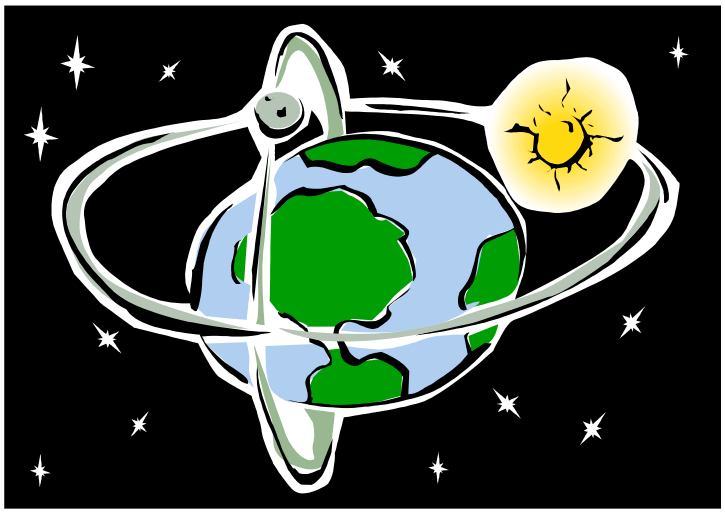
-PTOLEMY II-



HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

John Davis, II
Ron Galicia
Mudit Goel
Christopher Hylands
Edward A. Lee
Jie Liu
John Reekie
Neil Smyth
Yuhong Xiong

Department of Electrical Engineering and Computer Science
University of California at Berkeley

Note: This document is a draft of the final design document for Ptolemy II. It reflects work in progress. Most is still under design, and thus is subject to fairly radical changes.

— DRAFT — NOT FOR DISTRIBUTION —

“Java” is a registered trademark of Sun Microsystems.

Contents

1

Introduction

1.1 Objectives

Ptolemy II is a complete, from the ground up, redesign of the Ptolemy design environment, which supports heterogeneous concurrent modeling and design [4]. Some of the major capabilites that we believe to be new technology in design and simulation environments include:

- *Higher level concurrent design in Java™*. Java support for concurrent design is very low level. Maintaining safety and liveness can be quite difficult [12]. Ptolemy II provides a number of *domains* that support design of concurrent systems at a much higher level of abstraction. These include simulation models of various types, process networks, communicating sequential processes (rendezvous based), dataflow, synchronous/reactive modeling, continuous-time modeling, and hierarchical concurrent finite-state machines.
- *Interoperability through software components*. Ptolemy II uses CORBA in a number of ways. Components (actors) in a Ptolemy II application may be implemented on a remote server via CORBA. Also, components may be parameterized where parameter values are supplied by a CORBA server (this mechanism supports *reduced-order modeling*, where the model is provided by the server). Finally, a Ptolemy II application can be exported via a CORBA server.
- *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 CAD software, where design tools usually consist of 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.* The Ptolemy wormhole mechanism for coupling heterogeneous models of computation is improved to provide better support for models of computation that are very different from dataflow, the best supported model in prior versions of Ptolemy. These will include hier-

archical concurrent finite-state machines and a variety of continuous-time modeling techniques.

- *Thread-safe concurrent execution*. Ptolemy applications are typically concurrent, but in the past, support for concurrent execution of a Ptolemy application has been primitive. Ptolemy II supports concurrency throughout, allowing for instance for an application 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 [10] 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 [15][21][23] and design patterns [7] 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 [14][17].
- *A truly polymorphic type system*. Earlier implementations of Ptolemy support rudimentary polymorphism through the "anytype particle." Even with such limited polymorphism, type resolution has proved challenging, and current implementations are 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 will consist of finding a fixed point, using algorithms recently developed within the project [5]..
- *Improved design refinement*. 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 compatability across implementation technologies is assured [22].

1.2 Package Structure

The package structure is shown in figure 1.1. The role of each of the packages is explained below.

- *kernel*. This package provides the software architecture for the key abstract syntax, clustered graphs. The classes support entities with ports and relations that link the ports. Clustering is where a collection of entities is encapsulated in a single entity and a subset of the ports of the inside entities are exposed as ports of the cluster entity.
- *actor*. This package supports executable entities that receive and send data through ports. It includes a library of polymorphic actors.
- *automata*. This package supports sequential computation where entities represent a state or phase of computation.
- *domains*. This set of packages support particular models of computation.
- *actor libraries*. This set of packages collect actors for particular models of computation or groups of models of computation.
- *graph*. This package provides algorithms for manipulating and anlyzing mathematical graphs.
- *data*. This package provides classes that encapsulate and manipulate data that is transported by Ptolemy applications. It supports a rich expression parser and interpreter.
- *math*. This package encapsulates mathematical functions and methods for operating on matrixes and vectors.
- *plot*. This package provides 2-dimensional signal plotting widgets.

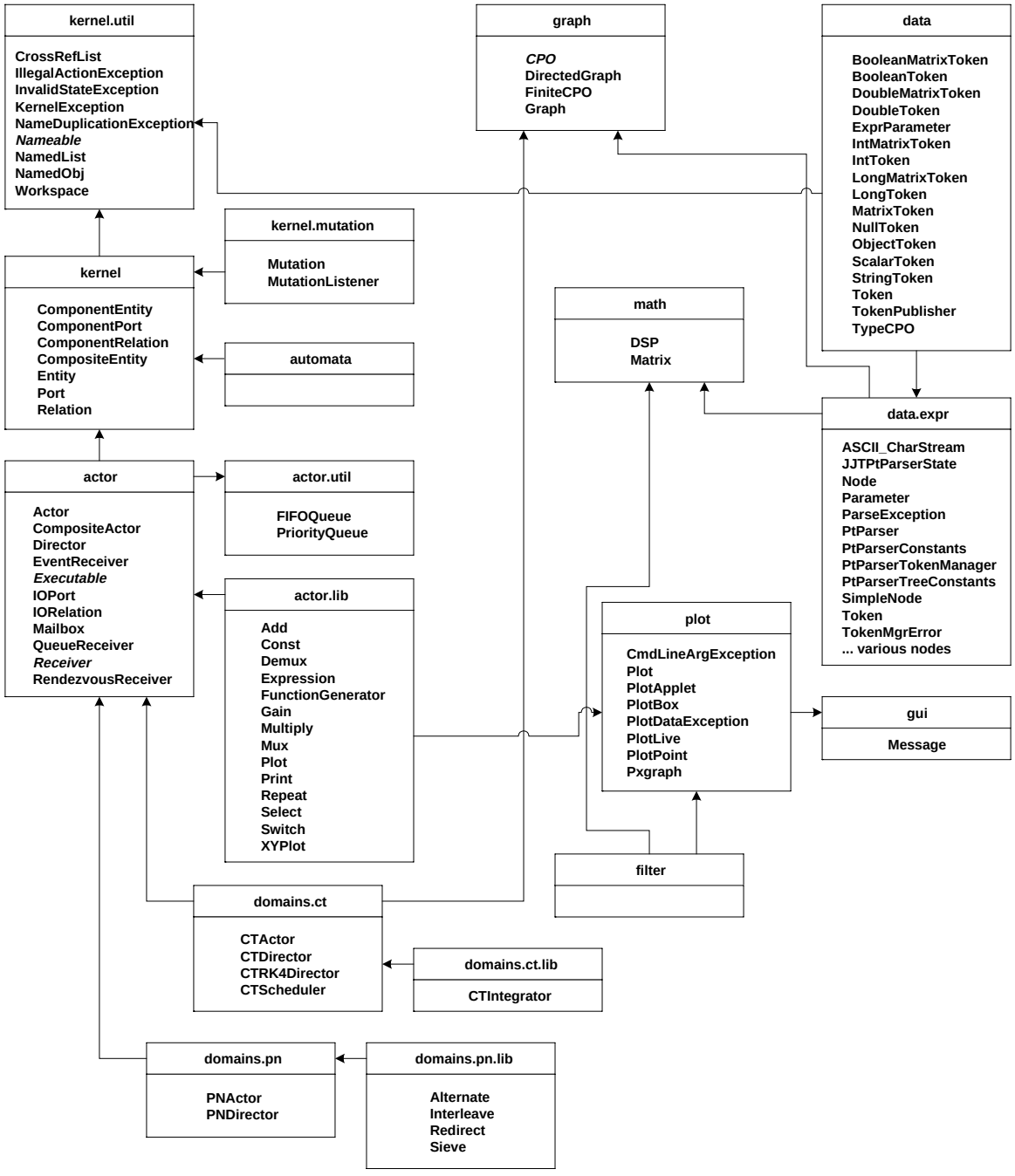


FIGURE 1.1. The package structure of Ptolemy II.

- *filter*. This package encapsulates a set of classes supporting signal processing.

2

The Kernel

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 [4], 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 mathe-

matical 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*. [7]. “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 [6][20]. 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 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 inhertited. 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.

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

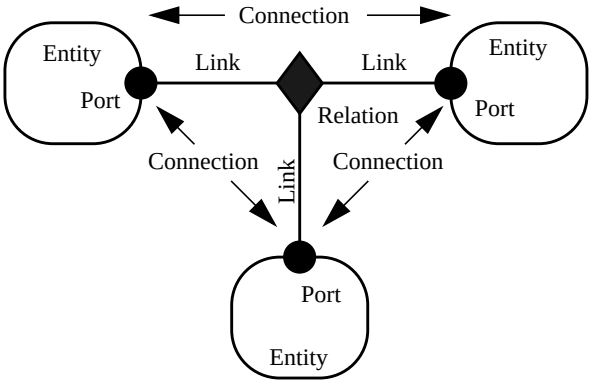


FIGURE 2.1. Visual notation and terminology.

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 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 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 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.

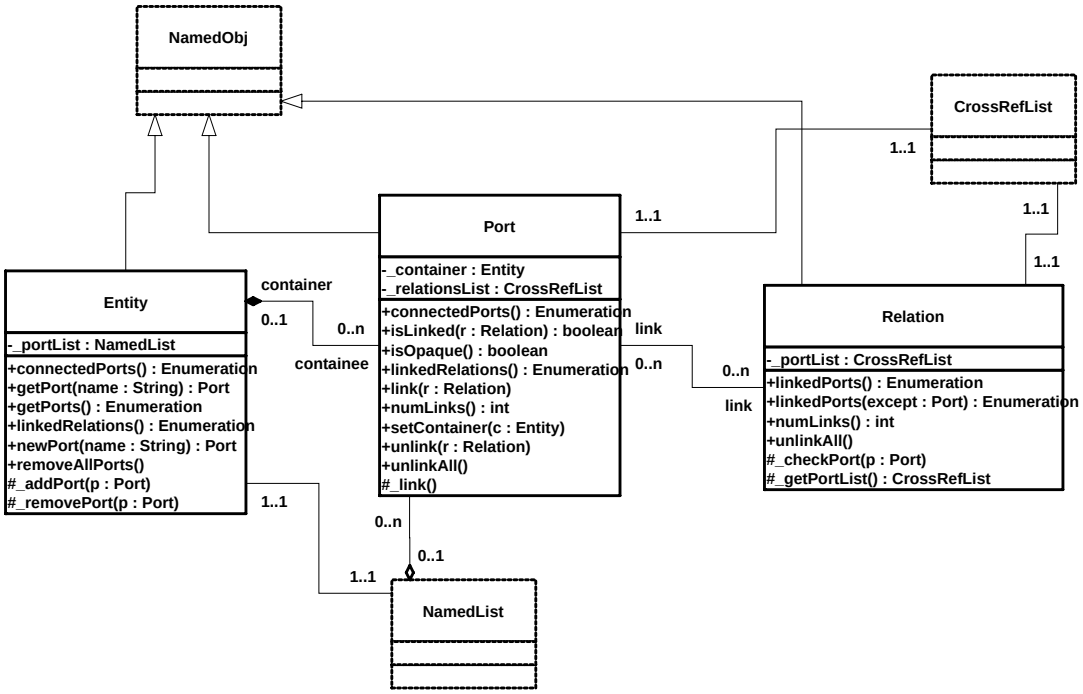


FIGURE 2.2. Key classes in the kernel package and their methods supporting basic (non-hierarchical) topologies. Methods that override those defined in a base class or implement those in an interface are not shown. The “+” indicates public visibility, “#” indicates protected, and “-” indicates private.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

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 parenthese that delimit the argument list.

2.3 Ptolemy II Conventions

We have made an effort to be consistent about naming of classes, methods and members. Class names are capitalized, with internal word boundaried 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.

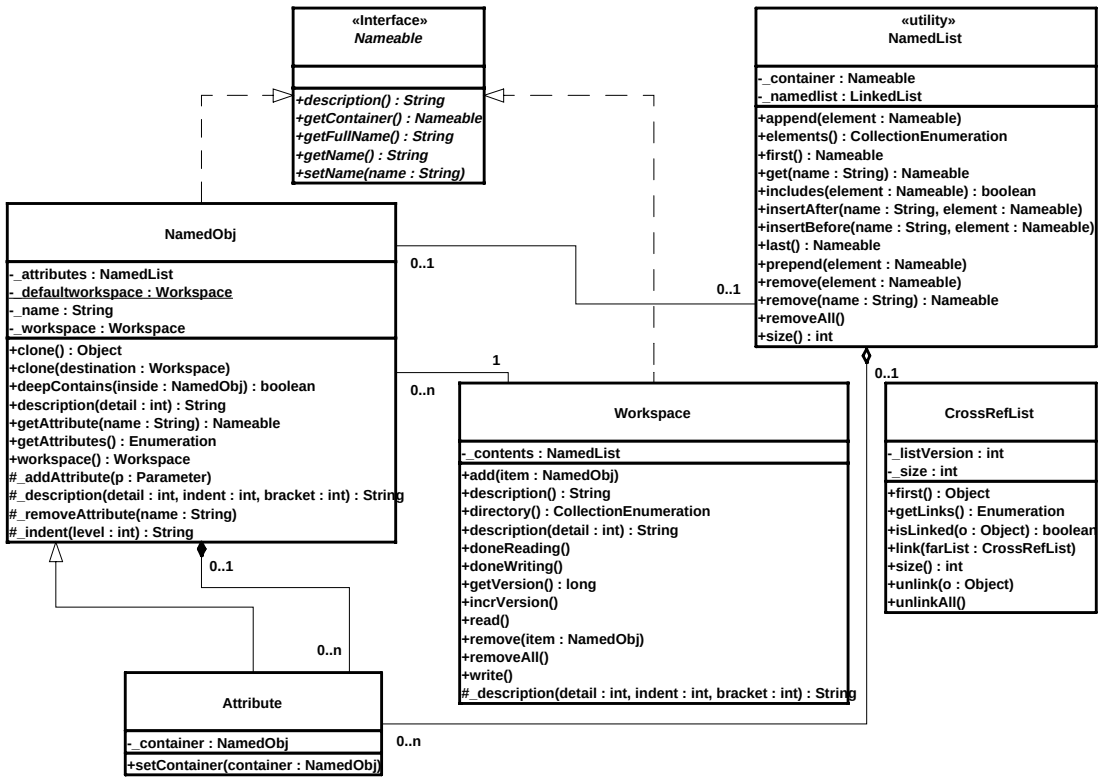


FIGURE 2.3. Support classes in the kernel.util package.

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. 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 at the top of figure 2.2. 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 involes 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 rudi-

mentary 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" [12] 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 hiearchy 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 [24], 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 2.2. 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 Named-

Obj 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 [7].

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 [16]. 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[1]. 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

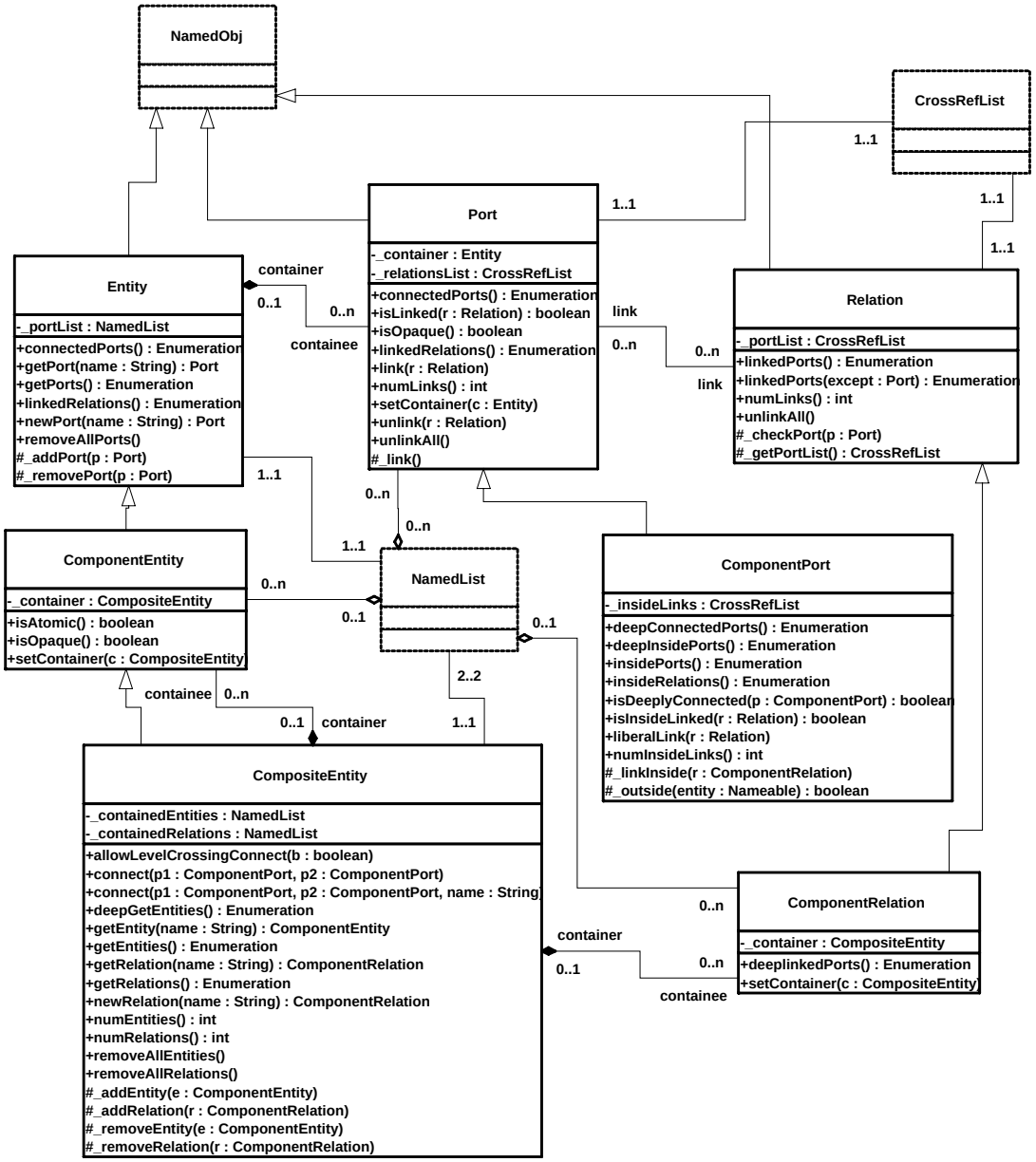


FIGURE 2.4. Key classes supporting clustered graphs.

1. Unless level-crossing links are allowed, which is discouraged.

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, 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.

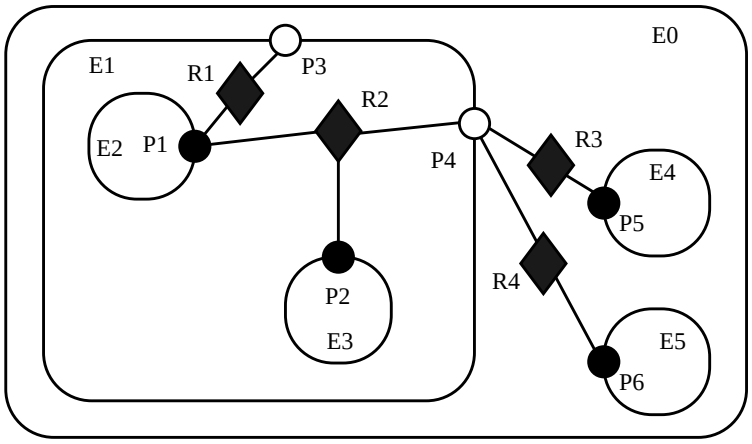


FIGURE 2.5. Transparent ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

1. 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.

2.6.2 Level-Crossing Connections

For a few applications, such as Statecharts [8], 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-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.

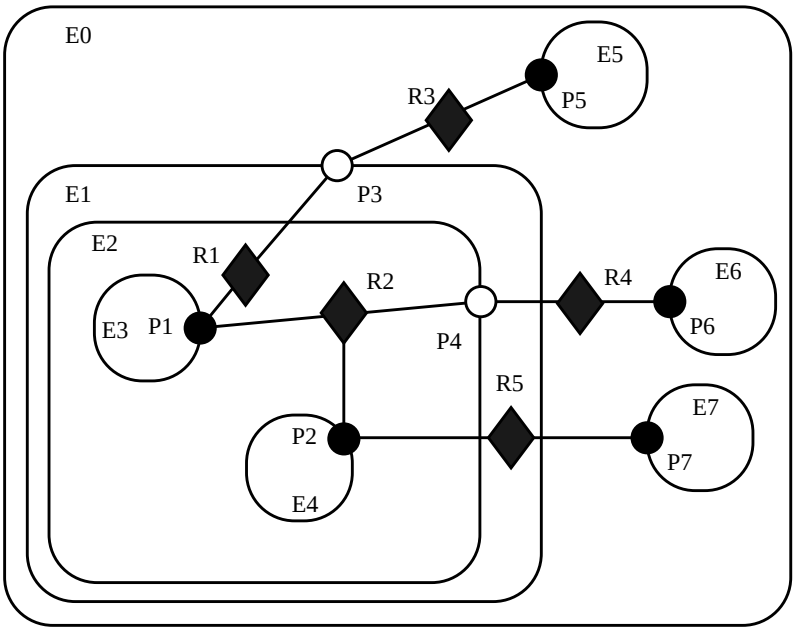


FIGURE 2.6. An example with level-crossing transitions.

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 deep graph traversals will see right through it.

Support for tunneling entities is a major increment in capability over the previous Ptolemy kernel [4] (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 [13] 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

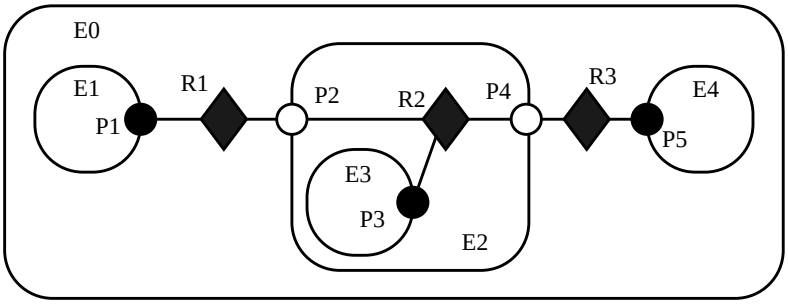


FIGURE 2.7. A tunneling entity contains a relation with inside links to more than one port.

"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.

There is a key restriction to cloning. Because they breaks 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.

```
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.

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 [18] 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 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.

2.6.7 Mutations

Often it is necessary to carefully constrain when changes can be made in a topology. For example,

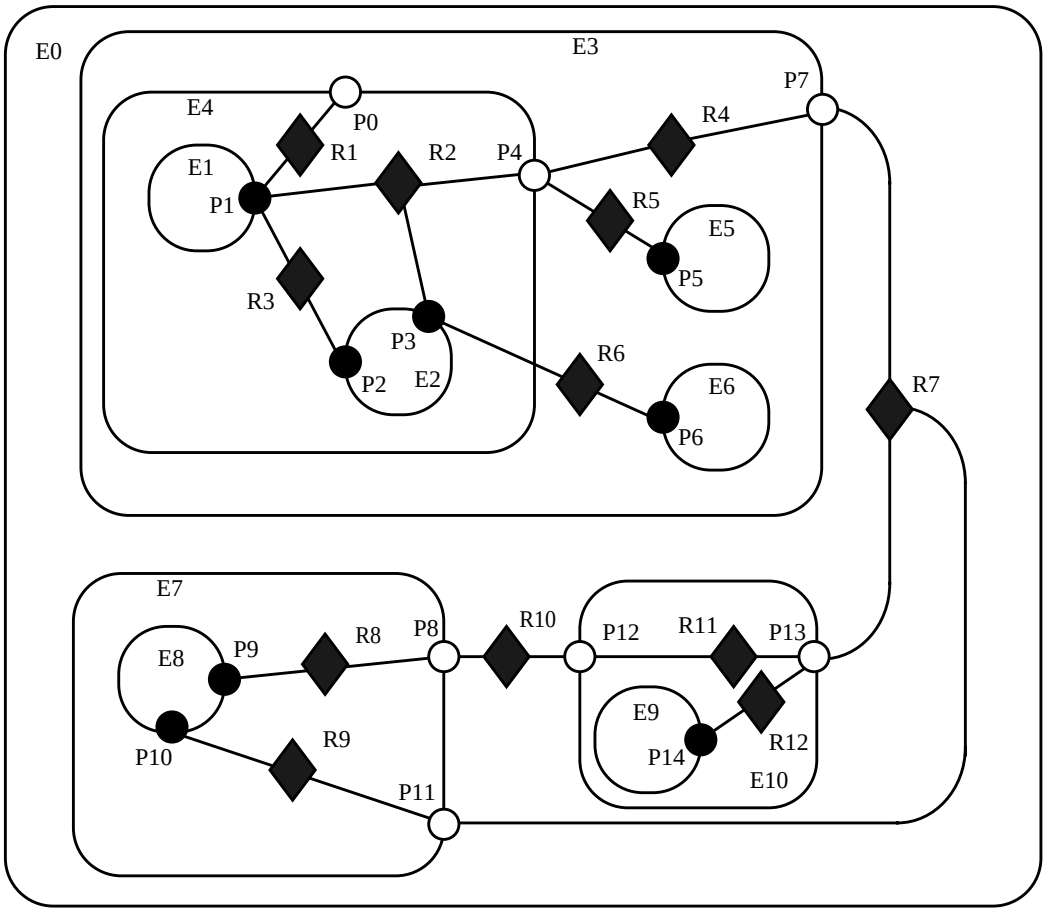


FIGURE 2.9. An example of a clustered graph.

```
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.

```
# 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 it.

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. A subpackage of the kernel, called the mutation package, provides support for carefully controlled mutations. This subpackage contains only two interfaces, shown in figure 2.13.

The typical usage pattern is to create an object that implements Mutation and queue it with whatever object is in charge of the control flow (whatever object knows when it is safe to perform mutations). That object then performs the mutation when it is safe to do so. In addition, it informs any registered listeners of the mutation so that they can react accordingly.

For example, the Director class in the actor package allows mutations to occur only between itera-

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.

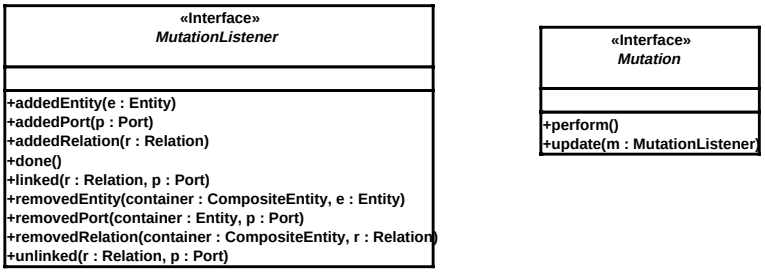


FIGURE 2.13. Interfaces in the kernel.mutation package.

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

The Mutation interface is typically used by creating an anonymous inner class that implements the interface. The class should implement two methods, perform() and update(). The first of these actually performs the mutation, and the second informs any listeners of the mutation. Here is a section of code that defines an instance of Mutation that creates a new entity:

```
import pt.kernel.mutation.*;

Mutation m = new Mutation() {
    private _newentity
    public void perform() {
        _newentity = new Entity("foo");
    }
    public void update(MutationListener listener) {
        listener.addedEntity(_newentity);
        listener.done();
    }
};
```

The instance *m* of this anonymous inner class would then typically be registered with a controller, such as a director, that can invoke the perform() method when it is safe to perform the mutation.

In addition, the controller invokes the update() method, passing it any registered listeners. A listener is any object that needs to be informed of the mutation once it has occurred. For example, again in the Director class of the actor package, when a new entity is added to an application, it needs to be initialized. A listener is registered with the director so that after the actual addition of the new entity, the entity can be initialized by the listener. Such a listener implements the addedEntity() method of MutationListener to either directly perform the initialization, or queue the initialization to occur at a later time. The ActorListener class of the actor package is a simple example of a MutationListener (see the Actors chapter).

2.7 Composite Opaque 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*.

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).

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 2.14. 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 2.14(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 2.14(b), the code body represented by ellipses (...) can be executed only after a lock has been acquired on object *obj*.

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

One possible solution is to ensure that locks are always acquired in the same order [12]. 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 containment structure before the containment structure can be used to acquire a lock! Some policy could certainly be defined, but the resulting code would be difficult to guarantee. Moreover, testing for deadlock conditions is notoriously difficult, so we implement a more conservative, and much simpler strategy.

2.8.2 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

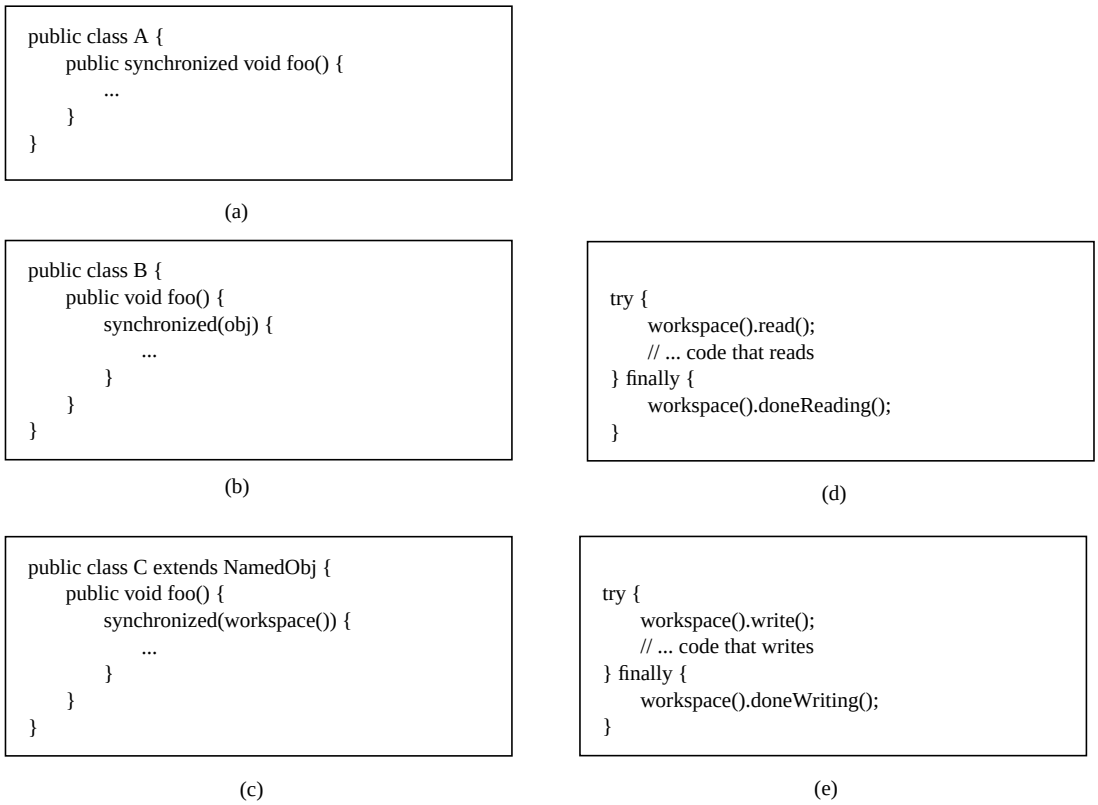


FIGURE 2.14. Using monitors for thread safety. The method used in Ptolemy II is in (d) and (e).

workspace (derived classes may be more liberal in certain circumstances). This “managed ownership” [12] is our central strategy in thread safety.

As shown in figure 2.14(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 2.14(d) and (e). In (d), a reader first calls the read() 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 a write lock on the workspace (the thread calling read() 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 2.14(e). The writer first calls the write() 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 2.14(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.

2.9 Exceptions

As a general rule, we use standard Java exceptions when they are appropriate. However, standard Java exceptions do not provide a uniform mechanism for reporting errors that takes advantage of their identification by full name. In order to obtain such uniformity, the Ptolemy II kernel has its own set of exceptions. These are summarized in the class diagram in figure 2.15.

2.9.1 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.

2.9.2 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. This exception supports all the constructor forms of KernelException.

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. The constructor forms are:

- NameDuplicationException(Nameable wouldBeContainee)
- NameDuplicationException(Nameable wouldBeContainee, String moreInfo)
- NameDuplicationException(Nameable container, Nameable wouldBeContainee)
- NameDuplicationException(Nameable container, Nameable wouldBeContainee, String moreInfo)

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. The constructor forms are:

- NoSuchItemException(String message)
- NoSuchItemException(Nameable container, String message)

2.9.3 Very 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.

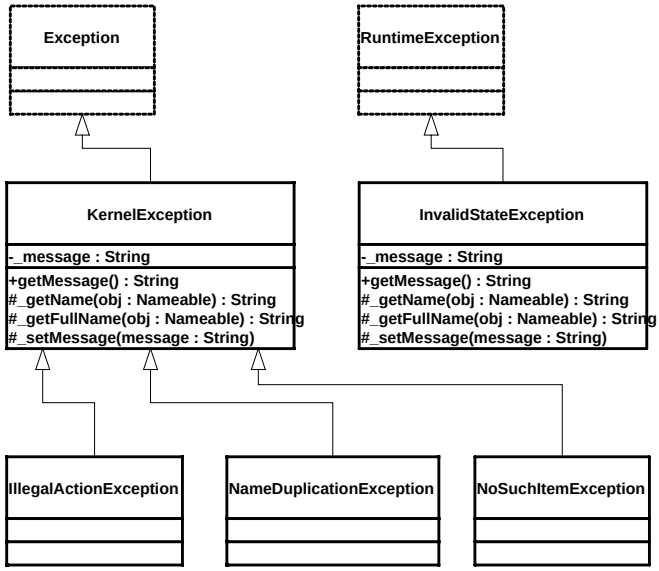


FIGURE 2.15. Summary of exceptions defined in the kernel.util package. These are used primarily through constructor calls. The form of the constructors is shown in the text. Exception and RuntimeException are Java exceptions.

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 supports all the constructor forms of KernelException, but is not derived from KernelException. It 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 supports only one constructor form, taking a string as an argument. It is derived from the Java RuntimeException.

3

Actors

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. 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 two key support functions. It supports message passing and the execu-

tion sequence. These are discussed in detail in the next two sections.

3.2 Message Passing

The actor package supports executable entities called *actors* that communicate with one another via message passing. Messages are encapsulated in *tokens* (see section 4 and figure 4.1). Messages are sent via ports. IOPort is the key class supporting message transport, and is shown in figure 3.1. An

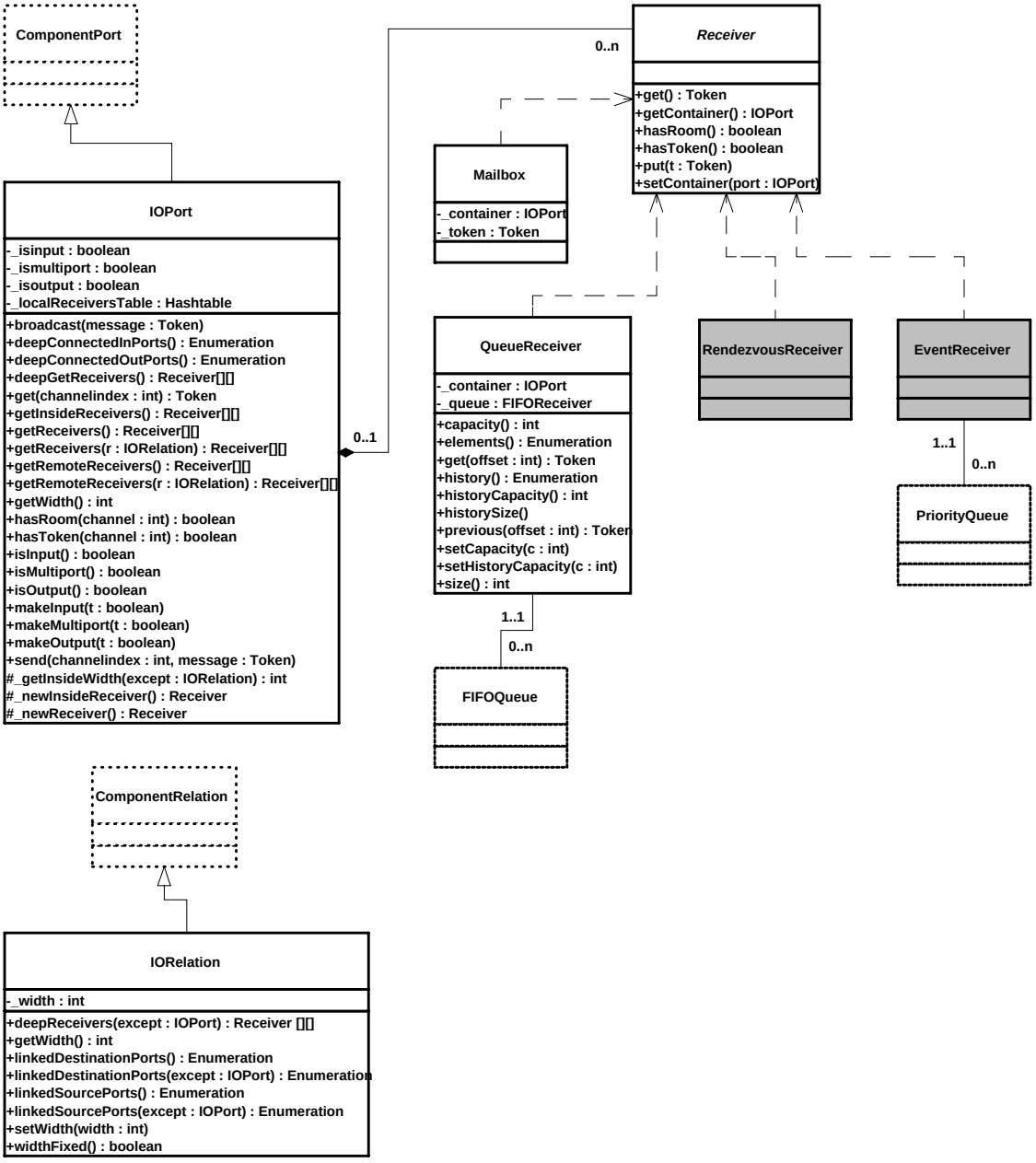


FIGURE 3.1. Port classes that support message passing under various communication protocols.

IOPort can only be connected to other IOPort instances, and only via IORelations. The IORelation class is also shown in figure 3.1.

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, or model of computation.

3.2.1 Data Transport

Data transport is depicted in figure 3.2. The originating actor E1 has an output port P1, indicated in the figure with an upwards arrow. The destination actor E2 has an input port P2, indicated in the figure with a downwards 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.

In figure 3.2 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 makeMultiport() 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.

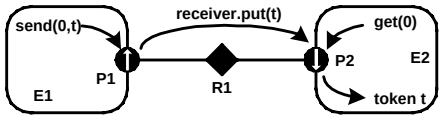


FIGURE 3.2. Message passing is mediated by the IOPort class. Its send() method obtains a reference to a remote receiver, and calls the put() method of the receiver, passing it the token *t*. The destination actor retrieves the token by calling the get() method of its input port.

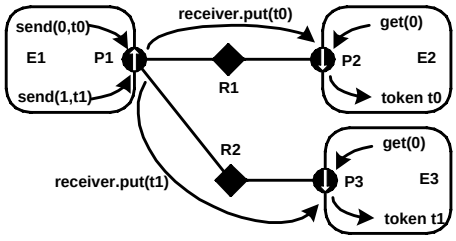


FIGURE 3.3. A port can support more than one channel, permitting an entity to send distinct data to distinct destinations via the same port. This feature is typically used when the number of destinations cannot be statically determined.

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*.

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 *clone* of the token is sent to all but the first destination (where the order is determined as usual by the order in which links are made). The first destination receives the original token. What is meant by a clone depends on the token, but for most simple tokens, a clone is simply a copy.

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

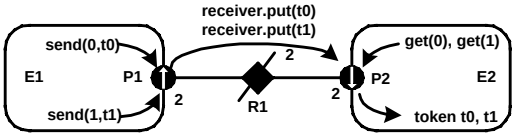


FIGURE 3.4. A bus is an IORelation that represents multiple channels. It is indicated by a relation with a slash through it, and the number adjacent to the bus is the width of the bus.

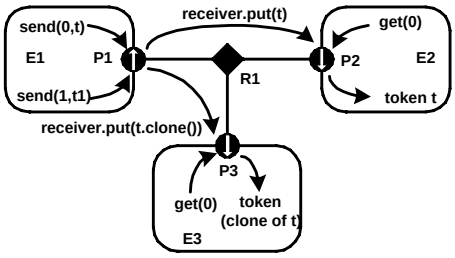


FIGURE 3.5. Channels may reach multiple destinations. This is represented by relations linking multiple input ports to an output port. It is accomplished by cloning the token that is sent.

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. Output ports are indicated with upwards arrows, input ports with downward arrows, and ports that are both an input and output with a bidirectional arrow. 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 a clone of these become the first two inputs of *P5*. The third output of *P1* goes nowhere. The fourth becomes the third input of *P5*, a clone becomes the first input of *P6*, and another clone goes to *P8*, which is both an input and an output. 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.

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 outputs are denoted with upwards arrows, inputs with downward arrows, and bidirectional ports with bidirectional arrows. 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 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

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

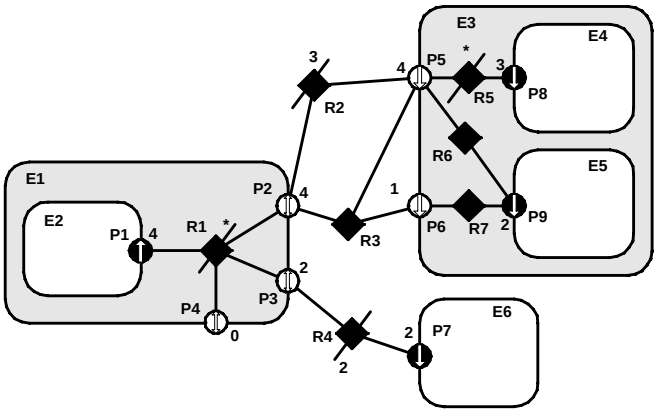


FIGURE 3.7. An example showing busses combined with input, output, and transparent ports.

```
# Top container
  set e0 [java::new pt.actor.CompositeActor]
  $e0 setExecutiveDirector $director
  $e0 setName E0
  # First level of the hierarchy
  set e1 [java::new pt.actor.CompositeActor $e0 E1]
  set p2 [java::new pt.actor.IOPort $e1 P2]
  $p2 makeMultiport true
  set p3 [java::new pt.actor.IOPort $e1 P3]
  $p3 makeMultiport true
  set p4 [java::new pt.actor.IOPort $e1 P4]
  $p4 makeMultiport true

  set e3 [java::new pt.actor.CompositeActor $e0 E3]
  set p5 [java::new pt.actor.IOPort $e3 P5]
  $p5 makeMultiport true
  set p6 [java::new pt.actor.IOPort $e3 P6]

  set e6 [java::new pt.actor.AtomicActor $e0 E6]
  set p7 [java::new pt.actor.IOPort $e6 P7]
  $p7 makeMultiport true
  $p7 makeInput true

  set r2 [java::new pt.actor.IORelation $e0 R2]
  $r2 setWidth 3
  set r3 [java::new pt.actor.IORelation $e0 R3]
  set r4 [java::new pt.actor.IORelation $e0 R4]
  $r4 setWidth 2

  $p2 link $r2
  $p2 link $r3
  $p3 link $r4
  $p5 link $r2
  $p5 link $r3
  $p6 link $r3
  $p7 link $r4

  # Inside E1
  set e2 [java::new pt.actor.AtomicActor $e1 E2]
  set p1 [java::new pt.actor.IOPort $e2 P1]
  $p1 makeMultiport true
  $p1 makeOutput true
  set r1 [java::new pt.actor.IORelation $e1 R1]
  $r1 setWidth 0
  $p1 link $r1
  $p2 link $r1
  $p3 link $r1
  $p4 link $r1

  # Inside E3
  set e4 [java::new pt.actor.AtomicActor $e3 E4]
  set p8 [java::new pt.actor.IOPort $e4 P8]
  $p8 makeMultiport true
  $p8 makeInput true
  set e5 [java::new pt.actor.AtomicActor $e3 E5]
  set p9 [java::new pt.actor.IOPort $e5 P9]
  $p9 makeMultiport true
  $p9 makeInput true
  set r5 [java::new pt.actor.IORelation $e3 R5]
  $r5 setWidth 0
  set r6 [java::new pt.actor.IORelation $e3 R6]
  set r7 [java::new pt.actor.IORelation $e3 R7]
  $p5 link $r5
  $p5 link $r6
  $p6 link $r7
  $p8 link $r5
  $p9 link $r7
  $p9 link $r6
```

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

- 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 an asterix 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 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 widthFixed() (see figure 3.1). 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.

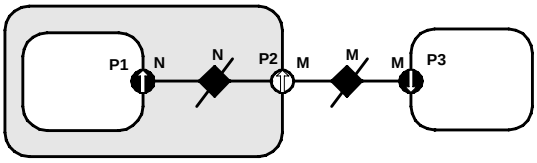


FIGURE 3.9. Bus widths inside and outside a transparent port need not agree..

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 3.1.

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, any practical use of the base class IOPort should schedule these calls so that these exceptions do not occur, or it should catch these exceptions.

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 [11], which is a generalization of dataflow [13], 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 [19].

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. 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.

FIXME: Show code for actors to send and receive data in various ways (broadcast, etc.).

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 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 supported by the RendezvousReceiver class. 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.

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 originating thread to indicate willingness to rendezvous will be the one that will transfer data. Guarded communication [3] is equally easy to implement.

FIXME: Show code for actors to send and receive data in various ways.

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. This is EventReceiver, which uses the SortedQueue class from the actor.util package.

FIXME: Design needed here. Describe CalendarQueue class.

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.

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 3.10, defines how an object can be invoked. There are five methods. The initialize() method is assumed to be invoked exactly once during the lifetime of an execution of an application. 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(). An *iteration* is defined to be one invocation of prefire(), any number of invocation of fire(), and one invocation of postfire(). The wrapup() method will be invoked exactly once per execution, when the execution terminates. Thus, 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*.

3.3.1 Director

The Executable interface, shown in figure 3.10, 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.

A *director* governs the execution of a composite entity. An example of the use of these classes is shown in figure 3.11. In that example, a top-level entity, E0, has an instance of Director, D0, that serves the role of its executive director. An *executive director* is responsible for the overall control of an application. It will start and stop an execution, and may have a control panel GUI associated with it. The top-level entity E0 also has a 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.

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 3.11, 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*.

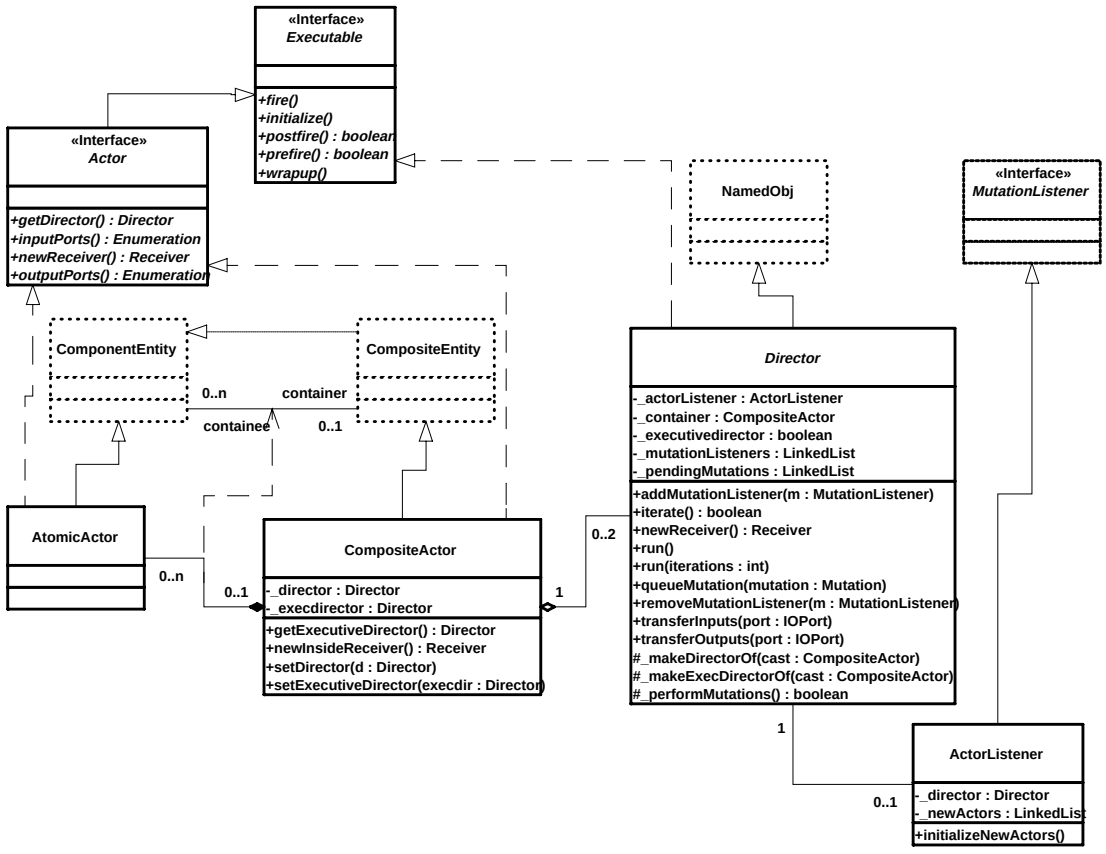


FIGURE 3.10. Classes in the actors package that support execution.

In Ptolemy II, we simply call it a composite opaque actor. It will be explained in more detail below in section 3.3.4.

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, and that director is what is returned by the getDirector() method of the Actor interface (see figure 3.10).

The Director class provides a default implementation of an execution sequence. Although specific domains may override this implementation, in order to ensure interoperability of domains, they should stick fairly closely to the sequence. A complete execution can be obtained by invoking the run() method, which optionally takes an argument specifying the number of iterations. An execution can alternatively be terminated by returning *false* to the postfire() method.

The implementation of the run() method is shown in figure 3.12. It invokes initialize(), followed by some number of invocations of iterate(), followed by wrapup(). The iterate() method is invoked until either it returns *false* or the specified number of iterations have been completed, if a number was specified. The behavior of the initialize() and wrapup() methods depends on whether the director is an executive director or a local director. If it is an executive director, then they simply invoke the corresponding methods of the composite actor under the control of the executive director. This composite actor in turn invokes the corresponding methods of its local director. If it is a local director, the they invoke the corresponding methods of the contained actors, in the order in which these actors were created.

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 and producing initial outputs. Note that while delays in dataflow were implemented as annotations on the arcs in Ptolemy 0.x, they become actors in Ptolemy II. Their initialize methods produce the initial tokens that they represent, and their fire() methods are never invoked (the scheduler simply does not schedule them).

The wrapup() method is also invoked exactly once by the run() method, unless an exception occurs. Typical actions include displaying final results of a run.

The iterate() method is a bit more complicated than initialize() or wrapup(), and hence it is expanded in figure 3.12. It controls the sequence of invocations of prefire(), fire(), and postfire(). The prefire() method returns a boolean, and if it is *false*, then the application is not ready for invocation of the fire() or postfire() methods, so they are skipped. If it returns *true*, then the fire() and postfire() methods are invoked. The behavior of the fire() and postfire() methods again depends on whether the

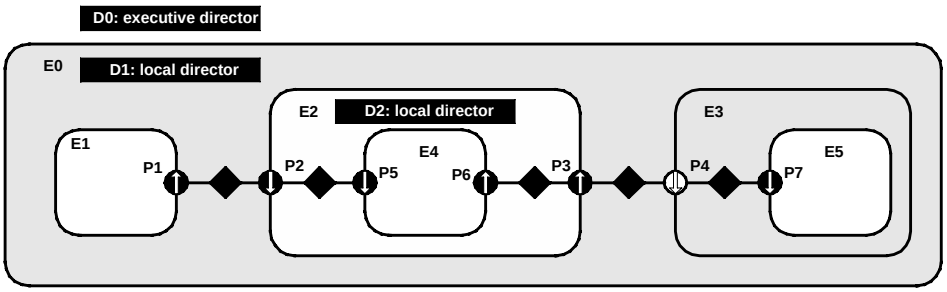


FIGURE 3.11. Example application, showing a typical arrangement of actors and directors.

director is a local or executive director. If it is an executive director, they invoke the corresponding methods of the actor. If it is a local director, then they invoke the corresponding methods of the constituent actors.

The prefire() method again is a bit more complicated, so it too is expanded in figure 3.12. This method supports *mutations*, where the topology of the graph changes during the execution of an application. Mutations are explained in the next section. If there are no mutations to be performed, then the prefire() method simply invokes the prefire() method of the actor under control (if it is an executive director) or of the component actors (if it is a local director).

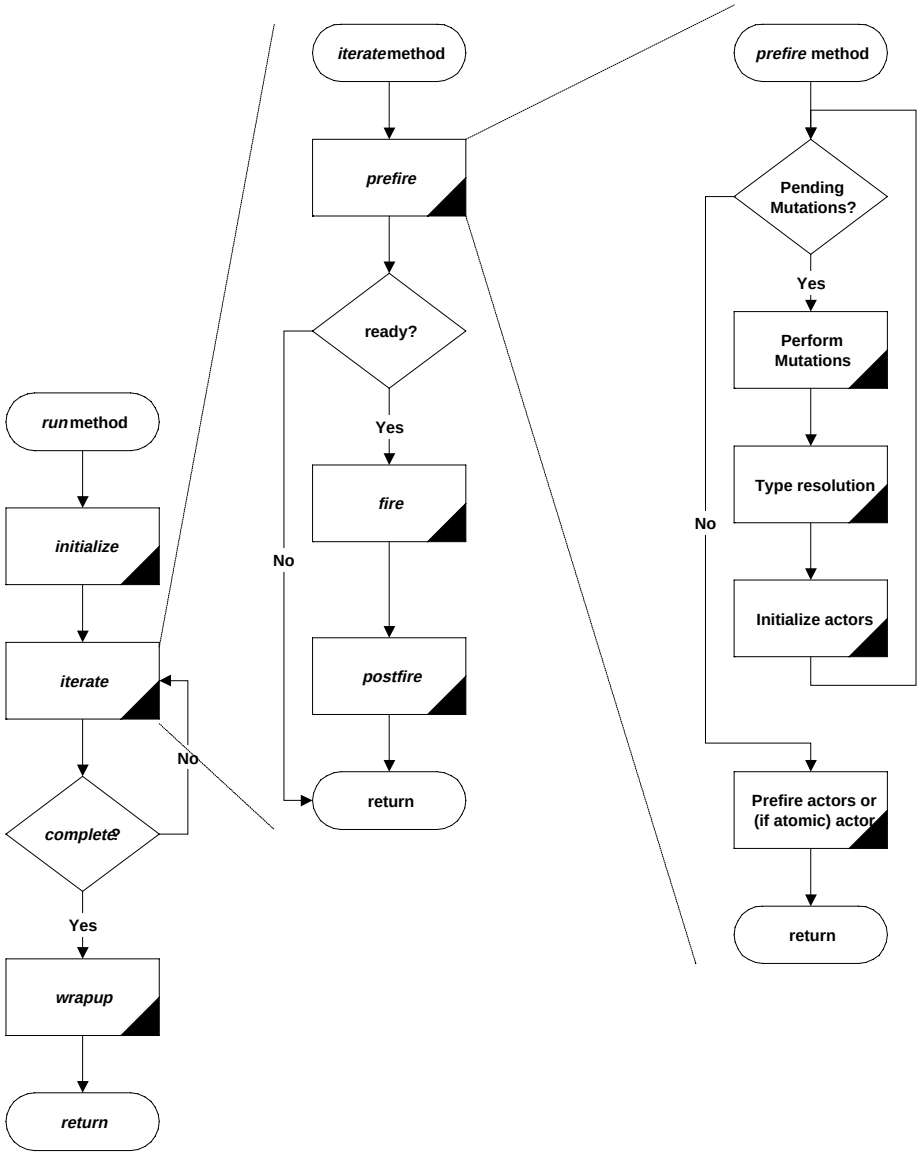


FIGURE 3.12. Execution sequence implemented by run() method of the Director class.

3.3.2 Mutations

A *mutation* is a run-time modification of an application. In most domains, it is not safe for mutations to occur at arbitrary times during an execution. For example, a scheduler may need to be re-run to take into account the mutation. Or a domain may wish to have tight control over when parameters of an application change.

The Director class leverages the mutation subpackage of the kernel, which provides two interfaces, Mutation and MutationListener. A class that implements Mutation has a method that actually performs the mutation. A class that implements MutationListener has methods that are called to inform it of mutations that have been performed.

The general strategy in Director is simple. Any code that wishes to perform mutation queues that mutation with the director rather than performing it directly (using the queueMutation() method, shown in figure 3.10). When it is safe, that mutation is performed, and all mutation listeners that have been registered with the director (using the addMutationListener() method) are informed of the mutation. In the Director class, the mutations are performed in the prefire() method.

When mutations are used, the Director class automatically registers a particular mutation listener, an instance of class ActorListener (see figure 3.10). This listener listens for the addition of new actors. When a new actor is added, this listener makes a record of it. When all pending mutations have been performed, the director asks the listener to initialize each new actor by invoking its initializeNewActors() method.

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 Director iterates, as shown in figure 3.12, until there are no further pending mutations.

3.3.3 Excecution Sequence for Process Networks MOC

FIXME: This needs to be updated when PN has been updated.

PNDirector is derived from Director in the actors package. It uses the default run() method of the Director class. The run method calls the initialize() method, which currently does not do anything in the PN domain. The iterate() method is called repeatedly until it returns true. When iterate returns true, the run() method calls wrapup() and returns.

PNDirector uses the default implementation of the iterate() method in the director. The iterate() method calls prefire(), and if prefire() returns true, then it calls fire() and postfire().

In case of PN, the prefire() calls the getNewActors(), which returns an enumeration of actors that have been added (by mutation of the graph). Note that all the actors are added to this list by calling registerNewActor() in their constructors. The prefire() method starts a thread for each of these actors. Then it clears the list by calling the clearNewActors().

After prefire returns true, the fire() method takes over. The fire() method in PN handles deadlock. The postfire() method does not do anything in case of PN domain.

The iterate() method returns false if mutations have occurred, resulting in the iterate method being called again. The iterate() method returns true to indicate a real deadlock, causing the wrapup() method to be called and the simulation to be terminated. The wrapup method terminates all threads.

FIXME: Need to show here how this execution sequence maps into a few of the MoCs.

3.3.4 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 3.13. 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 sequence of figure 3.12, E2 is treated as an atomic actor by D1; i.e. D1 acts as an executive director to E2. Thus, the prefire() method of D1 invokes the prefire() methods of E1, E2, and E3. The prefire() 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 prefire() method of D2, which in turn invokes the prefire() method 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 postfire() method of E2 is 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 model of computation.

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

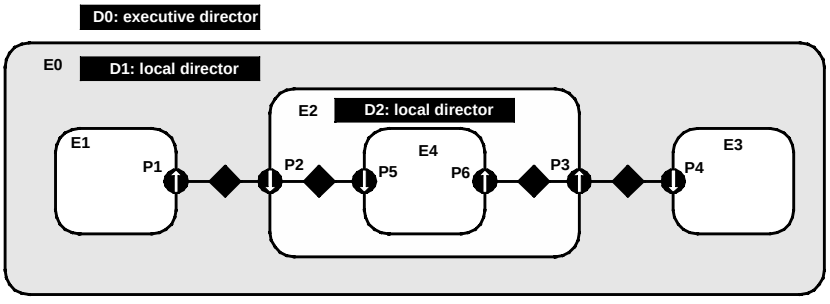


FIGURE 3.13. An example of a composite opaque actor. E0 and E2 both have local directors, not necessarily implementing the same model of computation.

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.

3.4 Utilities

FIXME: discussion of the util package.

3.5 Library

FIXME: discussion of the lib package.

4

Data

Figure 4.1 shows the classes that carry data between actors. The class hierarchy here defines a partial order that is used in type resolution. The bottom element of the partial order is the class Token (thus, the inheritance diagram is upside down compared to a representation of the partial order). The partial order is converted to a lattice by adding a top element called NotAType. This element is used to flag type conflicts found during type resolution.

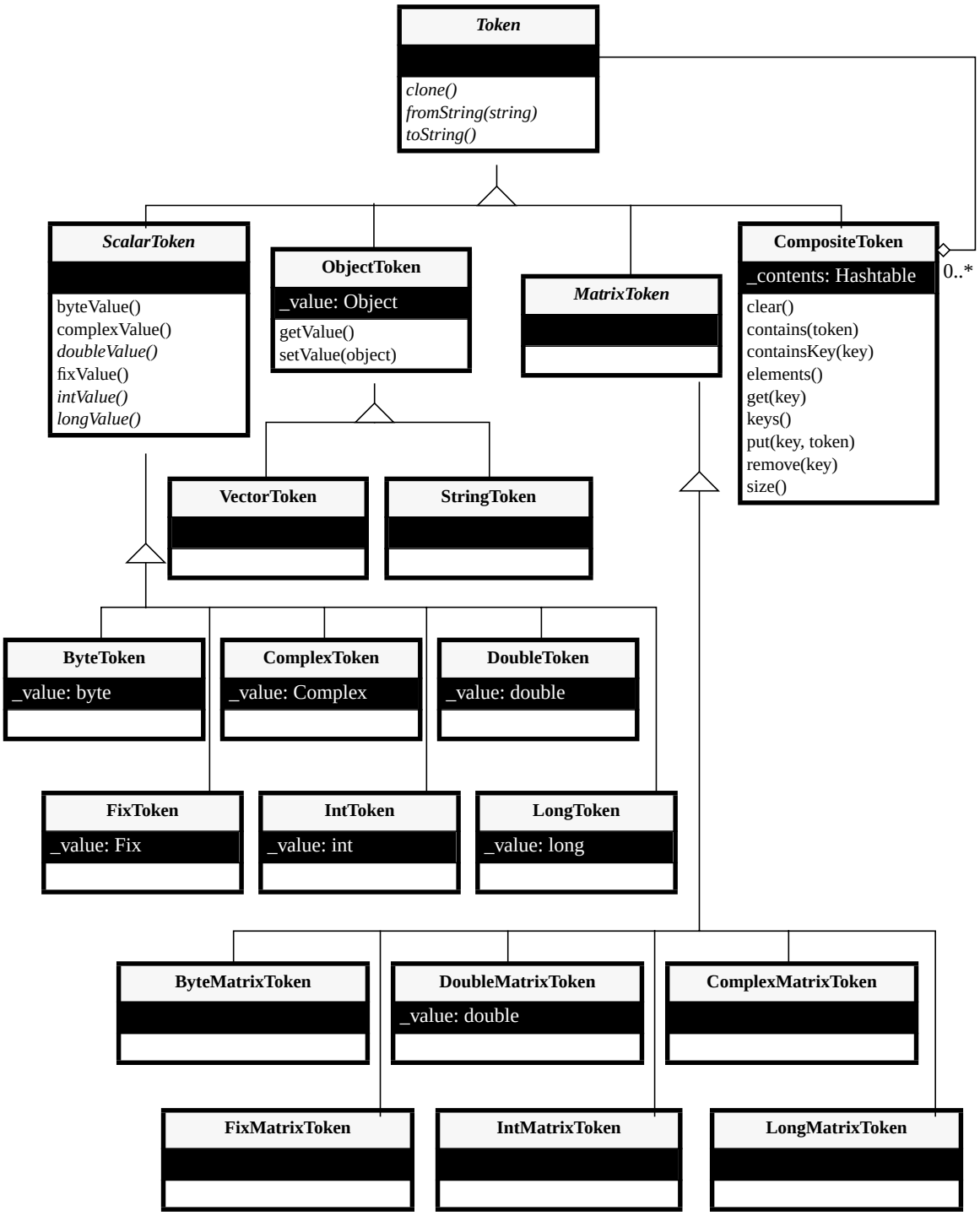


FIGURE 4.1. Token classes are used to convey data between actors.

5

Graph

Many Ptolemy II applications apply graph algorithms to representations of a topology. This is done for instance to support scheduling or synthesis. Two graph packages are included in Ptolemy II. The *staticgraph* package is the simpler of the two, and supports only graphs that are not changed after they are constructed. The *dynamicgraph* package uses the Ptolemy II abstract syntax, but specializes it to mathematical graphs, and then provides graph algorithms that operate on it.

figure 5.1.

Typical usage: Use CompositeEntity::deepGetEntities() to get an enumeration of objects that implement the GraphNode interface. Then pass that enumeration a graph algorithm.

Requirements:

- Need to be able to decorate nodes and edges.
- Need to be able to return a subset of the given graph (some edges, some nodes).

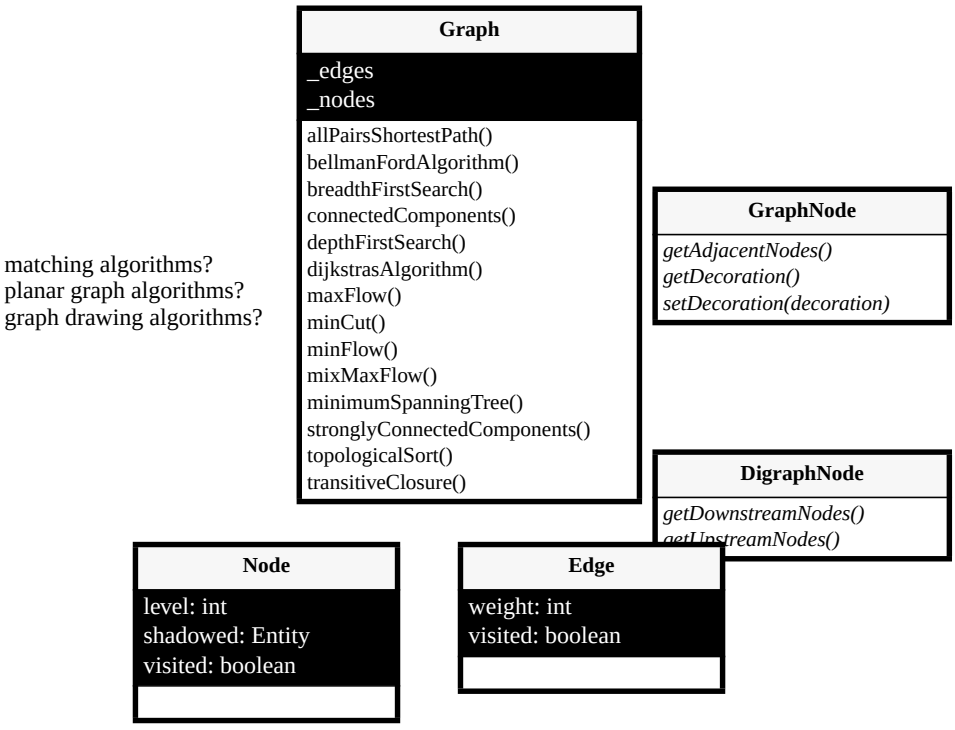


FIGURE 5.1. Interfaces for graph algorithms.

6

Higher-Order Functions

Higher order functions like those in the HOF domain of Ptolemy 0.x [13] are supported by the *hof* package. These can be used to construct scalable visual representations and data parallel specifications [9].

7

Automata

Actors have entities that perform computations. A rather different kind of topology in Ptolemy II uses entities to represent states of computation rather than the computations themselves. The transitions package supports this style of topology.

8

Synthesis

8.1 Separating Interface from Implementation

8.1.1 Syntactic Properties of the Interface

Core/Corona model.

8.1.2 Semantic Properties of the Interface

Agha [2] argues that that no model of concurrency can or should allow all communication abstractions to be directly expressed. Simple message passing is akin to “gotos” in their lack of structure. Instead, actors should be composed using an interaction policy or “interaction pattern.” Agha suggests that there are two useful interaction policies, called “atomicity” and “precedence constraints.” These are similar to our rendezvous and dataflow, but in fact there are several other interaction patterns that prove useful.

FIXME: SR, AN, DE.

FIXME: How the corona/core concept transcends domains for functional actors. Point out that non-functional actors may have more dependence on the interaction patterns, e.g. a queue in DE with a demand input (actually, is this a good example? The same queue could work in DDF).

9

Conclusions

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] 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).

[5] 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/)

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

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

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

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

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

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

[12] D. Lea, *Concurrent Programming in Java™*, Addison-Wesley, Reading, MA, 1997.

[13] 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)

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

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

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

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

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

[19] 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).

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

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

[22] J. Rowson and A. Sangiovanni-Vincentelli, "Interface Based Design," *Proc. of DAC '97*. FIXME- check reference

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

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

Index