Chapter 1 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture) ," Technical Memorandum UCB/ERL M04/16, University of California, Berkeley, CA USA 94720, June 24, 2004.

1

The Kernel

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

1.1 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. They also provide the basic infrastructure for an an actor-oriented version of classes, subclasses, inner classes, and inheritance. 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.*

A topology is a collection of *entities* and *relations*. We use the graphical notation shown in figure 1.1, 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.

We begin by explaining the classes that support topologies with no hierarchy, and then show how these classes are extended to support hierarchy.

1.2 Non-Hierarchical Topologies

The classes shown in figure 1.2 support non-hierarchical topologies, like that shown in figure 1.1. Figure 1.2 is a UML static structure diagram (see appendix A of chapter 1).

1.2.1 Links

An entity contains any number of ports; such an aggregation is indicated by the association with an unfilled diamond and the label "0..n" to show that the entity can contain any number of ports, and the label "0..1" to show that the port is contained by at most one entity. This association uses the NamedList class shown at the bottom of figure 1.2 and defined fully in figure 1.4. There is exactly one instance of NamedList associated with Entity used to aggregate the ports.

A port is associated with any number of relations (the association is called a *link*), and a relation is associated with any number of ports. Link associations use CrossRefList, shown in figure 1.4. There is one instance of CrossRefList associated with each port and each relation. The links define a web of interconnected entities.

On the port side, links have an order. They are indexed from 0 to *n*, where *n* is the number returned by the numLinks() method of Port.

1.2.2 Consistency

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

- Every link between a port an a relation 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 setCon-

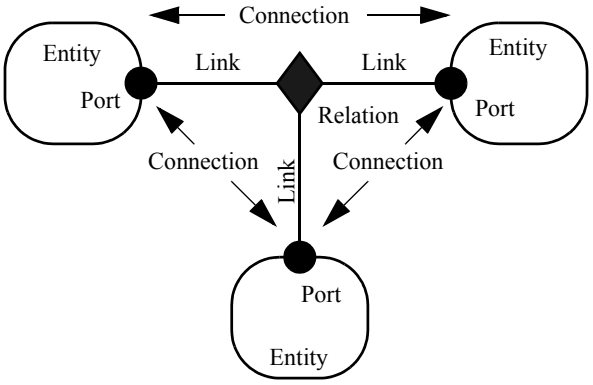

FIGURE 1.1. Visual notation and terminology.

tainer() with a null argument.

A change in a containment association involves several distinct objects, and therefore must be atomic, in the sense that other threads must not be allowed to intervene and modify or access relevant attributes halfway through the process. This is ensured by synchronization on the workspace, as explained below in section 1.6. Moreover, if an exception is thrown at any point during the process of changing a containment association, any changes that have been made are undone so that a consistent state is restored.

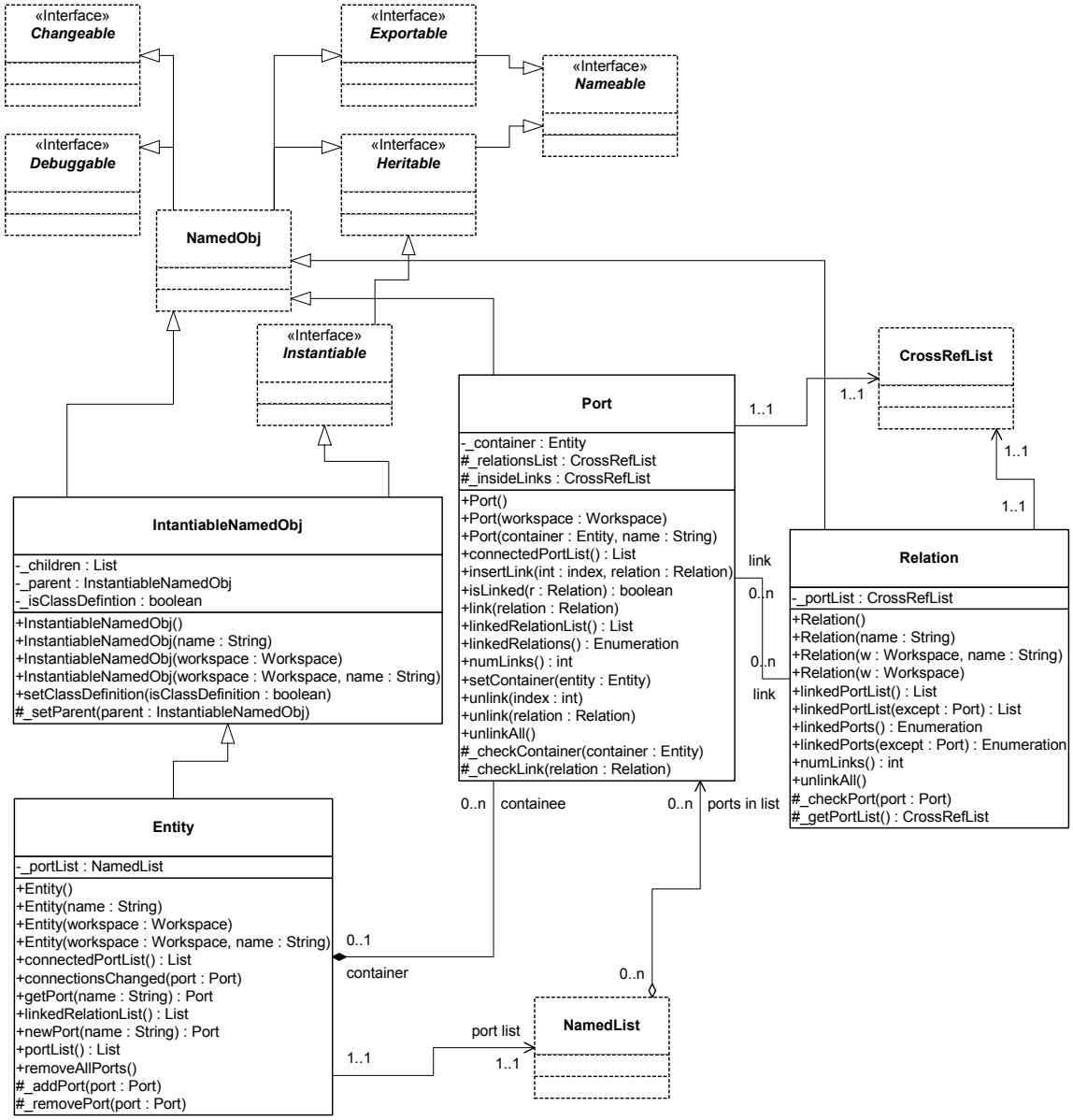

FIGURE 1.2. Key classes in the kernel package and their methods supporting basic (non-hierarchical) topologies. Methods that override those defined in a base class or implement those in an interface are not shown. The "+" indicates public visibility, "#" indicates protected, and "-" indicates private. Capitalized methods are constructors. The classes and interfaces shown with dashed outlines are in the kernel.util subpackage.

1.3 Support Classes

The kernel package has a subpackage called kernel.util that provides the key base class for almost all Ptolemy II objects, NamedObj, shown in figure 1.3. This class defines notions basic to Ptolemy II (containment, naming, parameterization, and inheritance) and provides generic support for relevant data structures. Although nominally the Nameable interface is what defines the naming and containment relationships, in practice, much of Ptolemy II relies on implementations of Nameable being instances of NamedObj.

1.3.1 Containers

Although NamedObj does not provide support for constructing clustered graphs, it provides rudimentary support for *container* associations. An instance can have at most one container. That container is viewed as the owner of the object, and "managed ownership" [70] is used as a central tool in thread safety, as explained in section 1.6 below.

In the base classes shown in figure 1.2, 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 shown have no container, and their getContainer() method will return null. Below we will discuss derived classes that have containers.

Every object is associated with exactly one instance of Workspace, as shown in figure 1.4, but the workspace is not viewed as a container. A workspace is specified 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. An object with a container always inherits its workspace from the container.

1.3.2 Name and Full Name

The Nameable interface shown in figure 1.3 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 full name is used extensively for error reporting. A top-level object always has a period as the first character of its full name. The full name is returned by the getFullName() method of the Nameable interface.

NamedObj is a concrete class implementing the Nameable interface. It also serves as an aggregation of attributes, as explained below in section 1.3.4. It supports inheritance (via its implementation of the Derivable interface), persistence (via the MoMLExportable interface), debugging (via the Debuggable interface), and mutations (via the Changeable interface).

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 [128], 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).

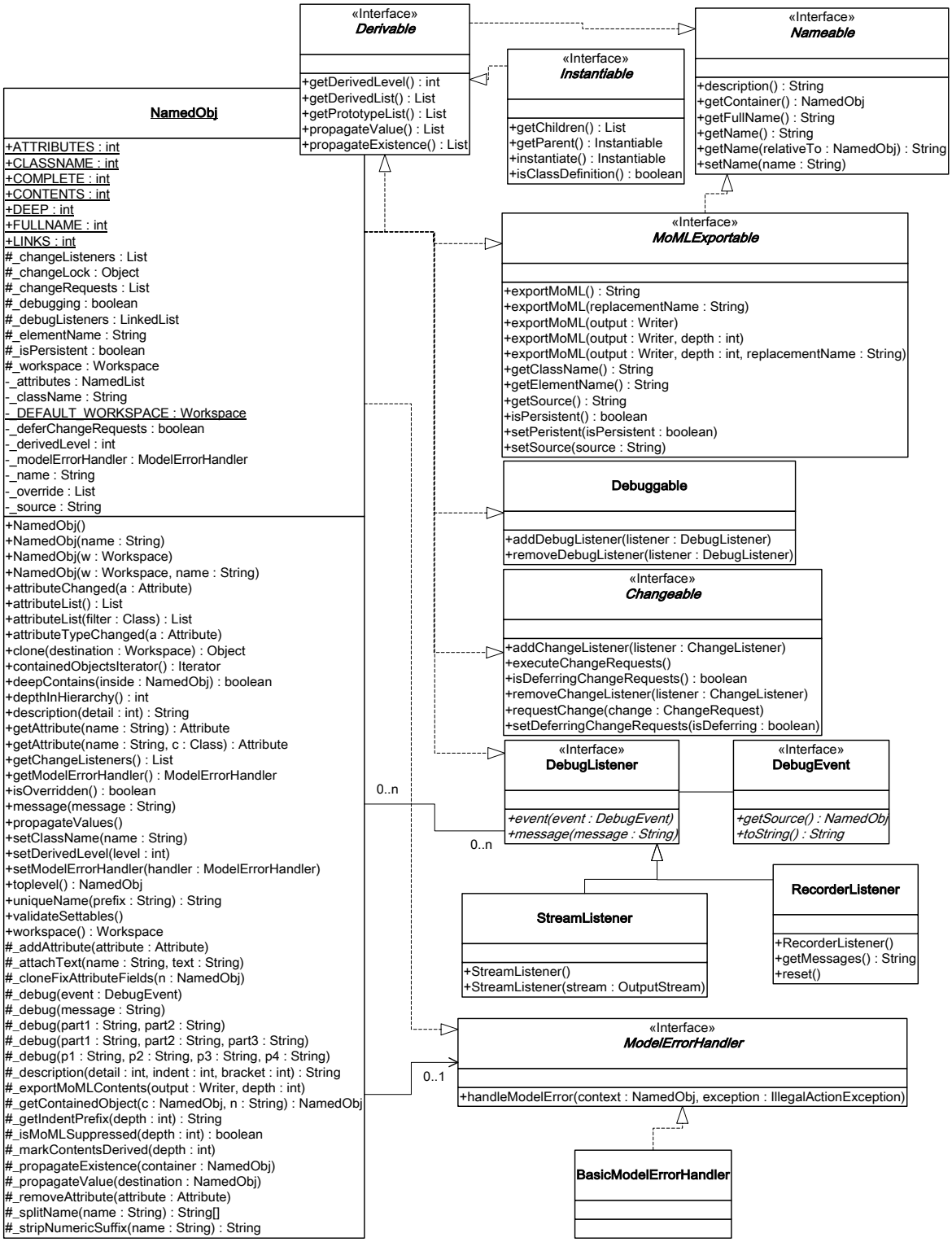

FIGURE 1.3. Support classes in the kernel.util package.

1.3.3 Workspace

Workspace is a concrete class that implements the Nameable interface, as shown in figure 1.4. All objects in a Ptolemy II model 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 1.6 below.

1.3.4 Attributes

In almost all applications of Ptolemy II, entities, ports, and relations need to be parameterized. An instance of NamedObj (figure 1.3) can have any number of instances of the Attribute class attached to it, as shown in figure 1.5. 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. Alternatively, the container can be given as a constructor argument. They are

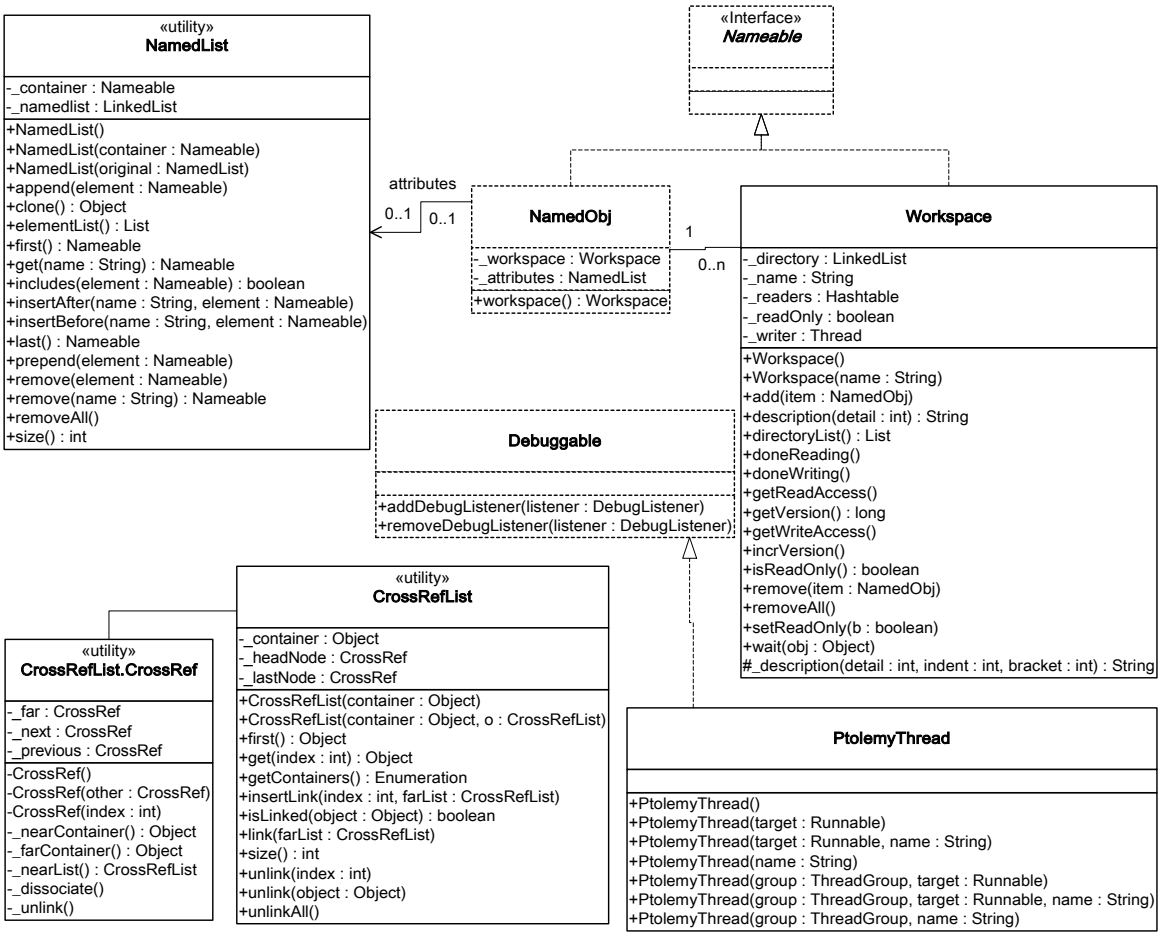

FIGURE 1.4. Some key utility classes. Workspace is the key gatekeeper class supporting multithreaded access to Ptolemy II models. It supports exclusive write access and shared read access. Every instance of NamedObj is associated with exactly one instance of Workspace. NamedList is a utility class used for lists of instances of NamedObj. CrossRefList manages cross references that must be kept consistent.

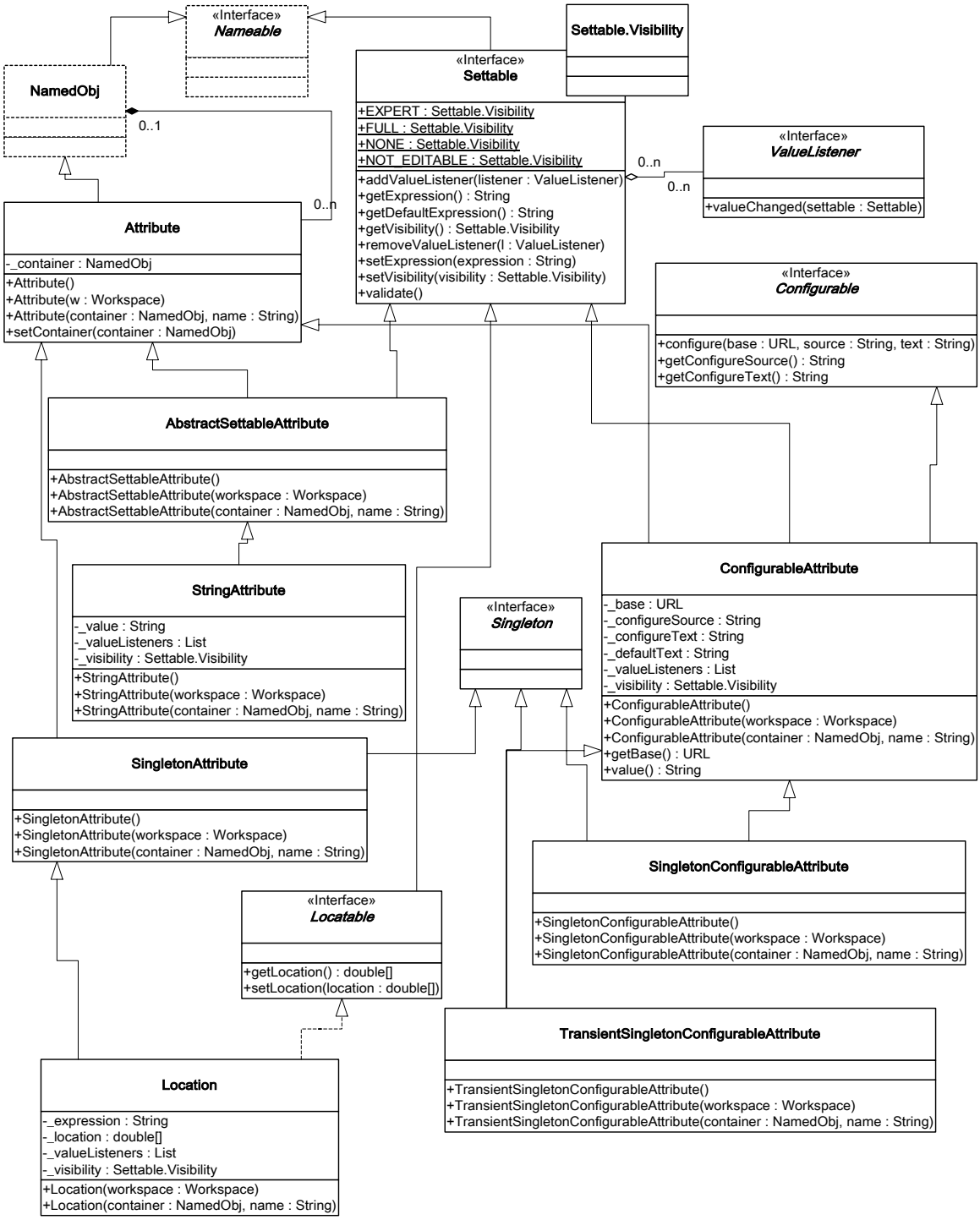

FIGURE 1.5. An instance of NamedObj can contain any number of instances of Attribute. The Ptolemy II kernel provides a few basic attributes, as shown here. Attributes that have values implement the Settable interface. Attributes whose values are numeric data, expressions, or data structures are described in the Data Package chapter.

removed by calling setContainer() with a null argument. The NamedObj class provides the getAttribute() method, which takes an attribute name as an argument and returns the attribute, and the attributeList() method, which returns a list of the attributes contained by the object. Both of these methods have versions that also takes a Class argument, and returns only attributes that are instances of the specified Java class.

By itself, an instance of the Attribute class carries only a name, which may not be sufficient to parameterize objects. Several derived classes implement the Settable interface, which indicates that they can be assigned a value via a string. A simple attribute implementing the Settable interface is the StringAttribute. It has a value that can be any string. A more sophisticated parameter called StringParameter is defined in the data package and has a value that is a string that can include references to other parameter values. A derived class called Variable that implements the Settable interface is defined in the data package. The value of an instance of Variable is typically an arithmetic expression. The Variable class is described in the Data chapter.

Some attributes are *configurable*, which means that their value is set via (typically XML) text that is nested in a MoML configure tag. See the MoML chapter for details. An attribute that is not an instance of Settable or Configurable is called a pure attribute. Its mere presence has significance.

Attribute names can be any string that does not include periods, but it is recommend to stick to alphanumeric characters, the space character, and the underscore. Names beginning with an underscore are reserved for system use. The following names, for example, are in use:

Table 1.1:Names of special attributes

name	class	use
_createdBy	ptolemy.kernel.util.VersionAttribute	Version of Ptolemy II that last wrote the file.
_doc	ptolemy.actor.gui.Documentation	Default documentation attribute name.
_generator	ptolemy.codegen.gui.GeneratorTableauAttribute	Parameters for code generators.
_icon	ptolemy.vergil.toolbox.EditorIcon	Icon renderer attribute.
_iconDescription	ptolemy.kernel.util.StringAttribute	XML description of an icon.
_library	ptolemy.moml.LibraryAttribute	Associates an actor library with a model.
_libraryMarker	ptolemy.kernel.util.Attribute	Marks its container as a library vs. a composite entity.
_location	ptolemy.moml.Location	Records the location of a visual rendition of an object.
_nonStrictMarker	ptolemy.kernel.util.Attribute	Marks its container as a non-strict entity.
_parser	ptolemy.moml.ParserAttribute	Records the MoML parser used.
_url	ptolemy.moml.URLAttribute	Identifies the URL for the model definition.
_vergilLocation	ptolemy.actor.gui.LocationAttribute	Location of the vergil window.
_vergilSize	ptolemy.actor.gui.SizeAttribute	Size of the graph pane in the vergil window.

1.3.5 List Classes

Figures 1.2 and 1.3 show two list classes that are used extensively in Ptolemy II, NamedList and CrossRefList. These pre-date the extensive list classes in the java.util package, and could probably be replaced with those today. NamedList implements an ordered list of objects with the Nameable inter-

face. 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, for example, to maintain a list of attributes and to maintain the list of ports contained by an entity.

The class CrossRefList (figure 1.4) 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.

1.4 Clustered Graphs and Hierarchy

The classes shown in figure 1.2 provide only partial support for hierarchy, through the concept of a container. Subclasses, shown in figure 1.6, 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 [42].

1.4.1 Abstraction

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

A ComponentPort contained by a CompositeEntity has inside as well as outside links. It maintains two lists of links, those to relations inside and those to relations outside. Such a port serves to expose ports in the contained entities as ports of the composite. This is the converse of the “hiding” operator often found in process algebras [103]. In Ptolemy, 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 1.7[2]. Some of the ports in figure 1.7 are filled in white rather than black. These ports are said to be *transparent*. Transparent ports

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

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

ComponentPort, ComponentRelation, and CompositeEntity have a set of methods with the prefix "deep," as shown in figure 1.6. These methods flatten the hierarchy by traversing it. Thus, for example, the ports that are "deeply" connected to port P1 in figure 1.7 are P2, P5, and P6. No transparent port is included, so note that P3 and P4 are 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 deepEntityList() 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 6.3, 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.

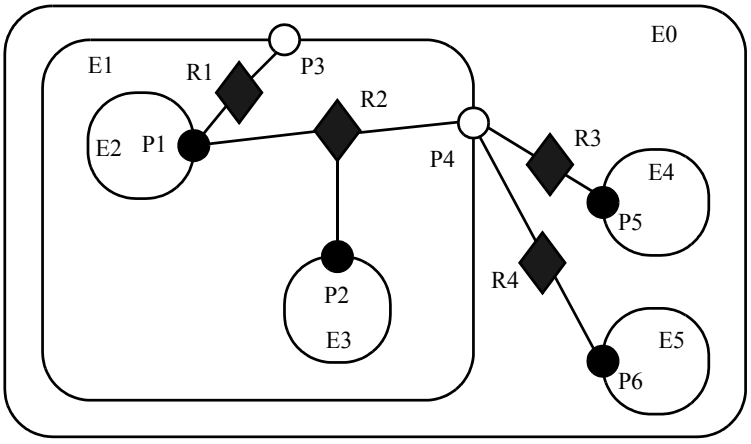

FIGURE 1.7. Transparent ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

2. In that figure, every object has been given a unique name. This is not necessary since names only need to be unique within a container. In this case, we could refer to P5 by its full name .E0.E4.P5 (the leading period indicates that this name is absolute). However, using unique names makes our explanations more readable.

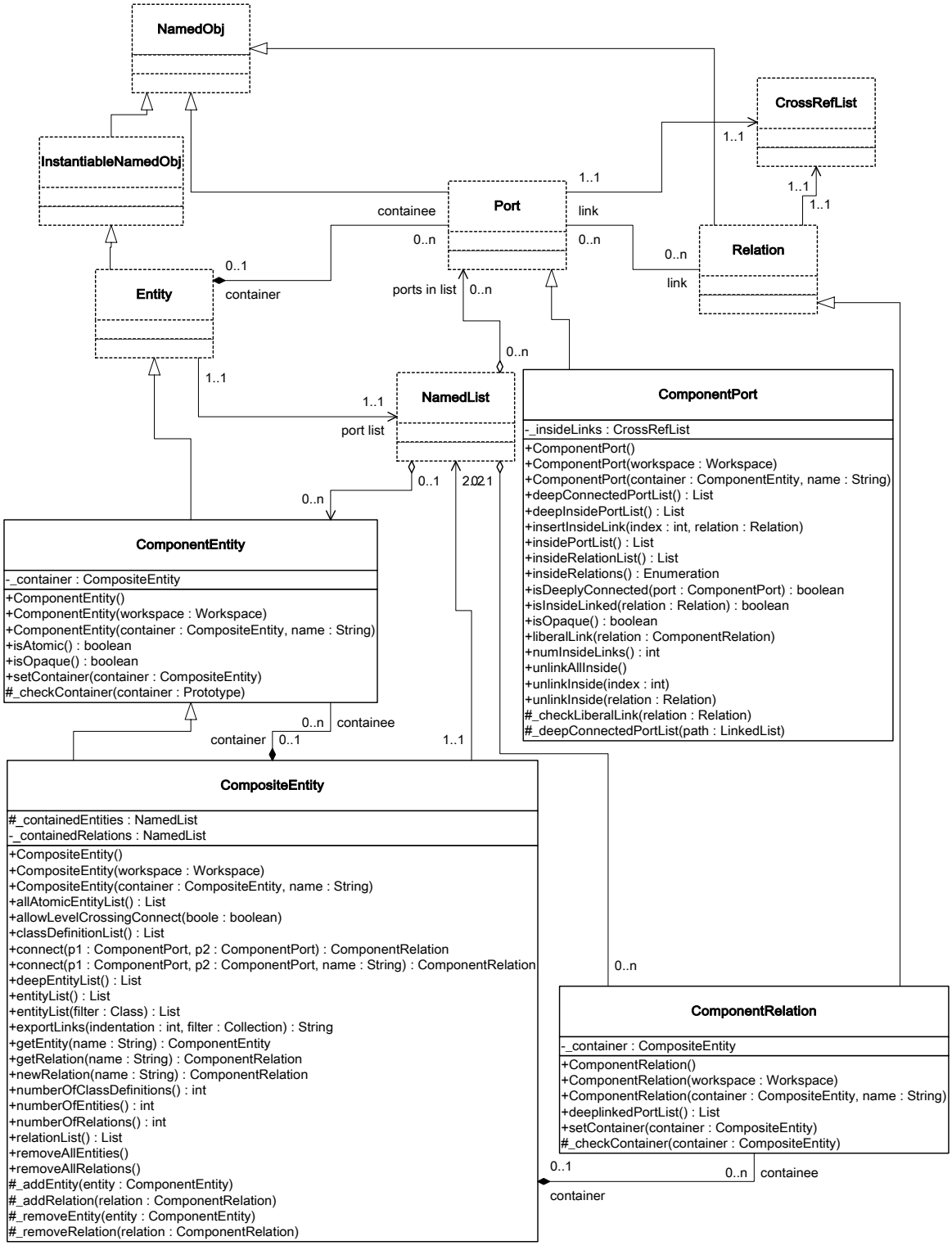

FIGURE 1.6. Key classes supporting clustered graphs.

1.4.2 Level-Crossing Connections

For a few applications, such as Statecharts [50], level-crossing links and connections are needed. The example shown in figure 1.8 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 1.8 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 1.8 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.

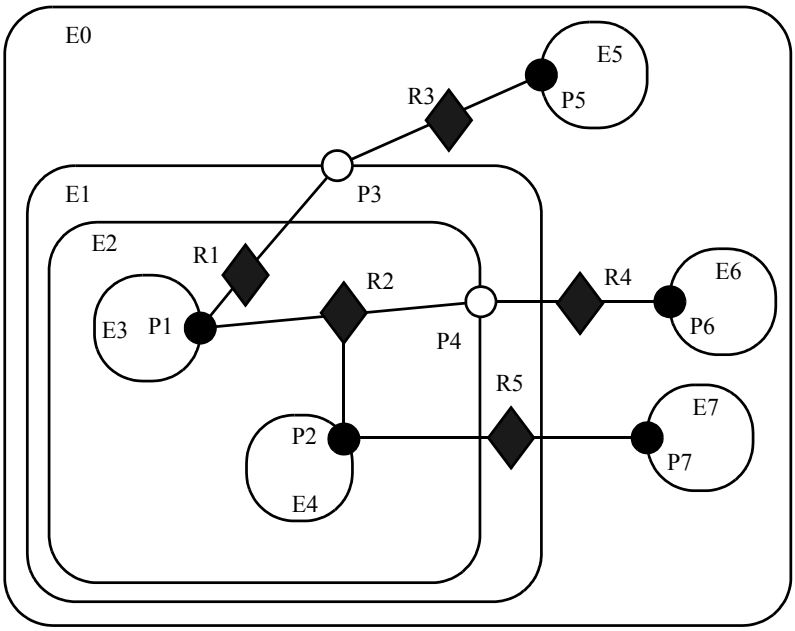

FIGURE 1.8. An example with level-crossing transitions.

1.4.3 Tunneling Entities

The transparent port mechanism we have described supports connections like that between P1 and P5 in figure 1.9. 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 [22] (Ptolemy Classic). That infrastructure required an entity (which was called a *star*) to intervene in any connection through a composite entity (which was called a *galaxy*). Two significant limitations 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 [82] was made much more complicated. These higher-order functions had to be careful to avoid mutations that created tunneling.

1.4.4 Cloning

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

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

1.4.5 An Elaborate Example

An elaborate example of a clustered graph is shown in figure 1.10. This example includes instances of all the capabilities we have discussed. The top-level entity is named "E0." All other enti-

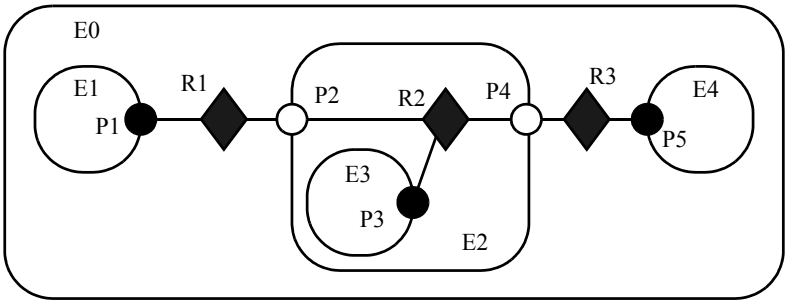

FIGURE 1.9. A tunneling entity contains a relation with inside links to more than one port.

ties in this example have containers. A Java class that implements this example is shown in figure 1.11. A script in the Tcl language [115] that constructs the same graph is shown in figure 1.12. This script uses Tcl Blend, an interface between Tcl and Java that is distributed by Scriptics. Such scripts are used extensively in the Ptolemy II regression test suite.

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 1.11 and 1.12 is top-to-bottom and left-to-right in figure 1.10. 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 1.13. This table can be studied to better understand the precise meaning of each of the methods.

1.5 Opaque Composite Entities

One of the major tenets of the Ptolemy project is that of modeling heterogeneous systems through the use of hierarchical heterogeneity. Information-hiding is a central part of this. In particular, transparent ports and entities compromise information hiding by exposing the internal topology of an entity. In

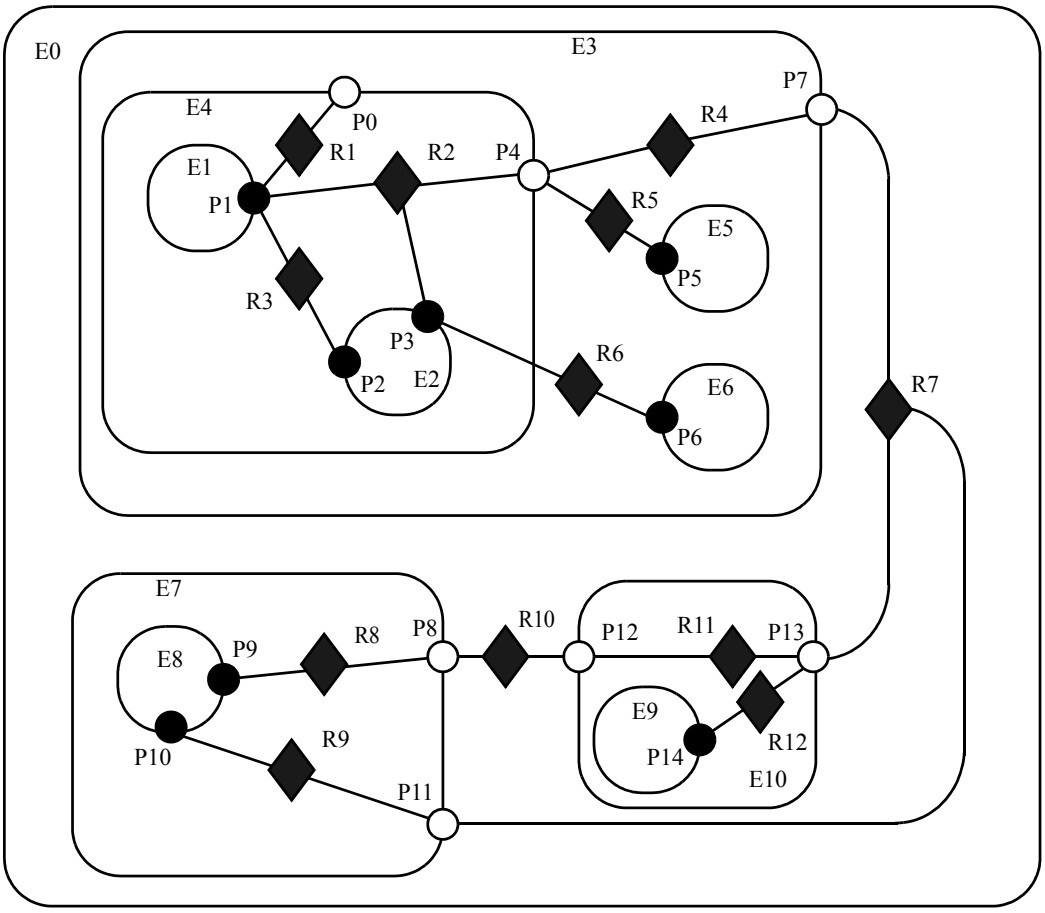

FIGURE 1.10. An example of a clustered graph.

```
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 1.11. The same topology as in figure 1.10 implemented as a Java class.

some circumstances, this is inappropriate, for example when the entity internally operates under a different model of computation from its environment. The entity should be opaque in this case.

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

```
# 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 1.12. The same topology as in figure 1.10 described by the Tcl commands to create it.

1.6 Concurrency

Concurrency is an expected property in many models. Network topologies may represent the structure of computations which themselves may be concurrent, and a user interface may be interacting with the topologies while they execute their computation. Moreover, Ptolemy II objects may interact with other objects concurrently over the network via RMI, datagrams, TCP/IP, 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 1.2.2).

Concurrency could easily corrupt a topology if a modification to a symmetric pair of references is interrupted by another thread that also tries to modify the pair. Inconsistency could result if, for example, one thread sets the reference to the container of an object while another thread adds the same object to a different container's list of contained objects. Ptolemy II prevents such inconsistencies from occurring. Such enforced consistency is called *thread safety.*

1.6.1 Limitations of Monitors

Java threads provide a low-level mechanism called a *monitor* for controlling concurrent access to data structures. A monitor locks an object preventing other threads from accessing the object (a design pattern called *mutual exclusion*). Unfortunately, the mechanism is fairly tricky to use correctly. It is

Table 1.1: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 1.2: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 1.13. Key methods applied to figure 1.10.

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 1.14. 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 1.14(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 1.14(b), the code body represented by brackets {...} 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 [70]. For example, we could use the containment hierarchy and always acquire 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:

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

(a)

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

(b)

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

(c)

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

(d)

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

(e)

FIGURE 1.14. Using monitors for thread safety. The method used in Ptolemy II is in (d) and (e).

```
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.

1.6.2 Read and Write Access Permissions for Workspace

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

Ptolemy II uses monitors on instances of the class Workspace. As shown in figure 1.3, 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 1.3, and no methods are provided to modify it. Moreover, in instances of these kernel classes, a container and its containees must share the same workspace (derived classes may be more liberal in certain circumstances). This “managed ownership” [70] is our central strategy in thread safety.

As shown in figure 1.14(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 1.14(d) and (e). In (d), a reader first calls the getReadAccess() method of the Workspace class. That method does not return until it is safe to read data anywhere in the workspace. It is safe if there is no other thread concurrently holding (or requesting) a write lock on the workspace (the thread calling getReadAccess() may safely hold both a read and a write lock). When the user is finished reading the workspace data, it must call doneReading(). Failure to do so will result in no writer ever again gaining write access to the workspace. Because it is so important to call this method, it is enclosed in the finally clause of a try statement. That clause is executed even if an exception occurs in the body of the try statement.

The code for writers is shown in figure 1.14(e). The writer first calls the getWriteAccess() method of the Workspace class. That method does not return until it is safe to write into the workspace. It is

safe if no other thread has read or write permission on the workspace. The calling thread, of course, may safely have both read and write permission at the same time. Once again, it is essential that doneWriting() be called after writing is complete.

This solution, while not as conservative as the single monitor of figure 1.14(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.

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

The way to avoid this situation is to use the wait() method of Workspace, passing as an argument the object on which you wish to wait (see Workspace methods in figure 1.3). That method first relinquishes all read permissions before calling wait on the target object. When wait() returns, notice that it is possible that the topology has changed, so callers should be sure to re-read any topology-dependent information. In general, this technique should be used whenever a thread suspends while it holds read permissions.

1.7 Mutations

Often it is necessary to carefully constrain when changes can be made in a topology. For example, an application that uses the actor package to execute a model defined by a topology may require the topology to remain fixed during segments of the execution. The util subpackage of the kernel package provides support for carefully controlled mutations that can occur during the execution of a model. The relevant classes and interfaces are shown in figure 1.15. Also shown in the figure is the most useful mutation class, MoMLChangeRequest, which uses MoML to specify the mutation. That class is in the moml package.

The usage pattern involves a source that wishes to have a mutation performed, such as an actor (see the Actor Package chapter) or a user interface component. The originator creates an instance of the class ChangeRequest and enqueues that request by calling the requestChange() of any object in the Ptolemy II hierarchy. That object typically delegates the request to the top-level of the hierarchy, which in turn delegates to the manager. When it is safe, the manager executes the change by calling execute() on each enqueued ChangeRequest. In addition, it informs any registered change listeners of the mutations so that they can react accordingly. Their changeExecuted() method is called if the change succeeds, and their changeFailed() method is called if the change fails. The list of listeners is maintained by the manager, so when a listener is added to or removed from any object in the hierarchy, that request is delegated to the manager.

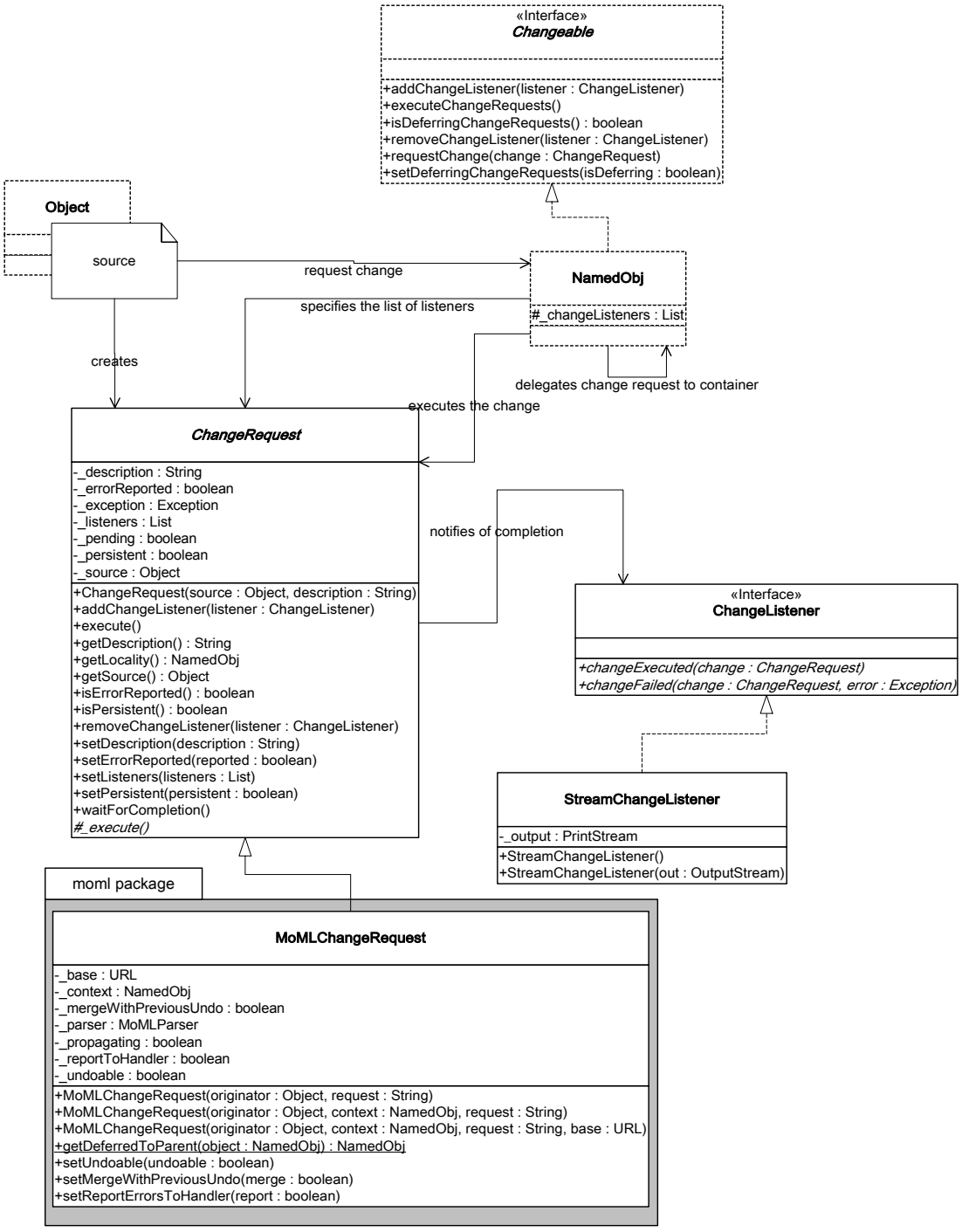

FIGURE 1.15. Classes and interfaces that support controlled topology mutations. A source requests topology changes and a manager performs them at a safe time.

1.7.1 Change Requests

A manager processes a change request by calling its execute() method. That method then calls the protected _execute() method, which actually performs the change. If the _execute() method completes successfully, then the ChangeRequest object notifies listeners of success. If the _execute() method throws an exception, then the ChangeRequest object notifies listeners of failure.

The ChangeRequest class is abstract. Its _execute() method is undefined. In a typical use, an originator will define an anonymous inner class, like this:

```
CompositeEntity container = ... ;
ChangeRequest change = new ChangeRequest(originator, "description") {
   protected void _execute() throws Exception {
      ... perform change here ...
   }
};
container.requestChange(change);
```

By convention, the change request is usually posted with the container that will be affected by the change. The body of the _execute() method can create entities, relations, ports, links, etc. For example, the code in the _execute() method to create and link a new entity might look like this:

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

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

A key concrete class extending ChangeRequest is implemented in the moml package, as shown in figure 1.15. The MoMLChangeRequest class supports specification of a change in MoML. See the MoML chapter for details about how to write MoML specifications for changes. The *context* argument to the second constructor typically gives a composite entity within which the commands should be interpreted. Thus, the same change request as above could be accomplished as follows:

```
CompositeEntity container = ... ;
String moml = "<group>"
   + "<entity name=\"\" class=\"MyEntityClass\"/>"
   + "<link port=\"portname\" relation=\"relationname\"/>"
   + "</group>";
ChangeRequest change =
   new MoMLChangeRequest(originator, container, moml);
container.requestChange(change);
```

1.7.2 NamedObj and Listeners

The NamedObj class provides addChangeListener() and removeChangeListener() methods, so that interested objects can register to be notified when topology changes occur. In addition, it provides a method that originators can use to queue requests, requestChange().

A change listener is any object that implements the ChangeListener interface, and will typically

include user interfaces and visualization components. The instance of ChangeRequest is passed to the listener. Typically the listener will call getOriginator() to determine whether it is being notified of a change that it requested. This might be used for example to determine whether a requested change succeeds or fails.

The ChangeRequest class also provides a waitForCompletion() method. This method will not return until the change request completes. If the request fails with an exception, then waitForCompletion() will throw that exception. Note that this method can be quite dangerous to use. It will not return until the change request is processed. If for some reason change requests are not being processed (due for a example to a bug in user code in some actor), then this method will never return. If you make the mistake of calling this method from within the event thread in Java, then if it never returns, the entire user interface will freeze, no longer responding to inputs from the keyboard or mouse, nor repainting the screen. The user will have no choice but to kill the program, possibly losing his or her work.

1.8 Actor-Oriented Classes

The kernel and kernel.util packages provide support for a significant innovation that was introduced with Ptolemy II version 4.0, namely actor-oriented classes, subclasses, and inner classes, with inheritance. In this mechanism, an entity can serve as either a class definition or as an instance of a class. It can also be a subclass of another entity that is a class definition. When a change is made to a class definition, then the change propagates to all subclasses and instances. The mechanism is described in [73].

The key principle that is followed in Ptolemy II is called the *derivation invariant*. The derivation invariant is an assertion that if any class definition contains an object (an entity, port, relation, attribute), then all subclasses and instances contain a *derived object*, which has the same name and Java class. The derived object is said to be *implied by* and *derived from* the first object. The first object is called the *prototype* of the derived object. Thus, the structure of a subclass or instance includes at least the objects that are included in the class definition.

A class definition is said to be the *parent* of a subclass or an instance. The subclass or instance is said to be the *child*.

Inner classes are supported in the following sense: a class definition x can contain an entity y that is itself a class definition. Consider the example shown in figure 1.16. If x has an instance x', then x' contains a class definition y', by the derivation invariant. If x contains an instance z of y, then x' contains an instance z' of y'. Notice that z' is an instance of y', but is also *implied by* z in x, by the derivation invariant. Thus, if a change is made to any of y, z or y', (e.g., an attribute is added) a corresponding change must be made to z' so that the derivation invariant is satisfied. This is a disciplined form of *multiple inheritance*.

As mentioned before, some attributes implement the Settable interface, shown in figure 1.5. Such attributes have a *value* (a representation of which is returned by the getExpression() method of the Settable interface). If an attribute is implied by another attribute, then by default it has the same value as

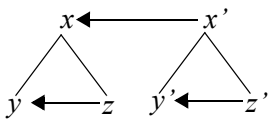

FIGURE 1.16. Example showing containment relations as vertical lines and parent-child relations as horizontal arrows.

the other object. However, that value can be *overridden*. Since there is multiple inheritance, there may be multiple paths by which a value propagates from an attribute to another that is derived from it. The key principle in Ptolemy II is that *local value changes take precedence over less local value changes.*

Consider the example given above. Suppose that y, z, y', and z' all have values. Suppose further than y' overrides that value. Then unless z' also overrides the value, then z' will inherit its value from y'. Suppose that after this override is established, the value of y is changed. That new value will be inherited by z, *but not by* y', and z', which override it. The reason is that the derivation path from y to y', and z' is higher in the hierarchy (and hence less local) than the path from y' to z'.

We will now outline how this mechanism is implemented. The two key interfaces are Derivable and Instantiable, shown in figure 1.3. NamedObj implements Derivable, which means that any NamedObj can be derived from another NamedObj. Only InstantiableNamedObj, shown in figure 1.2, implements Instantiable. Since Entity is a subclass of InstantiableNamedObj, an instance of Entity or any subclass can be either a class definition or an instance.

The Derivable interface has a getDerivedLevel() method that returns an indicator of locality. It has a getDerivedList() method that returns a list of derived objects, with more locally derived objects appearing earlier in the list than less locally derived objects. In the example above, the derived list for that y is $\{z, y', z'\}$, in that order. It has a getPrototypeList() method that returns the list of prototypes (if there are any) for a specified object. This list includes only direct prototypes (those not traversing more than one parent-child relation). So for z', the prototype list is $\{z, y'\}$ but not z.

The derivation invariant is realized by the propagateExistence() method of the Derivable interface. The inheritance of values is realized by the propagateValue() method.

The other key interface, Instantiable, supports the parent-child relation, and provides an instantiate() method that is used to create either subclasses or instances. Instantiation is accomplished by cloning, although there are significant subtleties in the implementation. For instance, when x in the above example is instantiated to create x', within the instance, it is important that the parent of z' is y' not y. The instantiate() method ensures this.

1.9 Exceptions

Ptolemy II includes a set of exception classes that provide a uniform mechanism for reporting errors that takes advantage of the identification of named objects by full name. These exception are summarized in the class diagram in figure 1.17.

1.9.1 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 an optional cause (a Throwable) plus an optional detail message (a String). The arguments provided are arranged in a default organization that is overridden in derived classes.

The cause argument to the constructor is a Throwable that caused the exception. The cause argument is used when code throws an exception and we want to rethrow the exception but print the stacktrace where the first exception occurred. This is called exception chaining.

JDK1.4 supports exception chaining. We are implementing a version of exception chaining here ourselves so that we can use JVMs earlier than JDK1.4.

In this implementation, we have the following differences from the JDK1.4 exception chaining implementation:

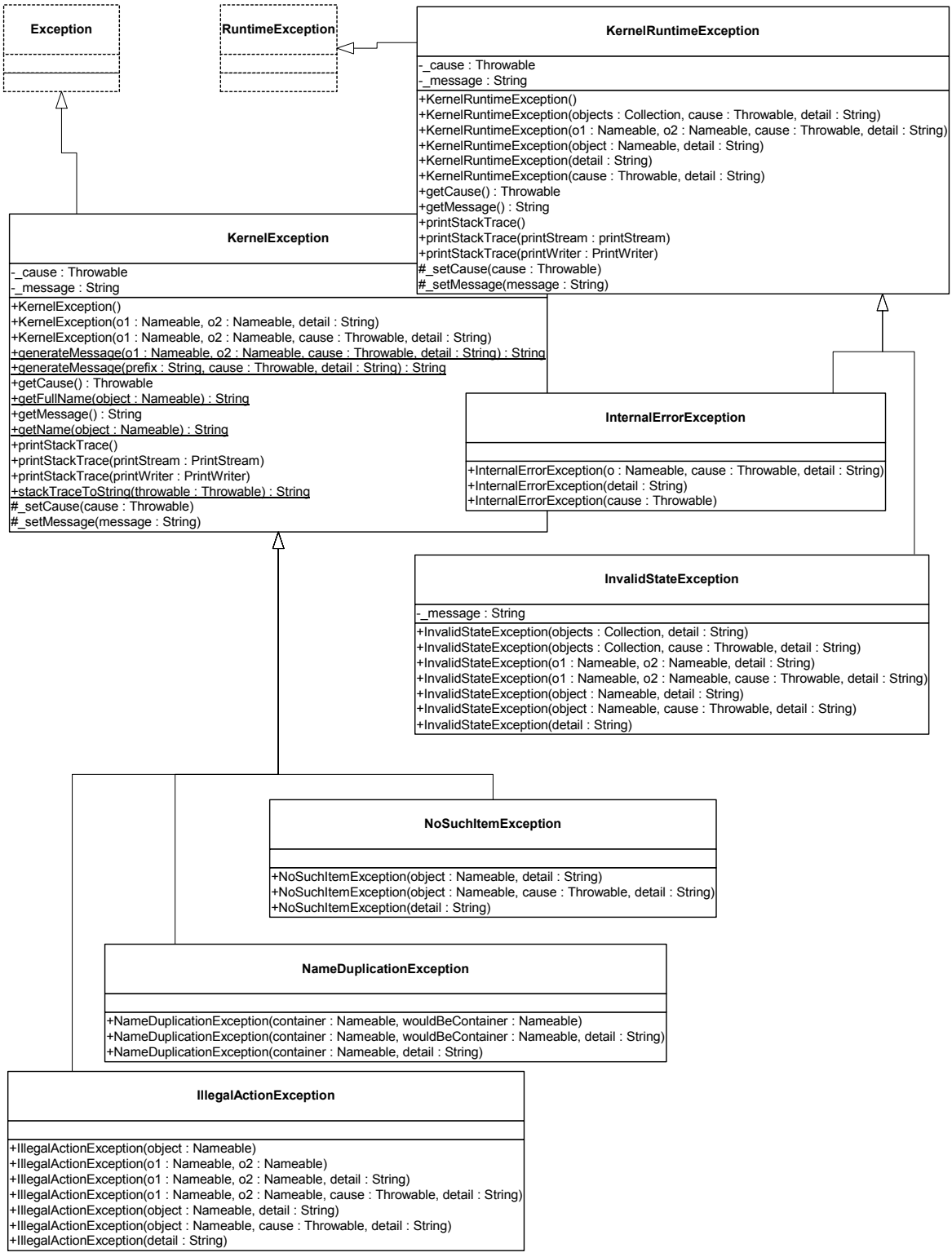

FIGURE 1.17. Summary of exceptions defined in the kernel.util package. These are used primarily through constructor calls. The form of the constructors is shown in the text. Exception and RuntimeException are Java exceptions.

- In this implementation, the detail message includes the detail message from the cause argument.
- In this implementation, we implement a protected _setCause() method, but not the public initCause() method that JDK1.4 has.

1.9.2 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. Example: an attempt to set the container of an object to be another object that cannot contain it because it is of the wrong class.

NameDuplicationException. Thrown on an attempt to add a named object to a collection that requires unique names, and finding that there already is an object by that name in the collection.

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

1.9.3 More Severe Exceptions

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

KernelRuntimeException. Base class for runtime exceptions. This class extends the basic Java RuntimeException with a constructor that can take a Nameable as an argument. This exception supports all the constructor forms of KernelException, but is implemented as a RuntimeException so that it does not have to be declared.. In particular, it provides methods that take zero, one, or two Nameable objects an optional cause (a Throwable) plus an optional detail message (a String). The arguments provided are arranged in a default organization that is overridden in derived classes. The cause argument is used to implement a form of exception chaining.

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

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