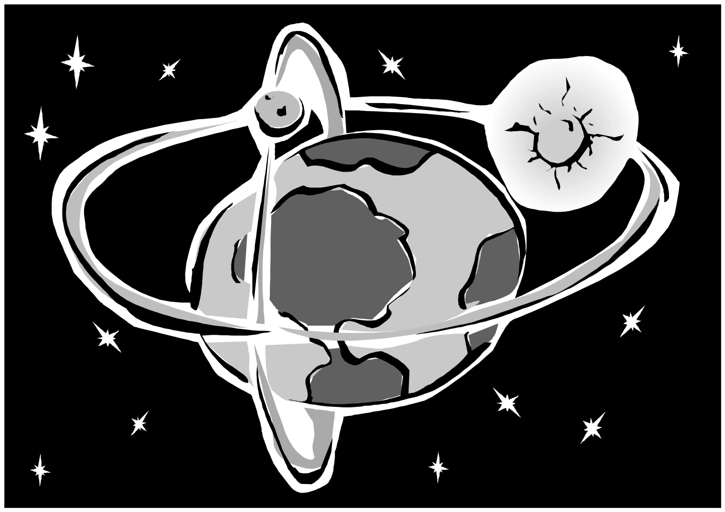
-PTOLEMY II -



HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

John Davis, II
Ron Galicia
Mudit Goel
Christopher Hylands
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

Document Version 0.1.1
February 12, 1999

Version 0.1 was ERL Technical Report UCB/ERL No. M98/71, dated November 23, 1998

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

Contents

1

Introduction

Edward A. Lee

1.1 Modeling and Design

The Ptolemy project studies heterogeneous modeling 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.*

1.2 Models of Computation

There are a rich variety of models of computation that deal with concurrency and time in different ways. In this section, we outline some of the most useful models for embedded systems. All of these will lend a semantics to the same bubble-and-arc, or block-and-arrow diagram shown in figure 1.1.

1.2.1 Differential Equations

One possible semantics for the syntax in figure 1.1 is that of differential equations. The arcs represent continuous functions of a continuum that is interpreted as time. The bubbles represent relations between these functions. The job of a simulator is to find a fixed-point, i.e., a set of functions that satisfy all the relations.

Differential equations are excellent for modeling analog circuits and many physical systems. This is the model of computation used in Simulink, Saber, and VHDL-AMS, and is closely related to that in Spice circuit simulators. However, they have disadvantages. Since they directly describe a physical system, they are tightly bound to an implementation, leaving few implementation options. Moreover, they are only applicable to relatively well-understood technologies, where lumped-parameter modeling is appropriate. They must be generalized to partial differential equations for less understood technologies, where solution techniques such as finite elements can be quite costly. For well-understood technologies, they can be expensive to simulate compared to digital representations of comparable functionality (and hence, they can be expensive to implement in software).

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, in which case there is a fundamental mismatch in models of computation. Joint modeling of a continuous subsystem with digital electronics is known as *mixed signal modeling*.

1.2.2 Difference Equations

Differential equations can be discretized to get difference equations, a commonly used model of computation in digital signal processing. This model of computation can be further generalized to support multirate difference equations. In either case, a global *clock* defines the discrete points at which signals have values (at the *ticks*).

Difference equations are considerably easier to implement in software, and hence leave more freedom of implementation. Their key weaknesses are the global synchronization implied by the clock, and the awkwardness of specifying irregularly timed events and control logic.

The *synchronous dataflow* (SDF) domain in Ptolemy II is extended with a model of time to model difference equations. Dataflow models are discussed below in section 1.2.7.

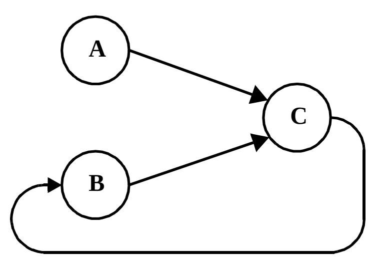


FIGURE 1.1. A single *syntax* (bubble-and-arc or block-and-arrow diagram) can have a number of possible *semantics* (interpretations).

1.2.3 Finite-State Machines

In FSMs, bubbles represent system *state* and arcs represent state *transitions*. The simple FSM model of computation is not concurrent. Execution is a strictly ordered sequence of state transitions.

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. Moreover, FSMs are easily mapped to either hardware or software implementations.

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 partially 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 [15]. FSMs have also been combined with differential equations, yielding the so-called *hybrid systems* model of computation [17].

A major (ongoing) result of the Ptolemy project has been to show that FSMs can be hierarchically combined with a huge variety of concurrent models of computation. We call the resulting formalism "*charts" (pronounced "starcharts") where the star represents a wildcard [14].

1.2.4 Synchronous/Reactive Models

In the synchronous/reactive (SR) model of computation [4], the arcs represent data values that are aligned with global clock ticks. Thus, they are discrete signals, as with difference equations, but unlike difference equations, a signal need not have a value at every clock tick. The bubbles 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 [6], Signal [5], Lustre [9], and Argos [24].

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.

1.2.5 Discrete-Event Models

In discrete-event (DE) models of computation, the arcs represent sets of *events* placed in time. An event consists of a *value* and *time stamp*. This model of computation is popular for specifying hardware and simulating telecommunications systems, and has been realized in a large number of simulation environments, simulation languages, and hardware description languages, including VHDL and Verilog. Unlike the SR model, there is no global clock tick, but like SR, differential equations, and difference equations, there is a globally consistent notion of time.

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. A key weakness is that it is relatively expensive to implement in software, as evidenced by the relatively slow simulators.

1.2.6 Synchronous Message Passing

In synchronous message passing, processes communicate in atomic, instantaneous actions called *rendezvous*. 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) [19] and Milner’s *calculus of communicating systems* (CCS) [27]. 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 weakness of rendezvous-based models is that maintaining determinacy can be difficult. Proponents of the approach, of course, cite the ability to model nondeterminacy as a key strength.

1.2.7 Asynchronous Message Passing

In asynchronous message passing, 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. There are several variants of this technique, but we focus on those that ensure determinate computation, namely Kahn process networks [20] and dataflow models.

In a process network (PN) model of computation, the arcs represent sequences of data values (tokens), and the bubbles 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. Dataflow models, popular in signal processing, are a special case of process networks [22].

PN models are excellent for signal processing. They 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.

Several special cases of PN are useful in certain circumstances. 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. Boolean dataflow (BDF) is a generalization that sometimes yields 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. The general case, process networks (PN), is implemented in Ptolemy II using Java threads for the processes.

1.2.8 Timed CSP and Timed PN

CSP and PN both involve threads that communicate via message passing, synchronously in the former case and asynchronously in the latter. Neither model intrinsically includes a notion of time, which can make it difficult to interoperate with models that do include a notion of time. In fact, message events are partially ordered, rather than totally ordered as they would be were they placed on a time line.

Both models of computation can be augmented with a notion of time to promote interoperability. Threads assume that time does not advance while they are active, but can advance when they stall on inputs, outputs, or explicitly indicate that time can advance. By this vehicle, additional constraints are imposed on the order of events, and determinate interoperability with timed models of computation becomes possible.

1.3 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 [23].

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. 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. Differential equations, for instance, are very good at describing the interaction of point masses in a model of a MEMS system, but not as good at describing the discrete control logic that may be ultimately controlling the actuators in the MEMS system. Similarly, finite-state machines are good at modeling at least simple control logic, but hopelessly inadequate for modeling the interaction of point masses. Thus, to design interesting systems, designers need to use heterogeneous models.

1.4 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.

> *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.

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.

1.5 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.

1.5.1 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.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 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 trans-

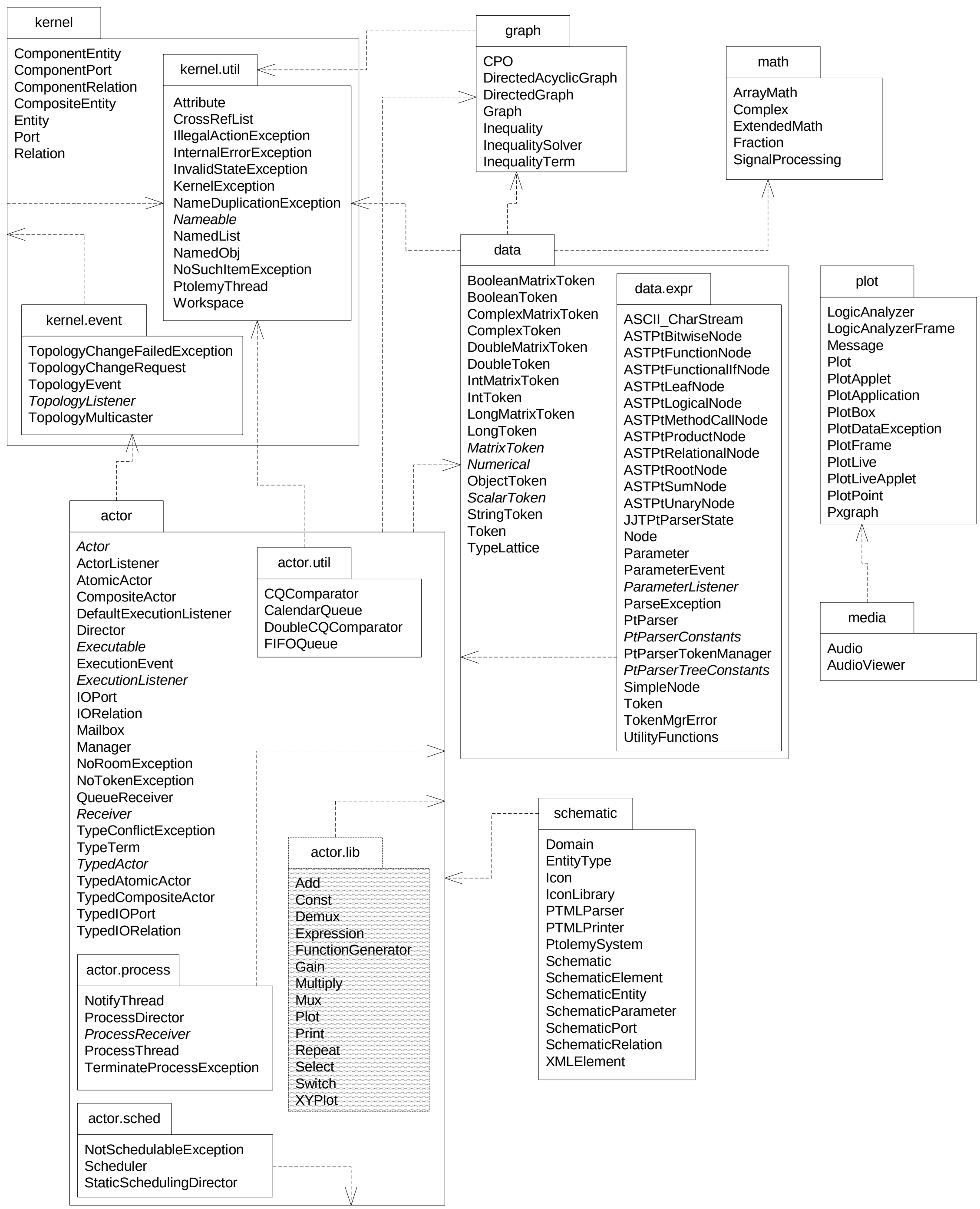


FIGURE 1.2. The package structure of Ptolemy II. The actor.lib package has not yet been fully constructed.

	ported 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.
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.
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.

1.5.2 Overview of Key Classes

Some of the key classes in Ptolemy II are shown in figure 1.3. This is a *static structure diagram* in UML (unified modeling language). 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 syntax of this diagram and 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 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

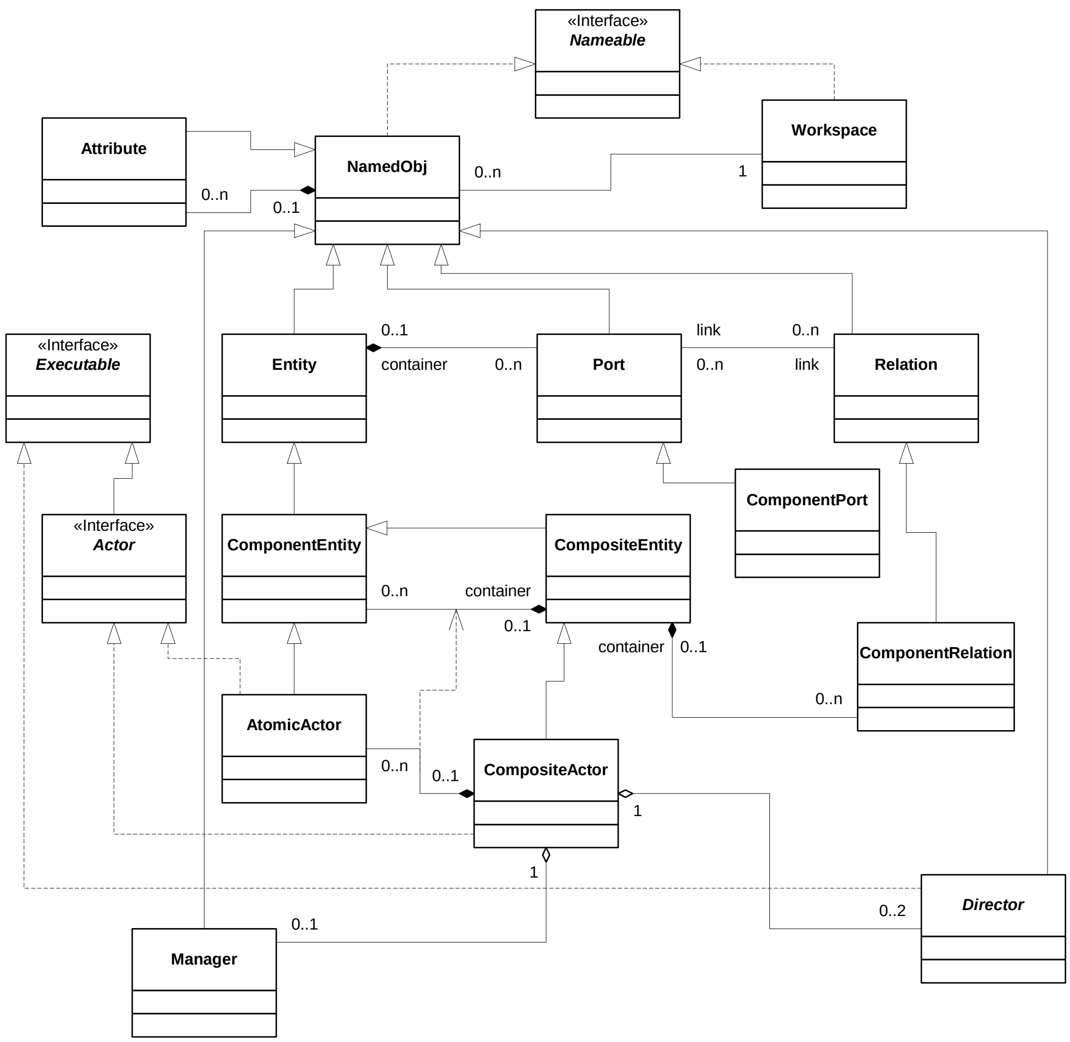


FIGURE 1.3. Some of the key classes in Ptolemy II. These are defined in the *kernel*, *kernel.util*, and *actor* packages.

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.

1.5.3 Capabilities

Ptolemy II is a second generation system. Its predecessor, Ptolemy 0.x, 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 [21]. Ptolemy II includes a number of domains that support design of concurrent systems at a much higher level of abstraction. These include, at varying levels of maturity, process networks, communicating sequential processes (rendezvous based), dataflow, synchronous/reactive modeling, continuous-time modeling, and hierarchical concurrent finite-state machines.
- *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.* Previous realizations of Ptolemy provided a wormhole mechanism for hierarchically coupling heterogeneous models of computation. This mechanism is improved in Ptolemy II through the use of opaque composite actors, which provide better support for models of computation that are very different from dataflow, the best supported model in prior versions of Ptolemy software. 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 [19] built upon the lower level synchronization primitives of Java.
- *A software architecture based on object modeling*. Since the first Ptolemy implementation, software engineering has seen the emergence of sophisticated object modeling [26][35][37] and design pattern [13] 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 [25][29].
- *A truly polymorphic type system*. Earlier implementations of Ptolemy 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 [28].

- *Domain-polymorphic actors*. In earlier implementations of Ptolemy, 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*.

1.5.4 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, JINI, or COM, 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*. Earlier versions of Ptolemy 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 separate the interface definition of component blocks from their implementation, allowing libraries to be constructed where compatibility across implementation technologies is assured [36]. This work is currently being prototyped in Ptolemy 0.7.1.
- *Integrated verification tools.* Modern verification tools based on model checking [18] 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.

2

The Kernel

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

2.1 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 [7], 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 2.1, 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*. [13]. “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.

2.2 UML Notation

The most basic classes in the Ptolemy II kernel package and their relationships are shown in figure 2.2, using UML notation [12][33]. Such relationships are called an *object model*, and represent many essential features about the design. We show only the *static structure diagrams*, or *class diagrams* of UML.

The class name is shown at the top of each box, its *attributes* are shown below that, and its methods below that. The attributes are usually not directly visible to a programmer using these classes (they are implemented as private members). But they are a useful part of the object model because they indicate the state information contained by an instance of the class.

Subclasses are indicated by lines with white triangles (or outlined arrow heads). The class on the

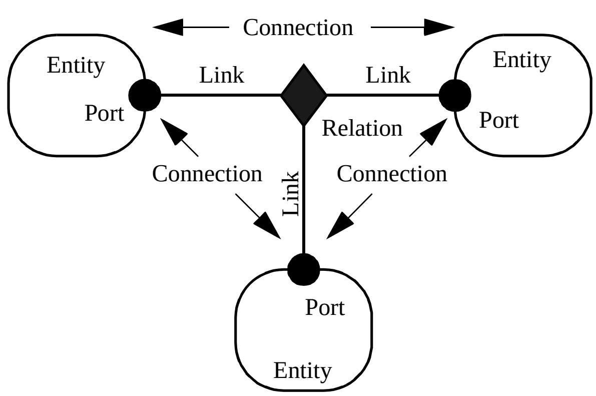


FIGURE 2.1. Visual notation and terminology.

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 inherited from a base class. For example, in figure 2.3, Attribute has all the methods of NamedObj, 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.

Our object models do not show private methods, which are not inherited. For completeness, our object models do show all public and protected methods of these classes, although a proper object model might only show those relevant to the issues being discussed. We also show the constructors, which always have the same name as the class and no return type.

Attributes with leading underscores, such as _portList, are private or protected members or methods. In the UML diagrams, private members are indicated with a leading "-". Public methods have a leading "+" and protected methods a leading "#".

Classes shown in boxes outlined with dashed lines, such as NamedObj, CrossRefList, and NamedList in figure 2.2, 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. In the case of figure 2.2, those classes are shown fully in figure 2.3.

Figure 2.3 also shows an example of an *interface*, Nameable, which is indicated by the label "<<Interface>>" and by italics in the name. An interface defines a set of methods without providing an

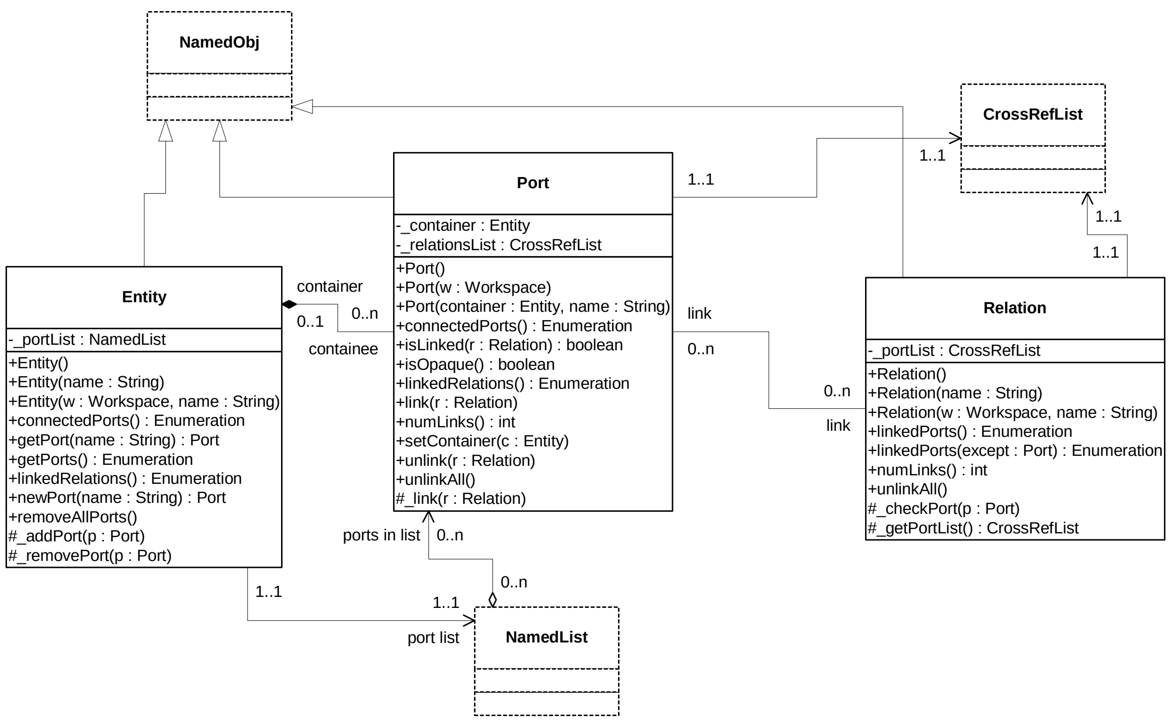


FIGURE 2.2. Key classes in the kernel package and their methods supporting basic (non-hierarchical) topologies. Methods that override those defined in a base class or implement those in an interface are not shown. The "+" indicates public visibility, "#" indicates protected, and "-" indicates private. Capitalized methods are constructors. The classes shown with dashed outlines are in the kernel.util subpackage.

implementation for them. 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

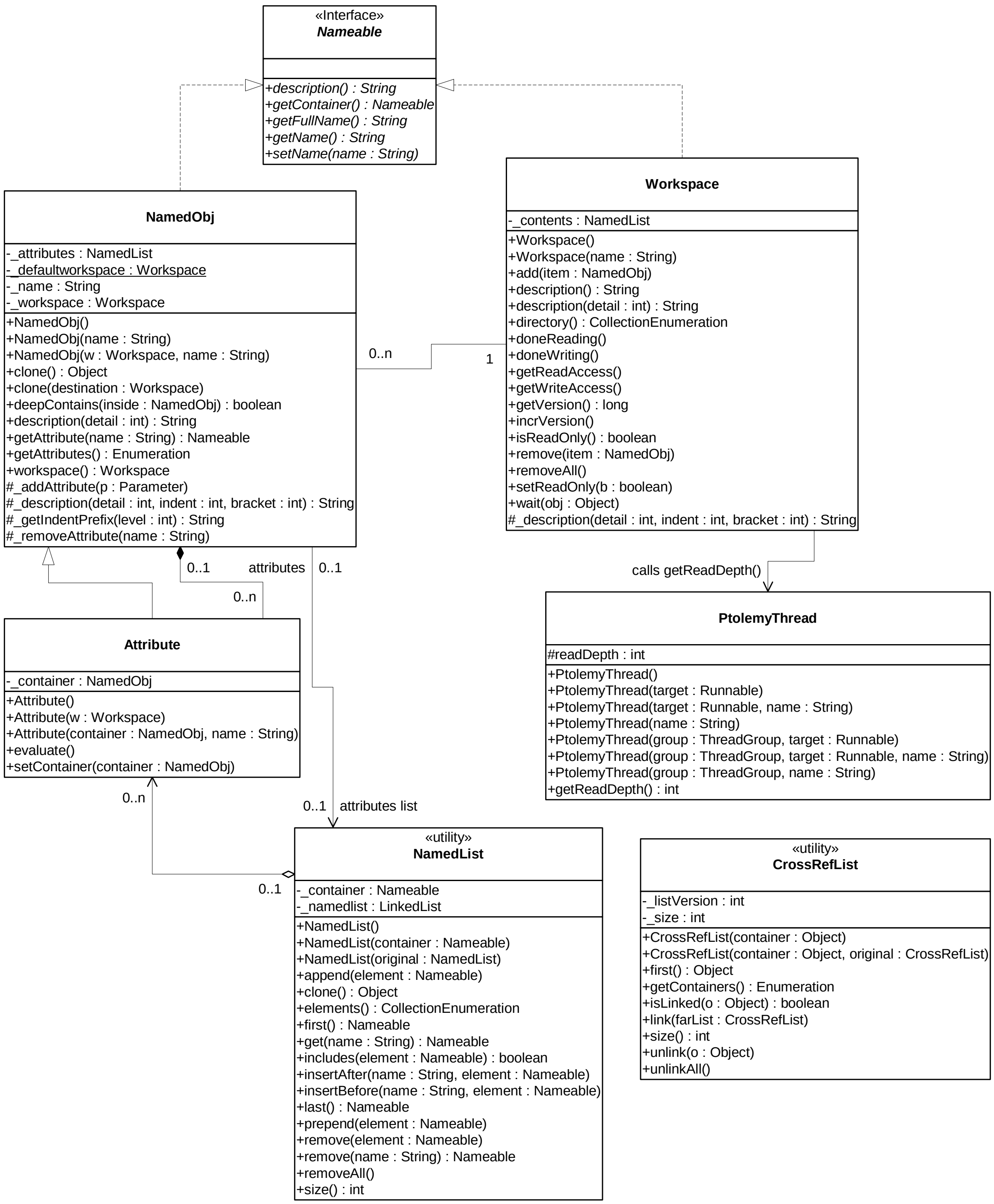


FIGURE 2.3. Support classes in the kernel.util package.

show those methods explicitly in the concrete class, just like inherited methods, but their presence is implicit in the relationship.

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.

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, or that Entity is a *composition* of Ports.

This containment is mediated by the NamedList utility class, shown in figure 2.3. Unlike the containment association, however, the Port has no reference to a NamedList that refers to it, and any number of NamedList instances can refer to it. Only one Entity can contain it. The stronger form of aggregation (containment or composition) is indicated by the filled diamond, while the weaker form is indicated by the unfilled diamond.

As usual in UML, return types of methods are shown after a colon. Types of arguments are also shown after a colon, but within the parentheses that delimit the argument list.

2.3 Ptolemy II Naming Conventions

We have made an effort to be consistent about naming of classes, methods and members. Class names are capitalized, with internal word boundaries also capitalized (as in "NamedObj"). Method names that are plural, such as getPorts(), usually return an enumeration (or sometimes an array). As explained before, private and protected members and methods have a leading underscore. Members and methods are not capitalized, although internal word boundaries usually are. Considerable discussion was involved in the choice of most class and method names, although inevitably, we had to make some compromises.

2.4 Non-Hierarchical Topologies

The classes shown in figure 2.2 support non-hierarchical topologies, like that shown in figure 2.1.

2.4.1 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 2.2 and defined fully in figure 2.3. There is exactly one 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 2.3. There is exactly one instance of CrossRefList associated with each port and each relation. The links define a web of interconnected entities.

2.4.2 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.

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 explained below in section 2.8. 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.

2.5 Support Classes

The kernel package has a subpackage called kernel.util that provides some underlying support classes, some of which are shown in figure 2.3. These classes define notions basic to Ptolemy II of containment, naming, and parameterization, and provide generic support for relevant data structures.

2.5.1 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" [21] is used as a central tool in thread safety, as explained in section 2.8 below.

In the base classes shown in figure 2.2, 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 2.3, only Attribute has a setContainer() method.

Every object is associated with exactly one instance of Workspace, as shown in figure 2.3, 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.

2.5.2 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 2.5.4.

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 [38], 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).

2.5.3 Workspace

Workspace is a concrete class that implements the Nameable interface, as shown in figure 2.3. 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 2.8 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.

2.5.4 Attributes

In almost all applications of Ptolemy II, entities, ports, and relations need to be parameterized. The base classes shown in figure 2.3 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.

2.5.5 List Classes

Figure 2.3 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 2.3 to maintain a list of attributes, and in figure 2.2 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.

2.6 Clustered Graphs

The classes shown in figure 2.2 provide only partial support for hierarchy, through the concept of a container. Subclasses, shown in figure 2.4, 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 [13].

2.6.1 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 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 [27]. 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 2.5[2]. Some of the ports in figure 2.5 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 2.4. These methods flatten the hierarchy by traversing it. Thus, for example,

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.

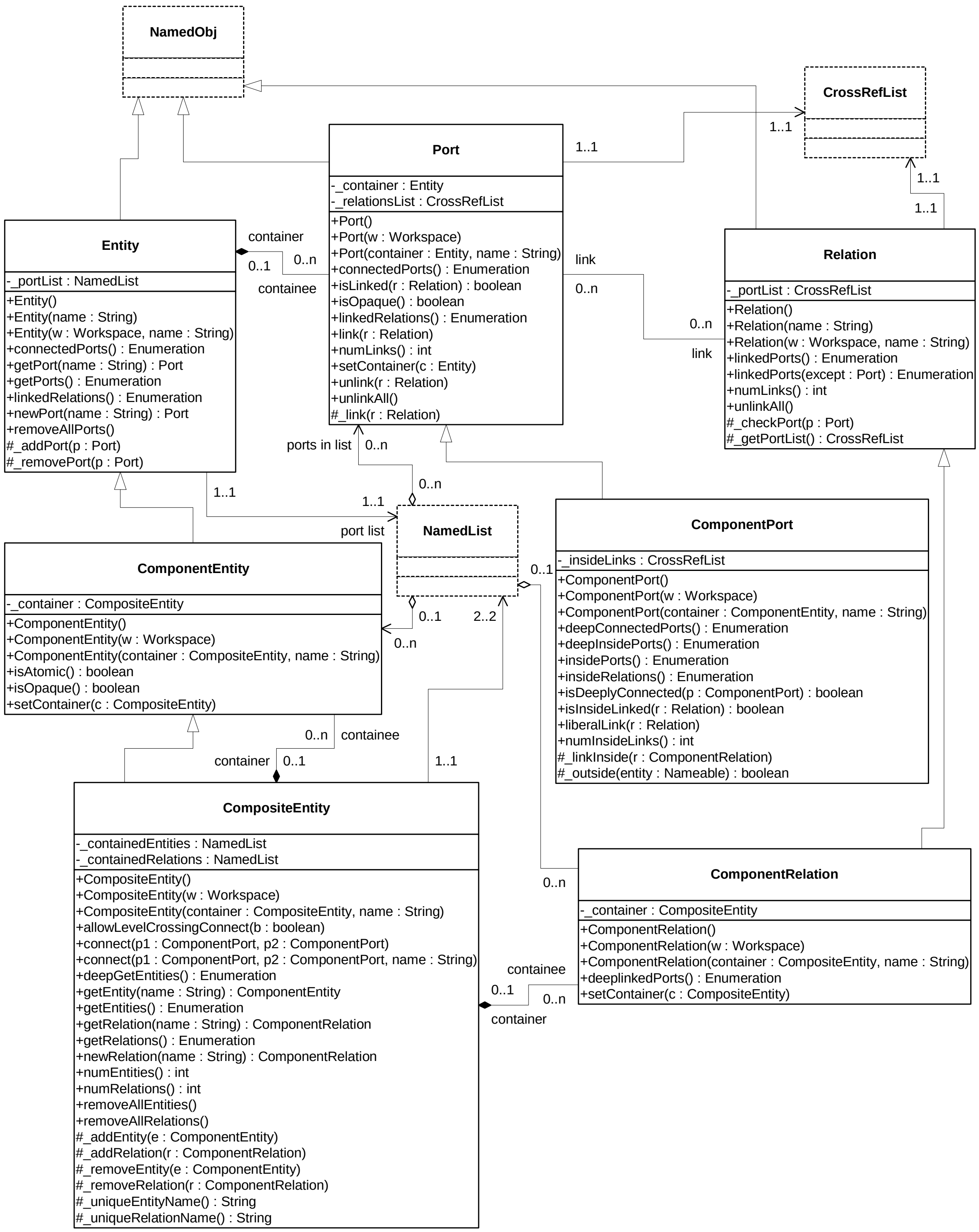


FIGURE 2.4. Key classes supporting clustered graphs.

the ports that are “deeply” connected to port P1 in figure 2.5 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 2.2, 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.

2.6.2 Level-Crossing Connections

For a few applications, such as Statecharts [15], level-crossing links and connections are needed. The example shown in figure 2.6 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-

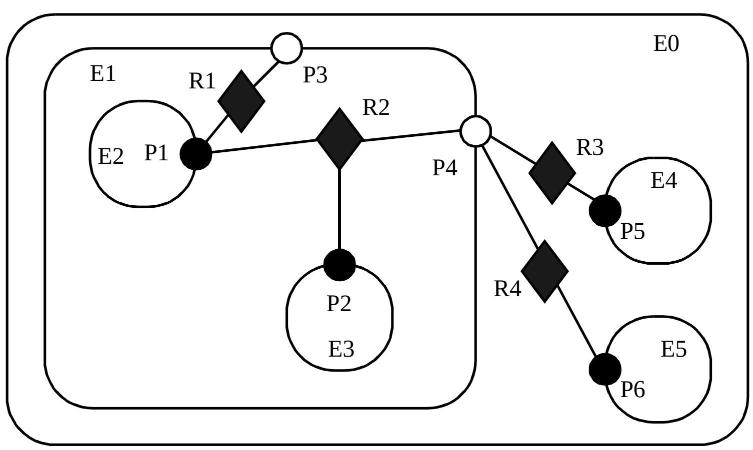


FIGURE 2.5. Transparent ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

crossing links using connect(), first call allowLevelCrossingConnect() with a *true* argument.

The simplest level-crossing connection in figure 2.6 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 2.6 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.

2.6.3 Tunneling Entities

The transparent port mechanism we have described supports connections like that between P1 and P5 in figure 2.7. 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

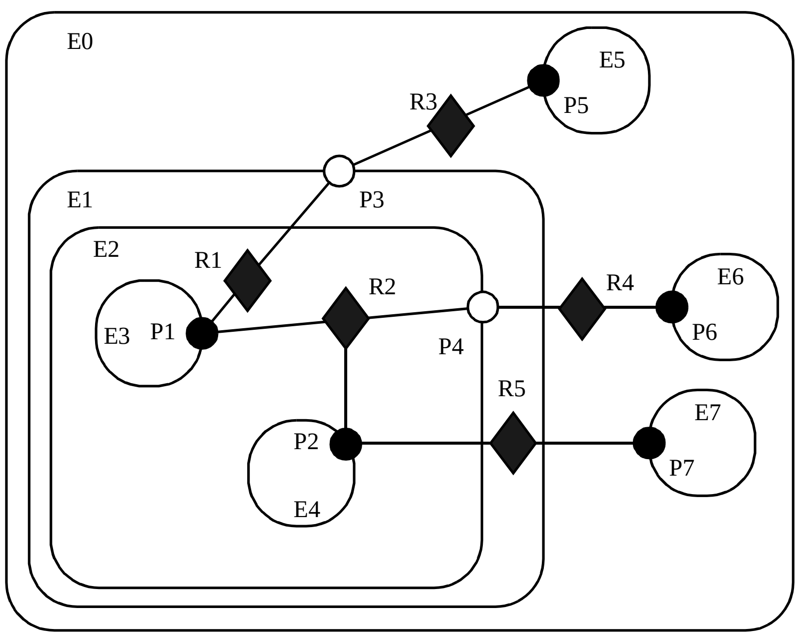


FIGURE 2.6. An example with level-crossing transitions.

deep graph traversals will see right through it.

Support for tunneling entities is a major increment in capability over the previous Ptolemy kernel [7] (Ptolemy 0.x). 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 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 [22] was made much more complicated. These higher-order functions had to be careful to avoid mutations that created tunneling.

2.6.4 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 2.3) 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 2.8, which describes the topology in figure 2.7. 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 2.8, 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.

2.6.5 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.

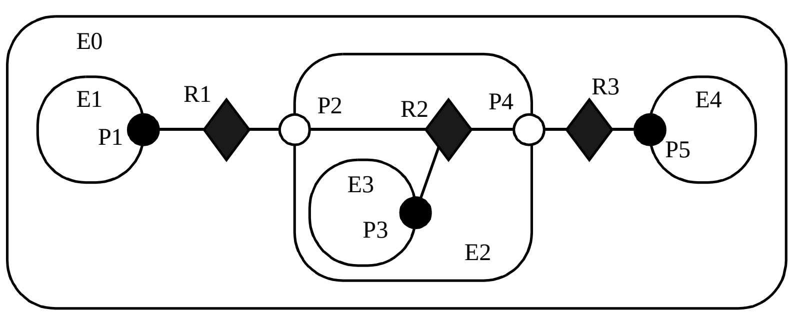


FIGURE 2.7. A tunneling entity contains a relation with inside links to more than one port.

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.

2.6.6 An Elaborate Example

An elaborate example of a clustered graph is shown in figure 2.9. 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 2.10. A script in the Tcl language [30] that constructs the same graph is shown in figure 2.11. This script uses TclBlend, an interface between Tcl and Java that is distributed by Sun Microsystems.

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 2.10 and 2.11 is top-to-bottom and

```
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 2.8. An example of the syntax returned by the description() method.

left-to-right in figure 2.9. 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 2.12. This table can be studied to better understand the precise meaning of each of the methods.

2.7 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 Actors chapter. In the previous generation system, Ptolemy 0.x, composite opaque entities were called *wormholes*.

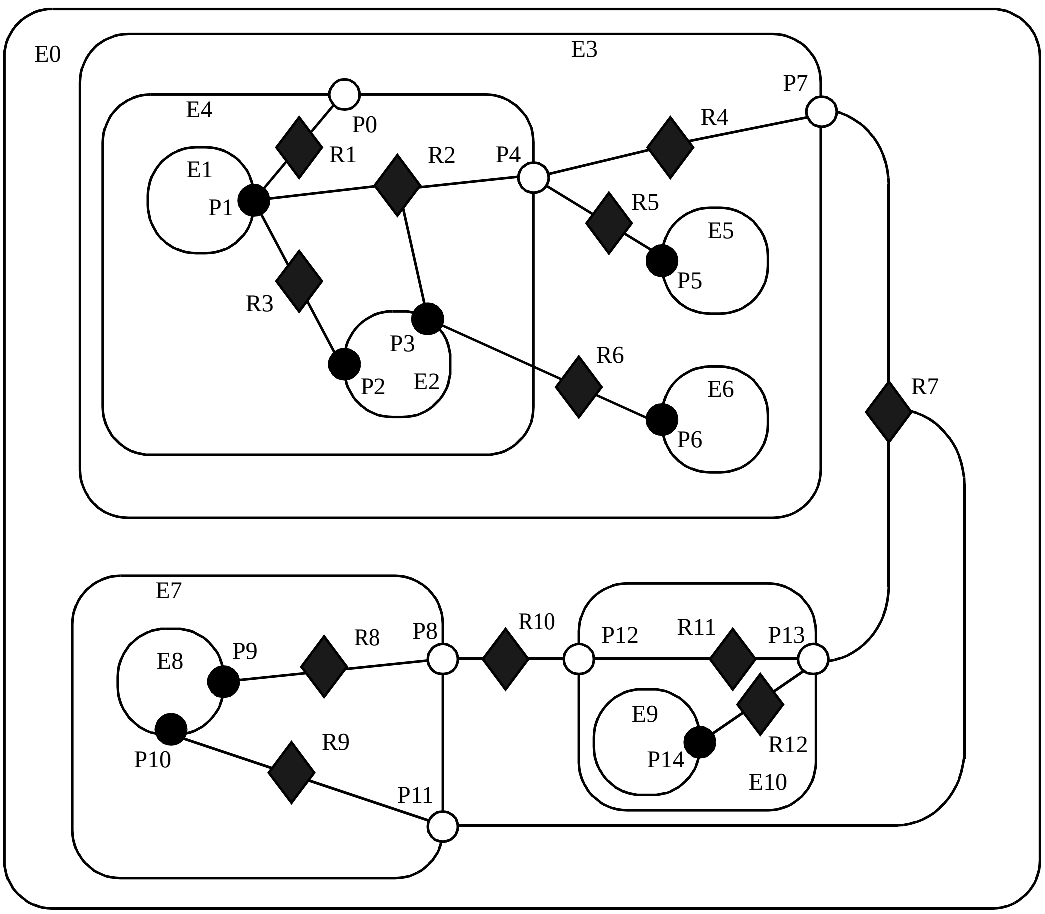


FIGURE 2.9. An example of a clustered graph.

```
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 2.10. The same topology as in figure 2.9 implemented as a Java class.

2.8 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 2.4.2).

```
# 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 2.11. The same topology as in figure 2.9 described by the TclBlend commands to create

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 *thread safety.*

2.8.1 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 code or a method, as shown in figure 2.13. 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 2.13(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 2.13(b), the code body represented by ellipses (...) can be executed only after a lock has been acquired on object *obj*.

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 2.12. Key methods applied to figure 2.9.

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 [21]. 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) {
   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 con-

```
public class A {
   public synchronized void foo() {
      ...
   }
}
```

(a)

```
public class B {
   public void foo() {
      synchronized(obj) {
         ...
      }
   }
}
```

(b)

```
public class C extends NamedObj {
   public void foo() {
      synchronized(workspace()) {
         ...
      }
   }
}
```

(c)

```
try {
   workspace().getReadAccess();
   // ... code that reads
} finally {
   workspace().doneReading();
}
```

(d)

```
try {
   workspace().getWriteAccess();
   // ... code that writes
} finally {
   workspace().doneWriting();
}
```

(e)

FIGURE 2.13. Using monitors for thread safety. The method used in Ptolemy II is in (d) and (e).

tainment 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.

2.8.2 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 2.3, every 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 2.3, 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” [21] is our central strategy in thread safety.

As shown in figure 2.13(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 2.13(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 2.13(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 2.13(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 2.3). 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.

2.8.3 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 2.3). 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.

2.9 Topology 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 program 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 2.8.3), 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 2.14. The style of this design is strongly inspired by the event model in the AWT.

The typical usage pattern involves a **source** that wishes to have a mutation performed, such as an actor (see the Actors chapter), and an object that can safely perform the mutation, such as a director, (again, see the Actors chapter). The source creates an instance of the class TopologyChangeRequest and enqueues that request by calling the queueTopologyChangeRequest() of the director. When it is safe, the director executes the change by calling performRequest() on each enqueued TopologyChangeRequest. In addition, it informs any registered listeners of the mutations so that they can react accordingly.

We have taken some liberties with the notation in figure 2.14. There is no class called Source, so the name is shown in parentheses and the class with dashed outlines. This class represents typical uses of the event package, but are not included in the event package.

The Director class in the actor package allows topology mutations to occur only between iterations

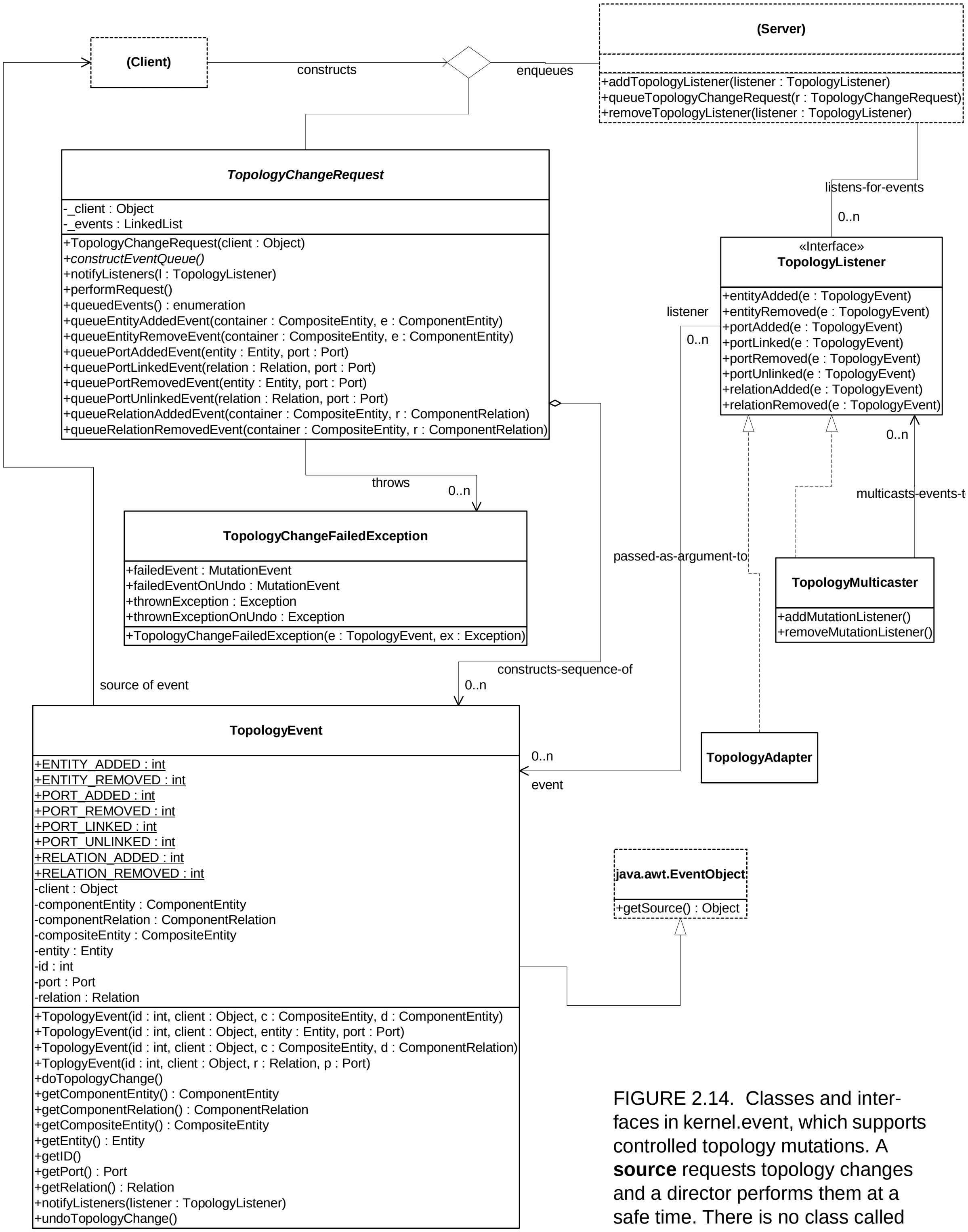


FIGURE 2.14. Classes and interfaces in kernel.event, which supports controlled topology mutations. A **source** requests topology changes and a director performs them at a safe time. There is no class called "Source," so its name is shown in parentheses.

of an iterative execution of an application (see the Actors chapter). To support this, it has a queueTopologyChangeRequest() method that permits any other code to specify a mutation to be performed at the first opportunity.

2.9.1 Structure of Topology Change Request

Internally, each topology change request contains a sequence of topology events. Each event represents a primitive operation on the topology, such as adding an entity or creating a link between a port and a relation. Thus, a topology change request is an aggregation of topology events. Although is it not enforced (and probably cannot be), each such aggregation should be constructed so that the graph is transformed from one consistent, executable state to another.

A director processes a topology change request by calling its performRequest() method. This method activates each event by calling its doTopologyChange() method. If any of the events fails to perform (that is, it throws an exception) the entire request is rolled back – that is, undone – so that the graph remains in a consistent state. To do so, the undoEventAction() method of each event. If the undo completes successfully, doTopologyChange() throws a TopologyChangeFailedException containing the original exception that cause the event to fail. If the undo fails to complete, it throws a TopologyChangeFailedException containing both the original exception and the exception thrown by the unsuccessful undo operation.

Note that the attempt to roll back an unsuccessful topology change request can only go back as far as the start of the failed request. In general, directors and other classes that process topology changes should not attempt to roll back requests that have already been successfully completed, especially if there may be listeners that have already been notified about the change.

If the request completes successfully, the director then notifies all topology change listeners attached to it. It does this by calling the notifyListeners() method of the TopologyChangeRequest objects, which in turn calls the notifyListeners() method of the TopologyEvent objects. The notifyListeners() method of the TopologyEvent object calls the appropriate method in the listener interface.

Because of the possibility of failure of a change request, the director performs each request and notifies listeners immediately, rather than performing all requests and then notifying listeners of all of them. Any other classes that later support topology changes should be written to follow this pattern. The assumption is that changes that can be aggregated will be aggregated in a single instance of TopologyChangeRequest.

The TopologyChangeRequest class is abstract. In particular, one method, constructEventQueue() is abstract. Thus, to build a topology change request, a source will typically define an anonymous inner class, like this:

```
TopologyChangeRequest change = new TopologyChangeRequest() {
    public void constructEventQueue () {
        ...
    }
}
director.queueTopologyChangeRequest(change);
```

The body of the constructEventQueue() method should create entities, relations, ports, and so on. It should not directly insert these into the topology, however. Instead, it should call the (final) methods queueEntityAddedEvent(), queuePortAddedEvent(), and so on. These methods indicate what mutations are to be performed later. These methods internally create instances of TopologyEvent. The source is responsible for ensuring that the complete event, when performed, will leave the kernel structure in an executable state.

For example, the code in the constructEventQueue() method to create, add, and link a new entity might look like this:

```
Entity fred = new MyEntityClass("Fred");
ComponentEntity myContainer = getContainer();
queueEntityAddedEvent(myContainer, fred);
queuePortLinkedEvent(fred.getPort("input"), someRelation);
```

When performRequest() is called, the entity named *fred* will be created, added to *myContainer*, and linked to *someRelation*. When notifyListeners() is called, listeners will have their entityAdded() methods called with an event containing *myContainer* and *fred*, and then their portLinked() method called with an event containing *fred's* input port and *someRelation*.

2.9.2 Directors and Listeners

Any director can safely perform topology changes. (In general, other objects may also perform topology changes.) It provides addTopologyListener() and removeTopologyListener() methods, so that interested objects can register to be notified when topology changes occur. In addition, it provides a method that topology change sources can use to queue requests. The director is responsible for obtaining write access to the workspace, calling the performRequest() and notifyListeners() methods of each queued request, and releasing write access. It also deals with failure of any topology change request, and decides if and how to recover from such an exception.

A topology listener is any object that implements the TopologyListener interface, and will typically include user interfaces, visualization components, schedulers, and so on. These objects can attach themselves to a director with the method addTopologyListener().

The notifyListeners() method accepts a TopologyListener object as argument, and dispatches each event in the queue to that listener. It does not check that the events represent mutations that have been performed. It is up to the director to ensure that mutations are performed before listeners are notified.

The TopologyEvent contains all information about an atomic topology change, such as adding an entity or linking to a port. Its *id* signifies what kind of event it is – for example, whether it represents adding an entity to a composite entity, linking a relation to a port, and so on. The fields *compositeEntity*, *entity*, *port*, *relation*, *componentEntity*, and *componentRelation* contain the objects involved in the event. For any given event, only two of these fields will be non-null, and the listener is assumed to be written correctly and to use the right ones. The event class contains methods for setting and getting these public fields. In addition, it contains methods to process the topology change represented by the event and to undo it.

The TopologyListener follows the AWT listener conventions, and contains a series of methods, one of which is called for each event type. Each method accepts a single TopologyEvent as argument, which contains the information about what the event means. There are two concrete classes that implement TopologyListener in this package: TopologyMulticaster, which can have other listeners attached to it and to which it forwards each method call; and TopologyAdaptor, which is an empty implementation of TopologyListener that makes it more convenient to create anonymous event listener classes.

2.10 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 2.15.

2.10.1 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.

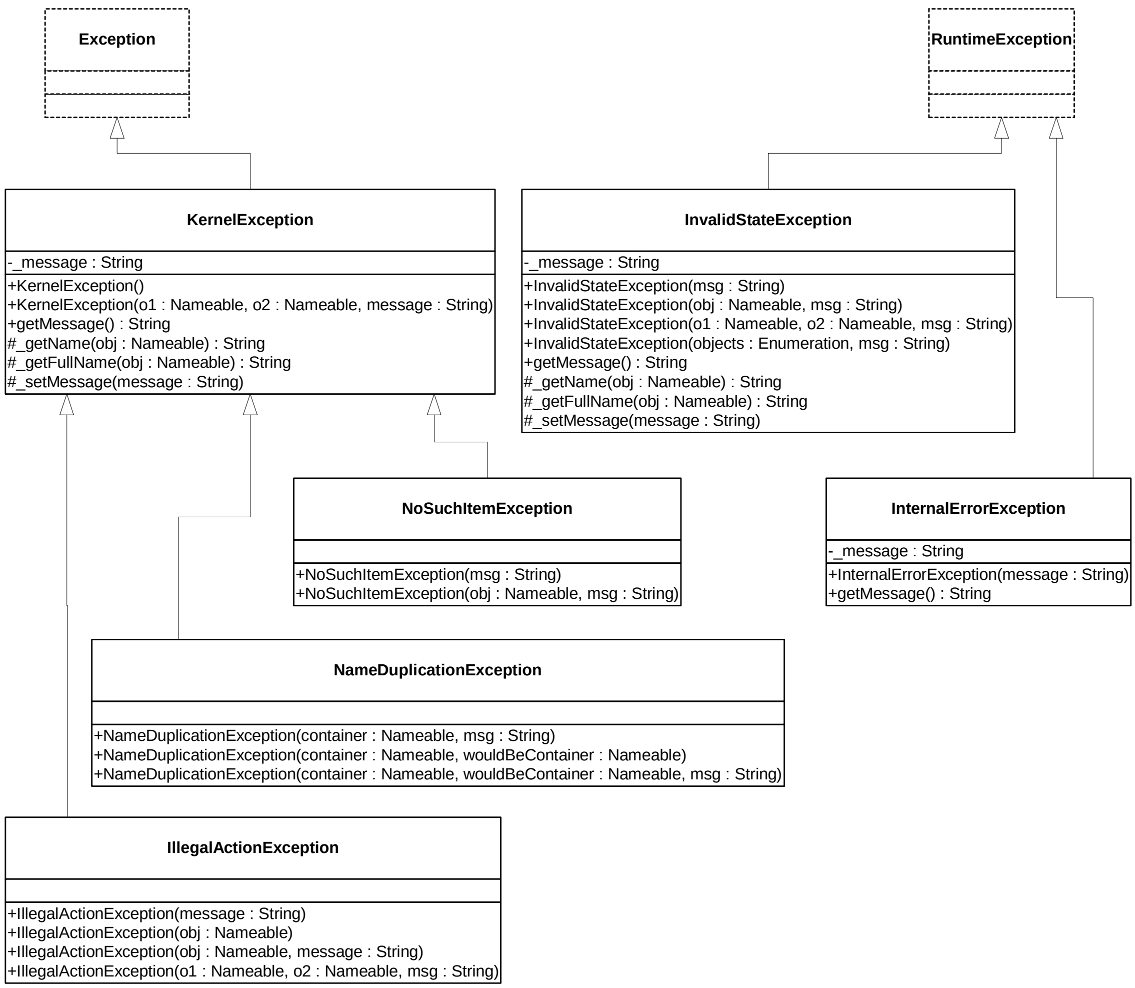


FIGURE 2.15. Summary of exceptions defined in the kernel.util package. These are used primarily through constructor calls. The form of the constructors is shown in the text. Exception and RuntimeException are Java exceptions.

2.10.2 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.

2.10.3 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 exception is derived from the Java RuntimeException.

3

Actors

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

3.1 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, 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. Agha's *actors* [1] extend the concept of objects to concurrent computation. His 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 than Agha’s, in that ours 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 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.

3.2 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 chapter). Messages are sent via ports. IOPort is the key class supporting message transport, and is shown in figure 3.2. An IOPort can only be connected to other IOPort instances, and only via IORelations. The IORelation class is also shown in figure 3.2. TypedIOPort and TypedIORelation are subclasses that manage type resolution. This is described in detail in the Types 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.

3.2.1 Data Transport

Data transport is depicted in figure 3.1. 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 implement the communication protocol pertinent to that domain. A domain that uses asynchronous message passing, for example, can usually use the QueueReceiver shown in figure 3.2. A domain that uses synchronous message passing (rendezvous) has to provide a new receiver class.

In figure 3.1 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 3.3. 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 3.4. 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 3.4, 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 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*.

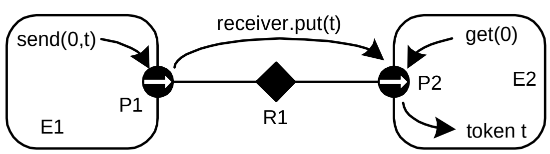


FIGURE 3.1. Message passing is mediated by the IOPort class. Its send() method obtains a reference to a remote receiver, and calls the put() method of the receiver, passing it the token *t*. The destination actor retrieves the token by calling the get() method of its input port.

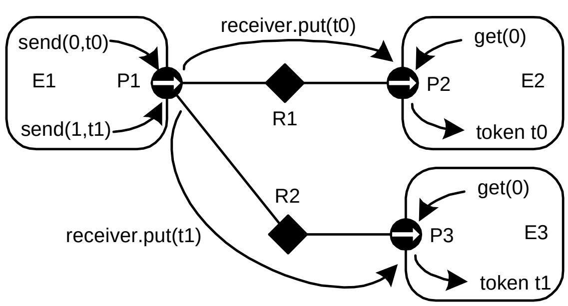


FIGURE 3.3. A port can support more than one channel, permitting an entity to send distinct data to distinct destinations via the same port. This feature is typically used when the number of destinations varies in different instances of the source actor.

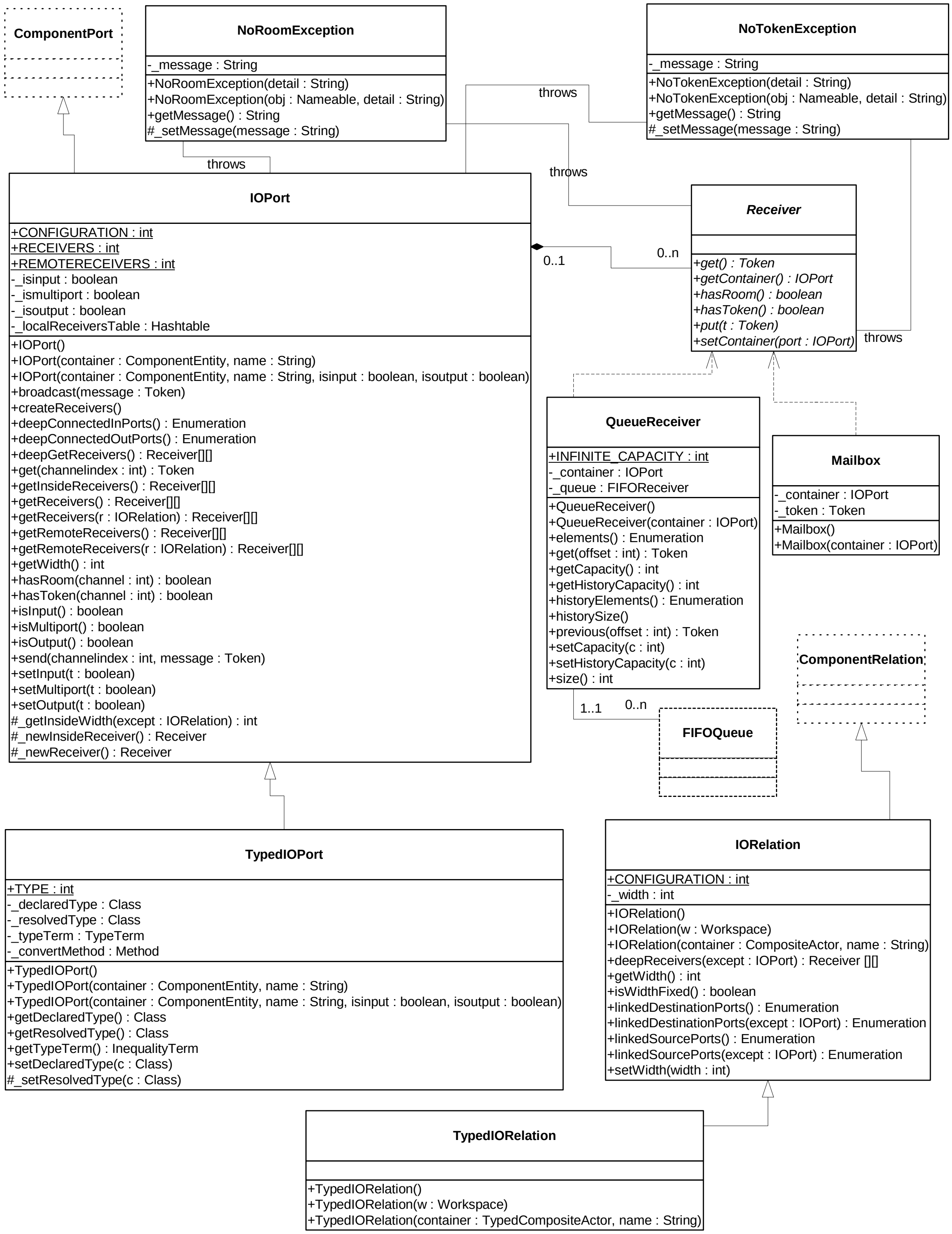


FIGURE 3.2. Port and receiver classes that provide infrastructure for message passing under

A given channel may reach multiple ports, as shown in figure 3.5. 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.

3.2.2 Example

An elaborate example showing all of the above features is shown in figure 3.6. 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.

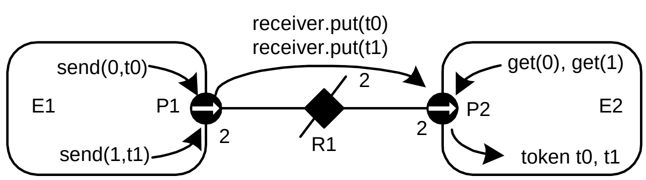


FIGURE 3.4. A bus is an IORelation that represents multiple channels. It is indicated by a relation with a slash through it, and the number adjacent to the bus is the width of the bus.

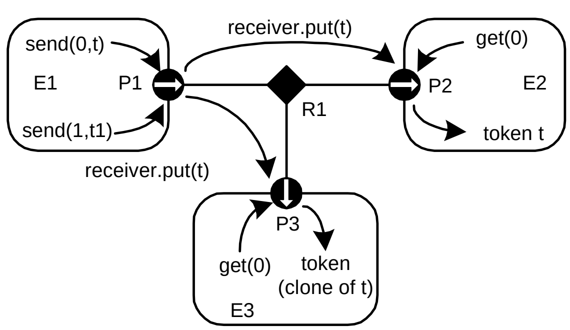


FIGURE 3.5. Channels may reach multiple destinations. This is represented by relations linking multiple input ports to an output port.

3.2.3 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 3.7 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 TclBlend code to construct this example is shown in figure 3.8.

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

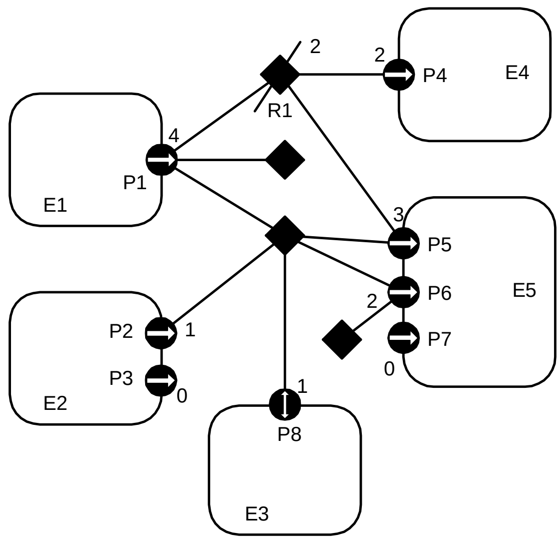


FIGURE 3.6. An elaborate example showing several features of the data transport mechanism.

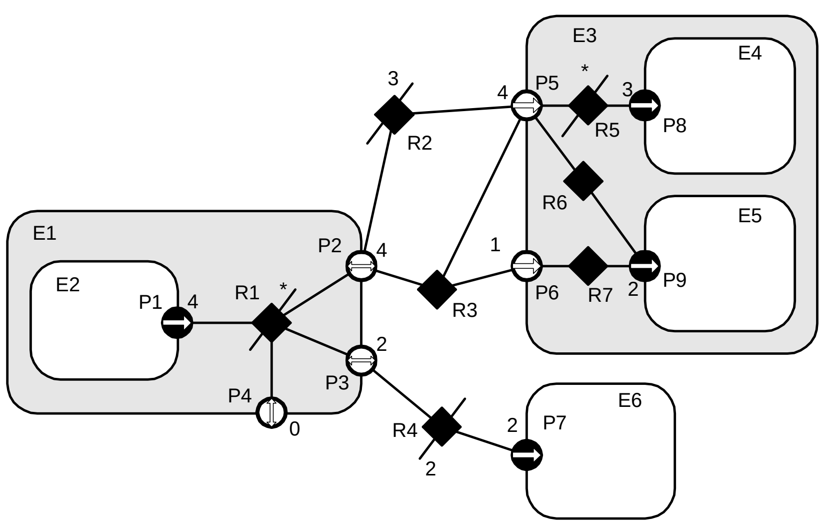


FIGURE 3.7. An example showing busses combined with input, output, and transparent ports.

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 3.8. TclBlend code to construct the example in figure 3.7.

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 3.2). 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 3.9, 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.

3.2.4 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 3.3 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 3.2.

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.

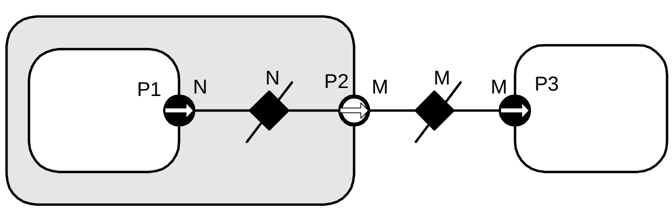


FIGURE 3.9. Bus widths inside and outside a transparent port need not agree..

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 [20], which is a generalization of dataflow [22], 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 [32].

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 3.10. 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 3.11. 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 AtomicActor {

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

   private IOPort _input;
   private IOPort _output;
}
```

FIGURE 3.11. 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 3.11 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-

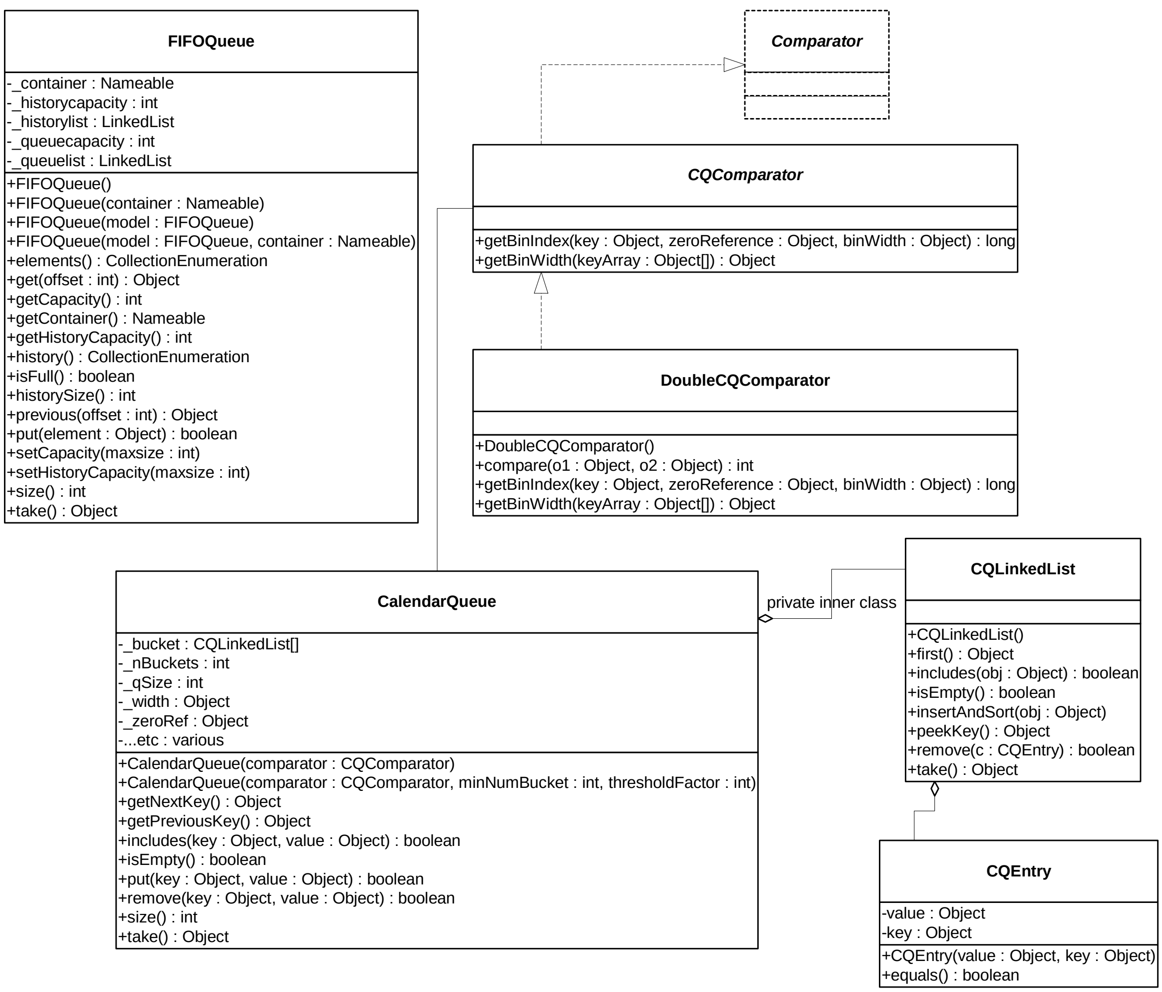


FIGURE 3.10. Static structure diagram for the actor.util package.

inating thread to indicate willingness to rendezvous will be the one that will transfer data. Guarded communication [3] 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 3.10, provides the CalendarQueue class, which gives an efficient and flexible implementation of such a sorted queue.

3.2.5 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 simulations of 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 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.

3.3 Execution

The Executable interface, shown in figure 3.12, 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.

The Executable interface defines how an object can be invoked. There are six 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 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 executable interface are executed in order during the normal flow of an iteration, the terminate() method can

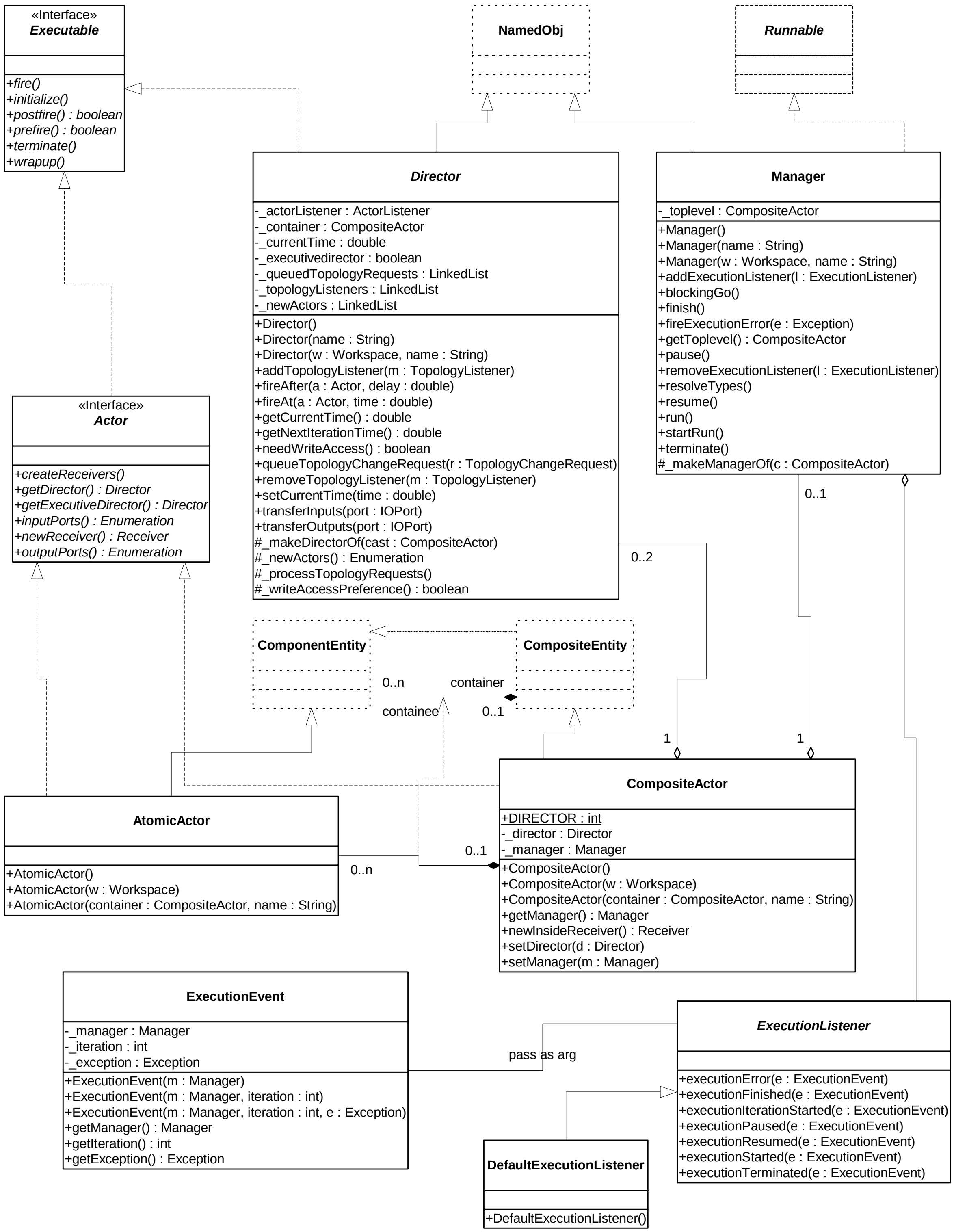


FIGURE 3.12. Basic classes in the actor package that support execution.

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 0.x. 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 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 3.3.5 below.

The fire() method may be invoked multiple times during an iteration. In most domains, this method defines the computation performed by the actor.

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 indicate that the actor should not be fired again. It has concluded its mission.

The wrapup() method is invoked exactly once during the execution of a model, unless an exception prevents its invocation. Typically, wrapup() is responsible for cleaning up after execution has completed, and perhaps flushing output buffers before execution ends.

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.

3.3.1 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 3.13. 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.

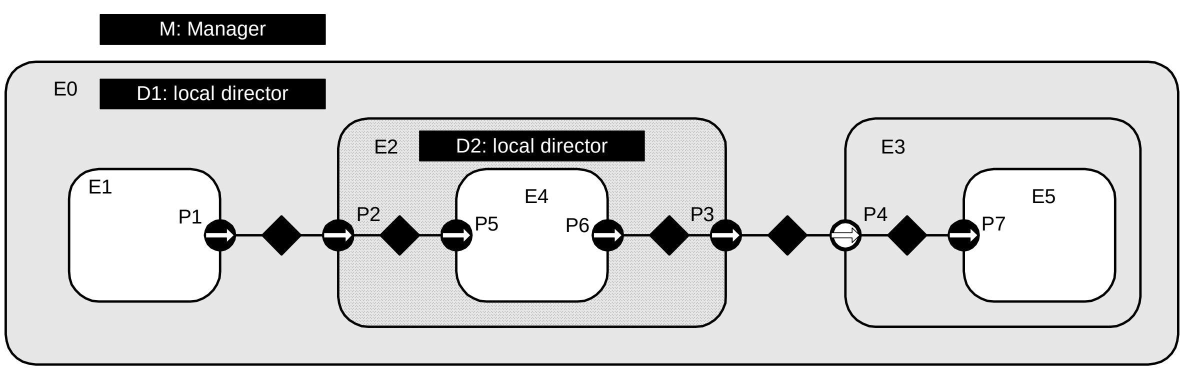


FIGURE 3.13. Example application, showing a typical arrangement of actors, directors, and managers.

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 3.13, 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 0.x, 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 3.3.5.

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 3.12).

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 3.14 and 3.15. These differ in the shaded areas, which define the domain-specific sequencing of actor firings. In figure 3.14, 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 3.15, 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().

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()

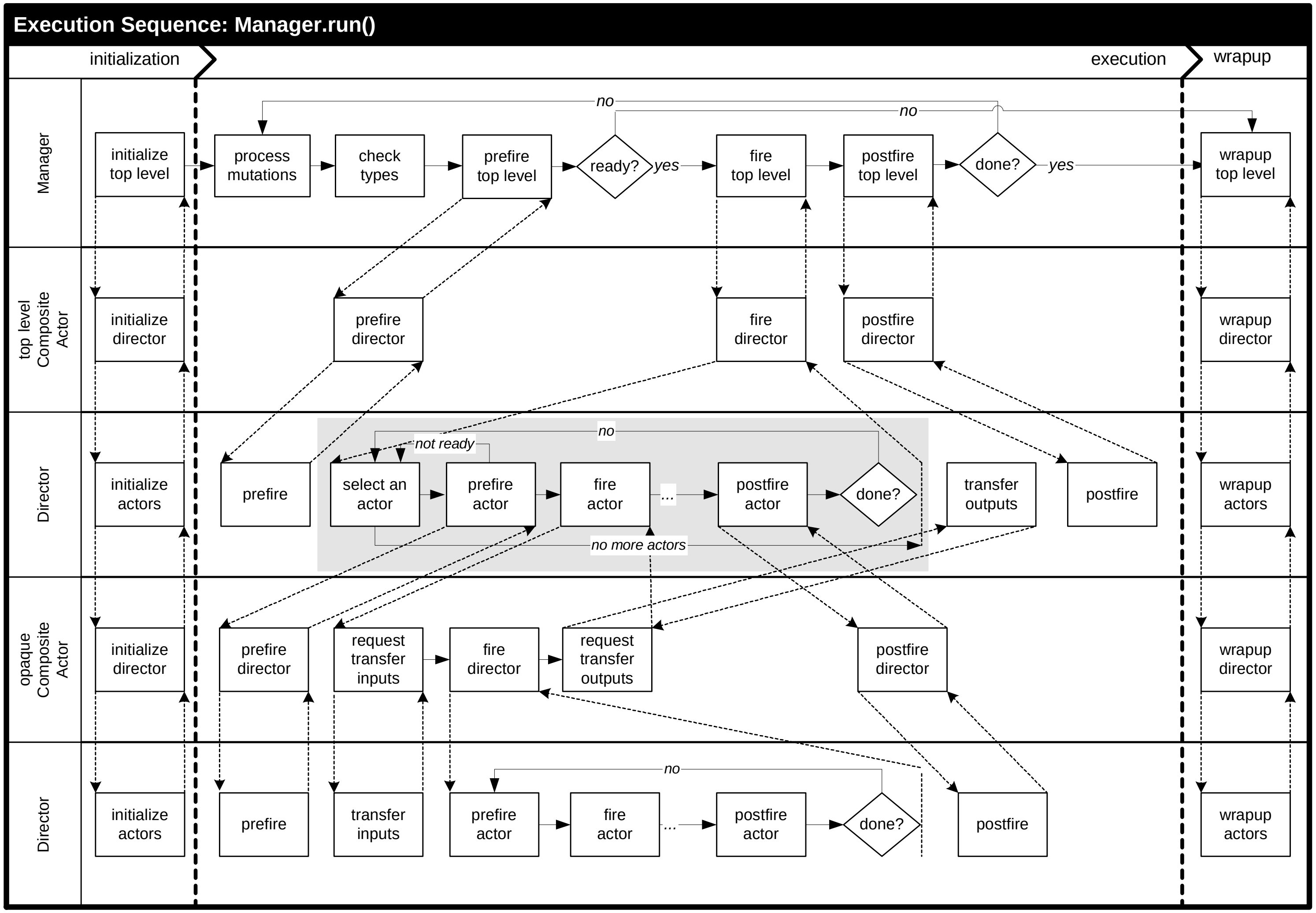


FIGURE 3.14. Example execution sequence implemented by run() method of the Director class.

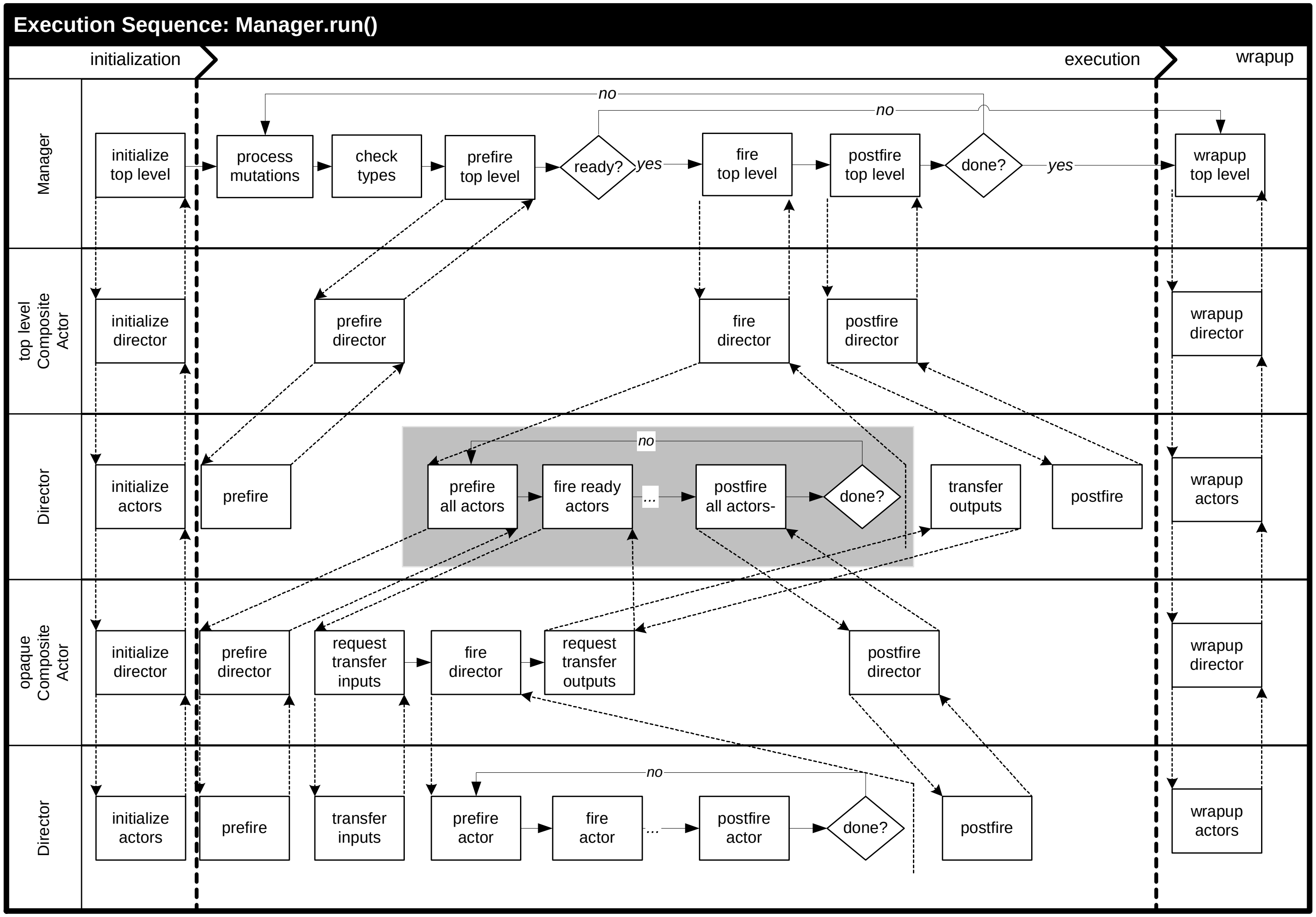


FIGURE 3.15. Alternative execution sequence implemented by run() method of the Director class.

is called. Of course, in order to promote interoperability, directors should define a finite execution that they perform in the fire() method.

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. Work is under way to provide classes that support this style of execution. There is also a family of domains where actors are associated with threads. Work is under way to provide classes to support this as well.

3.3.2 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 3.12. Execution can be initiated in a manager by either of two methods, run() and startRun(). The startRun() method spawns a thread that calls run(), and then immediately returns. After initializing the hierarchy by calling initialize() in the top-level composite actor, the manager will run multiple iterations within the top-level composite. This continues until postfire() in the top-level composite actor returns false, or either terminate() or finish() is called in the manager. The terminate() method corresponds to an immediate halt of execution, and should be used only if other more graceful methods for ending an execution fail. 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().

Execution may also be paused between top-level iterations by calling the pause() method. After each top-level iteration, the manager checks to see if pause() has been called. If so, 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, and pause() and resume() will have no effect.

3.3.3 ExecutionListener and ExecutionEvent

The ExecutionListener interface and the ExecutionEvent class provide 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 ExecutionListeners with an ExecutionEvent that describes the context of the event.

Several kinds of events are defined in the ExecutionListener interface. A listener is notified of these events by calling the appropriate method. The executionStarted() method indicates that execution has successfully begun and the system is about to be initialized. Likewise, executionFinished() indicates that execution has completed without error and all the actors in this system have completed execution of wrapup(). The executionTerminated() method indicates that the user requested that execution cease immediately, through the terminate() method, and the system has done whatever possible to return itself to a consistent state. The executionPaused() and executionResumed() methods are called when execution successfully pauses and resumes.

Note that in general, while these notification methods roughly correspond to methods in the Manager class, calling the methods in a manager does not imply that the events will occur immediately. The events are only issued to the listeners after the associated action is actually performed. An example of this is that calling the pause() method multiple times in a Manager without interleaving calls to resume() will result in a maximum of one executionPaused() call in the listeners.

Prior to each top-level iteration, the executionIterationStarted() method is called. This is intended to provide end users with confidence that execution is under way.

The executionError() method is called when the Manager catches an exception that was thrown during execution. All such exceptions are caught in the Manager, and if any execution listeners are currently registered with the Manager, then an ExecutionEvent will be created with the Exception encapsulated. If no execution listeners are registered then the stack trace will be printed to the standard output. In any domain that begins independent threads of execution, exceptions created during execution will be passed up to the run() method of each thread. In order to view these exceptions through the executionError() call to listeners, the independent threads should catch all exceptions and pass them along to the Manger using the fireExecutionError() method in the Manager class.

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

3.3.4 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 re-calculated to take into account the mutation. Type resolution may need to be re-done. Or a domain may wish to have tight control over when parameters of an application change.

The Director class leverages 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 rather than performing it directly (using the queueTopologyChangeRequest() method, shown in figure 3.12). When it is safe, that mutation is performed, and all mutation listeners that have been registered with the director (using the addTopologyListener() method) are informed of the mutation. In subclasses of Directory, the mutations are typically performed in the prefire() method.

The _newActors() method of Director returns an enumeration of actors that are created in a batch of mutations. This list is used to initialize these actors. It is possible that initialization of the new actors will result in further mutations (for example, if they are higher-order functions). Thus, the prefire() method of subclasses of Director needs to iteratively initialize new actors until there are no more pending mutations.

3.3.5 Composite Opaque 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() return *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 3.16. 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 3.14 and 3.15, 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.

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.

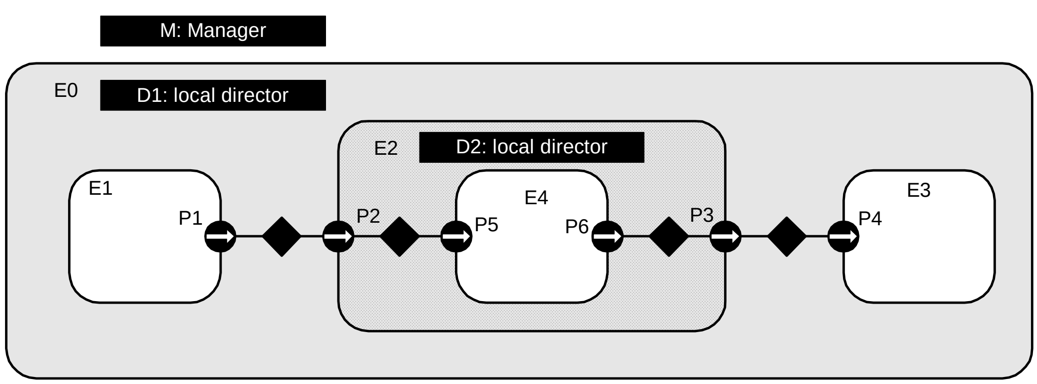


FIGURE 3.16. An example of an opaque composite actor. E0 and E2 both have local directors, not necessarily implementing the same model of computation.

4

Data

Neil Smyth
Yuhong Xiong

4.1 Introduction

The data package provides data encapsulation, polymorphism, parameter handling, an expression language, and a type system. Figure 4.1 shows the key classes in the main package (subpackages will be discussed later).

4.2 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, the value of a parameter may depend on the value of other parameters. The value of a parameter is

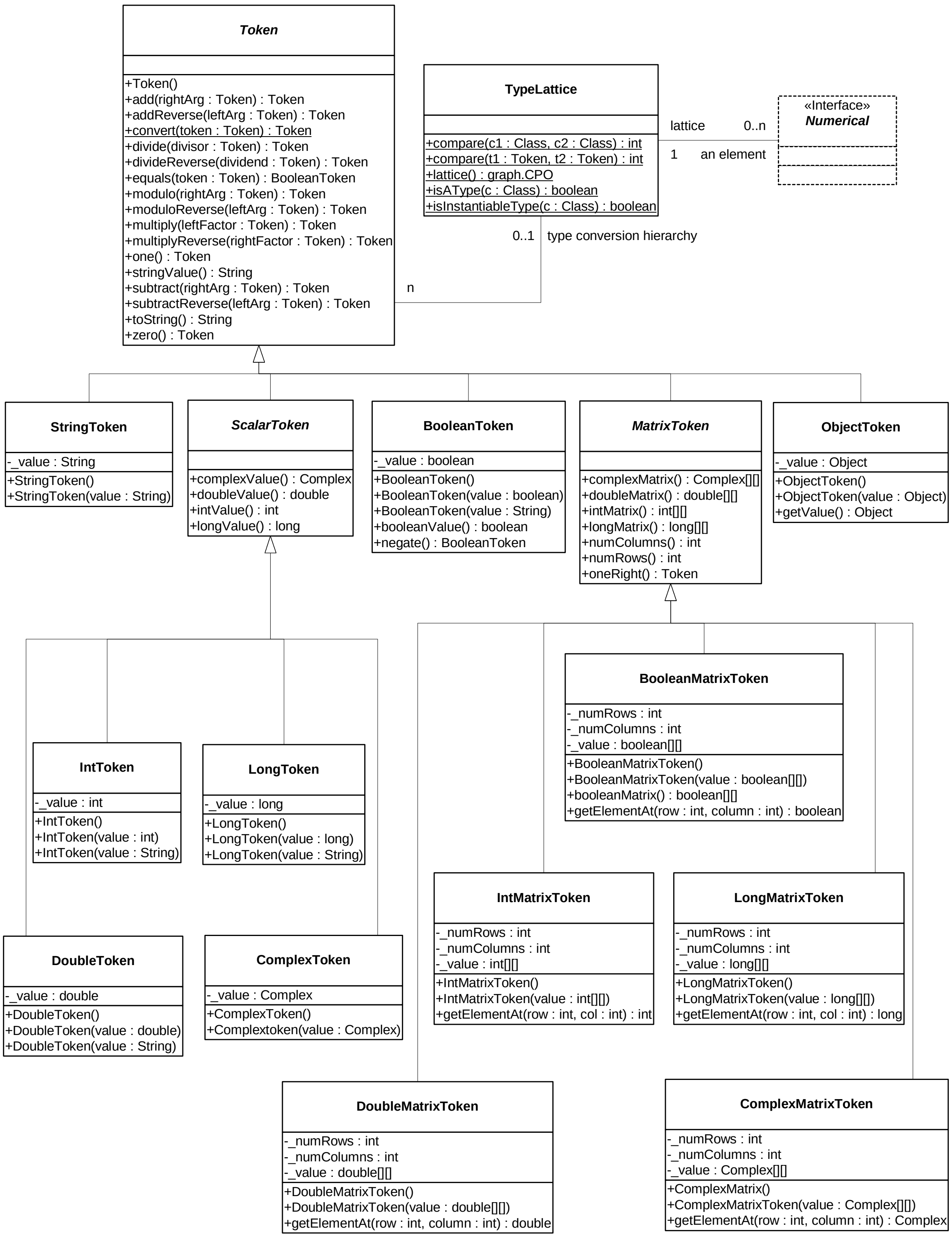


FIGURE 4.1. Static Structure Diagram (Class Diagram) for the classes in the data package.

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.

4.3 Polymorphism

4.3.1 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)
```

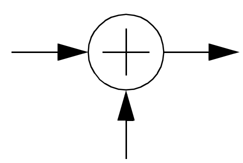
gives a new DoubleToken with value 7.2,

```
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.

4.3.2 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 4.2. 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

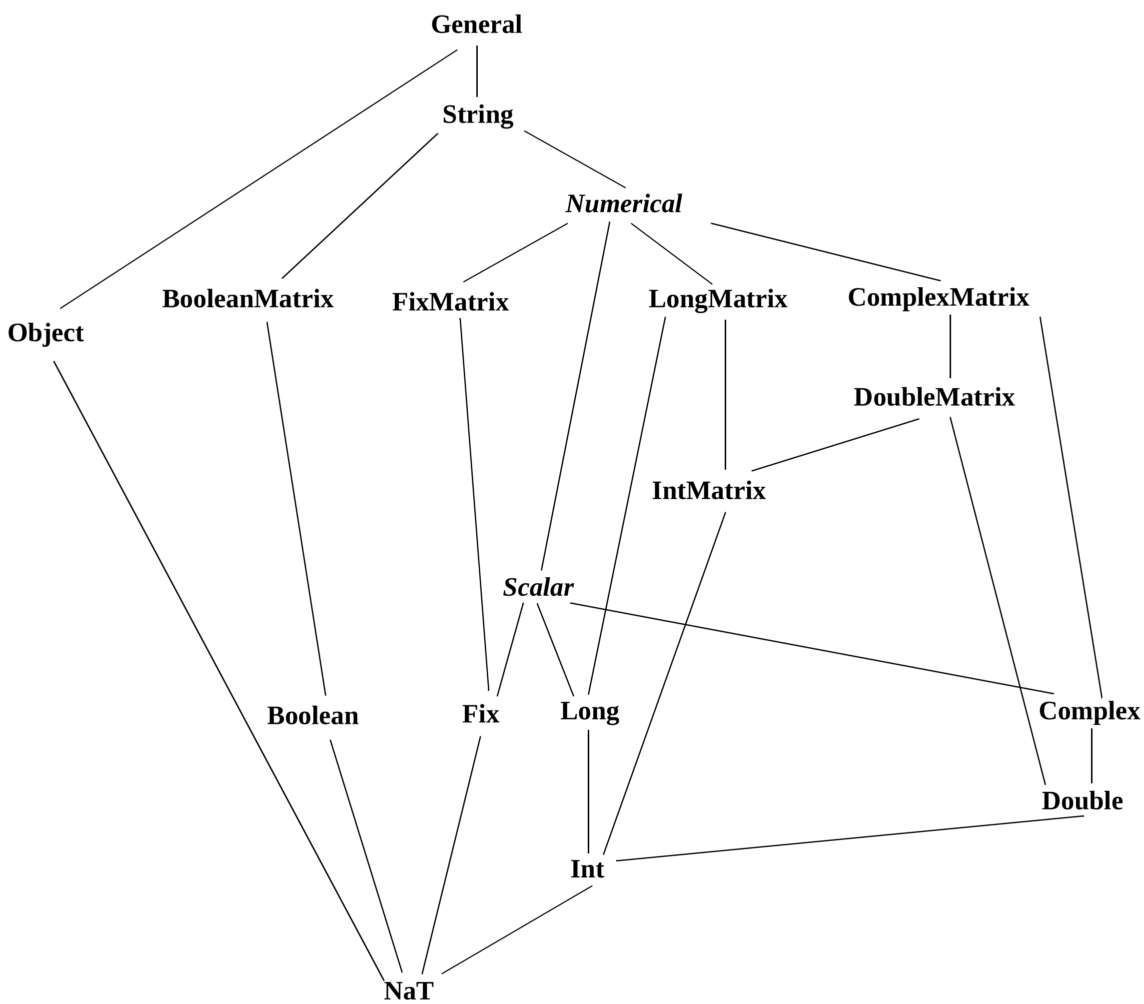


FIGURE 4.2. The type lattice.

`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 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()`, `longMatrix()`, `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.

4.3.3 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.

4.4 Parameters

A Parameter in Ptolemy II is a container for a Token. Ptolemy II has only a single Parameter class, shown in figure 4.3, in contrast to Ptolemy 0.x, which had a plethora of type-specific parameter classes. The Parameter class is derived from ptolemy.kernel.util.Attribute and so that instances can be attached to any instance of NamedObj. Since most interesting classes in Ptolemy II are derived from NamedObj, most classes can have parameters attached to them.

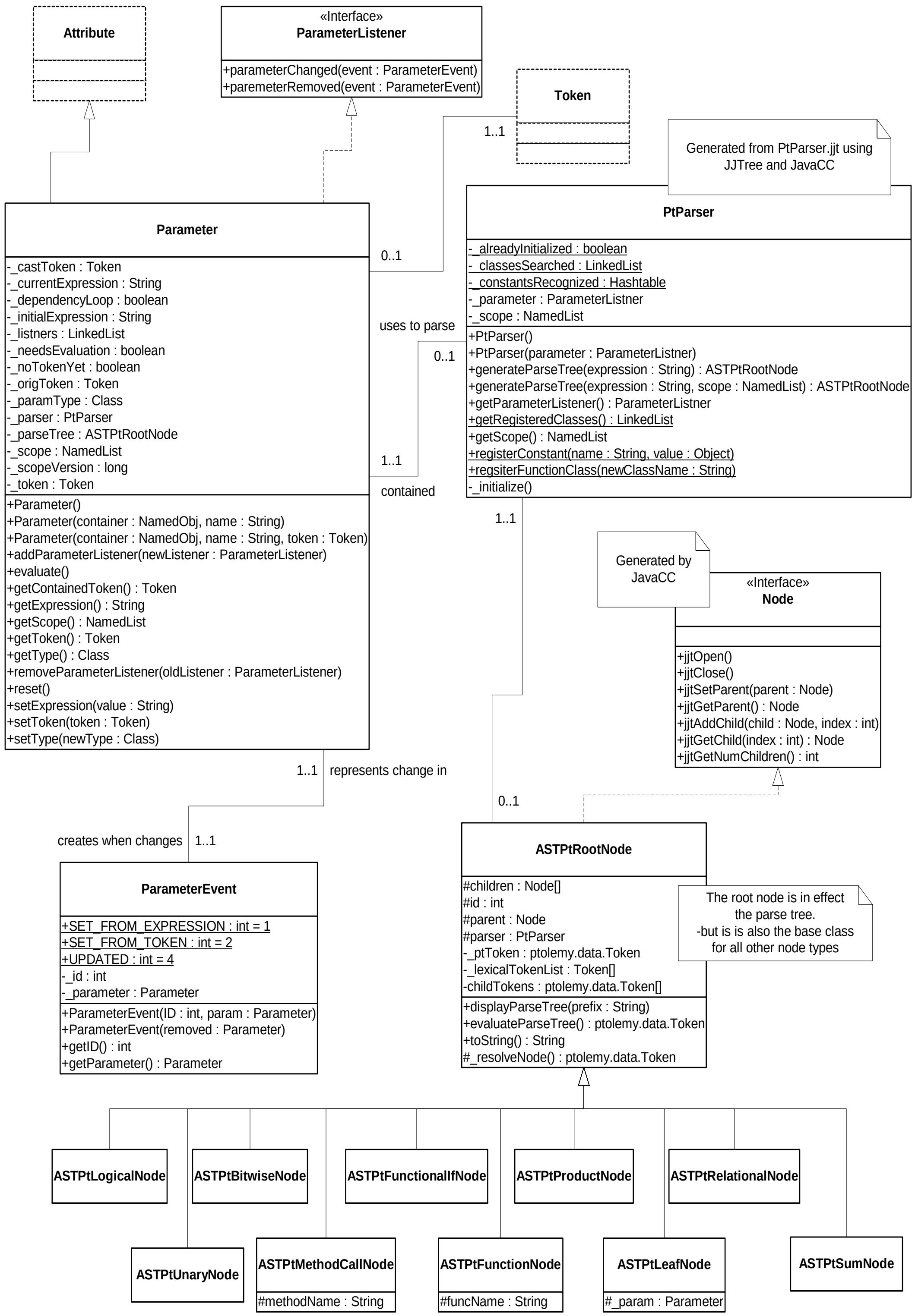


FIGURE 4.3. Static structure diagram for the data.expr package.

4.4.1 Values

The token stored in a parameter can be set in two ways, either by simply placing an instance of Token in the parameter, or by giving the parameter an expression (represented as a string) which is evaluated to produce a token. If the parameter is given a string expression, the expression must be evaluated via a call to `evaluate()`. The expression language recognized by parameters is explained below. If there is an expression waiting to be evaluated and `getToken()` is invoked, then the expression will be automatically evaluated before returning the token. As an example, consider the following code segment:

```
Parameter p = new Parameter();
p.setExpression(“1.6 + 8.3”);
p.evaluate();
System.out.println(p.getToken().stringValue());

// Now put a new token into the Parameter
DoubleToken t = new DoubleToken(7.7);
p.setToken(t);
System.out.println(p.getToken().stringValue());

// Now set the value from a String expression
p.setExpression("(true) ? 5.5 : \“hello\”");
System.out.println(p.getToken().stringValue());
```

The result of running this code segment will be:

```
9.9
7.7
5.5
```

The syntax of expressions is described below in section 4.5.

4.4.2 Variables

An expression given to a parameter can refer by *name* to other parameters within the **scope** of this parameter. The scope of a parameter is the set of parameters contained by the same NamedObj and those contained by the NamedObj one level up in the hierarchy (i.e. contained by the container of the container of this Parameter, if it has one). If a parameter may be referred to by other parameters, it cannot have any spaces in its name. In the above example, no container was specified, so there are no other parameters within the scope of the parameter. The following code segment illustrates expressions that use other parameters:

```
Entity e = new Entity();
Parameter p1 = new Parameter(e,“param1”, new DoubleToken(1.1));
Parameter p2 = new Parameter(e,“param2”, new DoubleToken(2.2));
Parameter p3 = new Parameter(e,“param3”);
p3.setExpression(p1.getName() + “ + param2“ );
System.out.println(p3.getToken().stringValue());
```

This results in

```
3.3
```

Note therefore that parameters can be used as variables. If two parameters have complicated expressions with a common factor, this common factor could be placed in a separate parameter to be referenced by the other parameters. The syntax of expressions is described below in section 4.5.

4.4.3 Type

The **type** of a parameter is defined to be the type of the first token placed in it (which may be the result of evaluating an expression, or may be given directly). For example if the first Token placed in a Parameter is of type DoubleToken, then the parameter is of type double. Any Token that can be converted in a lossless fashion to a token matching the type of the parameter can be placed in the parameter. Thus, for example, an IntToken could be placed in a parameter of type double, but a ComplexToken could not. An attempt to do so will trigger an exception. Lossless type conversion will be covered below.

The type of a parameter can be modified via a call to `setType()`. The new type of the parameter must be compatible with the token already in the Parameter. This can be used to relax the type of a parameter. For example you may want to have a parameter constrained to tokens of type double, even though the first token placed in it was an IntToken. The parameter could not contain a DoubleToken until `setType()` is called with the appropriate argument.

The `setType()` method can also be used to constrain the type of a Parameter. For example if the first token placed in a parameter was a DoubleToken, but it now contains an IntToken, then you could constrain the parameter to only contain IntTokens from now on by calling `setType()` with an IntToken class argument.

At any stage, the type of a parameter may be different from the type of the token contained in it. If this is the case, then the getToken() method will convert the token contained by the parameter into a token of the type of the parameter. Note that this can always be done because of the constraints on the type described above. For example, if a parameter contains an IntToken with value 1, but it is of type DoubleToken, then getToken() will return a DoubleToken with value 1.0. To access the token contained by a parameter, without it being converted to the type of the parameter, the getContainedToken() method should be used.

4.4.4 Dependencies

If an object (generally another parameter) wants to be kept notified of changes in a particular parameter, it must implement the `ParameterListener` interface and register itself as a listener of that parameter. When the parameter evaluates an expression which refers to another parameter within its scope, it is registered as a listener with that parameter. When a parameter changes it creates a `ParameterEvent` which contains information about how it changed, and passes it to all registered listeners by calling their `parameterChanged()` method. In this way a parameter gets updated whenever the value of a parameter it references changes. For example, if *param1* and *param2* are two parameters attached to the same `NamedObj`, both with the same names as the variables used to reference them, then after the following code

```
param1.setToken(new DoubleToken(5.5));
```

```
param2.setExpression(param1 + 2);
param2.evaluate();
```

param2 will be registered as a ParameterListener of *param1*. Then if the Token in *param1* changes, it will create a `ParameterEvent` and pass it to *param2* by invoking `parameterChanged()` on *param2*. In this way *param2* gets notified of any changes in *param1* and updated. This mechanism, of course, is the classic publish-and-subscribe design pattern.

4.5 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.

4.5.1 The Ptolemy II Expression Language

The Ptolemy II expression language uses elements of the syntax of Java. Unlike Java, however, it uses operator overloading. 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 and string. We expect the language to eventually support all the matrix types as well. 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 4.3.2). 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. An example is given above in section 4.4.2. 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 only the constants PI and E are registered, but as we will see later, this can easily be extended by a user. 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.

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.UtilityFunctions class. As we will see below in section 4.5.2, 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 required. 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.

4.5.2 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”)`.

4.5.3 Limitations

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

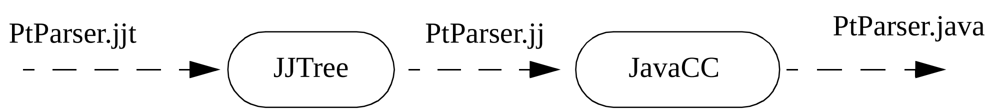
- The class ptolemy.data.util.UtilityFunctions containing the utility functions has not yet been fully written. Currently the functions available are `eval()` and `readFile()`.
- Functions in the math package need to be supported in much the same way that java.lang.Math is supported.
- Method calls are 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 A: 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.

A.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.

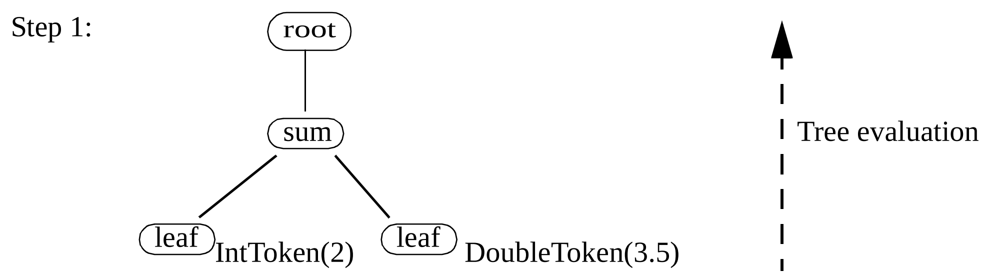
One of the key properties of the expression language is the ability to refer to other tokens by name. Since an expression that refers to other parameters may need to be evaluated several times (when the referred parameter changes), it is important that the parse tree does not need to be recreated every 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.

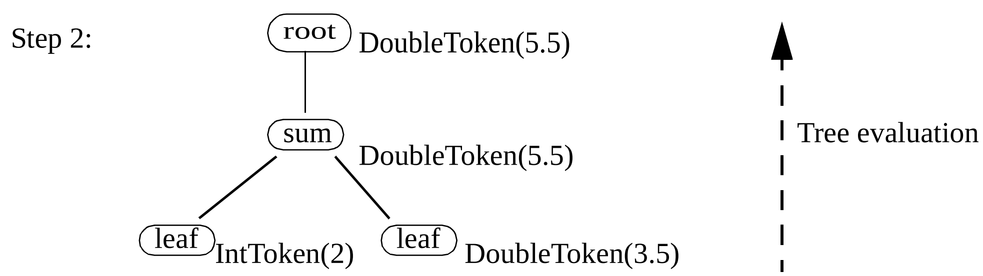
A.2 Evaluating the parse tree

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.

A.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.

A.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.

5

Graph

Jie Liu
Yuhong Xiong

5.1 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 heirarchy 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 existence 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.

5.2 Classes and Interfaces in the Graph Package

Figure 5.1 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-

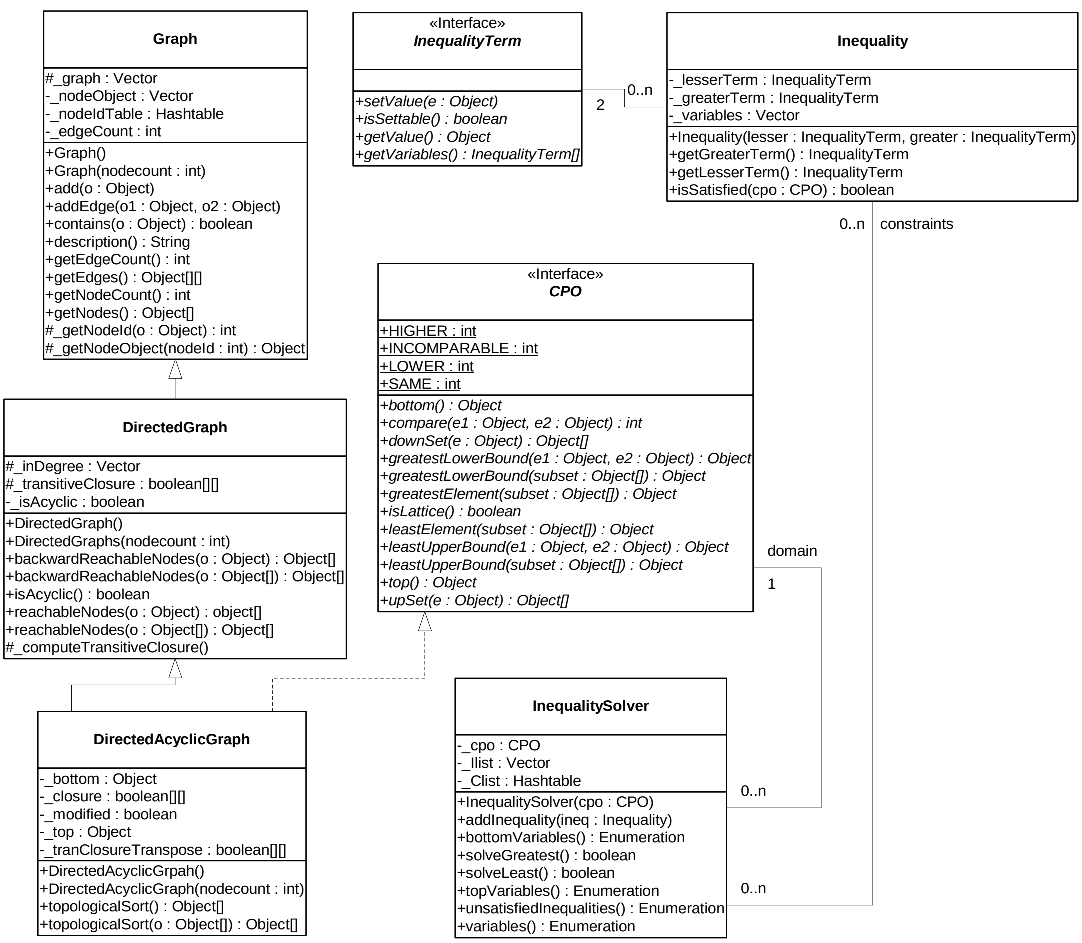


FIGURE 5.1. Classes in the graph package

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.

5.2.1 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 5.2(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 5.2(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.

5.2.2 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

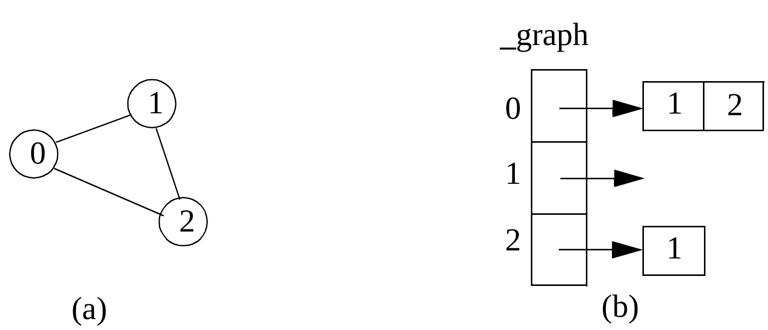


FIGURE 5.2. An undirected graph

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.

5.2.3 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 [10]. 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.

5.2.4 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 5.3:

$\alpha \le w$

$\beta \le x \wedge \alpha$

$\alpha \le \beta$

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

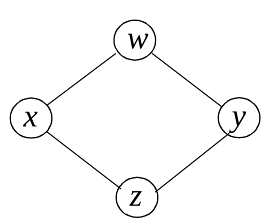


FIGURE 5.3. A 4-point CPO that also happens to be a lattice.

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 [34]. 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.

5.3 Example Use

5.3.1 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();
}
```

5.3.2 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 5.3, forms a set of constraints with three inequalities, and solves for both the least and greatest solutions. The inequalities are $a \le w$; $b \le a$; $b \le z$, 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;
    }
}
```

6

Types

Yuhong Xiong
Edward A. Lee

6.1 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 [8].

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 6.1 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-

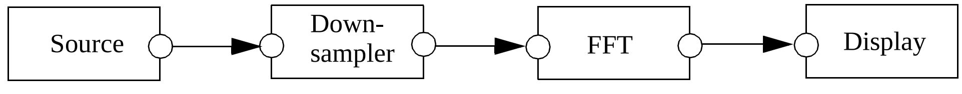


FIGURE 6.1. An imaginary Ptolemy II application

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 6.1 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 6.1, 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 4.2). 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 [31] 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 6.1 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.

6.2 Formulation

6.2.1 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 4.2. I.e.,

$$outType \leq inType \tag{1}$$

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 containing 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 (1). 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 6.2.

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:

$$int \le \alpha$$
$$double \le \beta$$
$$\gamma \le double$$

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:

$$\alpha \le \gamma$$
$$\beta \le \gamma$$
$$\gamma \le Complex$$

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 [28]. 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 [34], which is described in the appen-

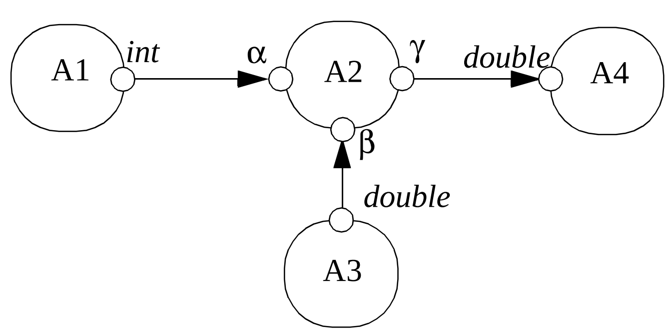


FIGURE 6.2. A topology with types.

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 6.2 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.

6.2.2 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.

6.3 Implementation Classes

6.3.1 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 6.3. 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 6.2, 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:

```
// 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:

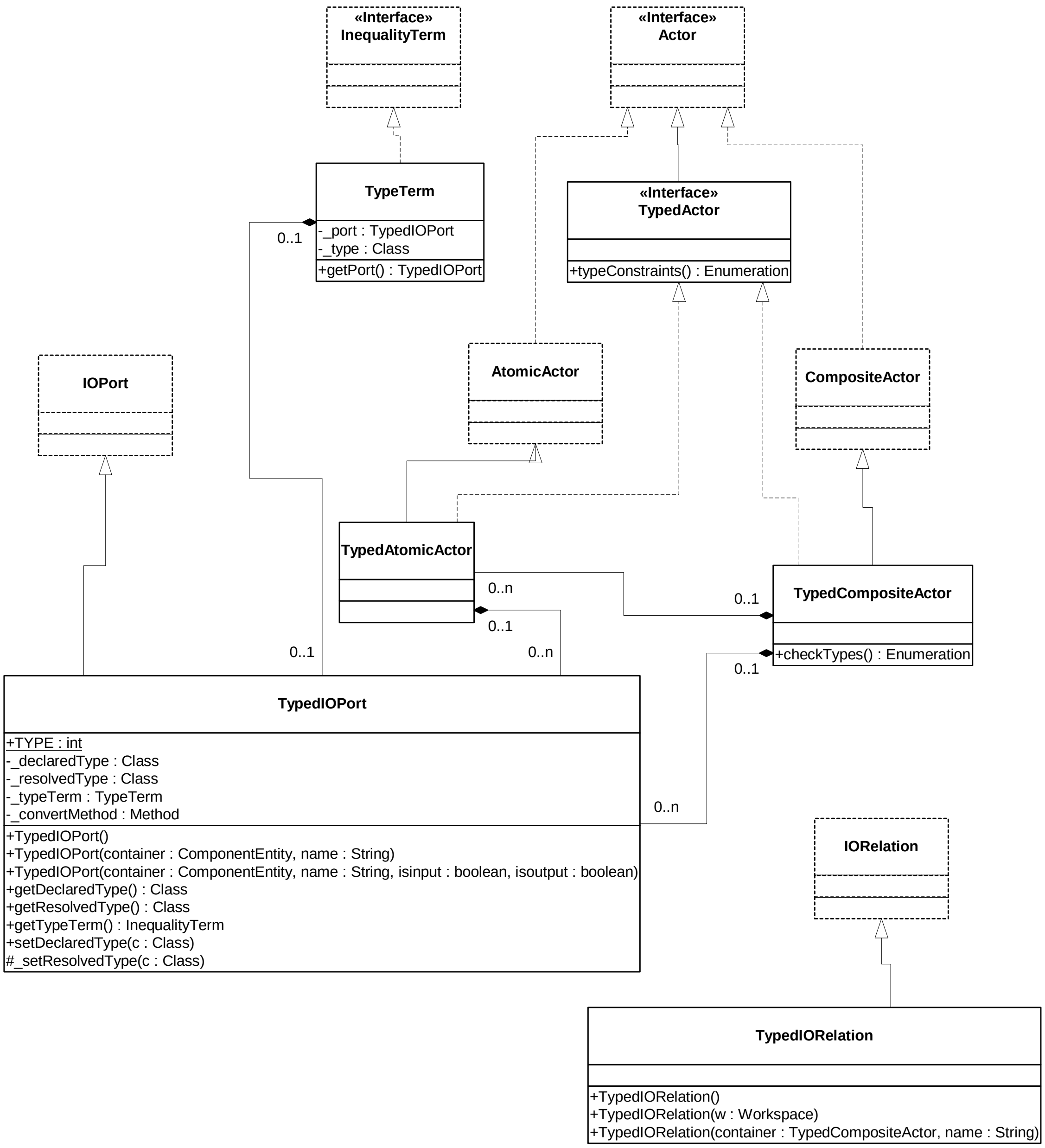


FIGURE 6.3. Classes in the actor package that support type checking.

```
// 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 6.1. It also covers most of the constraints for arithmetic actors such as the adder in figure 6.2. 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.

6.3.2 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.

6.4 Examples

6.4.1 Polymorphic Downsampler

In figure 6.1, if the Downsampler is designed to do downsampling for any kind of token, its type constraint is just *samplerIn* ≤ *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 ≤ *samplerIn*

samplerIn ≤ *samplerOut*

samplerOut ≤ *Complex*

Complex ≤ *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 run-time, 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.

6.4.2 Fork Connection

Consider two simple topologies in figure 6.4. where a single output is connected to two inputs in 6.4(a) and two outputs are connected to a single input in 6.4(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 6.4(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 6.4(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.

6.4.3 A Sampler System

Figure 6.5 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 signal" 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 6.5 assumes the output type is *Double*. The input port of the Ramp

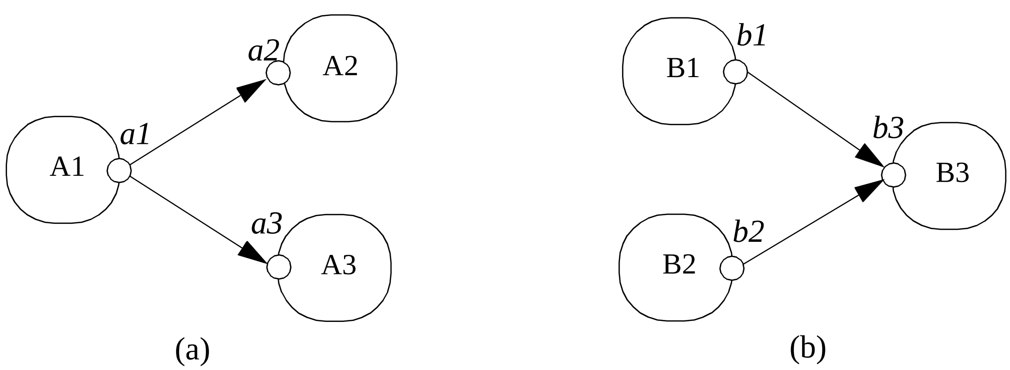


FIGURE 6.4. Two simple topologies with types.

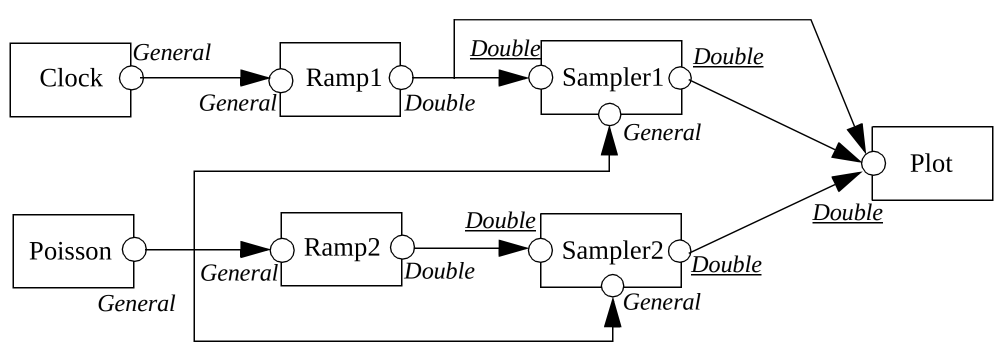


FIGURE 6.5. A Sampler system built in the DE domain.

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 B: 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 6.2, *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

$$\alpha = int \vee NaT = int \qquad (2)$$

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 (2) 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 (2). The evaluation reaches a fixed point when a set of type variable assignments satisfying the constraints in *Cvar* is found.

Rehof and Mogensen [34] 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.

7

Plot

Edward A. Lee
Christopher Hylands

7.1 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 7.1. 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).

7.2 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-

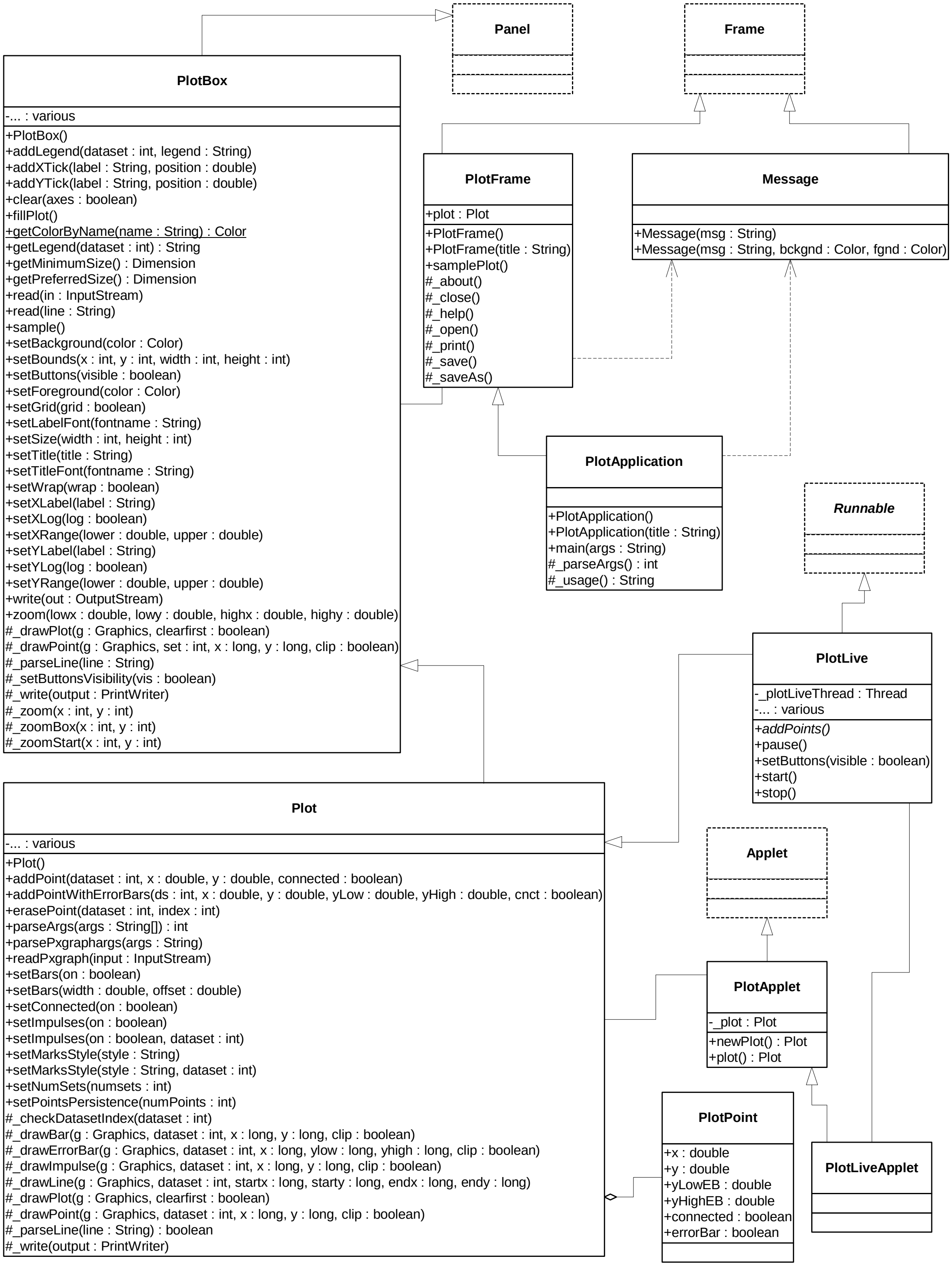


FIGURE 7.1. The classes of the plot package.

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.

7.3 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.

7.3.1 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.

7.3.2 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.

7.4 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.

7.5 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 EPS export for graphs.
- There is no mechanism for customizing the colors used in a plot.

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] G. R. Andrews, *Concurrent Programming — Principles and Practice*, Addison-Wesley, 1991.

[4] 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.

[5] 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.

[6] G. Berry and G. Gonthier, “The Esterel synchronous programming language: Design, semantics, implementation,” *Science of Computer Programming*, 19(2):87-152, 1992.

[7] 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/papers/JEurSim).

[8] Luca Cardelli, *Type Systems*, Handbook of Computer Science and Engineering, CRC Press, 1997.

[9] 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.

[10] B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[11] 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/)

[12] M. Fowler and K. Scott, *UML Distilled*, Addison-Wesley, 1997.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading MA, 1995.

[14] 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/papers/98/starcharts)

[15] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.*, vol 8, pp. 231-274, 1987.

[16] P. G. Harrison, "A Higher-Order Approach to Parallel Algorithms," *The Computer Journal*, Vol. 35, No. 6, 1992.

[17] 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.

[18] 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.

[19] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[20] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.

[21] D. Lea, *Concurrent Programming in Java^{TM}*, Addison-Wesley, Reading, MA, 1997.

[22] 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/papers/processNets)

[23] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation,", March 12, 1998. (Revised from ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997).
(http://ptolemy.eecs.berkeley.edu/papers/98/framework/)

[24] 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.

[25] S. McConnell, *Code Complete : A Practical Handbook of Software Construction*, Microsoft Press, 1993.

[26] B. Meyer, *Object Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.

[27] R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[28] R. Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences 17, pp. 384-375, 1978.

[29] NASA Office of Safety and Mission Assurance, *Software Formal Inspections Guidebook*, August 1993 (http://satc.gsfc.nasa.gov/fi/gdb/fitext.txt).

[30] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.

[31] J. K. Ousterhout, *Scripting: Higher Level Programming for the 21 Century*, IEEE Computer magazine, March 1998.

[32] 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/papers/parksThesis).

[33] Rational Software Corporation, *UML Notation Guide*, Version 1.1, September 1997, http://www.rational.com/uml/html/notation/.

[34] Jakob Rehof and Torben 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.

[35] A. J. Riel, *Object Oriented Design Heuristics*, Addison Wesley, 1996.

[36] J. Rowson and A. Sangiovanni-Vincentelli, “Interface Based Design,” *Proc. of DAC ‘97*.

[37] J. Rumbaugh, *et al*. *Object-Oriented Modeling and Design* Prentice Hall, 1991.

[38] J. Rumbaugh, *OMT Insights*, SIGS Books, 1996.

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 0.x.

anytype.............................. The Ptolemy 0.x name for *undeclared type*.

atomic actor....................... A primitive actor. That is, one that is not a composite actor. This was called a *star* in Ptolemy 0.x.

attribute .... A named property associated with a named object in Ptolemy II.

block.................................. The Ptolemy 0.x name for an *actor.*

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.

composite actor .... An actor that is internally composed of other actors and relations. This was called a *galaxy* in Ptolemy 0.x.

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 polymorphic.............. Capable of operating 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 0.x.

domain polymorphic......... Capable of operating 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 0.x 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 0.x.

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 0.x.

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 0.x.

particle............................... The Ptolemy 0.x name for a *token*.

port .... A named interface of an entity to which connections be made.

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.

star .... The Ptolemy 0.x name for an *atomic actor.*

state.................................... The Ptolemy 0.x 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 0.x.

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 0.x.

universe.............................. The Ptolemy 0.x 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 0.x name for an *opaque composite actor.*

Index

H

I

J