DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720

PTOLEMY II

HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA™

— DRAFT 1 — NOT FOR DISTRIBUTION —

January 24, 1998

John Davis, II
Ron Galicia
Mudit Goël
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

1. Ptolemy II

Ptolemy II is a complete, from the ground up, redesign of the Ptolemy design environment, which supports heterogeneous concurrent modeling and design [1]. Some of the major capabilities 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 [4]. 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, and hierarchical concurrent finite-state machines.
- *Better modularization through the use of packages and components.* Ptolemy II is divided into packages that can be used independently and distributed on the net, or drawn on demand from a server. In addition, foreign software components can be imported into Ptolemy designs, and Ptolemy designs can be exported as software components, using the Java™ Beans architecture.
- *Complete separation of the abstract syntax from the semantics*. Ptolemy designs are structured as hierarchical graphs. Ptolemy II defines a clean and thorough abstract syntax for such hierarchical 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 hierarchical 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 hierarchical graph structure) while the user interface simultaneously modifies the structure in different ways. Consistency is maintained through the use of monitors built upon the lower level synchronization primitives of Java™.
- *A well-designed software architecture.* Since the first Ptolemy implementation, software engineering has seen the emergence of sophisticated object modeling and design patterns [2] 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.
- *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. We will design and use 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.
- *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 constructed where compatibility across implementation technologies is assured.

2. The Kernel Package — Classes Supporting the Abstract Syntax

The kernel defines a small set of Java classes that implement a data structure supporting a general form of uninterpreted hierarchical graphs, plus methods for accessing and manipulating such graphs. These graphs provide an abstract syntax for netlists, state transition diagrams, block diagrams, etc. A particular graph configuration is called a *topology*.

Although this idea of an uninterpreted abstract syntax is present in the original Ptolemy kernel [1], 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 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 mathematical graphs normally have no notion of hierarchy.

Relations are intended to serve a mediators, in the sense of the Mediator design pattern of Gamma, et al [2]. "Mediator promotes loose coupling by keeping objects from referring to each other explicitly..." For example, a relation could be used to handle 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.

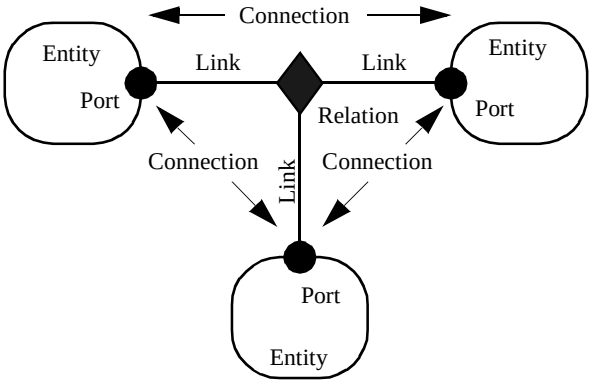

FIGURE 1. Visual notation and terminology.

2.1 BASIC JAVA CLASSES, SUPPORTING NON-HIERARCHICAL GRAPHS

The most basic classes in Ptolemy II and their relationships are shown in figure 2, using UML notation. The class name is shown at the top of each box, its attributes are shown below that, and its methods below that. Subclasses are indicated by lines with white triangles. Associations of various sorts are indicated by other lines, often annotated with ranges like "0..*" or by white diamonds. These annotations will be explained as the need arises.

The classes in figure 2 support non-hierarchical topologies, like that shown in figure 1. For completeness, we have shown all public methods of these classes, although a proper object model would only show those relevant to the issues being discussed.

Although these classes do not provide support for constructing hierarchical graphs, they provide rudimentary support for *container* associations. An instance of these classes can have at most one container. That container is viewed as the owner of the object, and "managed ownership" [4] is a central tool in thread safety, as explained in section 2.4 below.

The Nameable interface supports hierarchy in the naming so that individual named objects in a hierarchy can be uniquely identified. By convention, the *full name* of an object is a concatenation of the full name of its container (if there is one), a period ("."), and the name of the object. The period is only included if the object has a container. The full name is used extensively for error reporting. A NamedObj base class supports naming of entities, ports, and relations.

Workspace is a concrete class that implements the Nameable interface. All objects in a topology

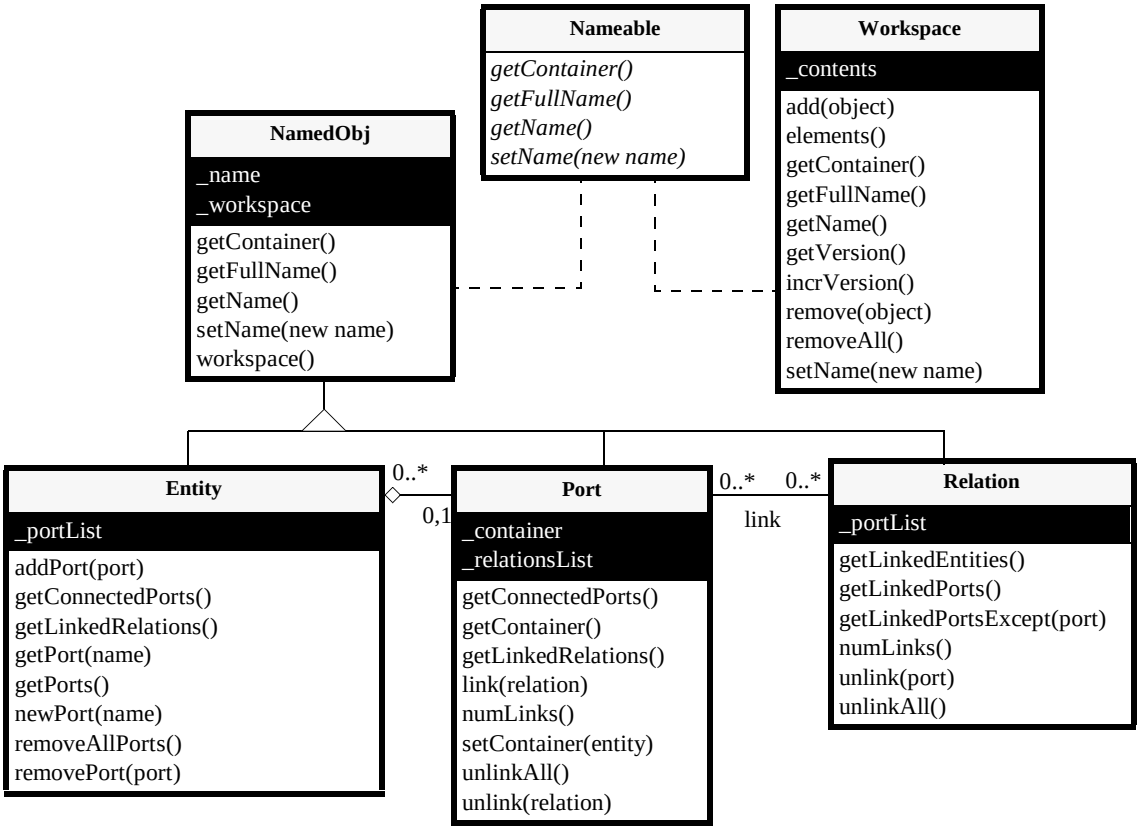

FIGURE 2. Key classes and their methods supporting basic (non-hierarchical) topologies.

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 2.4 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.

NamedObj is also a concrete class implementing the Nameable interface. Here, names are a property of the instances themselves, rather than properties of an association between entities. As argued by Rumbaugh in [7], 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).

An Entity contains any number of Ports; such an aggregation is indicated by the association with a white diamond and the label "0..*" 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. 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. Thus together they define a web of interconnected entities.

Some comments about our naming convention may help understand these classes. Method names that are plural, such as getPorts(), return an enumeration. Attributes with leading underscores, such as _portList, are implemented as private members (or, much more rarely, protected members). Protected and private methods are not shown.

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 interface 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. For example, the _portList attribute is altered only by three methods:

```
addPort(port)
newPort(name)
removePort(port)
```

The addPort() method checks the port argument to see if it already has a container, and if so, removes it from that entity before adding it to this entity. Moreover, it sets the _container attribute of the port to refer to the new container. These three changes, involving three distinct objects, must be atomic, in the sense that other threads must not be allowed to intervene and modify these same attributes halfway through the process. This issue is discussed further below in section 2.4.

The newPort() method creates a port with the entity as container and returns a reference to the port. This method may be overridden in derived classes to create a domain-specific object of a class derived from Port. The method uses addPort() to again ensure symmetric associations. Thus, from the entity side, there is no way to add a port that does not refer to the entity as its container.

From the Port side, the setContainer() method does the following:

1) Check to see if the container is already set, and if so, remove the port from that entity.

2) Add the port to the new entity.

3) Set the port to refer to the new entity as its container.

So from the port side there is also no way to create an inconsistent topology. These methods are designed so that if an exception is thrown at any point during the process, any changes that have been made are undone so that a consistent state is restored.

2.2 SUPPORT CLASSES

Support classes are shown in figure 3. 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 here to maintain the list of ports in 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.

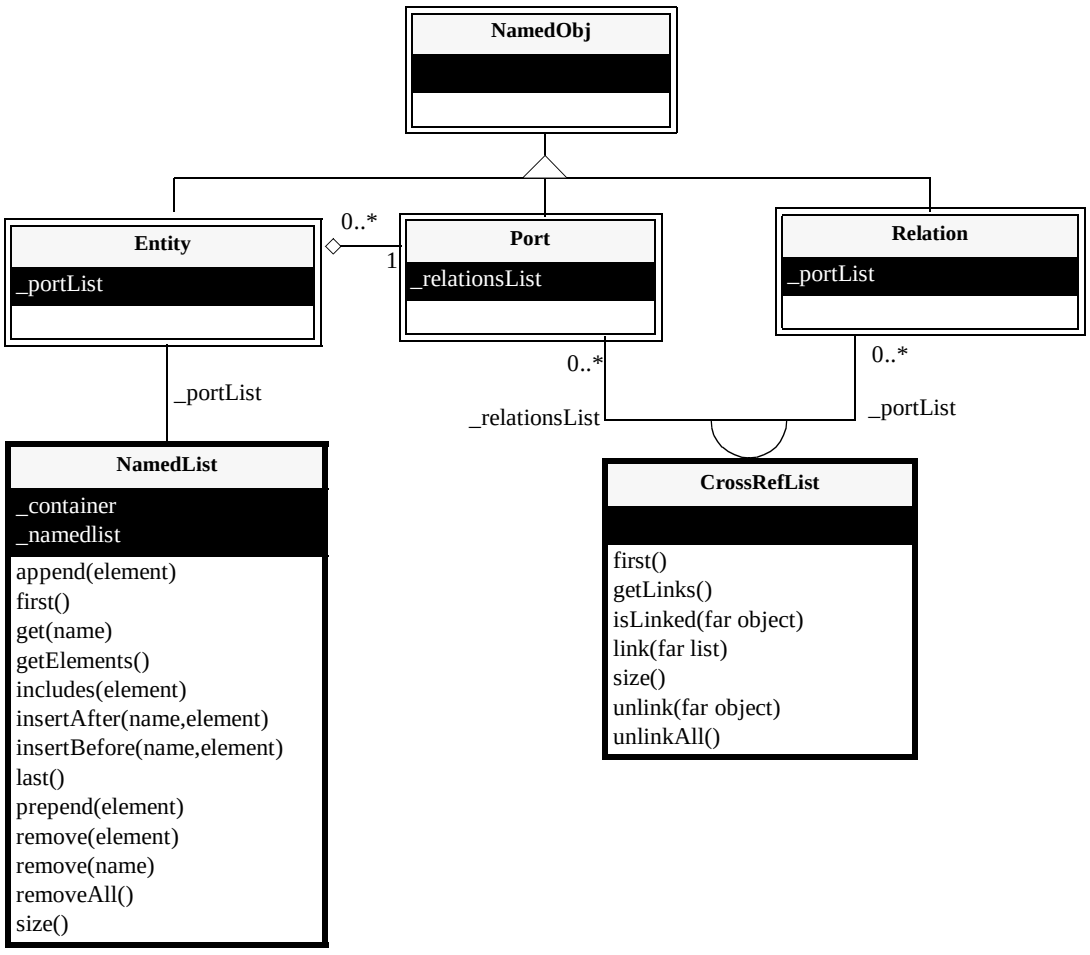

FIGURE 3. Support classes and their methods. For the classes with double borders, only the most relevant methods are shown.

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.

2.3 HIERARCHICAL GRAPHS

The classes shown in figure 2 provide only partial support for hierarchy, through the concept of a container. Subclasses, shown in figure 5, extend these with more complete support for hierarchy. ComponentEntity, ComponentPort, and ComponentRelation are used whenever a hierarchical 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 [2].

Composite entities can contain a graph, and ports in the contained graph can be exposed as ports of the container entity. This is the converse of the “hiding” operator often found in process algebras. Ports within an entity are hidden by default, and must be explicitly exposed to be visible (linkable) from outside the entity.

The mechanism we use to expose ports is illustrated by the example in figure 4. In that figure, every object has been given a unique name. This is not necessary of course, since only the full name of an object needs to be unique. For example, the full name of P5 is .E0.E4.P5, assuming the workspace has no name. However, using unique names makes our explanations more readable.

Some of the ports in figure 4 are filled in white rather than black. These ports are said to be *transparent.* Transparent ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

Any instance of ComponentPort may be transparent or not. It maintains two lists of links, those to relations inside and those to relations outside. It is transparent if it has one or more inside links.

ComponentPort, ComponentRelation, and CompositeEntity have set of methods with the prefix

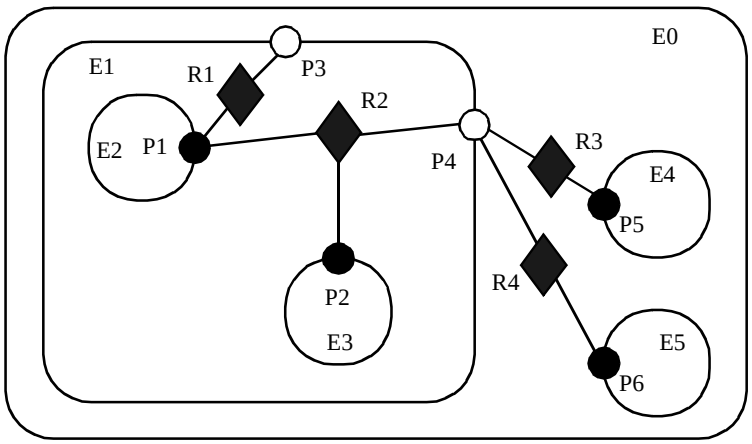

FIGURE 4. Transparent ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

"deep," as shown in figure 5. These methods flatten the hierarchy by traversing it. Thus, for example, the ports that are "deeply" connected to port P1 in figure 4 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.

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, the Workspace class includes a getVersion() and incrVersion() method. All methods of objects within a workspace that modify the topology in any way are expected to increment the version count of the workspace. That way, when a deep access is performed by a ComponentPort, it can locally store the resulting list and the current version of the workspace. The next time the deep access is requested, it checks the version of the workspace. If it is still the same, then it returns the locally cached list. Otherwise, it reconstructs it.

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

2.3.1 Level-Crossing Transitions

For a few applications, such as Statecharts [3], level-crossing connections are needed. The example shown in figure 6 has three level-crossing connections that are slightly different from one another.

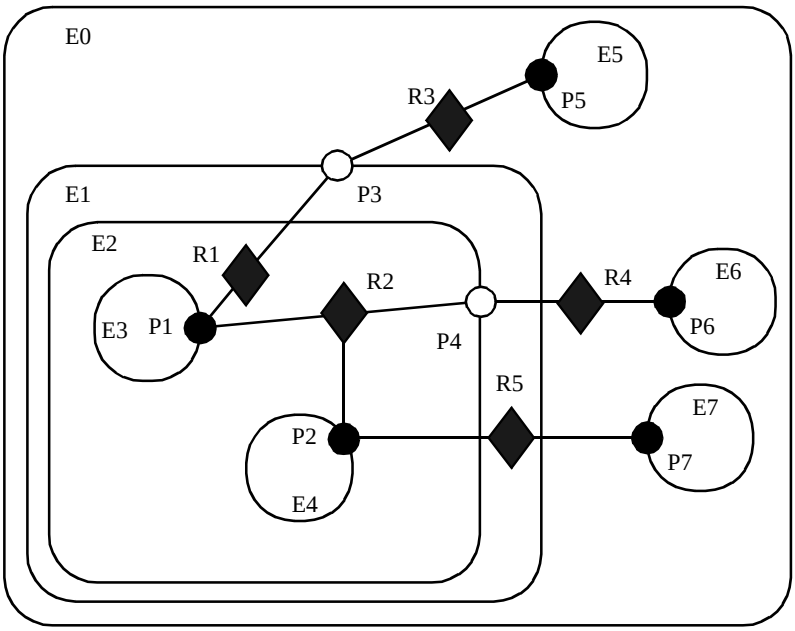

FIGURE 6. An example with level-crossing transitions.

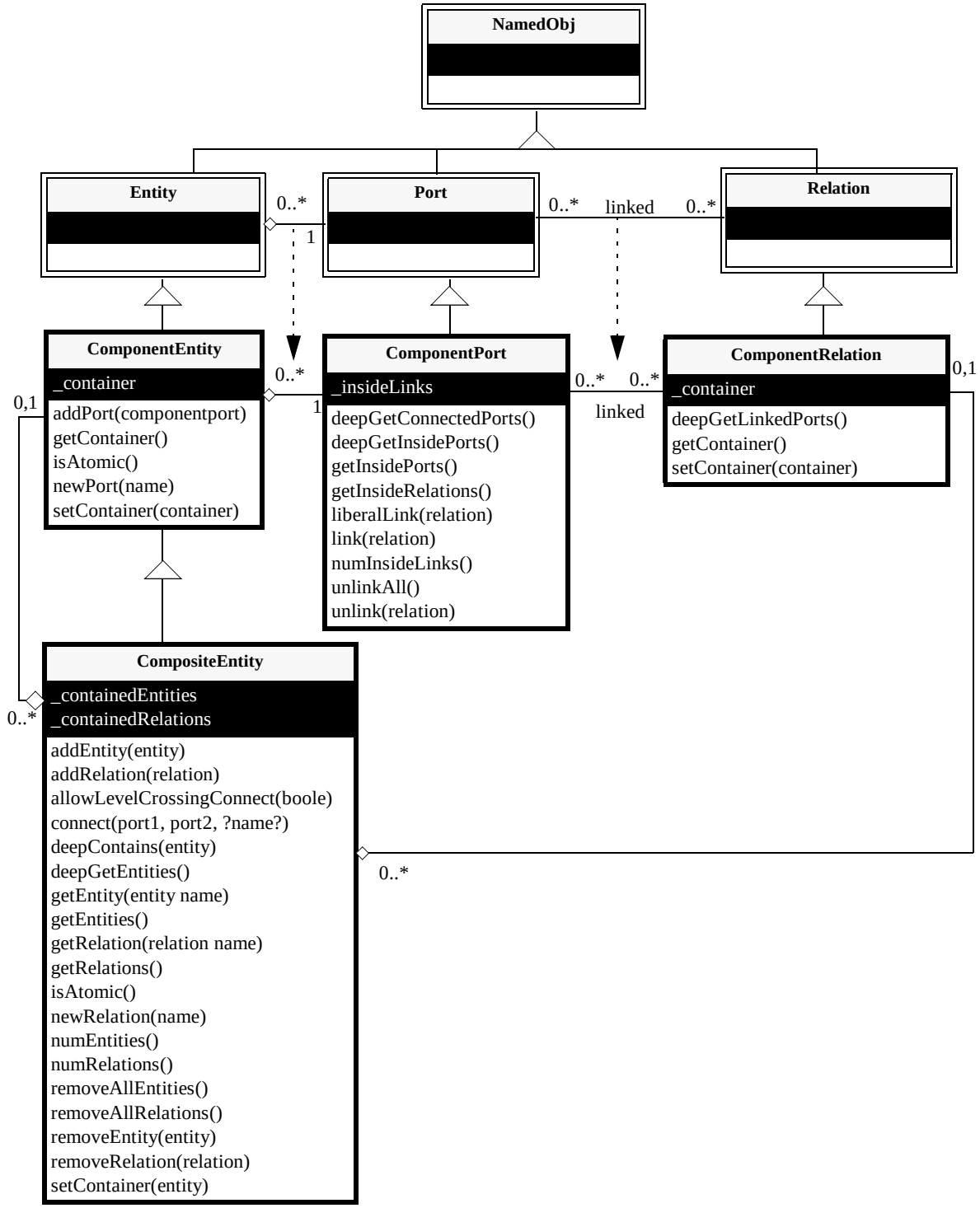

FIGURE 5. Key classes supporting hierarchical graphs.

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).

The simplest level-crossing connection in figure 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 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 CompositeEntity supports this test.

2.3.2 Transparent Entities

The transparent port mechanism we have described supports connections like that between P1 and P5 in figure 7. That connection passes through the entity E2. The relation R2 has an inside link to each of P2 and P4, in addition to its outside link to 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 *transparent entity* is one that contains a relation with inside links to more than one port. It may of course also contain more standard links, but the term transparent suggests that at least some deep graph traversals will see right through it.

Support for transparent entities is a major increment in capability over the previous Ptolemy kernel [1]. 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 [5] was made much more complicated. These higher-order functions had to be careful to avoid mutations that converted an opaque entity into a transparent one.

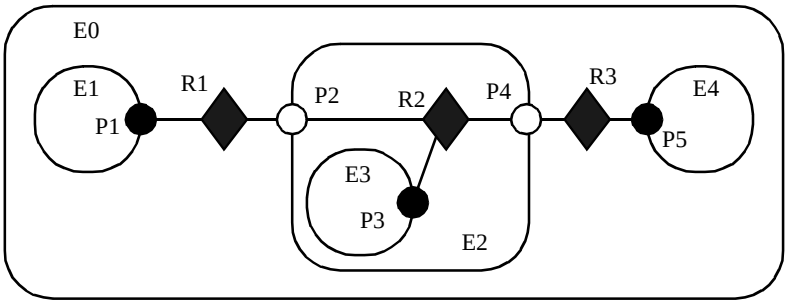

FIGURE 7. A transparent entity contains a relation with inside links to more than one port.

2.3.3 Wormholes

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 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. Consider figure 7, and assume that entity E2 does not wish to expose its internal structure. The ports P2 and P4 must be converted to ports that appear transparent from the inside, but not from the outside.

2.3.4 An Elaborate Example

An elaborate example of a hierarchical graph is shown in figure 8. This example includes instances of all the capabilities we have discussed. The top-level entity is named "E0." It has no container other than the workspace. All other entities in this example have containers. A Java class that implements this example is shown in figure 9. A script in the Tcl language [6] that constructs the same graph is shown in figure 10. 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

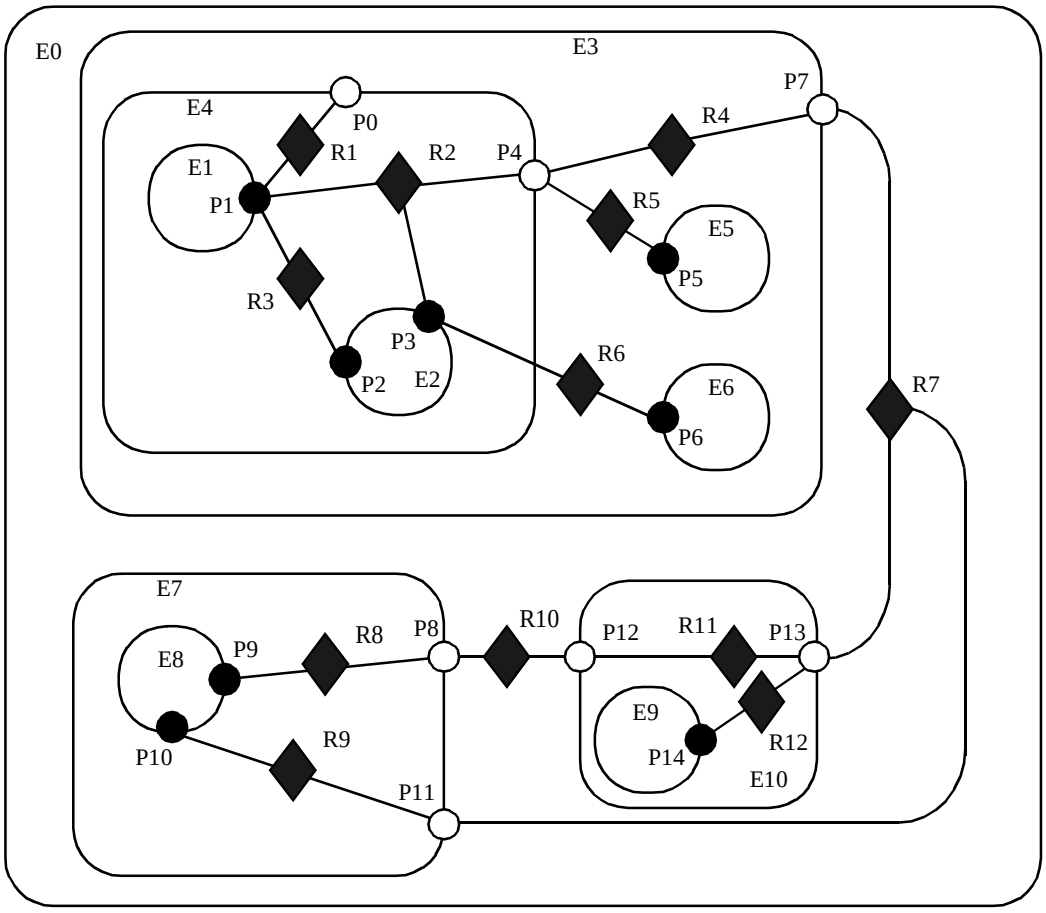

FIGURE 8. An example of a hierarchical graph.

```
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 9. The same topology as in figure 8 implemented as a Java class.

objects preserve this order. The order implemented in both figures 9 and 10 is top-to-bottom and left-to-right in figure 8. 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 11. This table can be studied to better understand the precise meaning of each of the methods.

```
# 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 10. The same topology as in figure 8 described by the TclBlend commands to create it.

2.4 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. Both the computations 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 2.1).

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.

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 12. 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 12(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

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
getLinkedPorts		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 11. Key methods applied to figure 8.

12(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 [4]. 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.

Our solution is also to exploit the hierarchy in the topology. But we implement monitors only on

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

(a)

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

(b)

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

(c)

FIGURE 12. Two syntaxes in Java implementing monitors (a and b) and their use in Ptolemy II (c).

the top-level objects, which are always instances of the class Workspace. Each body of code that alters or depends on the topology must acquire a lock on its workspace, as shown in figure 12(c). 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, and no methods are provided to modify it. Moreover, in instances of these kernel classes, a container and its containees must share the same workspace (derived classes may be more liberal in certain circumstances). This "managed ownership" [4] is our central strategy in thread safety

This solution is conservative in that mutual exclusion is applied even on actions that are independent of one another if they share the same workspace. This effectively serializes some computations that might otherwise occur in parallel. However, there is no constraint 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.

3. The Actors Package

Define *actor* to refer to an entity that processes data that it receives through its ports and/or sends data to other entities through its ports.

Classes that support the exchange of data are shown in figure 13.

IOPort is an abstract base class that defines the interface for handling input and/or output of data, where the data is encapsulated in *tokens*. An IOPort can only be connected to other IOPort instances. The sole purpose of the class IORelation is to ensure that IOPorts are only connected to IOPorts (it does this by overriding the protected method _getPortList). When the put() method of an IOPort is called, it uses deepGetConnectedPorts() to find all of the ports it is connected to. For the first of these whose isInput() method returns true, it passes the token to that port by calling receive(). For subsequent input ports on the list, it passes a clone of the token. This data duplication is necessary to preserve determinacy if actors are permitted to modify a token and pass it on.

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

OneSourceRelation refines IORelation to constrain the links so there is at most one output port linked to the relation. This is needed in some situations to preserve determinacy.

The receive() method is abstract because this base class does not actually provide any mechanism for transferring data. That mechanism depends on the model of computation being used. The get() method, which is again abstract, is intended to be used by the receiving entity to extract data that is received. The role of these methods is depicted informally in figure 14.

This mechanism supports both synchronous (rendezvous) and asynchronous (dataflow) styles of data exchange. For synchronous exchange, implemented by the class RendezvousPort, the receive() method will stall the calling thread until the corresponding get() method is called. If the get() method is called before the receive() method, then its calling thread will stall until the receive() method is called.

For asynchronous (dataflow) exchange, which is implemented by the class FlowPort, the receive() method will simply insert the token in a queue. The get() method will extract tokens from the queue. In the process networks model, which is a generalization of dataflow, the thread calling get() will stall if the queue is empty. If the size of the queue is bounded, then the thread calling receive() may stall if the

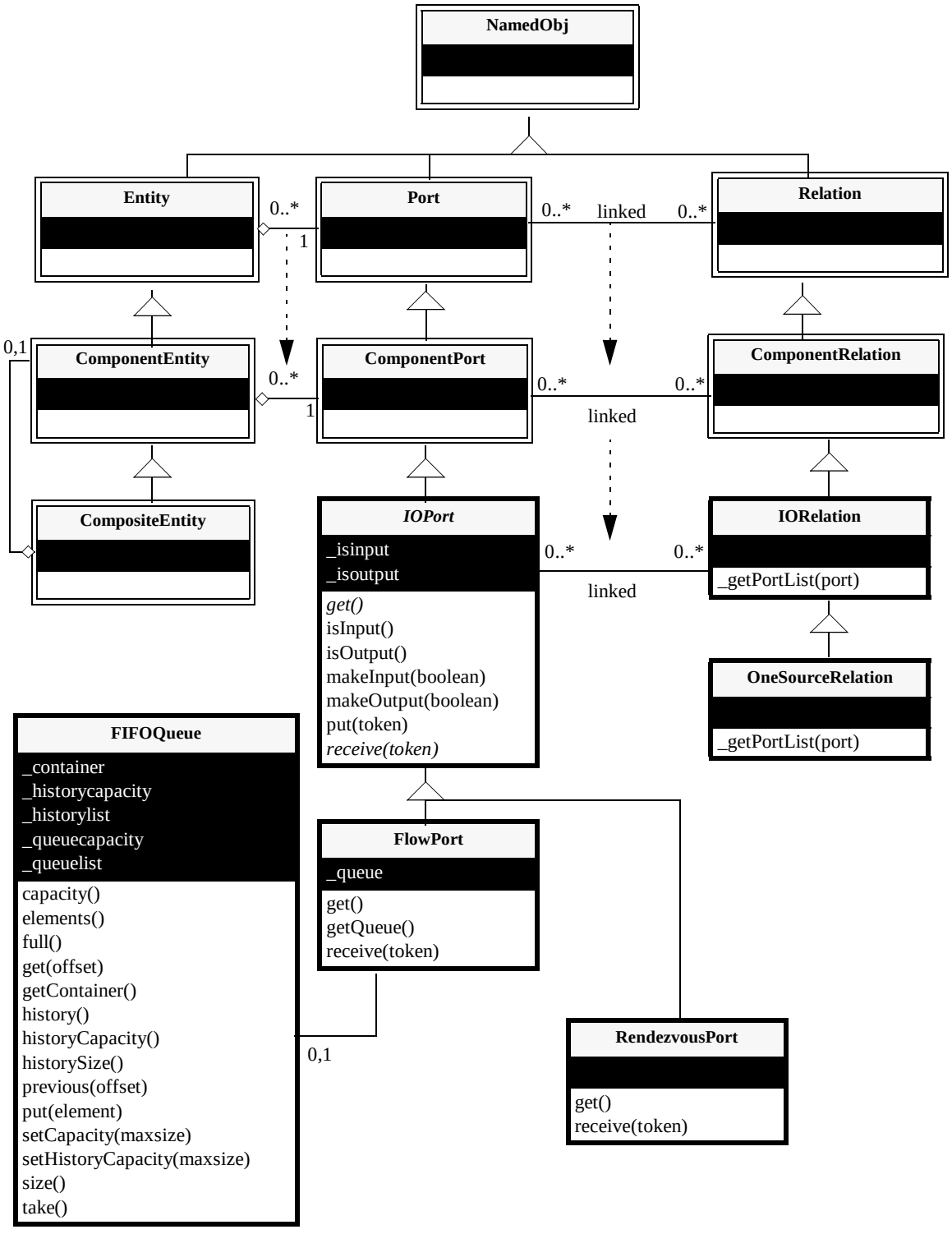

FIGURE 13. Classes in the actors package that support exchange of data.

queue is full. This mechanism supports implementation of a strategy that ensures bounded queues whenever possible. When FlowPort is used with OneSourceRelation, the model of computation is exactly that of Kahn process networks, a special case of which is dataflow. With certain technical restrictions on the functionality of the entities (they must implement monotonic functions under prefix ordering of sequences), this particular combination ensures determinacy in that the history (the sequence of tokens exchanged) does not depend on the order in which the entities carry out their computation. This is particularly important when deterministic multithreaded interaction is desired.

FlowPort uses FIFOQueue, 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.

4. 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.

4.1 BASE CLASS

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

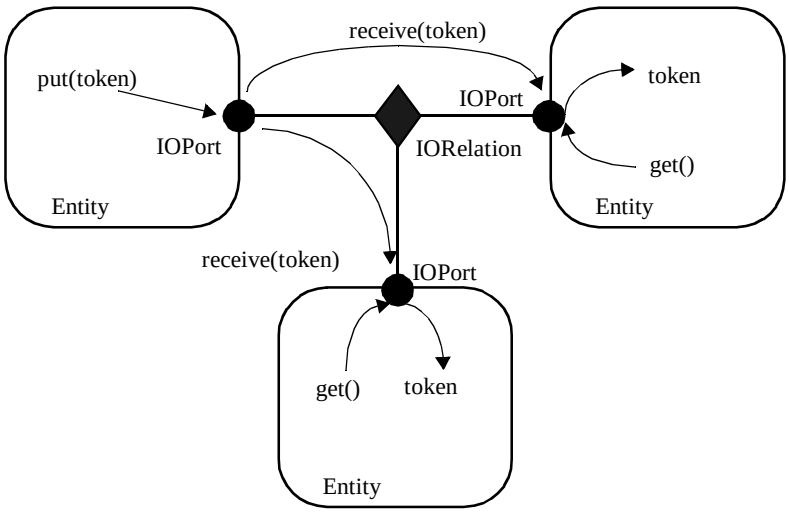

FIGURE 14. Informal schematic showing the mechanism for the transfer of data in IOPort. This mechanism supports both synchronous (rendezvous) and asynchronous (dataflow) transfers.

4.2 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 would result in an inconsistent or contradictory data structure if it were allowed to complete. E.g., attempt to remove a port from an entity when the port does not belong to the entity. Another example would be an attempt to add an item with no name to a named list. 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 (by name) 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)

4.3 VERY SEVERE EXCEPTION

The following exception should never trigger. If it triggers, it indicates a serious inconsistency in the topology. At the very least, the topology being operated on should be abandoned and reconstructed from scratch. It is used when the Java compiler requires us to catch an exception or declare that it is thrown, but we know that it should never occur.

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 it 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.

5. References

[1] 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).

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

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

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

[5] 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)

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

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