling its newReceiver() method (the Director class will be discussed in section 6.3 below). Thus, the director controls the communication protocol, in addition to its primary function of determining the flow of control. Here we discuss the receivers that are made available in the actor package. This should not be viewed as an exhaustive set, but rather as a particularly useful set of receivers. These receivers are shown in figure 6.2.

Mailbox Communication. The Director base class by default returns a simple receiver called a Mailbox. A *mailbox* is a receiver has capacity for a single token. It will throw an exception if it is empty and get() is called, or it is full and put() is called. Thus, a subclass of Director that uses this should schedule the calls to put() and get() so that these exceptions do not occur, or it should catch these exceptions.

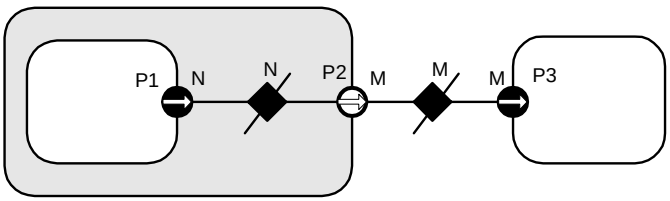


FIGURE 6.9. Bus widths inside and outside a transparent port need not agree..

Asynchronous Message Passing. This is supported by the QueueReceiver class. A QueueReceiver contains an instance of FIFOQueue, from the actor.util package, which implements a first-in, first-out queue. This is appropriate for all flavors of dataflow as well as Kahn process networks.

In the Kahn process networks model of computation [40], which is a generalization of dataflow [48], each actor has its own thread of execution. The thread calling get() will stall if the corresponding queue is empty. If the size of the queue is bounded, then the thread calling put() may stall if the queue is full. This mechanism supports implementation of a strategy that ensures bounded queues whenever possible [68].

In the process networks model of computation, the *history* of tokens that traverse any connection is determinate under certain simple conditions. With certain technical restrictions on the functionality of the actors (they must implement monotonic functions under prefix ordering of sequences), our implementation ensures determinacy in that the history does not depend on the order in which the actors carry out their computation. Thus, the history does not depend on the policies used by the thread scheduler.

FIFOQueue is a support class that implements a first-in, first-out queue. It is part of the actor.util package, shown in figure 6.10. This class has two specialized features that make it particularly useful in this context. First, its capacity can be constrained or unconstrained. Second, it can record a finite or infinite history, the sequence of objects previously removed from the queue. The history mechanism is useful both to support tracing and debugging and to provide access to a finite buffer of previously consumed tokens.

An example of an actor definition is shown in figure 6.11. This actor has a multiport output. It reads successive input tokens from the input port and distributes them to the output channels. This actor is written in a domain-polymorphic way, and can operate in any of a number of domains. If it is used in the PN domain, then its input will have a QueueReceiver and the output will be connected to ports with instances QueueReceiver.

Rendezvous Communications. Rendezvous, or synchronous communication, requires that the originator of a token and the recipient of a token both be simultaneously ready for the data transfer. As with process networks, the originator and the recipient are separate threads. The originating thread indicates

```
public class Distributor extends TypedAtomicActor {

    public TypedIOPort _input;
    public TypedIOPort _output;

    public Distribute(CompositeActor container, String name)
       throws NameDuplicationException, IllegalActionException {
          super(container, name);
          _input = new TypedIOPort(this, "input", true, false);
          _output = new TypedIOPort(this, "output", false, true);
          _output.setMultiport(true);
    }

    public void fire() throws IllegalActionException {
          for (int i=0; i < _output.getWidth(); i++) {
                _output.send(index, _input.get(0));
          }
    }
}
```

FIGURE 6.11. An actor that distributes successive input tokens to a set of output channels.

a willingness to rendezvous by calling send(), which in turn calls the put() method of the appropriate receiver. The recipient indicates a willingness to rendezvous by calling get() on an input port, which in turn calls get() of the designated receiver. Whichever thread does this first must stall until the other thread is ready to complete the rendezvous.

This style of communication is implemented in the CSP domain. In the receiver in that domain, the put() method suspends the calling thread if the get() method has not been called. The get() method suspends the calling thread if the put() method has not been called. When the second of these two methods is called, it wakes up the suspended thread and completes the data transfer. The actor shown in figure 6.11 works unchanged in the CSP domain, although its behavior is different in that input and output actions involve rendezvous with another thread.

Nondeterministic transfers can be easily implemented using this mechanism. Suppose for example that a recipient is willing to rendezvous with any of several originating threads. It could spawn a thread for each. These threads should each call get(), which will suspend the thread until the originator is willing to rendezvous. When one of the originating threads is willing to rendezvous with it, it will call put(). The multiple recipient threads will all be awakened, but only of them will detect that its rendezvous has been enabled. That one will complete the rendezvous, and others will die. Thus, the first orig-

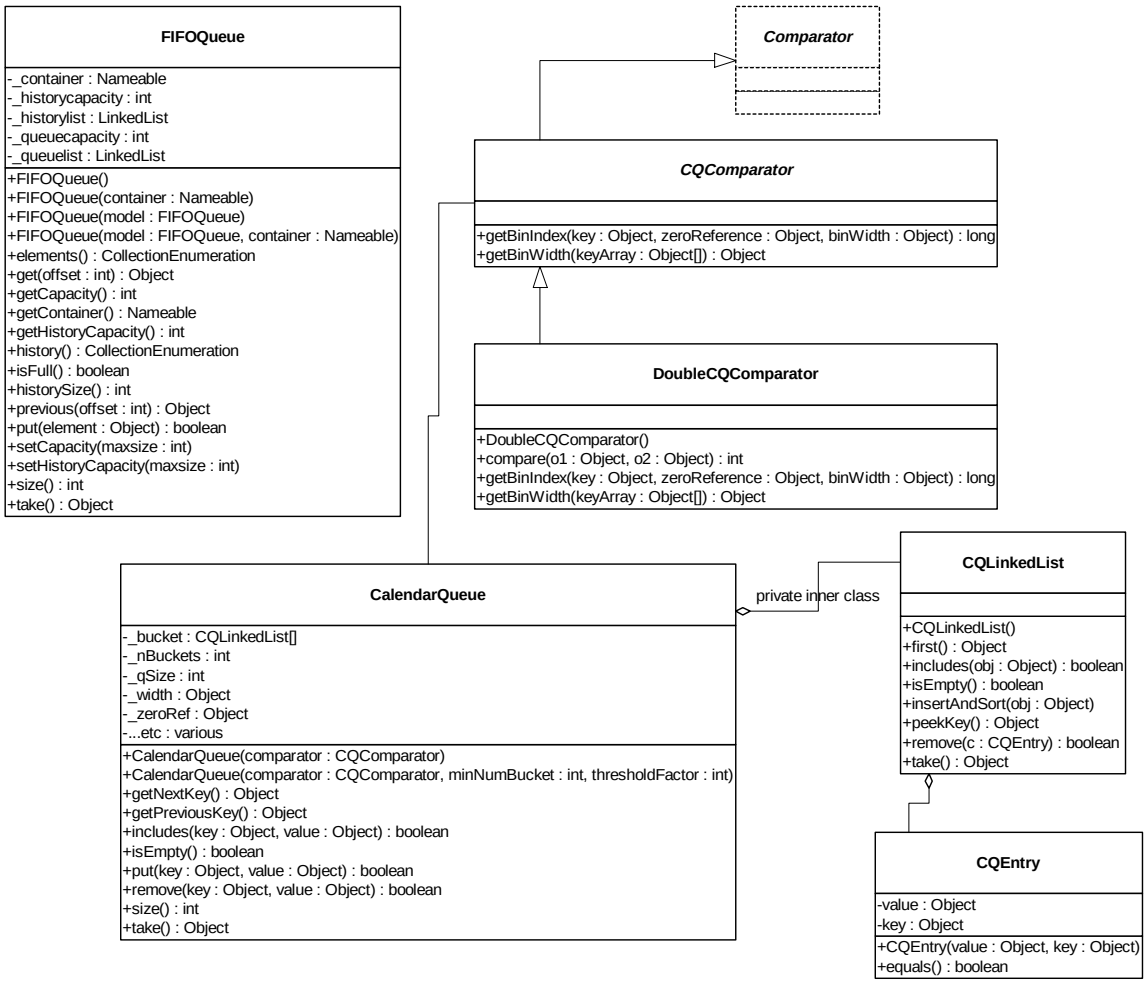


FIGURE 6.10. Static structure diagram for the actor.util package.

inating thread to indicate willingness to rendezvous will be the one that will transfer data. Guarded communication [4] can also be implemented.

Discrete-Event Communication. In the discrete-event model of computation, tokens that are transferred between actors have a *time stamp*, which specifies the order in which tokens should be processed by the recipients. The order is chronological, by increasing time stamp. To implement this, a discrete-event system will normally use a single, global, sorted queue rather than an instance of FIFOQueue in each input port. The kernel.util package, shown in figure 6.10, provides the CalendarQueue class, which gives an efficient and flexible implementation of such a sorted queue.

6.2.5 Discussion of the Data Transfer Mechanism

This data transfer mechanism has a number of interesting features. First, note that the actual transfer of data does not involve relations, so a model of computation could be defined that did not rely on relations. For example, a global name server might be used to address recipient ports. For example, to construct highly dynamic networks, such as wireless communication systems, it may be more intuitive to model a system as a aggregation of unconnected actors with addresses. A name server would return a reference to a port given an address. This could be accomplished simply by overriding the getRemoteReceivers() method of IOPort or TypedIOPort, or by providing an alternative method for getting references to receivers. The subclass of IOPort would also have to ensure the creation of the appropriate number of receivers. The base class relies on the width of the port to determine how many receivers to create, and the width is zero if there are no relations linked.

Note further that the mechanism here supports bidirectional ports. An IOPort may return true to both the isInput() and isOutput() methods.

6.3 Execution

The Executable interface, shown in figure 6.12, is implemented by the Director class, and is extended by the Actor interface. An *actor* is an executable entity. There are two types of actors, AtomicActor, which extends ComponentEntity, and CompositeActor, which extends CompositeEntity. As the names imply, an AtomicActor is a single entity, while a CompositeActor is an aggregation of actors. Two further extensions implement a type system, TypedAtomicActor and TypedCompositeActor.

The Executable interface defines how an object can be invoked. There are seven methods. The initialize() method is assumed to be invoked exactly once during the lifetime of an execution of a model. It may be invoked again to restart an execution. The prefire(), fire(), and postfire() methods will usually be invoked many times. The fire() method may be invoked several times between invocations of prefire() and postfire(). The stopFire() method is invoked to request suspension of firing. The wrapup() method will be invoked exactly once per execution, at the end of the execution.

The terminate() method is provided as a last-resort mechanism to interrupt execution based on an external event. It is not called during the normal flow of execution. It should be used only to stop runaway threads that do not respond to more usual mechanism for stopping an execution.

An *iteration* is defined to be one invocation of prefire(), any number of invocation of fire(), and one invocation of postfire(). An *execution* is defined to be one invocation of initialize(), followed by any number of iterations, followed by one invocation of wrapup(). The methods initialize(), prefire(), fire(), postfire(), and wrapup() are called the *action methods*. While, the action methods in the execut-

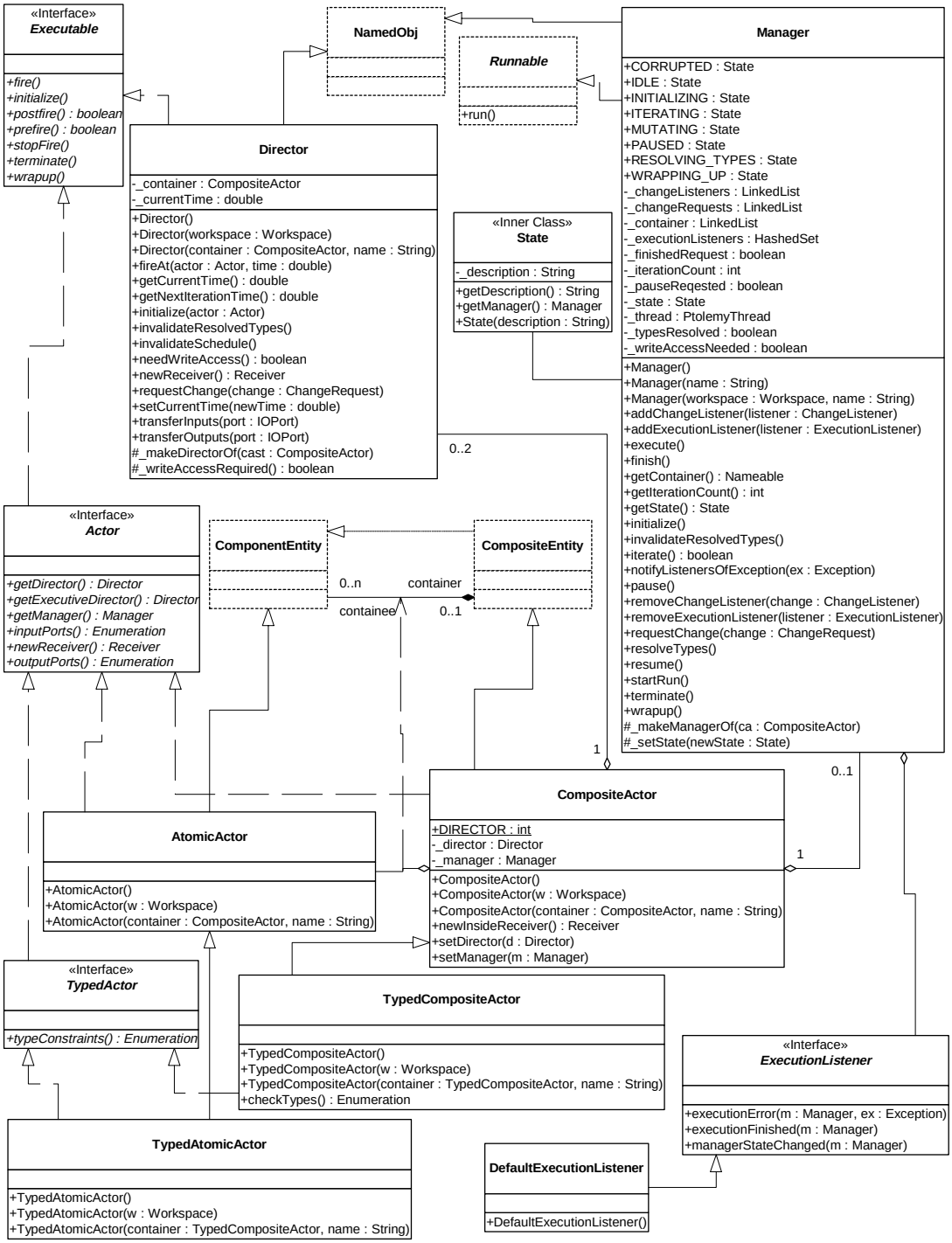


FIGURE 6.12. Basic classes in the actor package that support execution.

able interface are executed in order during the normal flow of an iteration, the terminate() method can be executed at any time, even during the execution of the other methods.

The initialize() method of each actor gets invoked exactly once, much like the begin() method in Ptolemy Classic. Typical actions of the initialize() method include creating and initializing private data members. In domains that use typed ports and/or schedulers, type resolution and scheduling has not been performed when initialize() is invoked. Thus, the initialize() method may define the types of the ports and may set parameters that affect scheduling.

The prefire() method may be invoked multiple times during an execution, but only once per iteration. The prefire() returns true to indicate that the actor is ready to fire. In other words, a return value of true indicates "you can safely invoke my fire method," while a false value from prefire means "My preconditions for firing are not satisfied. Call prefire again later when conditions have change." For example, a dynamic dataflow actor might return false to indicate that not enough data is available on the input ports for a meaningful firing to occur.

In opaque composite actors, the prefire() method is responsible for transferring data from the opaque ports of the composite actor to the ports of the contained actors. See section 6.3.5 below.

The fire() method may be invoked multiple times during an iteration. In most domains, this method defines the computation performed by the actor. Some domains will invoke fire() repeatedly until some convergence condition is reached. Thus, fire() should not change the state of the actor. Instead, update the state in postfire().

In some domains, the fire method initiates an open-ended computation. The stopFire() method may be used to request that firing be ended and that the fire() method return as soon as practical.

The postfire() method will be invoked exactly once during an iteration, after all invocations of the fire() method in that iteration. An actor may return false in postfire to request that the actor should not be fired again. It has concluded its mission. However, a director may elect to continue to fire the actor until the conclusion of its own iteration. Thus, the request may not be immediately honored.

The wrapup() method is invoked exactly once during the execution of a model, even if an exception causes premature termination of an execution. Typically, wrapup() is responsible for cleaning up after execution has completed, and perhaps flushing output buffers before execution ends and killing active threads.

The terminate() method may be called at any time during an execution, but is not necessarily called at all. When terminate() is called, no more execution is important, and the actor should do everything in its power to stop execution right away. This method should be used as a last resort if all other mechanisms for stopping an execution fail.

6.3.1 Director

A *director* governs the execution of a composite entity. A *manager* governs the overall execution of a model. An example of the use of these classes is shown in figure 6.13. In that example, a top-level entity, E0, has an instance of Director, D1, that serves the role of its local director. A *local director* is responsible for execution of the components within the composite. It will perform any scheduling that might be necessary, dispatch threads that need to be started, generate code that needs to be generated, etc. In the example, D1 also serves as an executive director for E2. The *executive director* associated with an actor is the director that is responsible for firing the actor.

A composite actor that is not at the top level may or may not have its own local director. If it has a local director, then it defined to be opaque (isOpaque() returns *true*). In figure 6.13, E2 has a local director and E3 does not. The contents of E3 are directly under the control of D1, as if the hierarchy

were flattened. By contrast, the contents of E2 are under the control of D2, which in turn is under the control of D1. In the terminology of the previous generation, Ptolemy Classic, E2 was called a *wormhole*. In Ptolemy II, we simply call it a composite opaque actor. It will be explained in more detail below in section 6.3.5.

We define the *director* (vs. local director or executive director) of an actor to be either its local director (if it has one) or its executive director (if it does not). A composite actor that is not at the top level has as its executive director the director of the container. Every executable actor has a director except the top-level composite actor, and that director is what is returned by the getDirector() method of the Actor interface (see figure 6.12).

When any action method is called on an opaque composite actor, the composite actor will generally call the corresponding method in its local director. This interaction is crucial, since it is domain-independent and allows for communication between different models of computation. When fire() is called in the director, the director is free to invoke iterations in the contained topology until the stopping condition for the model of computation is reached.

The postfire() method of a director returns false to stop its execution normally. It is the responsibility of the next director up in the hierarchy (or the manager if the director is at the top level) to conclude the execution of this director by calling its wrapup() method.

The Director class provides a default implementation of an execution, although specific domains may override this implementation. In order to ensure interoperability of domains, they should stick fairly closely to the sequence.

Two common sequences of method calls between actors and directors are shown in figure 6.14 and 6.15. These differ in the shaded areas, which define the domain-specific sequencing of actor firings. In figure 6.14, the fire() method of the director selects an actor, invokes its prefire() method, and if that returns true, invokes its fire() method some number of times (domain dependent) followed by its postfire() method. In figure 6.15, the fire() method of the director invokes the prefire() method of all the actors before invoking any of their fire() methods.

When a director is initialized, via its initialize() method, it invokes initialize() on all the actors in the next level of the hierarchy, in the order in which these actors were created. The wrapup() method works in a similar way, *deeply* traversing the hierarchy. In other words, calling initialize() on a composite actor is guaranteed to initialize in all the objects contained within that actor. Similarly for wrapup().

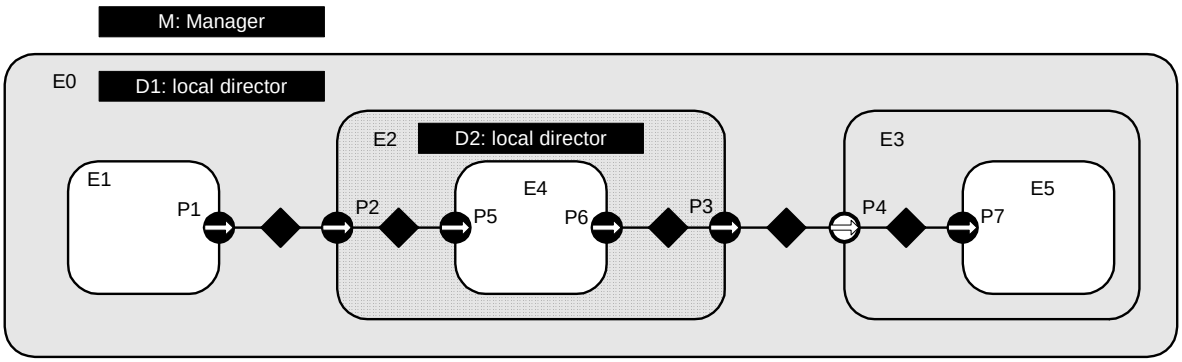


FIGURE 6.13. Example application, showing a typical arrangement of actors, directors, and managers.

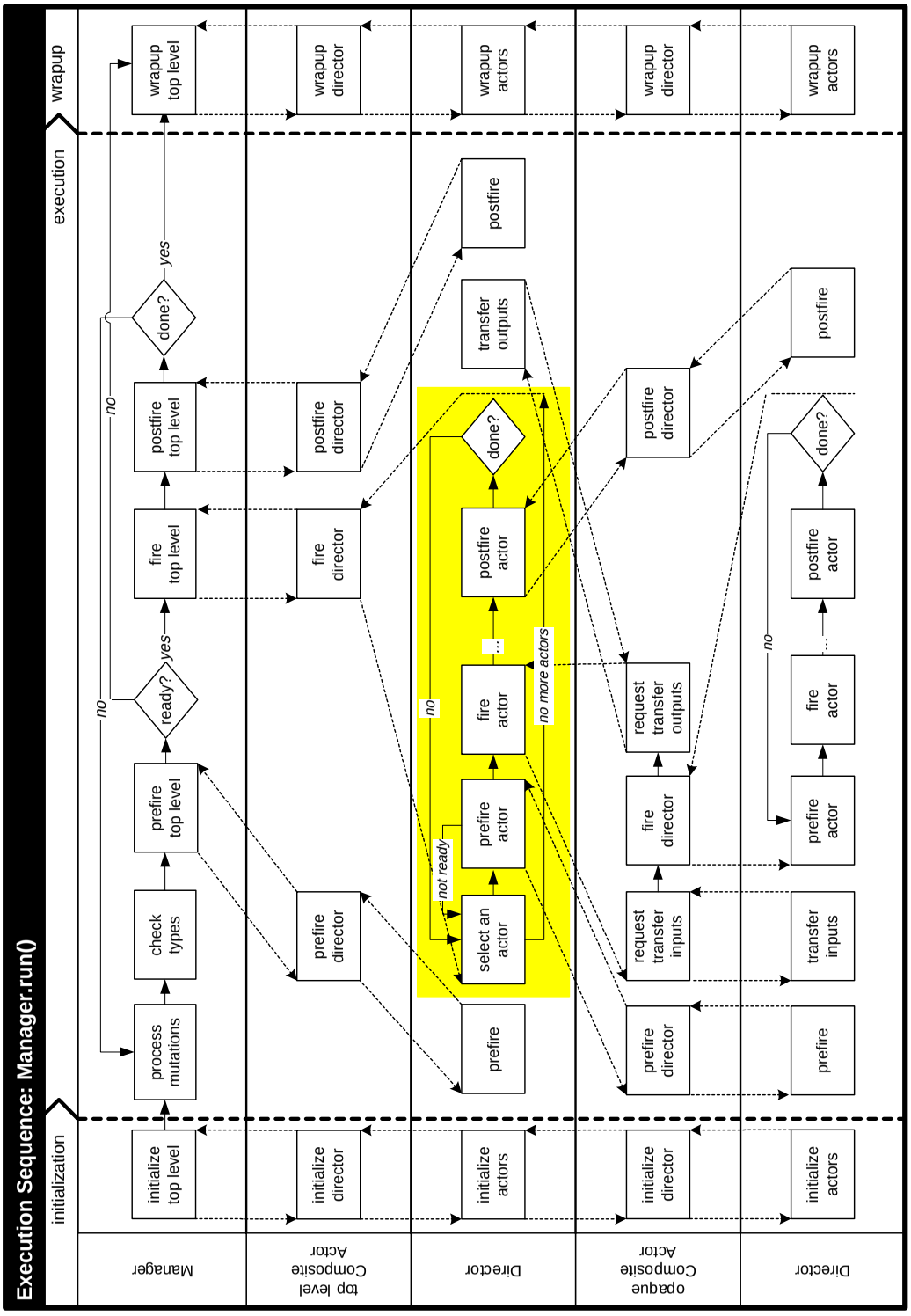


FIGURE 6.14. Example execution sequence implemented by run() method of the Director class.

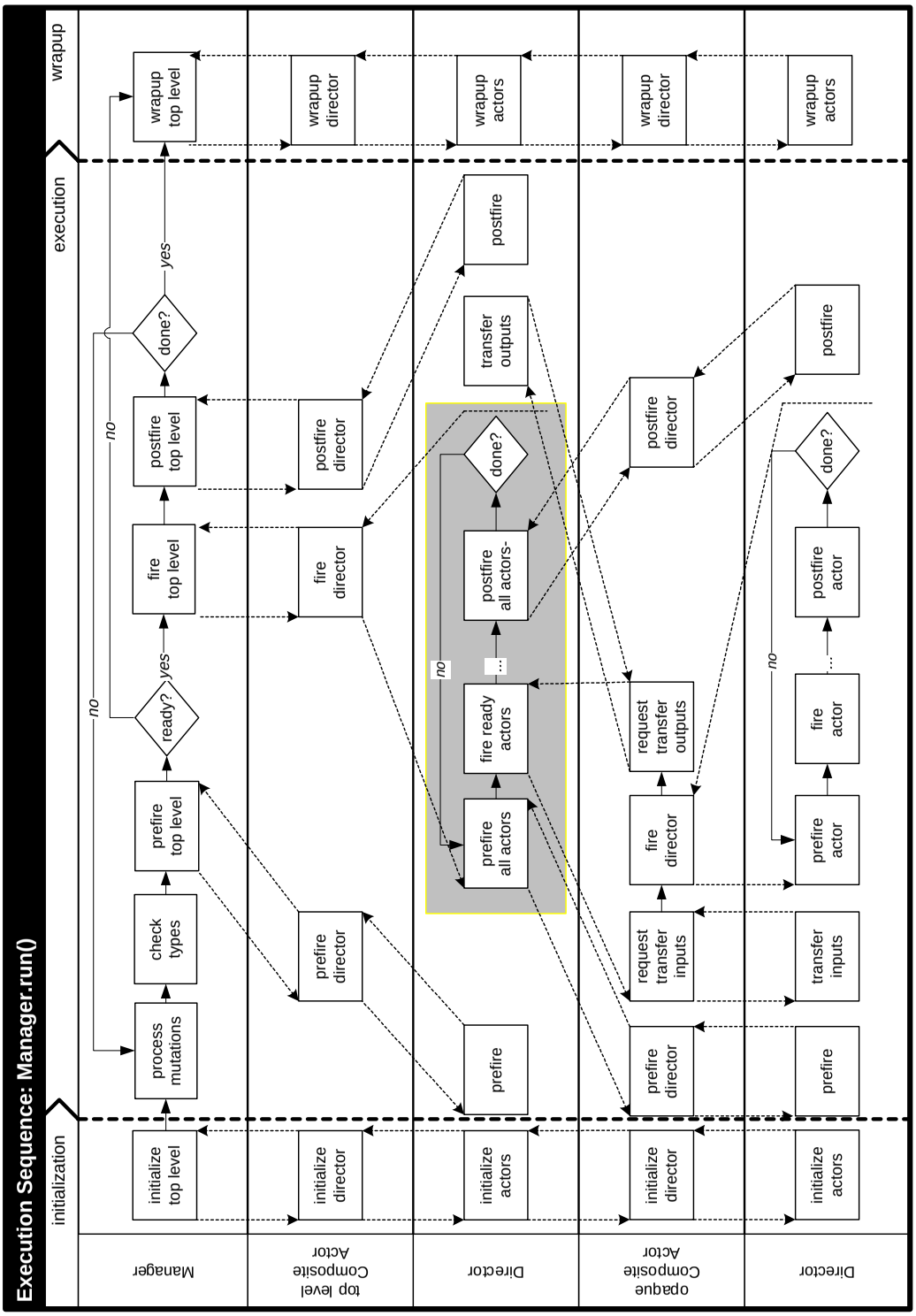


FIGURE 6.15. Alternative execution sequence implemented by run() method of the Director class.

The methods prefire() and postfire(), on the other hand, are not deeply traversing functions. Calling prefire() on a director does not imply that the director call prefire() on all its actors. Some directors may need to call prefire() on some or all contained actors before being able to return, but some directors may not need to call prefire() on any contained objects at all. A director may even implement short-circuit evaluation, where it calls prefire() on only enough of the contained actors to determine its own return value. Postfire() works similarly, except that it may only be called after at least one successful call to fire().

The fire() method is where the bulk of work for a director occurs. When a director is fired, it has complete control over execution, and may initiate whatever iterations of other actors are appropriate for the model of computation that it implements. It is important to stress that once a director is fired, outside objects do not have control over when the iteration will complete. The director may not iterate any contained actors at all, or it may iterate the contained actors forever, and not stop until terminate() is called. Of course, in order to promote interoperability, directors should define a finite execution that they perform in the fire() method.

In case it is not practical for the fire() method to define a bounded computation, the stopFire() method is provided. A director should respond when this method is called by returning from its fire() method as soon as practical.

In some domains, the firing of a director corresponds exactly to the sequential firing of the contained actors in a specific predetermined order. This ordering is known as a *static schedule* for the actors. Some domains support this style of execution. There is also a family of domains where actors are associated with threads.

6.3.2 Manager

While a director implements a model of computation, a *manager* controls the overall execution of a model. The manager interacts with a single composite actor, known as a *top level composite actor.* The Manager class is shown in figure 6.12. Execution of a model is implemented by three methods, execute(), run() and startRun(). The startRun() method spawns a thread that calls run(), and then immediately returns. The run() method calls execute(), but catches all exceptions and reports them to listeners (if there are any) or to the standard output (if there are no listeners).

More fine grain control over the execution can be achieved by calling initialize(), iterate(), and wrapup() on the manager directly. The execute() method, in fact, calls these, repeating the call to iterate() until it returns false. The iterate method invokes prefire(), fire() and postfire() on the top-level composite actor, and returns false if the postfire() in the top-level composite actor returns false.

An execution can also be ended by calling terminate() or finish() on the manager. The terminate() method triggers an immediate halt of execution, and should be used only if other more graceful methods for ending an execution fail. It will probably leave the model in an inconsistent state, since it works by unceremoniously killing threads. The finish() method allows the system to continue until the end of the current iteration in the top-level composite actor, and then invokes wrapup(). Finish() encourages actors to end gracefully by calling their stopFire() method.

Execution may also be paused between top-level iterations by calling the pause() method. This method sets a flag in the manager and calls stopFire() on the top-level composite actor. After each top-level iteration, the manager checks the flag. If it has been set, then the manager will not start the next top-level iteration until after resume() is called. In certain domains, such as the process networks domain, there is not a very well defined concept of an iteration. Generally these domains do not rely on repeated iteration firings by the manager. The call to stopFire() requests of these domains that they sus-

pend execution.

6.3.3 ExecutionListener

The ExecutionListener interface provides a mechanism for a manager to report events of interest to a user interface. Generally a user interface will use the events to notify the user of the progress of execution of a system. A user interface can register one or more ExecutionListeners with a manager using the method addExecutionListener() in the Manager class. When an event occurs, the appropriate method will get called in all the registered listeners.

Two kinds of events are defined in the ExecutionListener interface. A listener is notified of the completion of an execution by the executionFinished() method. The executionError() method indicates that execution has ended with an error condition. The managerStateChanged() indicates to the listener that the manager has changed state. The new state can be obtained by calling getState() on the manager.

A default implementation of the ExecutionListener interface is provided in the DefaultExecutionListener class. This class reports all events on the standard output.

6.3.4 Mutations

A *mutation* is a run-time modification of a model. In most domains, it is not safe for mutations to occur at arbitrary times during an execution. For example, a schedule may need to be recalculated to take into account the mutation. Type resolution may need to be redone.

The Director and Manager classes leverage the event subpackage of the kernel, which provides support for requesting and tracking changes in the topology. This support is documented in the Kernel chapter. The general strategy in Director is simple. Any code that wishes to perform a mutation queues that mutation with the director or manager rather than performing it directly (using the requestChange() method, shown in figure 6.12). When it is safe, that mutation is performed, and all change listeners that have been registered with the director (using the addChangeListener() method of Manager) are informed of the mutation. Most directors delegate the implementation of mutations to the manager, which performs them between iterations.

For convenience, certain kinds of common mutations are supported by concrete base classes of ChangeRequest, as shown in figure 6.16. These classes can be instantiated and the queued with a director or manager using their requestChange() method. They can also be grouped into a ChangeList so that they will execute all at once. Further details about mutations are covered in the Kernel chapter.

6.3.5 Opaque Composite Actors

One of the key features of Ptolemy II is its ability to hierarchically mix models of computation in a disciplined way. The way that it does this is to have actors that are composite (non-atomic) and opaque. Such an actor was called a *wormhole* in the earlier generation of Ptolemy. Its ports are opaque and its contents are not visible via methods like deepGetEntities().

Recall that an instance of CompositeActor that is at the top level of the hierarchy must have a local director in order to be executable. A CompositeActor at a lower level of the hierarchy may also have a local director, in which case, it is opaque (isOpaque() returns *true*). It also has an executive director, which is simply the director of its container. For a composite opaque actor, the local director and executive director need not follow the same model of computation. Hence hierarchical heterogeneity.

The ports of a composite opaque actor are opaque, but it is a composite (it can contain actors and

relations). This has a number of implications on execution. Consider the simple example shown in figure 6.17. Assume that both E0 and E2 have local directors (D1 and D2), so E2 is opaque. The ports of E2 therefore are opaque, as indicated in the figure by their solid fill. Since its ports are opaque, when a token is sent from the output port P1, it is deposited in P2, not P5.

In the execution sequences of figures 6.14 and 6.15, E2 is treated as an atomic actor by D1; i.e. D1 acts as an executive director to E2. Thus, the fire() method of D1 invokes the prefire(), fire(), and postfire() methods of E1, E2, and E3. The fire() method of E2 is responsible for transferring the token from P2 to P5. It does this by delegating to its local director, invoking its transferInputs() method. It then invokes the fire() method of D2, which in turn invokes the prefire(), fire(), and postfire() methods of E4.

During its fire() method, E2 will invoke the fire() method of D2, which typically will fire the actor E4, which may send a token via P6. Again, since the ports of E2 are opaque, that token goes only as far as P3. The fire() method of E2 is then responsible for transferring that token to P4. It does this by delegating to its *executive* director, invoking its transferOutputs() method.

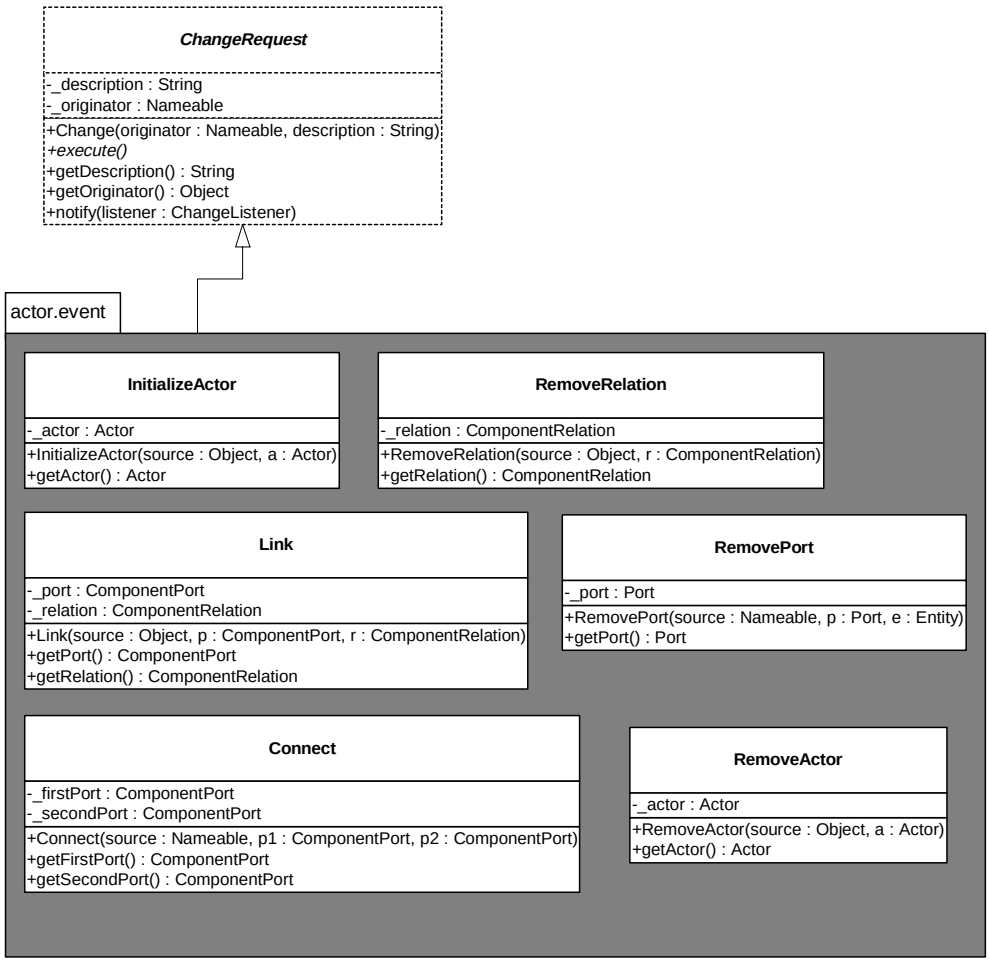


FIGURE 6.16. A set of concrete base classes of ChangeRequest provided in the actor.event package.

The CompositeActor class delegates transfer of its inputs to its local director, and transfer of its outputs to its executive director. This is the correct organization, because in each case, the director appropriate to the model of computation of the destination port is the one handling the transfer. It can therefore handle it in a manner appropriate to the receiver in that port.

Note that the port P3 is an output, but it has to be capable of receiving data from the inside, as well as sending data to the outside. Thus, despite being an output, it contains a receiver. Such a receiver is called an *inside receiver*. The methods of IOPort offer only limited access to the inside receivers (only via the getInsideReceivers() method and getReceivers(*relation*), where *relation* is an inside linked relation).

In general, a port may be both an input and an output. An opaque port of a composite opaque actor, thus, must be capable of storing two distinct types of receivers, a set appropriate to the inside model of computation, obtained from the local director, and a set appropriate to the outside model of computation, obtained from its executive director. Most methods that access receivers, such as hasToken() or hasRoom(), refer only to the outside receivers. The use of the inside receivers is rather specialized, only for handling composite opaque actors, so a more basic interface is sufficient.

6.3.6 Scheduler and Process Support

The actor package has two subpackages, actor.sched, which provides rudimentary support for domains that use static schedulers to control the invocation of actors, and actor.process, which provides support for domains where actors are processes. The UML diagram is shown in figure 6.18.

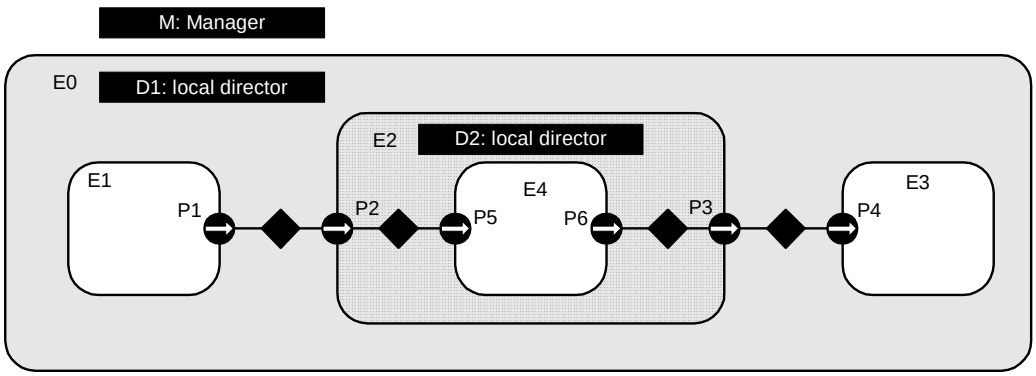


FIGURE 6.17. An example of an opaque composite actor. E0 and E2 both have local directors, not necessarily implementing the same model of computation.

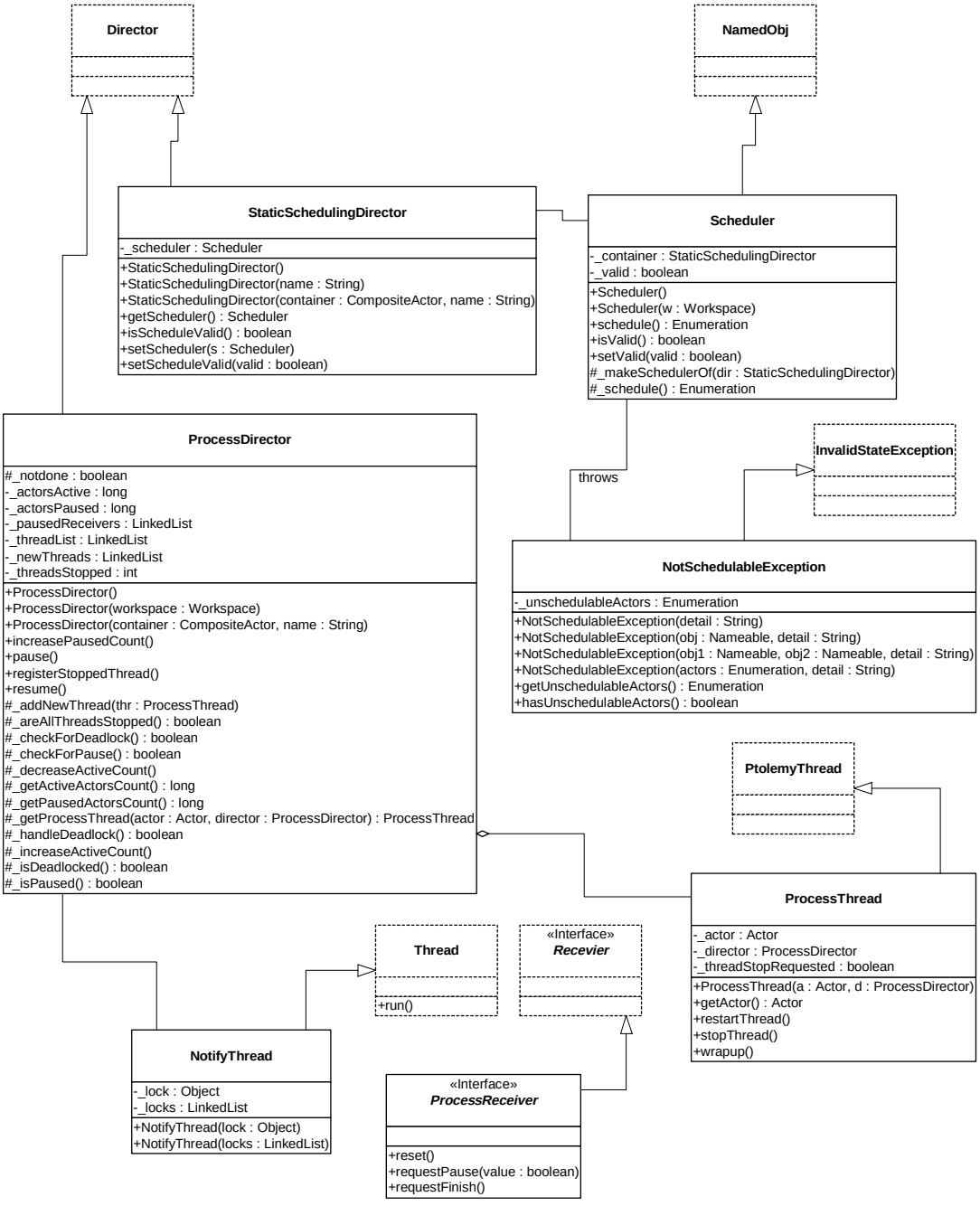


FIGURE 6.18. UML static structure diagram for the actor.sched and actor.process packages.

7

Data Package

Authors: Neil Smyth
Yuhong Xiong
Edward A. Lee

7.1 Introduction

The data package provides data encapsulation, polymorphism, parameter handling, an expression language, and a type system. Figure 7.1 shows the key classes in the main package (subpackages will be discussed later).

7.2 Data Encapsulation

The Token class and its derived classes encapsulate application data. The encapsulated data can be transported via message passing between Ptolemy II objects. Alternatively, it can be used to parameterize Ptolemy II objects. Encapsulating the data in such a way provides a standard interface so that such data can be handled uniformly regardless of its detailed structure. Such encapsulation allows for a great degree of extensibility, permitting developers to extend the library of data types that Ptolemy II can handle. It also permits a user interface to interact with application data without detailed prior knowledge of the structure of the data.

Tokens in Ptolemy II, except ObjectToken, are immutable. This means that their value cannot be changed after the instance of Token is constructed. The value of a token must therefore be specified as a constructor argument, and there must be no other mechanism for setting the value. If the value must be changed, then a new instance of Token must be constructed.

There are several reasons for making tokens immutable.

- First, when a token is to be sent to several receivers, we want to be sure that all receivers get the same data. Each receiver is sent a reference to the same token. If the Token were not immutable, then it would be necessary to clone the token for all receivers after the first one.
- Second, we use tokens to parameterize objects, and parameters have mutual dependencies. That is,

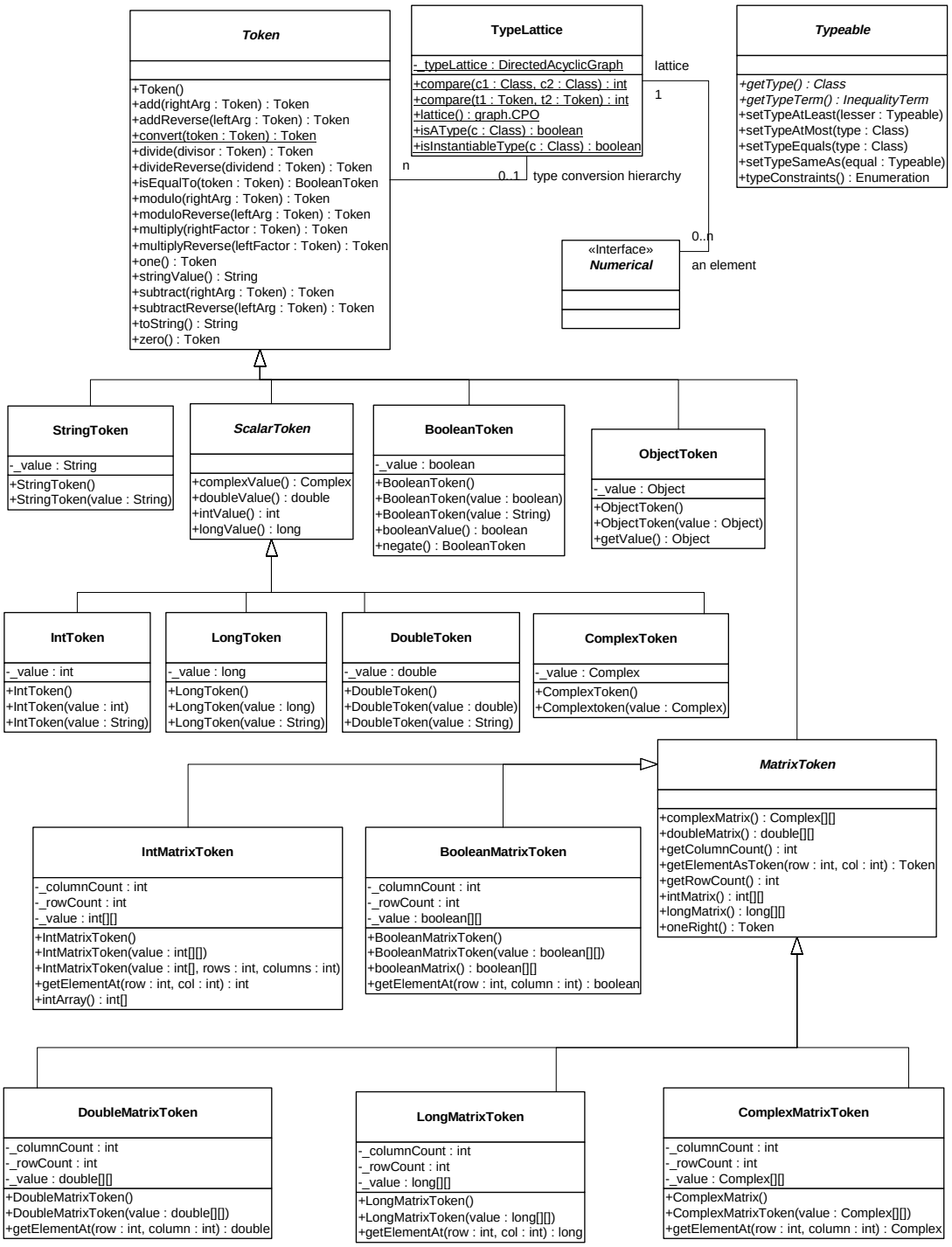


FIGURE 7.1. Static Structure Diagram (Class Diagram) for the classes in the data package.

the value of a parameter may depend on the value of other parameters. The value of a parameter is represented by an instance of Token. If that token were allowed to change value without notifying the parameter, then the parameter would not be able to notify other parameters that depend on its value. Thus, a mutable token would have to implement a publish-and-subscribe mechanism so that parameters could subscribe and thus be notified of any changes. By making tokens immutable, we greatly simplify the design.

- Finally, having our Tokens immutable makes them similar in concept to the data wrappers in Java, like Double, Integer, etc., which are also immutable.

An ObjectToken contains a reference to an arbitrary Object created by the user. Since the user may modify the Object after the token is constructed, ObjectToken is an exception to immutability. Moreover, the getValue() method returns a reference to the token. That reference can be used to modify the object. Although ObjectToken could clone the object in the constructor and return another clone in getValue(), this would require the object to be cloneable, which severely limits the use of the ObjectToken. In addition, even if the object is cloneable, since the default implementation of clone() only makes a shallow copy, it is still not enough to enforce immutability. In addition, cloning a large object could be expensive. For these reasons, the ObjectToken does not enforce immutability, but rather relies on the cooperation of the user. Violating this convention could lead to unintended non-determinism.

For matrix tokens, immutability requires the contained matrix (Java array) to be copied when the token is constructed, and when the matrix is returned in response to the queries such as intMatrix(), doubleMatrix(), etc. This is because arrays are objects in Java. Since the cost of copying large matrix is non-trivial, the user should not make more queries then necessary. The getElementAt() method should be used to read the contents of the matrix.

7.3 Polymorphism

7.3.1 Polymorphic Arithmetic Operators

One of the goals of the data package is to support polymorphic operations between tokens. For this, the base Token class defines methods to overload the primitive arithmetic operations, which are add(), multiply(), subtract(), divide(), modulo() and equals(). Derived classes overload these methods to provide class specific operation where appropriate. The objective here is to be able to say, for example,

```
a.add(b)
```

where a and b are arbitrary tokens. If the operation a + b makes sense for the particular tokens, then the operation is carried out and a token of the appropriate type is returned. If the operation does not make sense, then an exception is thrown. Consider the following example

```
IntToken a = new IntToken(5);
DoubleToken b = new DoubleToken(2.2);
StringToken c = new StringToken(“hello”);
```

then

```
a.add(b)
```

gives a new DoubleToken with value 7.2,

```
a.add(c)
```

gives a new StringToken with value “5Hello”, and

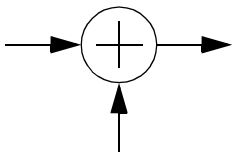
```
a.modulo(c)
```

throws an exception. Thus in effect we have overloaded the operators +, -, *, /, % and ==.

It is not always immediately obvious what is the correct implementation of an operation and what the return type should be. For example, the result of adding an integer token to a double-precision floating-point token should probably be a double, not an integer. The mechanism for making such decisions depends on a *type hierarchy* that is defined separately from the class hierarchy. This type hierarchy is explained in detail below.

The token classes also implement the methods zero() and one() which return the additive and multiplicative identities respectively. These methods are overridden so that each token type returns a token of its type with the appropriate value. For numerical matrix tokens, zero() returns a zero matrix whose dimension is the same as the matrix of the token where this method is called; and one() returns the left identity, i.e., it returns an identity matrix whose dimension is the same as the number of rows of the matrix of the token. Another method oneRight() is also provided in numerical matrix tokens, which return the right identity, i.e., the dimension is the same as the number of columns of the matrix in the token.

Since data is transferred between entities using Tokens, it is straightforward to write polymorphic actors that receive tokens on their inputs, perform one or more of the overloaded operations and output the result. For example an `add` actor that looks like this:



might contain some code like:

```
Token input1, input2, output;
// read Tokens from the input channels into input1 and input2 variables
output = input1.add(input2);
// send the output Token to the output channel.
```

We call such actors *data polymorphic* to contrast them from *domain polymorphic* actors, which are actors that can operate in multiple domains. Of course, an actor may be both data and domain polymorphic.

7.3.2 Lossless Type Conversion

For the above arithmetic operations, if the two tokens being operated on have different types, type conversion is needed. In Ptolemy II, only conversions that do not lose information are implicitly performed. Lossy conversions must be explicitly done by the user, either through casting or by other means. The lossless type conversion relation among different token types is modeled as a partially ordered set called the *type lattice*, shown in figure 7.2. In that diagram, type *A* is *greater than* type *B* if there is a path upwards from *B* to *A*. Thus, ComplexMatrix is greater than Int. Type *A* is *less than* type *B* if there is a path downwards from *B* to *A*. Thus, Int is less than ComplexMatrix. Otherwise, types *A*

and *B* are *incomparable*. Complex and Long, for example, are incomparable.

In the type lattice, a type can be losslessly converted to any type greater than it. This hierarchy is related to the inheritance hierarchy of the token classes in that a subclass is always less than its super class in the type lattice. However, some adjacent types in the lattice are not related by inheritance.

This hierarchy is realized by the TypeLattice class. Each element in the lattice is an instance of the Java class Class corresponding to a token type. The top element, *General*, which is "the most general type", is represented by the base class Token; the bottom element, *NaT* (Not a Type), is represented by java.lang.Void.TYPE. The TypeLattice class provides methods to compare two token types.

Two of the types, *Numerical* and *Scalar*, are abstract. They cannot be instantiated. This is indicated in the type lattice by italics.

Type conversion is done by the static method convert() in the token classes. This method converts the argument into an instance of the class implementing this method. For example, DoubleToken.convert(Token token) converts the specified token into an instance of DoubleToken. The convert() method can convert any token immediately below it in the type hierarchy into an instance of its own class. If the argument is higher in the type hierarchy, or is incomparable with its own class, convert() throws an exception. If the argument to convert() is already an instance of its own class, it is returned without any

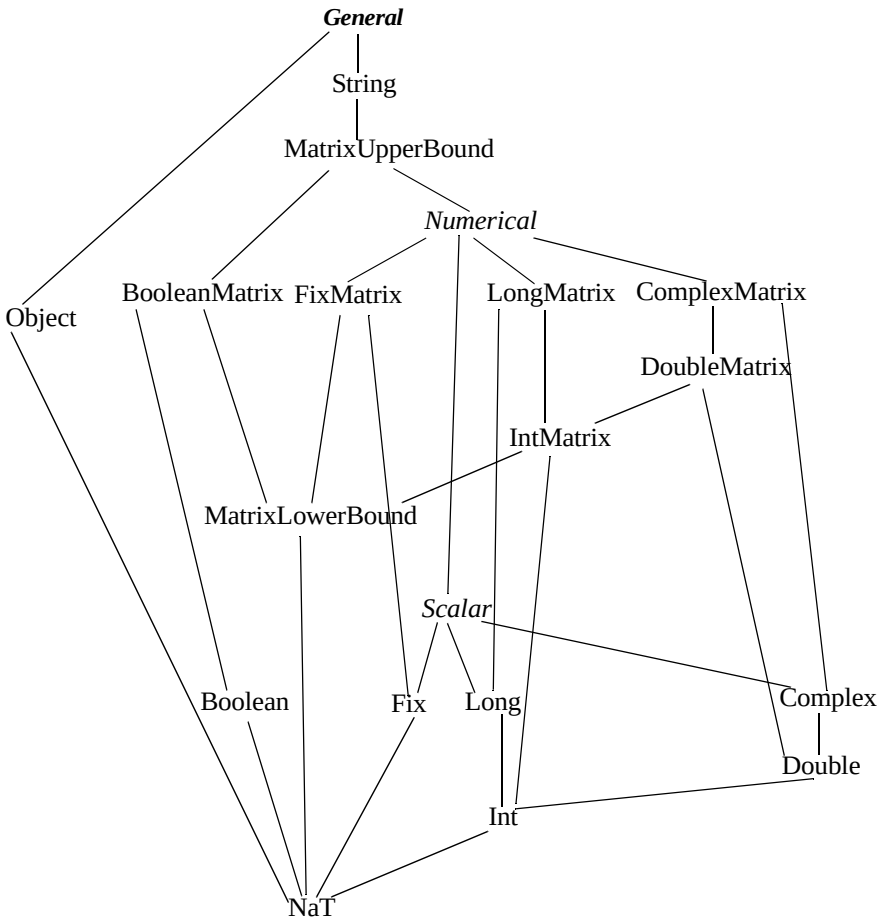


FIGURE 7.2. The type lattice.

change.

The implementation of the add(), subtract(), multiply(), divide(), modulo(), and equals() methods requires that the type of the argument and the implementing class be comparable in the type hierarchy. If this condition is not met, these methods will throw an exception. If the type of the argument is lower than the type of the implementing class, then the argument is converted to the type of the implementing class before the operation is carried out.

The implementation is more involved if the type of the argument is higher than the implementing class, in which case, the conversion must be done in the other direction. Since the `convert()` method only knows how to convert types lower in the type hierarchy up, the operation must take place in the class of the argument. Furthermore, since many of the supported operations are not commutative, for example, `"Hello" + "world"` is not the same as `"world" + "Hello"`, and 3-2 is not the same as 2-3, the implementation of the arithmetic operations cannot simply call the same method on the class of the argument. Instead, a separate set of methods must be used. These methods are addReverse(), subtractReverse(), multiplyReverse(), divideReverse(), and moduloReverse(). The equality check is always commutative so no equalsReverse() is needed. Under this setup, a.add(b) means $a+b$, and a.addReverse(b) means $b+a$, where *a* and *b* are both tokens. If, for example, when a.add(b) is invoked and the type of *b* is higher than *a*, the add() method of *a* will automatically call b.addReverse(a) to carry out the addition.

For scalar and matrix tokens, methods are also provided to convert the content of the token into another numeric type. In ScalarToken, these methods are intValue(), longValue(), doubleValue(), and ComplexValue() (fixValue() will be added later). In MatrixToken, the methods are intMatrix(), long-Matrix(), doubleMatrix(), and ComplexMatrix() (fixMatrix() will be added later). The default implementation in these two base classes just throw an exception. Derived classes override the methods if the corresponding conversion is lossless, returning a new instance of the appropriate class. For example, IntToken overrides all the methods defined in ScalarToken, but DoubleToken does not override intValue(). A double cannot, in general, be losslessly converted to an integer.

7.3.3 Limitations

As of this writing, the following issues remain open:

- FixToken and FixMatrixToken classes are planned for supporting fixed-point computation, but they do not exist yet.
- For numerical matrix tokens, only the add() and addReverse() methods are supported; other arithmetic operations are not implemented yet.

7.4 Variables and Parameters

In Ptolemy II, any instance of NamedObj can have attributes, which are instances of the Attribute class. A *variable* is an attribute that contains a token. Its value can be specified by an expression that can refer to other variables. A *parameter* is identical to a variable, but realized by instances of the Parameter class, which is derived from Variable and adds no functionality. See figure 7.3.

The reason for having two classes with identical interfaces and functionality, Variable and Parameter, is that their intended use is different. Parameters are meant to visible to the end user of a component, whereas variables are meant to operate behind the scenes, unseen. A GUI, for example, might present parameters for editing, but not variables.

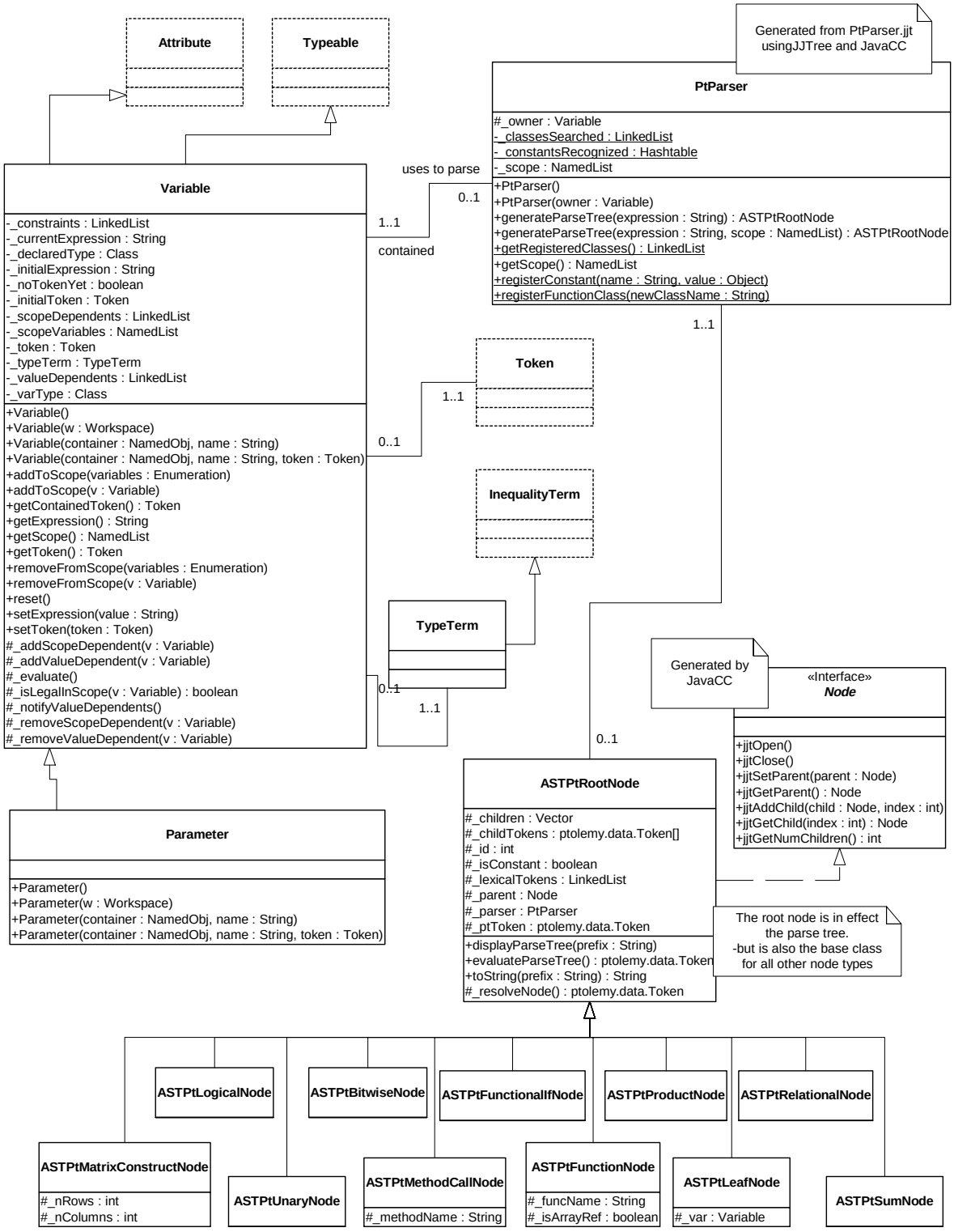


FIGURE 7.3. Static structure diagram for the data.expr package.

7.4.1 Values

The value of a variable can be specified by a token passed to a constructor, a token set using the setToken() method, or an expression set using the setExpression() method.

When the value of a variable is set by setExpression(), the expression is not actually evaluated until you call getToken() or getType(). This is important, because it implies that a set of interrelated expressions can be specified in any order. Consider for example the sequence:

```
Variable v3 = new Variable(container,"v3");
Variable v2 = new Variable(container,"v2");
Variable v1 = new Variable(container,"v1");
v3.setExpression("v1 + v2");
v2.setExpression("1.0");
v1.setExpression("2.0");
v3.getToken();
```

Notice that the expression for `v3` cannot be evaluated when it is set because `v2` and `v1` do not yet have values. But there is no problem because the expression is not evaluated until getToken() is called. Obviously, an expression can only reference variables that are added to the scope of this variable before the expression is evaluated (i.e., before getToken() is called). Otherwise, getToken() will throw an exception. By default, all variables contained by the same container, and those contained by the container's container, are in the scope of this variable. Thus, in the above, all three variables are in each other's scope because they belong to the same container. This is why the expression `"v1 + v2"` can be evaluated.

A variable can also be reset. If the variable was originally set from a token, then this token is placed again in the variable, and the type of the variable is set to equal that of the token. If the variable was originally given an expression, then this expression is placed again in the variable (but not evaluated), and the type is reset to null. The type will be determined when the expression is evaluated or when type resolution is done.

7.4.2 Types

Ptolemy II, in contrast to Ptolemy Classic, does not have a plethora of type-specific parameter classes. Instead, a parameter has a type that reflects the token it contains. You can constrain the allowable types of a parameter or variable using the following mechanisms:

- You can require the variable to have a specific type. Use the setTypeEquals() method.
- You can require the type to be at most some particular type in the type hierarchy (see the Type System chapter to see what this means).
- You can constrain the type to be the same as that of some other object that implements the Typeable interface.
- You can constrain the type to be at least that of some other object that implements the Typeable interface.

Except for the first type constraint, these are not checked by the Variable class. They must be checked by a type resolution algorithm, which is implemented in the graph package.

The type of the variable can be specified in a number of ways, all of which require the type to be consistent with the specified constraints (or an exception will be thrown):

- It can be set directly by a call to setTypeEquals(). If this call occurs after the variable has a value, then the specified type must be compatible with the value. Otherwise, an exception will be thrown. Type resolution will not change the type set through setTypeEquals() unless the argument of that call is null. If this method is not called, or called with a null argument, type resolution will resolve the variable type according to all the type constraints. Note that when calling setTypeEquals() with a non-null argument while the variable already contains a non-null token, the argument must be a type no less than then the type of the contained token. To set type of the variable lower than the type of the currently contained token, setToken() must be called with a null argument before setTypeEquals().
- Setting the value of the variable to a non-null token constrains the variable type to be no less than the type of the token. This constraint will be used in type resolution, together with other constraints.
- The type is also constrained when an expression is evaluated. The variable type must be no less than the type of the token the expression is evaluated to.
- If the variable does not yet have a value, then the type of a variable may be determined by type resolution. In this case, a set of type constraints is derived from the expression of the variable (which presumably has not yet been evaluated, or the type would be already determined). Additional type constraints can be added by calls to the setTypeAtLeast() and setTypeSameAs() methods.

Subject to specified constraints, the type of a variable can be changed at any time. Some of the type constraints, however, are not verified until type resolution is done. If type resolution is not done, then these constraints are not enforced. Type resolution is normally done by the Manager that executes a model.

The type of the variable may change when setToken() or setExpression() is called.

- If no expression, token, or type has been specified for the variable, then the type becomes that of the current value being set.
- If the variable already has a type, and the value can be converted losslessly into a token of that type, then the type is left unchanged.
- If the variable already has a type, and the value cannot be converted losslessly into a token of that type, then the type is changed to that of the current value being set.

If the type of a variable is changed after having once been set, the container is notified of this by calling its attributeTypeChanged() method. If an attribute does not allow type changes, it should throw an exception in this method. If the value is changed after having once been set, then the container is notified of this by calling its attributeChanged() method. If the new value is unacceptable to the container, it should throw an exception. The old value will be restored.

The token returned by getToken() is always an instance of the class given by the getType() method. This is not necessarily the same as the class of the token that was inserted via setToken(). It might be a distinct type if the contained token can be converted losslessly into one of the type given by getType(). In rare circumstances, you may need to directly access the contained token without any conversion occurring. To do this, use getContainedToken().

7.4.3 Dependencies

Expressions set by setExpression() can reference any other variable that is within scope. By default, the scope includes all variables contained by the same container, and all variables contained by the container's container. In addition, any variable can be explicitly added to the scope of a variable by calling addToScope().

When an expression for one variable refers to another variable, then the value of the first variable obviously depends on the value of the second. If the value of the second is modified, then it is important that the value of the first reflect the change. This dependency is automatically handled. When you call getToken(), the expression will be reevaluated if any of the referenced variables have changed values since the last evaluation.

7.5 Expressions

Ptolemy II includes a simple but extensible expression language. This language permits operations on tokens to be specified in a scripting fashion, without requiring compilation of Java code. The expression language can be used to define parameters in terms of other parameters, for example. It can also be used to provide end-users with actors that compute a user-specified expression that refers to inputs and parameters of the actor.

7.5.1 The Ptolemy II Expression Language

The Ptolemy II expression language uses operator overloading, unlike Java. Although we fully agree that the designers of Java made a good decision in omitting operator overloading, our expression language is used in situations where compactness of expressions is extremely important. Expressions often appear in crowded dialog boxes in the user interface, so we cannot afford the luxury of replacing operators with method calls.

The Token classes from the data package form the primitives of the language. For example the number 10 becomes an IntToken with the value 10 when evaluating an expression. Normally this is invisible to the user. The expression language is object-oriented, of course, so methods can be invoked on these primitives. A sophisticated user, therefore, can make use of the fact that "10" is in fact an object to invoke methods of that object.

The expression language is extensible. The basic mechanism for extension is object-oriented. The reflection package in Java is used to recognize method invocations and user-defined constants. We also expect the language to grow over time, so this description should be viewed as a snapshot of its capabilities.

Types. The types currently supported in the language are boolean, complex, double, int, long, string, and matrices. Note that there is no float or byte. Use double or int instead. A long is defined by appending an integer with "l" or "L", as in Java. A complex is defined by appending an "i" or a "j" to a double. This gives a purely imaginary complex number which can then leverage the polymorphic operations in the Token classes to create a general complex number. Thus `2 + 3i` will result in the expected complex number. The expression language supports the same lossless type conversion provided by the Token classes (see section 7.3.2). Lossy conversion has to be done explicitly via a method call.

Arithmetic operators. The arithmetic operators are +, -, *, / and %. These operators, along with ==, are

overloaded, so their implementation depends on the types being operated on. Operator overloading is achieved using the methods in the Token classes. These methods are add(), subtract(), multiply(), divide(), modulo() and equals().

Bit manipulation. The bitwise operators are &, |, ^ and ~. They operated on integers.

Relational operators. The relational operators are <, <=, >, >=, == and !=. They return booleans.

Logical operators. The logical boolean operators are &&, ||, !, & and |. They operate on booleans and return booleans. Note that 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.

Conditionals. The language is an expression language, not an imperative language with sequentially executed statements. Thus, it makes no sense to have the usual `if...then...else...` construct. Such a construct in Java (and most imperative languages) depends on side effects. However, Java does have a functional version of this construct (one that returns a value). The syntax for this is

```
boolean ? value1 : value2
```

If the boolean is true, `value1` is returned, else `value2` is returned. The Ptolemy II expression language uses this same syntax.

Comments. Anything inside **/*...*/** is ignored, as is the rest of a line following **//.** (Expressions can be split over multiple lines).

Variables. Expressions can contain references by name to parameters within the ***scope*** of the expression. Consider a parameter *P* with container *X* which is in turn contained by *Y.* The scope of an expression for *P* includes all the parameters contained by *X* and *Y.* The scope is implemented as an instance of NamedList, which provides a symbol table. Note that a class derived from Parameter may define scope differently.

Constants. If an identifier is encountered in an expression that does not match a parameter in the scope, then it might be a constant which has been registered as part of the language. By default, the constants *PI*, *pi*, *E*, *e*, *true*, *false*, *i*, and *j* are registered, but as we will see later, this can easily be extended by a user. (The constants *i* and *j* are complex numbers with value equal to the 0.0 + 1.0i). In addition, literal constants are supported. Anything between quotes, "...", is interpreted as a string constant. Numerical values without decimal points, such as "10" or "-3" are integers. Numerical values with decimal points, such as "10.0" or "3.14159" are doubles. Integers followed by the character "l" (el) are long integers. 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 a 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]".

Matrix references. Reference to matrices have the form "*name*(*n*, *m*)" where *name* is the name of the matrix variable (or a constant matrix), *n* is the row index, and *m* is the column index. Index numbers start with zero, as in Java, not 1, as in Matlab. With row vectors, it is not necessary to specify both indices. Thus, if *name* = "[1, 3, 5, 7]", then "*name*(2)" will evaluate to 5. To access elements of a column vector, you must specify both indices, so if *name* = "[1; 3; 5; 7]" then *name*(2,0) evaluates to 5.

Functions. The language includes an extensible set of functions, such as sin(), cos(), etc. The functions that are built in include all static methods of the java.lang.Math class and the ptolemy.data.expr.Utili-

tyFunctions class. As we will see below in section 7.5.2, this can easily be extended by a user by registering another class that includes static methods.

Methods. Every element and subexpression in an expression represents an instance of Token (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). The syntax for this is (*token*).*name*(*args*), where *name* 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, this could be used to convert a number to a string as follows

```
(2*4-6.5).stringValue()
```

This returns the string “1.5”. The expression `(2*4-6.5)` evaluates to a double token, and `stringValue()` is a method of DoubleToken.

Note that methods, unlike functions, must take arguments that are of type Token. This is logical because the methods belong to instances of class Token. Functions, however, are implemented as static methods of some other class, such as java.lang.Math. Those classes cannot be expected to define interfaces with Token. Thus, Tokens are converted, if this can be done losslessly, to the type expected by the function.

7.5.2 Functions

By default all of the static methods in java.lang.Math and ptolemy.data.expr.UtilityFunctions are available. The functions currently supported in ptolemy.data.util.UtilityFunctions are:

- reading the string from a file via a readFile("filename") function, and
- calling the parser again to process a String via eval("..."). For example one use of eval() might be eval(readFile("foo.bar")).

Eventually we hope to support calls to external packages such as matlab or tcl via tcl("...") or matlab("..."). Note this only requires that whatever is inside the parenthesis resolve to a StringToken, so we could have for example tcl("puts " + *xx*) where *xx* is a Parameter in the scope. We also plan to support access to system environment variables via env(“name”).

7.5.3 Limitations

The expression language has a rich potential, and only some of this potential has been realized. Here are some of the current limitations:

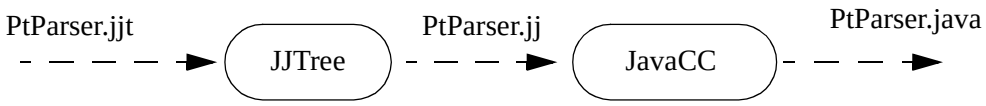
- The class ptolemy.data.util.UtilityFunctions containing the utility functions has not yet been fully written. Currently the functions available are eval() and readFile().
- Functions in the math package need to be supported in much the same way that java.lang.Math is supported.
- Method calls are currently only allowed on tokens in the ptolemy.data package.
- The language does not yet handle matrices or vectors.
- Statements are not supported. It is not clear that they ever will be, since currently the expression language is strictly functional, and converting it to imperative semantics could drastically change its flavor.

Appendix D: Expression Evaluation

The evaluation of an expression is done in two steps. First the expression is parsed to create an *abstract syntax tree* (AST) for the expression. Then the AST is evaluated to obtain the token to be placed in the parameter. In this appendix, “token” refers to instances of the Ptolemy II token classes, as opposed to lexical tokens generated when an expression is parsed.

D.1 Generating the parse tree

In PtolemyII the expression parser, called *PtParser*, is generated using JavaCC and JJTree. JavaCC is a compiler-compiler that takes as input a file containing both the definitions of the lexical tokens that the parser matches and the production rules used for generating the parse tree for an expression. The production rules are specified in *Backus normal form* (BNF). JJTree is a preprocessor for JavaCC that enables it to create an AST. The parser definition is stored in the file PtParser.jjt, and the generated file is PtParser.java. Thus the procedure is



Note that JavaCC generates top-down parsers, or LL(k) in parser terminology. This is different from yacc (or bison) which generate bottom-up parsers, or more formally LALR(1). The JavaCC file also differs from yacc in that it contains both the lexical analyzer and the grammar rules in the same file.

The input expression string is first converted into lexical tokens, which the parser then tries to match using the production rules for the grammar. Each time the parser matches a production rule it creates a node object and places it in the abstract syntax tree. The type of node object created depends on the production rule used to match that part of the expression. For example, when the parser comes upon a multiplication in the expression, it creates an ASTPtProductNode.

The parser takes as input a string, and optionally a NamedList of parameters to which the input expression can refer. That NamedList is the symbol table. If the parse is successful, it returns the root node of the abstract syntax tree (AST) for the given string. Each node object can contain a token, which represents both the type and value information for that node. The type of the token stored in a node, e.g. DoubleToken, IntToken etc., represents the type of the node. The data value contained by the token is the value information for the node. In the AST as it is returned from PtParser, the token types and values are only resolved for the leaf nodes of the tree.

One of the key properties of the expression language is the ability to refer to other tokens by name. Since an expression that refers to other parameters may need to be evaluated several times (when the referred parameter changes), it is important that the parse tree does not need to be recreated every time. When an identifier is parsed, the parser first checks whether it refers to a parameter within the current scope. If it does it creates a ASTPtLeafNode with a reference to that parameter. Note that a leaf node can have a parameter or a token. If it has a parameter then when the token to be stored in this node is evaluated, it is set to the token contained by the parameter. Thus the AST tree does not need to be recreated when a referenced parameter changes as upon evaluation it will just get the new token stored in the referenced parameter. If the parser was created by a parameter, the parameter passes in a reference

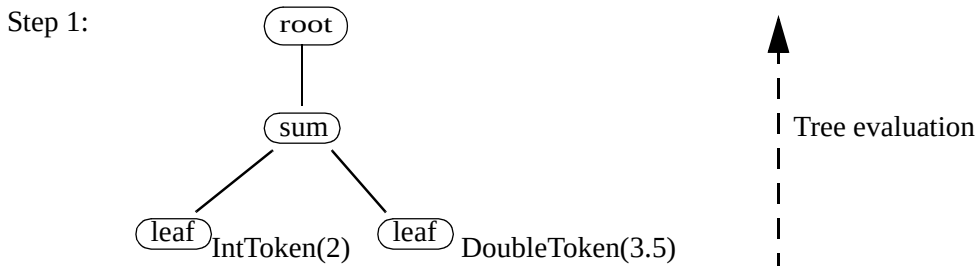
to itself in the constructor. Then upon parsing a reference to another parameter, the parser takes care of registering the parameter that created it as a listener with the referred parameter. This is how dependencies between parameters get registered. There is also a mechanism built into parameters to detect dependency loops.

If the identifier does not refer to a parameter, the parser then checks if it refers to a constant registered with the parser. If it does it creates a node with the token associated with the identifier. If the identifier is neither a reference to a parameter or a constant, an exception is thrown.

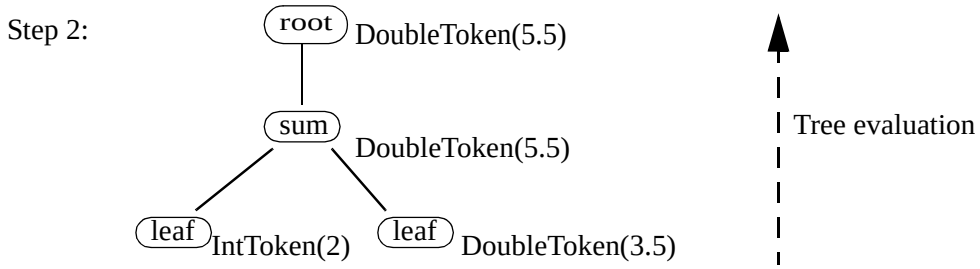
D.2 Evaluating the parse tree

The AST can be evaluated by invoking the method evaluateParseTree() on the root node. The AST is evaluated in a bottom up manner as each node can only determine its type after the types of all its children have been resolved. When the type of the token stored in the root node has been resolved, this token is returned as the result of evaluating the parse tree.

As an example consider the input string 2 + 3.5. The parse tree returned from the parser will look like this:



which will then get evaluated to this:



and DoubleToken(5.5) will be returned as the result.

As seen in the above example, when evaluateParseTree() is invoked on the root node, the type and value of the tokens stored at each node in the tree is resolved, and finally the token stored in the root node is returned. If an error occurs during either the creation of the parse tree or the evaluation of the parse tree, an IllegalArgumentException is thrown with a error message about where the error occurred.

If a node has more than two children, type resolution is done pairwise from the left. Thus “2 + 3 + “hello”” resolves to 5hello. This is the same approach that Java follows.

Each time the parser encounters a function call, it creates an ASTPtFunctionNode object. When this node is being evaluated, it uses reflection to look for that function in the list of classes registered

with the parser for that purpose. The classes automatically searched are java.lang.Math and ptolemy.data.expr.UntilityFunctions. To register another class to be searched when a function call is parsed, call registerFunctionClass() on the parser with the full name of the class to be added to the function search path.

When a parameter is informed that another parameter it references has changed, the parameter re-evaluates the parse tree for the expression to obtain the new value. It is not necessary to parse the expression again as the relevant leaf node stores a reference to the parameter, not the token contained in the parameter. Thus at any time, the value of a parameter is up to date.

D.2.1 Node types

There are currently eleven node classes used in creating the syntax tree. For some of these nodes the types of their children are fairly restricted and so type and value resolution is done in the node. For others, the operators that they represent are overloaded, in which case methods in the token classes are called to resolve the nodes type and value (i.e. the contained token!). By type resolution we are referring to the type of the token to be stored in the node.

ASTPtBitwiseNode. This is created when a bitwise operation(&, |, ^) happens. Type resolution occurs in the node. The & and | operators are only valid between two booleans, or two integer types. The ^ operator is only valid between two integer types.

ASTPtLeafNode. This represents the leaf nodes in the AST. The parser will always place either a token of the appropriate type (e.g. IntToken if “2” is what is parsed) or a parameter in a leaf node. A parameter is placed so that the parse tree can be reevaluated without reparsing whenever the value of the parameter changes. No type resolution is necessary in this node.

ASTPtRootNode. Parent class of all the other nodes. As its name suggests, it is the root node of the AST. It always has only one child, and its type and value is that of its child.

ASTPtFunctionNode. This is created when a function is called. Type resolution occurs in the node. It uses reflection to call the appropriate function with the arguments supplied. It searches the classes registered with the parser for the function. By default it only looks in java.lang.Math and ptolemy.data.expr.UtilityFunctions.

ASTPtFunctionalIfNode. This is created when a functional if is parsed. Type resolution occurs in the node. For a functional if, the first child node must contain a BooleanToken, which is used to chose which of the other two tokens of the child nodes to store at this node.

ASTPtMethodCallNode. This is created when a method call is parsed. Method calls are currently only allowed on tokens in the ptolemy.data package. All of the arguments to the method, and the return type, must be of type Token (or a subclass).

ASTPtProductNode. This is created when a *, / or % is parsed. Type resolution does not occur in the node. It uses the multiply(), divide() and modulo() methods in the token classes to resolve the nodes type.

ASTPtSumNode. This is created when a + or - is parsed. Type resolution does not occur in the node. It uses the add() and subtract() methods in the token classes to resolve the nodes type.

ASTPtLogicalNode. This is created when a && or || is parsed. Type resolution occurs in the node. All children nodes must have tokens of type BooleanToken. The resolved type of the node is also Boolean-

Token.

ASTPtRelationalNode. This is created when one of the relational operators(!=, ==, >, >=, <, <=) is parsed. The resolved type of the token of this node is BooleanToken. The "==" and "!=" operators are overloaded via the equals() method in the token classes. The other operators are only valid on ScalarTokens. Currently the numbers are converted to doubles and compared, this needs to be adjusted to take account of Longs.

ASTPtUnaryNode. This is created when a unary negation operator(!, ~, -) is parsed. Type resolution occurs in the node, with the resulting type being the same as the token in the only child of the node.

D.2.2 Extensibility

The Ptolemy II expression language has been designed to be extensible. The main mechanisms for extending the functionality of the parser is the ability to register new constants with it and new classes containing functions that can be called. However it is also possible to add and invoke methods on tokens, or to even add new rules to the grammar, although both of these options should only be considered in rare situations.

To add a new constant that the parser will recognize, invoke the method registerConstant(String name, Object value) on the parser. This is a static method so whatever constant you add will be visible to all instances of PtParser in the Java virtual machine. The method works by converting, if possible, whatever data the object has to a token and storing it in a hashtable indexed by name. By default, only the constants in java.lang.Math are registered.

To add a new Class to the classes searched for a a function call, invoke the method registerClass(String name) on the parser. This is also a static method so whatever class you add will be searched by all instances of PtParser in the JVM. The name given must be the fully qualified name of the class to be added, for example "java.lang.Math". The method works by creating and storing the Class object corresponding to the given string. If the class does not exist an exception is thrown. When a function call is parsed, an ASTPtFunctionNode is created. Then when the parse tree is being evaluated, the node obtains a list of the classes it should search for the function and, using reflection, searches the classes until it either finds the desired function or there are no more classes to search. The classes are searched in the same order as they were registered with the parser, so it is better to register those classes that are used frequently first. By default, only the classes java.Lang.Math and ptolemy.data.expr.UtilityFunctions are searched.

8

Graph Package

Authors: Jie Liu
Yuhong Xiong

8.1 Introduction

The Ptolemy II kernel provides extensive infrastructure for creating and manipulating clustered graphs of a particular flavor. Mathematical graphs, however, are simpler structures that consist of nodes and edges, without hierarchy. Edges link only two nodes, and therefore are much simpler than the relations of the Ptolemy II kernel. Moreover, in mathematical graphs, no distinction is made between multiple edges that may be adjacent to a node, so the ports of the Ptolemy II kernel are not needed. A large number of algorithms have been developed that operate on mathematical graphs, and many of these prove extremely useful in support of scheduling, type resolution, and other operations in Ptolemy II. Thus, we have created the *graph* package, which provides efficient data structures for mathematical graphs, and collects algorithms for operating on them. At this time, the collection of algorithms is nowhere near as complete as in some widely used packages, such as LEDA. But this package will serve as a repository for a growing suite of algorithms.

The graph package provides basic infrastructure for both undirected and directed graphs. Acyclic directed graphs, which can be used to model complete partial orders (CPOs) and lattices, are also supported with more specialized algorithms.

The graphs constructed using this package are lightweight, designed for fast implementation of complex algorithms more than for generality. This makes them maximally complementary to the clustered graphs of the Ptolemy II kernel, which emphasize generality. A typical use of this package is to construct a graph that represents the topology of a CompositeEntity, run a graph algorithm, and extract useful information from the result. For example, a graph might be constructed that represents data precedences, and a topological sort might be used to generate a schedule. In this kind of application, the hierarchy of the original clustered graph is flattened, so nodes in the graph represent only opaque entities.

The architecture of this package is somewhat different from LEDA, in part because of the exist-

ence of the complementary kernel package. Unlike LEDA, there are no dedicated classes representing nodes and edges in the graph. The nodes in this package are represented by arbitrary instances of the Java Object class, and the graph topology is stored in a structure similar to an adjacency list.

The facilities that currently exist in this package are those that we have had most immediate need for. Since the type system of Ptolemy II requires extensive operations on lattices and CPOs, support for these is better developed than for other types of graphs.

8.2 Classes and Interfaces in the Graph Package

Figure 8.1 shows the class diagram of the graph package. The classes Graph, DirectedGraph and DirectedAcyclicGraph support graph construction and provide graph algorithms. Currently, only topological sort and transitive closure are implemented; other algorithms will be added as needed. The CPO interface defines the basic CPO operations, and the class DirectedAcyclicGraph implements this interface. An instance of DirectedAcyclicGraph is also a finite CPO where all the elements and order relations are explicitly specified. Defining the CPO operations in an interface allows future expansion to support infinite CPOs and finite CPOs where the elements are not explicitly enumerated. The Ine-

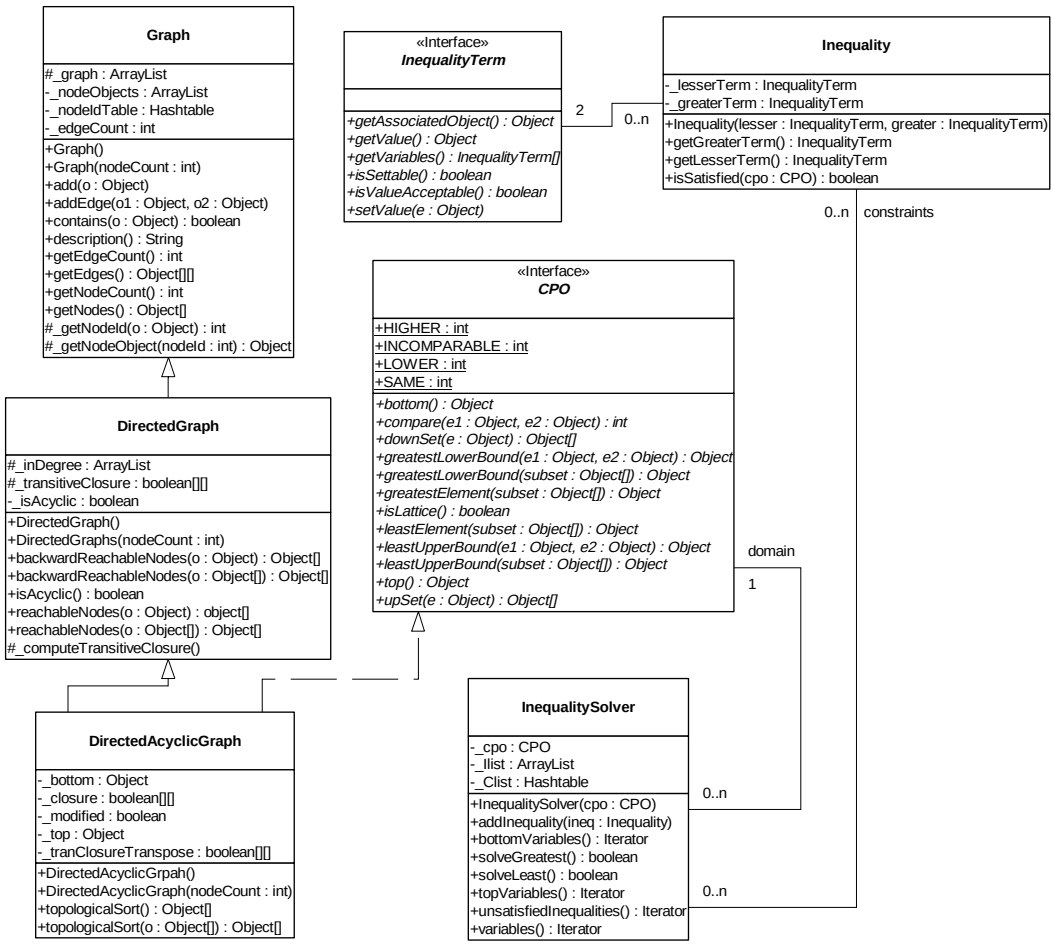


FIGURE 8.1. Classes in the graph package

qualityTerm interface and the Inequality class model inequality constraints over the CPO. The details of the constraints will be discussed later. The InequalitySolver class provides an algorithm to solve a set of constraints. This is used by the Ptolemy II type system, but other uses may arise.

The implementation of the above classes is not synchronized. If multiple threads access a graph or a set of constraints concurrently, external synchronization will be needed.

8.2.1 Graph

This class models a simple undirected graph. Each node in the graph is represented by an arbitrary Java object. The method add() is used to add a node to the graph, and addEdge() is used to connect two nodes in the graph. The arguments of addEdge() are two Objects representing two nodes already added to the graph. To mirror a topology constructed in the kernel package, multiple edges between two nodes are allowed. Each node is assigned a node ID based on the order the nodes are added. The translation from the node ID to the node Object is done by the _getNodeObject() method, and the translation in the other direction is done by _getNodeId(). Both methods are protected. The node ID is only used by this class and the derived classes, it is not exposed in any of the public interfaces. The topology is stored in the Vector _graph. The indexes of this Vector correspond to node IDs. Each entry of _graph is also a Vector, in which a list of node IDs are stored. When an edge is added by calling addEdge() with the first argument having node ID *i* and the second having node ID *j*, an Integer containing *j* is added to the Vector at the *i*-th entry of _graph. For example, if the graph in figure 8.2(a) is connected using the sequence of calls: addEdge(n0, n1); addEdge(n0, n2); addEdge(n2, n1), where n0, n1, n2 are Objects representing the nodes with IDs 0, 1, 2, respectively, then the data structure will be in the form of 8.2(b).

Note that in this undirected graph, the data format is dependent on the order of the two arguments in the addEdge() calls. Since each edge is stored only once, this data structure is not exactly the same as the adjacency list for undirected graphs, but it is quite similar. This structure is designed to be used by subclasses that model directed graphs, as well as by this base class. If it appears awkward when adding algorithms for undirected graph, a new class that derives from Graph may be added in the future to model undirected graph exclusively, in which case, Graph will provide the basic support for both undirected and directed graphs.

8.2.2 Directed Graphs

The DirectedGraph class is derived from Graph. The addEdge() method in DirectedGraph adds a directed edge to the graph. In this package, the direction of the edge is said to go from a *lower* node to

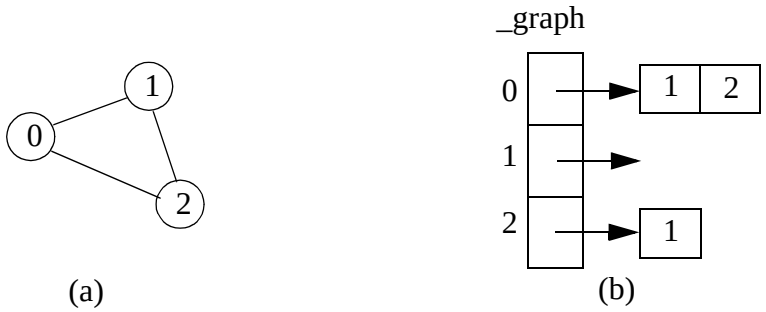


FIGURE 8.2. An undirected graph

a *higher* node, as opposed to from *source* to *sink*, *head* to *tail*, etc. The terms lower and higher conforms with the convention of the graphical representation of CPOs and lattices (the Hasse diagram), so they can be consistently used on both directed graphs and CPOs.

The computation of transitive closure is implemented in this class. The transitive closure is internally stored as a 2-D boolean matrix, whose indexes correspond to node IDs. The entry (*i*, *j*) is *true* if and only if there exists a path from the node with ID *i* to the node with ID *j*. This matrix is not exposed at the public interface; instead, it is used by this class and its subclass to do other operations. Once the transitive closure matrix is computed, graph operations like *reachableNodes* can be easily accomplished.

8.2.3 Directed Acyclic Graphs and CPO

The DirectedAcyclicGraph class further restricts DirectedGraph by not allowing cycles. For performance reasons, this requirement is not checked when edges are added to the graph, but is checked when any of the graph operations is invoked. An exception is thrown if the graph is found to be cyclic.

The CPO interface defines the common operations on CPOs. The mathematical definition of these operations can be found in [19]. Informal definitions are given in the class documentation. This interface is implemented by the class DirectedAcyclicGraph.

Since most of the CPO operations involve the comparison of two elements, and comparison can be done in constant time once the transitive closure is available, DirectedAcyclicGraph makes heavy use of the transitive closure. Also, since most of the operations on a CPO have a dual operation, such as least upper bound and greatest lower bound, least element and greatest element, etc., the code for the dual operations can be shared if the order relation on the CPO is reversed. This is done by transposing the transitive closure matrix.

8.2.4 Inequality Terms, Inequalities, and the Inequality Solver

The InequalityTerm interface and Inequality and InequalitySolver classes supports the construction of a set of inequality constraints over a CPO and the identification of a member of the CPO that satisfies the constraints. A constraint is an inequality defined over a CPO, which can involve constants, variables, and functions. As an example, the following is a set of constraints over the 4-point CPO in figure 8.3:

$\alpha \le w$

$\beta \le x \wedge \alpha$

$\alpha \le \beta$

where α and β are variables, and $\wedge$ denotes greatest lower bound. One solution to this set of constraints is $\alpha = \beta = x$.

An inequality term is either a constant, a variable, or a function over a CPO. The InequalityTerm

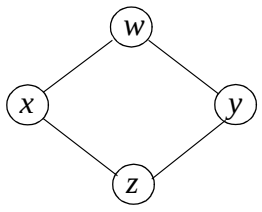


FIGURE 8.3. A 4-point CPO that also happens to be a lattice.

interface defines the operations on a term. If a term consists of a single variable, the value of the variable can be set to a specific element of the underlying CPO. The isSettable() method queries whether the value of a term can be set. It returns *true* if the term is a variable, and *false* if it is a constant or a function. The setValue() method is used to set the value for variable terms. The getValue() method returns the current value of the term, which is a constant if the term consists of a single constant, the current value of a variable if the term consists of a single variable, or the evaluation of a function based on the current value of the variables if the term is a function. The getVariables() method returns all the variables contained in a term. This method is used by the inequality solver.

The Inequality class contains two InequalityTerms, a lesser term and the greater term. The isSatisfied() method tests whether the inequality is satisfied over the specified CPO based on the current value of the variables. It returns *true* if the inequality is satisfied, and *false* otherwise.

The InequalitySolver class implements an algorithm to determine satisfiability of a set of inequality constraints and to find the solution to the constraints if they are satisfiable. This algorithm is described in [72]. It is basically an iterative procedure to update the value of variables until all the constraints are satisfied, or until conflicts among the constraints are found. Some limitations on the type of constraints apply for the algorithm to work. The method addInequality() adds an inequality to the set of constraints. Two methods solveLeast() and solveGreatest() can be used to solve the constraints. The former tries to find the least solution, while the latter attempts to find the greatest solution. If a solution is found, these methods return *true* and the current value of the variables is the solution. The method unsatisfiedInequalities() returns an enumeration of the inequalities that are not satisfied based on the current value of the variables. It can be used after solveLeast() or solveGreatest() return *false* to find out which inequalities cannot be satisfied after the algorithm runs. The bottomVariables() and topVariables() methods return enumerations of the variables whose current values are the bottom or the top element of the CPO.

8.3 Example Use

8.3.1 Generating A Schedule for A Composite Actor

The following is an example of using topological sort to generate a firing schedule for a CompositeActor of the actor package. The connectivity information among the Actors within the composite is translated into a directed acyclic graph, with each node of the graph represented by an Actor. The schedule is stored in an array, where each element of the array is a reference to an Actor.

```
Object[] generateSchedule(CompositeActor composite) {
   DirectedAcyclicGraph g = new DirectedAcyclicGraph();
   // add all the actors contained in the composite to the graph.
   Enumeration allactors = composite.deepGetEntities();
   while (allactors.hasMoreElements()) {
      Actor actor = (Actor)allactors.nextElement();
      g.add(actor);
   }

   // add all the connection in the composite as graph edges.
   allactors = composite.deepGetEntities();
   while (allactors.hasMoreElements()) {
      Actor loweractor = (Actor)allactors.nextElement();

      // find all the actors "higher" than the current one.
      Enumeration alloutports = loweractor.outputPorts();
      while (alloutports.hasMoreElements()) {
```

```
            IOPort outport = (IOPort)alloutports.nextElement();
            Enumeration allinports = outport.deepConnectedInPorts();
            while (allinports.hasMoreElements()) {
               IOPort inport = (IOPort)allinports.nextElement();
               Actor higheractor = (Actor)inport.getContainer();
               if (g.contains(higheractor)) {
                  g.addEdge(loweractor, higheractor);
               }
            }
         }
      }
      return g.topologicalSort();
   }
```

8.3.2 Forming and Solving Constraints over a CPO

The code below uses two classes implementing the InequalityTerm interface. They model constant and variable terms, respectively. The values of these terms are Strings. Inequalities can be formed using these two classes.

```
// A constant InequalityTerm with a String Value.
class Constant implements InequalityTerm {

   // construct a constant term with the specified String value.
   public Constant(String value) {
      _value = value;
   }

   // Return the constant String value of this term.
   public Object getValue() {
      return _value;
   }

   // Constant terms do not contain any variable, so return an array of size zero.
   public InequalityTerm[] getVariables() {
      return new InequalityTerm[0];
   }

   // Constant terms are not settable.
   public boolean isSettable() {
      return false;
   }

   // Throw an Exception on an attempt to change this constant.
   public void setValue(Object e) throws IllegalActionException {
      throw new IllegalActionException("Constant.setValue: This term is a constant.");
   }

   // the String value of this term.
   private String _value = null;
}

// A variable InequalityTerm with a String value.
class Variable implements InequalityTerm {

   // Construct a variable InequalityTerm with a null initial value.
      public Variable() {
   }

   // Return the String value of this term.
   public Object getValue() {
      return _value;
   }

   // Return an array containing this variable term.
```

```
   public InequalityTerm[] getVariables() {
      InequalityTerm[] variable = new InequalityTerm[1];
      variable[0] = this;
      return variable;
   }

   // Variable terms are settable.
   public boolean isSettable() {
      return true;
   }

   // Set the value of this variable to the specified String.
   // Not checking the type of the specified Object before casting for simplicity.
   public void setValue(Object e) throws IllegalActionException {
      _value = (String)e;
   }

   private String _value = null;
}
```

As a simple example, the following Java class constructs the 4-point CPO of figure 8.3, forms a set of constraints with three inequalities, and solves for both the least and greatest solutions. The inequalities are $a \leq w$; $b \leq a$; $b \leq z$, where w and z are constants in figure 2.3, and a and b are variables.

```
// An example of forming and solving inequality constraints.
public class TestSolver {
   public static void main(String[] arv) {
      // construct the 4-point CPO in figure 2.3.
      CPO cpo = constructCPO();

      // create inequality terms for constants w, z and
      // variables a, b.
      InequalityTerm tw = new Constant("w");
      InequalityTerm tz = new Constant("z");
      InequalityTerm ta = new Variable();
      InequalityTerm tb = new Variable();

      // form inequalities: a<=w; b<=a; b<=z.
      Inequality iaw = new Inequality(ta, tw);
      Inequality iba = new Inequality(tb, ta);
      Inequality ibz = new Inequality(tb, tz);

      // create the solver and add the inequalities.
      InequalitySolver solver = new InequalitySolver(cpo);
      solver.addInequality(iaw);
      solver.addInequality(iba);
      solver.addInequality(ibz);

      // solve for the least solution
      boolean satisfied = solver.solveLeast();

      // The output should be:
      // satisfied=true, least solution: a=z b=z
      System.out.println("satisfied=" + satisfied + ", least solution:"
          + " a=" + ta.getValue() + " b=" + tb.getValue());

      // solve for the greatest solution
      satisfied = solver.solveGreatest();

      // The output should be:
      // satisfied=true, greatest solution: a=w b=z
      System.out.println("satisfied=" + satisfied + ", greatest solution:"
          + " a=" + ta.getValue() + " b=" + tb.getValue());
   }

   public static CPO constructCPO() {
      DirectedAcyclicGraph cpo = new DirectedAcyclicGraph();
```

```
        cpo.add("w");
       cpo.add("x");
       cpo.add("y");
       cpo.add("z");

       cpo.addEdge("x", "w");
       cpo.addEdge("y", "w");
       cpo.addEdge("z", "x");
       cpo.addEdge("z", "y");

       return cpo;
    }
}
```

9

Type System

Authors: Yuhong Xiong
Edward A. Lee

9.1 Introduction

The computation infrastructure provided by the actor classes is not statically typed, i.e., the IOPorts on actors do not specify the type of tokens that can pass through them. This can be changed by giving each IOPort a type. One of the reasons for static typing is to increase the level of safety, which means reducing the number of untrapped errors [16].

In a computation environment, two kinds of execution errors can occur, trapped errors and untrapped errors. Trapped errors cause the computation to stop immediately, but untrapped errors may go unnoticed (for a while) and later cause arbitrary behavior. Examples of untrapped errors in a general purpose language are jumping to the wrong address, or accessing data past the end of an array. In Ptolemy II, the underlying language Java is quite safe, so errors rarely, if ever, cause arbitrary behavior.[1] However, errors can certainly go unnoticed for an arbitrary amount of time. As an example, figure 9.1 shows an imaginary application where a signal from a source is downsampled, then fed to a fast Fourier transform (FFT) actor, and the transform result is displayed by an actor. Suppose the FFT actor can accept ComplexToken at its input, and the behavior of the Downsampler is to just pass every second token through regardless of its type. If the Source actor sends instances of ComplexToken, every-

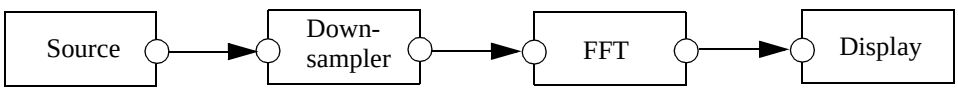


FIGURE 9.1. An imaginary Ptolemy II application

1. Synchronization errors in multi-thread applications are not considered here.

thing works fine. But if, due to an error, the Source actor sends out a StringToken, then the StringToken will pass through the sampler unnoticed. In a more complex system, the time lag between when a token of the wrong type is sent by an actor and the detection of the wrong type may be arbitrarily long.

In languages without static typing, such as Lisp and the scripting language Tcl, safety is achieved by extensive run-time checking. In Ptolemy II, if we imitated this approach, we would have to require actors to check the type of the received tokens before using them. For example, the FFT actor would have to verify that the every received token is an instance of ComplexToken, or convert it to ComplexToken if possible. This approach gives the burden of type checking to the actor developers, distracting them from their development effort. It also relies on a policy that cannot be enforced by the system. Furthermore, since type checking is postponed to the last possible moment, the system does not have fail-stop behavior, so a system may generate an error only after running for an extended period of time, as figure 9.1 shows. To make things worse, an actor may receive tokens from multiple sources. If a token with the wrong type is received, it might be hard to identify from which source the token comes. All these make debugging difficult.

To address this and other issues discussed later, we added static typing to Ptolemy II. This approach is consistent with Ptolemy 0.x. In general-purpose statically-typed languages, such as C++ and Java, static type checking done by the compiler can find a large fraction of program errors. In Ptolemy II, execution of a model does not involve compilation. Nonetheless, static type checking can correspondingly detect problems before any actors fire. In figure 9.1, if the Source actor declares that its output port type is *String*, meaning that it will send out StringTokens upon firing, the static type checker will identify this type conflict in the topology.

In Ptolemy II, because models are not compiled, static typing alone is not enough to ensure type safety at run-time. For example, even if the above Source actor declares its output type to be *Complex*, nothing prevents it from sending out a StringToken at run-time. So run-time type checking is still necessary. With the help of static typing, run-time type checking can be done when a token is sent out from a port. I.e., the run-time type checker checks the token type against the type of the output port. This way, a type error is detected at the earliest possible time, and run-time type checking (as well as static type checking) can be performed by the system instead of by the actors.

One design principle of Ptolemy II is that data type conversions that lose information are not implicitly performed by the system. In the data package, a lossless data type conversion hierarchy, called the type lattice, is defined (see figure 7.2). In that hierarchy, the conversion from a lower type to a higher type is lossless, and is supported by the token classes. This lossless conversion principle also applies to data transfer. This means that across every connection from an output port to an input, the type of the output must be the same as or lower than the type of the input. This requirement is called the type compatibility rule. For example, an output port with type *Int* can be connected to an input port with type *Double*, but a *Double* to *Int* connection will generate a type error during static type checking. This behavior is different from Ptolemy 0.x, but it should be useful in many applications where the users do not want lossy conversion to take place without their knowledge.

As can be seen from above examples, when a system runs, the type of a token sent out from an output port may not be the same as the type of the input port the token is sent to. If this happens, the token must be converted to the input port type before it is used by the receiving actor. This kind of run-time type conversion is done transparently by the Ptolemy II system (actors are not aware it). So the actors can safely cast the received tokens to the type of the input port. This makes the actor development easier.

Ousterhout [67] argues that static typing discourages reuse.

> *"Typing encourages programmers to create a variety of incompatible interfaces, each interface requires objects of specific type and the compiler prevents any other types of objects from being used with the interface, even if that would be useful".*

In Ptolemy II, typing does apply some restrictions on the interaction of actors. Particularly, actors cannot be interconnected arbitrarily if the type compatibility rule is violated. However, the benefit of typing should far outweigh the inconvenience caused by this restriction. In addition, the automatic run-time type conversion provided by the system permits ports of different types to be connected (under the type compatibility rule), which partly relaxes the restriction caused by static typing. Furthermore, there is one important component in Ptolemy that brings much flexibility to the actor interface, the type-polymorphic actors.

Type-polymorphic actors (called polymorphic actors in the rest of this chapter) are actors that can accept multiple types on their ports. For example, the Downsampler in figure 9.1 does not care about the type of token going through it; it works with any type of token. In general, the types on some or all of the ports of a polymorphic actor are not rigidly defined to specific types when the actor is written, so the actor can interact with other actors having different types, increasing reusability. In Ptolemy 0.x, the ports on polymorphic actors whose types are not specified are said to have ANYTYPE, but Ptolemy II uses the term *undeclared type*, since the type on those ports cannot be arbitrary in general. The acceptable types on polymorphic actors are described by a set of type constraints. The static type checker checks the applicability of a polymorphic actor in a topology by finding specific types for them that satisfy the type constraints. This process is called the type resolution, and the specific types are called the resolved types.

Static typing and type resolution have other benefits in addition to the ones mentioned above. Static typing helps to clarify the interface of actors and makes them more manageable. Just as typing may improve run-time efficiency in a general-purpose language by allowing the compiler to generate specialized code, when a Ptolemy system is synthesized to hardware, type information can be used for efficient synthesis. For example, if the type checker asserts that a certain polymorphic actor will only receive IntTokens, then only hardware dealing with integers needs to be synthesized.

To summarize, Ptolemy II takes an approach of static typing coupled with run-time type checking. Lossless data type conversions during data transfer are automatically implemented. Polymorphic actors are supported through type resolution.

9.2 Formulation

9.2.1 Type Constraints

In a Ptolemy II topology, the type compatibility rule imposes a type constraint across every connection from an output port to an input port. It requires that the type of the output port, *outType*, be the same as the type of the input port, *inType*, or less than *inType* under the type lattice in figure 7.2. I.e.,

$$outType \leq inType \tag{2}$$

This guarantees that information is not lost during data transfer. If both the *outType* and *inType* are declared, the static type checker simply checks whether this inequality is satisfied, and reports a type conflict if it is not.

In addition to the above constraint imposed by the topology, actors may also impose constraints. This happens when one or both of the *outType* and *inType* is undeclared, in which case the actor con-

taining the undeclared port needs to describe the acceptable types through type constraints. All the type constraints in Ptolemy II are described in the form of inequalities like the one in (2). If a port has a declared type, its type appears as a constant in the inequalities. On the other hand, if a port has an undeclared type, its type is represented by a variable, called the type variable, in the inequalities. The domain of the type variable is the elements of the type lattice. The type resolution algorithm resolves the undeclared types subject to the constraints. If resolution is not possible, a type conflict error will be reported. As an example of the inequality constraints, consider figure 9.2.

The port on actors A1 has declared type *int*; the ports on A3 and A4 have declared type *double*; and the ports on A2 have their types undeclared. Let the type variables for the undeclared types be α, β, and γ, the type constraints from the topology are:

$$int \leq \alpha$$
$$double \leq \beta$$
$$\gamma \leq double$$

Now, assume A2 is a polymorphic adder, capable of doing addition for integer, double, and complex numbers, and the requirement is that it does not lose precision during the operation. Then the type constraints for the adder can be written as:

$$\alpha \leq \gamma$$
$$\beta \leq \gamma$$
$$\gamma \leq Complex$$

The first two inequalities constrain the output precision to be no less than input, the last one requires that the data on the adder ports can be converted to *Complex* losslessly.

These six inequalities form the complete set of constraints and are used by the type resolution algorithm to solve for α, β, and γ.

This inequality formulation is inspired by the type inference algorithm in ML [59]. There, equalities are used to represent type constraints. In Ptolemy II, the lossless type conversion hierarchy naturally implies inequality relation among the types. In ML, the type constraints are generated from program constructs. In a heterogeneous graphical programming environment like Ptolemy II, the system does not have enough information about the function of the actors, so the actors must present their type information by either declaring the type on their port, or specify a set of type constraints to describe the acceptable types on the undeclared ports. The Ptolemy II system also generates type constraints based on (1).

This formulation converts type resolution into a problem of solving a set of inequalities. An efficient algorithm is available to solve constraints in finite lattices [72], which is described in the appen-

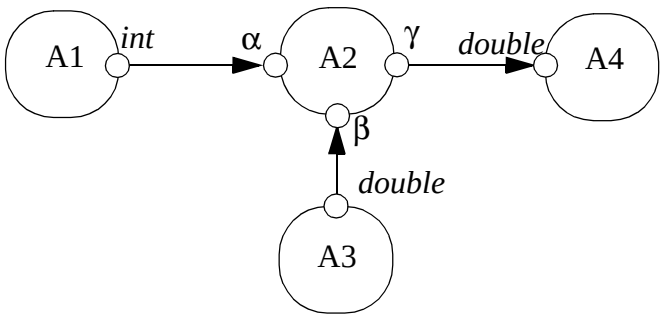


FIGURE 9.2. A topology with types.

dix through an example. This algorithm finds the set of most specific types for the undeclared types in the topology that satisfy the constraints, if they exist.

As mentioned earlier, the static type checker flags a type conflict error if the type compatibility rule is violated on a certain connection. There are other kind of type conflicts indicated by one of the following:

- The set of type constraints are not satisfiable.
- Some type variables are resolved to *NaT.*
- Some type variables are resolved to an abstract type, such as *Numerical* in the type hierarchy.

The first case can happen, for example, if the port on actor A1 in figure 9.2 has declared type *Complex*. The second case can happen if an actor does not specify any type constraints on an undeclared output port. This is due to the nature of the type resolution algorithm where it assigns all the undeclared types to *NaT* at the beginning. If the type constraints do not restrict a type variable to be greater than *NaT*, it will stay at *NaT* after resolution. The third case is considered a conflict since an abstract type does not correspond to an instantiable token class.

To avoid the second case above, any output port must either have a declared type, or some constraints to force its type to be greater than *NaT.* This requirement should be easily satisfied on most actors. A situation that needs some attention is the source actor. A source actor cannot leave its output port type unconstrained. One way to cope with this is to declare the type at a time after the type information is known, but prior to type resolution. For example, if the output data is determined by a parameter set by the user, the parameter can be evaluated during the initialization phase of the execution and the port type can be declared at the end of the initialization, which precedes type resolution.

9.2.2 Run-time Type Checking and Lossless Type Conversion

The declared type is a contract between an actor and the Ptolemy II system. If an actor declares an output port to have a certain type, it asserts that it will only send out tokens whose types are less than or equal to that type. If an actor declares an input port to have a certain type, it requires the system to only send tokens that are instances of the class of that type to that input port. Run-time type checking is the component in the system that enforces this contract. When a token is sent out from an output port, the run-time type checker finds its type using the run-time type identification (RTTI) capability of the underlying language (Java), and compares the type with the declared type of the output port. If the type of the token is not less than or equal to the declared type, a run-time type error will be generated.

As discussed before, type conversion is needed when a token sent to an input port has a type less than the type of the input port but is not an instance of the class of that type. Since this kind of lossless conversion is done automatically, an actor can safely cast a received token to the declared type. On the other hand, when an actor sends out tokens, the tokens being sent do not have to have the exact declared output port type. Any type that is less than the declared type is acceptable. For example, if an output port has declared type *double*, the actor can send *IntToken* from that port. As can be seen, the automatic type conversion simplifies the input/output handling of the actors.

Note that even with the convenience provided by the type conversion, actors should still declare the input types to be the most general that they can handle and the output types to be the most specific type that includes all tokens they will send. This maximizes their applications. In the previous example, if the actor only sends out *IntToken*, it should declare the output type to be *int* to allow the port to be connected with an input with type *int*.

If an actor has ports with undeclared types, its type constraints can be viewed as both a requirement and an assertion from the actor. The actor requires the resolved types to satisfy the constraints.

Once the resolved types are found, they serve the role of declared types at run time. I.e., the type checking and type conversion system guarantees to only put tokens that are instances of the class of the resolved type to input ports, and the actor asserts to only send tokens whose types are less than or equal to the resolved type from output ports.

9.3 Implementation Classes

9.3.1 Static Type Checking and Type Resolution

Type checking and type resolution are done in the actor package. The Actor interface, the AtomicActor, CompositeActor, IOPort and IORelation classes are extended with TypedActor, TypedAtomicActor, TypedCompositeActor, TypedIOPort and TypedIORelation, respectively, as shown in figure 9.3. The container for TypedIOPort must be a ComponentEntity implementing the TypedActor interface, namely, TypedAtomicActor and TypedCompositeActor. The container for TypedAtomicActor and TypedCompositeActor must be a TypedCompositeActor. TypedIORelation constraints that TypedIOPort can only be connected with TypedIOPort. TypedIOPort has a declared type and a resolved type, plus the methods to set and query them. Undeclared type is represented by a null declared type. If a port has a non-null declared type, the resolved type will be the same as the declared type. Calling setDeclaredType() with a non-null argument will set both the declared and resolved type.

Static type checking is done in the checkTypes() method of TypedCompositeActor. This method finds all the connection within the composite by first finding the output ports on deep contained entities, and then finding the deeply connected input ports to those output ports. Transparent ports are ignored for type checking. For each connection, if the types on both ends are declared, static type checking is performed using the type compatibility rule. If the composite contains other opaque TypedCompositeActors, this method recursively calls the checkTypes() method of the contained actors to perform type checking down the hierarchy. Hence, if this method is called on the top level TypedCompositeActor, type checking is performed through out the hierarchy.

If a type conflict is detected, i.e., if the declared type at the source end of a connection is greater than or incomparable with the type at the destination end of the connection, the ports at both ends of the connection are recorded and will be returned in an Enumeration at the end of type checking. Note that type checking does not stop after detecting the first type conflict, so the returned Enumeration contains all the ports that have type conflicts. This behavior is similar to a regular compiler, where compilation will generally continue after detecting errors in the source code.

The class Inequality in the graph package is used to represent type constraints. This class contains two objects implementing the InequalityTerm interface, which represent the lesser and greater terms. TypeTerm in the actor package is such a class that implements the InequalityTerm interface. In type resolution, an inequality term can be a type variable that represents the type of a port with undeclared type, a type constant that represent the type of a port with declared type, or a type constant not associated with a port. For example, in the constraint $int \leq \alpha$ in figure 9.2, α is a type variable representing the resolved type of one of the inputs of the adder A2, and *int* is a type constant representing the declared type (and also the resolved type) of the port on actor A1; in the constraint $\gamma \leq Complex$, *Complex* is a type constant not associated with any port. To accommodate these needs, the class TypeTerm provides two constructors, one with a TypedIOPort argument, the other with a Class argument which is a type in the type hierarchy. When an instance of TypeTerm is constructed using the first constructor, the value of the TypeTerm is the resolved type of the associated TypedIOPort, and the term may be either a constant or a variable, depending on whether the type of the port is declared or not. When a

TypeTerm is constructed using the second constructor, it represents a type constant not associated with a port. The class TypedIOPort has a method `getTypeTerm()`, which returns a TypeTerm associated with itself. To form a type constraint between two TypedIOPorts, the code can be written as:

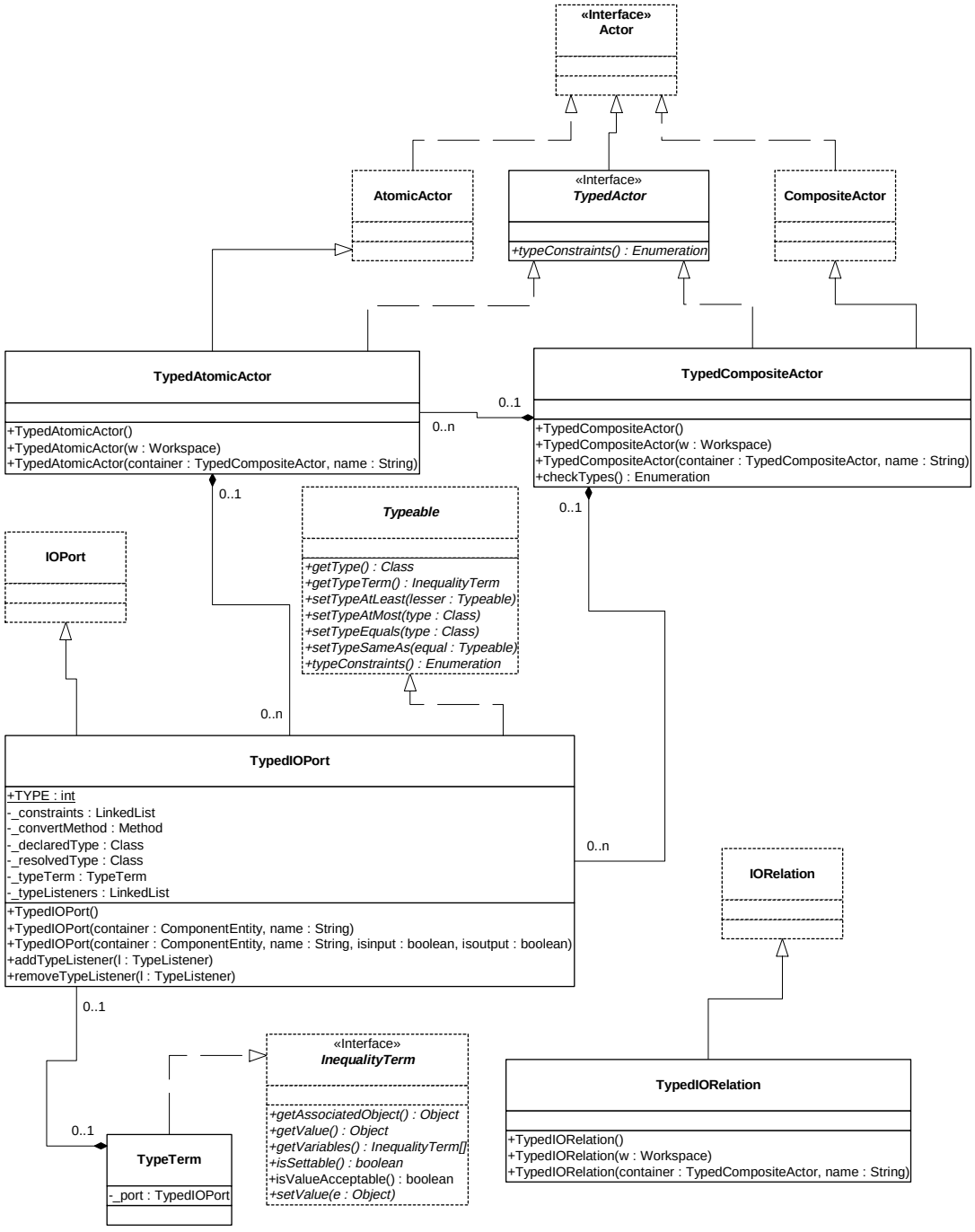


FIGURE 9.3. Classes in the actor package that support type checking.

```
// port1 and port2 are two TypedIOPorts, the constraint is that
// the type of port1 is less than or equal to the type of port2.
Inequality constraint = new Inequality(port1.getTypeTerm(), port2.getTypeTerm());
```

To form a type constraint like $\gamma \leq$ *Complex*, the code can be written as:

```
// port is the TypedIOPort associated with the type variable γ.
TypeTerm complexTerm = new TypeTerm(ComplexToken.class);
Inequality constraint = new Inequality(port.getTypeTerm(), complexTerm);
```

The TypedActor interface has a `typeConstraints()` method, which returns the type constraints of this actor. For atomic actors, the type constraints are different in different actors, but the TypedAtomicActor class provides a default implementation, which is that the type of any input port with undeclared type must be less than or equal to the type of any undeclared output port. Ports with declared types are not included in the default constraints. If all the ports have declared type, no constraints are generated. This default works for most of the control actors such as commutator, multiplexer, and the Downsampler in figure 9.1. It also covers most of the constraints for arithmetic actors such as the adder in figure 9.2. For the adder, the default type constraints covers $\alpha \leq \gamma$ and $\beta \leq \gamma$, the `typeConstraints()` method of the adder only needs to add $\gamma \leq$ *Complex*. This method can be written as:

```
public Enumeration typeConstraints() {
     LinkedList result = new LinkedList();
     result.appendElements(super.typeConstraints());

     TypeTerm complexTerm = new TypeTerm(ComplexToken.class);
     // _output is the output TypedIOPort.
     TypeTerm portTerm = _output.getTypeTerm();
     Inequality constraint = new Inequality(portTerm, complexTerm);

     result.insertLast(ineq);
     return result.elements();
}
```

The typeConstraints() method in TypedCompositeActor collects all the constraints within the composite. It works in a similar fashion as the checkTypes() method, where it recursively goes down the containment hierarchy to collect type constraints of the contained actors. It also scans all the connections and forms type constraints on connections involving undeclared types. As checkTypes(), if this method is called on the top level container, all the type constraints within the composite are returned.

The Manager class has a resolveTypes() method that invokes type checking and resolution. It uses the InequalitySolver class in the graph package to solve the constraints. If type conflicts are detected during type checking or after type resolution, this method throws TypeConflictException. This exception contains an Enumeration of TypedIOPorts where type conflicts occur. The resolveTypes() method is called inside Manager after all the mutations are processed. If TypeConflictException is thrown, it is caught within the Manager and an ExecutionEvent is generated to pass the exception information to the user interface.

9.3.2 Run-time Type Checking and Type Conversion

Run-time type checking is done in the send() method of TypedIOPort. The checking is simply a comparison of the type of the token being sent with the resolved type of the port. If the type of the token is less than or equal to the resolved type, type checking is passed, otherwise, an IllegalActionException is thrown.

The need for type conversion is also determined in the send() method. The type of the destination port is the resolved type of the port containing the receivers that the token is sent to. If the token is not an instance of the class of the destination resolved type, type conversion is needed.

The conversion is done by the convert() method in the token classes. This method is invoked through the Reflection interface of Java. Each TypedIOPort has a method _getConvertMethod() that returns a java.reflect.Method for the convert() method of the resolved type. When type conversion is needed, the send() method of the port sending out the token calls _getConvertMethod() of the destination port to get the convert() method, then invoke it to perform the conversion. Since both the send() and the _getConvertMethod() methods are in TypedIOPort, the _getConvertMethod() is private. For efficiency, the reference to the convert method is cached in TypedIOPort, and _getConvertMethod() will return the cached reference unless it is called for the first time after the resolved type changes.

9.4 Examples

9.4.1 Polymorphic Downsampler

In figure 9.1, if the Downsampler is designed to do downsampling for any kind of token, its type constraint is just $samplerIn \leq samplerOut$, where *samplerIn* and *samplerOut* are the types of the input and output ports, respectively. The default type constraints works in this case. Assuming the Display actor just calls the *stringValue()* method of the received tokens and displays the string value in a certain window, the declare type of its port would be *General*. Let the declared types on the ports of FFT be *Complex*, the The type constraints of this simple application are:

$sourceOut \leq samplerIn$

$samplerIn \leq samplerOut$

$samplerOut \leq Complex$

$Complex \leq General$

Where *sourceOut* represents the declared type of the Source output. The last constraint does not involve a type variable, so it is just checked by the static type checker and not included in type resolution. Depending on the value of *sourceOut*, the ports on the Downsampler would be resolved to different types. Some possibilities are:

- If $sourceOut = Complex$, the resolved types would be $samplerIn = samplerOut = Complex$.
- If $sourceOut = Double$, the resolved types would be $samplerIn = samplerOut = Double$. At runtime, DoubleTokens sent out from the Source will be passed to the DownSampler unchanged. Before they leave the Downsampler and sent to the FFT actor, they are converted to ComplexTokens by the system. The Complextoken output from the FFT actor are instances of Token, which corresponds to the *General* type, so they are transferred to the input of the Display without change.
- If $sourceOut = String$, the set of type constraints do not have a solution, a typeConflictException will be thrown by the static type checker.

9.4.2 Fork Connection

Consider two simple topologies in figure 9.4. where a single output is connected to two inputs in 9.4(a) and two outputs are connected to a single input in 9.4(b). Denote the types of the ports by *a1, a2, a3, b1, b2, b3,* as indicated in the figure. Some possibilities of legal and illegal type assignments are:

- In 9.4(a), if *a1 = Int, a2 = Double, a3 = Complex*. The topology is well typed. At run-time, the IntToken sent out from actor A1 will be converted to DoubleToken before transferred to A2, and converted to ComplexToken before transferred to A3. This shows that multiple ports with different types can be interconnected as long as the type compatibility rule is obeyed.
- In 9.4(b), if *b1 = Int*, b2 = *Double*, and *b3* is undeclared. The the resolved type for *b3* will be *Double*. If *b1 = int* and *b2 = Boolean*, the resolved type for *b3* will be *String* since it is the lowest element in the type hierarchy that is higher than both *Int* and *Boolean*. In this case, if the actor B3 has some type constraints that require *b3* to be less than *String*, then type resolution is not possible, a type conflict will be signaled.

9.4.3 A Sampler System

Figure 9.5 shows a more complete system built in Ptolemy II DE domain. The types are marked by the ports. The underline below some types means that the corresponding port has undeclared type and those types are the resolved type. The functions of the actors are:

1. Clock and Poisson: The Clock actor generates events at regular interval. Its output is a “pure sig-

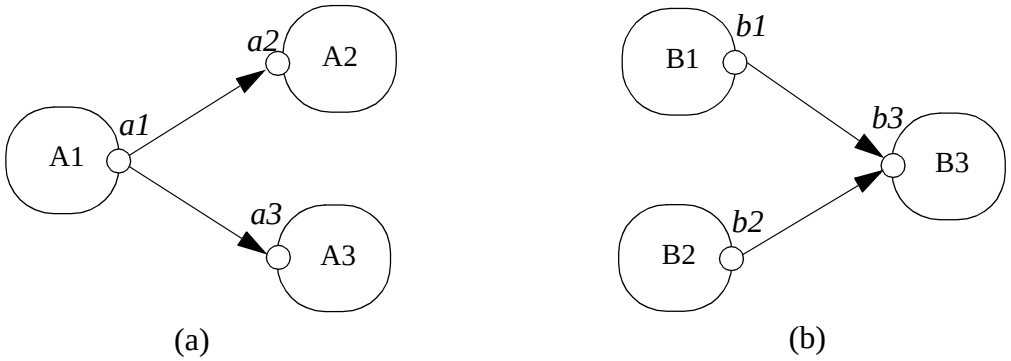


FIGURE 9.4. Two simple topologies with types.

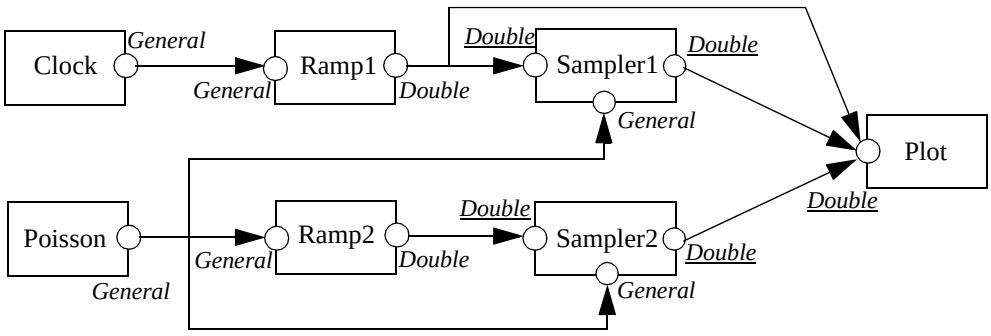


FIGURE 9.5. A Sampler system built in the DE domain.

nal” without value, so the output port type is *General*, which corresponds to the base Token class. The Poisson actor is similar to Clock except that the time spacing between the events follows the Poisson probability distribution.

2. Ramp: This actor sends out events whose value changes by a constant amount every time. It has two Parameters for the initial value and step size. These two Parameters are set by the user and evaluated at the initialization stage. The type of the output is the higher type of the two Parameters. For example, if the initial value Parameter has type *Int* and the step size has type *Double*, the output type is *Double*. Figure 9.5 assumes the output type is *Double*. The input port of the Ramp serves as a trigger. The output event is sent out when a token is received from the input. Since the trigger input does not care the value of the received token, its type is declared as *General*, which means that any type of token can trigger the output.

3. Sampler: This is a polymorphic actor. It passes a token from its input port on the left to the output on the right when a token is received from the bottom input port, so the bottom input is also a trigger input. This actor can do sampling for any type of token, but to ensure that information is not lost, it requires that the type on the left input is less than or equal to the type on the right output. This constraint is covered by the default implementation of the type constraint in TypedAtomicActor, so the Sampler class does not need to override the typeConstraints() method.

4. Plot: This is also a polymorphic actor. It plots the value of the received token in a certain window. Assuming that it requires the input to be a kind of ScalarToken, then the type constraint of this actor is that the input type is less than or equal to *Scalar.*

In this example, all the ports with undeclared type are resolved to *Double*.

Appendix E: The Type Resolution Algorithm

The type resolution algorithm starts by assigning all the type variables the bottom element of the type hierarchy, *NaT*, then repeatedly updating the variables to a greater element until all the constraints are satisfied, or when the algorithm finds that the set of constraints are not satisfiable. The kind of inequality constraints the algorithm can determine satisfiability are the ones with the greater term (the right side of the inequality) being a variable, or a constant. The algorithm allows the left side of the inequality to contain monotonic functions of the type variables, but not the right side. The first step of the algorithm is to divide the inequalities into two categories, *Cvar* and *Ccnst*. The inequalities in *Cvar* have a variable on the right side, and the inequalities in *Ccnst* have a constant on the right side. In the example of figure 9.2, *Cvar* consists of:

$$int \le \alpha$$
$$double \le \beta$$
$$\alpha \le \gamma$$
$$\beta \le \gamma$$

And *Ccnst* consists of:

$$\gamma \le double$$
$$\gamma \le Complex$$

The repeated evaluations are only done on *Cvar*, *Ccnst* are used as checks after the iteration is finished, as we will see later. Before the iteration, all the variables are assigned the value *NaT*, and *Cvar* looks like:

$$int \le \alpha(NaT)$$
$$double \le \beta(NaT)$$
$$\alpha(NaT) \le \gamma(NaT)$$
$$\beta(NaT) \le \gamma(NaT)$$

Where the current value of the variables are inside the parenthesis next to the variable.

At this point, *Cvar* is further divided into two sets: those inequalities that are not currently satisfied, and those that are satisfied:

Not-satisfied	Satisfied
$int \le \alpha(NaT)$	$\alpha(NaT) \le \gamma(NaT)$
$double \le \beta(NaT)$	$\beta(NaT) \le \gamma(NaT)$

Now comes the update step. The algorithm takes out an arbitrary inequality from the Not-satisfied set, and forces it to be satisfied by assigning the variable on the right side the least upper bound of the values of both sides of the inequality. Assuming the algorithm takes out $int \le \alpha(NaT)$, then

$$\alpha = int \vee NaT = int \tag{3}$$

After α is updated, all the inequalities in *Cvar* containing it are inspected and are switched to either the Satisfied or Not-satisfied set, if they are not already in the appropriate set. In this example, after this step, *Cvar* is:

Not-satisfied	Satisfied
$double \le \beta(NaT)$	$int \le \alpha(int)$
$\alpha(int) \le \gamma(NaT)$	$\beta(NaT) \le \gamma(NaT)$

The update step is repeated until all the inequalities in *Cvar* are satisfied. In this example, β and γ

will be updated and the solution is:

$$\alpha = int,\ \beta = \gamma = double$$

Note that there always exists a solution for *Cvar*. An obvious one is to assign all the variables to the top element, *General*, although this solution may not satisfy the constraints in *Ccnst*. The above iteration will find the least solution, or the set of most specific types.

After the iteration, the inequalities in *Ccnst* are checked based on the current value of the variables. If all of them are satisfied, a solution to the set of constraints is found.

This algorithm can be viewed as repeated evaluation of a monotonic function, and the solution is the fixed point of the function. Equation (3) can be viewed as a monotonic function applied to a type variable. The repeated update of all the type variables can be viewed as the evaluation of a monotonic function that is the composition of individual functions like (3). The evaluation reaches a fixed point when a set of type variable assignments satisfying the constraints in *Cvar* is found.

Rehof and Mogensen [72] proved that the above algorithm is linear time in the number of occurrences of symbols in the constraints, and gave an upper bound on the number of basic computations. In our formulation, the symbols are type constants and type variables, and each constraint contains two symbols. So the type resolution algorithm is linear in the number of constraints.

10

Plot Package

Author:Edward A. Lee
Contributor: Christopher Hylands

10.1 Overview

The plot package in Ptolemy II is one of several utility packages that provide support functionality for simulations and applets. It is available in a stand-alone distribution, or as part of the Ptolemy II system. The class diagram is shown in figure 10.1. The key classes are:

- PlotBox: A panel that draws a box with axes along the edges, tick marks along the axes, axis labels, a title, and a legend.
- Plot: An extension of PlotBox that supports a suite of two-dimensional plots of sets of data, including x-y plots, scatter plots, and bar graphs.
- PlotLive: An extension of Plot designed to run continuously in its own thread, continually updating an on-screen plot.
- PlotApplet: An applet that contains a single instance of Plot and that can read the data to be plotted from a URL.
- PlotLiveApplet: An extension of PlotApplet that contains an instance of PlotLive instead of Plot. This is used for applets with animated plots that are continually updated.
- PlotFrame: A window containing a single plot and a menu bar with commands for opening and displaying new data files.
- PlotApplication: An extension of PlotFrame that is an application (a standalone Java program).

10.2 User Interface

The user interface supported by these classes is very rudimentary. Zooming in and out is supported. To zoom in, drag the mouse downwards to draw a box. To zoom out, drag the mouse upward. In addition, several of these classes permit the placement of buttons that exercise certain simple con-

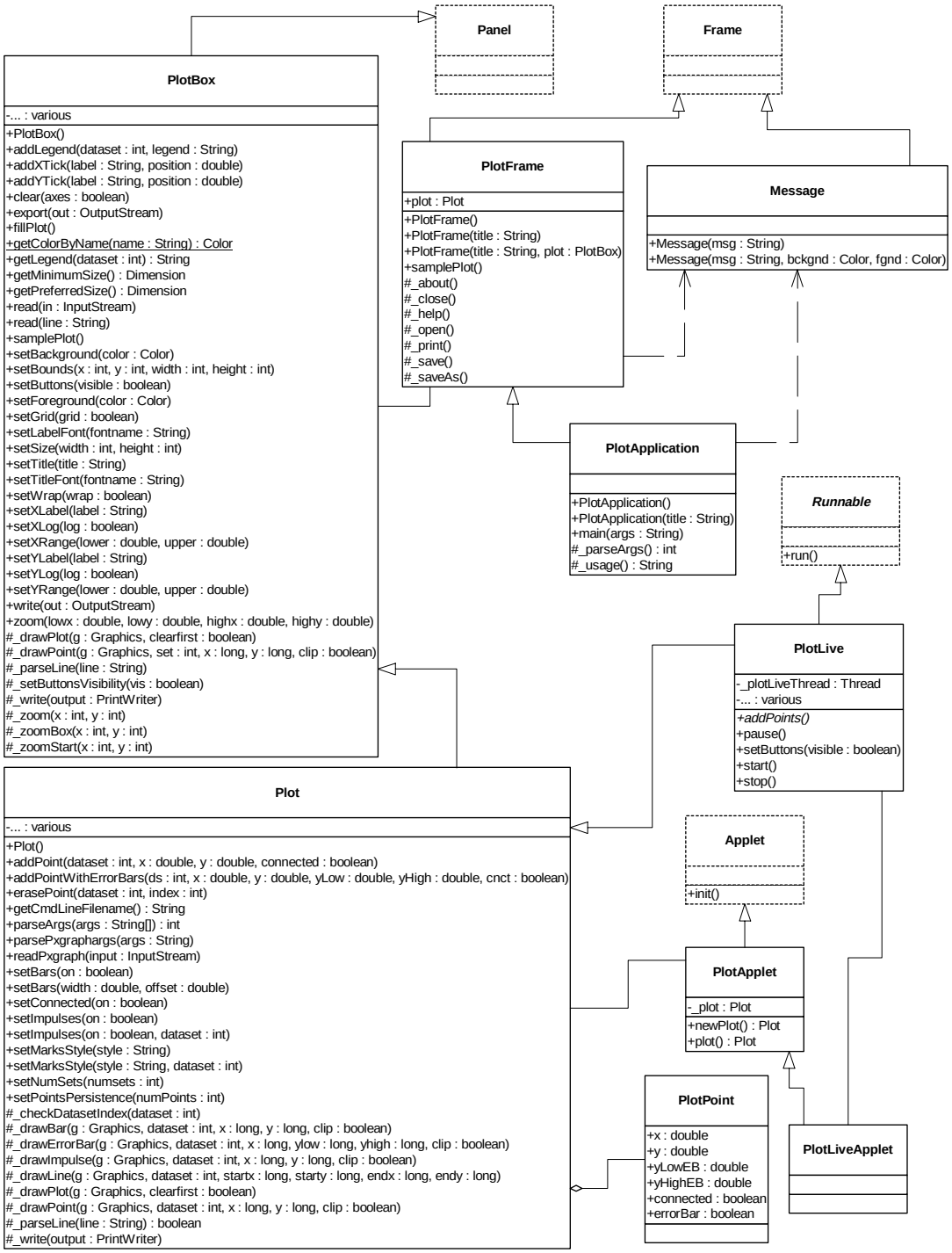


FIGURE 10.1. The classes of the plot package.

trols. The fill button fills the available space with the available data. The start and stop buttons, used by the PlotLive class, start and stop an animated plot.

The PlotFrame class adds a menu that contains a richer set of commands, including opening files, saving the plotted data to a file, printing, etc. Currently, the set of commands is far from complete. In the future, we hope that on-line formatting of the plots and exporting to popular graphics formats will be supported.

10.3 File Format

Instances of the PlotBox and Plot classes can read a simple file format that specifies the data to be plotted. These files can be accessed via URLs. Each file contains a set of commands, one per line, that essentially duplicate the methods of these classes. There are two sets of commands currently, those understood by the base class PlotBox, and those understood by the derived class Plot. Both classes ignore commands that they do not understand. In addition, both classes ignore lines that begin with "#", the comment character. The commands are not case sensitive. In addition, for backward compatibility, these classes can read the binary file format of a prior plotting program called pxgraph.

NOTE: We are likely to change the preferred file format to one based on XML. We hope to maintain backward compatibility with this format, but do not plan to extend this format.

10.3.1 Commands Configuring the Axes

The following commands are understood by the base class PlotBox. These commands can be placed in a file and then read via the read() method, or via a URL using the PlotApplet class. The recognized commands include:

- **TitleText**: *string*
- **XLabel**: *string*
- **YLabel**: *string*

These commands provide a title and labels for the X (horizontal) and Y (vertical) axes. A *string* is simply a sequence of characters, possibly including spaces. There is no need here to surround them with quotation marks, and in fact, if you do, the quotation marks will be included in the labels.

The ranges of the X and Y axes can be optionally given by commands like:

- **XRange**: *min*, *max*
- **YRange**: *min*, *max*

The arguments *min* and *max* are numbers, possibly including a sign and a decimal point. If they are not specified, then the ranges are computed automatically from the data and padded slightly so that datapoints are not plotted on the axes.

The tick marks for the axes are usually computed automatically from the ranges. Every attempt is made to choose reasonable positions for the tick marks regardless of the data ranges (powers of ten multiplied by 1, 2, or 5 are used). However, they can also be specified explicitly using commands like:

- **XTicks**: *label position, label position, ...*
- **YTicks**: *label position, label position, ...*

A *label* is a string that must be surrounded by quotation marks if it contains any spaces. A position is a number giving the location of the tick mark along the axis. For example, a horizontal axis for a frequency domain plot might have tick marks as follows:

```
XTicks: -PI -3.14159, -PI/2 -1.570795, 0 0, PI/2 1.570795, PI 3.14159
```

Tick marks could also denote years, months, days of the week, etc.

The X and Y axes can use a logarithmic scale with the following commands:

- **XLog**: on
- **YLog**: on

The tick labels, if computed automatically, represent powers of 10. Note that if a logarithmic scale is used, then the values must be positive. **Non-positive values will be silently dropped**.

By default, tick marks are connected by a light grey background grid. This grid can be turned off with the following command:

- **Grid**: off

It can be turned back on with

- **Grid**: on

Also, by default, the first ten data sets are shown each in a unique color. The use of color can be turned off with the command:

- **Color**: off

It can be turned back on with

- **Color**: on

Finally, the rather specialized command

- **Wrap**: on

enables wrapping of the X (horizontal) axis, which means that if a point is added with X out of range, its X value will be modified modulo the range so that it lies in range. This command only has an effect if the X range has been set explicitly. It is designed specifically to support oscilloscope-like behavior, where the X value of points is increasing, but the display wraps it around to left. A point that lands on the right edge of the X range is repeated on the left edge to give a better sense of continuity. The feature works best when points do land precisely on the edge, and are plotted from left to right, increasing in X.

All of the above commands can also be invoked directly by calling the corresponding public methods from some Java code.

10.3.2 Commands for Plotting Data

The set of commands understood by the Plot class support specification of data to be plotted and control over how the data is shown.

The style of marks used to denote a data point is defined by one of the following commands:

- **Marks**: none
- **Marks**: points
- **Marks**: dots
- **Marks**: various

Here, “points” are small dots, while “dots” are larger. If “various” is specified, then unique marks are used for the first ten data sets, and then recycled. Using no marks is useful when lines connect the points in a plot, which is done by default. If the above directive appears before any DataSet directive,

then it specifies the default for all data sets. If it appears after a DataSet directive, then it applies only to that data set.

To disable connecting lines, use:

- **Lines**: off

To re-enable them, use

- **Lines**: on

You can also specify “impulses”, which are lines drawn from a plotted point down to the x axis. Plots with impulses are often called “stem plots.” These are off by default, but can be turned on with the command:

- **Impulses**: on

or back off with the command

- **Impulses**: off

If that command appears before any DataSet directive, then the command applies to all data sets. Otherwise, it applies only to the current data set.

To create a bar graph, turn off lines and use any of the following commands:

- **Bars**: on
- **Bars**: *width*
- **Bars**: *width, offset*

The *width* is a real number specifying the width of the bars in the units of the x axis. The *offset* is a real number specifying how much the bar of the i-th data set is offset from the previous one. This allows bars to “peek out” from behind the ones in front. Note that the frontmost data set will be the first one. To turn off bars, use

- **Bars**: off

To specify data to be plotted, start a data set with the following command:

- **DataSet**: *string*

Here, *string* is a label that will appear in the legend. It is not necessary to enclose the string in quotation marks.

To start a new dataset without giving it a name, use:

- **DataSet**:

In this case, no item will appear in the legend.

If the following directive occurs:

- **ReuseDataSets**: on

then datasets with the same name will be merged. This makes it easier to combine multiple data files that contain the same datasets into one file. By default, this capability is turned off, so datasets with the same name are not merged.

The data itself is given by a sequence of commands with one of the following forms:

- *x, y*
- **draw**: *x, y*
- **move**: *x, y*
- *x, y, yLowErrorBar, yHighErrorBar*

- **draw**: *x, y, yLowErrorBar, yHighErrorBar*
- **move**: *x, y, yLowErrorBar, yHighErrorBar*

The “draw” command is optional, so the first two forms are equivalent. The “move” command causes a break in connected points, if lines are being drawn between points. The numbers *x* and *y* are arbitrary numbers as supported by the Double parser in Java (e.g. “1.2”, “6.39e-15”, etc.). If there are four numbers, then the last two numbers are assumed to be the lower and upper values for error bars. The numbers can be separated by commas, spaces or tabs.

The number of data sets to be plotted does not need to be specified.

10.4 Exporting

Currently, the ability to export a plot to other formats is rather limited. A small set of key bindings are provided:

- Cntr-c: Export the plot to the clipboard.
- D: Dump the plot to standard output.
- E: Export the plot to standard output in EPS format.
- F: Fill the plot.
- H or ?: Display a simple help message.

The encapsulated postscript (EPS) that is produced is tuned for black-and-white printers. In the future, more formats may supported. Also at this time (jdk 1.1.4), Java's interface the clipboard does not work, so Cntr-c might not accomplish anything.

Exporting to the clipboard and to standard output, in theory, is allowed for applets, unlike writing to a file. Thus, these key bindings provide a simple mechanism to obtain a high-resolution image of the plot from an applet, suitable for incorporation in a document. However, in some browsers, exporting to standard out triggers a security violation. You can use Sun's appletviewer instead.

10.5 Limitations

The plot package is a starting point, with a number of significant limitations.

- The PlotFrame and PlotApplication classes should be greatly extended to allow on-line changes in the format of the plots.
- A binary file format that includes plot format information is needed.
- If you zoom in far enough, the plot becomes unreliable. In particular, if the total extent of the plot is more than 2^{32} times extent of the visible area, quantization errors can result in displaying points or lines. Note that 2^{32} is over 4 billion.
- The log axis facility has a number of limitations listed in the documentation of the _gridInit() method in the PlotBox class.
- Graphs cannot be currently copied via the clipboard.
- There is no mechanism for customizing the colors used in a plot.

PART 3:

DOMAINS

The chapters in this part describe existing Ptolemy domains. The domains implement models of computation, which are summarized in chapter 1. Most of these models of computation can be viewed as a framework for component-based design, where the framework defines the interaction mechanism between the components. Some of the domains (CSP, DDE, and PN) are thread-oriented, meaning that the components implement Java threads. These can be viewed, therefore, as abstractions upon which to build threaded Java programs. These abstractions are much easier to use (much higher level) than the raw threads and monitors of Java. Others (CT, DE, SDF) of the domains implement their own scheduling between actors, rather than relying on threads. This usual results in much more efficient execution. The non-threaded domains are described first, followed by the threaded domains. Within this grouping, the domains are ordered alphabetically (which is an arbitrary choice).

11

CT Domain

Author: Jie Liu

11.1 Introduction

The continuous-time (CT) domain in Ptolemy II aims to help the design and simulation of systems that can be modeled using ordinary differential equations (ODEs). ODEs are often used to model analog circuits, plant dynamics in control systems, lumped-parameter mechanical systems, lumped-parameter heat flows and many other physical systems.

Using digital computers to simulate continuous-time systems has been studied for more than two decades. One of the most well-known tools is Spice [62]. The CT domain differs from Spice-like continuous-time simulators in two ways — the system specification is somewhat different, and it is designed to interact with other models of computation.

11.1.1 Basic Terminology

We outline some basic terminology that are used in this chapter. This is not an overview of the numerical ODE solving methods, but it will help explain the operation of the domain and the motivation for its design. For a detailed understanding of ODEs and their numerical solutions, please refer to books on numerical solutions for ODEs, e.g. [26].

In general, an ODE-based continuous-time system has the following form:

$$\dot{x} = f(x, u, t) \tag{4}$$

$$y = g(x, u, t) \tag{5}$$

$$x(t_0) = x_0, \tag{6}$$

where, $t \in \Re$, $t \geq t_0$, a real number, is *continuous time*. At any time t, $x \in \Re^n$, an n-tuple of real numbers, is the *state* of the system; $u \in \Re^m$ is the m-dimensional *input* of the system; $y \in \Re^l$ is the l-dimensional *output* of the system; $\dot{x} \in \Re^n$ is the derivative of x with respect to time t, i.e.

$$\dot{x} = \frac{dx}{dt}. \tag{7}$$

Equations (4), (5), and (6)are called the *system dynamics*, the *output map*, and the *initial condition* of the system, respectively. The solution of the set of ODE is a continuous waveform $x(t)$, which is a function of time that satisfies the equation (4) and initial condition (6). The output of the system is then defined as the function of the input that satisfies (5).

Not all ODEs have a solution, and some ODEs have more than one solution. In such situations, we say that the solution is not well defined. This is usually a result of errors in the system modeling. The following are conditions for the existence and uniqueness of solutions of ODEs. We restrict our discussion to systems that have unique solutions.

We denote by D a set in $\Re$ which contains at most a finite number of points per unit interval, and $\Re\backslash D = \Re \cap D$ the difference set, those points in $\Re$ that are not in D. If $f(\ )$ is continuous on $\Re\backslash D$, and $f(\ )$ satisfies the Lipschitz condition, then ODE (4) with the initial condition (6) has a unique solution, which is a continuous function $\psi: \Re \rightarrow \Re^n$ such that,

$$\psi(t_0) = x_0 \tag{8}$$

and

$$\dot{\psi}(t) = f(\psi(t), u(t), t), \qquad \forall t \in \Re\backslash D. \tag{9}$$

The solution is sometimes called the *state trajectory* of the system. By applying the output map on the state trajectory, the output of the system can be obtained.

Usually, only the solution on a finite time interval $[t_0,t_f]$ is needed. A simulation of the system is performed on discrete time points in this interval. Here we denote by

$$Tc = \{t_0, t_1, t_2, \ldots t_n, \ldots t_f\}, Tc \subset [t_0, t_f], \tag{10}$$

where

$$t_0 < t_1 < t_2 < \ldots < t_n < \ldots < t_f, \tag{11}$$

the set of the discrete time points. A “snapshot” of all the signals at t is called the *behavior* of the system at time t. To explicitly illustrate the discretization of time and the difference between the precise solution and the numerical solution, we use the following notation in the rest of the chapter:

- t_n: the n-th time point, to explicitly show the discretization of time. We also write t if the order n is not important.
- $x[t_i, t_j]$: the (continuous) state trajectory from time t_i to t_j;
- $x(t_n)$: the *precise* solution of (4) at time t_n;
- x_{t_n}: the *numerical* solution of (4) at time t_n;
- $h_n = t_n - t_{n-1}$: step size of the discretization of time. We also write h if the position n in the sequence is not important. For accuracy reason, h may not be uniform.

A general way of simulating a continuous-time system numerically is to compute the state and the output of the system in increasing order of t_n. Such algorithms are called the *time-marching* algorithms, and, at this time, we only consider these algorithms. There are variety of numerical algorithms differ on how x_{t_n} is computed given $x_{t_0} \ldots x_{t_{n-1}}$. The choice of algorithm is application dependent, and usually reflects different speed and accuracy trade-offs.

11.1.2 Time

One distinct characterization of the CT model is the continuity of time. This implies that a continuous-time system must have a behavior at any time of interest. The simulation engine of the CT model, although it marches discretely in time, must be able to compute the behavior of the system at any time point. The discretization of time, appearing as integration step sizes, is affected by a time point of interest (e.g. a discontinuity), by the integration error, and by the convergence in solving algebraic equations.

Time is also global, which means that all components in the system share the same notion of time.

11.1.3 Fixed-Point Behavior

Numerical ODE solution algorithms approximate the derivative operator in (4) using the history and the current knowledge on the state trajectory. I.e. at time t_n, the derivative of x is approximated by

$$\dot{x}_{t_n} = p(x_{t_0}, \ldots, x_{t_{n-1}}, x_{t_n}). \tag{12}$$

Plugging in (4), we have

$$p(x_{t_0} \ldots x_{t_{n-1}}, x_{t_n}) = f(x_{t_n}, u(t_n), t_n) \tag{13}$$

Depending on whether x_{t_n} explicitly appears in (13), the algorithms are called explicit integration algorithms or implicit integration algorithms. By plugging the system dynamic (4) into (12), we end up solving a set of algebraic equations in one of the two forms:

$$x_{t_n} = F_E(x_{t_0}, \ldots, x_{t_{n-1}}) \tag{14}$$

or

$$x_{t_n} = F_I(x_{t_0}, \ldots, x_{t_n}), \tag{15}$$

where $F_E(\ )$, and $F_I(\ )$ are derived from the time t_n, the input $u(t_n)$, the function f, and the history of x and $\dot{x}$. Solving (14) or (15) at a particular time point is called an *iteration* of the CT simulation.

Equation (14) can be solved simply by a function evaluation and an assignment. But the solution of (15) is the *fixed point* of $F_I(\ )$, which may not exist, may not be unique, or may not be able to be found. The *contraction mapping theorem* [12] shows the existence and uniqueness of the fixed point solution, and provides one way to find it. Given the map $F_I(\ )$ is a local contraction map (generally true for small enough step sizes), let the initial guess σ_0 be in the contraction radius, and then the unique fixed point can be found by iteratively computing:

$$\sigma_1 = F_E(\sigma_0), \sigma_2 = F_E(\sigma_1), \sigma_3 = F_E(\sigma_2), \ldots \tag{16}$$

Solving both (14) and (15) should be thought of as finding the fixed-point behavior of the system at a particular time. This means both functions $F_E(\ )$ and $F_I(\ )$ should keep unchanged during one iteration of the simulation. This further implies that the topology of the system, all the parameters, and all the internal states that the firing functions depend on should maintain unchanged.

11.1.4 Discontinuity

The existence and uniqueness of the solution of an ODE (Theorem 1 in Appendix F) allows the

function *f* to be discontinuous at a countable number of discrete points, which are called *breakpoints* (also called the *discontinuous points* in some literature). These breakpoints may be caused by the discontinuity of input signal *u*, or by an intrinsic property of *f*. In theory, the solutions at these points are not well defined. But the left and right limit are. So instead of solving the ODE at those points, we would actually try to find the left and right limit.

A breakpoint may be known beforehand, in which case it is called a *predictable breakpoint* (PBP). For example, a square wave source actor can tell its next flip time. This information can be used to control the discretization of time. A breakpoint can also be *unpredictable* (UBP), which mean it is unknown until the time it occurs. For example, an actor that varies its functionality when the input signal crosses a threshold can only report a "missed" breakpoint after an integration step is finished.

One impact of the discontinuities on the ODE solvers is that the history solutions before the discontinuous points are useless in approximating the derivative of *x* after the breakpoints. The solver should resolve the new initial conditions and start the solving process as if it is at the starting point.

11.2 System Specification

There are usually two ways to specify a continuous time system, the conservative law model and the signal-flow model [38]. The conservative law model, also called the nodal analysis in circuit simulation [35], defines a system by its physical components, which specifies relations of *cross* and *through* variables, and the *conservative laws* are used to compile the component relations into global system equations. For example, in circuit simulations, the cross variables are voltages, the through variables are currents, and the conservative laws are the Kirchhoff's laws. This model directly reflects the physical components of a system, thus is easy to construct from a potential implementation. The actual mathematical form of the system is hidden. In the signal-flow model, entities in the system are maps that define the mathematical relation between the input and output signals. Entities communicate by passing signals. This model directly reflects the mathematical relations among signals, and is more convenient for specifying the systems that do not have an explicit implementation yet.

In the CT domain of Ptolemy II, the signal-flow model is chosen as the interaction semantics. The conservative law semantics may be used within an entity to define its I/O relation. There are four major reasons for this decision:

1. *The signal-flow model is more abstract*. Ptolemy is focused on the system-level design and behavior simulation. It is usually the case that at this stage of a design, users are working with simplified mathematical models of a system, and the implementation details are unknown or not cared about.

2. *The signal flow model is more flexible and extensible*, in the sense that it is easy to make topology changes in the problem level. For models like hybrid systems, it is more convenient to manipulate the system at this level.

3. *The signal flow model is consistent with other models of computation in Ptolemy II*. Most models of computation in Ptolemy use message-passing as the interaction semantics. Choosing the signal-flow model for CT makes it consistent with other domains, so the interaction of heterogeneous systems is easy to study and implement.

4. *The signal flow model is compatible with the conservative law model*. For physical systems that are based on conservative laws, it is usually possible to wrap them into an entity in the signal flow model. The inputs of the entity are the excitations, like the voltages on voltage sources, and the outputs are the variables that the rest of the system may be interested in.

The signal flow block diagram of the system in (4) - (6) is shown in figure 11.1. In the figure, u, $\dot{x}$, x, and y, are continuous signals (waveforms) flowing from one block to the next. Since time is shared by all blocks, it is not considered an input. At any fixed time t, if the "snapshot" values $x(t)$ and $u(t)$ are given, $\dot{x}(t)$ and $y(t)$ can be found by evaluating $f(\ )$ and $g(\ )$, which can be achieved by firing the respective blocks.

11.2.1 An Example

Let us consider a second order linear system,

$$\begin{aligned} m\ddot{x}(t) + b\dot{x}(t) + kx(t) &= u(t) \\ y(t) &= c \cdot x(t) \\ x(0) = 0,\ \dot{x}(0) &= 0 \end{aligned} \tag{17}$$

It can be written in the form of (4)-(6), if we define a vector

$$z(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix}. \tag{18}$$

The system equations are, in terms of $z(t)$,

$$\dot{z}(t) = \frac{1}{m}\begin{bmatrix} 0 & 1 \\ -k & -b \end{bmatrix} z(t) + \begin{bmatrix} 0 \\ 1/m \end{bmatrix} u(t) \tag{19}$$

$$y(t) = \begin{bmatrix} c & 0 \end{bmatrix} z(t)$$

$$z(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

The equations could be a model for an analog circuit as shown in figure 11.2(a), where

$$\begin{aligned} m &= R1 \cdot R2 \cdot C1 \cdot C2 \\ k &= R1 \cdot C1 + R2 \cdot C2 \\ b &= 1 \\ c &= \frac{R4}{R3 + R4}. \end{aligned} \tag{20}$$

Or it could be a lumped-parameter model of a spring-mass mechanical model for the system shown in

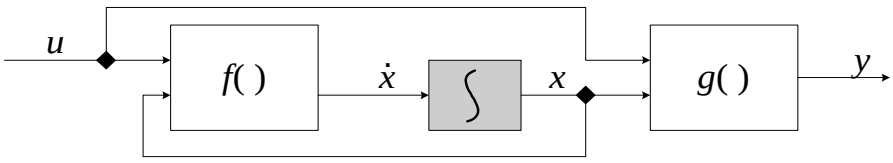
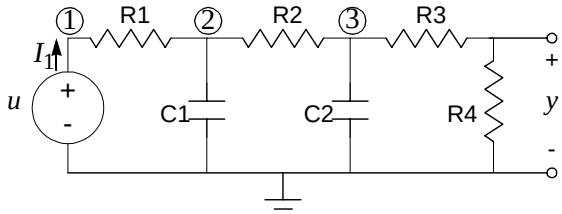


FIGURE 11.1. A conceptual block diagram for continuous time systems.

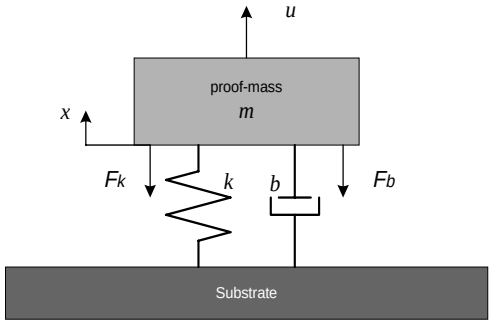
figure 11.2(b), where *m* is the mass, *k* is the spring constant, *b* is the damping parameter, and *c* = 1. At the system level, we would care more about the mathematical form (17) or (19), rather than the implementation detail. The system, in the signal-flow block diagram form, is shown in figure 11.3. To be concrete, the conservative law model (modified nodal analysis) of the implementation figure 11.2(a) is shown in (21).

$$\begin{bmatrix} \frac{1}{R1} & -\frac{1}{R1} & 0 & 0 & -1 \\ -\frac{1}{R1} & \frac{1}{R1}+\frac{1}{R2}+C1\frac{d}{dt} & -\frac{1}{R2} & 0 & 0 \\ 0 & -\frac{1}{R2} & \frac{1}{R2}+\frac{1}{R3}+C2\frac{d}{dt} & -\frac{1}{R3} & 0 \\ 0 & 0 & -\frac{1}{R3} & \frac{1}{R3}+\frac{1}{R4} & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ y \\ I_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ u \end{bmatrix} \tag{21}$$

By doing some math, we can see that (21) and (19) are in fact equivalent. Equation (21) can be easily assembled from the circuit, but it is more complicated than (19). Notice that in (21) d/dt is the deriva-



(a) A circuit implementation.



(b) A mechanical implementation.

FIGURE 11.2. Possible implementations of the system equations.

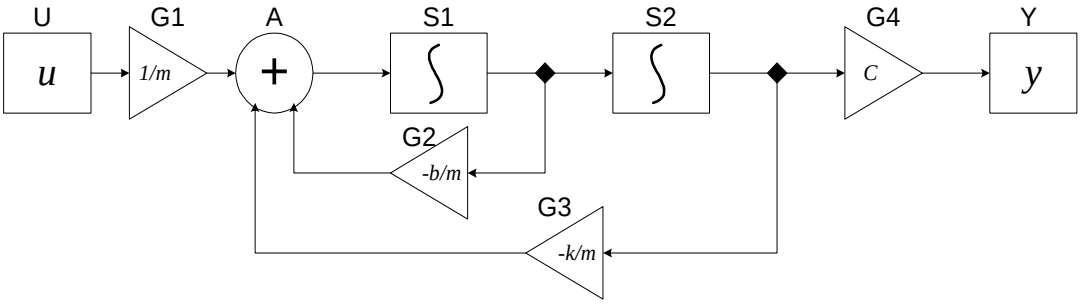


FIGURE 11.3. The block diagram for the example system.

tive operator, which is replace by an integration algorithm at each time step, and the system equations reduce to a set of algebraic equations. Spice software is known to have a very good simulation engine for the systems in form of (21).

11.3 CT Actors

A CT system can be built up using actors in the ptolemy.domains.ct.lib and domain polymorphic actors that have continuous behaviors (only those implementing the SequenceActor interface need to be excluded). The central actor in CT is the integrator. It serves the unique position of wiring up ODEs. Other actors in a CT system are usually stateless. A general understanding is that all the information — the state — of a CT system is stored in the integrators.

In order to combine CT with other domains (which are usually discrete), some actors are designed to convert continuous signals to discrete signals (events), and vise versa. These actors are called *event generators* and *event interpreters*. Although these actors does not strictly belong to any domain, they are in the ptolemy.domains.ct.lib package. When building systems, the CT part can always provide a discrete interface to other domains.

11.3.1 Integrator and ODE Solvers

An integrator is a single-input single-output actor, where the input is the derivative of the output (with respect to time). Since integration is not an operation that can be computed directly using digital computers, we rely on numerical integration methods to approximate it. There are a large variety of numerical integration method with different speed and accuracy trade-offs.

During the execution of a continuous-time model, the functionality of the integrators is provided by the ODE solver in charge. To ease switching ODE solvers during execution, (which is essential for handling breakpoints), the integration is delegated to the ODE solvers.

There are five ODE solvers implemented in the ptolemy.domains.ct.kernel.solver package. These solvers are ForwardEulerSolver, BackwardEulerSolver, ExplicitRK23Solver, TrapezoidalRuleSolver, and ImpulseBESolver.

In general, simulating a continuous-time system (4)-(6) by a time-marching ODE solver involves the following execution steps:

1. Given the state of the system $x_{t_0} \ldots x_{t_{n-1}}$ at time points $t_0 \ldots t_{n-1}$, if the current integration step size is h, i.e. $t_n = t_{n-1} + h$, compute the new state x_{t_n} using the numerical integration algorithms. During the application of an integration algorithm, each evaluation of the $f(a, b, t)$ function is achieved by the following sequence:

- Integrators emit tokens corresponding to a;
- Source actors emit tokens corresponding to b;
- The current time is set to t;
- The tokens are passed through the topology (in a data-driven way) until they reach the integrators again. The returned tokens are $\dot{a} = f(a, b, t)$.

2. After the new state x_{t_n} is computed, test whether this step is successful. Local truncation error and unpredictable breakpoints are the issues to be concerned with, those that could lead to an unsuccessful step.

3. If the step is successful, predict the next step size. Otherwise, reduce the step size and try again.

Due to the signal-flow representation of the system, the numerical ODE solving algorithms are implemented as actor firings and token passings under proper scheduling.

The ODE solvers implemented in the CT domain include:

(1) ForwardEulerSolver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + h_{n+1} \cdot \dot{x}_{t_n} \\ &= x_{t_n} + h_{n+1} \cdot f(x_{t_n}, u_{t_n}, t_n) \end{aligned} \tag{22}$$

(2) BackwardEulerSolver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + h_{n+1} \cdot \dot{x}_{t_{n+1}} \\ &= x_{t_n} + h_{n+1} \cdot f(x_{t_{n+1}}, u_{t_{n+1}}, t_{n+1}) \end{aligned} \tag{23}$$

(3) ExplicitRK23Solver: (2(3)-order Explicit Runge-Kutta method)

$$\begin{aligned} K_0 &= h_{n+1} \cdot f(x_{t_n}, u_{t_n}, t_n) \\ K_1 &= h_{n+1} \cdot f(x_{t_n} + K_0/2, u_{t_n + h_{n+1}/2}, t_n + h_{n+1}/2) \\ K_2 &= h_{n+1} \cdot f(x_{t_n} + 3K_1/4, u_{t_n + 3h_{n+1}/4}, t_n + 3h_{n+1}/4) \\ \tilde{x}_{t_{n+1}} &= x_{t_n} + \frac{2}{9}K_0 + \frac{1}{3}K_1 + \frac{4}{9}K_2 \end{aligned} \tag{24}$$

with error control:

$$\begin{aligned} K_3 &= h_{n+1} \cdot f(\tilde{x}_{t_{n+1}}, u_{t_{n+1}}, t_{n+1}) \\ LTE &= -\frac{5}{72}K_0 + \frac{1}{12}K_1 + \frac{1}{9}K_2 - \frac{1}{8}K_3 \end{aligned} \tag{25}$$

if $|LTE| < ErrorTolerance$, $x_{t_{n+1}} = x_{t_{n+1}}$, otherwise, fail. If this step is successful, the next integration step size is predicted by:

$$h_{n+2} = h_{n+1} \cdot max(0.5, 0.8 \cdot \sqrt[3]{(ErrorTolerance)/|LTE|}) \tag{26}$$

(4) TrapezoidalRuleSolver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + \frac{h_{n+1}}{2}(\dot{x}_{t_n} + \dot{x}_{t_{n+1}}) \\ &= x_{t_n} + \frac{h_{n+1}}{2}(\dot{x}_{t_n} + f(x_{t_{n+1}}, u_{t_{n+1}}, t_{n+1})) \end{aligned} \tag{27}$$

(5) ImpulseBESolver:

$$\begin{aligned} \tilde{x}_{n+1} &= x_n + h_{n+1} \cdot \dot{\tilde{x}}_{n+1} \\ x_{n+1} &= \tilde{x}_{n+1} - h_{n+1} \cdot \dot{x}_{n+1} \end{aligned} \tag{28}$$

Notice that although the two time points are labeled *n* and *n*+1, they have the same (absolute) time value.

Among these solvers, (1) and (2) are fixed-step-size solvers, which do not predict step sizes according to the error tolerance; (3) and (4) are variable-step-size solvers; and (5) is a solver that specifically deals with impulses.

11.3.2 Actor Library

1. **Integrator**: The integrator for continuous-time simulation. An integrator has one input port and one output port. Conceptually, the input is the differential of the output. So an ordinary differential equation

$$\dot{x} = f(x, t) \tag{29}$$

can be represented as in figure 11.4

An integrator is a *dynamic actor* that emits a token (with value equal to its internal state) at the beginning of the simulation. An integrator is a *step size control actor*, which can control the accuracy of the ODE solution by adjusting step sizes. An integrator has memory, its state.

To help resolving the new state, a set of variables are used:

- *state*: This is the new state at a time point, which has been confirmed by all the step size control actors.
- *tentative state*: This is the resolved state which has not been confirmed. It is a starting point for other actor to control the successfulness of this integration step.
- *history*: The previous states, which may be used by some integration methods.

For different ODE solving methods, the functionality of a integrator could be different. This class provide a basic implementation of the integrator, so some solver-dependent methods are delegated to the current ODE solver.

An integrator has one parameter: InitialState. At the initialization stage of the simulation, the state of the integrator is set to the initial state. The InitialState will not impact the simulation after the simulation starts. The default value of the parameter is 0. An integrator can possibly have several auxiliary variables — _auxVariables. The _auxVariables are for the convenience of the ODE solver.

The integrator remembers the history states and their derivatives for the past several steps. The history is used for multistep methods.

2. **CTPeriodicalSampler**. This actor periodically samples the input signal and generates events with the value of the input signal. The sampling rate is given by parameter SamplePeriod, which has default value 0.1.

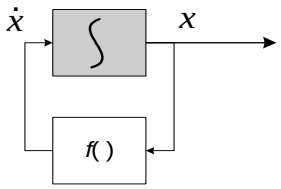


FIGURE 11.4. An integrator with feedback.

3. **CTZeroCrossingDetector**. This is an event detector that monitors the signal coming in from the trigger input. If the trigger is zero, then output the token from the input port. This actor controls the integration step size to accurately resolve the time that the zero crossing happens. It has a parameter ErrorTolerance, which controls how accurate the zero crossing is.

4. **CTButtonEvent**. This is a continuous time actor that responds to button clicks on screen. It is not an event generator. It outputs a continuous signal with value true or false. At the beginning of the execution it outputs false. If the parameter Button is changed, it will output true in the next iteration. I.e. there is, at most, one step size delay in responding to the button event. The true value will be kept for one iteration, after that the output goes back to false.

5. **CTZeroOrderHold**. An actor that converts events into continuous signals. This actor acts as the zero-order hold. It consumes the token when the consumeCurrentEvent() is called. This value will be held and emitted every time it is fired, until the next consumeCurrentEvent() is called. This actor has one single input port of type DoubleToken, one single output port of type DoubleToken, and no parameters.

6. **CTThresholdMonitor**. This actor monitors the integration steps so that the given threshold (on the input) is not crossed in one step. This actor has one input port, but no output ports. Its functionality is solely devoted to controlling the integration step size. It has two parameters ThresholdWidth and ThresholdCenter, which have default value 1e-2 and 0, respectively.

11.3.3 Domain Polymorphic Actors

- **Stateless actors**: All stateless actors can be used in CT. In fact, most CT systems can be built by integrators and stateless actors.
- **Timed actors**: All the actors that implement the TimedActor interface can be used in CT, as long as they do not also implement SequenceActor. These actors include plotters that are designed to plot timed signals.
- **Sequence actors**: None of the sequence actors can be used in the CT domain. The reason is that the inputs presented to an actor in CT are fundamentally not sequences. They are samples of continuous-time signals. Thus, although these samples are in fact presented as a sequence, the sequence depends on decisions the solver makes, which dynamically varies the sample times.

11.4 CT Directors

There are four CT directors -- CTSingleSolverDirector, CTMultiSolverDirector, CTMixedSignalDirector, and CTEmbeddedDirector. The first two can only serve as the top-level director, CTMixedSignalDirector can be both top-level or embedded director, and CTEmbeddedDirector can only be contained in a composite actor. In terms of mixing models of computation, all the directors can direct composite actors that implement other models of computation. Only CTMixedSignalDirector and CTEmbeddedDirector can be contained by other domains. The outside domain of a composite actor with CTMixedSignalDirector can be any discrete domain, such as DE, SDF, PN, CSP, etc. The outside domain of a composite actor with CTEmbeddedDirector can be another CT composite actor or FSM, if the outside domain of the FSM model is CT.

11.4.1 CT Director Parameters

The CT director maintains a set of parameters which controls the simulations. The parameters shared by all CT directors are listed in Table 18 on page 11. Individual directors may have their own (additional) parameters, which will be discussed in their sections.

Table 18: CT Director Parameters

Name	Description	Type	Default Value
ValueResolution	This is used in (implicit method) fixed-point iterations. If in two successive iterations the difference of the states is within this resolution, then the integration step is called converged, and the fixed point is considered reached.	double	1e-6
ErrorTolerance	The upper bound of the local truncation error. Actors that perform integration error control (usually integrators in variable step size ODE solving methods) will compare the estimated local error to this value. If the local error is greater than this value, then the integration step is considered inaccurate, and should be restarted.	double	1e-4
InitStepSize	This is the step size that the user specifies as the desired step size. For fixed step size solvers, this step size will be used in all iterations. For variable step size solvers, this is only a suggestion.	double	0.1
MaxIterationsPer-Step	This is used to avoid the infinite loops in (implicit method) fixed-point iterations. If the number of iterations exceed this value, but the fixed point is still not found, then the fixed-point procedure is considered failed. The step size will be reduced by half and simulation step will be restarted.	int	20
MaxStepSize	The maximum step size used in a simulation. This is an upper bound for adjusting step sizes in variable step-size methods. This value can be used to avoid sparse time points when the system dynamic is simple.	double	1.0
MinStepSize	The lower bound for adjusting step sizes. This step size is used for the first step after the breakpoint. In addition, at any time of a variable step size simulation, if this step size is used and the errors are still not tolerable, the simulation aborts.	double	1e-5
StartTime	The start time of the simulation. This is only applicable when CT is the top level domain. Otherwise, the CT director should follow the times of its executive director.	double	0.0
StopTime	The stop time of the simulation. This is only applicable when CT is the top level domain.	double	1.0
TimeResolution	This controls the comparison of time. Since time in CT is a double precision real number, it is sometime impossible to reach or step at a specific time point. If two time points are within this resolution, then they are considered identical.	double	1e-10

11.4.2 CTSingleSolverDirector

As the name implies, this director can only have one ODE solver, and the simulation is done using this ODE solving method. This is the simplest (and sometimes not so accurate) CT director. It is sometimes more accurate to use multiple solvers to deal with different sections of a simulation; for example, it is usually better to use another ODE solver at the breakpoints. But when the system does not have breakpoints or the user just wants a first-order approximation, this director is a lightweight choice.

This director has an additional parameter, ODESolver, as shown in Table 19 on page 12.

Table 19: Additional Parameter for CTSingleSolverDirector

Name	Description	Type	Default Value
ODESolver	The fully qualified class name for the ODE solver class.	string	ptolemy.domains.ct.kernel.solver.ForwardEulerSolver

A CTSingleSolverDirector can direct a system that has composite actors implementing other models of computation. One simulation iteration is done in two phases: the continuous phase and the discrete phase. Let the current iteration be *n*. In the continuous phase, the differential equations are integrated from time t_{n-1} to t_n. After that, in the discrete phase, all (discrete) events which happen at t_n are processed. The step size control mechanism will guarantee that no events will happen between t_{n-1} and t_n.

11.4.3 CTMultiSolverDirector

A CTMultiSolverDirector can be considered a fancy CTSingleSolverDirector. The director can have two ODE solvers — one for ordinary use and one specially for breakpoints. The first integration step after a breakpoint is somewhat special, since the history information is useless. Some integration methods, like high-order implicit methods, are not capable of self starting, so another (low-order) solver may be introduced to regenerate the history information.

This director has one more parameter than the CTSingleSolverDirector, as shown in Table 20 on page 12.

Table 20: Additional Parameter for CTMultiSolverDirector

Name	Description	Type	Default Value
Breakpoint-ODE-Solver	The fully qualified class name for the breakpoint ODE solver class.	string	ptolemy.domains.ct.kernel.solver.BackwardEulerSolver

At the first step after the breakpoint, the integration step size is set to the minimum step size.

11.4.4 CTMixedSignalDirector

This director is design to be an embedded director when a CT subsystem is contained in an event-based system, like DE or DT. As proved in [51], when a CT subsystem is contained in an event-based domain, the CT subsystem should run ahead of the global time, and be ready for rollback. This director implements this optimistic execution.

Since the outside domain is event-based, each time the embedded CT subsystem is fired, the input data are events. In order to interpret the events as continuous signals, breakpoints have to be introduced. So this director extends CTMultiSolverDirector, which always has two ODE solvers. There is one more parameter used by this director — the MaxRunAheadLength, as shown in Table 21 on

page 13.

Table 21: Additional Parameter for CTMixedSignalDirector

Name	Description	Type	Default Value
MaxRunAhead-Length	The maximum length of time for the CT subsystem to run ahead of the global time.	double	1.0

When the CT subsystem is fired, the CTMixedSignalDirector will get the current time τ and the next iteration time τ' from the outer domain, and take the $\min(\tau - \tau', l)$ as the fire end time, where l is the value of the parameter MaxRunAheadLength. The execution is ended either when the fire end time is reached or an output event is detected.

This director supports rollback; that is when the state of the continuous subsystem is confirmed (by knowing that no events with a time earlier than the CT current time will present), the state of the system is marked. If an optimistic execution is known to be wrong, the state of the CT subsystem will roll back to the latest marked state.

11.4.5 CTEmbeddedDirector

This director is used when a CT subsystem is embedded in another continuous system, either directly or through a hierarchy of finite state machines. This director is typically used in the hybrid system scenario [52]. This director can pass the step size control information up to the container domain. To achieve this, the director much be contained in a CTCompositeActor, which will pass the step size control queries from the outer domain to the inner domain.

This director extends CTMultiSolverDirector, but has no additional parameters. The biggest difference of this director and the CTMixedSignalDirector is that this director does not support rollback. In fact, when CT subsystem is embedded in a continuous time environment, it does not need to rollback.

11.5 CT Domain Demos

There are some demos in the CT domain showing how the domain works and the interaction with other domains.

11.5.1 Lorenz System

The Lorenz System (see, for example, pp. 213-214 in [22]) is a famous nonlinear dynamic system that shows the unstable close orbit. The system is given by:

$$\begin{aligned} \dot{x}_1 &= \sigma(x_2 - x_1) \\ \dot{x}_2 &= (\lambda - x_3)x_1 - x_2 \\ \dot{x}_3 &= x_1 \cdot x_2 - b \cdot x_3 \end{aligned} \tag{30}$$

The system is built by integrators and domain polymorphic actors, as shown in figure 11.5.

The LorenzApplet shows the chaotic attractor of the system state projected on to the (x_1, x_2) plane. The initial conditions of the state variables are all 1.0. The default value of the parameters are: $\sigma = 1, \lambda = 25, b = 2.0$.

11.5.2 Micro Accelerator with Digital Feedback.

Micro accelerometers are MEMS devices that use beams, gaps, and electrostatics to measure acceleration. Beams and anchors, separated by gaps, form parallel plate capacitors. When the device is accelerated in the sensing direction, the displacement of the beams causes the change of the gap sizes, which further causes the change of the capacity. By measuring the change of capacity (using the capacitor bridge), the accelerate can be obtained accurately. Feedback can be applied to the beams by charging the capacitors. Using feedback can reduce the sensitivity to process variations, eliminate mechanical resonances, and increase sensor bandwidth, selectivity and dynamic range.

Sigma-delta modulation [15], also called the pulse density modulation or the bang-bang control, is a digital feedback technique, which gets the A/D conversion functionality “for free”. Figure 11.6 shows the conceptual diagram of system. The central part of the digital feedback is an one bit quantizer.

We implemented the system as Mark Alan Lemkin designed in [50]. As shown in the figure 11.7, the second order CT subsystem is used to model the beam. The voltage on the beam-gap capacitor is sampled every *T* seconds (much faster than the required output of the digital signal), then filtered by a

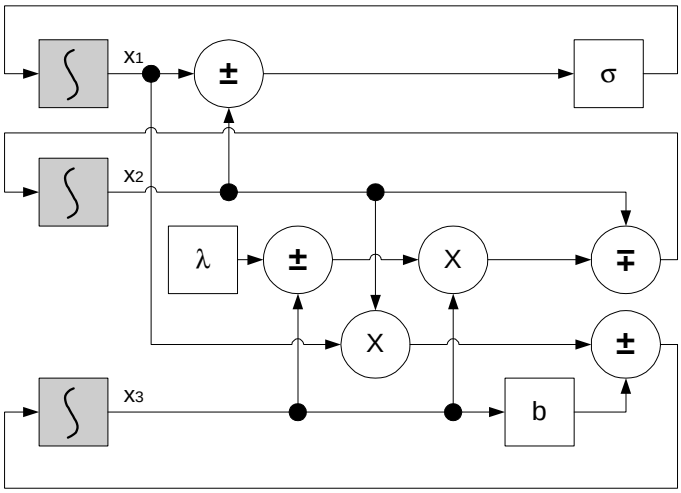


FIGURE 11.5. Block diagram for the Lorenz system.

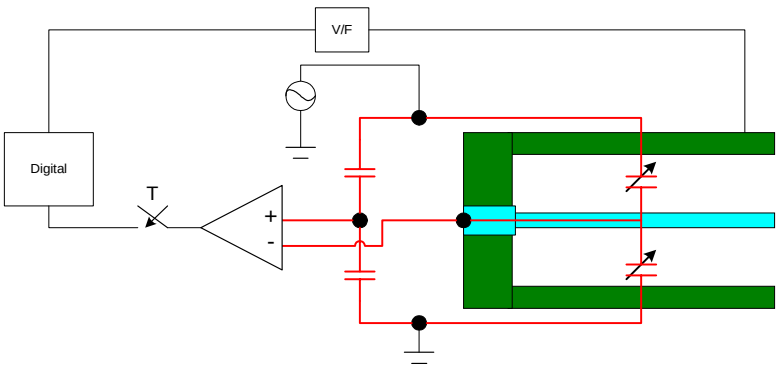


FIGURE 11.6. Micro-accelerator with digital feedback

lead compensator (FIR filter), and fed to an one-bit quantizer. The outputs of the quantizer are converted to force and fed back to the beams. The outputs are also counted and averaged every *NT* seconds to produce the digital output. In our example, the external acceleration is a sine wave.

11.5.3 Thermostat System

The ThermostatApplet shows a simple thermostat system. The temperature of the room is expected to be controlled between 0 and 0.2.

The system has two states, *heating* and *cooling*, as shown in figure 11.8. In the heating state, the temperature of the room is increased linearly, or, in terms of a differential equation:

$$\dot{x} = 1 \tag{31}$$

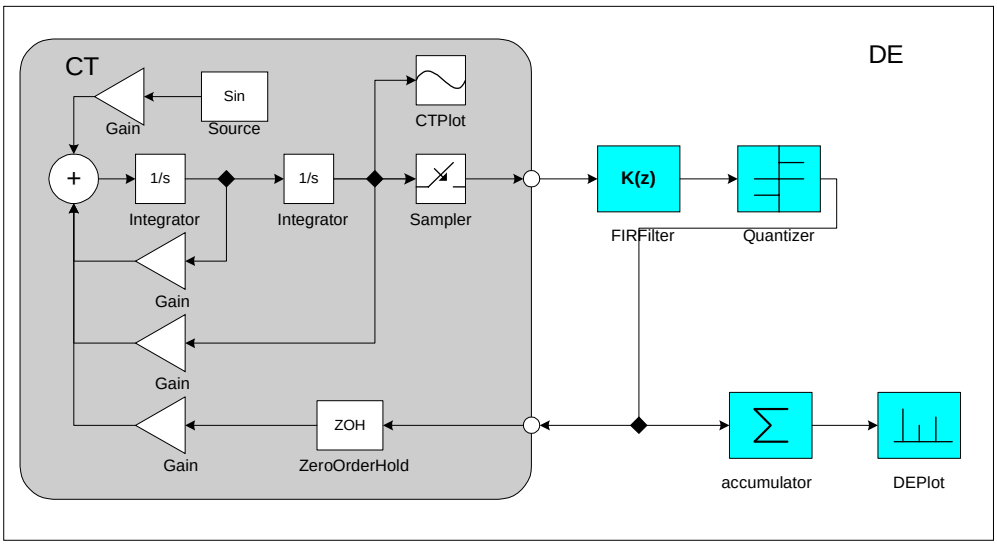


FIGURE 11.7. Block diagram for the micro-accelerator system.

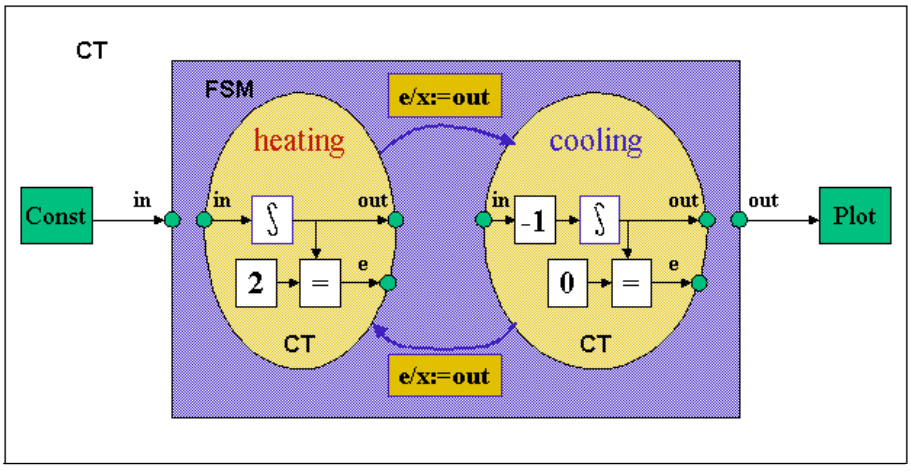


FIGURE 11.8. Block diagram for the thermostat system.

In the cooling state, the temperature is dropped linearly, i.e.

$$\dot{x} = -1 \tag{32}$$

The control rule is that if the temperature reaches 0.2 degree, then switch the controller to the cooling state; if the temperature decreases to 0 degree then switch the controller to the heating state.

Although the system does not have very interesting dynamics, it can be used to illustrate the accuracy of detecting events, and the ability to simulate hybrid systems in Ptolemy II.

11.6 Implementations

The CT domain consists of the following packages, shown in figure 11.9.

The ct.kernel.util package provides the basic data structure — TotallyOrderedSet, which is used to implement the list to store break points. The UML for this package is shown in figure 11.10. A totally ordered set is a set (i.e. no duplicated elements), in which the elements are totally comparable. This data structure is used to store the break points since break points are processed in chronological order.

The ct.kernel package is key. It provided interfaces to classify actors. The classes, including the CTBaseIntegrator class and the ODESolver class, are shown in figure 11.11.

CT directors implement the semantics of the continuous time execution. These classes, including

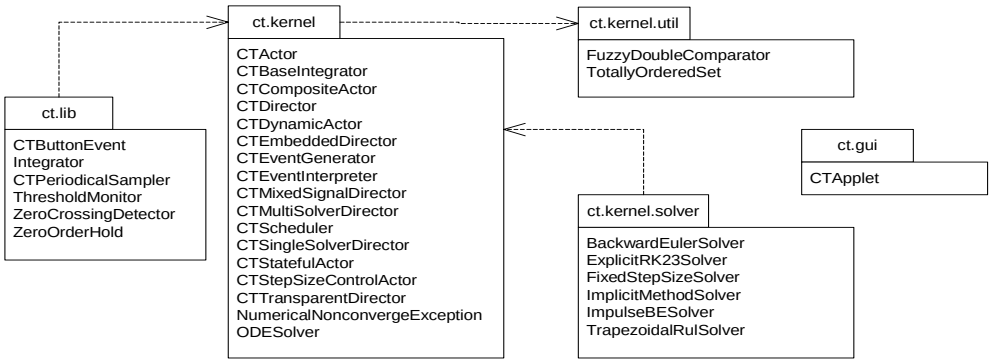


FIGURE 11.9. The packages in the CT domain.

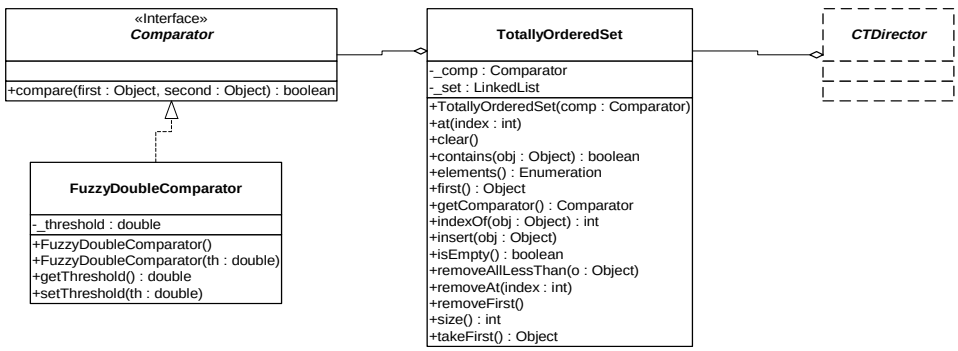


FIGURE 11.10. UML for ct.kernel.util package.

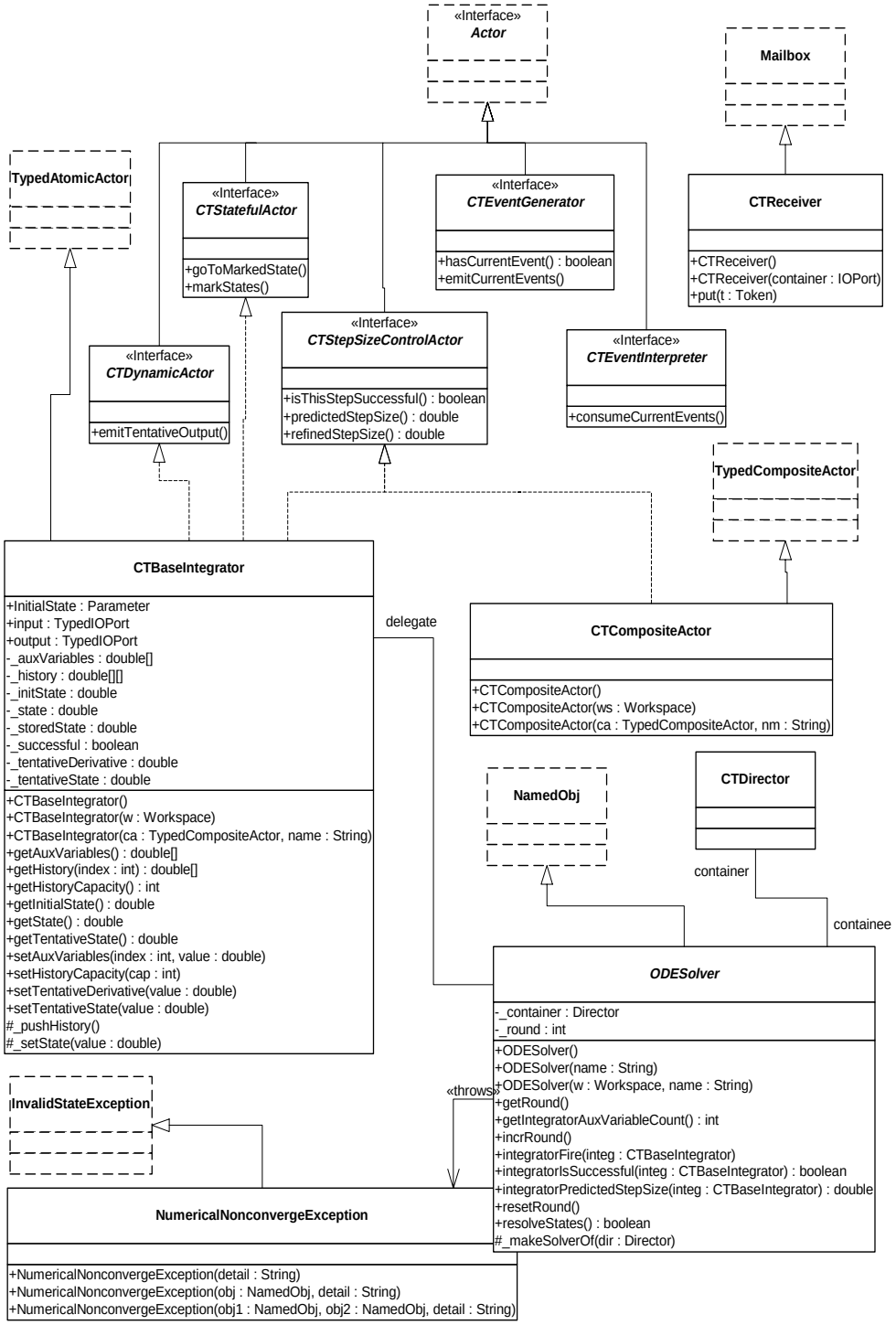


FIGURE 11.11. UML for ct.kernel package, actor related classes.

`CTScheduler`, are shown in figure 11.12.

The ct.kernel.solver package provided a set of ODE solvers. The classes are shown in figure 11.13.

11.7 Technical Details

11.7.1 Scheduling

The scheduler partitions a CT system into two clusters: the *state transition cluster* and the *output cluster.* In a particular system, these clusters may overlap.

The state transition cluster includes all the actors that are in the signal flow path for evaluating the *f*( ) function in (4). It starts from the source actors and the outputs of the integrators, and ends at the inputs of the integrators. A topological sort of the cluster provides an enumeration of the actors in the order of their firings. This enumeration is called the *state transition schedule.* After the integrators produce tokens representing x_t, one iteration of the state transition schedule gives the tokens representing $\dot{x}_t = f(x_t, u(t), t)$ back to the integrators.

The output cluster consists actors that are involved in the evaluation of the output map *g*( ) in (5). It is also similarly sorted in topological order. The *output schedule* starts from the source actors and the integrators, and ends at the sink actors.

For example, for the system shown in figure 11.3, the state transition schedule is

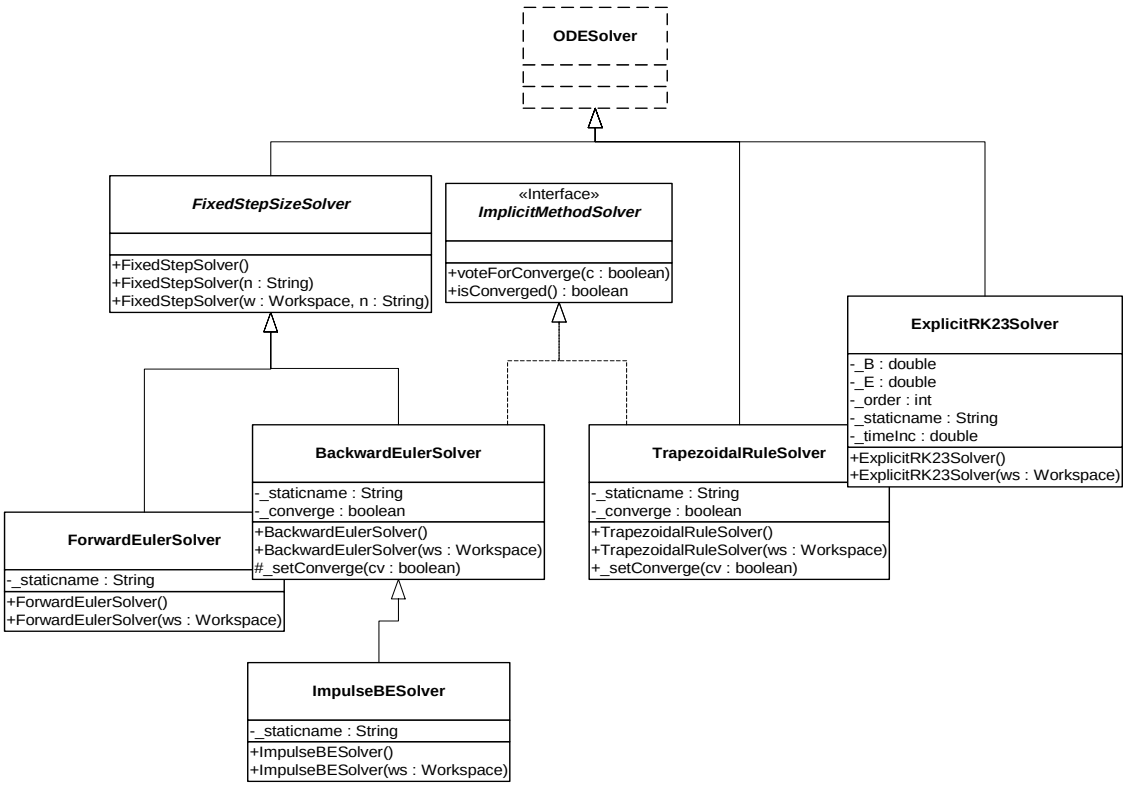


FIGURE 11.13. UML for ct.kernel.solver package.

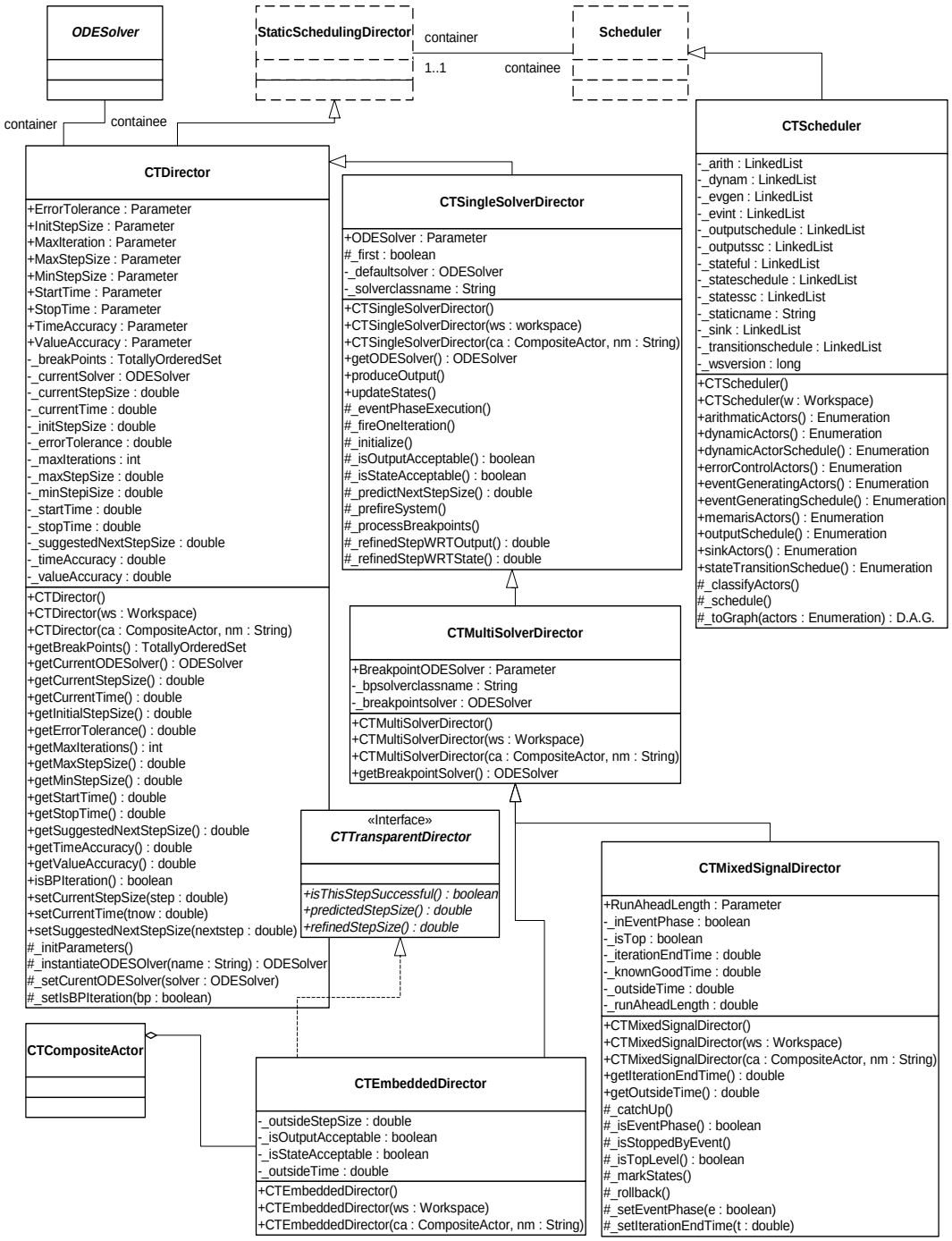


FIGURE 11.12. UML for ct.kernel package, director related classes.

```
U-G1-G2-G3-A
```

where the order of `G1`, `G2`, and `G3` are interchangeable. The output schedule is

```
G4-Y
```

The event generating schedule is empty.

A special situation that must be taken care of is the firing order of a chain of integrators, as shown in figure 11.14. For the implicit integration algorithms, the order of firings determines two distinct kinds of fixed point iterations. If the integrators are fired in the topological order, namely $x_1 \rightarrow x_2$ in our example, the iteration is called the *Gauss-Seidel iteration*. That is, x_2 always uses the new guess from x_1 in this iteration for its new guess. On the other hand, if they are fired in the reverse topological order, the iteration is called the *Gauss-Jacobi iteration*, where x_2 uses the guess of x_1 in the last iteration for its new guess. The two iterations both have their pros and cons, which are thoroughly discussed in [64]. Gauss-Seidel iteration is considered faster in the speed of convergence than Gauss-Jacobi. For explicit integration algorithms, where the new states x_{t_n} are calculated solely from the history inputs up to $\dot{x}_{t_{n-1}}$, the integrators must be fired in their reverse topological order. For simplicity, the scheduler of the CT domain, at this time, always returns the reversed topological order of a chain of integrators. This order is considered safe for all integration algorithms.

11.7.2 Controlling Step Sizes

Choosing the right time points to approximate a continuous time system behavior is one of the major tasks of simulation. There are three factors that may impact the choice of the step size.

- *Error control*. For all integration algorithms, the *local error* at time t_n is defined as a vector norm (say, the 2-norm) of the difference between the actual solution $x(t_n)$ and the approximation x_{t_n} calculated by the integration method, given that the last step is accurate. That is, assuming $x_{t_{n-1}} = x(t_{n-1})$ then

$$E_{t_n} = \left\| x_{t_n} - x(t_n) \right\| . \tag{33}$$

  It can be shown that by carefully choosing the parameters in the integration algorithms, the local error is approximately of the *p*-th order of the step size, where *p*, an integer closely related to the number of *f*( ) function evaluations in one integration step, is called the *order* of the integration algorithm. I.e. $E_{t_n} \sim O((t_n - t_{n-1})^p)$. Therefore, in order to achieve an accurate solution, the step size should be chosen to be small. But on the other hand, small step sizes means long simulation time. In general, the choice of step size reflects the trade-off between speed and accuracy of a simulation.
- *Convergence*. The local contraction mapping theorem (Theorem 2 in Appendix F) shows that for implicit ODE solvers, in order to find the fixed point at t_n, the map $F_I(\ )$ in (15) must be a (local) contraction map, and the initial guess must be within an ε ball (the contraction radius) of the solution. It can be shown that $F_I(\ )$ can be made contractive if the step size is small enough. (The choice of the step size is closely related to the Lipschitz constant). So the general approach for resolving the fixed point is that if the iterating function $F_I(\ )$ does not converge at one step size,

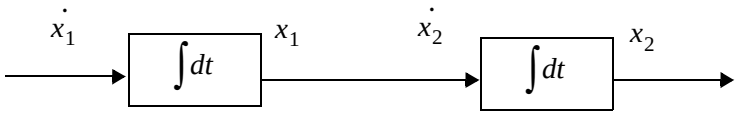


FIGURE 11.14. A chain of integrators.

then reduce the step size by half and try again.

- *Discontinuity.* At discontinuous points, the derivatives of the signals are not continuous, so the integration formula is not applicable. That means the discontinuous points can not be crossed by one integration step. In particular, suppose the current time is *t* and the intended next time point is *t+h.* If there is a discontinuous points at $t + \delta$, where $\delta < h$, then the next step size should be reduced to $t + \delta$. For a PDP, the director can adjust the step size accordingly before starting an integration step. However for an UDP, which is reported "missed" after an integration step, the director should be able to discard its last step and restart with a smaller step size to locate the actual discontinuous point.

Notice that convergence and accuracy concerns only apply to some ODE solvers. For example, explicit algorithms do not have the convergence problem, and fixed step size algorithms do not have the error control feature. On the other hand, discontinuity control is a "generic" feature that is independent on the choice of ODE solvers.

11.7.3 Interaction with other domains

See [51] and [52].

Appendix F: Brief Mathematical Background

Theorem 1. [Existence and uniqueness of the solution of an ODE] Consider the initial value ODE problem

$$\begin{aligned} \dot{x} &= f(x, t) \quad . \\ x(t_0) &= x_0 \end{aligned} \tag{34}$$

If *f* satisfies the conditions:

1. [*Continuity Condition*] Let *D* be the set of possible discontinuity points; it may be empty. For each fixed $x \in \Re^n$ and $u \in \Re^m$, the function $f: \Re \setminus \mathrm{D} \rightarrow \Re^n$ in (34) is continuous. And $\forall \tau \in D$, the left-hand and right-hand limit $f(x, u, \tau^-)$ and $f(x, u, \tau^+)$ are finite.
2. [*Lipschitz Condition*] There is a piecewise continuous bounded function $k: \Re \rightarrow \Re^+$, where $\Re^+$ is the set of non-negative real numbers, such that $\forall t \in \Re$, $\forall \zeta, \xi \in \Re^n$, $\forall u \in \Re^m$

$$\| f(\xi, u, t) - f(\zeta, u, t) \| \leq k(t) \| \xi - \zeta \| . \tag{35}$$

Then, for each initial condition $(t_0, x_0) \subseteq \Re \times \Re^n$ there exists a *unique* continuous function $\psi: \Re \rightarrow \Re^n$ such that,

$$\psi(t_0) = x_0 \tag{36}$$

and

$$\dot{\psi}(t) = f(\psi(t), u(t), t) \qquad \forall t \in \Re \setminus \mathrm{D} . \tag{37}$$

This function $\psi(t)$ is called the *solution* through (t_0, x_0) of the ODE (34).

◆

Theorem 2. [Contraction Mapping Theorem.] If $F: \Re^n \rightarrow \Re^n$ is a local contraction map at *x* with contraction radius ε, then there exists a unique fixed point of *F* within the ε ball centered at *x*. I.e. there exists a unique $\sigma \in \Re^n$, $\| \sigma - x \| \leq \varepsilon$, such that $\sigma = F(\sigma)$. And $\forall \sigma_0 \in \Re^n$, $\| \sigma_0 - x \| \leq \varepsilon$, the sequence

$$\sigma_1 = F(\sigma_0), \sigma_2 = F(\sigma_1), \sigma_3 = F(\sigma_2), \ldots \tag{38}$$

converges to σ.

◆

12

DE Domain

Authors: Lukito Muliadi
Edward A. Lee

12.1 Introduction

The discrete-event (DE) domain supports time-oriented models of systems such as queueing systems, communication networks, and digital hardware. In this domain, actors communicate by sending *events*, where an event is a data value (a token) and a *time stamp*. A DE scheduler ensures that events are processed chronologically according to this time stamp by firing those actors whose available input events are the oldest (having the earliest time stamp of all pending events).

A key strength in our implementation is that simultaneous events (those with identical time stamps) are handled systematically and deterministically. A second key strength is that the global event queue uses an efficient structure that minimizes the overhead associated with maintaining a sorted list with a large number of events.

12.1.1 Model Time

In the DE model of computation, time is *global*, in the sense that all actors share the same global time. The *current time* of the model is often called the *model time* or *simulation time* to avoid confusion with current real time.

As in most Ptolemy II domains, actors communicate by sending tokens through ports. Ports can be input ports, output ports, or both. Tokens are sent by an output port and received by all input ports connected to the output port through relations. When a token is sent from an output port, it is packaged as an event and stored in a global event queue. By default, the time stamp of an output is the model time, although specialized DE actors can produce events with future time stamps.

Actors may also request that they be fired at some time in the future by calling the fireAt() method of the director. This places a *pure event* (one with a time stamp, but no data) on the event queue. A pure event can be thought of as setting an alarm clock to be awakened in the future. Sources (actors

with no inputs) are thus able to be fired despite having no inputs to trigger a firing. Moreover, actors that introduce *delay* (outputs have larger time stamps than the inputs) can use this mechanism to schedule a firing in the future to produce an output.

In the global event queue, events are sorted based on their time stamps and their destination ports (or actors, in the case of pure events). An event is removed from the global event queue when the *model time* reaches its time stamp, and if it has a data token, then that token is put into the destination input port.

At any point in the execution of a model, the events stored in the global event queue have time stamps greater than or equal to the model time. The DE director is responsible for advancing (i.e. incrementing) the model time when all events with time stamps equal to the current model time have been processed (i.e. the global event queue only contains events with time stamps strictly greater than the current time). The current time is advanced to the smallest time stamp of all events in the global event queue. This advancement marks the beginning of an iteration.

12.1.2 Iteration

At each iteration, after advancing the current time, the director chooses some events in the global event queue based on their time stamps and destination actors. The DE director chooses events according to these rules:

- Find a non-empty set *E* of events associated with the smallest time stamp.
- If the set of events contains more than one event, find the one with highest priority, e_{max}. Simultaneous events are prioritized carefully by means of topological sort of the actors (to ensure deterministic behavior).
- Choose all events in *E* that have the same destination actor as e_{max}. Note that the destination actor of an event is the actor containing the destination input port of that event, or if it is a pure event, the actor to be fired.

The chosen events are then removed from the global event queue and their data tokens are inserted into the appropriate input ports of the destination actor. Then, the director iterates the destination actor; i.e. it invokes prefire(), fire(), and postfire().

A firing may produce additional events at the current model time (the actor reacts *instantaneously*, or has *zero delay*). There also may be other events with time stamp equal to the current model time still pending on the event queue. The DE director repeats the above procedure until there are no more events with time stamp equal to the current time. This concludes one iteration of the model.

12.1.3 Getting a Model Started

Before one of the iterations described above can be run, there have to be initial events in the global event queue. Actors may produce initial pure events in their initialize() method. Normally, they cannot produce events with data in their initialize() method because type resolution has not been done, so the types of the ports are not set. Thus, to get a model started, at least one actor must be used that produces such pure events. All the domain-polymorphic timed sources described in the Actor Libraries chapter produce such events. We can define the *start time* to be the smallest time stamp of these initial events.

12.1.4 Stopping Execution

Execution stops when one of these conditions become true:

- The current time reaches the *stop time*, set by calling the setStopTime() method of the DE director.

- The global event queue becomes empty.

Events at the stop time are processed before stopping the model execution. The execution ends by calling the wrapup() method of all actors.

12.2 Overview of The Software Architecture

The UML static structure diagram for the DE kernel package is shown in figure 12.1. For model builders, the important classes are DEDirector, DEActor and DEIOPort. At the heart of DEDirector is a global event queue that sorts events according to their time stamps and priorities.

The DEDirector uses an efficient implementation of the global event queue, a calendar queue data structure [11]. The time complexity for this particular implementation is O(1) in both enqueue and dequeue operations. This means that the time complexity for enqueue and dequeue operations is independent of the number of pending events in the global event queue. For extensibility, different implementations of the global event queue can be realized by implementing the DEEventQueue interface and specifying the event queue using the appropriate constructor for DEDirector.

The DEActor class provides convenient methods to access time, since time is an essential part of a timed domain like DE. Nonetheless, actors in a DE model are not required to be derived from the DEActor class. Simply deriving from TypedAtomicActor gives you the same capability, but without the convenience. In the latter case, time is accessible through the director.

The DEIOPort class is be used by actors that are specialized to the DE domain. It supports annotations that inform the scheduler about delays through the actor. It also provides two additional methods, overloaded versions of broadcast() and send(). The overloaded versions have a second argument for the time delay, allowing actors to send output data with a time delay (relative to current time).

Domain polymorphic actors, such as those described in the Actor Libraries chapter, have as ports instances of TypedIOPort, not DEIOPort, and therefore cannot produce events in the future directly by sending it through output ports. Note that tokens sent through TypedIOPort are treated as if they were sent through DEIOPort with the time delay argument equal to zero. Domain polymorphic actors can produce events in the future indirectly by using the fireAt() method of the director. By calling fireAt(), the actor requests a refiring in the future. The actor can then produce a delayed event during the refiring.

12.3 The DE Actor Library

The DE domain has a small library of actors in the ptolemy.domains.de.lib package, shown in figure 12.2. These actors are particularly characterized by implementing both the TimedActor and SequenceActor interfaces. These actors use the current model time, and in addition, assume they are dealing with sequences of discrete events. Some of them use domain-specific infrastructure, such as the convenience class DEActor and the base class DETransformer. The DETransformer class provides in input and output port that are instances of DEIOPort. The Delay and Server actors use facilities of these ports to influence the firing priorities.

12.4 Mutations

The DE director tolerates changes to the model during execution. The change should be queued

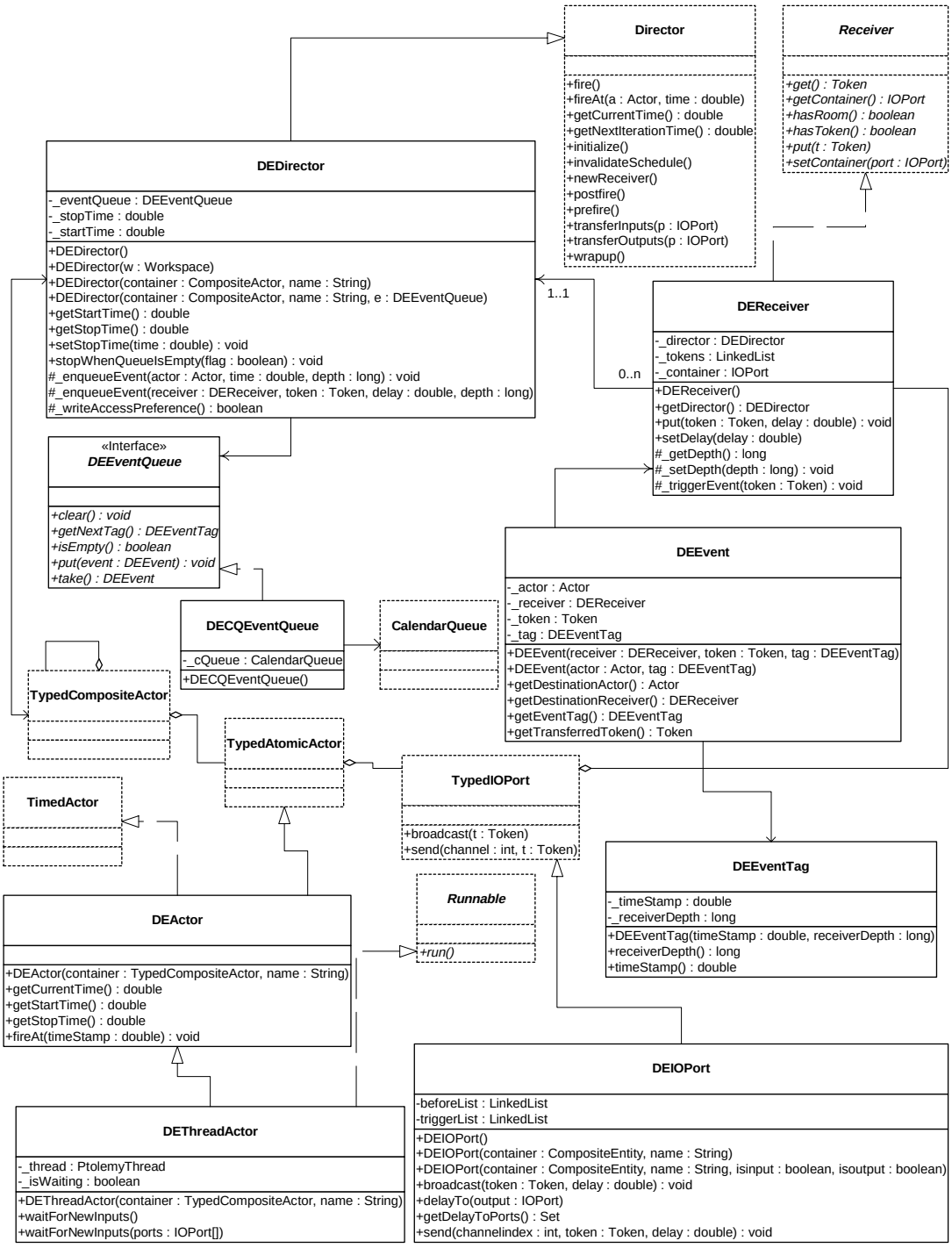


FIGURE 12.1. UML static structure diagram for the DE kernel package.

with the director or manager using requestChange(). While invoking those changes, the method invalidateSchedule() is expected to be called, notifying the director that the topology it used to calculate the priorities of the actors is no longer valid. This will result in the priorities being recalculated the next time prefire() is invoked.

However, there is one subtlety. If an actor produces events in the future via DEIOPort, then the destination actor will be fired even if it has been removed from the topology by the time the execution reaches that future time. This may not always be the expected behavior. The Delay actor in the DE library behaves this way.

12.5 Writing DE Actors

It is very common in DE modeling to include custom-built actors. No pre-defined actor library seems to prove sufficient for all applications. For the most part, writing actors for the DE domain is no different than writing actors for any other domain. Some actors, however, need to exercise particular control over time stamps and actor priorities. Such actors use instances of DEIOPort rather than TypedIOPort. The first section below gives general guidelines for writing DE actors and domain-polymorphic actors that work in DE. The second section explains in detail the priorities, in preparation for the following section, which gives an example. The final section discusses actors that operate as a Java thread.

12.5.1 General Guidelines

The points to keep in mind are:

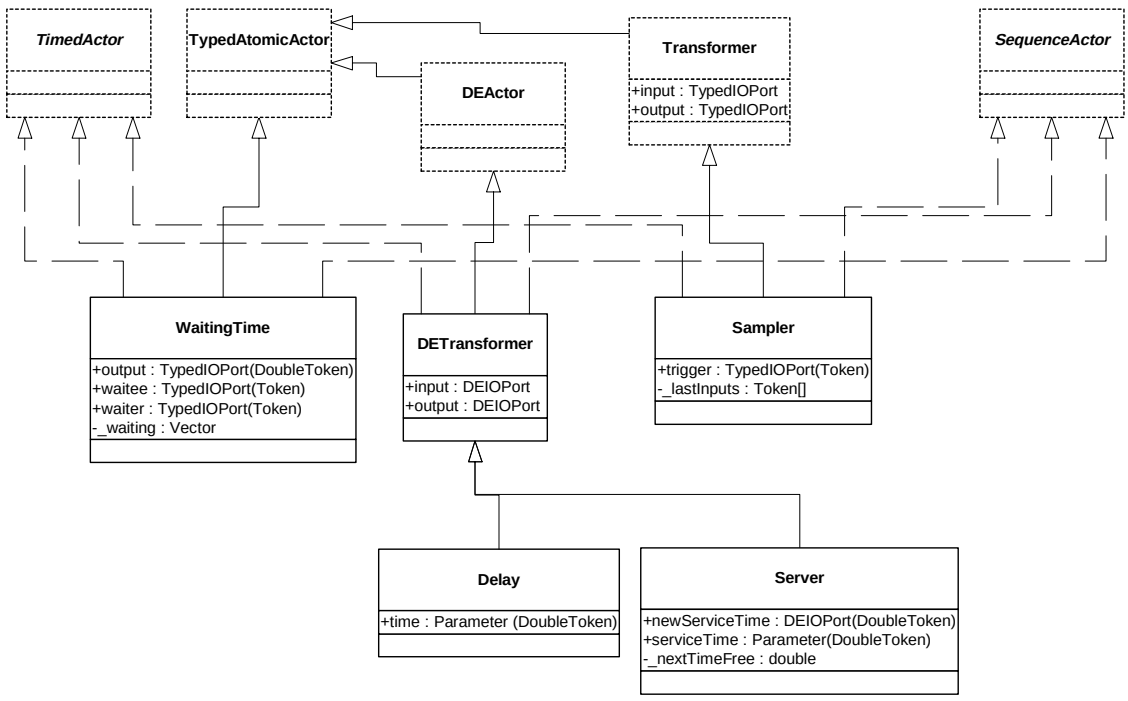


FIGURE 12.2. The library of DE-specific actors.

- When an actor fires, not all ports have tokens, and some ports may have more than one token. The time stamps of the events that contained these tokens are no longer explicitly available. The current model time is assumed to be the time stamp of the events.
- If the actor leaves unconsumed tokens on its input ports, then it will be iterated again before model time is advanced. This ensures that the current model time is in fact the time stamp of the input events. However, occasionally, an actor will want to leave unconsumed tokens on its input ports, and not be fired again until there is some other new event to be processed. To get this behavior, it should return *false* from prefire(). This indicates to the DE director that it does not wish to be iterated.
- If the actor returns *false* from postfire(), then the director will not fire that actor again. Events that are destined for that actor are discarded.
- When an actor produces an output token, the time stamp for the output event is taken to be the current model time. If the actor wishes to produce an event at a future model time, one way to accomplish this is to call the director's fireAt() method to schedule a future firing, and then to produce the token at that time. A second way to accomplish this is to use instances of DEIOPort and use the overloaded send() or broadcast() methods that take a time delay argument.
- The DEIOPort class (see figure 12.1) can produce events in the future, but there is an important subtlety with using these methods. Once an event has been produced, it cannot be retracted. In particular, even if the actor is deleted before model time reaches that of the future event, the event will be delivered to the destination. If you use fireAt() instead to generate delayed events, then if the actor is deleted (or returns *false* from postfire()) before the future event, then the future event will not be produced.
- By convention in Ptolemy II, actors update their state only in the postfire() method. In DE, the fire() method is only invoked once per iteration, so there is no particular reason to stick to this convention. Nonetheless, we recommend that you do in case your actor becomes useful in other domains. The simplest way to ensure this is follow the following pattern. For each state variable, such as a private variable named *_count*,

```
private int _count;
```

create a shadow variable

```
private int _countShadow;
```

Then write the methods as follows:

```
public void fire() {
   _countShadow = _count;
   ... perform some computation that may modify _countShadow ...
}
public boolean postfire() {
   _count = _shadowCount;
   return super.postfire();
}
```

This ensures that the state is updated only in postfire().

In a similar fashion, delayed outputs (produced by either mechanism) should be produced only in the postfire() method, since a delayed outputs are persistent state. Thus, fireAt() should be called in postfire() only, as should the overloaded send() and broadcast() of DEIOPort.

12.5.2 Simultaneous Events

An important aspect of a DE domain is the prioritizing of simultaneous events. This gives the domain a dataflow-like behavior for events with identical time stamps. It is done by assigning ranks to actors. The ranks are drawn from the set of non-negative integers. They are uniquely assigned; i.e. no two distinct actors are assigned the same rank. Simultaneous events with highest priority are those destined to actors with the lowest ranks. The ranks are determined by a *topological sort* of a *directed acyclic graph* (DAG) of the actors.

The DAG of actors follows the topology of the graph, except when there are declared delays. Consider the simple topology shown in figure 12.3. Assume that actor *Y* is a zero-delay actor, meaning that its output events have the same time stamp as the input events (this is suggested by the dashed arrow). Suppose that actor *X* produces an event with time stamp τ. That event is available at ports *B* and *D*, so the scheduler could choose to fire actors *Y* or *Z*. Which should it fire? Intuition tells us it should fire the upstream one first, *Y*, because that firing may produce another event with time stamp τ at port *D* (which is presumably a multiport). It seems logical that if actor *Z* is going to get one event on each input channel with the same time stamp, then it should see those events in the same firing. If there are simultaneous events at *B* and *D*, then the one at *B* should have higher priority.

Once the DAG is constructed, it is sorted topologically. This simply means that an ordering of actors is assigned such that an upstream actor in the DAG is earlier in the ordering than a downstream actor. This ordering is not unique, meaning that the priorities assigned to actors are somewhat arbitrary. As long as actors are communicating only via events, however, then these arbitrary choices will have no impact on the end result of executing the model. We say that the execution is *deterministic*.

There are situations where constructing a DAG following the topology is not possible. Consider the topology shown in figure 12.4. It is evident from the figure that the topology is not acyclic. Indeed,

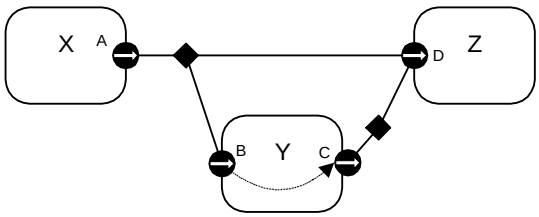


FIGURE 12.3. If there are simultaneous events at B and D, then the one at B should have higher priority because it may trigger another simultaneous event at E.

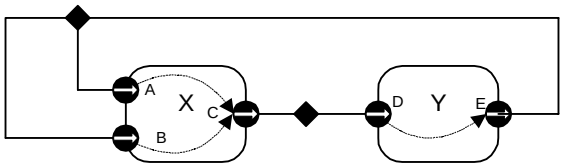


FIGURE 12.4. An example of a directed zero-delay loop.

figure 12.4 depicts a zero-delay directed loop where topological sort cannot be done. The director will refuse to run the model, and will terminate with an error message. This is called a *zero-delay loop.*

The Delay actor in DE is a domain-specific actor that asserts a delay relationship between its input and output. Thus, if we insert a Delay actor in the loop, as shown in figure 12.5, then constructing the DAG becomes once again possible. The Delay actor breaks the precedences.

Note in particular that the Delay actor breaks the precedences *even if its delay parameter is set to zero*. Thus, the DE domain is perfectly capable of modeling zero-delay loops, but the model builder has to specify the order in which events should be processed by placing a Delay actor with a zero value for its parameter.

12.5.3 Examples

Simplified Delay Actor. An example of a domain-specific actor for DE is shown in figure 12.6. This actor delays input events by some amount specified by a parameter. The domain-specific features of the actor are shown in bold. They are:

- It uses DEIOPort rather than TypedIOPort.
- It has the statement:

```
input.delayTo(output);
```

  This statement declares to the director that this actor implements a delay from input to output. The actor uses this to break the precedences when constructing the DAG to find priorities.
- It uses an overloaded send() method, which takes a delay argument, to produce the output. Notice that the output is produced in the postfire() method, since by convention in Ptolemy II, persistent state is not updated in the fire() method, but rather is updated in the postfire() method.

Server Actor. The Server actor in the DE library (see figure 12.2) uses a rich set of behavioral properties of the DE domain. A server is a process that takes some amount of time to serve "customers." While it is serving a customer, other arriving customers have to wait. This actor can have a fixed service time (set via the parameter *serviceTime*, or a variable service time, provided via the input port *newServiceTime*). A typical use would be to supply random numbers to the *newServiceTime* port to generate random service times. These times can be provided at the same time as arriving customers to get an effect where each customer experiences a different, randomly selected service time.

The (compacted) code is shown in figure 12.7. This actor extends DETransformer, which has two public members, *input* and *output*, both instances of DEIOPort. The constructor makes use of the delayTo() method of these ports to indicate that the actor introduces delay between its inputs and its output.

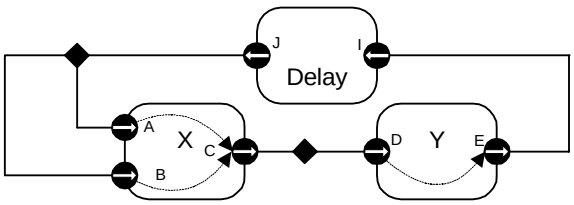


FIGURE 12.5. A Delay actor can be used to break a zero-delay loop.

The actor keeps track of the time at which it will next be free in the private variable _nextTimeFree. This is initialized to minus infinity to indicate that whenever the model begins executing, the server is free. The prefire() method determines whether the server is free by comparing this private variable against the current model time. If it is free, then this method returns true, indicating to the scheduler that it can proceed with firing the actor. If the server is not free, then the prefire() method checks to see whether there is a pending input, and if there is, requests a firing when the actor will become free. It then returns false, indicating to the scheduler that it does not wish to be fired at this time. Note that the prefire() method uses the methods getCurrentTime() and fireAt() of DEActor, which are simply convenient interfaces to methods of the same name in the director.

The fire() method is invoked only if the server is free. It first checks to see whether the newSer-

```
package ptolemy.domains.de.lib.test;

import ptolemy.actor.TypedAtomicActor;
import ptolemy.domains.de.kernel.DEIOPort;
import ptolemy.data.DoubleToken;
import ptolemy.data.Token;
import ptolemy.data.expr.Parameter;
import ptolemy.actor.TypedCompositeActor;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;
import ptolemy.kernel.util.Workspace;

public class SimpleDelay extends TypedAtomicActor {

    public SimpleDelay(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        input = new DEIOPort(this, "input", true, false);
        output = new DEIOPort(this, "output", false, true);
        delay = new Parameter(this, "delay", new DoubleToken(1.0));
        delay.setTypeEquals(DoubleToken.class);
        input.delayTo(output);
    }

    public Parameter delay;
    public DEIOPort input;
    public DEIOPort output;
    private Token _currentInput;

    public Object clone(Workspace ws) throws CloneNotSupportedException {
        SimpleDelay newobj = (SimpleDelay)super.clone(ws);
        newobj.delay = (Parameter)newobj.getAttribute("delay");
        newobj.input = (DEIOPort)newobj.getPort("input");
        newobj.output = (DEIOPort)newobj.getPort("output");
        return newobj;
    }

    public void fire() throws IllegalActionException {
        _currentInput = input.get(0);
    }

    public boolean postfire() throws IllegalActionException {
        output.send(0, _currentInput,
                ((DoubleToken)delay.getToken()).doubleValue());
        return super.postfire();
    }
}
```

FIGURE 12.6. A domain-specific actor in DE.

```
package ptolemy.domains.de.lib;
import statements ...

public class Server extends DETransformer {

    public DEIOPort newServiceTime;
    public Parameter serviceTime;

    private Token _currentInput;
    private double _nextTimeFree = Double.NEGATIVE_INFINITY;

    public Server(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException  {
         super(container, name);
         serviceTime = new Parameter(this, "serviceTime", new DoubleToken(1.0));
         serviceTime.setTypeEquals(DoubleToken.class);
         newServiceTime = new DEIOPort(this, "newServiceTime", true, false);
         newServiceTime.setTypeEquals(DoubleToken.class);
         output.setTypeAtLeast(input);
         input.delayTo(output);
         newServiceTime.delayTo(output);
     }

    ... attributeChanged(), clone() methods ...

    public void initialize() throws IllegalActionException {
         super.initialize();
         _nextTimeFree = Double.NEGATIVE_INFINITY;
    }

    public boolean prefire() throws IllegalActionException {
         if (getCurrentTime() >= _nextTimeFree) {
             return true;
         } else {
             // Schedule a firing if there is a pending token so it can be served.
             if (input.hasToken(0)) {
                 fireAt(_nextTimeFree);
             }
             return false;
         }
     }

    public void fire() throws IllegalActionException {
         if (newServiceTime.getWidth() > 0 && newServiceTime.hasToken(0)) {
             DoubleToken time = (DoubleToken)(newServiceTime.get(0));
             serviceTime.setToken(time);
         }
         if (input.getWidth() > 0 && input.hasToken(0)) {
             _currentInput = input.get(0);
             double delay = ((DoubleToken)serviceTime.getToken()).doubleValue();
             _nextTimeFree = getCurrentTime() + delay;
         } else {
             _currentInput = null;
         }
    }

    public boolean postfire() throws IllegalActionException {
         if (_currentInput != null) {
             double delay = ((DoubleToken)serviceTime.getToken()).doubleValue();
             output.send(0, _currentInput, delay);
         }
         return super.postfire();
    }
}
```

FIGURE 12.7. Code for the Server actor. For more details, see the source code.

viceTime port is connected to anything, and if it is, whether it has a token. If it does, the token is read and used to update the *serviceTime* parameter. No more than one token is read, even if there are more in the input port, in case one token is being provided per pending customer.

The fire() method then continues by reading an input token, if there is one, and updating _nextTimeFree. The input token that is read is stored temporarily in the private variable _currentInput. The postfire() method then produces this token on the output port, with an appropriate delay. This is done in the postfire() method rather than the fire() method in keeping with the policy in Ptolemy II that persistent state is not updated in the fire() method. Since the output is produced with a future time stamp, then it is persistent state.

Note that when the actor will not get input tokens that are available in the fire() method, it is essential that prefire() return false. Otherwise, the DE scheduler will keep firing the actor until the inputs are all consumed, which will never happen if the actor is not consuming inputs!

Like the SimpleDelay actor in figure 12.6, this one produces outputs with future time stamps, using the overloaded send() method of DEIOPort that takes a delay argument. There is a subtlety associated with this design. If the model mutates during execution, and the Server actor is deleted, it cannot retract events that it has already sent to the output. Those events will be seen by the destination actor, even if by that time neither the server nor the destination are in the topology! This could lead to some unexpected results, but hopefully, if the destination actor is no longer connected to anything, then it will not do much with the token.

12.5.4 Thread Actors

In some cases, it is useful to describe an actor as a thread that waits for input tokens on its input ports. The thread suspends while waiting for input tokens and is resumed when some or all of its input ports have input tokens. While this description is functionally equivalent to the standard description explained above, it leverages on the Java multi-threading infrastructure to save the state information.

Consider the code for the ABRecognizer actor shown in figure 12.8. The two code listings imple-

```
public class ABRecognizer extends DEThreadActor {
   StringToken msg = new StringToken(“Seen AB”);

   // the run method is invoked when the thread
   // is started.
   public void run() {
      while (true) {
         waitForNewInputs();
         if (inportA.hasToken(0)) {
            IOPort[] nextinport = {inportB};
            waitForNewInputs(nextinport);
            outport.broadcast(msg);
         }
      }
   }
```

```
public class ABRecognizer extends DEActor {
   StringToken msg = new StringToken(“Seen AB”);

   // We need an explicit state variable in
   // this case.
   int state = 0;

   public void fire() {
      switch (state) {
         case 0:
            if (inportA.hasToken(0)) {
               state = 1;
               break;
            }
         case 1:
            if (inportB.hasToken(0)) {
               state = 0;
               outport.broadcast(msg);
            }
      }
   }
}
```

FIGURE 12.8. Code listings for two style of writing the ABRecognizer actor.

ment two actors with equivalent behavior. The left one implements it as a threaded actor, while the right one implements it as a standard actor. We will from now on refer to the left one as the threaded description and the right one as the standard description. In both description, the actor has two input ports, `inportA` and `inportB`, and one output port, `outport`. The behavior is as follows.

> *Produce an output event at outport as soon as events at inportA and inportB occurs in that particular order, and repeat this behavior.*

Note that the standard description needs a state variable `state`, unlike the case in the threaded description. In general the threaded description encodes the state information in the position of the code, while the standard description encodes it explicitly using state variables. While it is true that the context switching overhead associated with multi-threading application reduces the performance, we argue that the simplicity and clarity of writing actors in the threaded fashion is well worth the cost in some applications.

The infrastructure for this feature is shown in figure 12.1. To write an actor in the threaded fashion, one simply derives from the DEThreadActor class and implements the run() method. In many cases, the content of the run() method is enclosed in the infinite '`while(true)`' loop since many useful threaded actors do not terminate.

The waitForNewInputs() method is overloaded and has two flavors, one that takes no arguments and another that takes an IOPort array as argument. The first suspends the thread until there is at least one input token in at least one of the input ports, while the second suspends until there is at least one input token in any one of the specified input ports, ignoring all other tokens.

In the current implementation, both versions of waitForNewInputs() clear all input ports before the thread suspends. This guarantees that when the thread resumes, all tokens available are new, in the sense that they were not available before the waitForNewInput() method call.

The implementation also guarantees that between calls to the waitForNewInputs() method, the rest of the DE model is suspended. This is equivalent to saying that the section of code between calls to the waitForNewInput() method is a critical section. One immediate implication is that the result of the method calls that check the configuration of the model (e.g. hasToken() to check the receiver) will not be invalidated during execution in the critical section. It also means that this should not be viewed as a way to get parallel execution in DE. For that, consider the DDE domain.

It is important to note that the implementation serializes the execution of threads, meaning that at any given time there is only one thread running. When a threaded actor is running (i.e. executing inside its run() method), all other threaded actors and the director are suspended. It will keep running until a waitForNewInputs() statement is reached, where the flow of execution will be transferred back to the director. Note that the director thread executes all non-threaded actors. This serialization is needed because the DE domain has a notion of global time, which makes parallelism much more difficult to achieve.

The serialization is accomplished by the use of monitor in the DEThreadActor class. Basically, the fire() method of the DEThreadActor class suspends the calling thread (i.e. the director thread) until the threaded actor suspends itself (by calling waitForNewInputs()). One key point of this implementation is that the threaded actors appear just like an ordinary DE actor to the DE director. The DEThreadActor base class encapsulates the threaded execution and provides the regular interfaces to the DE director. Therefore the threaded description can be used whenever an ordinary actor can, which is everywhere.

The code shown in figure 12.9 implements the run method of a slightly more elaborate actor with

the following behavior:

> *Emit an output O as soon as two inputs A and B have occurred. Reset this behavior each time the input R occurs.*

Future work in this area may involve extending the infrastructure to support various concurrency constructs, such as preemption, parallel execution, etc. It might also be interesting to explore new concurrency semantics similar to the threaded DE, but without the 'forced' serialization.

12.6 Composing DE with Other Domains

One of the major concepts in Ptolemy II is modeling heterogeneous systems through the use of

```
public void run() {
    try {
        while (true) {
            // In initial state..
            waitForNewInputs();
            if (R.hasToken(0)) {
                // Resetting..
                continue;
            }
            if (A.hasToken(0)) {
                // Seen A..
                IOPort[] ports = {B,R};
                waitForNewInputs(ports);
                if (!R.hasToken(0)) {
                    // Seen A then B..
                    O.broadcast(new DoubleToken(1.0));
                    IOPort[] ports2 = {R};
                    waitForNewInputs(ports2);
                } else {
                    // Resetting
                    continue;
                }
            } else if (B.hasToken(0)) {
                // Seen B..
                IOPort[] ports = {A,R};
                waitForNewInputs(ports);
                if (!R.hasToken(0)) {
                // Seen B then A..
                O.broadcast(new DoubleToken(1.0));
                IOPort[] ports2 = {R};
                waitForNewInputs(ports2);
            } else {
                // Resetting
                continue;
            }
        } // while (true)
    } catch (IllegalActionException e) {
        getManager().notifyListenersOfException(e);
    }
}
```

FIGURE 12.9. The run() method of the ABRO actor.

hierarchical heterogeneity. Actors on the same level of hierarchy obey the same set of semantics rules. Inside some of these actors may be another domain with a different model of computation. This mechanism is supported through the use of opaque composite actors. An example is shown in figure 12.10. The outermost domain is DE and it contains seven actors, two of them are opaque and composite. The opaque composite actors contain subsystems, which in this case are in the DE and CT domains.

12.6.1 DE inside Another Domain

The DE subsystem completes one iteration whenever the opaque composite actor is fired by the outer domain. One of the complications in mixing domains is in the synchronization of time. Denote the current time of the DE subsystem by t_{inner} and the current time of the outer domain by t_{outer}. An iteration of the DE subsystem is similar to an iteration of a top-level DE model, except that prior to the iteration tokens are transferred from the ports of the opaque composite actors into the ports of the contained DE subsystem, and after the end of the iteration, the director requesting a refire at the smallest time stamp in the event queue of the DE subsystem.

The first of these is done in the transferInputs() method of the DE director. This method is extended from its default implementation in the Director class. The implementation in the DEDirector class advances the current time of the DE subsystem to the current time of the outer domain, then calls super.transferInputs(). It is done in order to correctly associate tokens seen at the input ports of the opaque composite actor, if any, with events at the current time of the outer domain, t_{outer}, and put these events into the global event queue. This mechanism is, in fact, how the DE subsystem synchronize its current time, t_{inner}, with the current time of the outer domain, t_{outer}.(Recall that the DE director

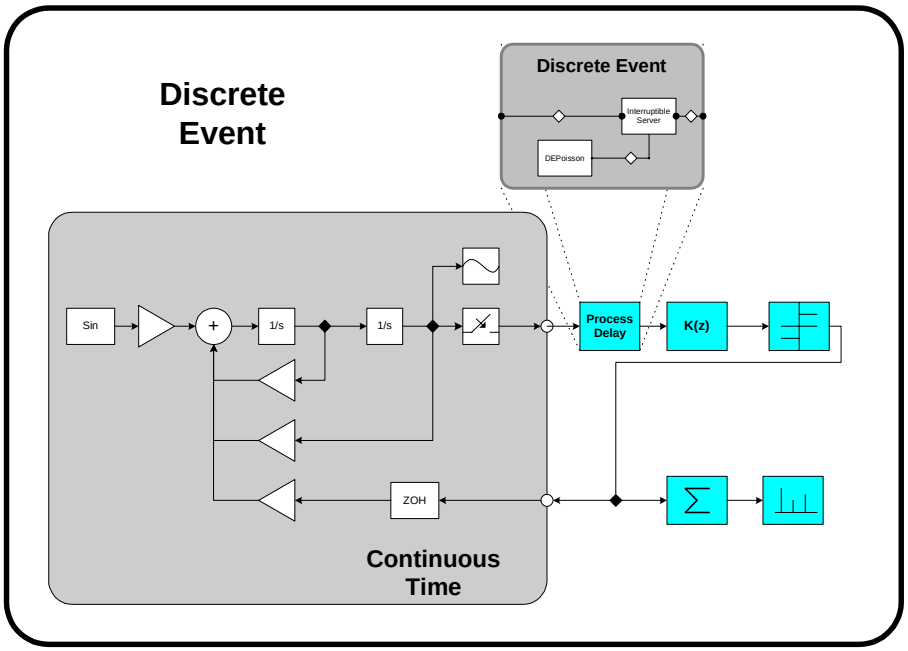


FIGURE 12.10. An example of heterogeneous and hierarchical composition. The CT subsystem and DE subsystem are inside an outermost DE system. This example is developed by Jie Liu [51].

advances time by looking at the smallest time stamp in the event queue of the DE subsystem). Specifically, before the advancement of the current time of the DE subsystem t_{inner} is less than or equal to the t_{outer}, and after the advancement t_{inner} is equal to the t_{outer}.

Requesting a refiring is done in the postfire() method of the DE director by calling the fireAt() method of the executive director. Its purpose is to ensure that events in the DE subsystem are processed on time with respect to the current time of the outer domain, t_{outer}.

Note that if the DE subsystem is fired due to the outer domain processing a refire request, then there may not be any tokens in the input port of the opaque composite actor at the beginning of the DE subsystem iteration. In that case, no new events with time stamps equal to t_{outer} will be put into the global event queue. Interestingly, in this case, the time synchronization will still work because t_{inner} will be advanced to the smallest time stamp in the global event queue which, in turn, has to be equal t_{outer} because we always request a refire according to that time stamp.

12.6.2 Another Domain inside DE

Due to its nature, the opaque composite actor is opaque and therefore, as far as the DE Director is concerned, behaves exactly like a domain polymorphic actor. Recall that domain polymorphic actors are treated as functions with zero delay in computation time. To produce events in the future, domain polymorphic actors request a refire from the DE director and then produce the events when it is refired.

13

SDF Domain

Author: Steve Neuendorffer

13.1 Overview

The synchronous dataflow (SDF) domain is useful for modeling simple dataflow systems without control, such as signal processing systems. Under the SDF domain, the execution order of actors is statically determined prior to execution. This results in execution with minimal overhead, as well as bounded memory usage and a guarantee that deadlock will never occur. Unfortunately, static schedules cannot be computed for all dataflow graphs. Dataflow graphs that cannot be statically scheduled should be executed using the process networks (PN) domain instead.

13.1.1 Properties

SDF is an untimed model of computation. All actors under SDF consume input tokens, perform their computation and produce outputs in one atomic operation. If an SDF model is embedded within a timed model, then the SDF model will behave as a zero-delay actor.

In addition, SDF is a statically scheduled domain. The firing of a composite actor corresponds to a single iteration of the contained model. An SDF iteration consists of one execution of the precalculated SDF schedule. The schedule is calculated so that the number of tokens on each relation is the same at the end of an iteration as at the beginning. Thus, an infinite number of iterations can be executed, without deadlock or infinite accumulation of tokens on each relation.

Execution in SDF is extremely efficient because of the scheduled execution. However, in order to execute so efficiently, some extra information must be given to the scheduler. Most importantly, the data rates on each port must be declared prior to execution. The data rate represents the number of tokens produced or consumed on a port during every firing. The data rates must be determined prior to execution and must be constant throughout execution. In addition, explicit delays must be added to feedback loops to prevent deadlock.

13.1.2 Scheduling

The first step in constructing the schedule is to solve the balance equations [47]. These equations determine the number of times each actor will fire during an iteration. For example, consider the system in figure 13.1. The scheduler will create the following system of equations, where *ProductionRate* and *ConsumptionRate* are declared properties of each port, and *Firings* is a property of each actor that will be solved for:

$$Firings(\mathrm{A}) \times ProductionRate(\mathrm{A1}) = Firings(\mathrm{B}) \times ConsumptionRate(\mathrm{B1}) \quad (39)$$

$$Firings(\mathrm{A}) \times ProductionRate(\mathrm{A2}) = Firings(\mathrm{C}) \times ConsumptionRate(\mathrm{C1}) \quad (40)$$

$$Firings(\mathrm{C}) \times ProductionRate(\mathrm{C2}) = Firings(\mathrm{B}) \times ConsumptionRate(\mathrm{B2}) \quad (41)$$

These equations express constraints that the number of tokens created on a relation during an iteration is equal to the number of tokens consumed. These equations usually have an infinite number of linearly dependent solutions, and the least positive integer solution for *Firings* is chosen as the *firing vector*[1].

In some cases, a non-zero solution to the balance equations does not exist. Such models are said to be *inconsistent*, and are illegal to execute under SDF. Inconsistent graphs inevitably result in either deadlock or unbounded memory usage for any schedule. As such, inconsistent graphs are usually bugs in the design of a model. However, inconsistent graphs can still be executed using the PN domain, if the behavior is truly necessary. Examples of consistent and inconsistent graphs are shown in figure 13.2.

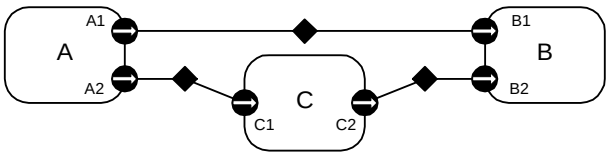


FIGURE 13.1. An example model.

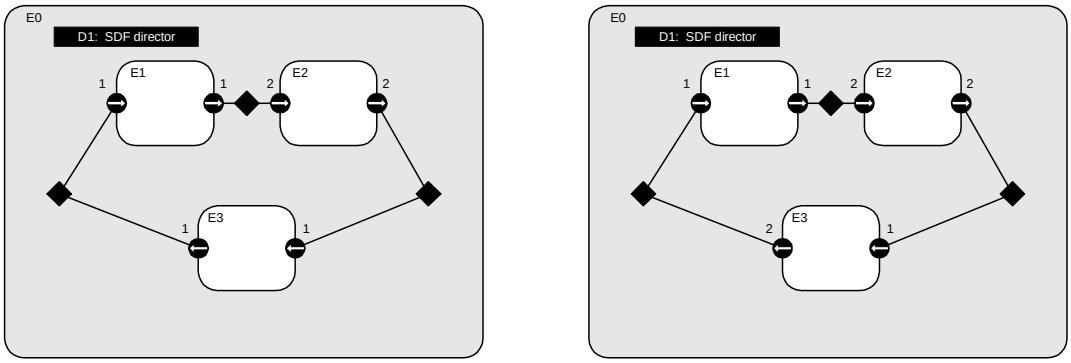


FIGURE 13.2. The model on the left is consistent. The model on the right is inconsistent.

1. The firing vector is also known as the repetitions vector.

The second step in constructing an SDF schedule is dataflow analysis. Dataflow analysis orders the firing of actors, based on the relations between them. Since each relation represents the flow of data, the actor producing data must fire before the consuming actor. Converting these data dependencies to a sequential list of properly scheduled actors is equivalent to topologically sorting the SDF graph, if the graph is acyclic. Dataflow graphs with cycles cause somewhat of a problem, since such graphs cannot be topologically sorted. In order to determine which actor of the loop to fire first, a delay must be explicitly inserted somewhere in the cycle. This delay is represented by an initial token on some relation in the cycle. The presence of the delay allows the scheduler to break the dependency cycle and determine which actor in the cycle to fire first. Cyclic graphs not properly annotated with delays cannot be executed under SDF. An example of a cyclic graph properly annotated with a delay is shown in figure 13.3.

13.2 Kernel

The SDF kernel package implements the SDF model of computation. The structure of the classes in this package is shown in figure 13.4.

13.2.1 SDF Director

The SDFDirector class extends the StaticSchedulingDirector class. When an SDF director is created, it is automatically associated with an instance of the default scheduler class, SDFScheduler. This scheduler is intended to be relatively fast and valid, but not optimal in all situations. As such, future development will likely result in a wide range of schedulers with difference performance goals and trade-offs. The SDF scheduler does not currently restrict the schedulers that may be used with it.

The director has a single parameter, *iterations*, which determines a limit on the number of times the director wishes to be fired. After the director has been fired the given number of times, it will always return false in its postfire() method, indicating that it does not wish to be fired again. This parameter must contain a non-negative integer value. The default value is an IntToken with value 0, indicating that there is no preset limit for the number of times the director will fire. Users will likely specify a non-zero value for the number of iterations for the toplevel composite actor. When used this

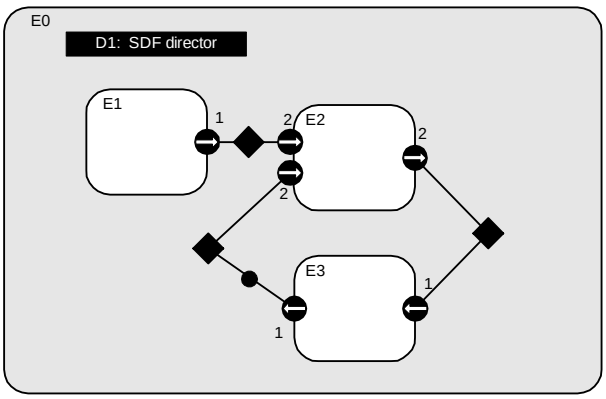


FIGURE 13.3. A consistent cyclic graph, properly annotated with delays. A one token delay is represented by a black circle.

way, this parameter acts similarly to the Time-to-Stop parameter in Ptolemy Classic.

The newReceiver() method in SDF directors is overloaded to return instances of the SDFReceiver class. This receiver contains optimized method for reading and writing blocks of tokens. For more information about SDF receivers and the extra methods that they support, see section 13.2.3.

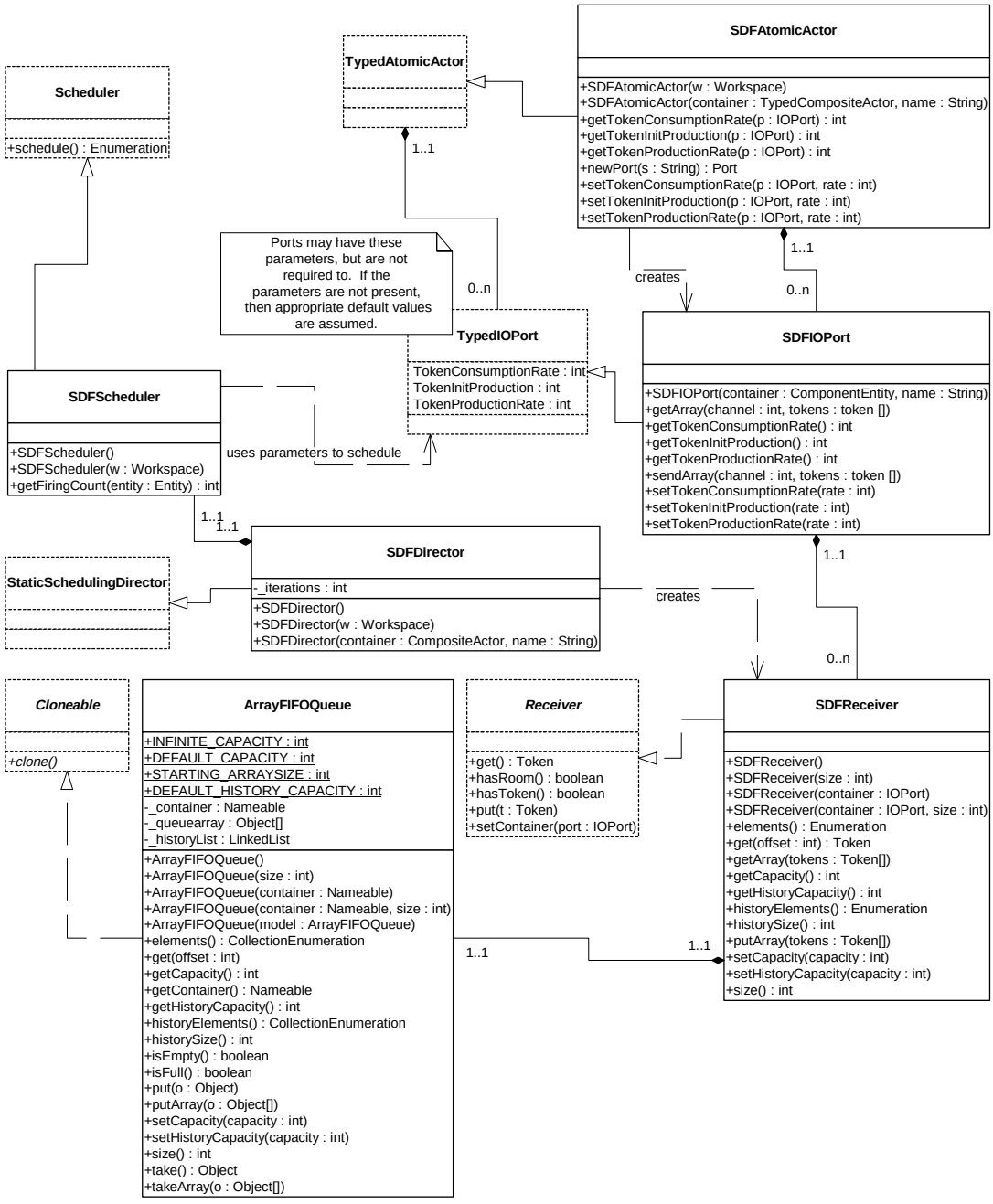


FIGURE 13.4. The static structure of the SDF kernel classes.

13.2.2 Scheduling

The basic SDFScheduler derives directly from the Scheduler class. This scheduler provides unlooped, sequential schedules suitable for use on a single processor. No attempt is made to optimize the schedule by minimizing data buffer sizes, minimizing the size of the schedule, or detecting parallelism to allow execution on multiple processors. We anticipate that as these schedulers become interesting, they will be added.

The scheduling algorithm is based on the simple multirate algorithms in [47]. Currently, only single processor schedules are supported. The multirate scheduling algorithm relies on the actors in the system declaring the data rates in each port. If the rates are not declared, then the scheduler assumes that the actor is *homogeneous*, meaning that it consumes exactly one token from each input port and produces exactly one token on each output port.

Data rates on ports are specified using three parameters: *tokenConsumptionRate*, *tokenProductionRate*, and *tokenInitProduction*. The production parameters are valid only for output ports, while the consumption parameter is valid only for input ports. If a parameter exists that is not valid for a given port, then the value of the parameter must be zero, or the scheduler will throw an exception. If a valid parameter is not specified when the scheduler runs, then appropriate values of the parameters will be assumed[1], however the parameters are not then created.

In Ptolemy classic, hierarchical SDF models were generally flattened prior to scheduling. This technique allowed the most efficient schedule to be constructed for a model, and avoided certain composability problems. In Ptolemy II, this algorithm can be replicated by using transparent composite actors to define the hierarchy. However, Ptolemy II also supports a stronger version of hierarchy, in the form of opaque composite actors. In this case, the scheduler needs to do a little more work. Prior to scheduling a graph containing opaque composite actors, the scheduler queries each contained opaque composite actor that may contain another scheduler and calls schedule() on that scheduler. The SDF scheduler also creates and sets the appropriate rate parameters on any ports it encounters that are contained within its director's container.

Disconnected graphs. SDF graphs should generally be strongly connected. If an SDF graph is not strongly connected, then there is some concurrency between the disconnected parts that is not captured by the SDF rate parameters. In such cases, another model of computation (such as process networks) should be used to explicitly specify the concurrency. As such, the current SDF scheduler disallows disconnected graphs, and will throw an exception if you attempt to schedule such a graph. However, sometimes it is useful to avoid introducing another model of computation, so it is likely that a future scheduler will allow disconnected graphs with a default notion of concurrency.

Multiports. Notice that it is impossible to set a rate parameter on individual channels of a port. This is intentional, and all the channels of an actor are assumed to have the same rate. For example, when the AddSubtract actor fires under SDF, it will consume exactly one token from each channel of its input *plus* port, consume one token from each channel of its *minus* port, and produce one token on each channel of its output port. Notice that although the domain-polymorphic adder is written to be more general than this (it will consume *up to* one token on each channel of the input port), the SDF scheduler will ensure that there is always at least one token on each input port before the actor fires.

1. The assumed values correspond to a homogeneous, zero-delay actor. Input ports are assumed to have a consumption rate of one, output ports are assumed to have a production rate of one, and no tokens are produced during initialization.

Dangling ports. Ports should, in general, be connected under the SDF domain. A regular port that is not connected cannot be fulfilled by the SDF scheduler and the scheduler will always throw an exception. The SDF scheduler also detects multiports that are not connected to anything (and thus have zero width). Such ports are interpreted to not actually exist, and are legal under SDF. The scheduler will ignore the presence of a disconnected multiport.

13.2.3 SDF ports and receivers

Unlike most domains, multirate SDF systems tend to produce and consume large blocks of tokens during each firing. Since there can be significant overhead in data transport for these large blocks, SDF ports and receivers have optimized methods for sending and receiving a block of tokens *en masse*.

The SDFReceiver class implements the Receiver interface. Instead of using the FIFOQueue class to store data, which is based on a linked list structure, SDF receivers use the ArrayFIFOQueue class, which is based on a circular buffer. This choice is much more appropriate for SDF, since the size of the buffer is bounded, and can be determined statically[1]. Circular buffers also have less memory and object allocation overhead for a queue of a given size.

In addition to the normal receiver methods, the SDFReceiver provides sendArray() and getArray() methods. These two methods operate on arrays of tokens identically to the ways the send and get methods operate on individual tokens. Calling the sendArray() method on an array of tokens is equivalent to calling the send() method on each element of the array, only faster[2].

The SDFIOPort class extends the TypedIOPort class. It adds two methods, sendArray() and getArray(). If the remote port contains an SDF receiver, then the receiver's sendArray() method will be used instead of the send() method. The getArray() method operates similarly. Currently, SDF ports do not support block operations on the history tokens of the ports.

13.2.4 ArrayFIFOQueue

The ArrayFIFOQueue class implements a first in, first out (FIFO) queue by means of a circular array buffer. Functionally it is very similar to the FIFOQueue class. It provides a token history and an adjustable, possibly infinite, bound on the number token it contains.

If the bound on the size is finite, then the array is exactly the size of the bound. In other words, the queue is full when the array becomes full. However, if the bound is infinite, then such an array cannot be created! In this case, the circular buffer is given a small starting size, but allowed to grow. Whenever the circular buffer fills up, it is copied into a new buffer that is twice the original size.

13.2.5 SDFAtomicActor

The SDFAtomicActor class extends the TypedAtomicActor class. It exists mainly for convenience when creating actors in the SDF domain. It overrides the newPort() method to create SDF ports. It also

1. Although the buffer sizes can be statically determined, the current mechanism for creating receivers does not easily support it. The SDF domain currently relies on the buffer expanding algorithm that the ArrayFIFOQueue uses to implement circular buffers of unbounded size. Although there is some overhead during the first iteration, the overhead is minimal during subsequent iterations (since the buffer is guaranteed never to grow larger).
2. The array operations in ArrayFIFOQueue use the java.lang.system.arraycopy method. This method is capable of removing certain checks required by the Java language. On most Java implementations, this is significantly faster than a hand coded loop for large arrays. However, depending on the Java implementation it could actually be slower for small arrays. The cost is usually negligible, but can be avoided when the size of the array is small and known when the actor is written.

provides methods for setting and accessing the rate parameters on the actor’s ports.

14

CSP Domain

Author: Neil Smyth
Contributor: John S. Davis II

14.1 Introduction

The communicating sequential processes (CSP) domain in Ptolemy II models a system as a network of sequential processes that communicate by passing messages synchronously through channels. If a process is ready to send a message, it blocks until the receiving process is ready to accept the message. Similarly if a process is ready to accept a message, it blocks until the sending process is ready to send the message. This model of computation is non-deterministic as a process can be blocked waiting to send or receive on any number of channels. It is also highly concurrent.

The CSP domain is based on the model of computation (MoC) first proposed by Hoare [36][37] in 1978. In this MoC, a system is modeled as a network of processes communicate solely by passing messages through unidirectional channels. The transfer of messages between processes is via *rendezvous*, which means both the sending and receiving of messages from a channel are *blocking*: i.e. the sending or receiving process stalls until the message is transferred. Some of the notation used here is borrowed from Gregory Andrews' book on concurrent programming [4], which refers to rendezvous-based message passing as *synchronous message passing*.

Applications for the CSP domain include resource management and high level system modeling early in the design cycle. Resource management is often required when modeling embedded systems, and to further support this, a notion of time has been added to the model of computation used in the domain. This differentiates our CSP model from those more commonly encountered, which do not typically have any notion of time, although several versions of timed CSP have been proposed [34]. It might thus be more accurate to refer to the domain using our model of computation as the "Timed CSP" domain, but since the domain can be used with and without time, it is simply referred to as the CSP domain.

14.2 CSP Communication Semantics

At the core of CSP communication semantics are two fundamental ideas. First is the notion of atomic communication and second is the notion of nondeterministic choice. It is worth mentioning a related model of computation known as the calculus of communicating systems (CCS) that was independently developed by Robin Milner in 1980 [58]. The communication semantics of CSP are identical to those of CCS.

14.2.1 Atomic Communication: Rendezvous

Atomic communication is carried out via rendezvous and implies that the sending and receiving of a message occur simultaneously. During rendezvous both the sending and receiving processes block until the other side is ready to communicate; the act of sending and receiving are indistinguishable activities since one can not happen without the other. A real world analogy to rendezvous can be found in telephone communications (without answering machines). Both the caller and callee must be simultaneously present for a phone conversation to occur. Figure 14.1 shows the case where one process is ready to send before the other process is ready to receive. The communication of information in this way can be viewed as a distributed assignment statement.

The sending process places some data in the message that it wants to send. The receiving process assigns the data in the message to a local variable. Of course, the receiving process may decide to ignore the contents of the message and only concern itself with the fact that a message arrived.

14.2.2 Choice: Nondeterministic Rendezvous

Nondeterministic choice provides processes with the ability to randomly select between a set of possible atomic communications. We refer to this ability as nondeterministic rendezvous and herein lies much of the expressiveness of the CSP model of computation. The CSP domain implements nondeterministic rendezvous via *guarded communication statements*. A guarded communication statement has the form

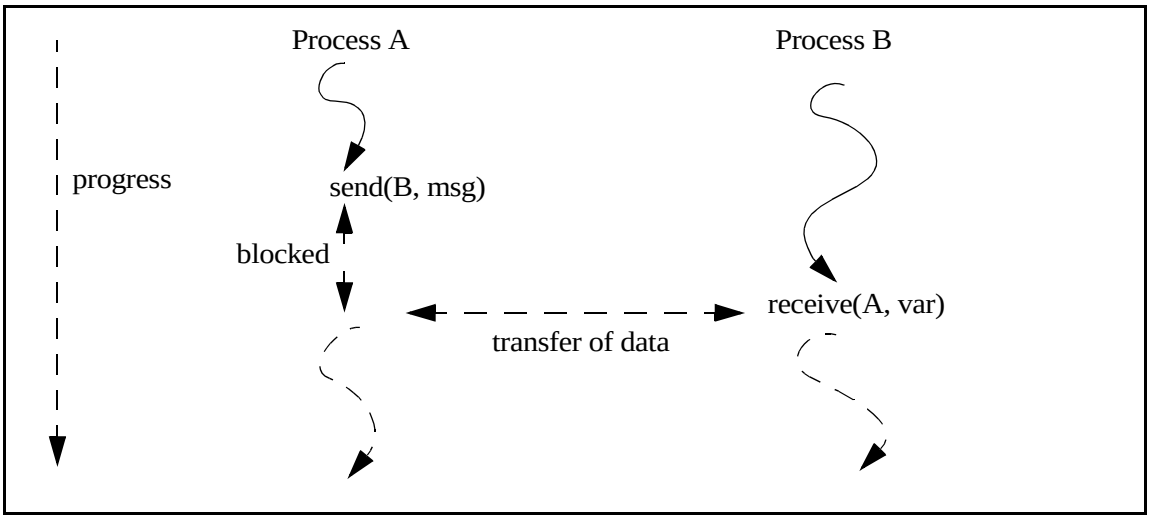


FIGURE 14.1. Illustrating how processes block waiting to rendezvous

```
guard; communication => statements;
```

The *guard* is only allowed to reference local variables, and its evaluation cannot change the state of the process. For example it is not allowed to assign to variables, only reference them. The *communication* must be a simple send or receive, i.e. another conditional communication statement cannot be placed here. *Statements* can contain any arbitrary sequence of statements, including more conditional communications.

If the guard is false, then the communication is not attempted and the statements are not executed. If the guard is true, then the communication is attempted, and if it succeeds, the following statements are executed. The guard may be omitted, in which case it is assumed to be true.

There are two conditional communication constructs built upon the guarded communication statements: **CIF** and **CDO**. These are analogous to the *if* and *while* statements in most programming languages. They should be read as “conditional if” and “conditional do”. Note that each guarded communication statement represents one *branch* of the CIF or CDO. The communication statement in each branch can be either a send or a receive, and they can be mixed freely.

CIF: The form of a CIF is

```
CIF {
     G1;C1 => S1;
[]
     G2;C2 => S2;
[]
     ...
}
```

For each branch in the CIF, the guard (*G1*, *G2*,...) is evaluated. If it is true (or absent, which implies true), then the associated communication statement is enabled. If one or more branch is enabled, then the entire construct blocks until one of the communications succeeds. If more than one branch is enabled, the choice of which enabled branch succeeds with its communication is made nondeterministically. Once the successful communication is carried out, the associated statements are executed and the process continues. If all of the guards are false, then the process continues executing statements after the end of the CIF.

It is important to note that, although this construct is analogous to the common *if* programming construct, its behavior is very different. In particular all guards of the branches are evaluated concurrently, and the choice of which one succeeds does not depend on its position in the construct. The notation “[]” is used to hint at the parallelism in the evaluation of the guards. In a common *if*, the branches are evaluated sequentially and the first branch that is evaluated to true is executed. The CIF construct also depends on the semantics of the communication between processes, and can thus stall the progress of the thread if none of the enabled branches is able to rendezvous.

CDO: The form of the CDO is

```
CDO {
     G1;C1 => S1;
[]
     G2;C2 => S2;
[]
     ...
}
```

The behavior of the CDO is similar to the CIF in that for each branch the guard is evaluated and the choice of which enabled communication to make is taken nondeterministically. However the CDO repeats the process of evaluating and executing the branches until *all* the guards return false. When this happens the process continues executing statements after the CDO construct.

An example use of a CDO is in a buffer process which can both accept and send messages, but has to be ready to do both at any stage. The code for this would look similar to that in figure 14.2. Note that in this case both guards can never be simultaneously false so this process will execute the CDO forever.

14.2.3 Deadlock

A deadlock situation is one in which none of the processes can make progress: they are all either blocked trying to rendezvous or they are delayed (see the next section). Thus two types of deadlock can be distinguished:

real deadlock - all active processes are blocked trying to communicate

time deadlock - all active processes are either blocked trying to communicate or are delayed, and at least one processes is delayed.

14.2.4 Time

In the CSP domain, *time* is centralized. That is, all processes in a model share the same time, referred to as the *current model time*. Each process can only choose to *delay* itself for some period relative to the current model time, or a process can wait for time deadlock to occur at the current model time. In both cases, a process is said to be *delayed*.

When a process delays itself for some length of time from the current model time, it is suspended until time has sufficiently advanced, at which stage it wakes up and continues. If the process delays itself for zero time, this will have no effect and the process will continue executing.

A process can also choose to delay its execution until the next occasion a time deadlock is reached. The process resumes at the same model time at which it delayed, and this is useful as a model can have several sequences of actions at the same model time. The next occasion time deadlock is reached, any

```
CDO {
    (room in buffer?); receive(input, beginningOfBuffer) => update pointer to beginning of buffer;
[]
    (messages in buffer?); send(output, endOfBuffer) => update pointer to end of buffer;
}
```

FIGURE 14.2. Example of how a CDO might be used in a buffer

processes delayed in this manner will continue, and time will not be advanced. An example of using time in this manner can be found in section 14.3.2.

Time may be *advanced* when all the processes are delayed or are blocked trying to rendezvous, and at least one process is delayed. If one or more processes are delaying until a time deadlock occurs, these processes are woken up and time is not advanced. Otherwise, the current model time is advanced just enough to wake up at least one process. Note that there is a semantic difference between a process delaying for zero time, which will have no effect, and a process delaying until the next occasion a time deadlock is reached.

Note also that time, as perceived by a single process, cannot change during its normal execution; only at rendezvous points or when the process delays can time change. A process can be aware of the centralized time, but it cannot influence the current model time except by delaying itself. The choice for modeling time was in part influenced by Pamela [27], a run time library that is used to model parallel programs.

14.2.5 Differences from Original CSP Model as Proposed by Hoare

The model of computation used by the CSP domain differs from the original CSP [36] model in two ways. First, a notion of time has been added. The original proposal had no notion of time, although there have been several proposals for timed CSP [34]. Second, as mentioned in section 14.2.2, it is possible to use both send and receive in guarded communication statements. The original model only allowed receives to appear in these statements, though Hoare subsequently extended their scope to allow both communication primitives [37].

One final thing to note is that in much of the CSP literature, send is denoted using a "!", pronounced "bang", and receive is denoted using a "?", pronounced "query". This syntax was what was used in the original CSP paper by Hoare. For example, the languages Occam [14] and Lotos [21] both follow this syntax. In the CSP domain in Ptolemy II we use *send* and *get*, the choice of which is influenced by the desire to maintain uniformity of syntax across domains in Ptolemy II that use message passing. This supports the heterogeneity principle in Ptolemy II which enables the construction and interoperability of executable models that are built under a variety of models of computation. Similarly, the notation used in the CSP domain for conditional communication constructs differs from that commonly found in the CSP literature.

14.3 Example CSP Applications

Several example applications have been developed which serve to illustrate the modeling capabilities of the CSP model of computation in Ptolemy II. Each demonstration incorporates several features of CSP and the general Ptolemy II framework. Below, four demonstrations have been selected that each emphasize particular semantic capabilities over others. The applications are described here, but not the code. See the directory $PTII/ptolemy/domains/csp/demo for the code.

The first demonstration, *dining philosophers*, serves as a natural example of core CSP communication semantics. This demonstration models nondeterministic resource contention, e.g., five philosophers randomly accessing chopstick resources. Nondeterministic rendezvous serves as a natural modeling tool for this example. The second example, *hardware bus contention*, models deterministic resource contention in the context of time. As will be shown, the determinacy of this demonstration constrains the natural nondeterminacy of the CSP semantics and results in difficulties. Fortunately these difficulties can be smoothly circumvented by the timing model that has been integrated into the

CSP domain. The third demonstration, *sieve of Eratosthenes*, serves to demonstrate the mutability that is possible in CSP models. In this demonstration, the topology of the model changes during execution. The final demonstration, *M/M/1 queue*, features the pause/resume mechanism of Ptolemy II that can be used to control the progression of a model's execution in the CSP domain.

14.3.1 Dining Philosophers

Nondeterministic Resource Contention. This implementation of the dining philosophers problem illustrates both time and conditional communication in the CSP domain. Five philosophers are seated at a table with a large bowl of food in the middle. Between each pair of philosophers is one chopstick, and to eat, a philosopher needs both the chopsticks beside him. Each philosopher spends his life in the following cycle: thinks for a while, gets hungry, picks up one of the chopsticks beside him, then the other, eats for a while and puts the chopsticks down on the table again. If a philosopher tries to grab a chopstick but it is already being used by another philosopher, then the philosopher waits until that chopstick becomes available. This implies that no neighboring philosophers can eat at the same time and at most two philosophers can eat at a time.

The Dining Philosophers problem was first dreamt up by Edsger W. Dijkstra in 1965. It is a classic concurrent programming problem that illustrates the two basic properties of concurrent programming:

Liveness. How can we design the program to avoid deadlock, where none of the philosophers can make progress because each is waiting for someone else to do something?

Fairness. How can we design the program to avoid starvation, where one of the philosophers could make progress but does not because others always go first?

This implementation uses an algorithm that lets each philosopher randomly chose which chopstick to pick up first (via a CDO), and all philosophers eat and think at the same rates. Each philosopher and each chopstick is represented by a separate process. Each chopstick has to be ready to be used by either philosopher beside it at any time, hence the use of a CDO. After it is grabbed, it blocks waiting for a message from the philosopher that is using it. After a philosopher grabs both the chopsticks next to him, he eats for a random time. This is represented by calling delay() with the random interval to eat for. The same approach is used when a philosopher is thinking. Note that because messages are passed by rendezvous, the blocking of a philosopher when it cannot obtain a chopstick is obtained for free.

This algorithm is fair, as any time a chopstick is not being used, and both philosophers try to use it, they both have an equal chance of succeeding. However this algorithm does not guarantee the absence of deadlock, and if it is let run long enough this will eventually occur. The probability that deadlock

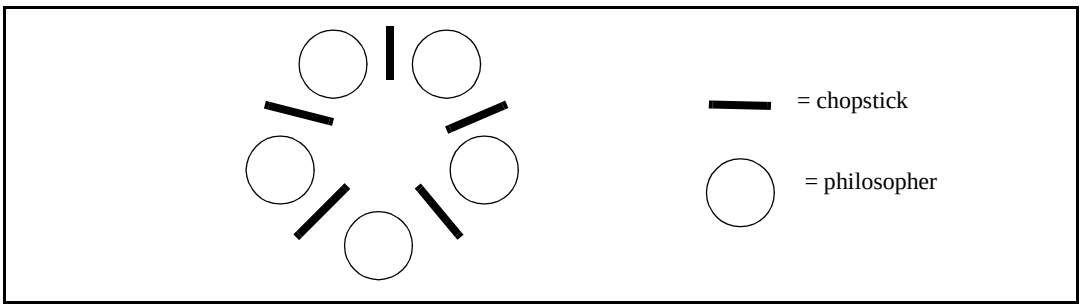


FIGURE 14.3. Illustration of the dining philosophers problem.

occurs sooner increases as the thinking times are decreased relative to the eating times.

14.3.2 Hardware Bus Contention

Deterministic Resource Contention. This demonstration consists of a controller, *N* processors and a memory block. At randomly selected points in time, each processor requests permission from the controller to access the memory block. The processors each have priorities associated with them and in cases where there is a simultaneous memory access request, the controller grants permission to the processor with the highest priority. Due to the atomic nature of rendezvous, it is impossible for the controller to check priorities of incoming requests at the same time that requests are occurring. To overcome this difficulty, an alarm is employed. The alarm is started by the controller immediately following the first request for memory access at a given instant in time. It is awakened when a delay block occurs to indicate to the controller that no more memory requests will occur at the given point in time. Hence, the alarm uses CSP's notion of delay blocking to make deterministic an inherently nondeterministic activity.

14.3.3 Sieve of Eratosthenes

Dynamic Topology. This example implements the *sieve of Eratosthenes*. This is an algorithm for generating a list of prime numbers, illustrated in figure 14.5. It originally consists of a source generating integers, and one sieve filtering out all multiples of two. When the end sieve sees a number that it cannot filter, it creates a new sieve to filter out all multiples of that number. Thus after the sieve filtering out multiples of two sees the number three, it creates a new sieve that filters out multiples of three. This then continues with the three sieve eventually creating a sieve to filter out all multiples of five, and so on. Thus after a while there will be a chain of sieves each filtering out a different prime number. If any number passes through all the sieves and reaches the end with no sieve waiting, it must be another prime and so a new sieve is created for it.

This demo is an example of how changes to the topology can be made in the CSP domain. Each topology change here involves creating a new CSPSieve actor and connecting it to the end of the chain of sieves.

14.3.4 An M/M/1 Queue

Pause/Resume. The example in figure 14.6 illustrates a simple M/M/1 queue. It has three actors, one representing the arrival of customers, one for the queue holding customers that have arrived and have

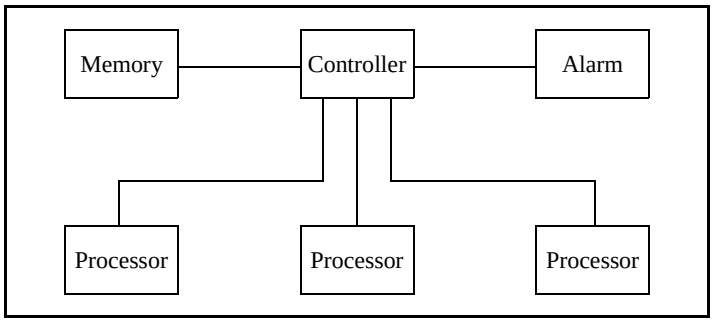


FIGURE 14.4. Processors contending for memory access

not yet been served, and the third representing the server. Both the inter-arrival times of customers and the service times at the server are exponentially distributed, which of course is what makes this an M/M/1 queue.

This demo makes use of basic rendezvous, conditional rendezvous and time. By varying the rates for the customer arrivals and service times, and varying the length of the buffer, you can see various trade-offs. For example if the buffer length is too short, customers may arrive that cannot be stored and so are missed. Similarly if the service rate is faster than the customer arrival rate, then the server could spend a lot of time idle.

Another example demonstrates how pausing and resumption works. The setup is exactly the same

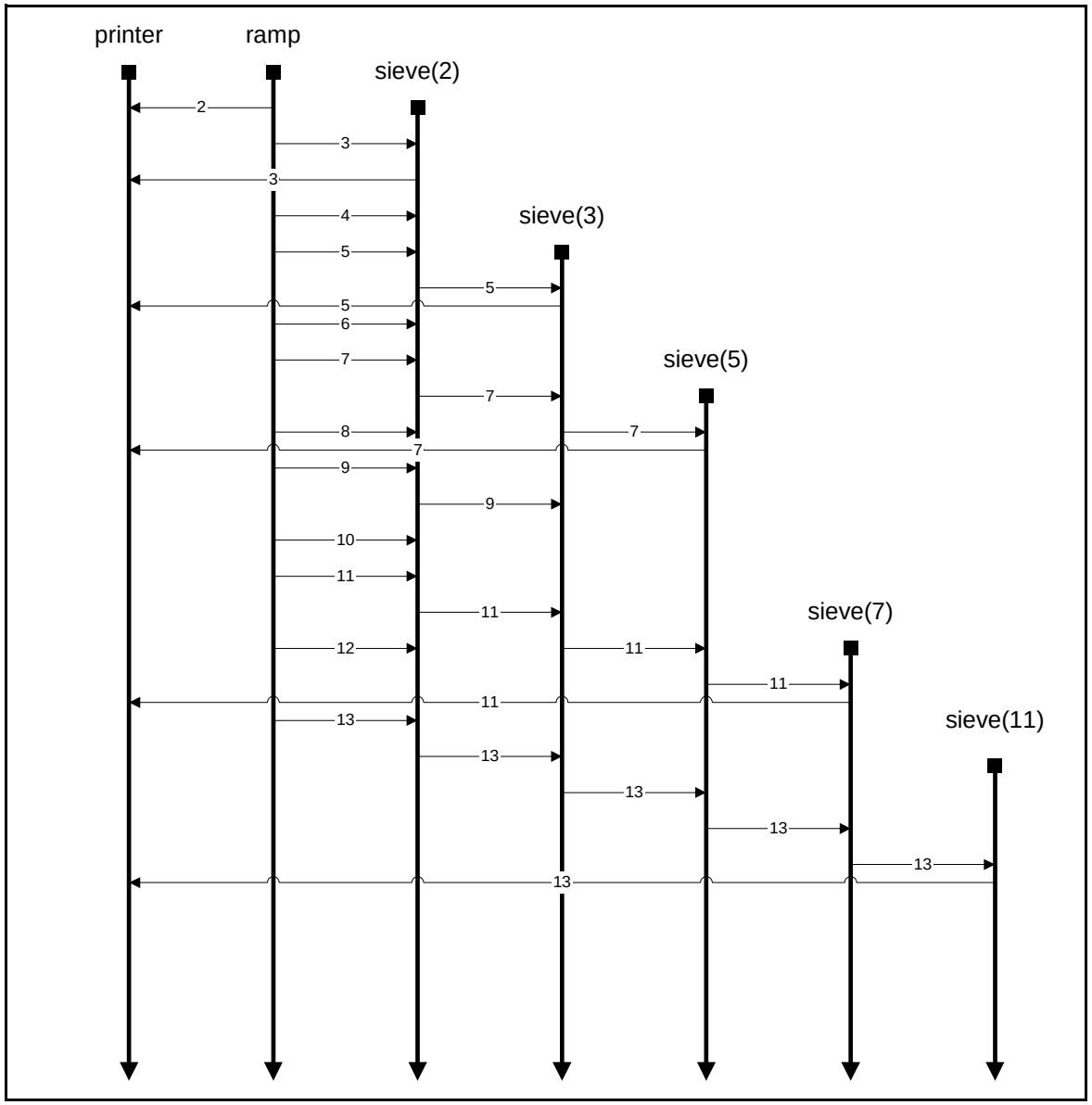


FIGURE 14.5. Illustration of *Sieve of Eratosthenes* for obtaining first six primes.

as in the M/M/1 demo, except that the thread executing the model calls pause() on the director as soon as the model starts executing. It then waits two seconds, as arbitrary choice, and then calls resume(). The purpose of this demo is to show that the pausing and resuming of a model does not affect the model results, only its rate of progress. The ability to pause and resume a model is primarily intended for the user interface.

14.4 Building CSP Applications

For a model to have CSP semantics, it must have a CSPDirector controlling it. This ensures that the receivers in the ports are CSPReceivers, so that all communication of messages between processes is via rendezvous. Note that each *actor* in the CompositeActor under the control of the CSPDirector represents a separate *process* in the model.

14.4.1 Rendezvous

Since the ports contain CSPReceivers, the basic communication statements send() and get() will have rendezvous semantics. Thus the fact that a rendezvous is occurring on every communication is transparent to the actor code.

14.4.2 Conditional Communication Constructs

In order to use the conditional communication constructs, an actor must be derived from CSPActor. There are three steps involved:

1) Create a ConditionalReceive or ConditionalSend branch for each guarded communication statement, depending on the communication. Pass each branch a unique integer identifier, starting from zero, when creating it. The identifiers only need to be unique within the scope of that CDO or CIF.

2) Pass the branches to the chooseBranch() method in CSPActor. This method evaluates the guards, and decides which branch gets to rendezvous, performs the rendezvous and returns the identification number of the branch that succeeded. If all of the guards were false, -1 is returned.

3) Execute the statements for the guarded communication that succeeded.

A sample template for executing a CDO is shown in figure 14.7. The code for the buffer described in figure 14.7 is shown in figure 14.8. In creating the ConditionalSend and ConditionalReceive branches, the first argument represents the guard. The second and third arguments represent the port and channel to send or receive the message on. The fourth argument is the identifier assigned to the branch. The choice of placing the guard in the constructor was made to keep the syntax of using guarded communication statements to the minimum, and to have the branch classes resemble the

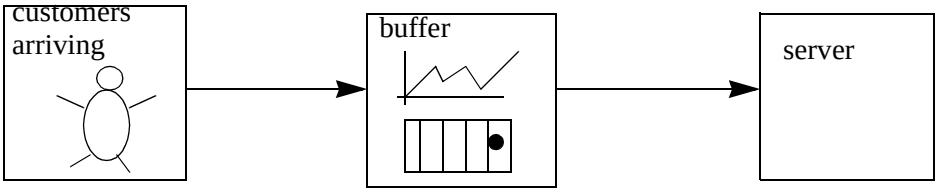


FIGURE 14.6. Actors involved in M/M/1 demo.

guarded communication statements they represent as closely as possible. This can give rise to the case where the Token specified in a ConditionalSend branch may not yet exist, but this has no effect because once the guard is false, the token in a ConditionalSend is never referenced.

The other option considered was to wrap the creation of each branch as follows:

```
if (guard) {
   // create branch and place in branches array
} else {
   // branches array entry for this branch is null
}
```

However this leads to longer actor code, and what is happening is not as syntactically obvious.

The code for using a CIF is similar to that in figure 14.7 except that the surrounding while loop is omitted and the case when the identifier returned is -1 does nothing. At some stage the steps involved in using a CIF or a CDO may be automated using a pre-parser, but for now the user must follow the approach described above.

It is worth pointing out that if most channels in a model are buffered, it may be worthwhile considering implementing the model in the PN domain which implicitly has an unbounded buffer on every channel. Also, if modeling time is the principal concern, the model builder should consider using the DE domain.

14.4.3 Time

If a process wishes to use time, the actor representing it must derive from CSPActor. As explained in section 14.2.4, each process in the CSP domain is able to delay itself, either for some period from the current model time or until the next occasion time deadlock is reached at the current model time.

```
boolean continueCDO = true;
while (continueCDO) {
   // step 1:
   ConditionalBranch[] branches = new ConditionalBranch[#branchesRequired];
   // Create a ConditionalReceive or a ConditionalSend for each branch
   // e.g. branches[0] = new ConditionalReceive((guard), input, 0, 0);

   // step 2:
   int result = chooseBranch(branches);

   // step 3:
   if (result == 0) {
      // execute statements associated with first branch
   } else if (result == 1) {
      // execute statements associated with second branch.
   } else if ... // continue for each branch ID

   } else if (result == -1) {
      // all guards were false so exit CDO.
      continueCDO = false;
   } else {
      // error
   }
}
```

FIGURE 14.7. Template for executing a CDO construct.

The two methods to call are delay() and waitForDeadlock(). Recall that if a process delays itself for zero time from the current time, the process will continue immediately. Thus delay(0.0) is not equivalent to waitForDeadlock()

If no processes are delayed, it is also possible to set the model time by calling the method setCurrentTime() on the director. However, this method can only be called when no processes are delayed, because the state of the model may be rendered meaningless if the model time is advanced to a time beyond the earliest delayed process. This method is present primarily for composing CSP with other domains.

As mentioned in section 14.2.4, as far as each process is concerned, time can only increase while it is blocked waiting to rendezvous or when delaying. A process can be aware of the current model time, but it should only ever affect the model time by delaying its execution, thus forcing time to advance. The method setCurrentTime() should never be called from a process.

By default every model in the CSP domain is timed. To use CSP without a notion of time, do not use the delay() method. The infrastructure supporting time does not affect the model execution if the delay() method is not used.

14.5 The CSP Software Architecture

14.5.1 Class Structure

In a CSP model, the director is an instance of *CSPDirector.* Since the model is controlled by a CSPDirector, all the receivers in the ports are *CSPReceivers*. The combination of the CSPDirector and

```
boolean guard = false;
boolean continueCDO = true;
ConditionalBranch[] branches = new ConditionalBranch[2];
while (continueCDO) {
   // step 1
   guard = (_size < depth);
   branches[0] = new ConditionalReceive(guard, input, 0, 0);
   guard = (_size > 0);
   branches[1] = new ConditionalSend(guard, output, 0, 1, _buffer[_readFrom]);

   // step 2
   int successfulBranch = chooseBranch(branches);

   // step 3
   if (successfulBranch == 0) {
      _size++;
      _buffer[_writeTo] = branches[0].getToken();
      _writeTo = ++_writeTo % depth;
   } else if (successfulBranch == 1) {
      _size--;
      _readFrom = ++_readFrom % depth;
   } else if (successfulBranch == -1) {
      // all guards false so exit CDO
      // Note this cannot happen in this case
      continueCDO = false;
   } else {
      throw new TerminateProcessException(getName() + ": " +
            "branch id returned during execution of CDO.");
   }
}
```

FIGURE 14.8. Code used to implement the buffer process described in figure 14.7.

CSPReceivers in the ports gives a model CSP semantics. The CSP domain associates each channel with exactly one receiver, located at the receiving end of the channel. Thus any process that sends or receives to any channel will rendezvous at a CSPReceiver. Figure 14.9 shows the static structure diagram of the five main classes in the CSP kernel, and a few of their associations. These are the classes that provide all the infrastructure needed for a CSP model.

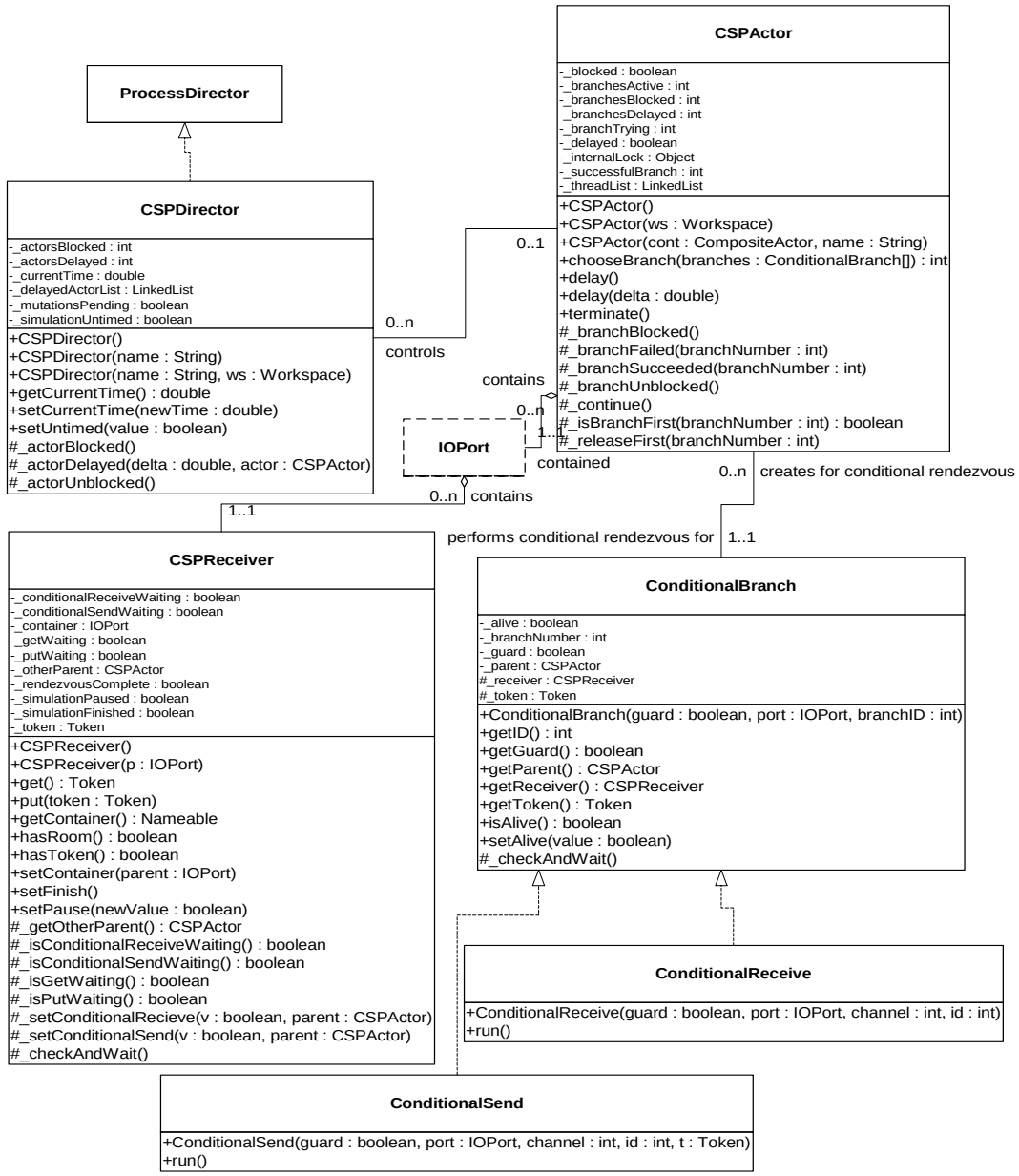


FIGURE 14.9. Static structure diagram for classes in the CSP kernel.

CSPDirector: This gives a model CSP semantics. It takes care of starting all the processes and controls/responds to both real and time deadlocks. It also maintains and advances the model time when necessary.

CSPReceiver: This ensures that communication of messages between processes is via rendezvous.

CSPActor: This adds the notion of time and the ability to perform conditional communication.

ConditionalReceive, ConditionalSend: This is used to construct the guarded communication statements necessary for the conditional communication constructs.

14.5.2 Starting the model

The director creates a thread for each actor under its control in its initialize() method. It also invokes the initialize() method on each actor at this time. The director starts the threads in its prefire() method, and detects and responds to deadlocks in its fire() method. The thread for each actor is an instance of ProcessThread, which invokes the prefire(), fire() and postfire() methods for the actor until it finishes or is terminated. It then invokes the wrapup() method and the thread dies.

Figure 14.11 shows the code executed by the ProcessThread class. Note that it makes no assumption about the actor it is executing, so it can execute any domain-polymorphic actor as well as CSP domain-specific actors. In fact, any other domain actor that does not rely on the specifics of its parent domain can be executed in the CSP domain by the ProcessThread.

14.5.3 Detecting deadlocks:

For deadlock detection, the director maintains three counts:

- the number of *active* processes which are threads that have started but have not yet finished
- the number of *blocked* processes which is the number of processes that are blocked waiting to rendezvous, and
- the number of *delayed* processes, which is the number of processes waiting for time to advance plus the number of processes waiting for time deadlock to occur at the current model time.

```
director.initialize() =>
   create a thread for each actor
   update count of active processes with the director
   call initialize() on each actor

director.prefire() => start the process threads =>
   calls actor.prefire()
   calls actor.fire()
   calls actor.postfire()
   repeat.

director.fire() => handle deadlocks until a real deadlock occurs.

director.postfire() =>
   return a boolean indicating if the execution of the model should continue for another iteration

director.wrapup() => terminate all the processes =>
   calls actor.wrapup()
   decrease the count of active processes with the director
```

FIGURE 14.10. Sequence of steps involved in setting up and controlling the model.

When the number of blocked processes equals the number of active processes, then real deadlock has occurred and the fire method of the director returns. When the number of blocked plus the number of delayed processes equals the number of active processes, and at least one process is delayed, then time deadlock has occurred. If at least one process is delayed waiting for time deadlock to occur at the current model time, then the director wakes up all such processes and does not advance time. Otherwise the director looks at its list of processes waiting for time to advance, chooses the earliest one and advances time sufficiently to wake it up. It also wakes up any other processes due to be awakened at the new time. The director checks for deadlock each occasion a process blocks, delays or dies.

For the director to work correctly, these three counts need to be accurate at all stages of the model execution, so when they are updated becomes important. Keeping the active count accurate is relatively simple; the director increases it when it starts the thread, and decreases it when the thread dies. Likewise the count of delayed processes is straightforward; when a process delays, it increases the count of delayed processes, and the director keeps track of when to wake it up. The count is decreased when a delayed process resumes.

However, due to the conditional communication constructs, keeping the blocked count accurate requires a little more effort. For a basic send or receive, a process is registered as being blocked when it arrives at the rendezvous point before the matching communication. The blocked count is then decreased by one when the corresponding communication arrives. However what happens when an actor is carrying out a conditional communication construct? In this case the process keeps track of all of the branches for which the guards were true, and when all of those are blocked trying to rendezvous, it registers the process as being blocked. When one of the branches succeeds with a rendezvous, the process is registered as being unblocked.

14.5.4 Terminating the model

A process can finish in one of two ways: either by returning false in its prefire() or postfire() meth-

```
public void run() {
   try {
      boolean iterate = true;
      while (iterate) {
         // container is checked for null to detect the termination
         // of the actor.
         iterate = false;
         if ((Entity)_actor).getContainer() != null && _actor.prefire()) {
            _actor.fire();
            iterate = _actor.postfire();
         }
      }
   } catch (TerminateProcessException t) {
      // Process was terminated early
   } catch (IllegalActionException e) {
      _manager.fireExecutionError(e);
   } finally {
      try {
         _actor.wrapup();
      } catch (IllegalActionExeption e) {
         _manager.fireExecutionError(e);
      }
      _director.decreaseActiveCount();
   }
}
```

FIGURE 14.11. Code executed by ProcessThread.run().

ods, in which case it is said to have finished *normally,* or by being terminated *early* by a TerminateProcessException. For example, if a source process is intended to send ten tokens and then finish, it would exit its fire() method after sending the tenth token, and return false in its postfire() method. This causes the ProcessThread, see figure 14.11, representing the process, to exit the while loop and execute the finally clause. The finally clause calls wrapup() on the actor it represents, decreases the count of active processes in the director, and the thread representing the process dies.

A TerminateProcessException is thrown whenever a process tries to communicate via a channel whose receiver has its *finished* flag set to true. When a TerminateProcessException is caught in ProcessThread, the finally clause is also executed and the thread representing the process dies.

To terminate the model, the director sets the *finished* flag in each receiver. The next occasion a process tries to send to or receive from the channel associated with that receiver, a TerminateProcessException is thrown. This mechanism can also be used in a selective fashion to terminate early any processes that communicate via a particular channel. When the director controlling the execution of the model detects real deadlock, it returns from its fire() method. In the absence of hierarchy, this causes the wrapup() method of the director to be invoked. It is the wrapup() method of the director that sets the finished flag in each receiver. Note that the TerminateProcessException is a runtime exception so it does not need to be declared as being thrown.

There is also the option of abruptly terminating all the processes in the model by calling terminate() on the director. This method differs from the approach described in the previous paragraph in that it stops all the threads immediately and does not give them a chance to update the model state. After calling this method, the state of the model is unknown and so the model should be recreated after calling this method. This method is only intended for situations when the execution of the model has obviously gone wrong, and for it to finish normally would either take too long or could not happen. It should rarely be called.

14.5.5 Pausing/Resuming the Model

Pausing and resuming a model does not affect the outcome of a particular execution of the model, only the rate of progress. The execution of a model can be paused at any stage by calling the pause() method on the director. This method is blocking, and will only return when the model execution has been successfully paused. To pause the execution of a model, the director sets a *paused* flag in every receiver, and the next occasion a process tries to send to or receive from the channel associated with that receiver, it is paused. The whole model is paused when all the active processes are delayed, paused or blocked. To resume the model, the resume() method can similarly be called on the director This method resets the paused flag in every receiver and wakes up every process waiting on a receiver lock. If a process was paused, it sees that it is no longer paused and continues. The ability to pause and resume the execution of a model is intended primarily for user interface control.

14.6 Technical Details

14.6.1 Brief Introduction to Threads in Java

The CSP domain, like the rest of Ptolemy II, is written entirely in Java and takes advantage of the features built into the language. In particular, the CSP domain depends heavily on *threads* and on *monitors* for controlling the interaction between threads. In any multi-threaded environment, care has to be taken to ensure that the threads do not interact in unintended ways, and that the model does not dead-

lock. Note deadlock in this sense is a bug in the *modeling environment*, which is different from the deadlock talked about before which may or may not be a bug in the *model* being executed.

A monitor is a mechanism for ensuring mutual exclusion between threads. In particular if a thread has a particular monitor, acquired in order to execute some code, then no other thread can simultaneously have that monitor. If another thread tries to acquire that monitor, it stalls until the monitor becomes available. A monitor is also called a *lock*, and one is associated with every object in Java.

Code that is associated with a lock is defined by the *synchronized* keyword. This keyword can either be in the signature of a method, in which case the entire method body is associated with that lock, or it can be used in the body of a method using the syntax:

```
synchronized(object) {
  // synchronized code goes here
}
```

This causes the code inside the brackets to be associated with the lock belonging to the specified object. In either case, when a thread tries to execute code controlled by a lock, it must either acquire the lock or stall until the lock becomes available. If a thread stalls when it already has some locks, those locks are not released, so any other threads waiting on those locks cannot proceed. This can lead to deadlock when all threads are stalled waiting to acquire some lock they need.

A thread can voluntarily relinquish a lock when stalling by calling *object.*wait() where *object* is the object to relinquish and wait on. This causes the lock to become available to other threads. A thread can also wake up any threads waiting on a lock associated with an object by calling notifyAll() on the object. Note that to issue a notifyAll() on an object it is necessary to own the lock associated with that object first. By careful use of these methods it is possible to ensure that threads only interact in intended ways and that deadlock does not occur.

Approaches to locking used in the CSP domain.
One of the key coding patterns followed is to wrap each wait() call in a while loop that checks some flag. Only when the flag is set to false can the thread proceed beyond that point. Thus the code will often look like

```
synchronized(object) {
  ...
  while(flag) {
    object.wait();
  }
  ...
}
```

The advantage to this is that it is not necessary to worry about what other thread issued the notifyAll() on the lock; the thread can only continue when the notifyAll() is issued *and* the flag has been set to false.

Another approach used is to keep the number of locks acquired by a thread as few as possible, preferably never more than one at a time. If several threads share the same locks, and they must acquire more than one lock at some stage, then the locks should always be acquired in the same order. To see how this prevent deadlocks, consider two threads, *thread1* and *thread2*, that are using two locks A and B. If *thread1* obtains A first, then B, and *thread2* obtains B first then A, then a situation could

arise whereby *thread1* owns lock A and is waiting on B, and *thread2* owns lock B and is waiting on A. Neither thread can proceed and so deadlock has occurred. This would be prevented if both threads obtained lock A first, then lock B. This approach is sufficient, but not necessary to prevent deadlocks, as other approaches may also prevent deadlocks without imposing this constraint on the program [43].

Finally, deadlock often occurs even when a thread, which already has some lock, tries to acquire another lock only to issue a notifyAll() on it. To avoid this situation, it is easiest if the notifyAll() is issued from a *new thread* which has no locks that could be held if it stalls. This is often used in the CSP domain to wake up any threads waiting on receivers, for example after a pause or when terminating the model. The class NotifyThread, in the ptolemy.actor.process package, is used for this purpose. This class takes a list of objects in a linked list, or a single object, and issues a notifyAll() on each of the objects from within a new thread.

The CSP domain kernel makes extensive use of the above patterns and conventions to ensure the modeling engine is deadlock free.

14.6.2 Rendezvous Algorithm

In CSP, the *locking point* for all communication between processes is the *receiver*. Any occasion a process wishes to send or receive, it must first acquire the lock for the receiver associated with the channel it is communicating over. Two key facts to keep in mind when reading the following algorithms are that each channel has exactly one receiver associated with it and that at most one process can be trying to send to (or receive from) a channel at any stage. The constraint that each channel can have at most one process trying to send to (or receive from) a channel at any stage is not currently enforced, but an exception will be thrown if such a model is not constructed.

The rendezvous algorithm is *entirely symmetric* for the put() and the get(), except for the direction the token is transferred. This helps reduce the deadlock situations that could arise and also makes the interaction between processes more understandable and easier to explain. The algorithm controlling how a get() proceeds is shown in figure 14.12. The algorithm for a put() is exactly the same except that put and get are swapped everywhere. Thus it suffices to explain what happens when a get() arrives at a receiver, i.e. when a process tries to receive from the channel associated with the receiver.

When a get() arrives at a receiver, a put() is either already waiting to rendezvous or it is not. Both the get() and put() methods are entirely synchronized on the receiver so they cannot happen simultaneously (only one thread can possess a lock at any given time). Without loss of generality assume a get() arrives before a put(). The rendezvous mechanism is basically three steps: a get() arrives, a put() arrives, the rendezvous completes.

(1) When the get() arrives, it sees that it is first and sets a flag saying a get is waiting. It then waits on the receiver lock while the flag is still true,

(2) When a put() arrives, it sets the *getWaiting* flag to false, wakes up any threads waiting on the receiver (including the get), sets the *rendezvousComplete* flag to false and then waits on the receiver while the *rendezvousComplete* flag is false,

(3) The thread executing the get() wakes up, sees that a put() has arrived, sets the *rendezvousComplete* flag to true, wakes up any threads waiting on the receiver and returns thus releasing the lock. The thread executing the put() then wakes up, acquires the receiver lock, sees that the rendezvous is complete and returns.

Following the rendezvous, the state of the receiver is exactly the same as before the rendezvous arrived, and it is ready to mediate another rendezvous. It is worth noting that the final step, of making

sure the second communication to arrive does not return until the rendezvous is complete, is necessary to ensure that the correct token gets transferred. Consider the case again when a get() arrives first, except now the put() returns immediately if a get() is already waiting. A put() arrives, places a token in the receiver, sets the get waiting flag to false and returns. Now suppose another put() arrives before the get() wakes up, which will happen if the thread the put() is in wins the race to obtain the lock on the receiver. Then the second put() places a new token in the receiver and sets the put waiting flag to true. Then the get() wakes up, and returns with the wrong token! This is known as a *race condition*, which will lead to unintended behavior in the model. This situation is avoided by our design.

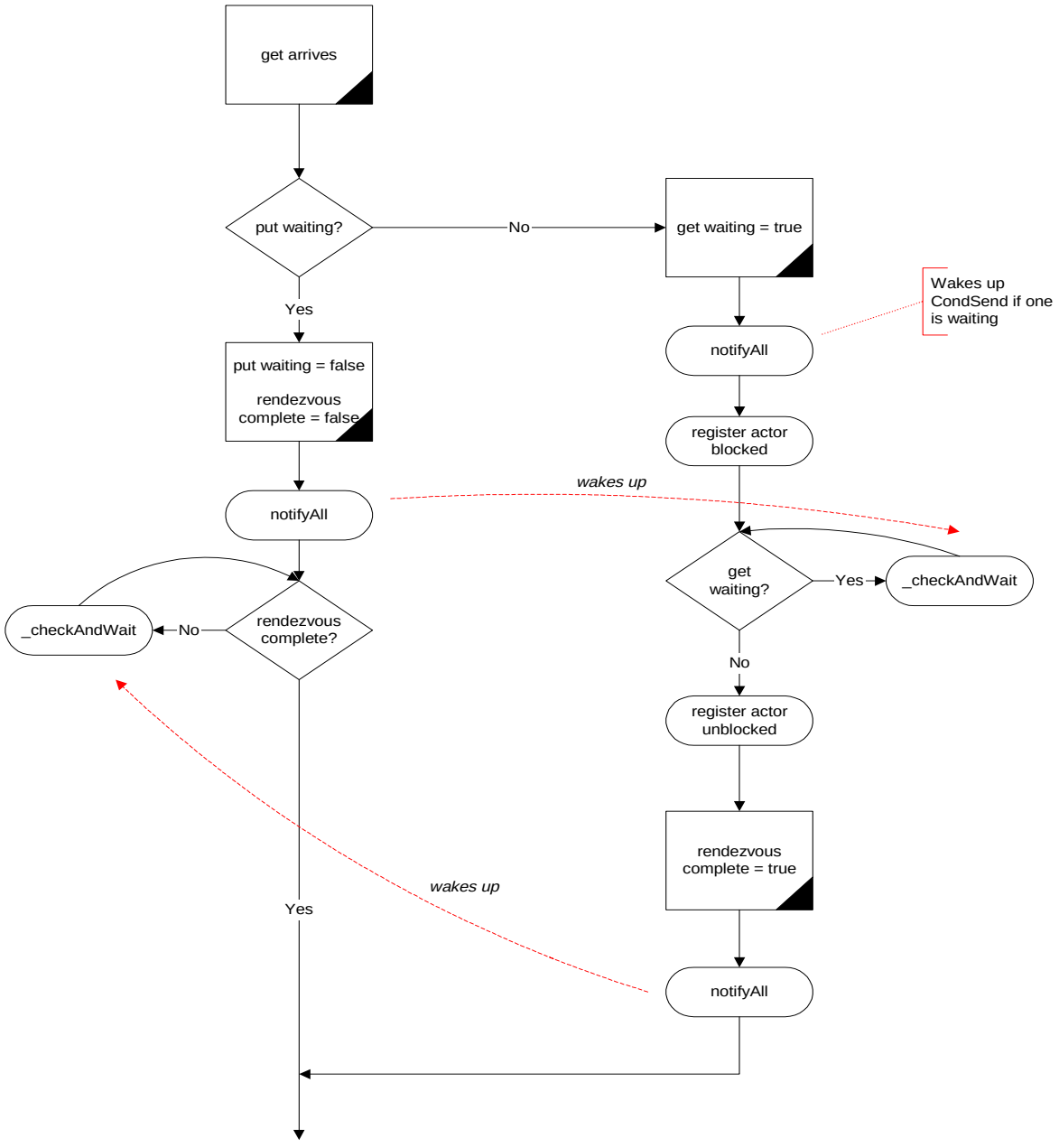


FIGURE 14.12. Rendezvous algorithm.

14.6.3 Conditional Communication Algorithm

There are two steps involved in executing a CIF or a CDO: first deciding which enabled branch succeeds, then carrying out the rendezvous.

Built on top of rendezvous:
When a conditional construct has more than one enabled branch (guard is true or absent), a new thread is spawned for each enabled branch. The job of the chooseBranch() method is to control these threads and to determine which branch should be allowed to successfully rendezvous. These threads and the mechanism controlling them are entirely separate from the rendezvous mechanism described in section 14.6.2, with the exception of one special case, which is described in section 14.6.4. Thus the conditional mechanism can be viewed as being built on top of basic rendezvous: conditional communication knows about and needs basic rendezvous, but the opposite is not true. Again this is a design decision which leads to making the interaction between threads easier to understand and is less prone to deadlock as there are fewer interaction possibilities to consider.

Choosing which branch succeeds.
The manner in which the choice of which branch can rendezvous is worth explaining. The chooseBranch() method in CSPActor takes an array of branches as an argument. If all of the guards are false, it returns -1, which indicates that all the branches failed. If exactly one of the guards is true, it performs the rendezvous directly and returns the identification number of the successful branch. The interesting case is when more than one guard is true. In this case, it creates and starts a new thread for each branch whose guard is true. It then waits, on an internal lock, for one branch to succeed. At that point it gets woken up, sets a finished flag in the remaining branches and waits for them to fail. When all the threads representing the branches are finished, it returns the identification number of the successful branch. This approach is designed to ensure that exactly one of the branches created successfully performs a rendezvous.

Algorithm used by each branch:
Similar to the approach followed for rendezvous, the algorithm by which a thread representing a branch determines whether or not it can proceed is entirely *symmetrical* for a ConditionalSend and a ConditionalReceive. The algorithm followed by a ConditionalReceive is shown figure 14.14. Again the locking point is the receiver, and all code concerned with the communication is synchronized on the receiver. The receiver is also where all necessary flags are stored.

Consider three cases.

(1) a conditionalReceive arrives and a put is waiting.

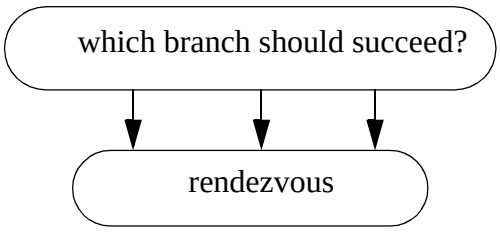


FIGURE 14.13. Conceptual view of how conditional communication is built on top of rendezvous.

In this case, the branch checks if it is the first branch to be ready to rendezvous, and if so, it is goes ahead and executes a get. If it is not the first, it waits on the receiver. When it wakes up, it checks if it is still alive. If it is not, it registers that it has failed and dies. If it is still alive, it starts again by trying to be the first branch to rendezvous. Note that a put cannot disappear.

(2) a conditionalReceive arrives and a conditionalSend is waiting

When both sides are conditional branches, it is up to the branch that arrives second to check whether the rendezvous can proceed. If both branches are the first to try to rendezvous, the conditionalReceive executes a get(), notifies its parent that it succeeded, issues a notifyAll() on the receiver and dies. If not, it checks whether it has been terminated by chooseBranch(). If it has, it

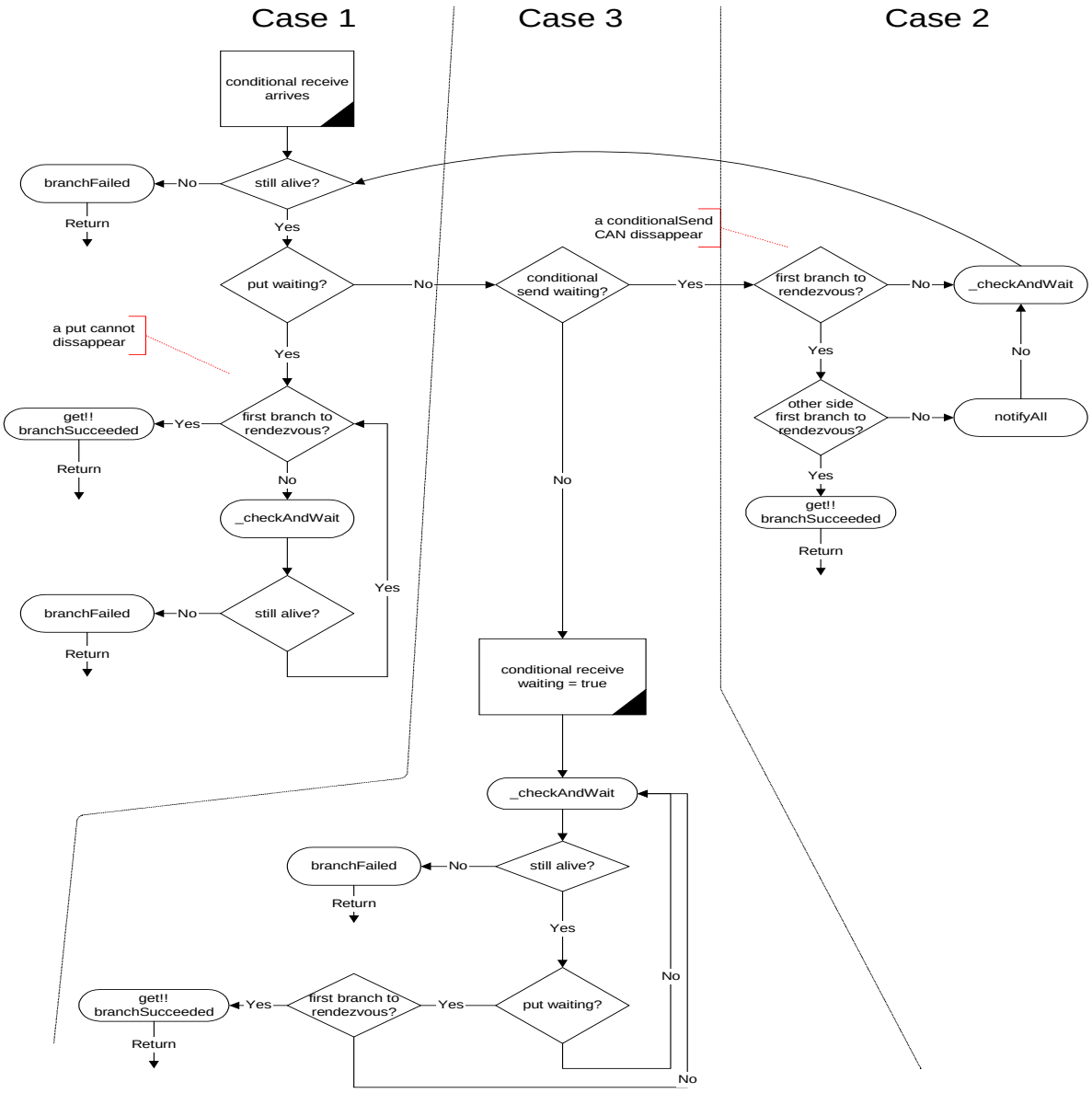


FIGURE 14.14. Algorithm used to determine if a conditional rendezvous branch succeeds or fails

registers with chooseBranch() that it has failed and dies. If it has not, it returns to the start of the algorithm and tries again. This is because a ConditionalSend could disappear. Note that the parent of the first branch to arrive at the receiver needs to be stored for the purpose of checking if both branches are the first to arrive.

This part of the algorithm is somewhat subtle. When the second conditional branch arrives at the rendezvous point it checks that *both* sides are the first to try to rendezvous for their respective processes. If so, then the conditionalReceive executes a get(), so that the conditionalSend is never aware that a conditionalReceive arrived: it only sees the get().

(3) a conditionalReceive arrives first.

It sets a flag in the receiver that it is waiting, then waits on the receiver. When it wakes up, it checks whether it has been killed by chooseBranch. If it has, it registers with chooseBranch that it has failed and dies. Otherwise it checks if a put is waiting. It only needs to check if a put is waiting because if a conditionalSend arrived, it would have behaved as in case (2) above. If a put is waiting, the branch checks if it is the first branch to be ready to rendezvous, and if so it is goes ahead and executes a get. If it is not the first, it waits on the receiver and tries again.

14.6.4 Modification of Rendezvous Algorithm

Consider the case when a conditional send arrives before a get. If all the branches in the conditional communication that the conditional send is a part of are blocked, then the process will register itself as blocked with the director. Then the get comes along, and even though a conditional send is waiting, it too would register itself as blocked. This leads to one too many processes being registered as blocked, which could lead to premature deadlock detection.

To avoid this, it is necessary to modify the algorithm used for rendezvous slightly. The change to the algorithm is shown in the dashed ellipse in figure 14.15. It does not affect the algorithm except in the case when a conditional send is waiting when a get arrives at the receiver. In this case the process that calls the get should wait on the receiver until the conditional send waiting flag is false. If the conditional send succeeded, and hence executed a put, then the get waiting flag and the conditional send waiting flag should both be false and the actor proceeds through to the third step of the rendezvous. If the conditional send failed, it will have reset the conditional send waiting flag and issued a notifyAll() on the receiver, thus waking up the get and allowing it to properly wait for a put.

The same reasoning also applies to the case when a conditional receive arrives at a receiver before a put.

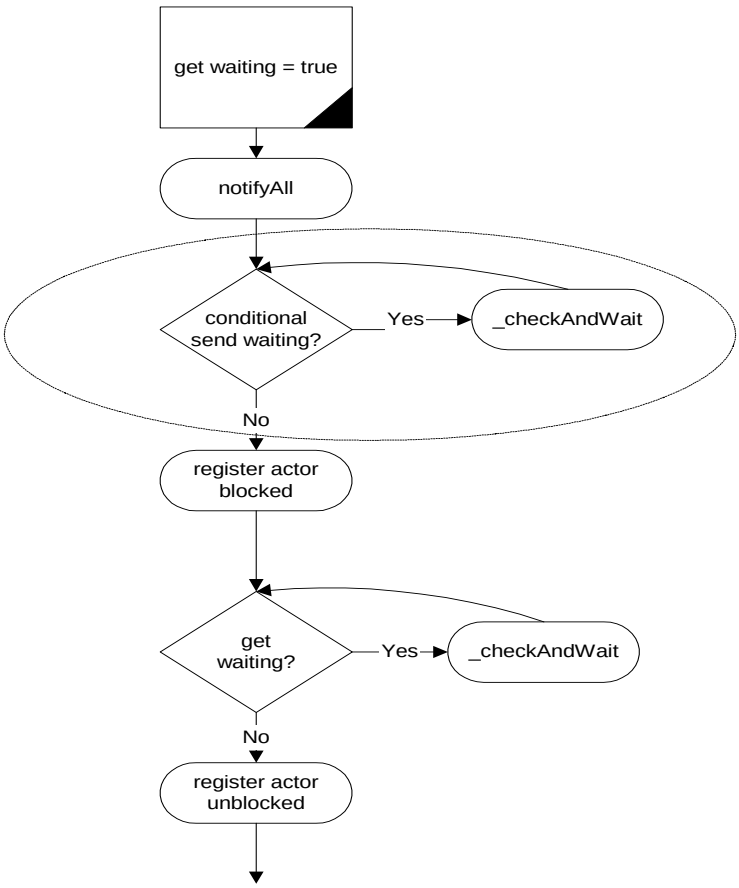


FIGURE 14.15. Modification of rendezvous algorithm, section 14.6.4, shown in ellipse.

15

DDE Domain

Author: John S. Davis II

15.1 Introduction

The distributed discrete event (DDE) model of computation incorporates a distributed notion of time into a dataflow style of communication. Time progresses in a DDE model when the actors in the model execute and communicate. Actors in a DDE model communicate by sending messages through bounded, FIFO channels. Time in a DDE model is distributed and localized, and the actors of a DDE model each maintain their own local notion of the current time. Local time information is shared between two connected actors whenever a communication between said actors occurs. Conversely, communication between two connected actors can occur only when constraints on the relative local time information of the actors are adhered to.

The DDE domain is based on distributed discrete event processing and leverages a wealth of research devoted to this topic. Several tutorial publications on this topic exist in [18][24][39][60]. The DDE domain implements a specific variant of distributed discrete event systems (DDES) as expounded by Chandy and Misra [18]. While the DDE domain has similarities with DDES, the distributed discrete event domain serves as a framework for studying DDES with two special emphases. First we consider DDES from a dataflow perspective; we view DDE as an implementation of the Kahn dataflow model [41] with distributed time added on top. Second we study DDES not with the goal of improving execution speed (as has been the case traditionally). Instead we study DDES to learn its usefulness in modeling and designing systems that are timed and distributed.

15.2 DDE Semantics

Operationally, the semantics of the DDE domain can be separated into two functionalities. The first functionality relates to how time advances during the communication of data and how communication proceeds via blocking reads and writes. The second functionality considers how a DDE model prevents deadlock due to local time dependencies. The technique for preventing deadlock involves the

communication of *null messages* that consist solely of local time information.

15.2.1 Enabling Communication: Advancing Time

Communicating Tokens. A DDE model consists of a network of sequential actors that are connected via unidirectional, bounded, FIFO queues. Tokens are sent from a sending actor to a receiving actor by placing a token in the appropriate queue where the token is stored until the receiving actor consumes it. If a process attempts to read a token from a queue that is empty, then the process will block until a token becomes available on the channel. If a process attempts to write a token to a queue that is full, then the process will block until space becomes available for more tokens in that queue. Note that this blocking read/write paradigm is equivalent to the operational semantics found in non-timed process networks (PN) as implemented in Ptolemy II (see the PN Domain chapter).

If all processes in a DDE model simultaneously block, then the model deadlocks. Deadlock that is due to processes that are either waiting to read from an empty queue, *read blocks*, or waiting to write to a full queue, *write blocks*, then we say that the model has experienced *non-timed deadlock*. Non-timed deadlock is equivalent to the notion of deadlock found in bounded process networks scheduling problems as outlined by Parks [68]. If a non-timed deadlock is due to a model that consists solely of processes that are read blocked, then we say that a *real deadlock* has occurred and the model is terminated. If a non-timed deadlock is due to a model that consists of at least one process that is write blocked, then the capacity of the full queues are increased until deadlock no longer exists. Such deadlocks are called *artificial deadlock*, and the policy of increasing the capacity of full queues was shown by Parks to guarantee the execution of a model in bounded memory whenever possible.

Communicating Time. Each actor in a DDE model maintains a local notion of time. As tokens are communicated between actors, time stamps are associated with each token. Whenever an actor consumes a token, the actor's *current time* is set to be equal to that of the consumed token's time stamp. The time stamp value applied to outgoing tokens of an actor is equivalent to that actor's *output time*. For actors that model a process in which there is delay between incoming time stamps and corresponding outgoing time stamps, then the output time is always greater than the current time; otherwise, the output time is equal to the current time. We refer to actors of the former case as *delay actors*.

For a given queue containing time stamped tokens, the time stamp of the first token currently contained by the queue is referred to as the *receiver time* of the queue. If a queue is empty, its receiver time is the value of the time stamp associated with the last token to flow through the queue, or 0.0 if no tokens have traveled through the queue. An actor may consume a token from an input queue given that the queue has a token available and the receiver time of the queue is less than the receiver times of all other input queues contained by the actor. If the queue with the smallest receiver time is empty, then the actor blocks until this queue receives a token, at which time the actor considers the updated receiver time in selecting a queue to read from.

Figure 15.1 shows three actors, each with three input queues. Actor *A* has two tokens available on the top queue, no tokens available on the middle queue and one token available on the bottom queue. The receiver times of the top, middle and bottom queue are respectively, 17.0, 12.0 and 15.0. Since the queue with the minimum receiver time (the middle queue) is empty, *A* must block on this queue before it proceeds. In the case of actor *B*, the minimum receiver time belongs to the bottom queue. Thus, *B* would proceed by consuming the token found on the bottom queue. After consuming this token, *B* would then compare all of its receiver times to determine which token it could consume from next. Actor *C* is an example of an actor that contains multiple input queues with identical receiver times. To accommodate this situation, each actor assigns a unique priority to each input queue. An actor can con-

sume a token from a queue if no other queue has a lower receiver time and if all queues that have an identical receiver time also have a lower priority.

Each receiver has a *completion time* that is set during the initialization of a model. The completion time of the receiver specifies the time after which the receiver will no longer operate. If the time stamp of the oldest token in a receiver exceeds the completion time, then that receiver will become *inactive.*

15.2.2 Maintaining Communication: Null Tokens

Deadlocks can occur in a DDE model in a form that differs from the deadlocks described in the previous section. This alternative form of deadlock occurs when an actor read blocks on an input port even though it contains other ports with tokens. The topology of a DDE model can lead to deadlock as read blocked actors wait on each other for time stamped tokens that will never appear. Figure 15.2 illustrates this problem. In this topology, consider a situation in which actor *A* only creates tokens on its lower output queue. This will lead to tokens being created on actor *C*'s output queue but no tokens will be created on *B*'s output queue (since *B* has no tokens to consume). This situation results in *D* read blocking indefinitely on its upper input queue even though it is clear that no tokens will ever flow through this queue. The result: *timed deadlock!* The situation shown in figure 15.2 is only one example of timed deadlock. In fact there are two types of timed deadlock: *feedforward* and *feedback*.

Figure 15.2 is an example of feedforward deadlock. Feedforward deadlock occurs when a set of connected actors are deadlocked such that all actors in the set are read blocked and at least one of the actors in the set is read blocked on an input queue that has a receiver time that is less than the local clock of the input queue's source actor. In the example shown above, the upper input queue of *B* has a receiver time of 0.0 even though the local clock of *A* has advanced to 8.0.

Feedback deadlock occurs when a set of cyclically connected actors are deadlocked such that all actors in the set are read blocked and at least one actor in the set, say actor *X*, is read blocked on an input queue that can read tokens which are directly or indirectly a result of output from that same actor (actor *X*). Figure 15.3 is an example of feedback timed deadlock. Note that *B* can not produce an output based on the consumption of the token timestamped at 5.0 because it must wait for a token on the upper input that depends on the output of *B*!

Preventing Feedforward Timed Deadlock. To address feedforward timed deadlock, *null tokens* are

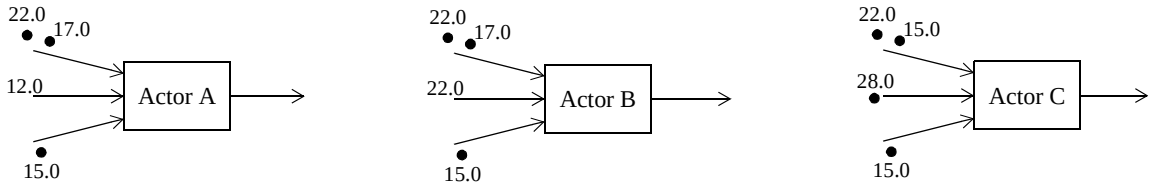


FIGURE 15.1. DDE actors and local time.

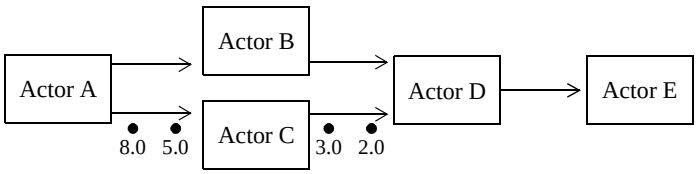


FIGURE 15.2. Timed deadlock (feedforward).

employed. A null token provides an actor with a means of communicating time advancement even though data (*real* tokens) are not being transmitted. Whenever an actor consumes a token, it places a null token on each of its output queues such that the time stamp of the null token is equal to the current time of the actor. Thus, if actor *A* of figure 15.2, produced a token on its lower output queue at time 5.0, it would also produce a null token on its upper output queue at time 5.0.

If an actor encounters a null token on one of its input queues, then the actor does the following. First it consumes the tokens of all other input queues it contains given that the other input queues have receiver times that are less than or equal to the time stamp of the null token. Next the actor removes the null token from the input queue, sets its current time to equal the time stamp of the null token and produces a null token on all output queues such that the produced null tokens are time stamped to the current time. As an example, if *B* in figure 15.2 consumes a null token on its input with a time stamp of 5.0 then it would also produce a null token on its output with a time stamp of 5.0.

The result of using null tokens is that time information is evenly propagated through a model's topology. The beauty of null tokens is that they inform actors of inactivity in other components of a model without requiring centralized dissemination of this information. Given the use of null tokens, feedforward timed deadlock is prevented in the execution of DDE models. It is important to recognize that null tokens are used solely for the purpose of avoiding deadlocks. Null tokens do not represent any actual components of the physical system being modeled. Hence, we do not think of a null token as a real token. Furthermore, the production of a null token that is the direct result of the consumption of a null token is not considered computation from the standpoint of the system being modeled. The idea of null tokens was first espoused by Chandy and Misra [18].

Preventing Feedback Timed Deadlock. We address feedback timed deadlock as follows. All feedback loops are required to have a cumulative time stamp increment that is greater than zero. In other words, feedback loops are required to contain delay actors. Peacock, Wong and Manning [69] have shown that a necessary condition for feedback timed deadlock is that a feedback loop must contain no delay actors. The delay value (delay = output time - current time) of a delay actor must be chosen wisely; it must be less then the smallest delta time of all other actors contained in the same feedback loop. *Delta time* is the difference between the time stamps of a token that is consumed by an actor and the corresponding token that is produced in direct response. If a system being modeled has characteristics that prevent a fixed, positive lower bound on delta time from being specified, then our approach can not solve feedback timed deadlock. Such a situation is referred to as a *Zeno condition*. An application involving an approximated Zeno condition is discussed in section 15.3 below.

The DDE software architecture provides one delay actor for use in preventing feedback timed deadlock: *FBDelay.* We have left out a key operational detail of FBDelay that relates to the process of initializing execution in a model with a feedback delay. This discussion can be found in section 15.4.3 below.

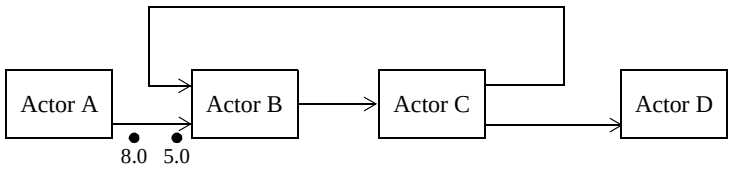


FIGURE 15.3. Timed Deadlock (Feedback)

15.2.3 Alternative Distributed Discrete Event Methods

The field of distributed discrete event simulation, also referred to as parallel discrete event simulation (PDES), has been an active area of research since the late 1970's [18][24][39][60][69]. Recently there has been a resurgence of activity [5][6][10]. This is due in part to the wide availability of distributed frameworks for hosting simulations and the application of parallel simulation techniques to non-research oriented domains. For example, several WWW search engines are based on network of workstation technology.

The field of distributed discrete event simulation can be cast into two camps that are distinguished by the blocking read approach taken by the actors. One camp was introduced by Chandy and Misra [18][24][60][69] and is known as *conservative* blocking. The second camp was introduced by David Jefferson through the Jet Propulsion Laboratory Time Warp system and is referred to as the *optimistic* approach [39][24]. The implementation found in the DDE domain follows the conservative approach.

15.3 Example DDE Applications

To illustrate distributed discrete event execution, we have developed an applet that features a feedback topology and incorporates polymorphic as well as DDE specific actors. The model, shown in figure 15.4, consists of a single source actor (ptolemy/actor/lib/Clock) and an upper and lower branch of four actors each. The upper and lower branches have identical topologies and are fed an identical stream of tokens from the Clock source with the exception that in the lower branch ZenoDelay replaces FBDelay.

As with all feedback topologies in DDE (and DE) models, a positive time delay is necessary in feedback loops to prevent deadlock. If the time delay of a given loop is lower bounded by zero but can not be guaranteed to be greater than a fixed positive value, then a Zeno condition occurs in which time will not advance beyond a certain point even though the actors of the feedback loop continue to execute without deadlocking. ZenoDelay extends FBDelay and is designed so that a Zeno condition will be encountered. When execution of the model begins, both FBDelay and ZenoDelay are used to feed back null tokens into Wire so that the model does not deadlock. After local time exceeds a preset value, ZenoDelay reduces its delay so that the lower branch approximates a Zeno condition.

In centralized discrete event systems, Zeno conditions prevent progress in the entire model. This is true because the feedback cycle experiencing the Zeno condition prevents time from advancing in the

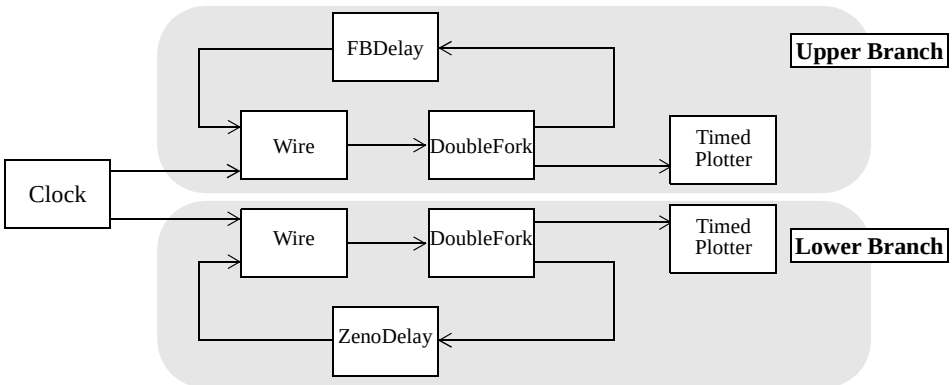


FIGURE 15.4. Localized Zeno condition topology.

entire model. In contrast, distributed discrete event systems localize Zeno conditions as much as is possible based on the topology of the system. Thus, a Zeno condition can exist in the lower branch and the upper branch will continue its execution unimpeded. Localizing Zeno conditions can be useful in large scale modeling in which a Zeno condition may not be discovered until a great deal of time has been invested in execution of the model. In such situations, partial data collection may proceed prior to correction of the delay error that resulted in the Zeno condition.

15.4 Building DDE Applications

To build a DDE application, a DDEDirector must be used. This will ensure that each actor under control of the director will be allocated DDEReceivers and that each actor will be assigned a TimeKeeper to manage the actor's local notion of time. The DDE domain is typed so that actors used in a model must be derived from ptolemy/actor/TypedAtomicActor. The DDE domain is designed to use both DDE specific actors as well as polymorphic actors. DDE specific actors can take advantage of DDEActor and DDEIOPort which are designed to provide convenient support for specifying time in the production and consumption of tokens.

15.4.1 DDEActor

The DDE model of computation makes one very strong assumption about the execution of an actor: *all input ports of an actor operating in a DDE model must be regularly polled to determine which input channel has the oldest pending event*. Any actor that adheres to this assumption can operate in a DDE model. Thus, many polymorphic actors found in ptolemy/actor/[lib, gui] are suitable for operation in DDE models. For convenience, DDEActor was developed to simplify the construction of actors that have DDE semantics. DDEActor has three key methods as follows:

getCurrentTime(). This method returns the actor thread's local notion of time. This method relies on Java's Thread.currentThread() returning the thread that is assigned to the actor and hence this method is intended for self-referencing. If actor *A* calls getCurrentTime() of actor *B*, then the result will be the current time of *A* (not *B* as one might expect).

getNextToken(). This method polls each input port of an actor and returns the (non-Null) token that represents the oldest event. This method blocks accordingly as outlined in section 15.2.1 (Communicating Time).

getLastPort(). This method returns the input IOPort from which the last (non-Null) token was consumed. This method presumes that getNextToken() is being used for token consumption.

15.4.2 DDEIOPort

DDEIOPort extends TypedIOPort with parameters for specifying time stamp values of tokens that are being sent to neighboring actors. Since DDEIOPort extends TypedIOPort, use of DDEIOPorts will not violate the type resolution process. DDEIOPort is not necessary to facilitate communication between actors executing in a DDE model; standard TypedIOPorts are sufficient in most communication. DDEIOPorts become useful when the time stamp to be associated with an outgoing token is greater than the current time of the sending actor. Hence, DDEIOPorts are only useful in conjunction with delay actors (see "Enabling Communication: Advancing Time" on page 15-2, for a definition of delay actor). Most polymorphic actors available for Ptolemy II are not delay actors.

15.4.3 Feedback Topologies

In order to execute feedback topologies that will not deadlock, FBDelay actors must be used. FBDelay is found in the DDE kernel package. FBDelay actors do not perform computation, but instead increment the time stamps of tokens that flow through them by a specified delay. The delay value of an FBDelay actor must be chosen to be less than the delta time of the feedback cycle in which the FBDelay actor is contained. Elaborate delay values can be specified by overriding the getDelay() method in subclasses of FBDelay. An example of such can be found in ptolemy/domains/dde/demo/LocalZeno/ZenoDelay.java.

A difficulty found in feedback cycles occurs in the initialization of a model's execution. In figure 15.5 we see that even if Actor B is an FBDelay actor, the system will deadlock if the first event is created by *A* since *C* will block on an event from *B*. To alleviate this problem a special time stamp value has been reserved: TimeQueuedReceiver.IGNORE. When an actor encounters an event with a time stamp of IGNORE (an *ignore event*), the actor will ignore the event and the input channel it is associated with. The actor then considers the other input channels in determining the next available event. After a non-ignore event is encountered and consumed by the actor, all ignore events will be cleared. If all of an actor's input channels contain ignore events, then the actor will clear all ignore events and then proceed with normal operation.

The initialize method of FBDelay produces an ignore event. Thus, in figure 15.5, if *B* is an FBDelay actor, the ignore event it produces will be sent to *C*'s upper input channel allowing *C* to consume the first event of *A*. The production of null tokens and feedback delays will then be sufficient to continue execution from that point on. Note that the production of an ignore event by a FBDelay actor serves as a major distinction between it and all other actors. *If a delay is desired simply to represent the computational delay of a given model, an FBDelay actor should not be used.*

The intricate operation of ignore events requires special consideration when determining the positioning of an FBDelay actor in a feedback topology. An FBDelay actor should be placed so that the ignore event it produces will be ignored in deference to the first real event that enters a feedback cycle. Thus, choosing actor *D* as an FBDelay actor in figure 15.5 would not be useful given that the first real event entering the cycle is created by *A*.

15.5 The DDE Software Architecture

For a model to have DDE semantics, it must have a DDEDirector controlling it. This ensures that the receivers in the ports are DDEReceivers. As with all process domains, each actor in a DDE model is under the control of a ProcessThread, or in the case of DDE, a DDEThread. DDEThreads contain a TimeKeeper that manages the local notion of time that is associated with the DDEThread's actor.

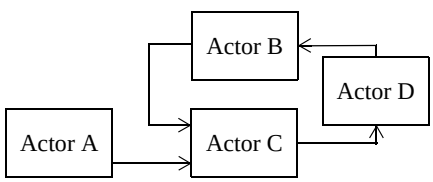


FIGURE 15.5. Initializing Feedback Topologies

15.5.1 Local Time Management

The UML diagram of the local time management system of the DDE domain is shown in figure 15.6 and consists of TimedQueueReceiver, DDEReceiver, DDEThread and TimeKeeper. Since time is localized, the DDEDirector does not have a direct role in this process. Note that DDEReceiver is derived from TimedQueueReceiver. The primary purpose of TimedQueueReceiver is to keep track of a receiver's local time information. DDEReceiver adds blocking read/write functionality to TimedQueueReceiver.

When a DDEDirector is initialized, it instantiates a DDEThread for each actor that the director manages. DDEThreads are derived from ProcessThreads. ProcessThreads provide functionality that is common to all of the process domains (e.g., CSP, DDE and PN). The directors of all process domains (including DDE) assign a single actor to each ProcessThread. ProcessThreads take responsibility of their assigned actor's execution by invoking the iteration methods of the actor. The iteration methods are prefire(), fire() and postfire(); ProcessThreads also invoke wrapup() on the actors they control.

DDEThread extends the functionality of ProcessThread. Upon instantiation, a DDEThread creates a TimeKeeper object and assigns this object to the actor that it controls. The TimeKeeper gets access to each of the DDEReceivers that the actor contains. Each of the receivers can access the TimeKeeper

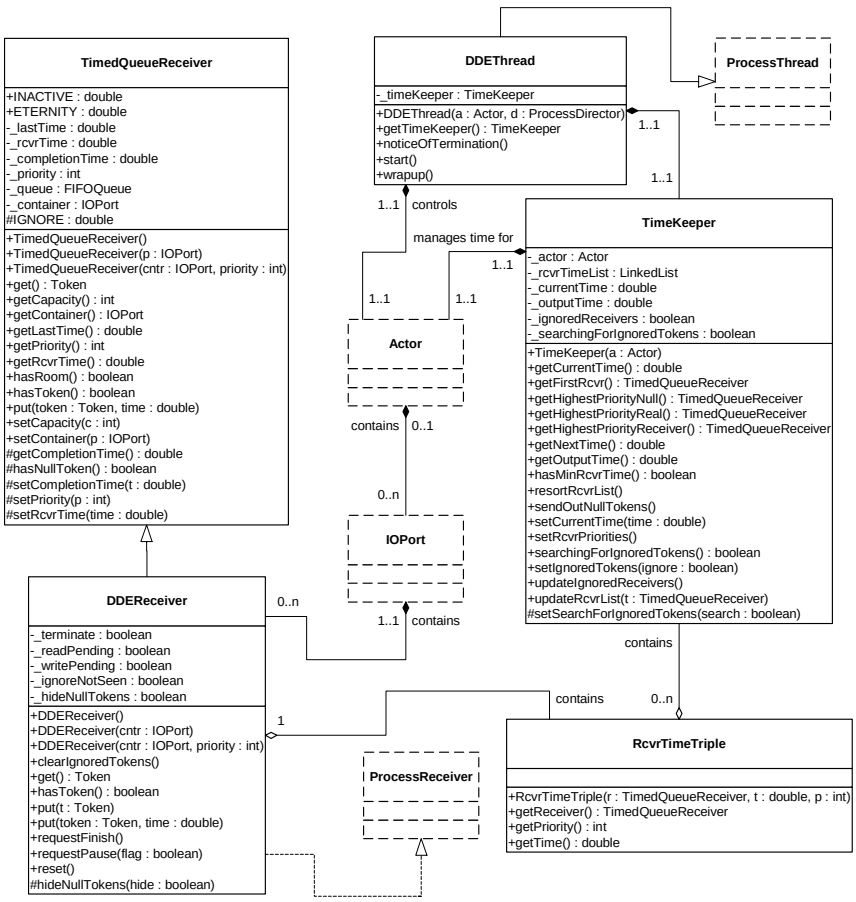


FIGURE 15.6. Key Classes for Locally Managing Time.

and through the TimeKeeper the receivers can then determine their relative receiver times. With this information, the receivers are fully equipped to apply the appropriate blocking rules as they get and put time stamped tokens.

DDEReceivers use a dynamic approach to accessing the DDEThread and TimeKeeper. To ensure domain polymorphism, actors (DDE or otherwise) do not have static references to the TimeKeeper and DDEThread that they are controlled by. To ensure simplified mutability support, DDEReceivers do not have a static reference to TimeKeepers. Access to the local time management facilities is accomplished via the Java Thread.currentThread() method. Using this method, a DDEReceiver dynamically accesses the thread responsible for invoking it. Presumably the calling thread is a DDEThread and appropriate steps are taken if it is not. Once the DDEThread is accessed, the corresponding TimeKeeper can be accessed as well. The DDE domain uses this approach to great gain in DDEReceiver.put(Token) and DDEReceiver.get().

DDEReceiver.put(Token) is derived from the Receiver interface and is accessible by all actors and domains. To facilitate local time advancement, DDEReceiver has a second put() method that has a time argument: DDEReceiver.put(Token, double). This second DDE-specific version of put() is taken advantage of without extensive code by using Thread.currentThread(). DDEReceiver.put() is shown below:

```
public void put(Token token)[1] {
  Thread thread = Thread.current-
Thread();
  double time = getLastTime();
  if( thread instanceof DDEThread ) {
    TimeKeeper timeKeeper = ((DDETh-
read)thread).getTimeKeeper();
    time = timeKeeper.getOutputTime();
  }
  put( token, time )[2];
}
```

[1] DDEReceiver.put(Token) is equivalent to the put() signature of the ptolemy/actor/Receiver interface.

[2] Polymorphic actors need not be aware of DDE-specific code such as DDEReceiver.put(Token,double).

Similar uses of Thread.currentThread() are found throughout DDEReceiver and DDEDirector. Note that while Thread.currentThread() can be quite advantageous, it means that if particular code is called by an inappropriate thread, problems may occur. Such an issue makes testing of code difficult.

15.5.2 Detecting Deadlock

The other kernel classes of the DDE domain are shown in figure 15.7. The purpose of the DDEDirector is to detect and (if possible) resolve timed and/or non-timed deadlock of the model it controls. Whenever a receiver blocks, it informs the director. The director keeps track of the number of active processes, and the number of processes that are either blocked on a read or write. Artificial deadlocks are resolved by increasing the queue capacity of write-blocked receivers.

15.5.3 Ending Execution

Execution of a model ends if either an unresolvable deadlock occurs, the director's completion time is exceeded by all of the actors it manages, or early termination is requested (e.g., by a user interface button). The director's completion time is set via the public *stopTime* parameter of DDEDirector. The completion time is passed on to each DDEReceiver. If a receiver's receiver time exceeds the com-

pletion time, then the receiver becomes inactive. If all receivers of an actor become inactive and the actor is not a source actor, then the actor will end execution and its wrapup() method will be called. In such a scenario, the actor is said to have terminated *normally.*

Early terminations and unresolvable deadlocks share a common mechanism for ending execution. Each DDEReceiver has a boolean `_terminate` flag. If the flag is set to true, then the receiver will throw a `TerminateProcessException` the next time any of its methods are invoked. TerminateProcessExceptions are part of the `ptolemy/actor/process` package and ProcessThreads know to end an actor's execution if this exception is caught. In the case of unresolvable deadlock, the `_terminate` flag of all blocked receivers is set to true. The receivers are then awakened from blocking and they each throw the exception.

15.6 Technical Details

15.6.1 Synchronization Hierarchy

Previously we have discussed in great detail the notion of timed and non-timed deadlock. Separate from these notions is a different kind of deadlock that can be inherent in a modeling environment if the environment is not designed properly. This notion of deadlock can occur if a system is not *thread safe*. Given the extensive use of Java threads throughout Ptolemy II, great care has been taken to ensure thread safety; we want no *bugs* to exist that might lead to deadlock based on the structure of the Ptolemy II modeling environment. Ptolemy II uses monitors to guarantee thread safety. A *monitor* is a method for ensuring mutual exclusion between threads that both have access to a given portion of

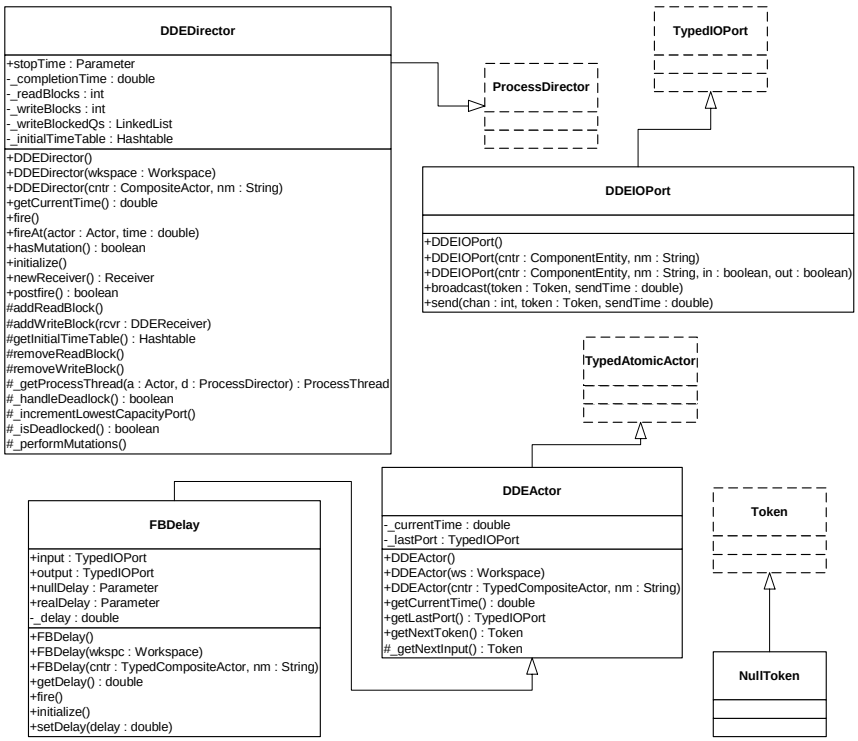


FIGURE 15.7. Additional Classes in the DDE Kernel.

code. To ensure mutual exclusion, threads must acquire a monitor (or *lock*) in order to access a given portion of code. While a thread owns a lock, no other threads can access the corresponding code.

There are several objects that serve as locks in Ptolemy II. In the process domains, there are four primary objects upon which locking occurs: Workspace, ProcessReceiver, ProcessDirector and AtomicActor. The danger of having multiple locks is that separate threads can acquire the locks in competing orders and this can lead to deadlock. A simple illustration is shown in figure 15.8. Assume that both lock *A* and lock *B* are necessary to perform a given set of operations and that both thread 1 and thread 2 want to perform the operations. If thread 1 acquires *A* and then attempts to acquire *B* while thread 2 does the reverse, then deadlock will occur.

There are several ways to avoid the above problem. One technique is to combine locks so that large sets of operations become atomic. Unfortunately this approach is in direct conflict with the whole purpose behind multi-threading. As larger and larger sets of operations utilize a single lock, the limit of the corresponding concurrent program is a sequential program!

Another approach is to adhere to a hierarchy of locks. A hierarchy of locks is an agreed upon order in which locks are acquired. In the above case, it may be enforced that lock *A* is always acquired before lock *B*. A hierarchy of locks will guarantee thread safety [43].

The process domains have an unenforced hierarchy of locks. It is strongly suggested that users of Ptolemy II process domains adhere to this suggested locking hierarchy. The hierarchy specifies that locks be acquired in the following order:

Workspace ⟶ **ProcessReceiver** ⟶ **ProcessDirector** ⟶ **AtomicActor**

The way to apply this rule is to prevent synchronized code in any of the above objects from making a call to code that is to the left of the object in question.

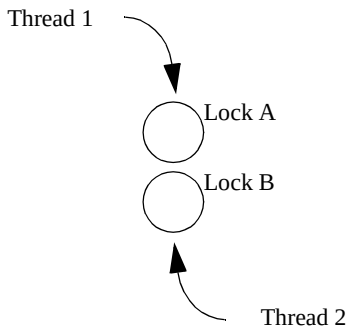


FIGURE 15.8. Deadlock Due to Unordered Locking.

16

PN Domain

Author: Mudit Goel

16.1 Introduction

The process networks (PN) domain in Ptolemy II models a system as a network of sequential processes, implemented as Java threads [65], that communicate by passing messages through unidirectional first-in-first-out (FIFO) channels. A process blocks when trying to read from an empty channel until a message becomes available on it. This model of computation is deterministic in the sense that if the processes are deterministic and communicate only via the channels, then the sequence of values communicated on the channels is completely determined by the model.

PN is a natural model for describing signal processing systems where infinite streams of data samples are incrementally transformed by a collection of processes executing in parallel. Embedded signal processing systems are typically designed to operate indefinitely with limited resources. Thus, we want to execute process network programs forever with bounded buffering on the communication channels whenever possible.

PN can also be used to model the concurrency in the various hardware components of an embedded system. The original process networks model of computation can model the functional behavior of these systems and test them for their functional correctness, but it cannot directly model their real-time behavior. To address this, the PN domain extends the model by introducing a notion of time.

In addition, some systems might display adaptive behavior like migrating code, agents, and arrivals and departures of processes. For this, we provide a mutation mechanism that supports addition, deletion, and changing of processes and channels. With untimed PN, this might display non-determinism, while with timed-PN, it becomes deterministic.

The PN model of computation a superset of the synchronous dataflow model of computation (see the SDF Domain chapter). Thus any SDF actor can be used in the PN domain. Similarly any domain-polymorphic actor can be used in the PN domain. A separate process is created for each of these actors.

The software architecture for PN is described in section 16.3 and the finer technical details are explained in section 16.4.

16.2 Process Network Semantics

16.2.1 Asynchronous Communication

Kahn and MacQueen [40][41] describe a model of computation where processes are connected by communication channels to form a network. Processes produce data elements or *tokens* and send them along a unidirectional communication channel where they are stored in a FIFO queue until the destination process consumes them. This is a form of asynchronous communication between processes. Communication channels are the *only* method processes may use to exchange information. A set of processes that communicate through a network of FIFO queues defines a *program*.

Kahn and MacQueen require that execution of a process be suspended when it attempts to get data from an empty input channel (*blocking reads*). A process may not poll the channel for presence or absence of data. At any given point, a process is either doing some computation (enabled) or it is blocked waiting for data (*read blocked*) on exactly one of its input channels; it cannot wait for data from more than one channel simultaneously. Systems that obey this model are determinate; the history of tokens produced on the communication channels does not depend on the execution order. Therefore, the results produced by executing a program are not affected by the scheduling of the various processes.

In case all the processes in a model are blocked while trying to read from some channel, then we have a *real deadlock*. In such a case, none of the processes can do anything further. The determinacy of the program also guarantees that a real deadlock is a program state and does not depend on the schedule of the various processes in the model.

16.2.2 Bounded Memory Execution

The high level of concurrency in process networks makes it an ideal match for embedded system software and for modeling hardware implementations. But the Kahn-MacQueen semantics do not guarantee bounded memory execution of process networks even if it is possible for the application to execute in bounded memory. Most real-time embedded applications and hardware processes are intended to run indefinitely with a limited amount of memory. Thus bounded memory execution of process networks becomes crucial for its usefulness for hardware and embedded software.

Parks [68] addresses this aspect of process networks and provides an algorithm to make a process network application execute in bounded memory whenever possible. He provides an implementation of the Kahn-MacQueen semantics using *blocking writes* that assigns a fixed capacity to each FIFO channel and forces processes to block temporarily if a channel is full. To avoid introducing deadlock, these capacities are increased if absolutely necessary. A process may not poll the channel for room. Thus a process has three states now: *running* (*executing*), *read blocked*, or *write blocked*.

Deadlocks can now occur when all processes are blocked either on a read or on a write to a channel. In case all the processes in a model are blocked with at least one process blocked on a write, then we have an *artificial deadlock*. On detection of an artificial deadlock, Parks chooses the channel with the smallest capacity among the channels on which processes are blocked on a write and increases its capacity to break the deadlock.

16.2.3 Time

The process networks model of computation lacks a notion of time. In some real time systems and

embedded applications, the real time behavior of a system is as important as the functional correctness. Developers can use process networks test applications for functional correctness and use some other timed model of computation, such as DE, for testing their timing behavior. Introducing a notion of time to the process networks model of computation is therefore a natural extension of PN. This is done in the PN domain in Ptolemy II.

In the PN domain, time is global. That is, all processes in a model share the same time, referred to as the *current time* or *model time*. A process can explicitly wait for time to advance. It can choose to *delay* itself for some period from the current time. When a process delays itself for some length of time from the current time, it is suspended until time has sufficiently advanced, at which stage it wakes up and continues. If the process delays itself for zero time, this will have no effect and the process will continue executing.

For a process, time cannot change during its normal execution (i.e. when the process is enabled). Time for a process may advance in only one of the two states:

1. The process is delayed and is explicitly waiting for time to advance (*delay block*).
2. The process is waiting for data to arrive on one of its input channels (*read block*).

Time is *advanced* when all the processes are blocked on either a delay or on a read from a channel with at least one process delayed. This state of the program is called a *timed deadlock*. In case of a timed deadlock, the current time is advanced just enough to wake up at least one process.

A process can be aware of the global time, but it cannot influence the current time except by delaying itself. This model of time is influenced by Pamela [27], a run time library that is used to model parallel programs.

16.2.4 Mutations

The PN domain tolerates mutations, which are run-time changes in the model structure. Normally, mutations are realized as *change requests* queued with the director or manager. In case of timed PN, requests for mutations are not processed until there is a timed deadlock. Because occurrence of a timed deadlock is determinate, mutations in timed PN are determinate.

In case of untimed PN, there is no determinate point where mutations can occur. The only determinate point in case of untimed PN is a read deadlock. Performing mutations at this point is unlikely to be useful because a real deadlock might never occur. A model with even one non-terminating source never experiences a real deadlock. Thus in case of untimed PN, mutations are performed as soon as they are requested (if they are queued with the director) and when real deadlock occurs (if they are queued with the manager). Since different schedules will result in different states of the model when mutations are performed, the former introduces non-determinism in untimed PN. The details of implementations are presented later in section 16.3.

16.3 The PN Software Architecture

16.3.1 PN Domain

Ptolemy II is modular and is divided into packages, each of which provide separate functionalities. The abstract syntax is separated from the mechanisms that attach semantics. In PN, the package that attaches the process networks semantics is the ptolemy.domains.pn.kernel. A UML static structure dia-

gram is shown in figure 16.1 (see appendix A of chapter 1). This section discusses the implementation in details.

16.3.2 The Execution Sequence

Director.

In process networks, each node of the graph is a separate process. In the PN domain in Ptolemy II, this is achieved by letting each actor have its own separate thread of execution. These threads are based

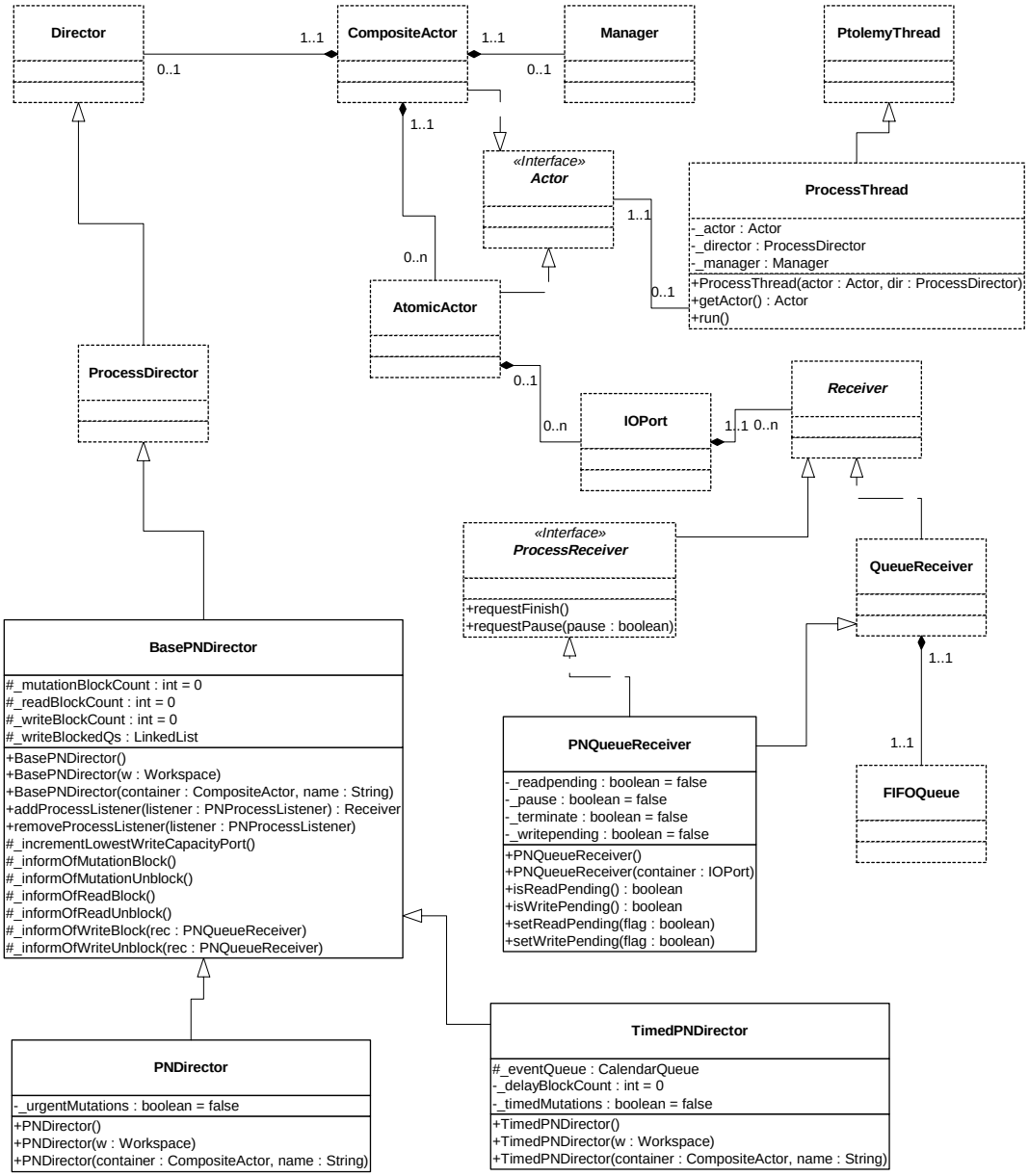


FIGURE 16.1. UML diagram for classes and methods related to the PN domain.

on the native Java threads [65][43] and are inst**ances of `ptolemy.actors.ProcessThread`.**

BasePNDirector:

This is the base class for the directors that govern the execution of a CompositeActor with Kahn process networks (PN) semantics. This base class attaches the Kahn-MacQueen process networks semantics to a composite actor. This director does not support mutations or a notion of model time. This provides a mechanism to perform blocking reads and bounded memory execution (using blocking writes) whenever possible. Thus it is capable of handling real and artificial deadlocks.

The first step in the execution is the call to the initialize() method of the director. This method creates the receivers in the input ports of the actors for all the channels and creates a thread for each actor. It initializes all the actors in the graph. It also sets the count of active actors in the model, which is required for detection of deadlocks and termination, to the number of actors in the composite actor.

The next stage is the iterations. It starts with a call to the prefire() method of the director. This method starts all the threads that were created for the actors in the initialize() method of the director. In PN, this method always returns true.

The fire() method of the director is called next. In PN, the fire() method is responsible for handling deadlocks. This director resolves artificial deadlocks as soon as they arise according to Parks's algorithm as explained in section 16.2.2. On detection of a real deadlock, the method returns.

The last stage of the iteration cycle of the director is the call to the postfire() method. This method returns false if the composite actor containing the director has no input ports. Otherwise it returns true. Returning true implies that if some new data is provided to the composite actor on the input ports, then the execution can resume. If it returns false, then this composite actor will not be fired again. In such a case the executive director or the manager will call the wrapup() method of the top-level composite actor. This in turn calls the wrapup() method of the director. The director then terminates the execution of the composite actor. Details of termination are discussed in section 16.3.4.

PNDirector:

PNDirector is the same as BasePNDirector with one additional functionality. This director supports mutations of a graph. The mutations are processed as soon as they are requested. The point at which the mutations are processed depends on the schedule of the threads in the model. Thus these mutations might introduce non-determinism to the model.

TimedPNDirector:

TimedPNDirector has two functionalities that distinguishes it from BasePNDirector. It introduces a notion of global time to the model and it allows deterministic mutations. Mutations are performed at the earliest timed-deadlock that occurs after they are queued. Since occurrence of timed-deadlock is completely deterministic, performing mutations at this point in the model makes mutations deterministic.

Execution of Actors. As mentioned earlier, a separate thread is responsible for the execution of each actor in PN. This thread is started in the prefire() method of the director. After starting, this thread repeatedly calls the prefire(), fire(), and postfire() methods of the actor. This sequence continues until the postfire() or the prefire() method returns false. The only way for an actor to terminate gracefully in PN is by returning from the fire() method and returning false in the postfire() or prefire() method of the actor. If an actor finishes execution as above, then the thread calls the wrapup() method of the actor. Once this method returns, the thread informs the director about the termination of this actor and finishes its own execution. This actor is not fired again unless the director creates and starts a new thread for the actor. Also, if an actor returns false in its prefire() method, the first time it is called, the actor would never be fired in PN.

Message Passing.

Recall that in Ptolemy II, data transfer between entities is achieved using ports and the receivers embedded in the input ports. Each receiver in an input port is capable of receiving messages from a distinct channel.

An actor calls the send() or broadcast() method on its output port to transmit a token to a remote actor. The port obtains a reference to a remote receiver (via the relation connecting them) and calls the put() method of the receiver, passing it the token. The destination actor retrieves the token by calling the get() method of its input port, which in turn calls the get() method of the designated receiver.

Both the get() and send() methods of the port take an integer index as an argument which the actor uses to distinguish between the different channels its port is connected to. This index specifies the channel to which the data is being sent to or being received from. If the ports are connected to a single channel, then the index is *0*. But if the port is connected to more than one channel (a multiport), say *N* channels, then the index ranges from *0* to *N-1*. The broadcast() method of the port does not require an index as it transmits the token to all the channels it is connected to.

In the PN domain, these receivers are instances of ptolemy.domains.pn.kernel.PNQueueReceiver. These receivers have a FIFO queue in them to provide the functionality of a FIFO channel in a process networks graph. In addition to this, these receivers are also responsible for implementing the blocking reads and blocking writes. They handle this using the get() and the put() methods. These methods are as shown in figures 16.2,16.3.

```
public Token get() {
   IOPort port = getContainer();
   Workspace workspace = port.workspace();
   Actor actor = (Actor)port.getContainer();
   PNDirector director = (PNDirector)actor.getDirector();
   Token result = null;
   synchronized (this) {
      while (!_terminate && !super.hasToken()) {
         director._readBlock();
         _readpending = true;
         while (_readpending && !_terminate) {
            workspace.wait(this);
         }
      }
      if (_terminate) {
         throw new TerminateProcessException("");
      } else {
         result = super.get();
         if (_writepending) {
            director._writeUnblock(this);
            _writepending = false;
            notifyAll(); //Wake up threads waiting on a write;
         }
      }
      while (_pause) {
         director.increasePausedCount();
         workspace.wait(this);
      }
      return result;
   }
}
```

FIGURE 16.2. get() method of PNQueueReceiver.

The get() method checks if the FIFO queue has any tokens. If not, then it increases the count, tracking the number of actors blocked on a read in the director and sets its _readpending flag to true. Then the calling thread is suspended until some actor puts a token in the FIFO queue and sets the _readpending flag of this receiver to false. (This is done in the put() method as described later.) On resuming, it reads and removes the first token from the FIFO queue. In case some process is blocked on a write to this receiver (the FIFO queue is full to capacity), it unblocks that process, notifies it, and returns. This method also handles the termination of the simulation as is explained later in section 16.3.4.

The put() method of the receiver is responsible for implementing the blocking writes. This method checks whether the FIFO queue is full to capacity. If it is, then it sets its _writepending flag to true and informs the director that a process is blocked on a write. Then it suspends the calling process until someone wakes it up after setting the _writepending flag to false. After this, it puts the token into the FIFO queue and checks if some process is blocked on a read from this receiver. If a process is blocked on a read, it unblocks it and informs it that a new token is now available for it to read. Then the method returns.

16.3.3 Detecting deadlocks:

The mechanism for detecting deadlocks in the Ptolemy II implementation of PN is based on the mechanism suggested in [42]. This mechanism requires keeping count of the number of threads currently active, paused, and blocked in the simulation. The number of threads that are currently active in the graph is set by a call to the _increaseActiveCount() of the director. This method is called whenever a new thread corresponding to an actor is created in the simulation. The corresponding method for decreasing the count of active actors (on termination of a process) is _decreaseActiveCount() in the director.

Whenever an actor blocks on a read from a channel, the count of actors blocked on a read is incre-

```
public void put(Token token) {
    IOPort port = getContainer();
    Workspace workspace = port.workspace();
    Actor actor = (Actor)port.getContainer();
    PNDirector director = (PNDirector)actor.getDirector();
    synchronized(this) {
        if (!super.hasRoom()) {
            _writepending = true;
            director._writeBlock(this);
            while(_writepending) {
                workspace.wait(this);
            }
        }
        super.put(token);
        if (_readpending) {
            director._readUnblock();
            _readpending = false;
            notifyAll();
        }
        while (_pause) {
            director.increasePausedCount();
            workspace.wait(this);
        }
    }
}
```

FIGURE 16.3. `put()` method of PNQueueReceiver

mented by calling the _informOfReadBlock() method in director. Similarly, the number of actors blocked on a write is incremented by a call to the _informOfWriteBlock() method of the director. The corresponding methods for decreasing the count of the actors blocked on a read or a write are _informOfReadUnblock() and _informOfWriteUnblock(), respectively. These methods are called from the instances of the PNQueueReceiver class when an actor tries to read from or write to a channel. Similarly, when a process queues a mutation, it informs the director by a call to the _informOfMutationBlock().

Every time an actor blocks, the director checks for a deadlock. If the total number of actors blocked or paused equals the total number of actors active in the simulation, a deadlock is detected. On detection of a deadlock, if one or more actors are blocked on a write, then this is an artificial deadlock. The channel with the smallest capacity among all the channels with actors blocked on a write is chosen and its capacity is incremented by 1. This implements the bounded memory execution as suggested by [68]. If a real deadlock is detected at the top-level composite actor, then the manager terminates the simulation.

16.3.4 Terminating the model:

A simulation can be ended (on detection of a real deadlock) by calling the wrapup() method on either the toplevel composite actor or the corresponding director. This method is normally called by the manager on the top-level composite actor. In PN, this method traverses the topology of the graph and calls the setFinish() method of the receivers in the input ports of all the actors. Since this method is called only when a real deadlock is detected, one can be sure that all the active actors in the simulation are currently blocked on a read from a channel and are waiting in the call to the get() method of a receiver. This fact is used to wrap up the simulation. The setFinish() method of the receiver sets the termination flag to true, and wakes up all the threads currently waiting in the get() method of the receiver(Figure 16.4). This is implemented using the wait() - notifyAll() mechanism of Java [65][43]. Once these threads wake up, they see that the termination flag is set. This results in the get() method of the receivers throwing a TerminateProcessException (a runtime exception in Ptolemy II). This exception is never caught in any of the actor methods and is eventually caught by the process thread. The thread catches this runtime exception, calls the wrapup() method of the actor and finishes its execution. Eventually after all threads catch this exception and finish executing, the simulation ends.

16.3.5 Mutations of a Graph

The PN domain in Ptolemy II allow graphs to mutate during execution. This implies that old processes or channels can disappear from the graph and new processes and channels can be created during the simulation.

Though other domains, like SDF, also support mutations in their graphs, there is a big difference between the two. In domains like SDF, mutations can occur only between iterations. This keeps the simulation determinate as changes to the topology occur only at a fixed point in the execution cycle. In PN, the execution of a graph is not centralized, and hence, the notion of an iteration is quite difficult to define. Thus, in PN, we let mutations happen as soon as they are requested, if they are queued with the

```
public synchronized void setFinish() {
    _terminate = true;
    notifyAll();
}
```

FIGURE 16.4. `setFinish()` method of PNQueueReceiver

director rather than the manager. This is the behavior of PNDirector. (TimedPNDirector performs mutations only when there is a timed-deadlock. Mutations in this form are deterministic.) The point in the execution where mutations occur would normally depend on the schedule of the underlying Java threads. Under certain conditions where the application can guarantee a fixed point in the execution cycle for mutations, or where the mutations are localized, they can still be determined.

In case of TimedPNDirector, all mutations are deterministic as request to perform mutations is not processed unless there is a timed deadlock. Since occurrence of a timed deadlock does not depend on the schedule of the underlying threads, the mutations are completely deterministic.

An actor can request a mutation by creating an instance of a class derived from ptolemy.kernel.event.ChangeRequest. It should override the method execute() and include the commands that it wants to use to perform mutations in this method.

16.4 Technical Details

There are two main issues that a developer should be aware of while extending PN. The first one is to get the mutual exclusion right and the second is to avoid undetected deadlocks.

16.4.1 Mutual Exclusion using Monitors

In PN, threads interact in various ways for message passing, deadlock detection, etc. This requires various threads to access the same data structures. Concurrency can easily lead to inconsistent states as threads could access a data structure while it is being modified by some other thread. This can result in race conditions and undesired deadlock [4]. For this, Java provides a low-level mechanism called a *monitor* to enforce mutual exclusion. Monitors are invoked in Java using the *synchronized* keyword. A block of code can be *synchronized* on a monitor lock as follows:

```
synchronized (obj) {
   ... //Part of code that requires exclusive lock on obj.
}
```

This implies that if a thread wants to access the synchronized part of the code, then it has to grab an exclusive lock on the monitor object, `obj`. Also while this thread has a lock on the monitor, no thread can access *any* code that is synchronized on the same monitor.

There are many actions (like mutations) that could affect the consistency of more than one object, such as the director and receivers. Java does not provide a mechanism to acquire multiple locks simultaneously. Acquiring locks sequentially is not good enough as this could lead to deadlocks. For example, consider a thread trying to acquire locks on objects *a* and *b* in that order. Another thread might try to obtain locks on the same objects in the opposite order. The first thread acquires a lock on *a* and stalls to acquire a lock on *b*, while the second thread acquires a lock on *b* and waits to grab a lock on *a*. Both threads stall indefinitely and the application is deadlocked.

The main problem in the above example is that different threads try to acquire locks in conflicting orders. One possible solution to this is to define an order or hierarchy of locks and require all threads to grab the locks in the same top-down order [43]. In the above example, we could force all the threads to acquire locks in a strict order, say *a* followed by *b*. If all the code that requires synchronization respects this order, then this strategy can work with some additional constraints, like making the order on locks immutable. Although this strategy can work, this might not be very efficient and can make the code a

lot less readable. Also Java does not permit an easy and straightforward way of implementing this.

We follow a similar but easier strategy in the PN domain of Ptolemy II. We define a three level strict hierarchy of locks with the lowest level being the director, the middle level being the various receivers and the highest level being the *workspace*. The rule that all threads have to respect after acquiring their first lock is to never try acquiring a lock at a higher or at the same level as their first lock. Specifically, a block of code synchronized on the director should not try to access a block of code that requires a lock on either the workspace or any of the receivers. Also, a block of code synchronized on a receiver should not try to call a block of code synchronized on either the workspace or any other receiver.

Some discussion about these locks in PN is presented in the following section.

16.4.2 Hierarchy of Locks

The highest level in the hierarchy of locks is the Workspace which is a class defined specifically for this purpose. This level of synchronization though is quite different from the other two forms. This synchronization is modeled explicitly in Ptolemy II and is another layer of abstraction based on the Java synchronization mechanism. The principle behind this mechanism is that if a thread wants to read the topology, then it wants to read it only in a consistent state. Also if a thread is modifying the topology, then no other thread should try to read the topology as it might be in an inconsistent state. To enforce this, we use a reader-writer mechanism to access the workspace (see the Kernel chapter). Any thread that wants to read the topology but does not modify it, requests a read access on the workspace. If the thread already has a read or write access on the workspace, it gets another read access immediately. Otherwise if no thread is currently modifying the topology, and no thread has requested a write access on the workspace, the thread gets the read access to the workspace. If the thread cannot get the read access currently, it stalls until it gets it. Similarly, if a thread requests a write access on the workspace, it stalls until all other threads give up their read and write access to the workspace. Thus though a thread does not have an exclusive lock on the workspace, the above mechanism provides a mutual exclusion between the activities of reading the topology and modifying the topology. This way of synchronizing on the workspace is distinctly different from possessing an exclusive lock on the workspace.

Once a thread has a read or write access on the workspace, it can call methods or blocks of code that are synchronized on a single receiver or the director. The receivers form the next level in the hierarchy of locks. These receivers are once again accessed by different threads (the reader and the writer to the queue) and need to be synchronized. For example, a writer thread might try to write to a receiver while another token is being read from it. This could leave the receiver in an inconsistent state. The state of a receiver might include information about the number of tokens in the queue, the information about any process blocked on a read or a write to the receiver and some other information. These methods or blocks of code accessing and modifying the state of the receivers, are forced to get an exclusive lock on the receiver. These blocks might call methods that require a lock on the director, but do not call methods that require a lock on any other receiver.

The lower most lock in the hierarchy is the PNDirector object. There are some internal state variables, such as the number of processes blocked on a read, that are accessed and modified by different threads. For this, the code that modifies any internal state variable should not let more than one thread access these variables at the same time. Since access to these variables is limited to the methods in director, the blocks of code modifying these state variables obtain an exclusive lock on the director itself. These blocks should not try to access any block of code that requires an exclusive lock on the receivers or requires a read or a write access on the workspace.

16.4.3 Undetected Deadlocks

Undetected deadlocks should be avoided while extending the PN domain in Ptolemy II. We discuss a significant but subtle issue that a developer should be aware of when trying to extend the PN domain. This concerns the release of locks from a suspended thread.

In Java, when a thread with an exclusive lock on multiple objects suspends by calling `wait()` on an object, it releases the lock only on that object and does not release other locks. For example, consider a thread that holds a lock on two objects, say *a* and *b*. It calls `wait()` on *b* and releases the lock on *b* alone. If another thread requires a lock on *a* to perform whatever action the first thread is waiting for, then deadlock will ensue. That thread cannot get a lock on *a* until the first thread releases its exclusive lock on *a*, and the first thread cannot continue until the second thread gets the lock on *a* from the first and performs whatever action it is waiting for.

This sort of scenario is currently avoided in PN, by following some simple rules. The first of them being that a method or block synchronized on the director never calls wait() on any object. Thus once a thread grabs a lock on director, it is guaranteed to release it. The second is that a block of code with an exclusive lock on a receiver does not call the wait() method on the workspace. (Note that the code should never synchronize directly on the workspace object and should always use the read and write access mechanism.) The third rule is that a thread should give up all the read permissions on the workspace before calling the wait() method on the receiver object. Note that in case of workspace, we require this because of the explicit modeling of mutual exclusion between the read and write activities on the workspace. If a thread does not release the read permissions on the workspace and suspends, while the second thread requires a write access on the workspace to perform the action that the first thread is waiting for, a deadlock results. Also to be in a consistent state with respect to the number of read accesses on the workspace, the thread should regain those read accesses after returning from the call to the wait() method. For this a wait(Object obj) method is provided in the class Workspace that releases all the read accesses to the workspace, calls wait() on the argument obj, and regains all the read accesses on waking up.

The above rules guarantee that a deadlock does not occur in the PN domain because of contention for various locks.

References

[1] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] G. A. 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] 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.

[4] G. R. Andrews, *Concurrent Programming — Principles and Practice*, Addison-Wesley, 1991.

[5] R. L. Bagrodia, "Parallel Languages for Discrete Event Simulation Models," *IEEE Computational Science & Engineering*, vol. 5, no. 2, April-June 1998, pp 27-38.

[6] 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.

[7] 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.

[8] 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.

[9] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, 19(2):87-152, 1992.

[10] 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.

[11] Randy Brown, "CalendarQueue: A Fast Priority Queue Implementation for The Simulation Event Set Problem", Communications of the ACM, October 1998, Volume 31, Number 10.

[12] V. Bryant, "Metric Spaces," Cambridge University Press, 1985.

[13] 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).

[14] A. Burns, *Programming in OCCAM 2*, Addison-Wesley, 1988.

[15] 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.

[16] L. Cardelli, *Type Systems*, Handbook of Computer Science and Engineering, CRC Press, 1997.

[17] 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.

[18] 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.

[19] B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[20] 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/)

[21] 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)

[22] P. A. Fishwick, *Simulation Model Design and Execution: Building Digital Worlds*, Prentice Hall, 1995.

[23] M. Fowler and K. Scott, *UML Distilled*, Addison-Wesley, 1997.

[24] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, no. 10, October 1990, pp 30-53.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading MA, 1995.

[26] C. W. Gear, "Numerical Initial Value Problems in Ordinary Differential Equations," Prentice Hall Inc. 1971.

[27] 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.

[28] 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)

[29] 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)

[30] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.*, vol 8, pp. 231-274, 1987.

[31] P. G. Harrison, "A Higher-Order Approach to Parallel Algorithms," *The Computer Journal*, Vol. 35, No. 6, 1992.

[32] 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.

[33] 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.

[34] M. G. Hinchey and S. A. Jarvis, *Concurrent Systems: Formal Developments in CSP*, McGraw-Hill, 1995.

[35] 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.

[36] C. A. R. Hoare, “Communicating Sequential Processes,” *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[37] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[38] 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

[39] D. Jefferson, Brian Beckman, et al, “Distributed Simulation and the Time Warp Operating System,” UCLA Computer Science Department: 870042, 1987.

[40] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.

[41] G. Kahn and D. B. MacQueen, “Coroutines and Networks of Parallel Processes,” *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[42] P. Laramie, R.S. Stevens, and M.Wan, “Kahn process networks in Java,” ee290n class project report, Univ. of California at Berkeley, 1996.

[43] D. Lea, *Concurrent Programming in Java™*, Addison-Wesley, Reading, MA, 1997.

[44] 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/Interaction-FSM/)

[45] 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, to appear, 1998. Also UCB/ERL Memorandum M98/7, March 4th 1998.(http://ptolemy.eecs.berkeley.edu/publications/papers/98/realtime)

[46] 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/)

[47] E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing,” *IEEE Trans. on Computers*, January, 1987.

[48] 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)

[49] E. A. Lee and A. Sangiovanni-Vincentelli, “A Framework for Comparing Models of Computation,”, to appear, *IEEE Transactions on CAD*, (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/)

[50] M. A. Lemkin, *Micro Accelerometer Design with Digital Feedback Control*, Ph.D. dissertation, University of California, Berkeley, Fall 1997.

[51] 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/)

[52] J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S. Sastry, and E. A. Lee, "Hierarchical Hybrid System Simulation", submitted to 1999 38th IEEE Conference on Decision and Control (CDC'99), Phoenix, Arizona.

[53] D. C. Luckham and J. Vera, “An Event-Based Architecture Definition Language,” *IEEE Transactions on Software Engineering*, 21(9), pp. 717-734, September, 1995.

[54] 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.

[55] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 1993.

[56] B. Meyer, *Object Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.

[57] R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[58] R. Milner, *“A Calculus of Communicating Systems”*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[59] R. Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences 17, pp. 384-375, 1978.

[60] J. Misra, “Distributed Discrete-Event Simulation,” *Computing Surveys*, vol. 18, no. 1, March 1986, pp 39 - 65.

[61] 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/)

[62] 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.

[63] NASA Office of Safety and Mission Assurance, *Software Formal Inspections Guidebook*, August 1993 (http://satc.gsfc.nasa.gov/fi/gdb/fitext.txt).

[64] A. R. Newton and A. L. Sangiovanni-Vincentelli, “Relaxation-Based Electrical Simulation,” *IEEE Tr. on Electronic Devices*, Vol. ed-30, No. 9, Sept. 1983.

[65] S. Oaks and H. Wong, *Java Threads*, O’Reilly, 1997.

[66] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.

[67] J. K. Ousterhout, *Scripting: Higher Level Programming for the 21 Century*, IEEE Computer magazine, March 1998.

[68] 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/)

[69] 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.

[70] Rational Software Corporation, *UML Notation Guide*, Version 1.1, September 1997, http://www.rational.com/uml/resources/documentation/notation

[71] 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/)

[72] 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.

[73] A. J. Riel, *Object Oriented Design Heuristics*, Addison Wesley, 1996.

[74] J. Rowson and A. Sangiovanni-Vincentelli, “Interface Based Design,” *Proc. of DAC ‘97*.

[75] J. Rumbaugh, *et al. Object-Oriented Modeling and Design* Prentice Hall, 1991.

[76] J. Rumbaugh, *OMT Insights*, SIGS Books, 1996.

[77] S. Saracco, J. R. W. Smith, and R. Reed, *Telecommunications Systems Engineering Using SDL*, North-Holland - Elsevier, 1989.

[78] 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/)

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.

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.

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.

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.

entity ..... A node in a Ptolemy II clustered graph.

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.

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.

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

Symbols

- in UML 1-17
! in CSP 14-5
in UML 1-17
*charts 1-6
+ in UML 1-17
? in CSP 14-5
@exception 4-19
@param 4-19
_createRunControls() method
    PtolemyApplet class 2-8
_director member 2-4
_go() method
    DEApplet class 2-11
_manager member 2-4
_newReceiver() method
    IOPort class 6-8
_toplevel member 2-4

A

absolute type constraint 4-4
AbsoluteValue actor 3-9
abstract class 1-19
abstract syntax 1-13, 5-1, G-1
abstract syntax tree 7-13
abstraction 5-8
accelerometer 11-14
acquaintances 6-2
action methods 3-6, 4-11, 6-11, G-1
active processes 14-13
actor 6-11, G-1
Actor interface 1-13, 6-11, 6-12
actor libraries 2-13
actor package 1-9, 3-2, 6-2
actor.gui package 1-9, 2-2, 3-1, 3-3, 3-4, 3-12, 3-13
actor.lib package 1-9, 3-1, 3-2, 3-10, 3-12, 4-5
actor.process package 1-10, 6-20, 6-21
actor.sched package 1-10, 6-20, 6-21
actor.util package 1-11, 6-9, 6-10
actors 1-4, 4-1, 6-1, 6-2
acyclic directed graphs 8-1
add() method
    Token class 7-3
addChangeListener() method
    Manager class 6-18
addExecutionListener() method
    Manager class 6-18
AddSubtract actor 3-3, 3-9, 13-5
addToScope() method
    Variable class 7-10
ADL 1-3
ADS 1-2
advancing time
    CSP domain 14-5
aggregation association 5-2
aggregation UML notation 1-19
allowLevelCrossingConnect() method
    CompositeEntity class 5-9
analog circuits 1-5
analog electronics 1-1
Andrews 14-1
animated plots 10-1
ANYTYPE 9-3
anytype G-1
anytype particle 1-15
applet 2-1, G-1
applets 1-9, 2-1, 10-6
appletviewer 2-3, 2-13, 10-6
application G-1
application framework 6-1
applications 1-9, 2-1
arc 5-2
architecture 1-3
architecture description languages 1-3
architecture design language 1-3
archive 2-13
arithmetic operators 7-3, 7-10
arraycopy method 13-6
ArrayFIFOQueue class 13-6
artificial deadlock 15-2, 16-2
associations 1-19
AST 7-13
ASTPtBitwiseNode class 7-15
ASTPtFunctionalIfNode class 7-15

ASTPtFunctionNode class 7-14, 7-15
ASTPtLeafNode class 7-15
ASTPtLogicalNode class 7-15
ASTPtMethodCallNode class 7-15
ASTPtProductNode class 7-15
ASTPtRelationalNode class 7-16
ASTPtRootNode class 7-15
ASTPtSumNode class 7-15
ASTPtUnaryNode class 7-16
asynchronous communication 6-9, 16-2
asynchronous message passing 1-7, 6-3
atomic actions 1-4
atomic actor G-1
atomic communication 14-2
AtomicActor class 1-13, 6-11, 6-12
attribute G-1
Attribute class 4-9, 5-6, 7-6
attributeChanged() method
NamedObj class 4-7, 7-9
Poisson actor 4-7
Scale actor 4-9
attributes 1-12, 1-17, 7-6
attributeTypeChanged() method
NamedObj class 4-8, 7-9
audio 1-11
Average actor 2-17, 3-14, 4-11, 4-13, 4-14, 4-15, 4-16
axes for plots 10-1

B

Backus normal form 7-13
balance equations 13-2
bang in CSP 14-5
bar graphs 10-1
Bars command 10-5
base class 1-18
bash 2-4
BDF 1-7
begin() method
Ptolemy 0 6-13
Bernoulli actor 3-11
bidirectional ports 6-5, 6-11
bison 7-13
bitwise operators 7-11
block G-1
block diagrams 1-9, 2-1
block-and-arrow diagrams 1-4
blocked processes 14-13
blocking reads 15-1, 16-2
blocking receive 14-1
blocking send 14-1
blocking writes 15-1, 16-2
BNF 7-13
boolean dataflow 1-7
BooleanMatrixToken class 7-2
BooleanToken class 7-2
bottom-up parsers 7-13
bounded buffering 16-1
bounded memory 13-1, 16-2
boundedness 1-7
broadcast() method 6-5
DEIOPort class 12-3, 12-6, 12-7
browser 2-1, G-1, G-3
bubble-and-arc diagrams 1-4
buffer 6-9
bus 6-3
bus contention 14-5
bus widths and transparent ports 6-8
busses, unspecified width 6-7

C

C 1-2
C++ 1-2
calculus of communicating systems 1-4, 14-2
calendar queue 1-5, 1-11, 12-3
CalendarQueue class 6-10, 6-11
capitalization 4-17
CCS 1-4, 14-2
CDO 14-3, 14-19
Chandy 15-1
change listeners 5-19
change requests 16-3
changed() method
QueryListener interface 2-11
ChangeRequest class 5-19
channel 6-2, G-1
channels 4-2
check box entry 2-17
checkTypes() method
TypedCompositeActor class 9-6
chooseBranch() method
CSPActor class 14-9
CIF 14-3, 14-10, 14-19
circular buffer 13-6
class diagrams 1-17
class names 1-20, 4-17
CLASSPATH environment variable 2-5
clipboard 10-6
Clock actor 2-9, 3-11
Clock class 2-4
clone() method
in actors 4-9

NamedObj class 5-11
Object class 4-9, 7-3
Scale actor 4-10
cloning 5-11
cloning actors 4-9
clustered graph G-1
clustered graphs 1-11, 1-13, 5-1
code duplication 4-1
code generation G-1
codebase 2-3
coding conventions 4-16
coin flips 3-11
Color command 10-4
COMMENT HTML tag 2-3
comments 4-17
expression language 7-11
communicating sequential processes 1-4, 1-13, 14-1
communication networks 12-1
communication protocol 6-2, 6-8
Commutator actor 3-14
compile-time exception 4-18
compiling applets 2-5
complete partial orders 8-1
completion time 15-3
complex numbers 1-11
ComplexMatrixToken class 7-2
ComplexToken class 7-2
component interactions 1-3
component-based design 3-1, 4-1
ComponentEntity class 1-12, 5-6, 5-7
ComponentPort class 1-12, 5-6, 5-7
ComponentRelation class 1-12, 5-6, 5-7
components 1-2, 1-15
composite actor G-1
Composite design pattern 1-19, 5-6
composite opaque actor 6-14
CompositeActor class 1-13, 6-11, 6-12
CompositeEntity class 1-13, 5-6, 5-7
concrete class 1-19
concrete syntax 5-1, G-1
concurrency 1-2
concurrent computation 6-1
concurrent design 1-13
concurrent finite state machines 1-6
concurrent programming 14-6
conditional 7-11
conditional communication 14-9
conditional do 14-3
conditional if 14-3
ConditionalReceive class 14-9
ConditionalSend class 14-9
connect() method
CompositeEntity class 5-9
connection 5-2, G-1
conservative blocking 15-5
consistency 5-3
Const actor 3-11
constants
expression language 7-11
constants in expressions 7-16
constraints on parameter values 4-7
constructive models 1-1
constructors
in UML 1-17
container 5-5, G-1
containment 1-19
contention 14-5
continuous time modeling 1-4
continuous-time modeling 1-13
continuous-time systems 3-6
contract 9-5
convert() method
Token class 9-9
Token classes 7-5
CORBA 1-15, 2-1
cos() method
Math class 2-12
cosine 3-9
CPO interface 8-4
CPOs 8-1
CQComparator interface 6-10
CrossRefList class 5-6
CSP 1-4, 14-1
CSP domain 3-5, 6-10
CSPActor class 14-9
CSPDirector class 14-11
CSPReceiver class 14-11
CT 1-4
CT domain 3-6
current time 3-3, 12-1, 16-3
CurrentTime actor 3-11

D

DAG 12-7
dangling ports
SDF domain 13-6
dangling relation 6-5, G-1
data encapsulation 7-1
data package 1-11, 7-1
data polymorphic 7-4
data polymorphism 3-1, 4-1, G-1
data rates 13-1

data.expr package 1-11
dataflow 6-9, 12-7, 13-1, 15-1
DataSet command 10-5
DCOM 1-15
DDE 1-6, 15-1
DDE domain 12-12
DDES 15-1
DDF 1-7
DE 1-5, 12-1
DE domain 3-6
DEActor class 12-3, 12-4
deadlock 1-7, 5-15, 5-18, 13-1, 16-2
CSP domain 14-4
DDE domain 15-1
DEApplet class 2-1, 2-4
DECQEventQueue class 12-4
DEDirector class 12-3, 12-4
deep traversals 5-8, G-1
deepContains() methodNamedObj class 5-9
deepGetEntities() method
CompositeEntity class 5-9, 6-18
DEEvent class 12-4
DEEventQueue interface 12-4
DEEventTag class 12-4
DefaultExecutionListener class 6-12, 6-18
defaultIterations parameter
SDF applets 2-9
defaultStopTime parameter
DE applets 2-8
DEIOPort class 12-3, 12-4, 12-6, 12-7, 12-8
delay 12-2, 12-3
CSP domain 14-4
PN domain 16-3
SDF domain 13-3
Delay actor 12-8
DE domain 12-3
delay actors
DDE domain 15-2
delay() method
CSPActor class 14-11
delayed processes 14-13
delayTo() method
DEIOPort class 12-8
delta functions 1-5
delta time 1-5, 15-4
demultiplexor actor 6-3
dependency loops 7-14
DEReceiver class 12-4
derived class 1-18
description() method 5-11, 5-12
design 1-1
design patterns 1-15
determinacy 1-7, 6-9
determinism 14-1, 16-1
deterministic 12-7
DEThreadActor class 12-12
DETransformer class 12-3, 12-8
digital electronics 1-1
digital hardware 1-5, 12-1
Dijkstra 14-6
dining philosophers 14-5, 14-6
Dirac delta functions 1-5
directed acyclic graph 12-7
directed graphs 5-2, 8-1
DirectedAcyclicGraph class 8-2, 8-4
DirectedGraph class 8-2, 8-3
director 1-13, 6-8, 6-13, 6-14, G-2
Director class 1-13, 6-8, 6-11, 6-12
disconnected graphs
SDF domain 13-5
disconnected port 6-3, G-2
discrete event domain 12-1
discrete-event domain 1-5
discrete-event model of computation 6-11
discrete-event modeling 1-13
discrete-time domain 1-6
distributed discrete event 15-1
distributed discrete event systems 15-1
distributed discrete-event domain 1-6
distributed models 1-6
distributed time 15-1
Distributor actor 3-14, 6-3, 6-9
divide() method
Token class 7-3, 7-15
domain 6-1, G-2
domain polymorphism 3-1, 4-1, 7-4, G-2
domain-polymorphism 1-15
domains 1-9, 1-13
domains package 1-11, 1-13
domains.de.kernel package 12-3
domains.de.lib package 12-3
doneReading() method
Workspace class 5-18
doneWriting() method
Workspace class 5-18
DoubleCQComparator interface 6-10
DoubleMatrixToken class 7-2
doubles 7-11
DoubleToken class 7-2
DT 1-6
dynamic dataflow 1-7
dynamic networks 6-11

E

E 7-11
e 7-11
edges 8-1
EDIF 5-1
EMBED HTML tag 2-3, 2-4
embedded systems 1-1
encapsulated postscript 10-6
entities 1-4, 5-1
entity G-2
Entity class 1-12, 5-2, 5-3
environment variable 2-4
EPS 10-6
equals() method
    Token class 7-3, 7-16
Eratosthenes 14-6, 14-7
evaluateParseTree() method
    ASTPtRootNode class 7-14
evaluation of expressions 7-8
event 1-5
event queue 12-1
event subpackage of kernel 5-19, 6-18
events 12-1
exceptions 4-17
exceptions in applets 2-6
executable entities 6-1
Executable interface 1-13, 6-11, 6-12
executable model 1-13
executable models 1-1
execute() method
    ChangeRequest class 5-19
execution 3-6, 6-11, G-2
executionError() method
    ExecutionListener interface 6-18
ExecutionEvent class 6-12
executionFinished() method
    ExecutionListener interface 2-17, 6-18
ExecutionListener class 6-12
ExecutionListener interface 6-18
executive director 6-13, 6-18, G-2
Expression actor 3-14, 4-12
expression evaluation 7-13
expression language 1-6, 1-11, 7-10
    extending 7-16
expression parser 7-13
expressions 2-12

F

fail-stop behavior 9-2
fairness 14-6
false 7-11
FBDelay actor 15-4
FIFO 6-2, 13-6, 16-1
FIFO Queue 1-11
FIFOQueue class 6-2, 6-9, 6-10, 13-6
file format for plots 10-3
file formats 1-15
FileWrite actor 3-12
fillOnWrapup parameter
    Plotter actor 3-12
finally keyword 5-18
finish() method
    Manager class 6-17
finished flag 14-15
finite buffer 6-9
finite state machines 1-9
finite-state machine domain 1-6
fire() method
    actor interface 12-2
    Average actor 4-15
    CompositeActor class 6-19
    Director class 6-19
    Executable interface 3-6, 6-11
    in actors 4-11
fireAt() method
    DEActor class 12-9
    DEDirector class 12-6
    Director class 3-10, 4-15, 12-1, 12-3, 12-6, 12-15
firing vector 13-2
firingCountLimit parameter
    SequenceSource actor 3-10, 4-15
first-in-first-out 16-1
fixed-point 1-4
fixed-point computation 7-6
fixed-point semantics 3-6
fixed-point simulations 1-16
floating-point simulations 1-16
formatting of code 4-16
fractions 1-11
FSM 1-6
full name 5-5
functional actors 3-6
functional if...then...else... 7-11
functions
    expression language 7-11

G

galaxy 5-10, G-2
Gaussian actor 3-11
General type 7-5
general type 9-11

generalize 1-18
get() method
IOPort class 6-2
Receiver interface 6-2
getArray() method
SDFIOPort class 13-6
SDFReceiver class 13-6
getAttribute() method
NamedObj class 5-6
getAttributes() method
NamedObj class 5-6
getContainer() method
Nameable interface 5-5
getCurrentTime() method
DEActor class 12-9
Director class 4-15
getDirector() method
Actor interface 6-14
getElementAt() method
MatrixToken classes 7-3
getInsideReceivers() method
IOPort class 6-20
getReadAccess() method
Workspace class 5-18
getReceivers() method
IOPort class 6-20
getRemoteReceivers() method 6-11
IOPort class 6-8
getState() method
Manager class 6-18
getTypeTerm() method
TypedIOPort class 9-7
getValue() method
ObjectToken class 7-3
getWidth() method
IORelation class 6-8
getWriteAccess() method
Workspace class 5-18
grammar rules 7-13
Graph class 8-2, 8-3
graph package 1-11, 8-1
graphical elements 2-6
graphical syntaxes 5-11
graphical user interface 3-1, 3-3
graphs 8-1
Grid command 10-4
guarded communication 6-11, 14-2, 14-9
guards 1-6
GUI 3-1, 3-3
gui package 1-11

H

hardware 1-1
hardware bus contention 14-5
Harel, David 1-6
hashtable 1-5
hasRoom() method
IOPort class 6-20
Hasse 8-4
Hasse diagram 8-4
hasToken() method
IOPort class 6-20
heterogeneity 1-13, 5-12, 6-18
Hewlett-Packard 1-2
hiding 5-8
hierarchical concurrent finite state machines 1-9
hierarchical heterogeneity 5-12, 6-18
hierarchy 5-6
higher node 8-4
HistogramPlotter actor 3-12
history 6-9
Hoare 14-1, 14-5
homogeneous actors 13-5
HTML 2-2, 4-18
HTTP 2-13
hybrid systems 1-6

I

i 7-11
IE 2-3
if...then...else... 7-11
IllegalActionException class 4-9
IllegalArgumentException class 7-14
image processing 1-11
immutability
tokens 7-1
Immutable 5-17
immutable 5-5
immutable property G-2
imperative semantics 1-2
implementation 2-1
implementing an interface 1-19
import 1-17
Impulses command 10-5
in CSP 14-3
incomparable 7-5
incomparable types 4-4
inconsistent models 13-2
indentation 4-16
Inequality class 8-3, 8-4, 9-6
InequalitySolver class 8-4
InequalityTerm interface 8-2, 8-4, 9-6

information-hiding 5-12
inheritance 1-18, 4-1
initial output tokens 4-11
initial token 13-3
initialize() method
Actor interface 12-2
Average actor 4-11
Director class 6-14
Executable interface 3-6, 6-11
in actors 4-11
input port 6-2
inputs
transparent ports 6-6
inside links 5-8
inside receiver 6-20
inspection paradox 2-17
instantaneous reaction 12-2
integers 7-11
intellectual property 1-6
interface 1-19
Internet Explorer 2-3
interoperability 1-2, 1-15
interpreter 1-11
IntMatrixToken class 7-2
IntToken class 7-2
invalidateResolvedTypes() method
Director class 4-8
invalidateSchedule() method
DEDirector class 12-5
Director class 4-7
IOPort class 6-2
IORelation class 6-2, 6-3
Irix 2-2
isAtomic() method
CompositeEntity class 5-8
isInput() method 6-11
isOpaque() method
ComponentPort 5-12
CompositeActor class 6-13, 6-18
CompositeEntity class 5-8, 6-6
isOutput() method 6-11
isWidthFixed() method
IORelation class 6-8
iteration 3-6, 6-11, G-2
iterations 4-11
iterations parameter
SDF applets 2-9
SDFDirector class 13-3

J

j 7-11
jar files 2-13
Java 1-2
Java Archive File 2-13
Java Plug-In 2-2
Java RMI 1-15
Java Runtime Environment 2-2
java.lang.Math 7-16
java.lang.Void.TYPE 7-5
JavaCC 7-13
Javadoc 4-5, 4-18
JDK 2-2
Jefferson 15-5
JIT 2-16
JJTree 7-13
JRE 2-2
just-in-time compiler 2-16

K

Kahn process networks 1-7, 6-9, 15-1
kernel package 1-11
kernel.event package 1-11, 6-18
kernel.util package 1-11, 4-17, 6-11

L

LALR(1) 7-13
lattice 7-4
lattices 8-1
layout manager 2-7
lazy 4-8
LEDA 8-1
level-crossing links 5-8, 5-9
lexical analyzer 7-13
lexical tokens 7-13
liberalLink() method
ComponentPort class 5-9
Lines command 10-5
lingering processes 2-16
link 5-2, 5-3, G-2
link() method
Port class 5-9
Linux 2-2
literal constants 7-11
liveness 1-13, 14-6
LL(k) 7-13
local director 6-13, 6-18
lock 5-16, 14-16
logarithmic axes for plots 10-4
logical boolean operators 7-11
long integers 7-11
LongMatrixToken class 7-2
LongToken class 7-2

Lorenz system 11-13
lossless type conversions 7-9
Lotos 1-4, 14-5
lower node 8-3

M

M/M/1 Queue 14-6
mailbox 6-8
Mailbox class 6-2, 6-8
make install 2-14
makefiles 2-14
managed ownership 5-5
manager 6-13, 6-17, G-2
Manager class 1-13, 6-12, 6-17
managerStateChanged() method
ExecutionListener interface 6-18
Marks command 10-4
Math class 2-12
math functions 7-16
math package 1-11
mathematical graphs 5-2, 8-1
Matlab 1-2
matrices 1-11
matrix tokens 7-3
MatrixToken class 7-2
Maximum actor 3-9
mechanical components 1-5
mechanical systems 1-5
media package 1-11
Mediator design pattern 5-2
MEMS 1-1, 1-5, 11-14
Message class 10-2
message passing 6-2
methods
expression language 7-12
microelectromechanical systems 1-1
microwave circuits 1-5
Milner 14-2
Minimum actor 3-9
Misra 15-1
mixed signal modeling 1-5
ML 1-15
MoC 14-1
modal model 1-5
modal models 1-6
model G-2
model of computation 1-2, 6-1, 6-2, G-2
model time 12-1, 14-4, 16-3
modeling 1-1
models of computation
mixing 6-18
modulo() method
Token class 7-3, 7-15
monitor 5-15
monitors 1-13, 1-15, 14-15
monotonic functions 6-9
multiply() method
Token class 4-6, 7-3, 7-15
MultiplyDivide actor 3-9
multiport 4-2, 6-3, 6-9, G-2
multiports
SDF domain 13-5
mutability
CSP domain 14-6
mutation 1-11, 1-15, 6-18
mutations 5-19, 16-1, 16-3
DE domain 12-3, 12-11
mutual exclusion 5-15, 14-16

N

name 5-5
name server 6-11
Nameable interface 1-12, 4-17, 5-3, 5-5
NamedList class 5-6
NamedObj class 1-12, 4-9, 5-3, 5-5
NameDuplicationException class 4-9
naming conventions 1-20
NaT 7-5, 9-12
newReceiver() method
Director class 6-8
node classes (parser) 7-15
nodes 8-1
non-determinism 14-1
nondeterminism with rendezvous 6-10
nondeterministic choice 14-2
non-timed deadlock 15-2
notifyAll() method
Object class 14-16
null messages 15-2
Numerical type 7-5

O

OBJECT HTML tag 2-3, 2-4
object model 1-17
object modeling 1-15
object models 1-9
object-oriented concurrency 6-1
object-oriented design 3-1
ObjectToken class 7-1, 7-2, 7-3
OCCAM 14-5
Occam 1-4
ODE solvers 1-13

one() method
Token class 7-4
oneRight() method
MatrixToken classes 7-4
opaque G-2
opaque actors 6-13, 6-18
opaque composite actor 6-14, 6-20, G-2
opaque composite actors 1-15
opaque composite entities 5-12
opaque port 5-8
operator overloading 7-10
optimistic approach 15-5
overloaded 4-18
override 1-18

P

package G-2
package diagrams 1-17
package structure 1-9
packages 1-13
Pamela 16-3
parallel discrete event simulation 15-5
parameter 7-6, G-2
Parameter class 2-10, 4-9, 7-6
parameters 1-11, 2-9, 4-6
constraints on values 4-7
Parks 16-2
parse tree 7-13
parser 7-13
partial order 1-15
partial orders 8-1
partial recursive functions 1-6
particle G-2
pause() method
CSPDirector class 14-15
Manager class 6-17
PDES 15-5
period parameter
Clock actor 2-10
PI 7-11
pi 7-11
Placeable interface 2-6, 3-3
Plot class 2-7, 10-1, 10-2
plot package 1-11, 10-1
plot public member
Plotter class 2-7
PlotApplet class 10-1, 10-2
PlotApplication class 10-1, 10-2
PlotBox class 10-1, 10-2
PlotFrame class 10-1, 10-2
PlotLive class 10-1, 10-2
PlotLiveApplet class 10-1, 10-2
PlotPoint class 10-2
Plotter class 3-3
plotting 1-11
Plug-In 2-2
plug-in 2-16
Plug-In HTML Converter 2-3
pluginspage 2-3
PN 1-7, 16-1
PN domain 3-5
Poisson actor 3-11, 4-7
polymorphic actors 7-4
polymorphism 1-15, 3-1, 4-1, 9-3
data 7-4
domain 7-4
port G-2
Port class 1-12, 5-2, 5-3
ports 4-2, 5-1
postfire() method
actor interface 12-2
Average actor 4-15
CompositeActor class 6-17
DE domain 12-6
DEDirector class 12-15
Executable interface 3-6, 6-11
in actors 4-11
Server actor 12-11
precedences 1-5
precondition 4-18
prefire() method
Actor interface 12-2
CompositeActor class 6-19
DE domain 12-6
Executable interface 3-6, 6-11
in actors 4-11, 4-12
Server actor 12-9
prefix monotonic functions 1-7
prefix order 6-9
prime numbers 14-7
Print actor 3-12
priorities 14-7
priorities for events 12-7
priority queue 1-5, 1-11
private methods 1-17
process algebras 5-8
process domains 6-20
process level type system 1-15
Process Network Semantics 16-2
process networks 1-13, 6-9, 13-1, 16-1
process networks domain 1-7, 6-17
process-oriented domains 3-6

ProcessThread class 14-13
production rules 7-13
protected members and methods 1-17
protocol 6-2
protocols 3-1
PTII environment variable 2-4
ptmkmodel 2-15
Ptolemy Classic 1-13, G-2
Ptolemy II G-2
Ptolemy Project G-2
PtolemyApplet class 2-2, 2-6
PtParser 7-13
public members and methods 1-17
Pulse actor 3-11
pure event 12-1
pure signal 9-10
put() method
    Receiver interface 6-2
pxgraph program 10-3

Q

Quantizer actor 3-9
Query class 2-10
query in CSP 14-5
QueryListener interface 2-11
queue 6-9, 13-6, 14-6
queueing systems 12-1
QueueReceiver class 6-2, 6-3, 6-9

R

race conditions 5-15
Ramp actor 3-12
ranks for actors 12-7
Rapide 1-3
read blocked 16-2
read blocks 15-2
read/write semaphores 1-15
readers and writers 5-18
read-only workspace 5-19
real deadlock 14-4, 15-2, 16-2
receiver
    wormhole ports 6-20
Receiver interface 6-2
receiver time 15-2
Recorder actor 2-17, 3-12
reduced-order modeling 1-15
reflection 7-14, 7-16
registerClass() method
    PtParser class 7-16
registerConstant() method
    PtParser class 7-16
registerFunctionClass() method
    PtParser class 7-15
relation G-2
Relation class 1-12, 5-3
relational operators 7-11
relations 1-4, 5-1
relative type constraint 4-4
reloading applets 2-16
rendezvous 1-4, 3-5, 6-3, 6-9, 14-1, 14-17
repetitions vector 13-2
report() method
    PtolemyApplet class 2-6
reporting errors in applets 2-6
requestChange() method
    Director class 5-19, 6-18, 12-5
    Manager class 5-19, 6-18, 12-5
resolved type 9-3, G-2
resolveTypes() method
    Manager class 9-8
resource contention 14-5
resource management 14-1
resume() method
    CSPDirector class 14-15
    Manager class 6-17
re-use 3-1
ReuseDataSets command 10-5
rollback 11-12
RTTI 9-5
Rumbaugh 5-5
run() method
    Manager class 6-17
Runtime Environment 2-2
run-time exception 4-18
run-time type checking 9-2, 9-5
run-time type conversion 9-2
run-time type identification 9-5
RuntimeException interface 4-18

S

Saber 1-2, 1-5
safety 1-13
Scalar type 7-5
ScalarToken class 7-2
Scale actor 3-9, 4-6, 4-8, 4-9, 4-10
scatter plots 10-1
Scheduler class 13-5
schedulers 6-20
scheduling 6-13, 13-3, 13-5
schematic package 1-11
scope 7-8, 7-11
Scriptics Inc. 5-11

scripting 7-10
SDF 1-7, 13-1
SDFAtomicActor class 13-6
SDFDirector class 13-3
SDFIOPort class 13-6
SDFReceiver class 13-4, 13-6
SDFScheduler class 13-3, 13-5
SDL 1-7
semantics 1-2, 1-13
send() method
DEIOPort class 12-3, 12-6, 12-7, 12-8, 12-11
IOPort class 6-2
TypedIOPort class 9-9
sendArray() method
SDFIOPort class 13-6
SDFReceiver class 13-6
SequenceActor interface 3-3, 3-10, 12-3
SequencePlotter actor 3-13
SequencePlotter class 3-3
SequenceSource class 4-15
Server actor 12-3, 12-8
servlet G-3
servlets 2-1
setConnected() method
Plot class 2-13
setContainer() method
kernel classes 5-3
Port class 3-14
setCurrentTime 14-11
setCurrentTime() method
Director class 4-15, 14-11
setDeclaredType() method
TypedIOPort class 9-6
setExpression() method
Parameter class 2-11
Variable class 7-8
setImpulses() method
Plot class 2-13
setMarksStyle() method()
Plot class 2-13
setMultiport() method
IOPort class 6-3
setPanel() method
Placeable interface 2-6, 3-3
SetParameter class 5-21
setReadOnly() method
Workspace class 5-19
setSize() method
Plot class 2-7
setStopTime() method
DEDirector class 2-7, 12-2
setTitle() method
Plot class 2-13
setToken() method
Variable class 7-8
setTypeAtLeast() method
Variable class 7-9
setTypeEquals() method
Variable class 7-8
setTypeSameAs() method
Variable class 7-9
setWidth() method
IORelation class 6-3, 6-8
setXLabel() method
Plot class 2-13
shallow copy 4-9
sieve of Eratosthenes 14-6, 14-7
signal processing 16-1
simulation 1-1, 2-1
simulation time 12-1
Simulink 1-2, 1-5
simultaneous events 1-5, 12-1, 12-7
sin() method
Math class 2-12
Sine actor 3-9, 4-13
single port 4-2
Sink class 4-1
software 1-1
software architecture 1-3
software components 1-15
software engineering 1-15
Solaris 2-2
source actors 3-10, 3-12, 3-13, 4-15
Source class 4-1
spaces 4-17
specialize 1-18
Spice 1-5
spreadsheet 1-11
SR 1-7
standard out 3-12
star 5-10, G-3
starcharts 1-6
Start menu 2-16
start time 12-2
startRun() method
Manager class 6-17
state 1-6, G-3
Statecharts 1-6, 1-7
stateless actors 3-6
static schedule 6-17
static schedulers 6-20
static scheduling 13-1

static structure diagram 1-12, 3-2, 3-3, 5-2
static structure diagrams 1-17
static typing 9-1
StaticSchedulingDirector class 13-3
stem plot 2-13
stop time 12-2
stopFire() method
Executable interface 6-11
stopTime parameter
DE Applets 2-7
TimedSource actor 3-10
stream 6-3
string constants 7-11
StringToken class 3-5, 7-2
stringValue() method
Query class 2-11
subclass 1-18
subclass UML notation 1-18
subdomains 1-15
subpackage G-3
subtract() method
Token class 7-3, 7-15
superclass 1-18
symbol table 7-13
synchronized keyword 5-15, 14-16
synchronous communication 6-9
synchronous dataflow 1-7, 1-13, 13-1
synchronous dataflow domain 1-7
synchronous message passing 1-4, 6-3, 14-1
synchronous/reactive models 1-7
syntax 1-9

T

Tab character 4-17
telecommunications systems 1-5
terminate() method
Director class 14-15
Executable interface 6-11
Manager class 6-17
TerminateProcessException class 14-15
terminating processes
CSP domain 14-15
testable precondition 4-18
thermostat 11-15
thread actors
DE domain 12-11
thread safety 5-5, 5-13, 5-15
threads 1-13, 6-9
thread-safety 1-15
threshold crossings 1-5
time 1-2
CSP domain 14-4
DDE domain 15-1
PN domain 16-3
time deadlock 14-4
time stamp 1-5, 6-11, 12-1
DDE domain 15-2
Time Warp system 15-5
timed deadlock 15-3
TimedActor interface 3-3, 3-10, 4-15, 12-3
TimedPlotter actor 2-6, 3-13
TimedPlotter class 2-5, 3-3
TimedSource actor 4-15
TitleText command 10-3
token 4-2, G-3
Token class 3-3, 3-4, 4-13, 7-1, 7-2
tokenConsumptionRate parameter
port classes 13-5
tokenInitProduction parameter
port classes 13-5
tokenProductionRate parameter
port classes 13-5
tokens 3-2, 6-2
tokens, lexical 7-13
top level composite actor 6-17
top-down parsers 7-13
topological sort 8-2, 12-7
topology 5-1, G-3
topology mutations 5-19
transferInputs() method
DEDirector class 12-14
Director class 6-19
transferOutputs() method
Director class 6-19
Transformer class 3-3, 4-1, 4-5
transitions 1-6
transitive closure 8-2, 8-4
transparent G-3
transparent composite actor G-3
transparent entities 5-8
transparent port G-3
transparent ports 5-8, 6-6
trapped errors 9-1
trigger input
Source actor 3-10
true 7-11
tunneling entity 5-10
type changes for variables 7-9
type compatibility rule 9-2
type conflict 9-3
type constraint 4-4, 9-3
type constraints 4-4, 9-3, 9-6, G-3

type conversion 9-5
type conversions 7-4
type hierarchy 7-4
type lattice 7-4, 7-5
type resolution 1-15, 6-13, 9-3, G-3
type resolution algorithm 9-12
type system 4-4
    process level 1-15
type variable 9-4
Typeable interface 7-8
typeConstraints() method 9-8
TypedActor class 9-6
TypedAtomicActor 9-6
TypedAtomicActor class 3-2, 6-11
TypedCompositeActor 9-6
TypedCompositeActor class 6-11
TypedIOPort 9-6
TypedIOPort class 4-2, 6-2, 12-3
TypedIORelation class 6-2
TypedOIRelation 9-6
TypeLattice class 7-5
type-polymorphic actor 9-3
types of parameters 7-8
TypeTerm class 9-6

U

UML 1-9, 1-12, 1-17, 3-2, 3-3, 5-2
    package diagram 1-9
undeclared type 9-3, G-3
undirected graphs 8-1
unified modeling language 1-17
unimorphic 4-4
uniqueness of names 5-5
universe G-3
untrapped errors 9-1

V

variable 7-6
variables
    expression language 7-11
vectors 1-11
Verilog 1-5, 1-9
vertex 5-2
VHDL 1-5, 1-9
VHDL-AMS 1-2, 1-5
visual dataflow 1-9
visual syntax 1-9

W

wait() method
    Object class 14-16
    Workspace class 5-18
waitForDeadlock() method
    CSPActor class 14-11
waitForNewInputs() method
    DEThreadActor class 12-12
web server 2-1, G-1, G-3
width of a port 4-2, 6-3, G-3
width of a relation 6-3, G-3
width of a transparent 6-7
Windows 2-2
Windows NT 2-4
wireless communication systems 6-11
workspace 5-17
Workspace class 5-3, 5-5, 5-18
wormhole 1-13, 5-13, 6-14, 6-18, G-3
wrapup() method
    Actor interface 12-3
    Executable interface 3-6, 6-11
Wright 1-3
write blocked 16-2
write blocks 15-2

X

XLabel command 10-3
XLog command 10-4
XML 1-15, 2-1
XRange command 10-3
XTicks command 10-3
XYPlotter actor 3-13
XYPlotter class 3-3

Y

yacc 7-13
YLabel command 10-3
YLog command 10-4
YRange command 10-3
YTicks command 10-3

Z

Zeno condition 15-4
zero delay actors 12-2
zero() method
    Token class 7-4
zero-delay loop 12-8
zooming on plots 10-1