PTOLEMY II

HETEROGENEOUS CONCURRENT MODELING AND DESIGN IN JAVA

Edited by:
Christopher Brooks, Edward A. Lee, Xiaojun Liu, Steve Neuendorffer, Yang Zhao, Haiyang Zheng

VOLUME 2: PTOLEMY II SOFTWARE ARCHITECTURE

Authors:
Shuvra S. Bhattacharyya
Christopher Brooks
Elaine Cheong
John Davis, II
Mudit Goel
Bart Kienhuis
Edward A. Lee
Man-Kit Leung
Jie Liu
Xiaojun Liu
Lukito Muliadi
Steve Neuendorffer
John Reekie
Neil Smyth
Jeff Tsay
Brian Vogel
Winthrop Williams
Yuhong Xiong
Yang Zhao
Haiyang Zheng
Gang Zhou

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

Document Version 6.0
for use with Ptolemy II 6.0
January 11, 2007

Technical Report No. UCB/EECS-2007-8

Earlier versions:

- *UCB/ERL M05/22, UCB/ERL M04/16, UCB/ERL M03/28, UCB/ERL M02/23 UCB/ERL M99/40, UCB/ERL M01/12*

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award #CCR-0225610), the State of California Micro Program, Rome AFRL, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, National Instruments, and Toyota.

VOLUME 2

PTOLEMY II SOFTWARE ARCHITECTURE

This volume describes the software architecture of Ptolemy II. The first chapter covers the kernel package, which provides a set of Java classes supporting clustered graph topologies for models. Cluster graphs provide a very general abstract syntax for component-based modeling, without assuming or imposing any semantics on the models. The actor package begins to add semantics by providing basic infrastructure for data transport between components. The data package provides classes to encapsulate the data that is transported. It also provides an extensible type system and an interpreted expression language. The graph package provides graph-theoretic algorithms that are used in the type system and by schedulers in the individual domains. The plot package provides a visual data plotting utility that is used in many of the applets and applications. The codegen package is a templated based code generator similar to the Ptolemy Classic code generators. The copernicus package is a code generator that performs static analysis on Java class files to produce smaller, faster executable models.

Volume 1 gives an introduction to Ptolemy II, including tutorials on the use of the software, and volume 3 describes the domains, each of which implements a model of computation.

This page intentionally left mostly blank.

Contents

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-

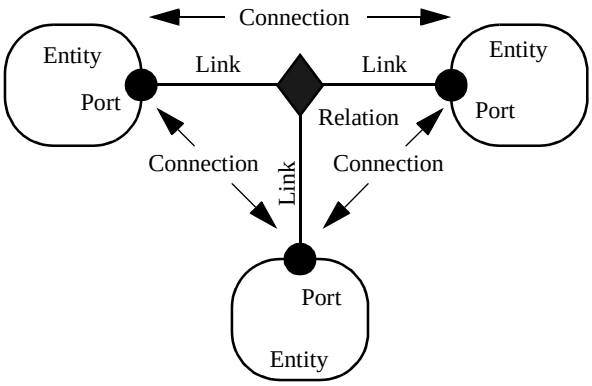


FIGURE 1.1. Visual notation and terminology.

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.

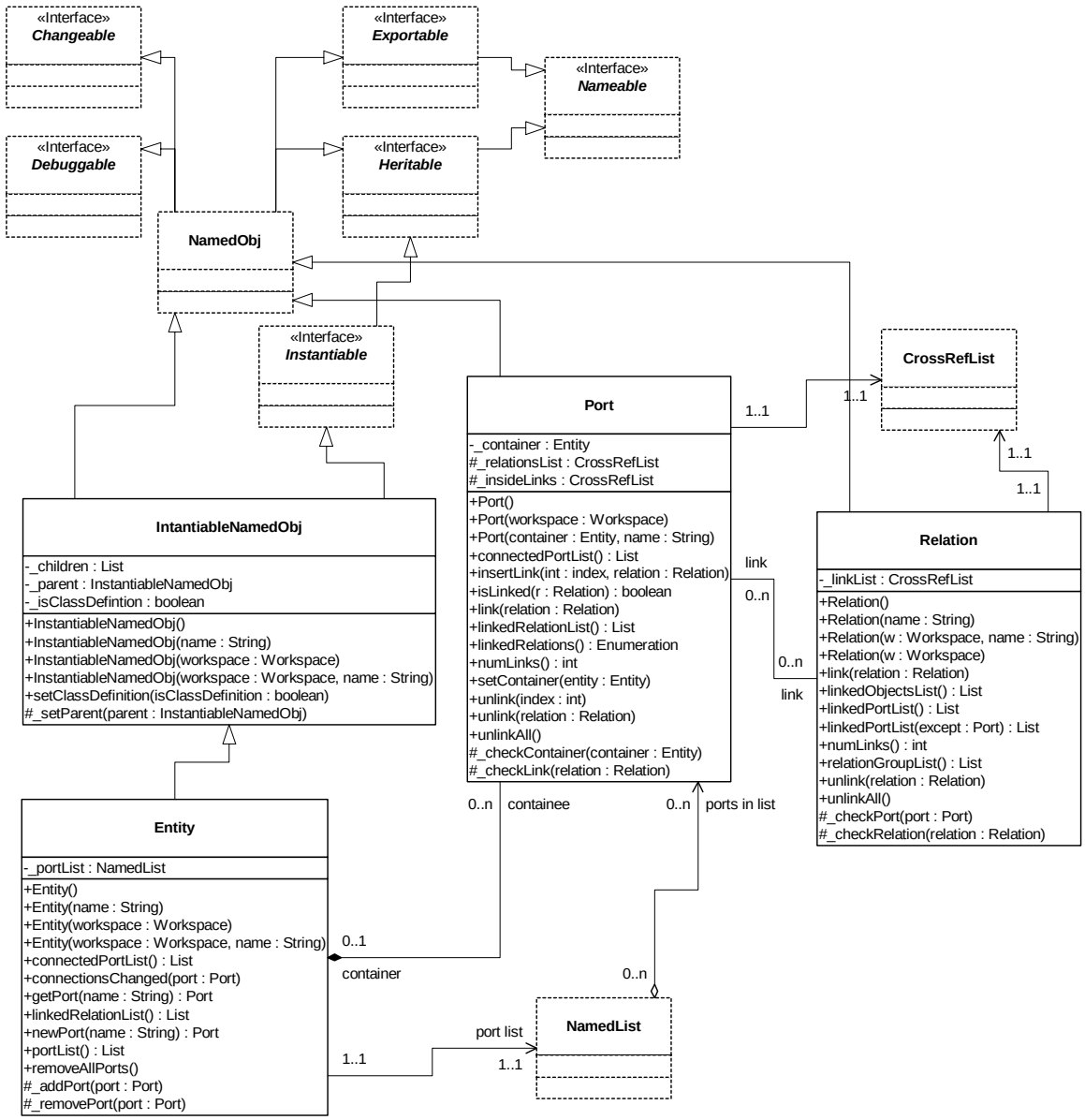


FIGURE 1.2. Key classes in the kernel package and their methods supporting basic (non-hierarchical) topologies. Methods that override those defined in a base class or implement those in an interface are not shown. The "+" indicates public visibility, "#" indicates protected, and "-" indicates private. 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" [74] 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 [137], 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).

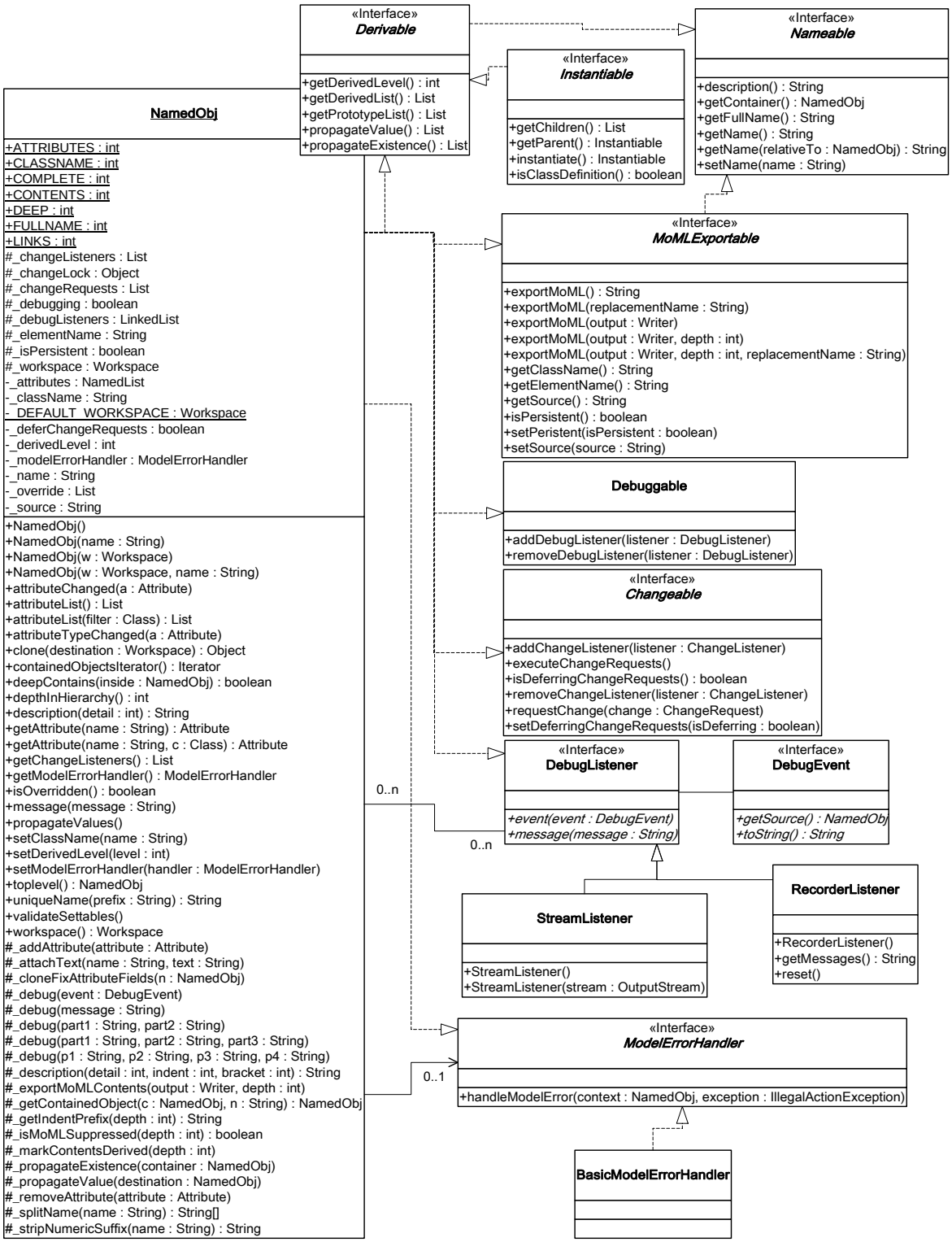


FIGURE 1.3. Support classes in the kernel.util package.

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

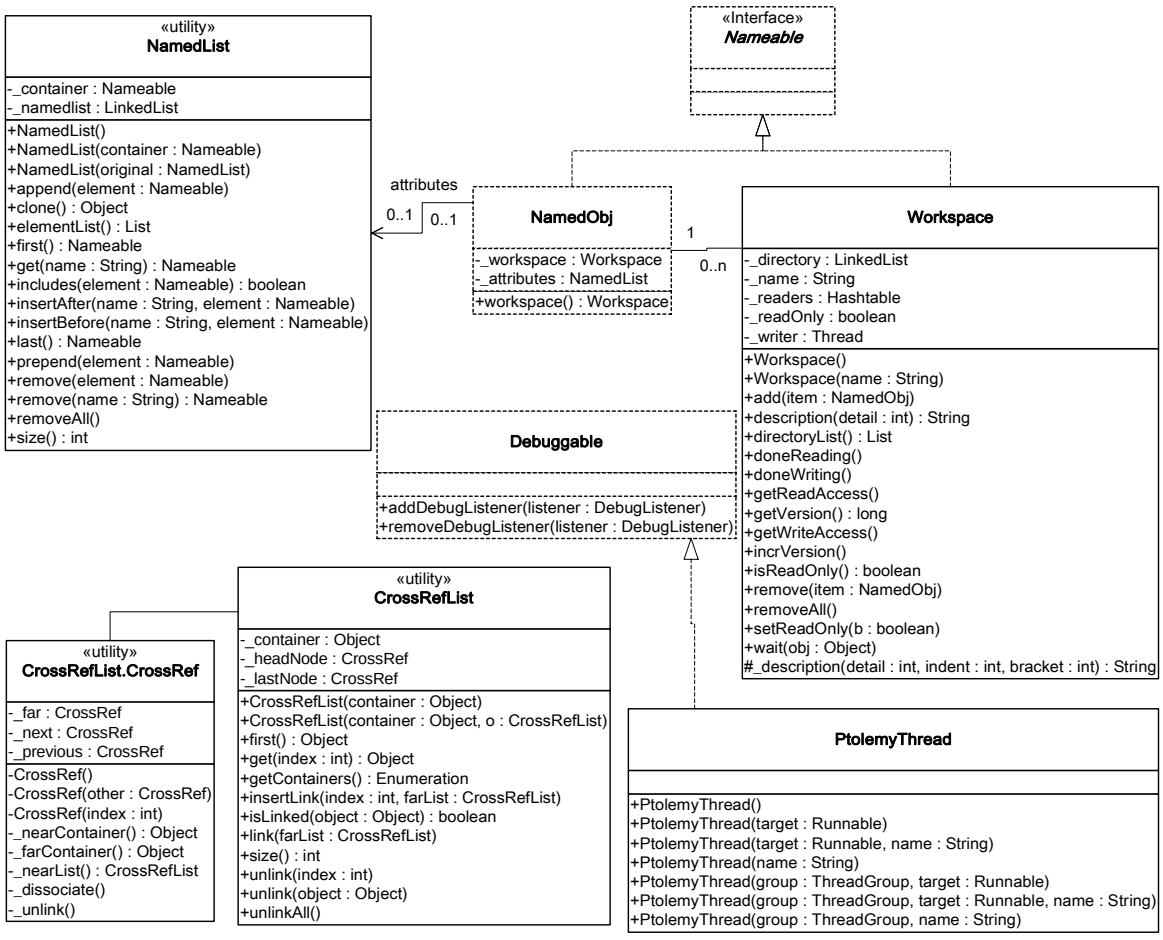


FIGURE 1.4. Some key utility classes. Workspace is the key gatekeeper class supporting multithreaded access to Ptolemy II models. It supports exclusive write access and shared read access. Every instance of NamedObj is associated with exactly one instance of Workspace. NamedList is a utility class used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

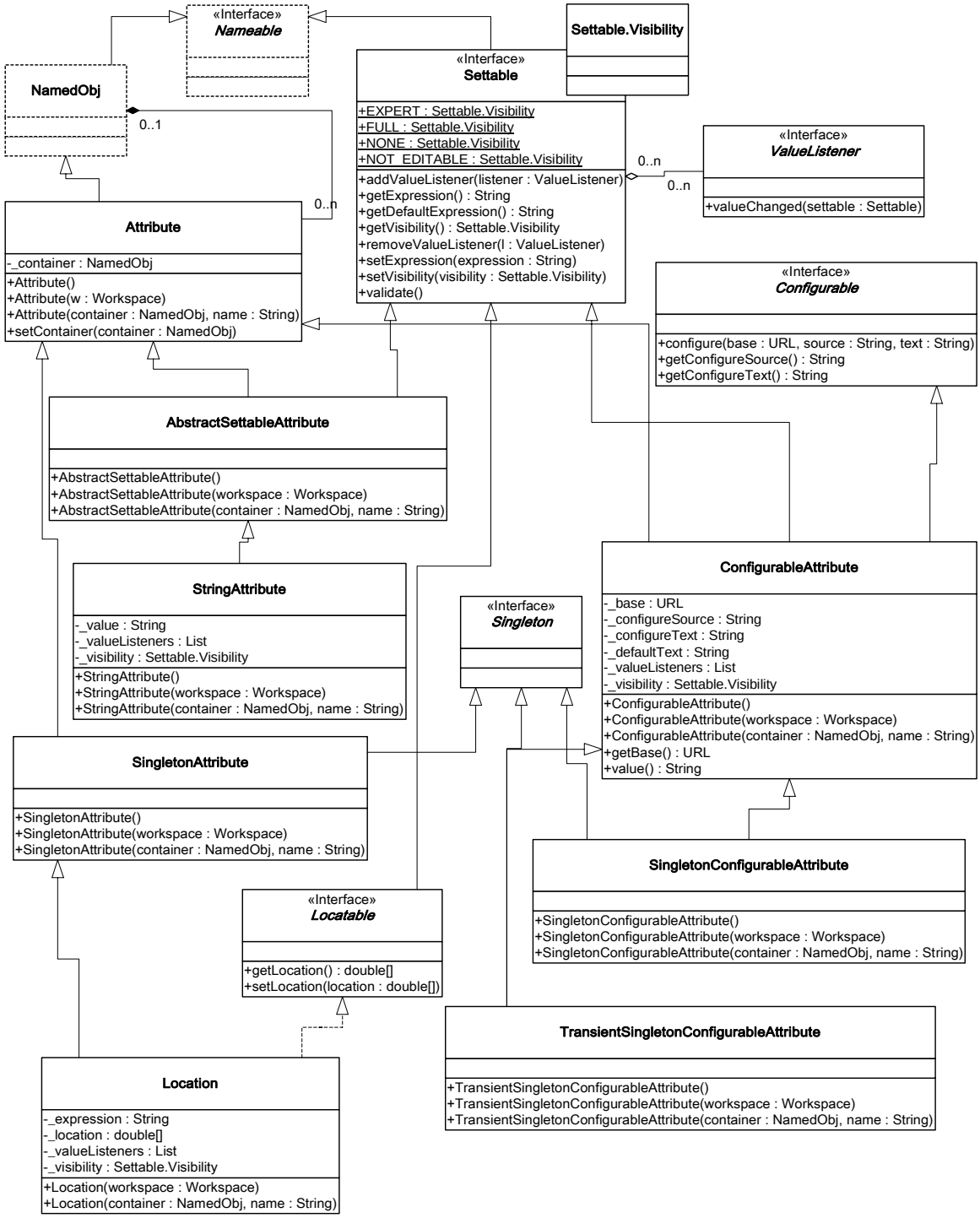


FIGURE 1.5. An instance of NamedObj can contain any number of instances of Attribute. The Ptolemy II kernel provides a few basic attributes, as shown here. Attributes that have values implement the Settable interface. Attributes whose values are numeric data, expressions, or data structures are described 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 [43].

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 [107]. 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.

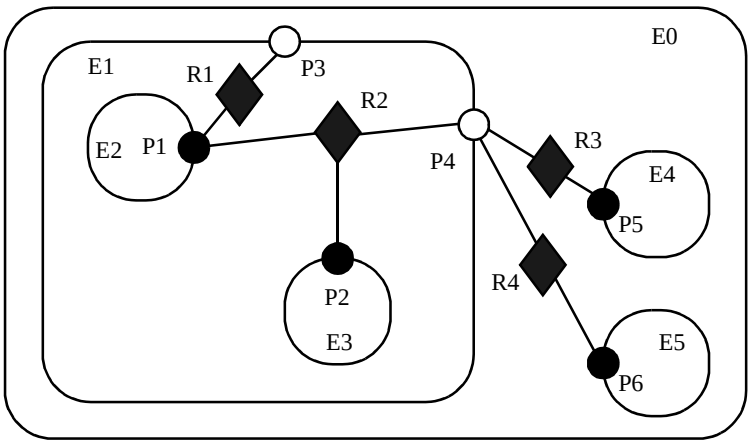


FIGURE 1.7. Transparent ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

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.

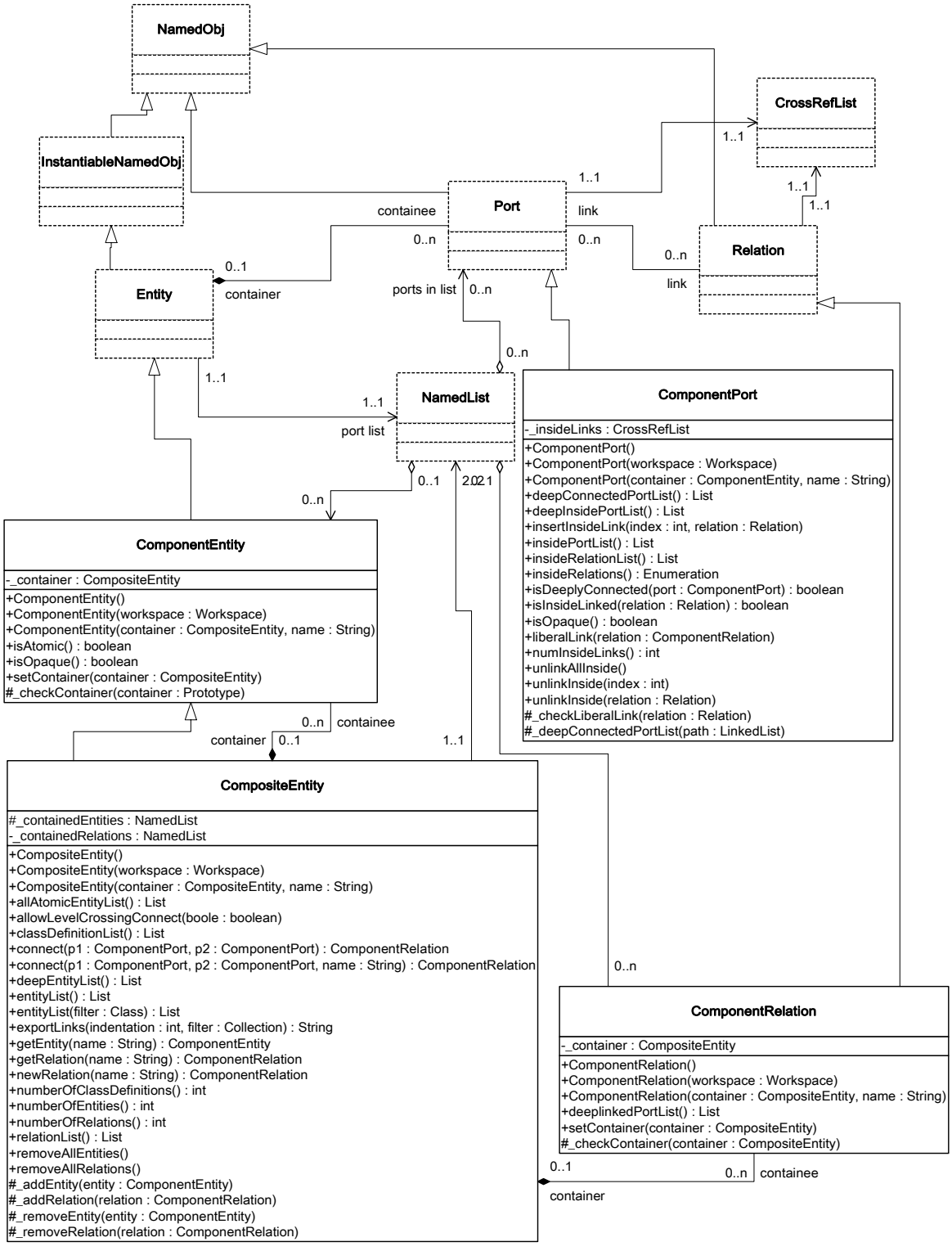


FIGURE 1.6. Key classes supporting clustered graphs.

1.4.2 Relation Groups

Relations mediate connections between ports. For flexibility, particularly with visual syntaxes, the Ptolemy II abstract syntax permits any number of relations to be involved in any one connection. Figure 1.8 illustrates this. Relations may be linked to other relations. Any two relations that are linked are said to be members of the same *relation group*. Specifically, a relation group is a maximal set of linked relations. Semantically, a relation group has the same meaning as a single relation. Thus, the two diagrams in figure 1.8 have the same meaning. The API of the Relation class, as shown in figure 1.2, support linking and unlinking relations, and also provides a method to obtain a list of all the relations in a relation group.

In a relation group, there is no significance to the order in which relations are linked, unlike the order in which ports are linked to relations. Also, unlike links between relations and ports, there is no significance to multiple links between the same relations. Any two relations are either linked or not linked.

1.4.3 Level-Crossing Connections

For a few applications, such as Statecharts [51], level-crossing links and connections are needed. The example shown in figure 1.9 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 1.9 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 1.9 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

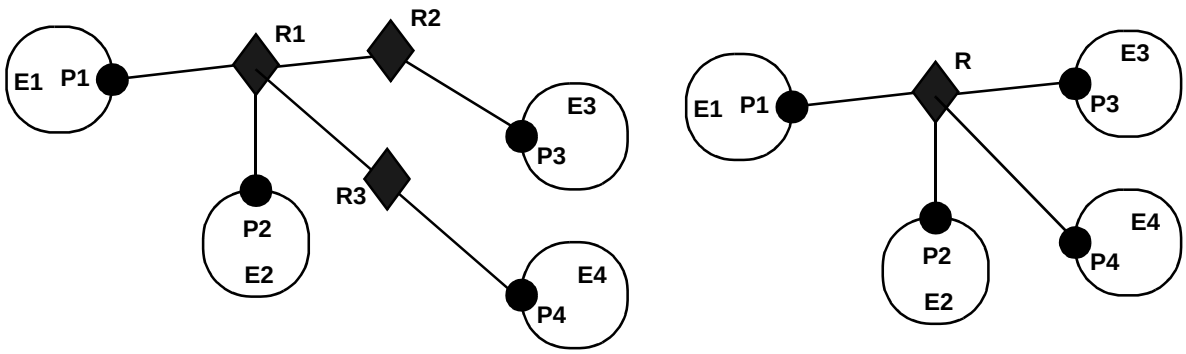


FIGURE 1.8. A relation group is maximal set of linked relations. At the left, R1, R2, and R3 form a relation group. A relation group is semantically identical to a single relation, so the two diagrams above have the same meaning.

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.

1.4.4 Tunneling Entities

The transparent port mechanism we have described supports connections like that between P1 and P5 in figure 1.10. That connection passes through the entity E2. The relation R2 is linked to the inside

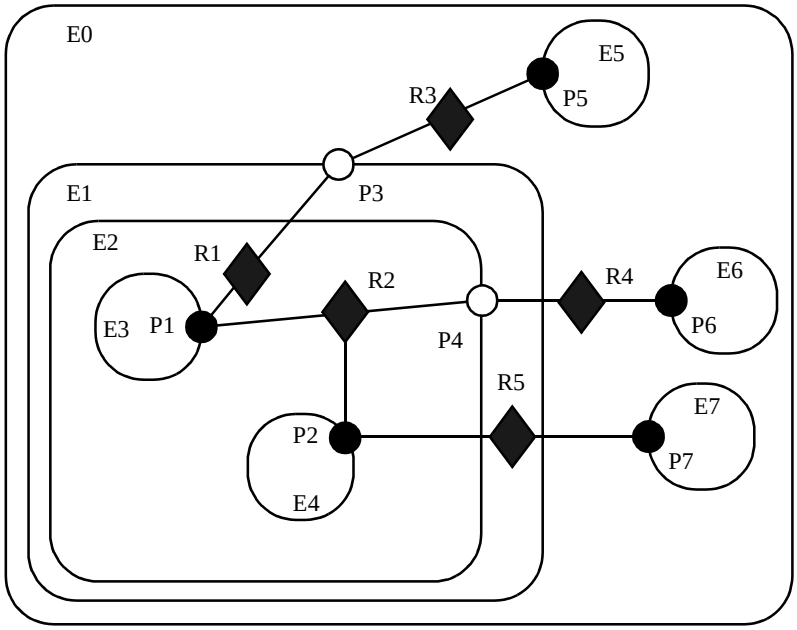


FIGURE 1.9. An example with level-crossing transitions.

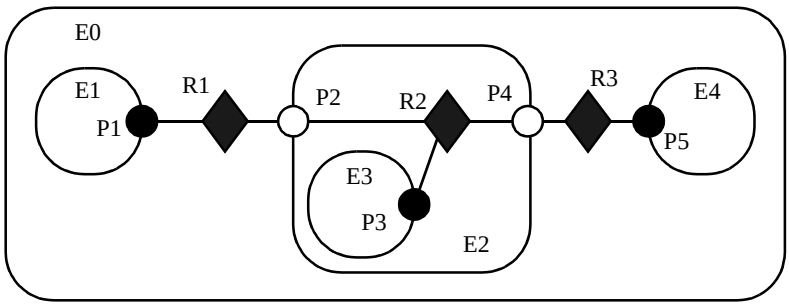


FIGURE 1.10. A tunneling entity contains a relation with inside links to more than one port.

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 [23] (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 [86] was made much more complicated. These higher-order functions had to be careful to avoid mutations that created tunneling.

1.4.5 Cloning

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

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

1.4.6 An Elaborate Example

An elaborate example of a clustered graph is shown in figure 1.11. This example includes instances of all the capabilities we have discussed. The top-level entity is named "E0." All other entities in this example have containers. A Java class that implements this example is shown in figure 1.12. A script in the Tcl language [122] that constructs the same graph is shown in figure 1.13. 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 1.12 and 1.13 is top-to-bottom and left-to-right in figure 1.11. 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 1.14. 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

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*.

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.

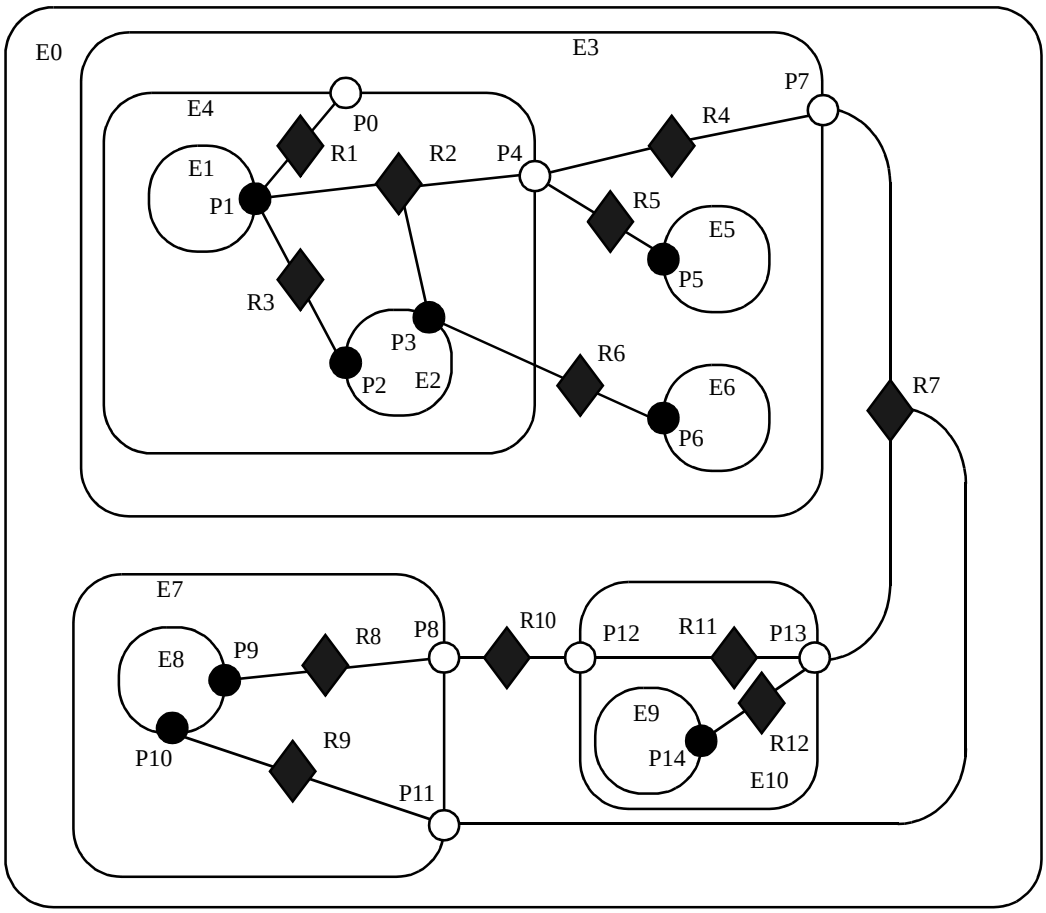


FIGURE 1.11. An example of a clustered graph.

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

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.*

```
# 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 1.13. The same topology as in figure 1.11 described by the Tcl commands to create it.

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 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 1.15. 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 1.15(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 1.15(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.

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 1.14. Key methods applied to figure 1.11.

One possible solution is to ensure that locks are always acquired in the same order [74]. 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:

```
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.

FIGURE 1.15. Using monitors for thread safety. The method used in Ptolemy II is in (d) and (e).

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" [74] is our central strategy in thread safety.

As shown in figure 1.15(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 1.15(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 1.15(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 1.15(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 1.16. 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.

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);
```

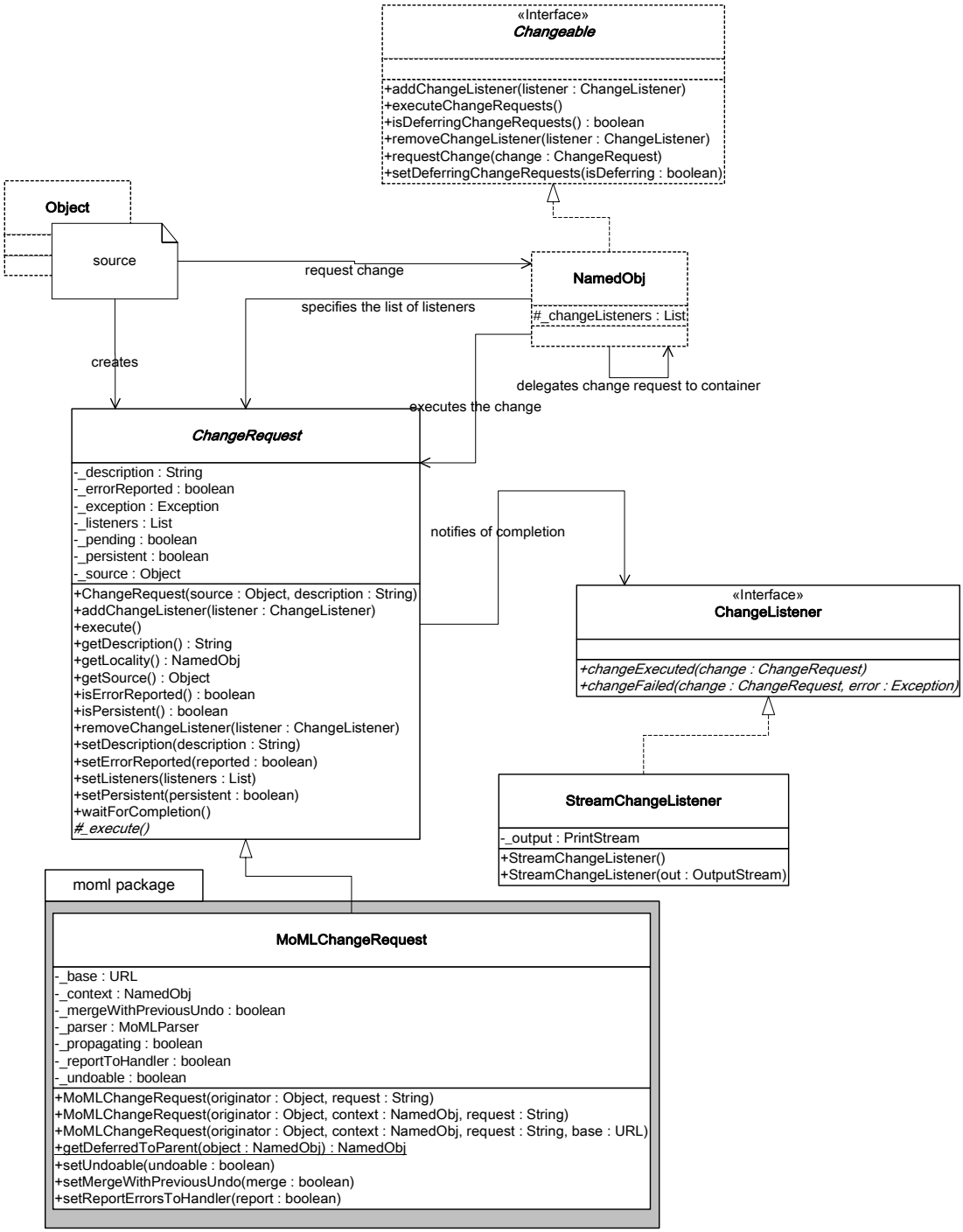


FIGURE 1.16. Classes and interfaces that support controlled topology mutations. A source requests topology changes and a manager performs them at a safe time.

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 1.16. 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 [79].

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 1.17. 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 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.

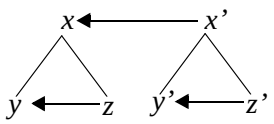


FIGURE 1.17. Example showing containment relations as vertical lines and parent-child relations as horizontal arrows.

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 1.18.

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:

- 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

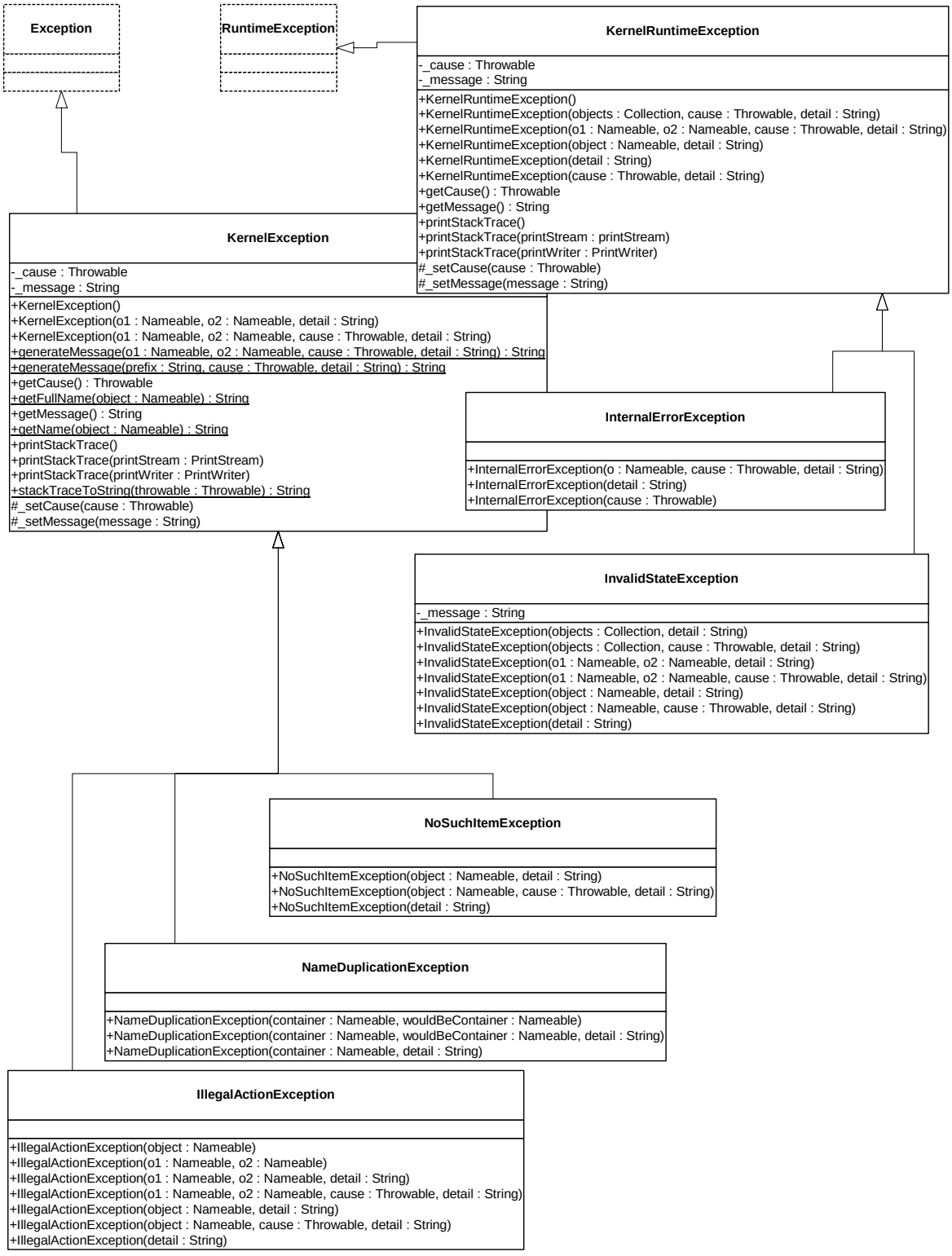


FIGURE 1.18. Summary of exceptions defined in the kernel.util package. These are used primarily through constructor calls. The form of the constructors is shown in the text. Exception and RuntimeException are Java exceptions.

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.

2

Actor Package

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

2.1 Concurrent Computation

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

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

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

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

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

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

2.2 Message Passing

The actor package provides templates for executable entities called *actors* that communicate with one another via message passing. Messages are encapsulated in *tokens* (see the Data Package chapter). Messages are sent and received via ports. IOPort is the key class supporting message transport, and is shown in figure 2.2. An IOPort can only be connected to other IOPort instances, and only via IORelations. The IORelation class is also shown in figure 2.2. TypedIOPort and TypedIORelation are subclasses that manage type resolution. These subclasses are used much more often, in order to benefit from the type system. This is described in detail in the Type System chapter.

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

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

2.2.1 Data Transport

Data transport is depicted in figure 2.1. The originating actor E1 has an output port P1, indicated in the figure with an arrow in the direction of token flow. The destination actor E2 has an input port P2, indicated in the figure with another arrow. E1 calls the send() method of P1 to send a token *t* to a remote actor. The port obtains a reference to a remote receiver (via the IORelation) and calls the put() method of the receiver, passing it the token. The destination actor retrieves the token by calling the get() method of its input port, which in turn calls the get() method of the designated receiver.

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

ment the communication protocol pertinent to that domain. A domain that uses asynchronous message passing, for example, can usually use the QueueReceiver shown in figure 2.2. A domain that uses synchronous message passing (rendezvous) has to provide a new receiver class.

In figure 2.1 there is only a single channel, indexed 0. The "0" argument of the send() and get() methods refer to this channel. A port can support more than one channel, however, as shown in figure 2.3. This can be represented by linking more than one relation to the port, or by linking a relation that has a width greater than one. A port that supports this is called a *multiport*. The channels are indexed $0, \ldots, N-1$, where N is the number of channels. An actor distinguishes between channels using this index in its send() and get() methods. By default, an IOPort is not a multiport, and thus supports only one channel (or zero, if it is left unconnected). It is converted into a multiport by calling its setMultiport() method with a *true* argument. After conversion, it can support any number of channels.

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

An IORelation, by default, represents a single channel. By calling its setWidth() method, however, it can be converted to a *bus*. A multiport may use a bus instead of multiple relations to distribute its data, as shown in figure 2.4. The *width of a relation* is the number of channels supported by the relation. If the relation is not a bus, then its width is one.

The *width of a port* is the sum of the widths of the relations linked to it. In figure 2.4, both the sending and receiving ports are multiports with width two. This is indicated by the "2" adjacent to each port. Note that the width of a port could be zero, if there are no relations linked to a port (such a port is said to be *disconnected*). Thus, a port may have width zero, even though a relation cannot. By convention, in Ptolemy II, if a token is sent from such a port, the token goes nowhere. Similarly, if a token is

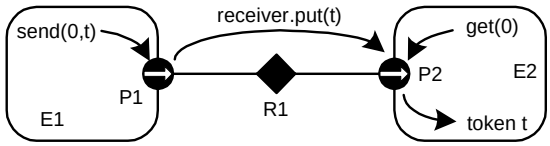


FIGURE 2.1. Message passing is mediated by the IOPort class. Its send() method obtains a reference to a remote receiver, and calls the put() method of the receiver, passing it the token *t*. The destination actor retrieves the token by calling the get() method of its input port.

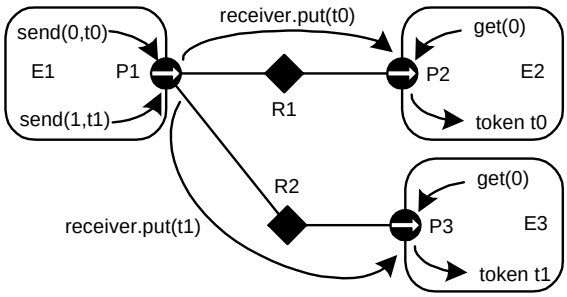


FIGURE 2.3. A port can support more than one channel, permitting an entity to send distinct data to distinct destinations via the same port. This feature is typically used when the number of destinations varies in different instances of the source actor.

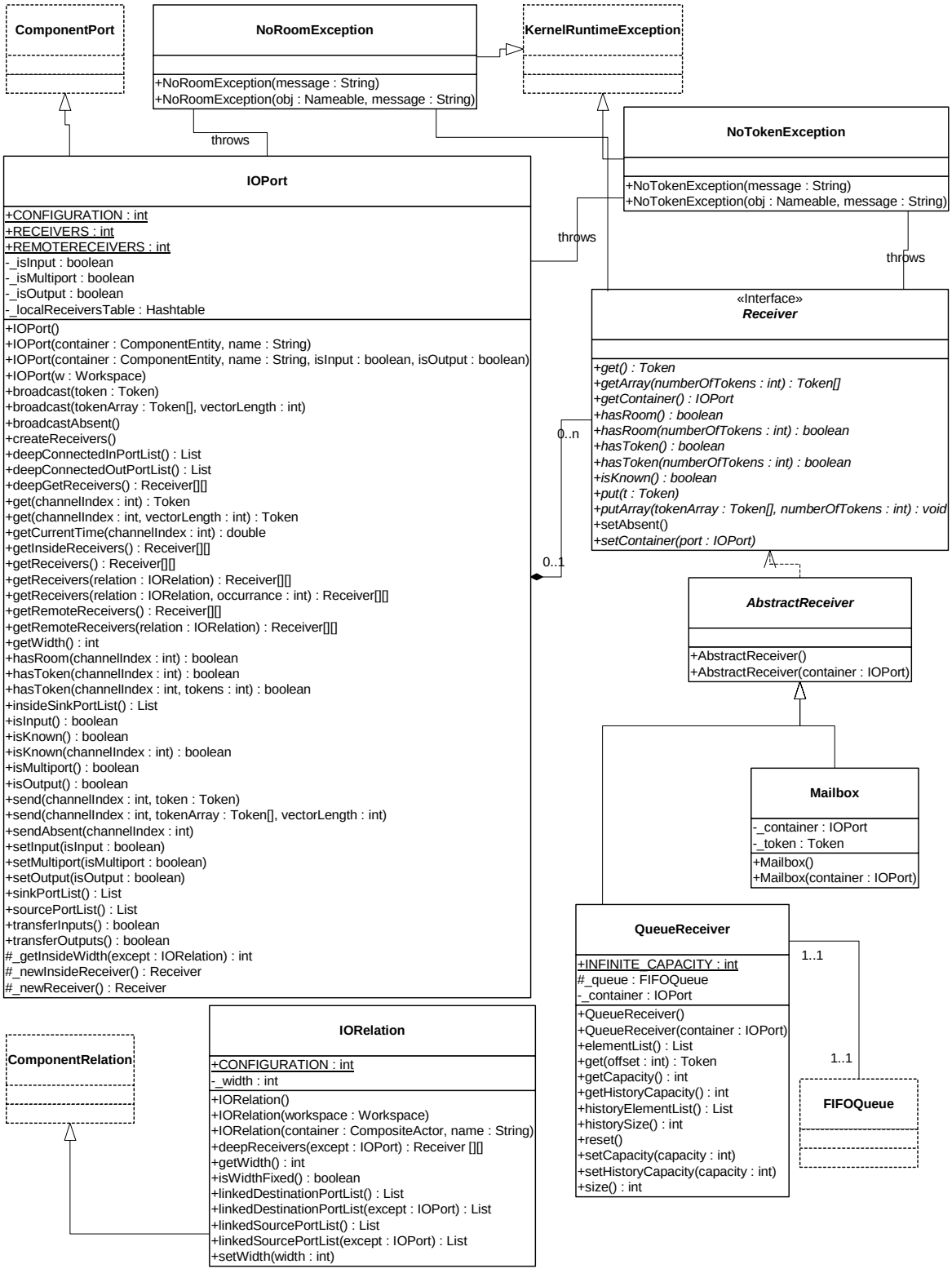


FIGURE 2.2. Port and receiver classes for message passing under various communication protocols.

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

A given channel may reach multiple ports, as shown in figure 2.5. This is represented by a relation that is linked to multiple input ports. In the default implementation, in class IOPort, a reference to the token is sent to all destinations. Note that tokens are assumed to be immutable, so the recipients cannot modify the value. This is important because in most domains, it is not obvious in what order the recipients will see the token.

The send() method takes a channel number argument. If the channel does not exist, the send() method silently returns without sending the token anywhere. This makes it easier for model builders, since they can simply leave ports unconnected if they are not interested in the output data.

IOPort provides a broadcast() method for convenience. This method sends a specified token to all receivers linked to the port, regardless of the width of the port. If the width is zero, of course, the token will not be sent anywhere.

2.2.2 Example

An elaborate example showing all of the above features is shown in figure 2.6. In that example, we assume that links are constructed in top-to-bottom order. The arrows in the ports indicate the direction of the flow of tokens, and thus specify whether the port is an input, an output, or both. Multiports are indicated by adjacent numbers larger than one.

The top relation is a bus with width two, and the rest are not busses. The width of port *P1* is four. Its first two outputs (channels zero and one) go to *P4* and to the first two inputs of *P5*. The third output of *P1* goes nowhere. The fourth becomes the third input of *P5*, the first input of *P6*, and the only input of *P8*, which is both an input and an output port. Ports *P2* and *P8* send their outputs to the same set of

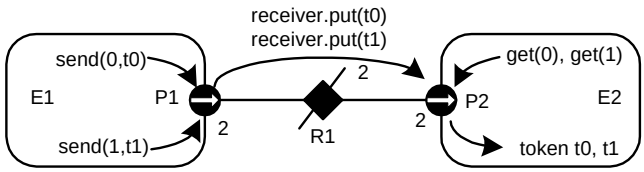


FIGURE 2.4. A bus is an IORelation that represents multiple channels. It is indicated by a relation with a slash through it, and the number adjacent to the bus is the width of the bus.

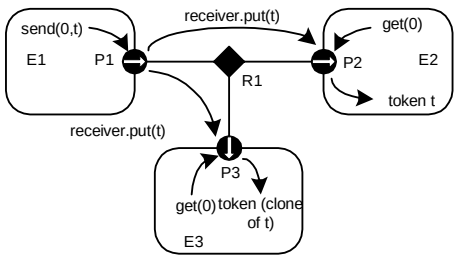


FIGURE 2.5. Channels may reach multiple destinations. This is represented by relations linking multiple input ports to an output port.

destinations, except that *P8* does not send to itself. Port *P3* has width zero, so its send() method returns without sending the token anywhere. Port *P6* has width two, but its second input channel has no output ports connected to it, so calling get(1) will trigger an exception that indicates that there is no data. Port *P7* has width zero so calling get() with any argument will trigger an exception.

2.2.3 Transparent Ports

Recall that a port is transparent if its container is transparent (isOpaque() returns *false*). A CompositeActor is transparent unless it has a local director. Figure 2.7 shows an elaborate example where busses, input, and output ports are combined with transparent ports. The transparent ports are filled in white, and again arrows indicate the direction of token flow. The Jacl code to construct this example is

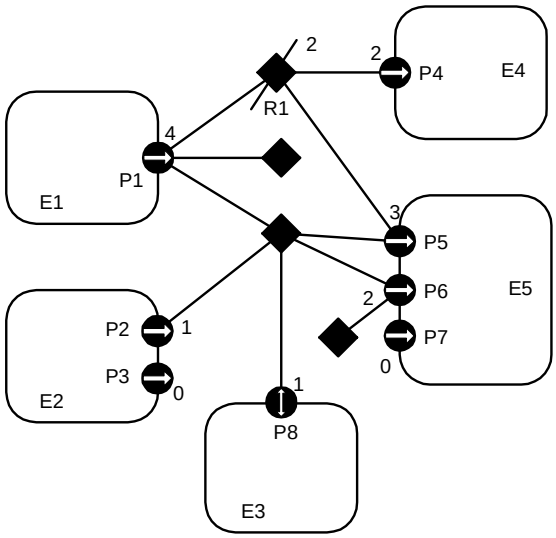


FIGURE 2.6. An elaborate example showing several features of the data transport mechanism.

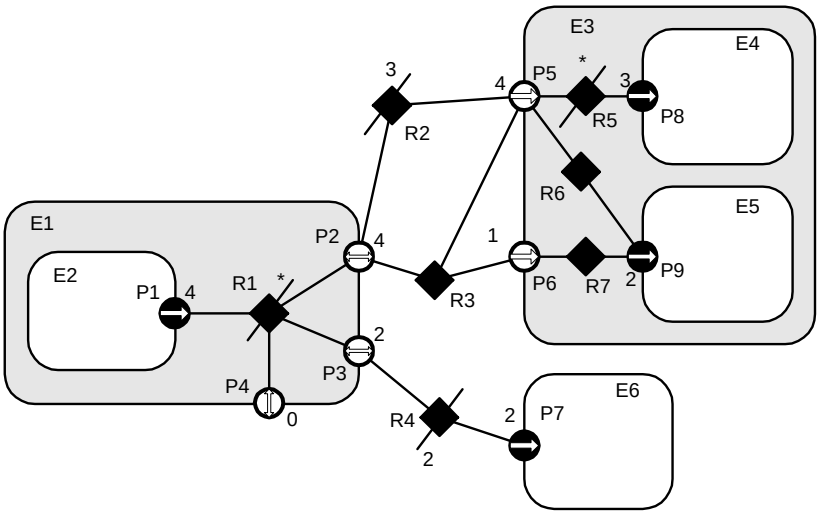


FIGURE 2.7. An example showing busses combined with input, output, and transparent ports.

shown in figure 2.8.

By definition, a transparent port is an input if either

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

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

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

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

Two of the relations that are inside composite entities (R1 and R5) are labeled as busses with a star (*) instead of a number. These are busses with unspecified width. The width is inferred from the topology. This is done by checking the ports that this relation is linked to from the inside and setting the width to the maximum of those port widths, minus the widths of other relations linked to those ports on the inside. Each such port is allowed to have at most one inside relation with an unspecified width, or an exception is thrown. If this inference yields a width of zero, then the width is defined to be one.

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

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

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

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

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

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

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

FIGURE 2.8. Tcl Blend code to construct the example in figure 2.7.

Thus, R1 will have width 4 and R5 will have width 3 in this example. The width of a transparent port is the sum of the widths of the relations it is linked to on the outside (just like an ordinary port). Thus, P4 has width 0, P3 has width 2, and P2 has width 4. Recall that a port can have width 0, but a relation cannot have width less than one.

When data is sent from P1, four distinct channels can be used. All four will go through P2 and P5, the first three will reach P8, two copies of the fourth will reach P9, the first two will go through P3 to P7, and none will go through P4.

By default, an IORelation is not a bus, so its width is one. To turn it into a bus with unspecified width, call setWidth() with a zero argument. Note that getWidth() will nonetheless never return zero (it returns at least one). To find out whether setWidth() has been called with a zero argument, call isWidthFixed() (see figure 2.2). If a bus with unspecified width is not linked on the inside to any transparent ports, then its width is one. It is not allowed for a transparent port to have more than one bus with unspecified width linked on the inside (an exception will be thrown on any attempt to construct such a topology). Note further that a bus with unspecified width is still a bus, and so can only be linked to multiports.

In general, bus widths inside and outside a transparent port need not agree. For example, if $M < N$ in figure 2.9, then first M channels from P1 reach P3, and the last $N - M$ channels are dangling. If $M > N$, then all N channels from P1 reach P3, but the last $M - N$ channels at P3 are dangling. Attempting to get a token from these channels will trigger an exception. Sending a token to these channels just results in loss of the token.

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

2.2.4 Data Transfer in Various Models of Computation

The receiver used by an input port determines the communication protocol. This is closely bound to the model of computation. The IOPort class creates a new receiver when necessary by calling its _newReceiver() protected method. That method delegates to the director returned by getDirector(), calling its newReceiver() method (the Director class will be discussed in section 2.3 below). Thus, the director controls the communication protocol, in addition to its primary function of determining the flow of control. Here we discuss the receivers that are made available in the actor package. This should not be viewed as an exhaustive set, but rather as a particularly useful set of receivers. These receivers are shown in figure 2.2.

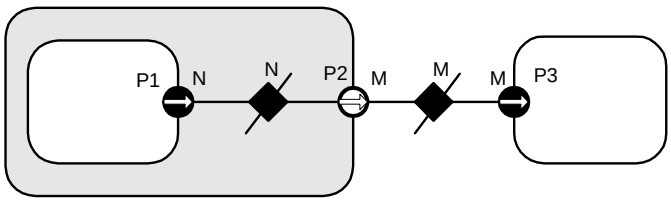


FIGURE 2.9. Bus widths inside and outside a transparent port need not agree..

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

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

In the Kahn process networks model of computation [66], which is a generalization of dataflow [86], each actor has its own thread of execution. The thread calling get() will stall if the corresponding queue is empty. If the size of the queue is bounded, then the thread calling put() may stall if the queue is full. This mechanism supports implementation of a strategy that ensures bounded queues whenever possible [124].

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

FIFOQueue is a support class that implements a first-in, first-out queue. It is part of the actor.util package, shown in figure 2.10. This class has two specialized features that make it particularly useful in this context. First, its capacity can be constrained or unconstrained. Second, it can record a finite or infinite history, the sequence of objects previously removed from the queue. The history mechanism is useful both to support tracing and debugging and to provide access to a finite buffer of previously consumed tokens.

An example of an actor definition is shown in figure 2.11. This actor has a multiport output. It reads successive input tokens from the input port and distributes them to the output channels. This actor is written in a domain-polymorphic way, and can operate in any of a number of domains. If it is

```
public class Distributor extends TypedAtomicActor {

    public TypedIOPort _input;
    public TypedIOPort _output;

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

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

FIGURE 2.11. An actor that distributes successive input tokens to a set of output channels.

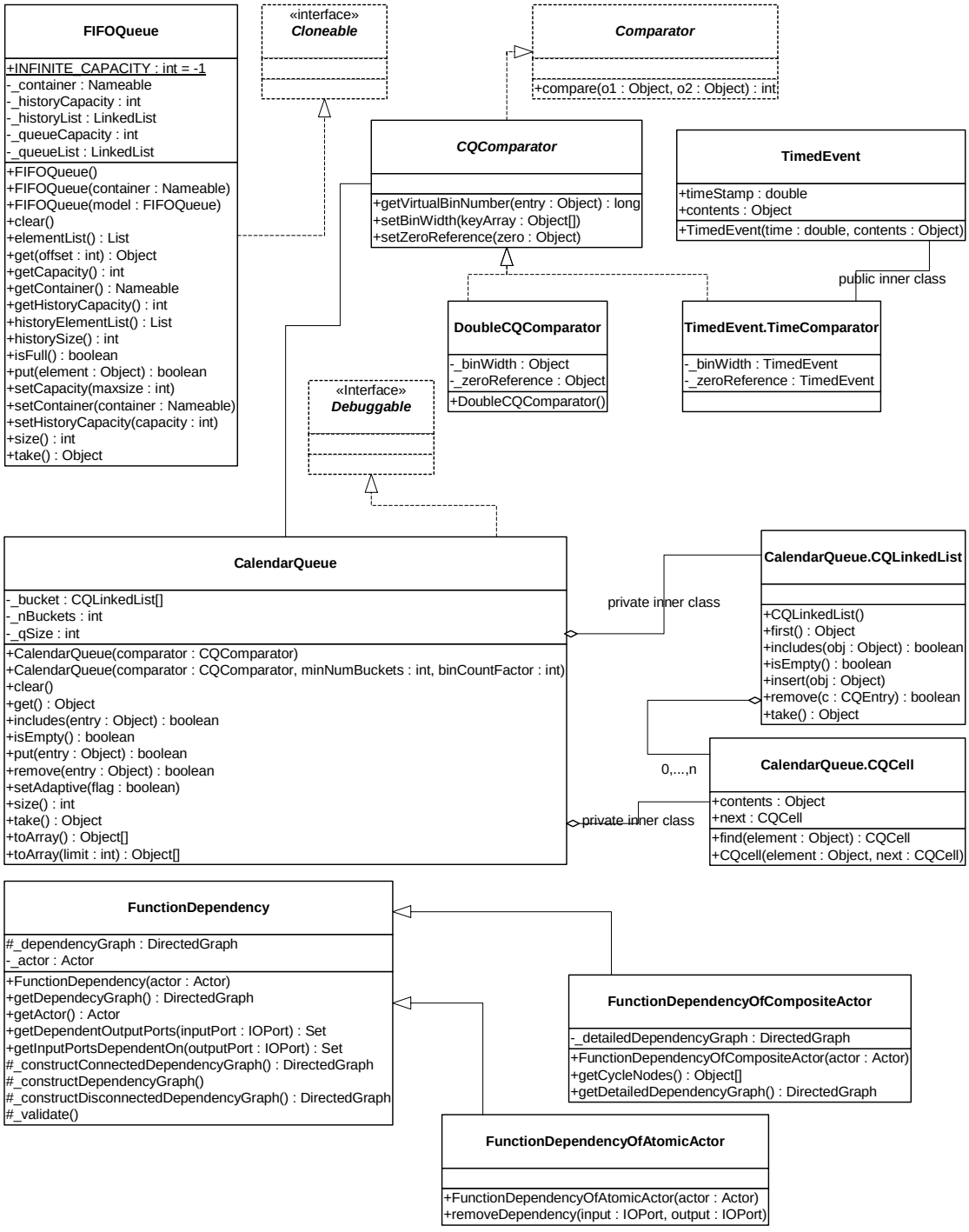


FIGURE 2.10. Static structure diagram for the actor.util package.

used in the PN domain, then its input will have a QueueReceiver and the output will be connected to ports with instances QueueReceiver.

Rendezvous Communications. Rendezvous, or synchronous communication, requires that the originator of a token and the recipient of a token both be simultaneously ready for the data transfer. As with process networks, the originator and the recipient are separate threads. The originating thread indicates a willingness to rendezvous by calling send(), which in turn calls the put() method of the appropriate receiver. The recipient indicates a willingness to rendezvous by calling get() on an input port, which in turn calls get() of the designated receiver. Whichever thread does this first must stall until the other thread is ready to complete the rendezvous.

This style of communication is implemented in the CSP domain. In the receiver in that domain, the put() method suspends the calling thread if the get() method has not been called. The get() method suspends the calling thread if the put() method has not been called. When the second of these two methods is called, it wakes up the suspended thread and completes the data transfer. The actor shown in figure 2.11 works unchanged in the CSP domain, although its behavior is different in that input and output actions involve rendezvous with another thread.

Nondeterministic transfers can be easily implemented using this mechanism. Suppose for example that a recipient is willing to rendezvous with any of several originating threads. It could spawn a thread for each. These threads should each call get(), which will suspend the thread until the originator is willing to rendezvous. When one of the originating threads is willing to rendezvous with it, it will call put(). The multiple recipient threads will all be awakened, but only one of them will detect that its rendezvous has been enabled. That one will complete the rendezvous, and others will die. Thus, the first originating thread to indicate willingness to rendezvous will be the one that will transfer data. Guarded communication [7] can also be implemented.

Discrete-Event Communication. In the discrete-event model of computation, tokens that are transferred between actors have a *time stamp*, which specifies the order in which tokens should be processed by the recipients. The order is chronological, by increasing time stamp. To implement this, a discrete-event system will normally use a single, global, sorted queue rather than an instance of FIFOQueue in each input port. The kernel.util package, shown in figure 2.10, provides the CalendarQueue class, which gives an efficient and flexible implementation of such a sorted queue.

2.2.5 Discussion of the Data Transfer Mechanism

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

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

2.3 Execution

The Executable interface, shown in figure 2.12, is implemented by the Director class, and is extended by the Actor interface. An *actor* is an executable entity. There are two types of actors, AtomicActor, which extends ComponentEntity, and CompositeActor, which extends CompositeEntity. As the names imply, an AtomicActor is a single entity, while a CompositeActor is an aggregation of actors. Two further extensions implement a type system, TypedAtomicActor and TypedCompositeActor.

The Executable interface defines how an object can be invoked. There are eight methods. The preinitialize() method is assumed to be invoked exactly once during the lifetime of an execution of a model and before the type resolution (see the type system chapter), and the initialize() methods is assumed to be invoked once after the type resolution. The initialize() method may be invoked again to restart an execution, for example, in the *-chart model (see the FSM domain). The prefire(), fire(), and postfire() methods will usually be invoked many times. The fire() method may be invoked several times between invocations of prefire() and postfire(). The stopFire() method is invoked to request suspension of firing. The wrapup() method will be invoked exactly once per execution, at the end of the execution.

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

An *iteration* is defined to be one invocation of prefire(), any number of invocations of fire(), and one invocation of postfire(). An *execution* is defined to be one invocation of preinitialize(), followed by one invocation of initialize(), followed by any number of iterations, followed by one invocation of wrapup(). The methods preinitialize(), initialize(), prefire(), fire(), postfire(), and wrapup() are called the *action methods*. While, the action methods in the executable interface are executed in order during the normal flow of an iteration, the terminate() method can be executed at any time, even during the execution of the other methods.

The preinitialize() method of each actor gets invoked exactly once. Typical actions of the preinitialize() method include creating receivers and defining the types of the ports. Higher-order function actors should construct their models in this method. The preinitialize() method cannot produce output data since type resolution is typically not yet done. It also gets invoked prior to any static scheduling that might occur in the domain, so it can change scheduling information.

The initialize() method of each actor gets invoked once after type resolution is done. It may be invoked again to restart the execution of an actor. Typical actions of the initialize() method include creating and initializing private data members. An actor may produce output data and schedule events in this method.

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

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

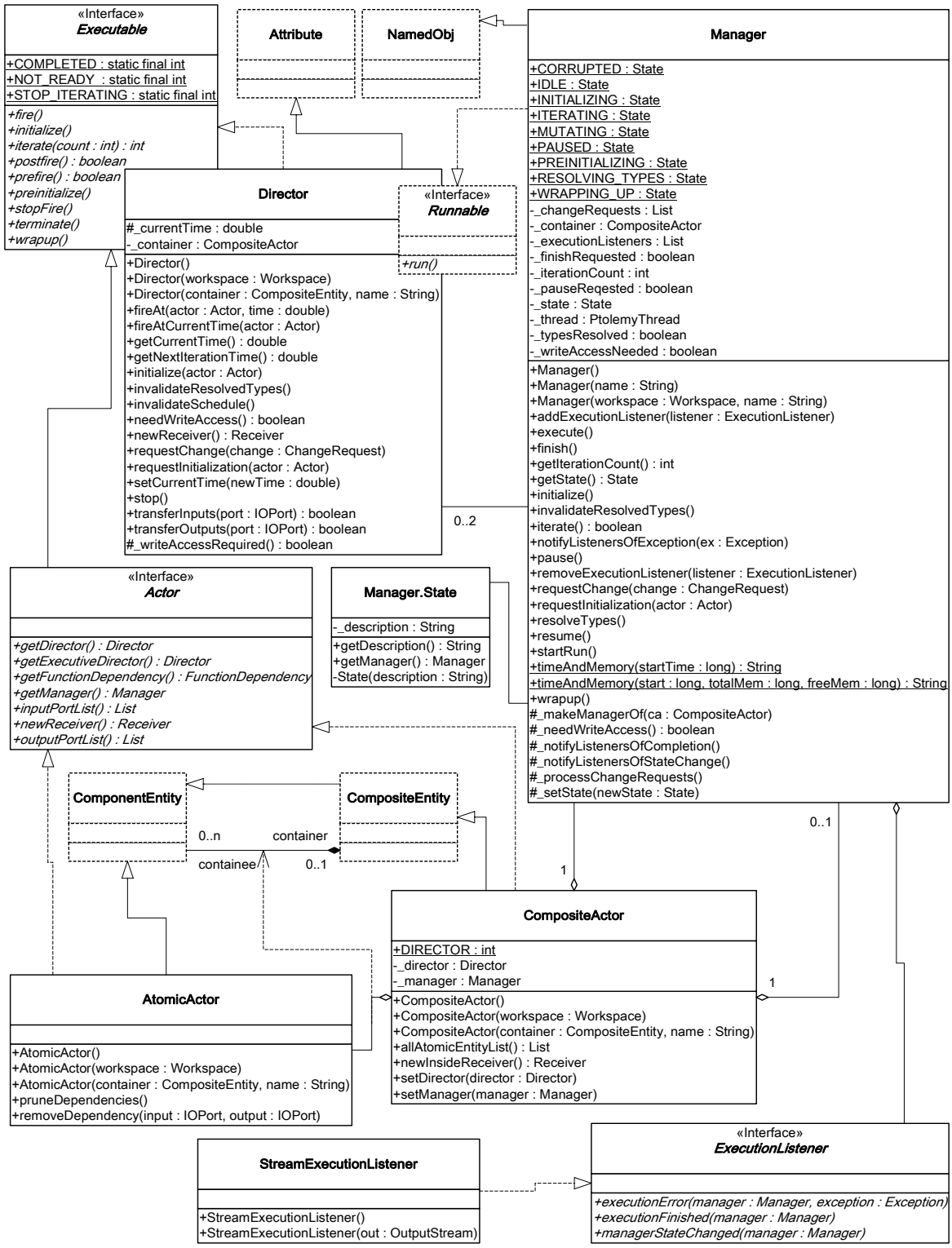


FIGURE 2.12. Basic classes in the actor package that support execution.

Instead, update the state in postfire().

In opaque composite actors, the fire() method is responsible for transferring data from the opaque ports of the composite actor to the ports of the contained actors, calling the fire() method of the director, and transferring data from the output ports of the composite actor to the ports of outside actors. See section 2.3.4 below.

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

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

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

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

2.3.1 Director

A *director* governs the execution of a composite entity. A *manager* governs the overall execution of a model. An example of the use of these classes is shown in figure 2.13. In that example, a top-level entity, E0, has an instance of Director, D1, that serves the role of its local director. A *local director* is responsible for execution of the components within the composite. It will perform any scheduling that might be necessary, dispatch threads that need to be started, generate code that needs to be generated, etc. In the example, D1 also serves as an executive director for E2. The *executive director* associated with an actor is the director that is responsible for firing the actor.

A composite actor that is not at the top level may or may not have its own local director. If it has a local director, then it defined to be opaque (isOpaque() returns *true*). In figure 2.13, E2 has a local director and E3 does not. The contents of E3 are directly under the control of D1, as if the hierarchy

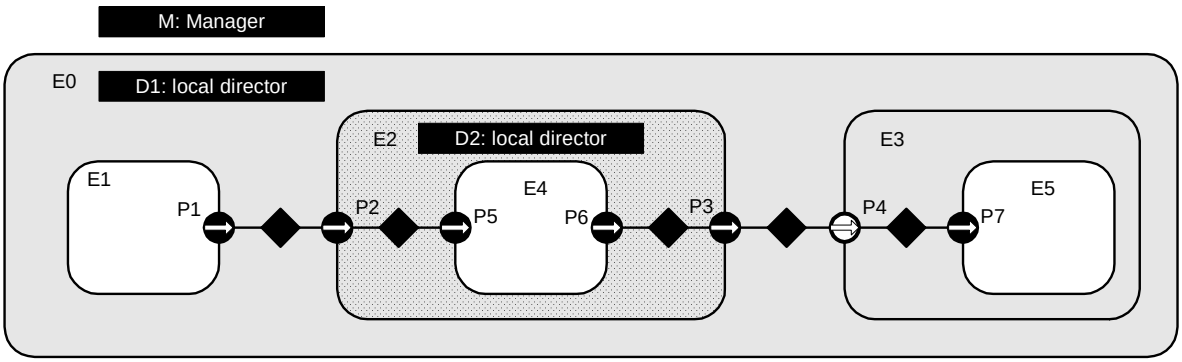


FIGURE 2.13. Example application, showing a typical arrangement of actors, directors, and managers.

were flattened. By contrast, the contents of E2 are under the control of D2, which in turn is under the control of D1. In the terminology of the previous generation, Ptolemy Classic, E2 was called a *wormhole*. In Ptolemy II, we simply call it a opaque composite actor. It will be explained in more detail below in section 2.3.4.

We define the *director* (vs. local director or executive director) of an actor to be either its local director (if it has one) or its executive director (if it does not). A composite actor that is not at the top level has as its executive director the director of the container. Every executable actor has a director except the top-level composite actor, and that director is what is returned by the getDirector() method of the Actor interface (see figure 2.12).

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

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

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

Two common sequences of method calls between actors and directors are shown in figure 2.14 and 2.15. These differ in the shaded areas, which define the domain-specific sequencing of actor firings. In figure 2.14, the fire() method of the director selects an actor, invokes its prefire() method, and if that returns true, invokes its fire() method some number of times (domain dependent) followed by its postfire() method. In figure 2.15, the fire() method of the director invokes the prefire() method of all the actors before invoking any of their fire() methods.

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

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

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

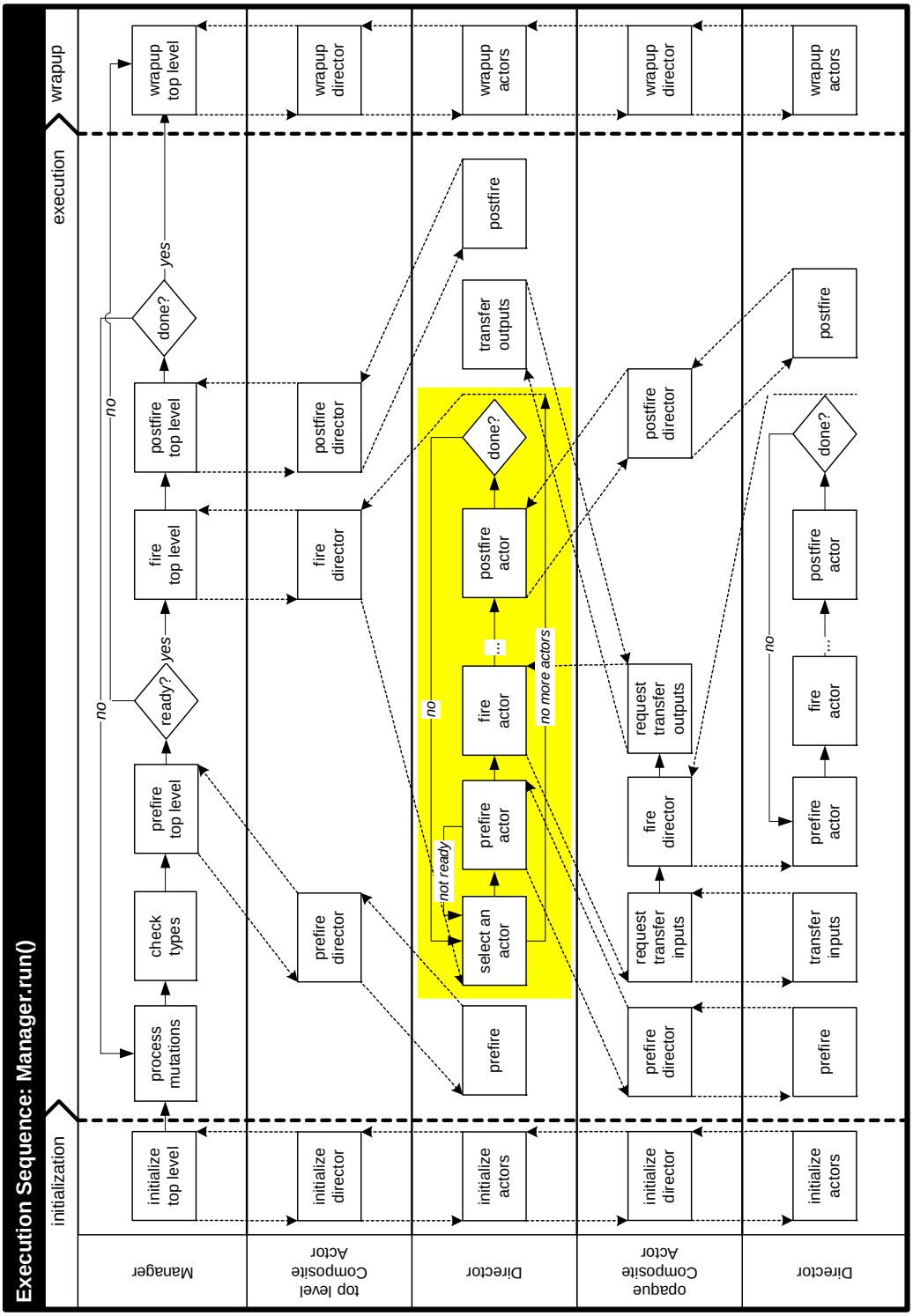


FIGURE 2.14. Example execution sequence implemented by run() method of the Director class.

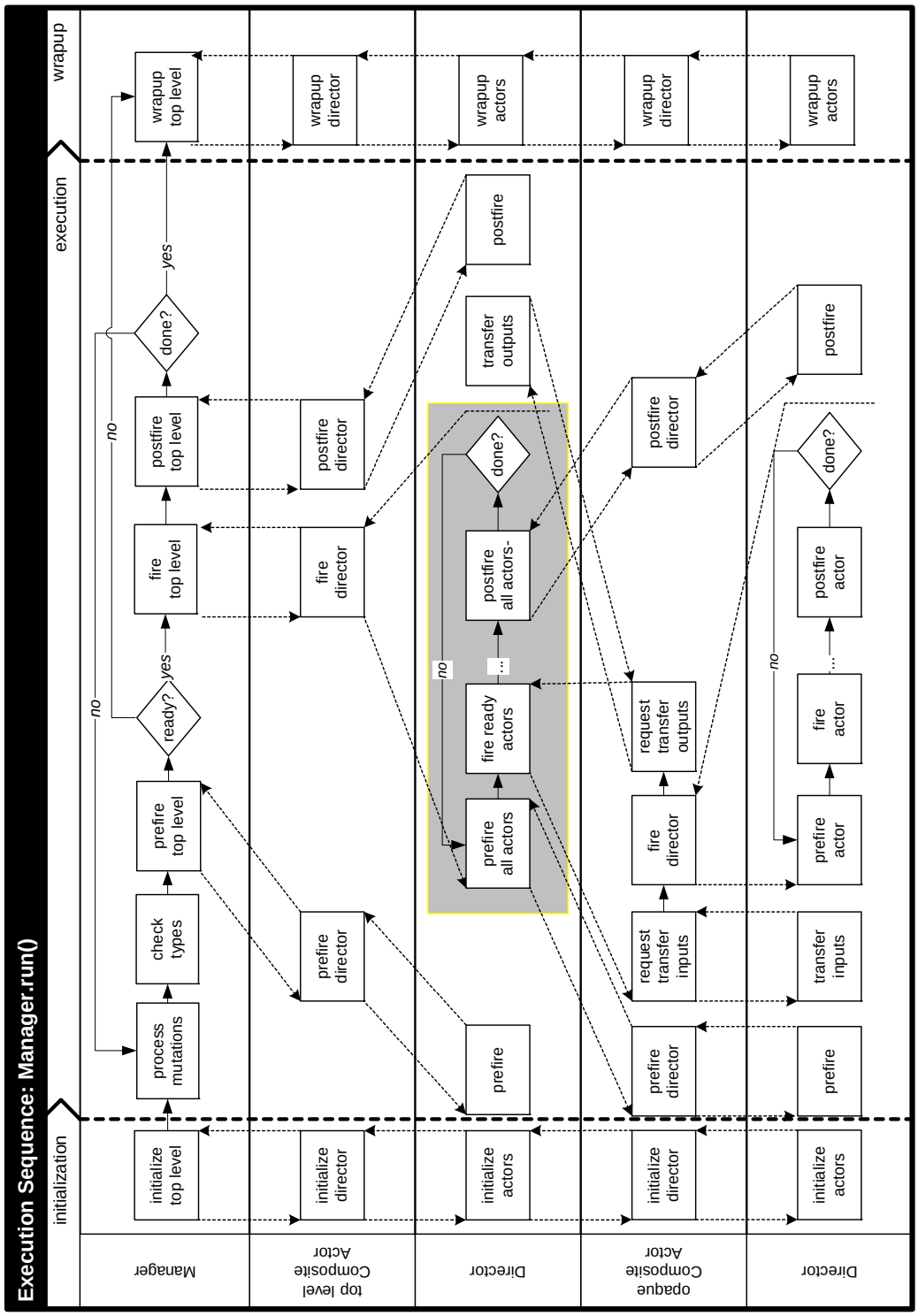


FIGURE 2.15. Alternative execution sequence implemented by run() method of the Director class.

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

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

2.3.2 Manager

While a director implements a model of computation, a *manager* controls the overall execution of a model. The manager interacts with a single composite actor, known as a *top level composite actor.* The Manager class is shown in figure 2.12. Execution of a model is implemented by three methods, execute(), run() and startRun(). The startRun() method spawns a thread that calls run(), and then immediately returns. The run() method calls execute(), but catches all exceptions and reports them to listeners (if there are any) or to the standard output (if there are no listeners).

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

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

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

2.3.3 ExecutionListener

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

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

A default implementation of the ExecutionListener interface is provided in the StreamExecution-

Listener class. This class reports all events on the standard output.

2.3.4 Opaque Composite Actors

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

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

The ports of a composite opaque actor are opaque, but it is a composite (it can contain actors and relations). This has a number of implications on execution. Consider the simple example shown in figure 2.16. Assume that both E0 and E2 have local directors (D1 and D2), so E2 is opaque. The ports of E2 therefore are opaque, as indicated in the figure by their solid fill. Since its ports are opaque, when a token is sent from the output port P1, it is deposited in P2, not P5.

In the execution sequences of figures 2.14 and 2.15, E2 is treated as an atomic actor by D1; i.e. D1 acts as an executive director to E2. Thus, the fire() method of D1 invokes the prefire(), fire(), and postfire() methods of E1, E2, and E3. The fire() method of E2 is responsible for transferring the token from P2 to P5. It does this by delegating to its local director, invoking its transferInputs() method. It then invokes the fire() method of D2, which in turn invokes the prefire(), fire(), and postfire() methods of E4.

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

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

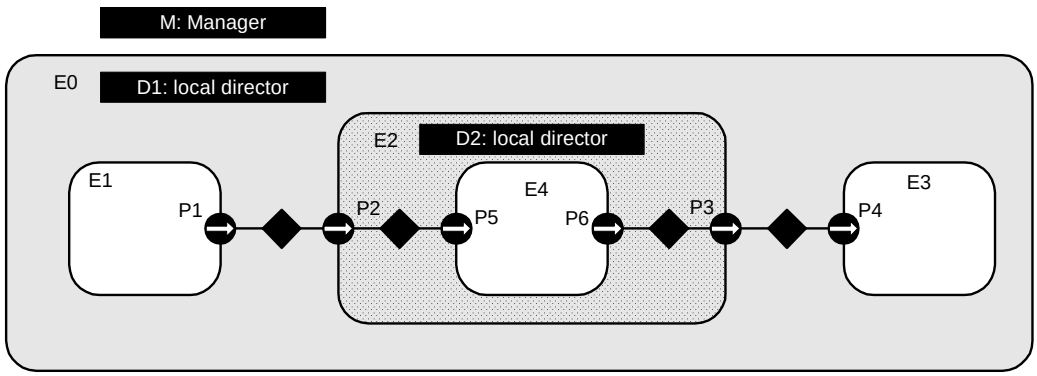


FIGURE 2.16. An example of an opaque composite actor. E0 and E2 both have local directors, not necessarily implementing the same model of computation.

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

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

2.4 Scheduler and Process Support

The ptolemy.actor.util package shown in figure 2.10 provides some infrastructure for domain designers by supporting efficient queues and dependency analysis. In addition, the actor package has two other subpackages, actor.sched, which provides rudimentary support for domains that use static schedulers to control the invocation of actors, and actor.process, which provides support for domains where actors are processes. The UML diagrams for these are shown in figure 2.17 and figure 2.18. This section describes some of this infrastructure.

2.4.1 Function Dependency

The FunctionDependency class and its subclasses in figure 2.10 provides support for domains that analyze data dependencies for scheduling or for checking correctness. In particular, an instance of FunctionDependency is associated with every actor and can be obtained from the getFunctionDependecy() method of the Actor interface (see figure 2.12). The instance of FunctionDependency describes the *function dependency* that output ports of the associated actor have on its input ports. An output port has a function dependency on an input port if in its fire() method, it sends tokens on the output port that depend on tokens gotten from the input port.

The FunctionDependency class uses a graph to describe the function dependency, where the nodes of the graph correspond to the ports and an edge indicates a function dependency. The edges go from input ports to output ports that depend on them. For atomic actors, this function dependency graph by default indicates that each output port depends on all input ports (this is called *complete dependency*). For some atomic actors, such as the TimedDelay actor in the DE domain, an output in a firing does not depend on an input port. Such actors override the pruneDependencies() method of AtomicActor (see figure 2.12) to remove dependencies between these ports. For example, the TimedDelay actor of the DE domain declares that its *output* port is independent of its *input* port by defining this method:

```
public void pruneDependencies() {
    super.pruneDependencies();
    removeDependency(input, output);
}
```

For composite actors, getFunctionDependency() returns an instance of FunctionDependencyOfCompositeActor (see figure 2.10). This class provides both the abstracted view, which gives the function dependency that output ports of the actor have on its input ports, and a *detailed view* from which it

constructs this information. The detailed view is a graph where the nodes correspond to the ports of the composite actor *and* to the ports of all deeply contained opaque actors, and the edges represent either the communication dependencies implied by the connections within the composite actor or the function dependencies of the contained opaque actors. The detailed view can be used by a director to construct a schedule. Also, the detailed view may reveal dependency loops, which in many domains means that the model cannot be executed. To check whether there are such loops, use the getCycleNodes() method. The method returns an array of IOPorts in such loops, of an empty array if there are no such loops.

2.4.2 Statically Scheduled Domains

The StaticSchedulingDirector class extends the Director base class to add a scheduler. The scheduler (an instance of the Scheduler class) creates an instance of the Schedule class which represents a statically determined sequence of actor firings. The scheduler also caches the schedule as necessary

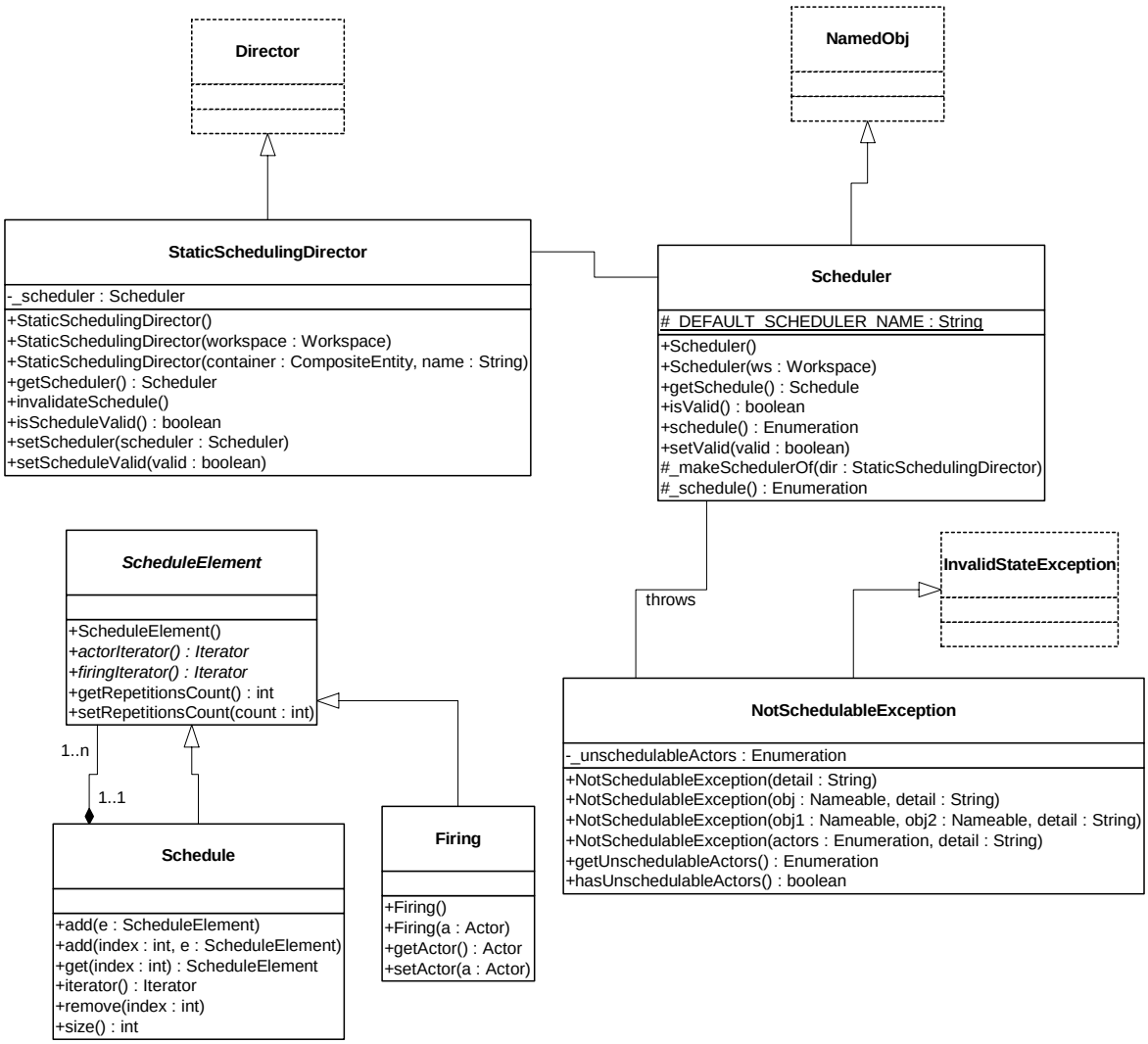


FIGURE 2.17. UML static structure diagram for the actor.sched package.

until it is invalidated by the director. This means that domains with a statically determined schedule (such as CT and SDF) need only implement the action methods in the director and a scheduler with the appropriate scheduling algorithm.

The Schedule base class contains a list of schedule elements, each with a repetitions factor that determines the number of times that element will be repeated. Since a schedule itself is a schedule ele-

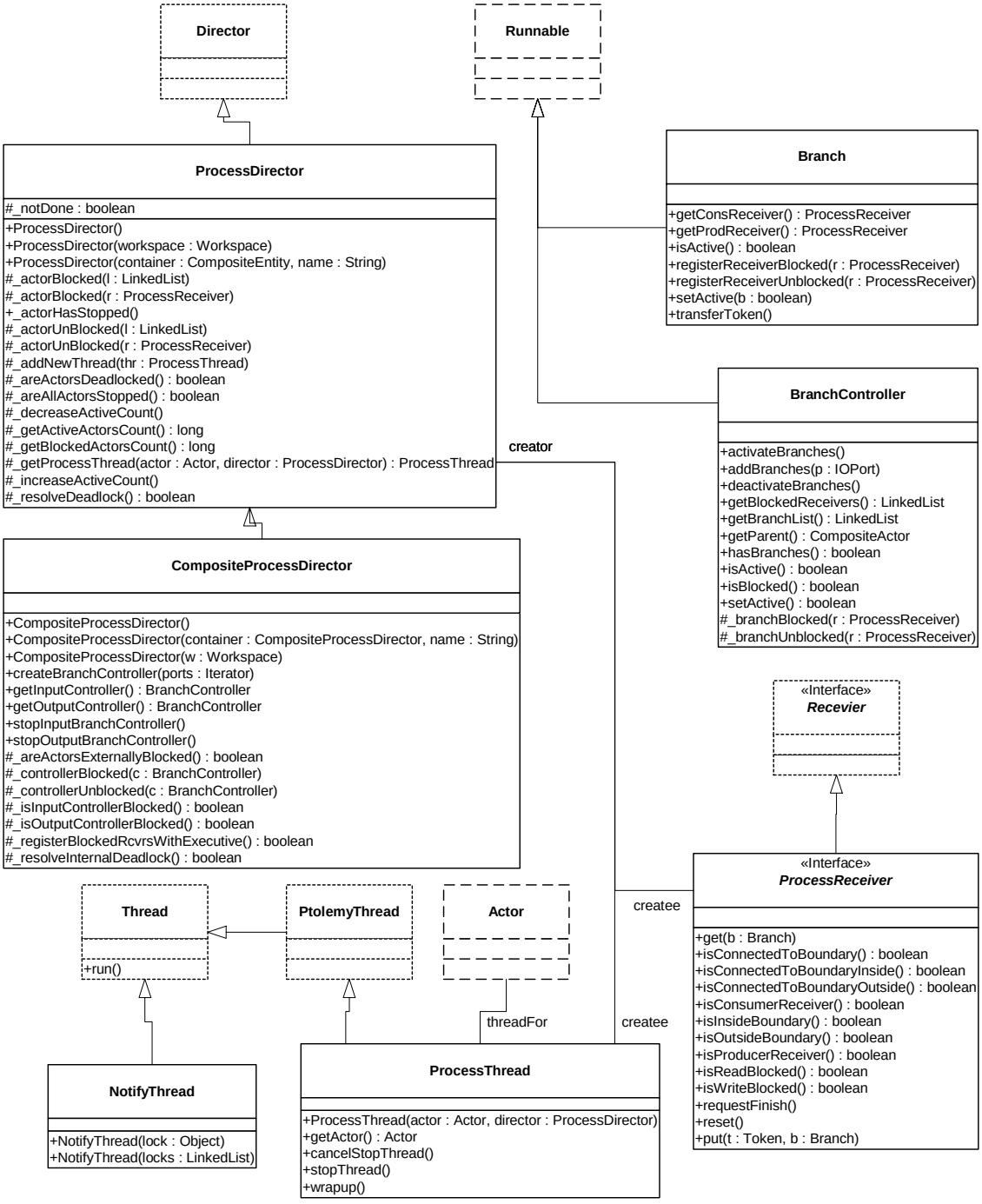


FIGURE 2.18. UML static structure diagram for the actor.process package.

ment, schedules can be defined recursively. Another type of schedule element is a firing, which represents a firing of a single actor. An iterator over all firings contained by a schedule is returned by the firingIterator() method on the schedule. In the iterator, the schedule is expanded recursively, with each firing repeated the appropriate number of times.[1]

2.4.3 Process Domains

Many domains, such as CSP, PN and DDE, consist of independent processes that are communicating in some way. These domains are collectively termed *process domains*. The actor.process package provides the following base classes that can be used to implement process domains.

ProcessThread. In a process domain, each actor represents an independently executing process. In Ptolemy II, this is achieved by creating a separate Java thread for each actor [120][74]. Each of these threads is an instance of ptolemy.actor.ProcessThread.

The thread for each actor is started in the prefire() method of the director. After starting, this thread repeatedly calls the prefire(), fire(), and postfire() methods of its associated actor. This sequence continues until the actor's postfire() method of returns false. The only way for an actor to terminate gracefully in PN is by returning from its fire() method and then returning false in its postfire() method. If an actor finishes execution as above, then the thread calls the wrapup() method of the actor. Once this method returns, the thread informs the director about the termination of this actor and finishes its own execution. The actor will not be fired again unless the director creates and starts a new thread for the actor.

ProcessReceiver. In the process domains, receivers represent the communication and synchronization points between different threads. To facilitate creating these domains, receivers in process domains should implement the ProcessReceiver interface. This interface provides extended information about status of the receiver, and the threads that may be interacting with the receiver.

ProcessDirector and CompositeProcessDirector. These classes are base classes for directors in the process-based domains. It provides some basic infrastructure for creating and managing threads. Most importantly, it provides a strategy pattern for handling deadlock between threads. Subclasses usually override methods in this class to handle deadlock in a domain-dependent fashion. In order to detect deadlocks, this base class maintains a count of how many actors in the system are executing and how many are blocked for some reason. This method of detecting deadlock is suggested in [73]. When no threads are able to run, the director calls the _resolveDeadlock() method to attempt to resolve the deadlock.

The initialize() method of the process director creates the receivers in the input ports of the actors, creates a thread for each actor and initializes these actors. It also initializes the count of active actors in the model to the number of actors in the composite actor. The prefire() method starts up all the created threads. This method returns true by default. The fire() method of a process director does not actually fire any contained actors. Instead, each actor is iterated by its associated process thread. The fire method simply blocks the calling thread until deadlock of the process threads occurs. In this case, the calling thread is unblocked and the fire method returns. The postfire() method simply returns true if the director was able to resolve the deadlock at the end of the fire method, or false otherwise. Returning true implies that if some new data is provided to the composite actor it can resume execution. Return-

1. Note that creating an iterator does not require expanding the data structure of the schedule into a list first.

ing false implies that this composite actor will not be fired again. In that case, the executive director or the manager will call the wrapup() method of the top-level composite actor, which in turn calls the wrapup() method of the director. This causes the director to terminate the execution of the composite actor.

Introduction to Java Threads. The process domains, like the rest of Ptolemy II, are written entirely in Java and take advantage of the features built into the language. In particular, they rely heavily on threads and on monitors for controlling the interaction between threads. In any multi-threaded environment, care has to be taken to ensure that the threads do not interact in unintended ways, and that the model does not deadlock. Note that deadlock in this sense is a bug in the *modeling environment*, which is different from the deadlock talked about before which may or may not be a bug in the model being executed.

A monitor is a mechanism for ensuring mutual exclusion between threads. In particular if a thread has a particular monitor, acquired in order to execute some code, then no other thread can simultaneously have that monitor. If another thread tries to acquire that monitor, it stalls until the monitor becomes available. A monitor is also called a *lock*, and one is associated with every object in Java.

Code that is associated with a lock is defined by the *synchronized* keyword. This keyword can either be in the signature of a method, in which case the entire method body is associated with that lock, or it can be used in the body of a method using the syntax:

```
synchronized(object) {
   ... //Part of code that requires exclusive lock on object
}
```

This causes the code inside the brackets to be associated with the lock belonging to the specified object. In either case, when a thread tries to execute code controlled by a lock, it must either acquire the lock or stall until the lock becomes available. If a thread stalls when it already has some locks, those locks are not released, so any other threads waiting on those locks cannot proceed. This can lead to deadlock when all threads are stalled waiting to acquire some lock they need.

A thread can voluntarily relinquish a lock when stalling by calling *object.*wait() where *object* is the object to relinquish and wait on. This causes the lock to become available to other threads. A thread can also wake up any threads waiting on a lock associated with an object by calling notifyAll() on the object. Note that to issue a notifyAll() on an object it is necessary to own the lock associated with that object first. By careful use of these methods it is possible to ensure that threads only interact in intended ways and that deadlock does not occur.

Approaches to locking used in the process domains. One of the key coding patterns followed is to wrap each wait() call in a while loop that checks some flag. Only when the flag is set to false can the thread proceed beyond that point. Thus the code will often look like

```
synchronized(object) {
   ...
   while(flag) {
      object.wait();
   }
   ...
}
```

The advantage to this is that it is not necessary to worry about what other thread issued the notifyAll() on the lock; the thread can only continue when the notifyAll() is issued *and* the flag has been set to false.

One place that contention between threads often occurs is when a thread tries to acquire another lock only to issue a notifyAll() on it. To reduce the contention, it often easiest if the notifyAll() is issued from a new thread which has no locks that could be held if it stalls. This is often used in the CSP domain to wake up any threads waiting on receivers after a pause or when terminating the model. The `ptolemy.actor.process.NotifyThread` class can be used for this purpose. This class takes a list of objects in a linked list, or a single object, and issues a notifyAll() on each of the objects from within a new thread.

Synchronization Hierarchy. Previously we have discussed how model deadlock is resolved in process domains. Separate from these notions is a different kind of deadlock that can occur in a modeling environment if the environment is not designed properly. This notion of deadlock can occur if a system is not *thread safe*. Given the extensive use of Java threads throughout Ptolemy II, great care has been taken to ensure thread safety; we want no *bugs* to exist that might lead to deadlock based on the structure of the Ptolemy II modeling environment. Ptolemy II uses monitors to guarantee thread safety. A *monitor* is a method for ensuring mutual exclusion between threads that both have access to a given portion of code. To ensure mutual exclusion, threads must acquire a monitor (or *lock*) in order to access a given portion of code. While a thread owns a lock, no other threads can access the corresponding code.

There are several objects that serve as locks in Ptolemy II. In the process domains, there are four primary objects upon which locking occurs: Workspace, ProcessReceiver, ProcessDirector and AtomicActor. The danger of having multiple locks is that separate threads can acquire the locks in competing orders and this can lead to deadlock. A simple illustration is shown in figure 2.19. Assume that both lock *A* and lock *B* are necessary to perform a given set of operations and that both thread 1 and thread 2 want to perform the operations. If thread 1 acquires *A* and then attempts to acquire *B* while thread 2 does the reverse, then deadlock can occur.

There are several ways to avoid the above problem. One technique is to combine locks so that large sets of operations become atomic. Unfortunately this approach is in direct conflict with the whole purpose behind multi-threading. As larger and larger sets of operations utilize a single lock, the limit of the corresponding concurrent program is a sequential program!

Another approach is to adhere to a hierarchy of locks. A hierarchy of locks is an agreed upon order in which locks are acquired. In the above case, it may be enforced that lock *A* is always acquired before lock *B*. A hierarchy of locks will guarantee thread safety [74].

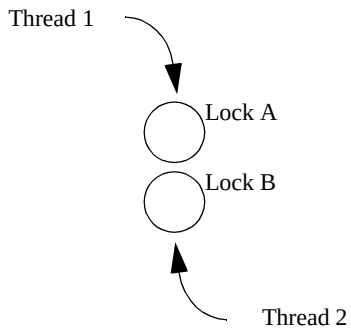


FIGURE 2.19. Deadlock Due to Unordered Locking.

The process domains have an unenforced hierarchy of locks. It is strongly suggested that users of Ptolemy II process domains adhere to this suggested locking hierarchy. The hierarchy specifies that locks be acquired in the following order:

Workspace ⟶ **ProcessReceiver** ⟶ **ProcessDirector** ⟶ **AtomicActor**

The way to apply this rule is to prevent synchronized code in any of the above objects from making a call to code that is to the left of the object in question.

There is one further rule that implementors of process domains should be aware of. A thread should give up all the read permissions on the workspace before calling the wait() method on the receiver object. This commonly happens in the get() and put() methods of process receivers, which implement the synchronization between threads. We require this because of the explicit modeling of mutual exclusion between the read and write activities on the workspace. If a thread holds read permission on the workspace and suspends while a second thread requires a write access on the workspace before performing the action that the first thread is waiting for, a deadlock results. Furthermore, a thread must also regain those read accesses after returning from the call to the wait() method. For this a wait(Object object) method is provided in the class Workspace that releases read accesses on the workspace, calls wait() on the argument object, and regains read access on the workspace before returning.

3

Data Package

Authors: *Rowland R. Johnson*
Bart Kienhuis
Edward A. Lee
Xiaojun Liu
Steve Neuendorffer
Neil Smyth
Yuhong Xiong

3.1 Introduction

The data package provides data encapsulation, polymorphism, parameter handling, an expression language, and a type system. Figure 3.1 shows the key classes in the main package (subpackages will be discussed later).

3.2 Data Encapsulation

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

Token classes are provided for encapsulating many different types of data, such as integers (IntToken), double precision floating point numbers (DoubleToken), and complex numbers (ComplexToken). A special Token subclass (EventToken) exists for representing the presence of a “pure event” that encapsulates no data. Tokens can encapsulate data structures of arbitrary size. All data tokens share several properties, including immutability, type-polymorphic operations, and the possibility for automatic type conversions. These properties will be described in later sections.

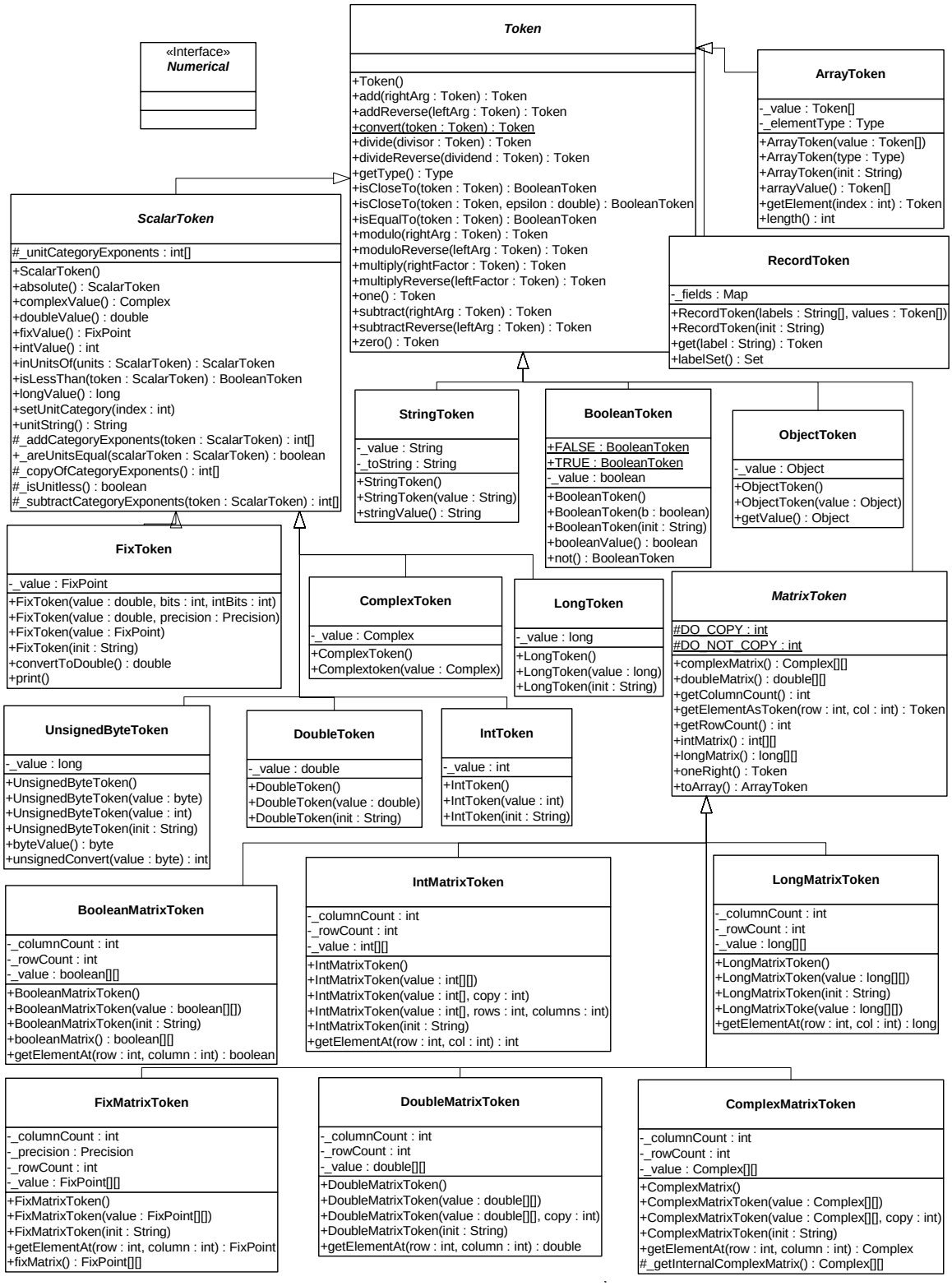


FIGURE 3.1. Static Structure Diagram (Class Diagram) for the classes in the data package.

3.2.1 Matrix data types

The MatrixToken base class provides basic structure for two-dimensional arrays of data. Various derived classes encapsulate data of different types, such as integers, and complex numbers. Standard matrix-matrix and scalar-matrix operations are defined.

3.2.2 Array and Record data types

An ArrayToken is a token that contains an array of tokens. All the element tokens must have the same type, but that type is arbitrary. For instance, it is possible to constructs arrays of arrays of any type of token. The ArrayToken class differs from the various MatrixToken classes in that MatrixTokens contain only be constructed for primitive data, such as int or double, while an array can be constructed for arbitrary token types. In other words, matrix tokens are specialized for storing two dimensional structures of primitive data, while array tokens offer more flexibility in type specifications.

A RecordToken contains a set of labeled values, and operates similarly to `struct` in the C language. The values can be arbitrary tokens and are not required to have the same type.

3.2.3 Fixed Point Data Type

The FixToken class encapsulates fixed point data. The UML diagram showing classes involved in the definition of the FixPoint data type is shown in Figure 3.2. The FixToken class encapsulates an instance of the FixPoint class in the math package. The underlying FixPoint class is implemented using Java's BigInteger class to represent fixed point values. The advantage of using the BigInteger package is that it makes this FixPoint implementation truly platform independent and furthermore, it doesn't put any restrictions on the maximal number of bits allowed to represent a value.

The precision of a FixPoint data type is represented by the Precision class. This class does the parsing and validation of the various specification styles we want to support. It stores a precision into two separate integers. One number represents the number of integer bits, and the other number represents the number of fractional bits. For convenience, the precision of fixed point data can be specified in two different ways:

(*m*/*n*): The total precision of the output is m bits, with the integer part having n bits. The fractional part thus has $m - n$ bits.

(*m*.*n*): The total precision of the output is $n + m$ bits, with the integer part having *m* bits, and the fractional part having *n* bits.

The Quantization class represents various quantization techniques. Creating a FixPoint value requires specifying a double value and an instance of the Quantization class. For convenience, static methods are provided in the Quantizer class that create FixPoint instances without referencing a Quantization explicitly. During conversion, the handling of overflow and underflow is handled by specifying instances of the Overflow class.

The convertToDouble() method in the FixToken class converts a fixed point value into a double representation. Note that the getDouble() method defined by Token is not used since conversion from a FixPoint to a double is not, in general, a lossless conversion and is hence not allowed automatically. For details about how to represent Fixed Point numbers in the expression language, see volume 1, the Expressions chapter.

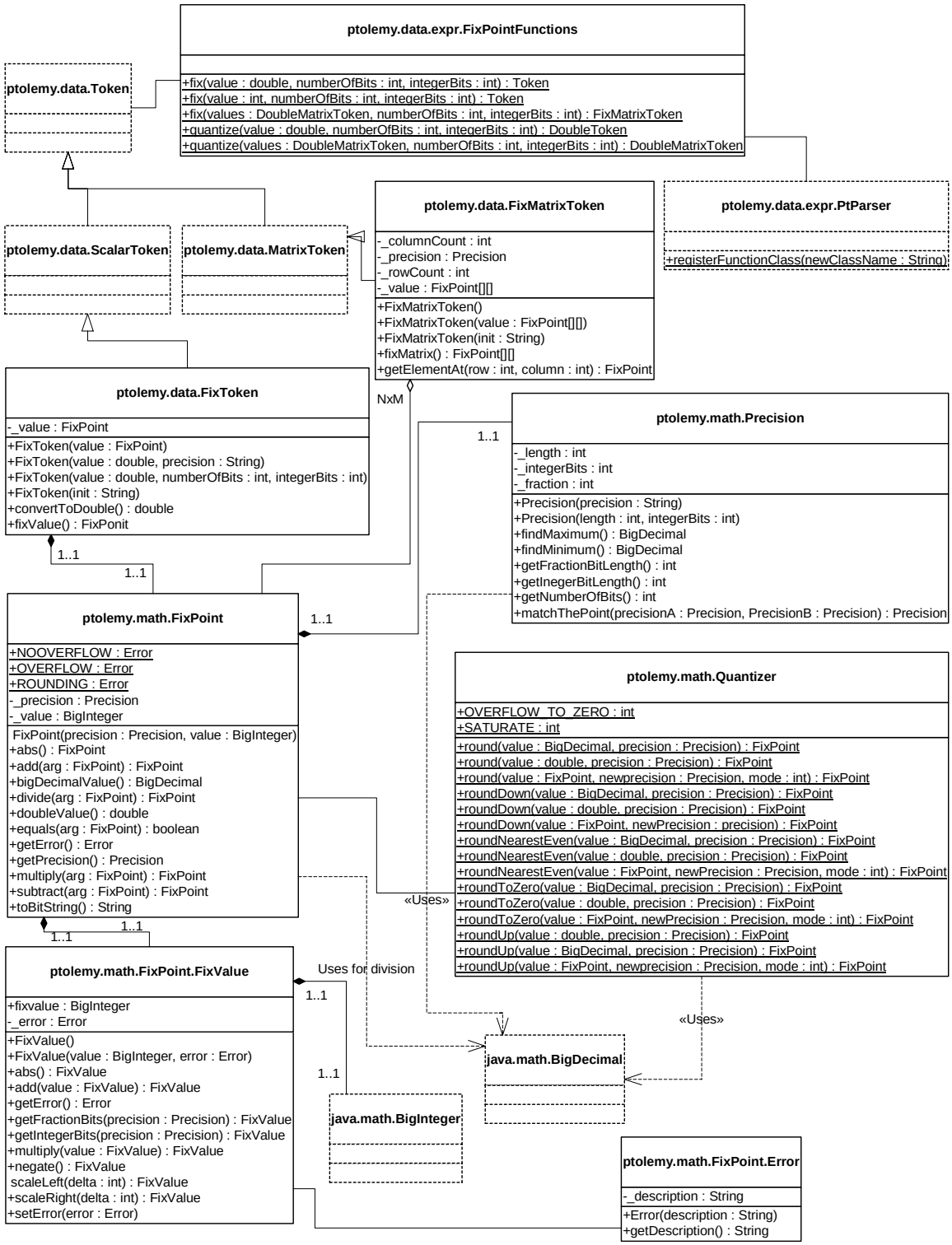


FIGURE 3.2. Organization of the FixPoint Data Type.

3.2.4 Function Closures

The FunctionToken class encapsulates functions that can be evaluated. These function closures can be passed as messages just like any other tokens. When a function closure is created, all identifiers that are not arguments to the function are evaluated. The arguments to the function, however are only evaluated when the function is applied. For information on how functions closures can be represented in the expression language, see volume 1.

3.2.5 Nil Tokens

Null or missing tokens are common in analytical systems like R and SAS where they are used to handle sparsely populated data sources. In database parlance, missing tokens are sometimes called null tokens. Since null is a Java keyword, we use the term "nil". Nil tokens are useful for analyzing real world data such as temperature where the value was not measured during every interval. In principle, an as yet unimplemented function such as average() could properly handle nil tokens - when the average() method sees a nil token, it should be ignored. Note that this can lead to uncertainty. For example, if average() is expecting 30 values and 29 of them are nil, then the average will not be very accurate.

If an operation such as add(), divide(), modulo(), multiply(), one(), subtract(), zero() or their corresponding “reverse” operations includes a nil token, then the output is nil. If one of the arguments for isCloseTo() or isEqualTo() is nil, then the method returns false. Methods that return a nil token return a nil token with a specific type so that type safety is preserved. The following tokens have NIL values defined: ArrayType, BooleanToken, ComplexToken, DoubleToken, IntToken, LongToken, StringToken, Token, UnsignedByteToken. There is no nil token for the various matrix tokens because the underlying matrices are java native type matrices that do not support nil.

3.3 Immutability

Tokens in Ptolemy II are, in general, immutable. This means that a token’s value cannot be changed after the token is constructed. The value of a token must be specified by constructor arguments, and there is no other mechanism for setting the value. If a token encapsulating another value is required, a new instance of Token must be constructed.

There are several reasons for making tokens immutable.

- First, when a token is sent to several receivers, we want to be sure that all receivers get the same data. Each receiver is sent a reference to the same token. If the Token were not immutable, then it would be necessary to clone the token for all receivers after the first one.
- Second, since a token is passed between two actors, they may both have a reference to the token. If the token were mutable, then the token would represent shared state of the two actors, requiring synchronization and limiting the ability to represent distributed computation. Immutable tokens passed between actors ensures that the concurrency of actors is determined solely by a model of computation.
- Third, we use tokens to parameterize actors, and parameters often have mutual dependencies. That is, the value of a parameter may depend on the value of other parameters. The value of a parameter is represented by an instance of Token. If that token were allowed to change value without notifying the parameter, then the parameter would not be able to notify other parameters that depend on its value. Thus, a mutable token would have to implement a publish-and-subscribe mechanism so

that parameters could subscribe and thus be notified of any changes. By making tokens immutable, we greatly simplify the design.

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

In most cases, the immutability of tokens is enforced by the design of the Token subclasses. One exception is the ObjectToken class. An ObjectToken contains a reference to an arbitrary Java object created by the user, and a reference to this object can be retrieve through the getValue() method. Since the user may modify the object after the token is constructed, the immutability of an ObjectToken is difficult to ensure. Although it could be possible to clone the object in the ObjectToken constructor and return another clone in the getValue() method, this would require the object to be cloneable, severely limiting the use of the ObjectToken. In addition, since the default implementation of clone() only makes a shallow copy, this approach is not able to enforce immutability on all cloneable objects. Cloning a large object could be prohibitively expensive. For these reasons, the ObjectToken does not attempt to enforce immutability, but rather relies on the cooperation from the user. Violating this convention could lead to unintended non-determinism.

For matrix tokens, enforced immutability requires the contained matrix (Java array) to be copied when the token is constructed and when the matrix is returned in response to queries such as intMatrix(), doubleMatrix(), etc. Since the cost of copying large arrays is non-trivial, the user should not make more queries than necessary. For optimization, some matrix token classes have a constructor that takes a flag, which specifies whether the given array needs to be copied or not. The getElementAt() method can be used to read the contents of the matrix without copying the internal array.

3.4 Polymorphism

3.4.1 Polymorphic Arithmetic Operators

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

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

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

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

then

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

gives a new DoubleToken with value 7.2,

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

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

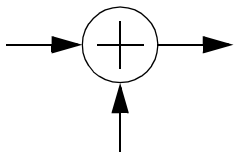
```
a.modulo(c)
```

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

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

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

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



might contain code like:

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

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

3.4.2 Automatic Type Conversion

For the above arithmetic operations, if the two tokens being operated on have different types, type conversion is needed. Generally speaking, Ptolemy II automatically performs conversions that do not lose numerical precision. Other conversion must be explicitly represented by the user. The admissible automatic type conversions between different token types are modeled as a partially ordered set called the *type lattice*, shown in figure 3.3. In that diagram, type *A* is *greater than* type *B* if there is a path upwards from *B* to *A*. Thus, [*complex*] (a complex matrix) is greater than *int*. Type *A* is *less than* type *B* if there is a path downwards from *B* to *A*. Thus, *int* is less than [*complex*]. Otherwise, types *A* and *B* are *incomparable*. Types *complex* and *long*, for example, are incomparable. In the type lattice, a type can be automatically converted to any type greater than it.

This hierarchy is realized by the TypeLattice class in the data.type subpackage. Each node in the lattice is an instance of the Type interface. The TypeLattice class provides methods to compare two

token types.

Two of the types, *matrix* and *scalar*, are union types. This means that an instance of this type can be any of the types immediately below them in the lattice. Recall that arrays always have entries with the same type. If that type is *matrix* or *scalar*, then the array may appear to have multiple types. For example, in the expression evaluator,

```
>> {1, 2.3}
{1.0, 2.3}
>> {1, 2.3, true}
{1, 2.3, true}
```

In the first case, the least common type is *double*, so the elements are all converted to *double*. In the second case, the least common element is *scalar*, so the elements are all converted to scalar, which in this example results in no conversion! To see that the type of the array is {*scalar*}, do this,

```
>> {1, 2.3, true}.getType()
```

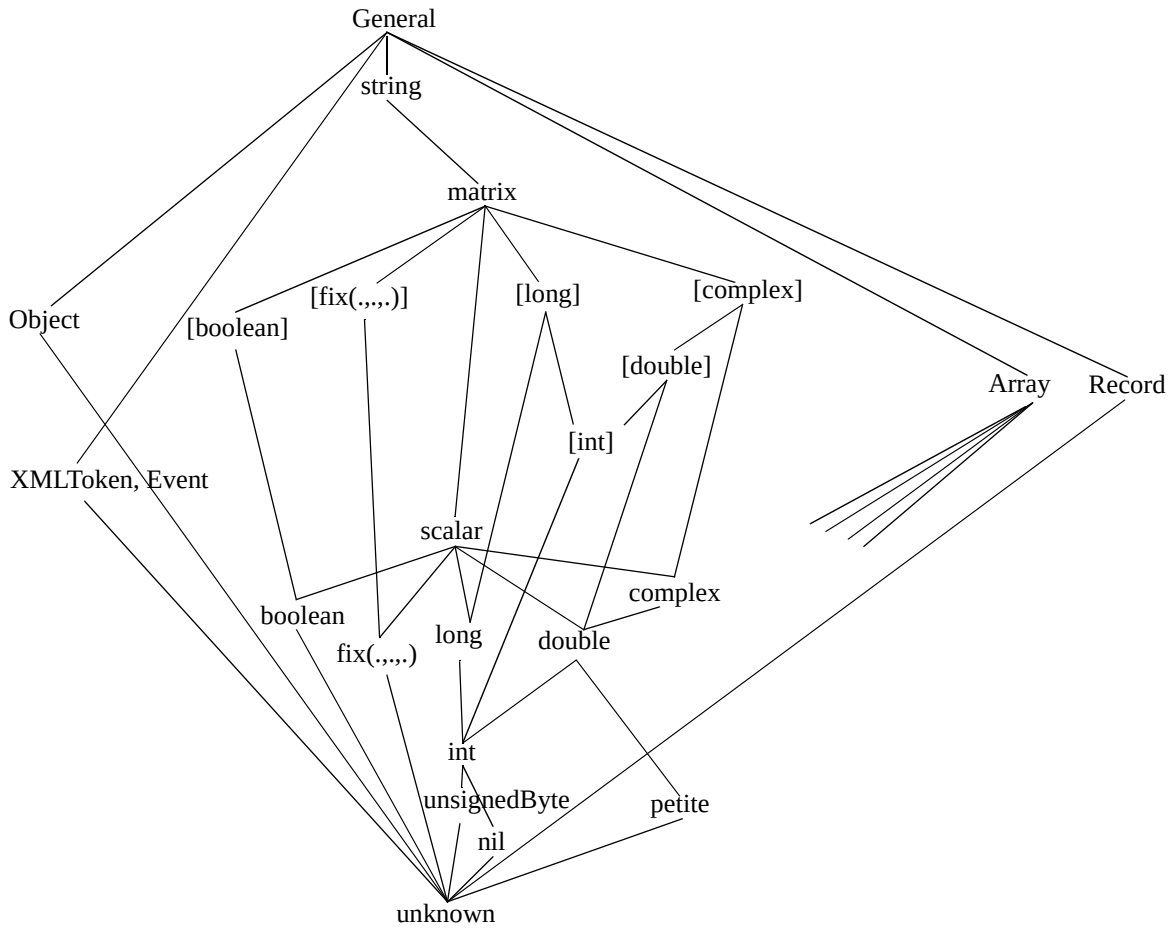


FIGURE 3.3. The type lattice.

```
object({scalar})
```

(The return value of getType() is not a Token, so the expression evaluator wraps it in an ObjectToken, which displays its value as "object(*toString*())".)

Similarly, for matrices,

```
>> {[int], [double]}
{[0.0], [0.0]}
>> {[long], [double]}
{[0L], [0.0]}
>> {[long], [double]}.getType()
object({matrix})
```

Type conversion is done by the convert() method in type classes. This method converts the argument into a token with the same type. For example, BaseType.DoubleType.convert(Token token) converts the specified token into an instance of DoubleToken. The convert() method can convert any token immediately below it in the type hierarchy into an instance of its own class. If the argument is higher in the type hierarchy, or is incomparable with its own class, the convert() method throws an exception. If the argument to convert() already has the correct type, it is returned without any change. Many of the simpler token classes also provide a static convert() method that can be used more simply than the convert() method of the corresponding type.

Most implementations of the add(), subtract(), multiply(), divide(), modulo(), and equals() methods require that the type of the argument and the implementing class be comparable in the type hierarchy. If this condition is not met, these methods will throw an exception. If the type of the argument is lower than the type of the implementing class, then the argument is usually converted to the type of the implementing class before the operation is carried out. One exception is the implementation of these methods for matrix tokens. To allow matrices to be multiplied and divided by scalars, the normal conversion is not performed. The MatrixToken base class deals specially with scalar-matrix operations.

To allow this, the implementation of most operations is somewhat more complicated if the type of the method argument is higher than the implementing class. In this case, we assume the operation is implemented in the class that has the higher type (the matrix token in the above example). Since token operations need not be commutative, for example, `"Hello" + "world"` is not the same as `"world" + "Hello"`, and 3-2 is not the same as 2-3, the implementation of arithmetic operations cannot simply call the same method on the class of the argument. Instead, a separate set of methods is provided, which perform token operations in the reverse order. These methods are addReverse(), subtractReverse(), multiplyReverse(), divideReverse(), and moduloReverse(). The equality check is always commutative so no equalsReverse() is needed. Under this setup, a.add(b) means $a+b$, and a.addReverse(b) means $b+a$, where *a* and *b* are both tokens. If, for example, when a.add(b) is invoked and the type of *b* is higher than *a*, the add() method of *a* will automatically call b.addReverse(a) to carry out the addition.

For scalar and matrix tokens, methods are also provided to convert the content of the token into another numeric type. In the ScalarToken base class, these methods are intValue(), longValue(), doubleValue(), fixValue(), and complexValue(). In the MatrixToken base class, the methods are intMatrix(), longMatrix(), doubleMatrix(), fixMatrix(), and complexMatrix(). The default implementation in these two base classes simply throws an exception. Derived classes override these methods according to the automatic type conversion relation of the type lattice. For example, the IntToken class overrides

all the methods defined in ScalarToken, but the DoubleToken class does not override the intValue() method, since automatic conversion is not allowed from a double to an integer.

3.5 Variables and Parameters

In Ptolemy II, any instance of NamedObj can have attributes, which are instances of the Attribute class. A *variable* is an attribute that contains a token. Its value can be specified by an expression that can refer to other variables. A *parameter*, implemented by the Parameter class, is in most ways functionally identical to a variable, but also appears modifiable from the user interface. See figure 3.4 and figure 3.5. The presence of these two separate classes allows variables to exist which are internal to an actor, and not visible to an end user. For the rest of this section we consider parameters and variables to be largely interchangeable.

3.5.1 Values

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

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

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

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

3.5.2 Types

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

- You can require the variable to have a specific type. Use the setTypeEquals() method.

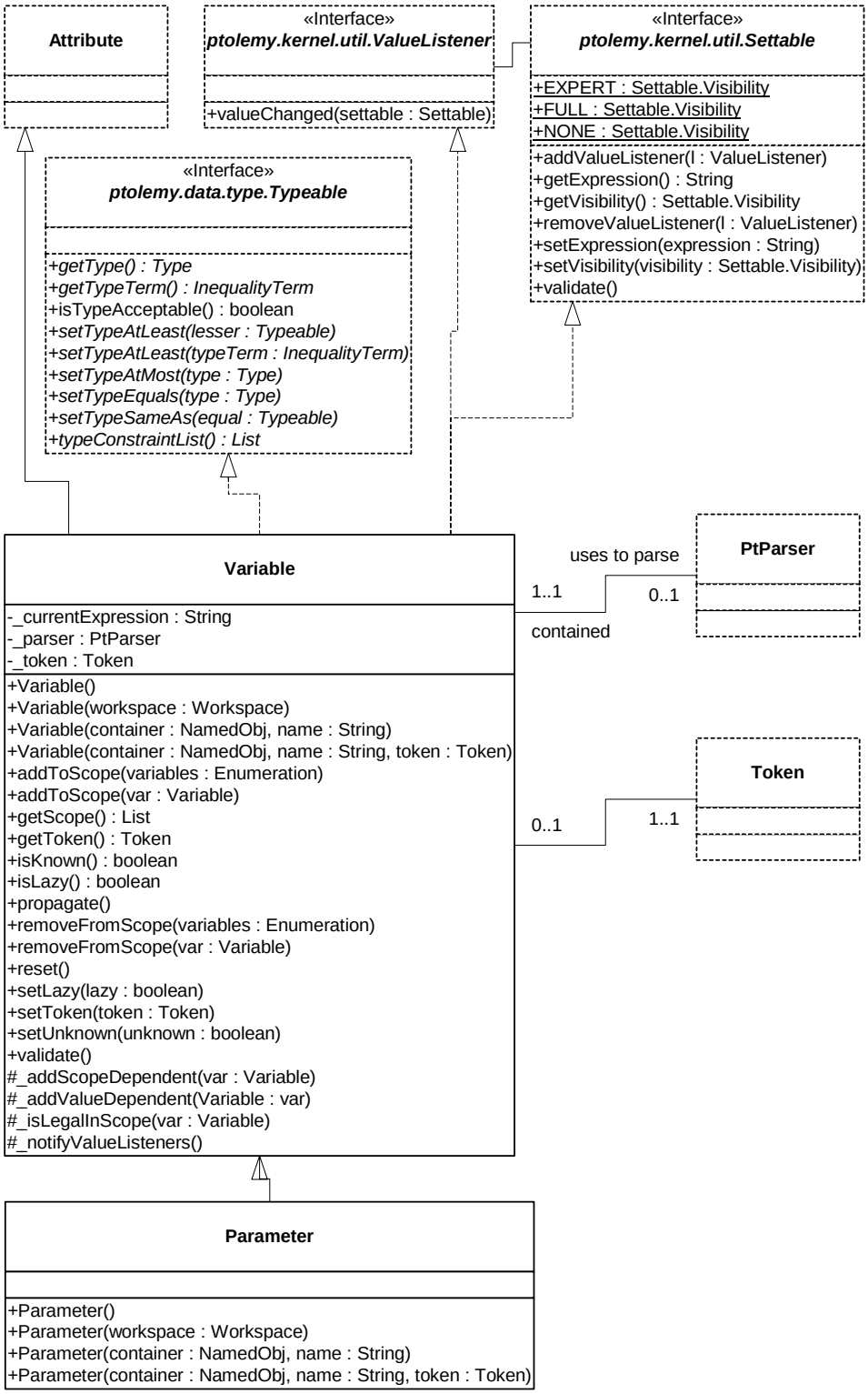


FIGURE 3.4. Static structure diagram for the Variable and Parameter classes in the data.expr package.

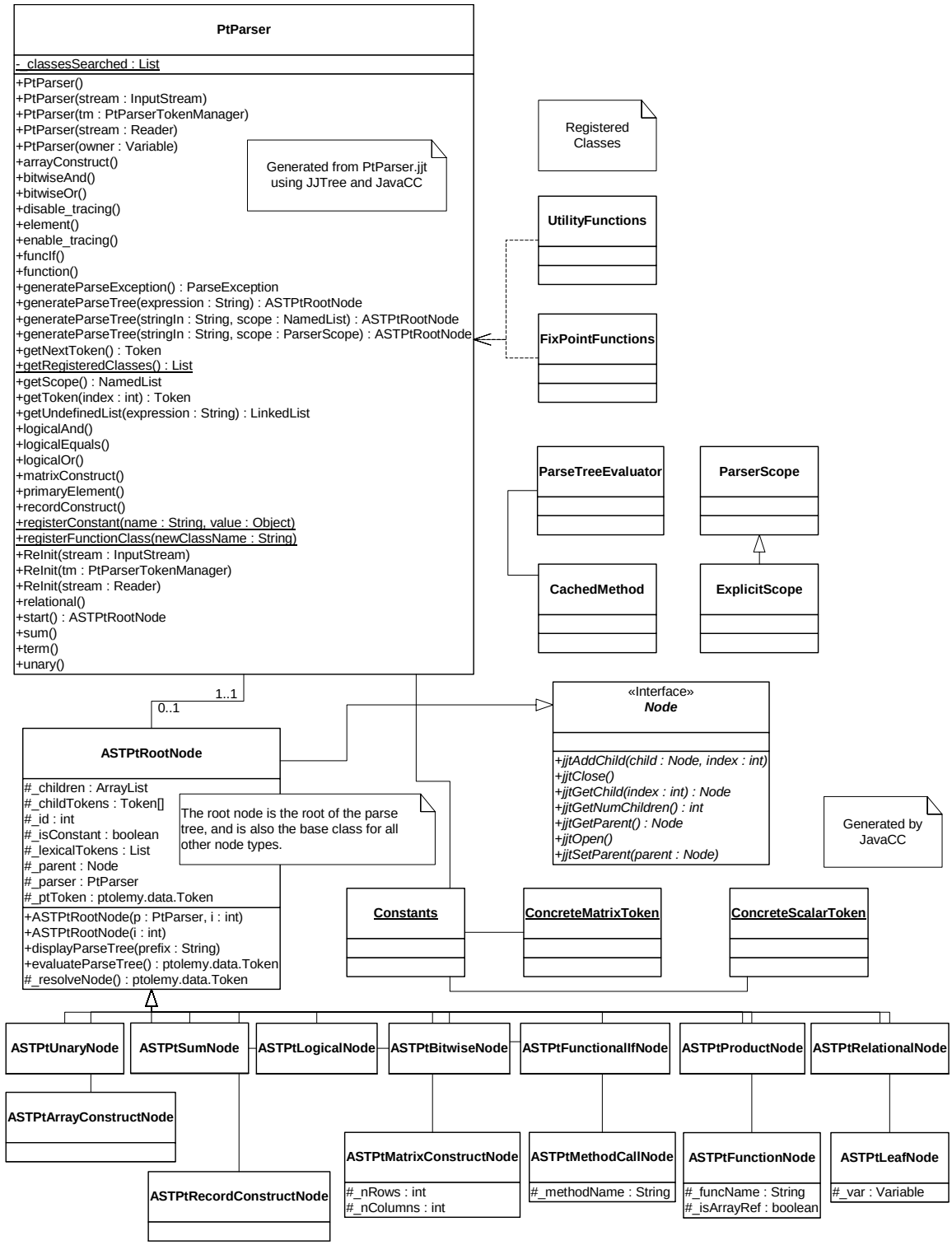


FIGURE 3.5. Static structure diagram for the parser classes in the data.expr package

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

Except for the first type constraint, these are not checked by the Variable class. They must be checked by a type resolution algorithm, which is executed before the model runs and after parameter values change.

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

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

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

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

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

If the type of a variable is changed after having once been set, the container is notified of this by calling its attributeTypeChanged() method. If the container does not allow type changes, it should throw

an exception in this method. If the value is changed after having once been set, then the container is notified of this by calling its attributeChanged() method. If the new value is unacceptable to the container, it should throw an exception. The old value will be restored.

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

3.5.3 Dependencies

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

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

3.6 Expressions

Ptolemy II includes a extensible expression language. This language permits operations on tokens to be specified in a scripting fashion, without requiring compilation of Java code. The language was designed to be extremely succinct, using overloaded operators instead of verbose references to methods in the token classes.[1] The expression language can be used to define parameters in terms of other parameters, for example. It is also used to provide end-users with the ability to describe simple stateless actors without resorting to writing Java code through the Expression actor. The expression language is also used to give guards and resets for finite state machines in an intuitive fashion. The use of the expression language is described in volume 1.

The expression language is extensible. The extension mechanism is based on the reflection package in Java used to add primitive functions and constants to the expression language. The expression language is also purely functional, meaning that it lacks sequencing constructs and side effects. Building state and sequencing into models is done through the use of models of computation, allowing a much richer set of concurrent control structures than is possible with traditional imperative languages. The language is higher-order, since it is integrated with the the FunctionToken class. This allows for new functions to be easily declared as part of a model, using expressions and for these expressions to be manipulated and passed through a model as data. Because the expression language is side-effect free this mechanism does not interact in unexpected ways with concurrent models of computation.

1. The Ptolemy II expression language uses operator overloading, unlike Java. Although we fully agree that the designers of Java made a good decision in omitting operator overloading, our expression language is used in situations where compactness of expressions is extremely important. Expressions often appear in crowded dialog boxes in the user interface, so we cannot afford the luxury of replacing operators with method calls. It is more compact to say “2*(PI + 2i)” rather than “2.multiply(PI.add(2i)),” although both will work in the expression language.

Lastly, the expression language is strongly typed, allowing transparent integration with the static type checking of components specified using expression. When combined with the higher-order constructs the resulting language has the feel of typed lambda calculus.

3.7 Unit System

The unit system in Ptolemy II is based on the paper “Automatic Units Tracking” by Christopher Rettig [132]. The basic idea is to define a suite of parameters to represent the various measurement units of a unit system, such as “meter,” “cm,” “feet,” “miles,” “seconds,” “hours,” and “days.” In each unit category (“length” or “time” for example), there is a base unit with respect to which all the others are specified. If the base unit of length is meters, then “cm” (centimeter) will be specified as “0.01 * meters”. Derived units are specified by just multiplying and dividing base units. For example “newton” is specified as “meter * kilogram / second^2”.

The unit parameters contain tokens just like other parameters. To track units, the category information is stored together with measurement data in scalar tokens, and is used when arithmetic operations, such as add() and multiply(), are performed. The subclasses of ScalarToken, including IntToken and DoubleToken, override these methods to perform unit checking.

The ptolemy.data.unit package provides three classes (BaseUnit, UnitCategory, and UnitSystem) that allow a unit system to be specified using MoML, as illustrated in figure 3.6. When such a unit system is added to the model shown in figure 3.7, the units can be used in expressions to specify the value of actor parameters. The displayed result of executing the model is “10.0 * m / s”.

Several basic unit systems are provided with Ptolemy II. In the Vergil graph editor, they appear in the utilities library. A unit system added to a composite actor can only be used inside that actor. The

```
<property name="Sample" class="ptolemy.data.unit.UnitSystem">
  <property name="m" class="ptolemy.data.unit.BaseUnit" value="1.0">
    <property name="Length" class="ptolemy.data.unit.UnitCategory"/>
  </property>
  <property name="cm" class="ptolemy.data.expr.Parameter" value="0.01*m"/>
  <property name="s" class="ptolemy.data.unit.BaseUnit" value="1.0">
    <property name="Time" class="ptolemy.data.unit.UnitCategory"/>
  </property>
  <property name="ms" class="ptolemy.data.expr.Parameter" value="0.001*s"/>
</property>
```

FIGURE 3.6. A sample unit system.

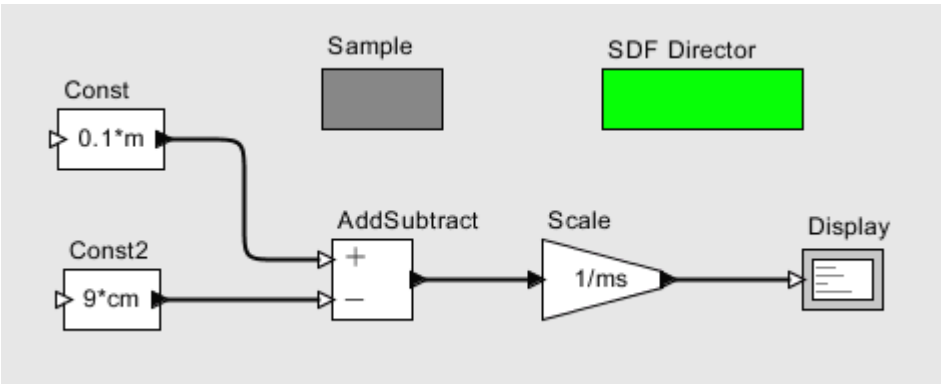


FIGURE 3.7. A model that uses the sample unit system.

user can customize a unit system by adding units, or create new unit systems based on those provided.

The current implementation of unit systems has the following limitations:

- Only scalar values can have units.
- The result of calling a function on a value with units is unit-less.

3.8 The Static Unit System

This section presents the static units system in Ptolemy. In contrast to the unit system in the previous section, which is dynamic, the purpose of the static unit system is to analyze a model before it is run. In particular, the static units system is used to analyze the structure of the model and determine if it is correct in terms of the units of measure.

In the dynamic unit systems units of measure is an integral part of a data value, i.e. it is encapsulated as part of the data Token. In contrast, in the static unit system information about units of measure is in the form of specifications attached to ports. This information can be interpreted as a implying that any data Token that passes through the port at run-time will have those unit specifications. That is, it is a constraint that must be met when the model is run. If the static unit system can determine that constraints of a model will be met at run-time then the model is said to be *units consistent*.

As an example, consider part of the model that is used in the StaticUnits demonstration and is

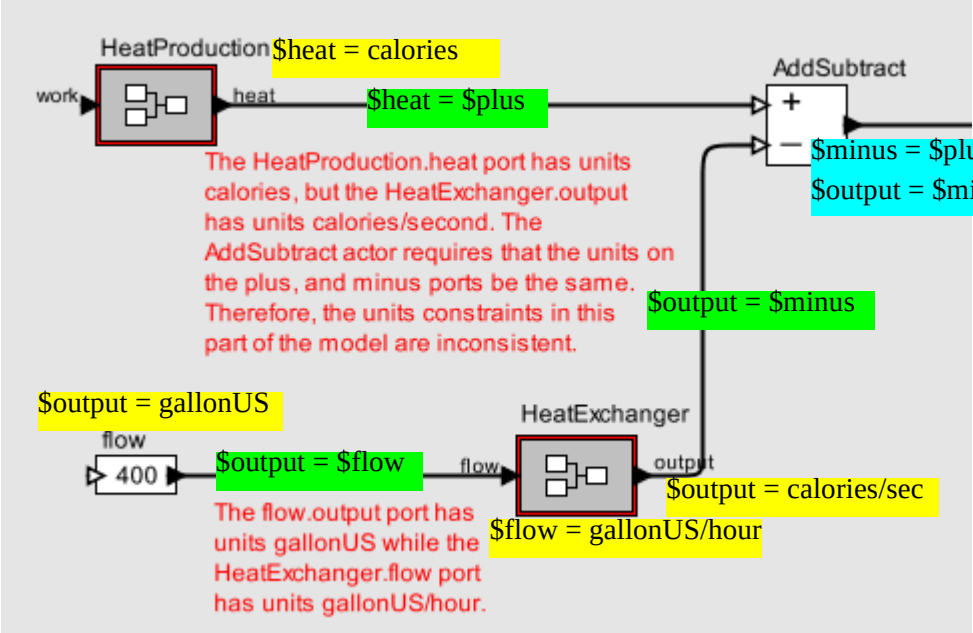


FIGURE 3.8. Part of the StaticUnits demonstration. The colored boxes indicate the unit constraints in this part of the model.

shown in Figure 3.8. There are several unit constraints shown here. Each constraint is in the form of an equation where variables begin with a “$” and refer to ports in the model. The constraints with green background specify that a port will pass data with specific units. For example, the equation “$heat = calories” next to the HeatProduction actor indicates that any data passing through the heat port will be in units of calories. The constraints with cyan background specifies the relationship among ports on a

particular actor. In this case the AddSubtract actor requires that the plus, minus, and out ports all have the same units without specifying what those units will be.The constraints with the yellow background exist when ports on different actors are connected via a relation.

It can be seen that equations of the model in Figure 3.8 are inconsistent. For example, the equations

```
$output = gallonUS
$output = $flow
$flow = gallonUS/hour
```

can not all be true.

In a real sense, static unit specifications are an extension of conventional data types. For example, a datum may be of type double, but, in addition, also be known to represent calories/sec. It is tempting to extend the analytic techniques applicable for conventional data types to the realm of units of measure. However, conventional data types analytic techniques do not appear to be effective in dealing with static unit specifications. Lattices defined on data type inequalities is the basis for powerful techniques for analyzing data types. Although type lattices can be defined for units specifications they seem trivial, and have not lead to any useful techniques. In contrast, the relationship amongst the units specifications of a model are best characterized with a set of equations. This approach is the basis for determining if a model is units consistent.

A strategic goal in the design and implementation was, both, to leave the dynamic unit capability intact, and to re-use the dynamic unit components where possible.

3.8.1 Unit Systems

A unit system is based on 1) an ordered set $\{D_1,D_2\ldots,D_n\}$ where each $\{D_i\}$ represents a dimension, and 2) a base unit for each dimension. The intent is that each dimension is orthogonal to all other dimensions, and that any unit of measure can be expressed as a combination of the base units. An example of a unit system in widespread use is the International System of Units shown in Table 9

Index	Dimension	Base Unit
1	Length	Meter
2	Time	Second
3	Temperature	Kelvin
4	Mass	Kilogram
5	Current	Ampere
6	Substance	Mole
7	Luminosity	Candela

Table 9: System International Unit System

In order to simplify the presentation of examples in this section the Simple unit system shown in

Index	Dimension	Base Unit
1	Length	Meter

Table 10: Simple Unit System

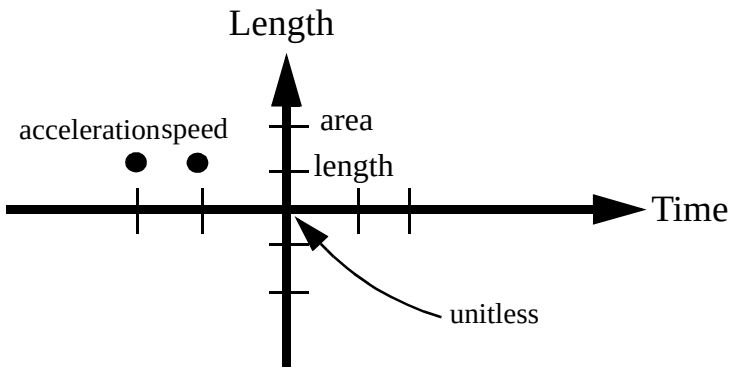
Index	Dimension	Base Unit
2	Time	Second
3	Mass	Kilogram

Table 10: Simple Unit System

Table 10 will be used throughout.

3.8.2 Units of Measurement Algebra

The *type* of a unit is expressed as $\langle e_1,\ldots,e_n\rangle$ where each e_i represents the exponent of the corresponding dimension. For example, $\langle 1, -1, 0\rangle$ (i.e. Length/Time) is a type in the Simple unit system, and is commonly referred to as speed. The set of types for a unit system are a set of points in N-dimensional space. For example, a unit system with just the categories Length and Time (i.e. without Mass) would look like



A type is said to be *singular* if it has the property that just one of the exponents is 1 and the rest are 0. The type $\langle 0, \ldots, 0\rangle$ is said to be the *unitless* type.

A *unit* is obtained by combining a scale with a type. A unit U is expressed as

$$U = \alpha\langle e_1,\ldots,e_n\rangle \qquad \text{where} \qquad \alpha \in \text{Reals} \tag{1}$$

where α is the scale of U. Please note that (1) does not imply the multiplication of $\langle e_1,\ldots,e_n\rangle$ by α. In particular, $\alpha\langle e_1,\ldots,e_n\rangle \neq \langle \alpha e_1,\ldots,\alpha e_n\rangle$ A unit is said to be *singular* if its type is singular. A unit is said to be *basic* if $\alpha = 1.0$.

Refer to Table 11 for some examples of units.

Unit	Descriptive Form(s)	Scale	Type	Properties
1.0<1, 0, 0>	Meter, m	1.0	<1, 0, 0>	basic, singular
1.0<0, 1, 0>	Sec, s	1.0	<0, 1,0>	basic, singular
1.0<0, 0, 1>	Kilogram, kg	1.0	<0, 0, 1>	basic, singular
0.01<1, 0, 0>	Centimeter, cm	0.01	<1, 0, 0>	singular
3600<0, 1, 0>	Hour, hr	3600	<0, 1, 0>	singular
0.453592<0, 0, 1>	Pound, lb	0.453592	<0, 0, 1>	singular
1.0<1, -1, 0>	meters/sec, m/s	1.0	<1, -1, 0>	basic
0.44704<1, -1, 0>	miles/hour, mph	0.44704	<1, -1, 0>	
9.80665<1, -2, 0>	g (gravity)	9.80665	<1, -2, 0>	
3.14158<0, 0, 0>	pi, π	3.14159	<0, 0, 0>	unitless

Table 11: Examples of Units

The unit *multiplication* of two units is the vector addition of the two types and the "normal" multiplication of the two scales. That is

$$U_1 \times U_2 = a_1 * a_2 \langle e_{1,1} + e_{1,2}, e_{2,1} + e_{2,2} \ldots e_{n,1} + e_{n,2} \rangle \text{ where } \left\{ \begin{array}{l} U_1 = a_1 \langle e_{1,1}, e_{2,1}, \ldots, e_{n,1} \rangle \\ U_2 = a_2 \langle e_{1,2}, e_{2,2}, \ldots, e_{n,2} \rangle \end{array} \right\}$$

Unit *exponentiation* follows from multiplication, i.e.

$$U^x = a^x \langle xe_1, xe_2, \ldots, xe_n \rangle$$

The notion of a unit variable is assumed and has the properties commonly found in computational systems. A unit *expression* has the form

$$U \times X_1^{P_1} \times \ldots \times X_q^{P_q} \text{ where } \left\{ \begin{array}{l} \text{U is a unit} \\ X_i^{P_i} \text{ is the variable } X_i \text{ raised to the } P_i \text{ power} \end{array} \right\}$$

A unit *equation* has the form

$$U_X \times X_1^{P_1} \times \ldots \times X_q^{P_q} = U_Y \times Y_1^{S_1} \times \ldots \times Y_r^{S_r} \text{ where } \left\{ \begin{array}{l} \text{each } X_i, Y_j \text{ is a variable} \\ U_X \text{ and } U_Y \text{ are Units} \end{array} \right\} \quad (2)$$

An equation is said to be in *canonical* form if the left side consists solely of powers of variables and the right side is a unit, i.e.

$$V_1^{P_1} \times \ldots \times V_N^{P_N} = U$$

Any equation in the form of (2) can be translated to an equivalent canonical form by first multiplying both sides of (2) by $U_X^{-1} \times Y_1^{-S_1} \times \ldots \times Y_r^{-S_r}$ yielding

$$X_1^{P_1} \times \ldots \times X_q^{P_q} \times Y_1^{-S_1} \times \ldots \times Y_r^{-S_r} = U_Y \times U_X^{-1} \quad (3)$$

The right side of (3) can be further reduced since U_Y and U_X are both units, i.e. they are not variables. If U is the value of the expression $U_Y \times U_X^{-1}$ then (3) can be rewritten as

$$X_1^{P_1} \times \ldots \times X_q^{P_q} \times Y_1^{-S_1} \times \ldots \times Y_r^{-S_r} = U \quad (4)$$

which has just powers of variables on the left side and a unit on the right side.

3.8.3 Descriptive Form Language

The descriptive form of a unit is a natural language string that humans normally use when referring to a unit of measure. Table 11, above, shows some examples of descriptive forms. In the Static Unit System descriptive forms are realized through a formal language, called Descriptive Form Language, and its associated grammar shown in Figure 3.9

```
Root ::= UnitEquation | UnitExpression
UnitEquation ::= UnitExpr "=" UnitExpr
UnitExpr ::= UnitTerm {"/" UnitTerm | "*" UnitTerm | UnitTerm}*
UnitTerm ::= UnitElement
            |UnitElement "^" <Number>
            |<Number>
            |"(" UnitExpr ")"
UnitElement ::= Unit| Variable
Variable ::= "$" <String>
```

FIGURE 3.9. BNF for Descriptive Form Language

For example, the parse tree for the expression “gallons/second” is shown in figure 3.10

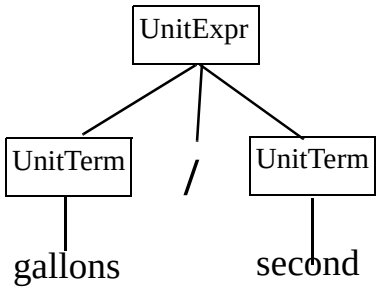


FIGURE 3.10. Parse Tree for “gallons/sec”

The Static Unit System parser, called UParser, is generated using JavaCC which is used to generate the PtParser, the expression parser in Ptolemy. See “Generating the parse tree” on page 4-86. One key difference is that the generation of UParser does not start from a JJTree specification. Instead the DFL grammar is specified in the UParser.jj which is input directly to JavaCC.

If it were possible it would have been advantageous to make DFL a subset of the expression language. However, there are two differences between the DFL and the data expression language that preclude this possibility. First, multiplication in DFL can be expressed via concatenation of the

multiplicands. For example, moment arm can be expressed as “foot pound” in DFL. There is no rule in the data expression grammar that provides for this construction. DFL could be modified so that multiplication requires a “*” operator, as in “foot*pound”(in fact, as a convenience, DFL accepts this construction). However, the form “foot pound” has been in use for decades and it was judged that casual users of Ptolemy would find the requirement of the “*” operator to be awkward.

The second difference stems from the necessity to have variables distinquishable from unit labels.The data expression language does not have a way to explicitly distinguish variables, as it is clear from the context. The example presented in Figure 3.8 shows the descriptive form “$heat = calories” expressing the constraint that the port with name heat has unit calories. Without the “$” to distinguish heat as a port name the parser would try to interpret heat as a unit.

3.8.4 Implementing the Static Unit System in Ptolemy

This subsection presents the internal form that is implemented as a set of classes in Ptolemy. Figure 3.11 presents the UML static structure for these classes.

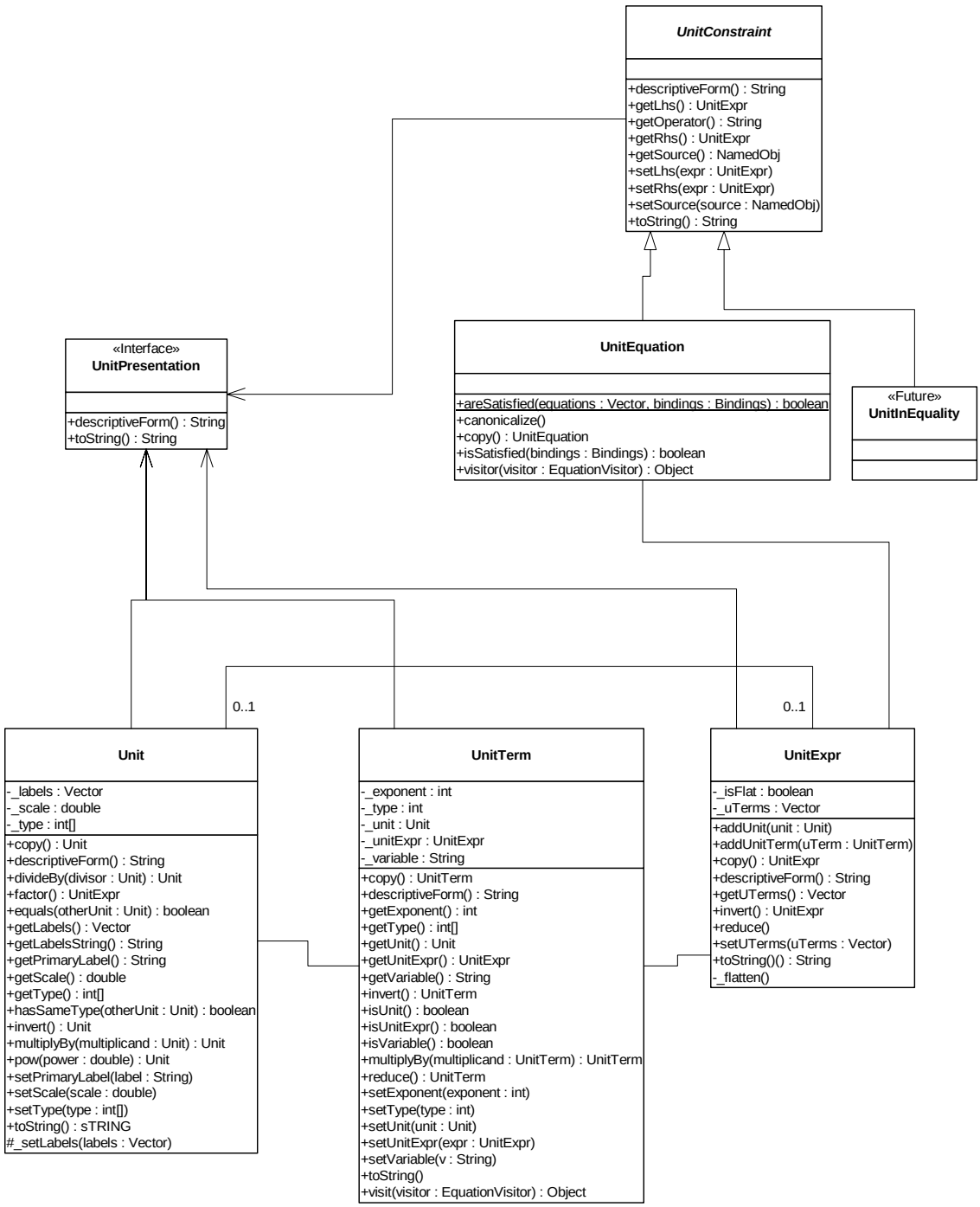


FIGURE 3.11. Static Structure of classes used ti implement units, unit expressions and unit equations.

3.8.5 The Unit Library

The dynamic units system is architected so that a unit system is associated with a model. Further, different models can have different unit systems. In contrast, the static unit systems architecture provides the equivalent functionality with a UnitLibrary. However, there is one common UnitLibrary in the Ptolemy system, and it is used by all models requiring the services of the static unit system. The UnitLibrary is loaded the first time Ptolemy attempts any operations that will require the UnitLibrary be present.

The specifications for a UnitLibrary are contained in a file with the exact same format as is used by the dynamic unit system. (Presently, the name of the file is hardwired to be ptolemy/data/unit/SI.xml.) When loaded by the static unit system a UnitSystem appropriate for the dynamic unit system is created as a side effect. It is the subsequent processing of this UnitSystem that creates the UnitLibrary.

Figures 3.12 and 3.13 show parts of this file

```
 <property name="cm" class="ptolemy.data.unit.BaseUnit" value=”1.0”>
  <property name="Length" class="ptolemy.data.unit.UnitCategory"/>
  </property>

<property name="second" class="ptolemy.data.unit.BaseUnit" value="1.0">
  <property name="Time" class="ptolemy.data.unit.UnitCategory"/>
  </property>
```

FIGURE 3.12. The Length and Time BasicUnits specifications from the SI.xml file

```
<property name="lightSpeed" class="ptolemy.data.expr.Parameter"
            value="299792458.0*meter/second"/>
<property name="gallonUS" class="ptolemy.data.expr.Parameter"
             value="3785.412*cm^3"/>
<property name="pi" class="ptolemy.data.expr.Parameter" value="3.1415"/>
<property name="planckConstant" class="ptolemy.data.expr.Parameter"
             value="hBar*2*pi"/>
```

FIGURE 3.13. The lightSpeed and gallonUS non-BasicUnit specifications from the SI.xml file.

Although the file format is identical for the two unit systems there is one additional requirement that the static unit system imposes. Non-BasicUnits are specified as a Ptolemy Parameter. In essence the dynamic unit system allows external references to units outside the UnitSystem. For example, the Parameter pi is defined elsewhere in the Ptolemy system and any UnitSystem can refer to that definition. In contrast, the static unit system is based on an architecture where the UnitLibrary is self contained. This is necessary in order to distinguish the case where a Parameter is not a unit. Therefore, for example, pi must be specified in the file as shown in figure 3.13.

3.8.6 Generating Descriptive Forms

In order to present the user with the results of the solver it is necessary to generate the descriptive form from the internal representation of a unit. If the unit exists in the UnitLibrary then generation requires only that the descriptive form be retrieved from the UnitLibrary.

However, it is often the case that the unit does not exist in the UnitLibrary. In the example shown in Figure 3.8 there is an inconsistency between the Flow.output port with units gallonUS and the HeatExchanger.flow port with units gallonUS/hour. The following table has the relevant information

	Descriptive Form	Internal Form	Source
1	gallonUS	3785.412<3,0,0>	UnitLibrary
2	hour	3600<0,1,0>	UnitLibrary
3	gallonUS / hour	1.05<3, -1,0>	derived by parsing the descriptive form
4	to be generated	.000277<0,-1,0>	derived by solving for X in the equation $378.412\langle 3,0,0\rangle \times \mathrm{X} = 1.05\langle 3,-1,0\rangle$

in determining the descriptive form of the transformation required to remove this inconsistency.

The internal form in lines 1 and 2 are values obtained from the UnitLibrary. The internal form in line 3 was obtained by parsing the descriptive form “gallonUS/hour” which causes the calculation 3785.412<3,0,0> / 3600<0,10> = 1.05<3,-1,0> to take place. The internal form in line 4 is the result of solving for X in the equation $378.412\langle 3,0,0\rangle \times \mathrm{X} = 1.05\langle 3,-1,0\rangle$. That is, X is the transformation required to remove the inconsistency arising from the sending port having units gallonUS to a receiving port having units gallonUS/hour. In order, for this result to be communicated to the user a descriptive form for 0.000277<0,-1,0> must be generated. By noting that 0.000277<0,-1,0> = 1 / 3600<0,-1,0> it can be determined that the descriptive form for 0.000277<0,-1,0> is 1/hour.

Stated formally, generating the descriptive form for a unit U requires that U be factored such that $U = U_1^{P_1} \times \ldots \times U_N^{P_N}$ where each U_i is in the UnitLibrary. In theory, the complexity of this factorization is infinite since the range of each P_i is infinite, and there is no limit on the size of N. By limiting N and the possible values of P_i the complexity can be made at least finite. In the current implementation $N \leq 2$ and $P_i \in \{-2,-1,1,2\}$. If a factorization does not exist under these limitations the descriptive form that is “generated” is just a string representation of the unit. I.E., something like “13.27E17<1, -3, 2, 0>”. In practice, this seems to be an acceptable limitation.

3.8.7 UnitsConstraint Solver

The purpose of the solver is to determine if the unit constraints of a model are consistent. It does this by transforming the unit constraints to a set of unit equations in canonical form and then performing a modified form of Gaussian elimination.

For a set of M unit equations let $\{V_1,\ldots,V_N\}$ be the set of variables that occur in any of the set of unit equations. The k^{th} unit equation can then be transformed to its canonical form $V_1^{P_{k_1}} \times \ldots \times V_N^{P_{k_N}} = U_k$. The set of equations in canonical form is represented by a data

structure called a powers matrix that is shown in Figure 3.14

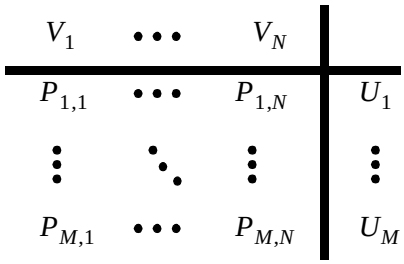


FIGURE 3.14. Power Matrix

A power matrix is said to be *reducable* if there exists a row k that has all but one of the $P_{k,j}$ being 0. Eliminate is an operator that can be applied to any reducable power matrix. The result of the eliminate operator is another power matrix. Let $P_{k,l}$ be that element not equal to 0, then the resulting power matrix has the property that all of the $P_{i,l}$ in column l will be 0 except for $P_{k,l}$ which will have the value 1. See Figure 3.15

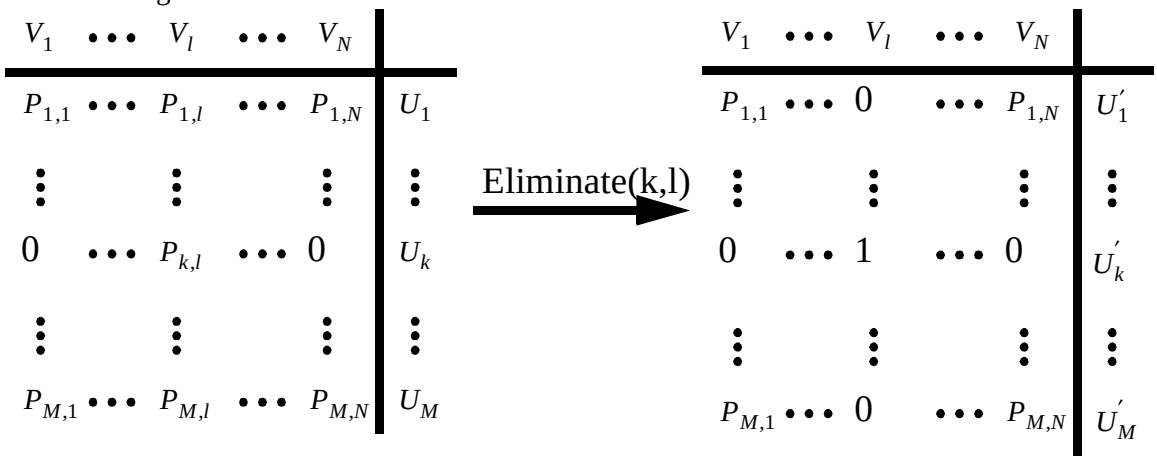


FIGURE 3.15. The eliminate operation. The k[th] row must have all 0 except for the l[th] column. The result is that the l[th] column will be all zeros except $P_{k,l} = 1$

The eliminate operation is accomplished in two steps. The first step replaces $P_{k,l}$ with 1, and the second step replaces all of the other $P_{i,l}$s in the l[th] column with 0. To accomplish the first step note that the k[th] row represents the equation $V_k^{P_{k,l}} = U_k$. If both sides of this equation are raised to the $1/P_{k,l}$ this equation becomes $V_k^1 = U_k^{1/P_{k,l}}$. Thus, $P_{k,l}$ has been replaced with, and $U_k' = U_k^{1/P_{k,l}}$

The second step requires that it be assumed that the value of the variable V_l is in fact $U_k^{1/P_{k,l}}$. Any other row i that is not the k[th] row represents the i[th] equation $V_1^{P_{i_1}} \times \ldots \times V_l^{P_{i_l}} \times \ldots \times V_N^{P_{i_N}} = U_i$. Multiplying both sides of this equation by $V_l^{-P_{i_l}}$ yields

$$V_1^{P_{i_1}} \times \ldots \times V_l^0 \times \ldots \times V_N^{P_{i_N}} = U_i V_l^{-P_{i_l}}$$

$$= U_i U_k^{-P_{i,l}/P_{k,l}}$$

Thus, $P_{i,l}$ has been replaced with 0, and $U_i' = U_i U_k^{\frac{-P_{i,l}}{P_{k,l}}}$

A power matrix is said to be *inconsistent* if there exists a row k with all $P_{k,j} = 0$ but $U_k \neq I$. Note that if a power matrix M is inconsistent then no elimination on M can yield a power matrix that is consistent. A power matrix is said to be *ambiguous* if there exists a column l with more than 1 of the $P_{i,l}$ not equal to 0. A power matrix that is consistent and non-ambiguous yields a set of bindings for the variables. a power matrix is said to be unique if there does not exist any column l where all of the $P_{i,l}$ are 0.

Gaussian elimination is the repeated application of the eliminate operation that terminates when the power matrix can not be further reduced. The final non-reducable power matrix is then used to determine the status of the original set of unit equations. Let M_r be the power matrix that resulting from a Gaussian elimination. If M_r is inconsistent or ambiguous then the set of unit equations does not have a solution. That is, there is no set of bindings for the variables that will cause all of the unit equations to be satisfied. If M_r is consistent and non-ambiguous then there does exist a set of bindings for the variables that will cause the unit equations to be true. Further, if M_r is unique then there exist a binding set that includes all the variables that will cause the unit equations to be true. Figure 3.16 shows examples of inconsistent and ambiguous power matrices.

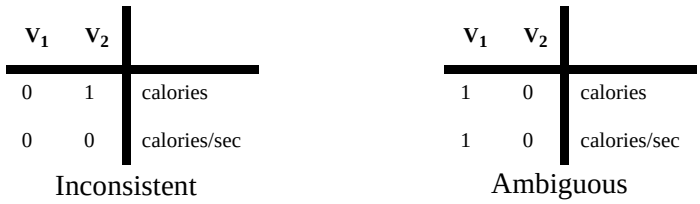


FIGURE 3.16. Examples of inconsistent, ambiguous power matrices

Figure 3.17 shows an example of a power matrices that are consistent, and non-ambiguous.

V_1	V_2	
0	1	calories
1	0	calories/sec
0	0	Identity

V_1	V_2	V_3	
0	1	0	calories
1	0	0	calories/sec
0	0	0	Identity

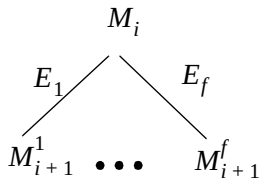
FIGURE 3.17. Consistent, non-ambiguous power matrices. The binding set for the left power matrix is {(V_1, calories), (V_2, calories/sec.)}. The binding set for the right power matrix is{(V_1, calories), (V_2, calories/sec.), (V_3, <unbound>)}

3.8.8 Minimal Span Solutions

Usually an inconsistent set of equations can be made consistent by modifying one or more of the equations and/or changing the membership of the set. In general, however, an inconsistent set of equations does not have a single cause for the inconsistency. That is, it is the set of equations that is inconsistent, not a particular equation in the set that is inconsistent. Stated another way, the cause of the inconsistency is ambiguous. As a result the algorithm presented in the previous subsection can not provide information about why a set of equations is inconsistent. However, it may be possible to provide information that will help the user determine which modifications are appropriate. This is done by presenting the user with a set of minimal span solutions.

A minimal span solution is a solution for a subset of the model, i.e. a subset of the rows in the powers matrix have been eliminated. Furthermore, the subset is one in which the components are connected. The minimal span solution is inconsistent. Finally, if any component is removed the remaining components are consistent. Stated differently, a minimal span solution is inconsistent, but just barely.

The set of minimal span solutions are derived by an adaptation of the Gaussian elimination algorithm that generates the full solution. A minimal span solution is also obtained by applying a sequence of eliminate operations but terminates when a power matrix M_t is generated that is either inconsistent, or is not reducable. Recall that, in general, a power matrix can have more than one eliminate operation applied to it. Thus, the generation of all possible minimal span solutions for a power matrix M_0 forms a tree with M_0 being the root and each leaf being a power matrix that satisfies the termination condition. The structure of each non-leaf node is



where $\{E_1,\ldots,E_f\}$ is the set of all possible eliminate operations that can be applied to M_i.

As an example, the power matrix for the model in Figure 3.8 is shown in Figure 3.18.

V_1	HeatProdcution.heat
V_2	AddSubtract.plus
V_3	AddSubtract.minus
V_4	HeatExchanger.output
V_5	HeatExchanger.flow
V_6	flow.output

V_1	V_2	V_3	V_4	V_5	V_6		source
1	0	0	0	0	0	calories	assignment of calories to HeatProduction.heat
-1	1	0	0	0	0	Identity	connect HeatProduction.heat to AddSubtract.plus
0	-1	1	0	0	0	Identity	AddSubtract requires plus = minus
0	0	-1	1	0	0	Identity	connect AddSubtract.minus to HeatExchanger.output
0	0	0	1	0	0	calories/sec	assignment of calories/sec to HeatExchanger.output
0	0	0	0	1	0	gallonUS/hour	assignment of gallon/US to HeatExchanger.flow
0	0	0	0	-1	1	Identity	connect HeatExchanger.flow to flow.output
0	0	0	0	0	1	gallonUS	assignment of gallonUS to flow.output

FIGURE 3.18. Power matrix for model shown in Figure 3.7

One minimum span solution is shown by

	V_1	V_2	V_3	V_4	V_5	V_6		source
	1	0	0	0	0	0	calories	assignment of calories to HeatProduction.heat
	-1	1	0	0	0	0	Identity	connect HeatProduction.heat to AddSubtract.plus
	0	-1	1	0	0	0	Identity	AddSubtract requires plus = minus
	0	0	-1	1	0	0	Identity	connect AddSubtract.minus to HeatExchanger.output
	0	0	0	1	0	0	calories/sec	assignment of calories/sec to HeatExchanger.output
	0	0	0	0	1	0	gallonUS/hour	assignment of gallon/US to HeatExchanger.flow
Inconsistent	0	0	0	0	-0	0	1/hour	connect HeatExchanger.flow to flow.output
	0	0	0	0	0	1	gallonUS	assignment of gallonUS to flow.output

Here, the minimal span solution has not been greyed out. The row labelled inconsistent caused the

application of the eliminate sequence to terminate. Note, that there are other rows that could be eliminated, but when this row was produced the sequence terminated because minimal span solutions are being generated. Note, also that the unit shown in the inconsistent row is 1/hour and that if the source of this row where replaced with a two port actor that transformed its input by 1/hour then this inconsistency would be removed.

There are two other minimal span solutions of interest

V_1	V_2	V_3	V_4	V_5	V_6		source
1	0	0	0	0	0	calories	assignment of calories to HeatProduction.heat
0	0	0	0	0	0	1/sec	connect HeatProduction.heat to AddSubtract.plus
0	1	0	0	0	0	calories/sec	AddSubtract requires plus = minus
0	0	1	0	0	0	calories/sec	connect AddSubtract.minus to HeatExchanger.output
0	0	0	1	0	0	calories/sec	assignment of calories/sec to HeatExchanger.output
0	0	0	0	1	0	gallonUS/hour	assignment of gallon/US to HeatExchanger.flow
0	0	0	0	-1	1	Identity	connect HeatExchanger.flow to flow.output
0	0	0	0	0	1	gallonUS	assignment of gallonUS to flow.output

Inconsistent

and

V_1	V_2	V_3	V_4	V_5	V_6		source
1	0	0	0	0	0	calories	assignment of calories to HeatProduction.heat
0	1	0	0	0	0	calories	connect HeatProduction.heat to AddSubtract.plus
0	0	1	0	0	0	calories	AddSubtract requires plus = minus
0	0	0	0	0	0	sec	connect AddSubtract.minus to HeatExchanger.output
0	0	0	1	0	0	calories/sec	assignment of calories/sec to HeatExchanger.output
0	0	0	0	1	0	gallonUS/hour	assignment of gallon/US to HeatExchanger.flow
0	0	0	0	-1	1	Identity	connect HeatExchanger.flow to flow.output
0	0	0	0	0	1	gallonUS	assignment of gallonUS to flow.output

Inconsistent

These two minimal span solutions are related in that they have eliminated the same rows in the power matrix. The first indicated that the inconsistency could be fixed by applying the 1/sec transformation

on the relation that connects HeatProduction.heat to AddSubtract.plus. While the second indicates that the inconsistency could be removed by applying the second transformation to the relation that connects AddSubtract.minus to HeatExchanger.output.

3.8.9 Implementing the Units Constraint Solver

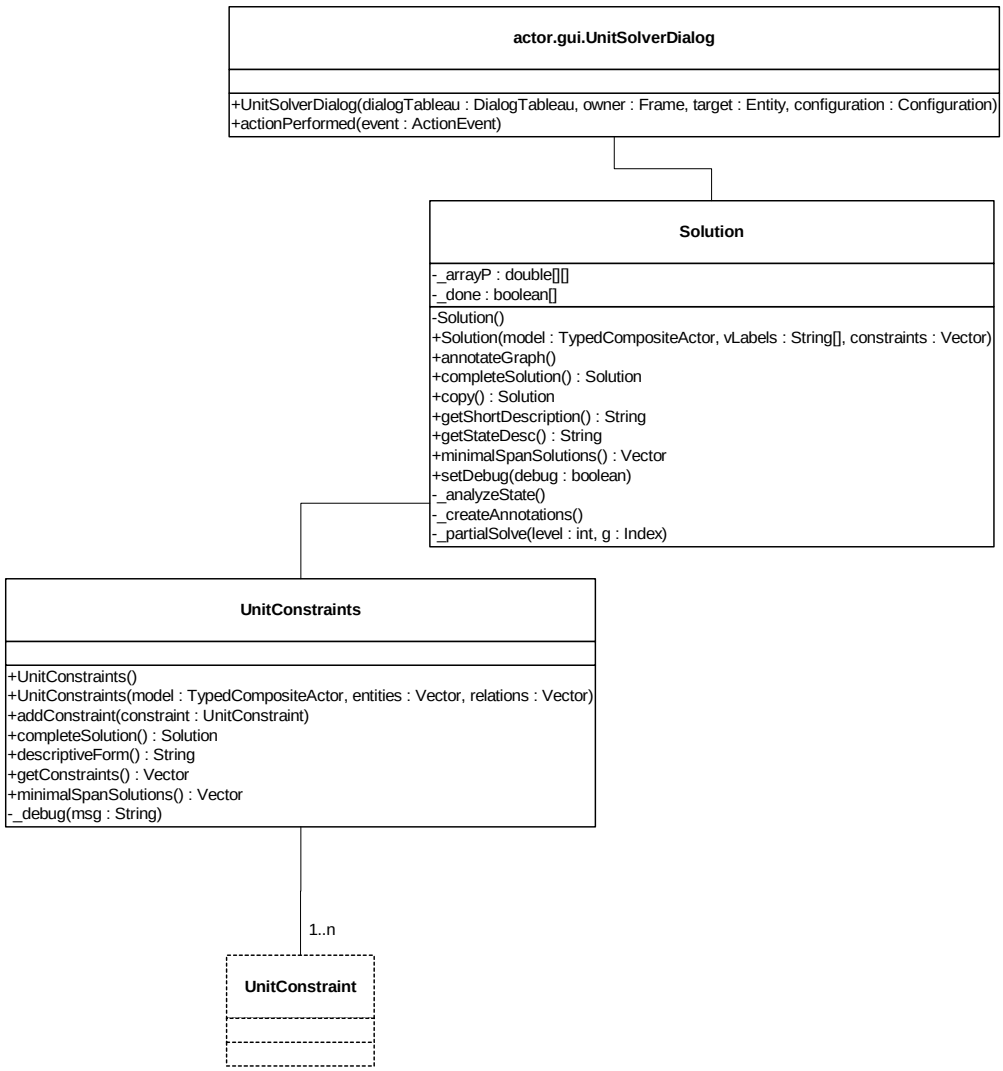


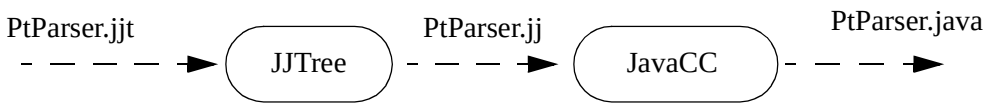
FIGURE 3.19. Static structure of the classes used in the UnitConstraints Solver

Appendix A: Expression Evaluation

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

A.1 Generating the parse tree

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



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

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

In order to reduce the size of the parse tree, nodes that representing many basic operations are designed to have more than two children, even for binary operations. For instance, the parse tree for the expression “2 + 3 + “hello”” only has one sum node. The children are evaluated in the correct order for the associativity of the operator. In this case, the expression evaluates to the string token with value “5hello”.

Note that although functions and constants are registered with the parser, the parser does not actually resolve the values of identifiers. This resolution is performed when the parse tree is actually evaluated. The evaluation process only resorts to registered functions and constants if there are no identifiers defined in the model. This prevents registered functions and constants from unexpectedly shadowing parameters in the model, leading to unexpected behavior. It also allows new functions and constants to be registered without changing the behavior of existing models. Essentially, functions and constants registered with the parser act as a global scope in which all models exist with their own local scopes.

One of the key properties of the expression language is the ability to refer to other parameters by name. Since an expression that refers to other parameters may need to be evaluated several times

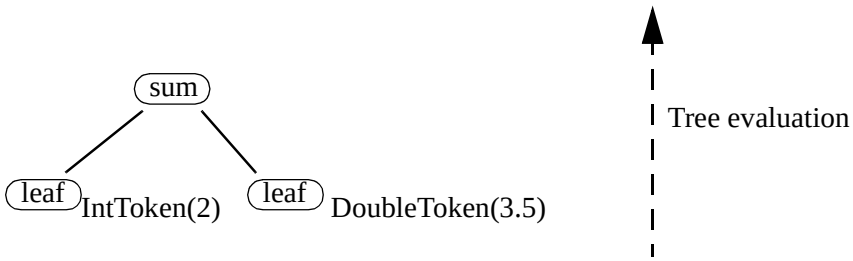
(when the referred parameter changes), it is important that the parse tree does not need to be recreated every time. The classes for representing the parse tree are designed to carry little state, other than the representation of an expression. Generally speaking, users of parse trees, such as the Variable class, cache parse trees for re-evaluation. A new parse tree is only generated when the expression changes. Note, however that the Parser itself is not cached, since it contains a significant amount of internal state.

A.2 Traversing the parse tree

After being generated, the parse tree can be manipulated or traversed, in order to analyze various properties of the original expression. In order to facilitate traversal of the parse tree, the classes representing parse tree nodes implement a visitor design pattern. Each node implements a visit() method that accepts an instance of the ParseTreeVisitor class. When the visit() method of a node is invoked, the node calls an appropriate method of the visitor corresponding to the same node class. The visitor can then operate on the node and recursively invoke the visit method of child nodes to traverse the entire parse tree. This pattern allows the entire logic of a parse tree traversal to be placed in a single class that is largely decoupled from the parse tree itself. Several visitors have been written, and are described below.

A.2.1 Evaluating the parse tree

Parse trees are evaluated using a visitor implemented by the ParseTreeEvaluator class. The parse tree is evaluated in a bottom up manner as each node can only determine its type after the types of all its children have been resolved. As an example consider the input string 2 + 3.5. The parse tree returned from the parser will look like this:



During evaluation, the value of the leaf nodes is first determined, which is trivial in this case, since the values of leaves are constants. These values are then propagated upwards, determining the value of each internal node, until the value of the root node is returned. In this case a DoubleToken with value 5.5 will be returned as the result. If an error occurs during evaluation of the parse tree, an IllegalActionException is thrown with a error message about where the error occurred.

When the ParseTreeEvaluator reaches a instance of the ASTPtLeafNode class that references an identifier, it resolves the identifier into a value through the ParserScope interface. By resolving the values of identifiers through a ParserScope, identifiers can be resolved in different ways depending on how the expression is used. This mechanism is used, for instance to implement the evaluation of function closures and the Expression actors, which interpret expressions differently from parameters. Only if an identifier is not found in scope, is the identifier resolved against the constants registered in the parser.

When the ParseTreeEvaluator reaches a instance of the ASTPtFunctionApplicationNode class, it is handled similarly to a leaf node with an identifier. The name of the function is resolved in the scope, in case the identifier refers to a function closure, an array token, or a matrix token. If the identifier is not found in scope, then reflection to look for that function in the list of classes registered with the parser.

A.2.2 Inferring types of parse trees

The ParseTreeTypeInference class visits parse trees to analyze the type of token resulting from evaluation. For the most part, this operates the same as the ParseTreeEvaluator class, using a ParserScope to resolve the types of identifiers. If identifiers are not present in scope, then they are searched for in the constants or functions registered with the parser.

One difficulty with type inference is that the type of tokens returned from a function invocation can often not be determined from the return type of the Java method. For instance, if the Java method has a return type corresponding to the Token base class, then any token class might be produced. To resolve the types of these methods, the ParseTreeTypeInference class uses Java reflection to find a method with a corresponding name that gives the return type of the original function. For instance, the max() method in the UtilityFunctions class returns the maximum value of an input ArrayToken. Since the ArrayToken can contain any type, the UtilityFunctions class contains a parallel maxReturnType() method that takes a single Type argument, and returns a type. During type inference, this method is found and invoked to properly infer the type returned from the max() method.

A.2.3 Retrieving identifiers in parse trees

The ParseTreeFreeVariableCollector class visits parse trees and extracts the names of all identifiers that need to be resolved to values outside of the expression. In particular, it does not return the names of identifiers that are bound to the arguments of function closures. These identifiers are not accessible outside of the expression. As an example, the expression “foo + bar” has two free variables that must be given values. However, the expression “function(foo:int) foo + bar” has only one free variable, since the identifier “foo” is bound to the argument of the function closure.

A.2.4 Specializing parse trees

The ParseTreeSpecializer class visit parse trees and simplifies them. Primarily, this involves replacing identifier references in leaf nodes with constant values. This operation is an important part of creating FunctionTokens, since the expression inside a function closure can only reference identifiers that are explicitly bound to arguments of the FunctionToken. By specializing the parse tree for the expression, we ensure that the FunctionToken has no dependence on the scope in which it was created. The specializer also analyses the parse tree, finding any internal nodes that are constant after replacing identifiers. These constant nodes are evaluated and replaced by leaf nodes.

A.3 Node types

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

ASTPtArrayConstructNode. This node is created when an array construction sub-expression is parsed. It contains one child node for each element of the array.

ASTPtAssignmentNode . This node is created when an assignment is parsed. It contains exactly two children. The first child is an ASTPtLeafNode corresponding to the identifier being assigned to and the second child corresponds to the assigned expression.

ASTPtBitwiseNode. This node is created when a bitwise operation (&, |, ^) is parsed. It contains at least two child nodes, and each element has the same operation applied.

ASTPtFunctionApplicationNode. This node is created when a function is invoked. The first child is always a node giving the function that will be invoked. For built-in functions, this child will be a leaf node containing an identifier naming the function. The remaining children correspond to arguments of application from left to right.

ASTPtFunctionDefinitionNode. This node is created when a function definition is parsed. For each argument of the function definition, there are two child nodes. The first child node is a leaf node that contains an identifier for the argument name, while the second gives an expression for the type of the argument. If no type is specified, then a child node is created that evaluates to a type of general. The last child node contains an expression tree that defines the function.

ASTPtFunctionalIfNode. This is created when a functional if is parsed. This node always has three children, the first for the boolean condition and the remaining two children for each branch of the expression.

ASTPtLeafNode. This represents the leaf nodes in the AST. The node contains either a token corresponding to constant values, or a string name for an identifier in the expression. This node contains no children.

ASTPtLogicalNode. This node is created when a logical operation (&&, ||) is parsed. It contains at least two child nodes, and each element has the same operation applied.

ASTPtMatrixConstructNode. This is created when a matrix construction sub-expression is parsed. If the matrix is specified explicitly, then this node contains one child node for each element of the matrix. If the matrix is specified using sequence notation for each row, then the node contains three children for each row of the matrix.

ASTPtMethodCallNode. This is created when a method call is parsed. The first child corresponds to the value the method is being invoked on, while the remaining children correspond to arguments of the method call.

ASTPtProductNode. This is created when an arithmatic product operation (*, /,%) is parsed. It contains at least two child nodes, although the same operation need not be applied to each child. The node contains a list of operations corresponding to the individual operations that need to be applied. This list has one fewer element than the number of children.

ASTPtRecordConstructNode. This is created when a record construct sub-expression is parsed. It contains one node for each value in the record and a list of names corresponding to the label for each value.

ASTPtRelationalNode. This is created when one of the relational operators (!=, ==, >, >=, <, <=) is

parsed. It contains exactly two child nodes.

ASTPtRootNode. Parent class of all the other nodes.

ASTPtSumNode. This is created when a arithmatic summation operation (+, -) is parsed. It contains at least two child nodes, although the same operation need not be applied to each child. The node contains a list of operations corresponding to the individual operations that need to be applied. This list has one fewer element than the number of children.

ASTPtUnaryNode. This is created when a unary negation operator (!, ~, -) is parsed. It always contains exactly one child node.

A.4 Extensibility

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

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

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

4

Graph Package

Authors: *Shuvra S. Bhattacharyya*
Shahrooz Shahparnia
Ming-Yung Ko
Jie Liu
Yuhong Xiong
Paul Whitaker

4.1 Introduction

The Ptolemy II kernel provides extensive infrastructure for creating and manipulating clustered graphs of a particular flavor. Mathematical graphs, however, are simpler structures that consist of nodes and edges, without hierarchy. Edges link pairs of nodes, and therefore are much simpler than the relations of the Ptolemy II kernel. Moreover, in mathematical graphs, no distinction is made between multiple edges that may be adjacent to a node, so the ports of the Ptolemy II kernel are not needed. A large number of algorithms have been developed that operate on mathematical graphs, and many of these prove extremely useful in support of scheduling, type resolution, and other operations in Ptolemy II. Thus, we have created the *graph* package, which provides efficient data structures for mathematical graphs, and collects algorithms for operating on them. At this time, the collection of algorithms is nowhere near as complete as in some widely used packages, such as LEDA [87]. But this package will serve as a repository for a growing suite of algorithms.

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

The graphs constructed using this package are designed to provide broad support for algorithms that operate on generic, mathematical graphs. A typical use of this package is to construct a graph that represents the topology of a CompositeEntity, run a graph algorithm, and extract useful information from the result. For example, a graph might be constructed that represents data precedences, and a

topological sort might be used to generate a schedule. In this kind of application, the hierarchy of the original clustered graph is flattened, so nodes in the graph represent only opaque entities.

4.2 Classes and Interfaces in the Graph Package

Figures 4.1and 4.2 show the class diagram of the graph package. The classes Node, Edge, Graph, DirectedGraph and DirectedAcyclicGraph support graph construction and provide graph algorithms. Currently, only a limited set of algorithms are implemented; other algorithms will be added as needed. The CPO interface defines the basic CPO operations, and the class DirectedAcyclicGraph implements this interface. An instance of DirectedAcyclicGraph is also a finite CPO where all the elements and order relations are explicitly specified. Defining the CPO operations in an interface allows future expansion to support infinite CPOs and finite CPOs where the elements are not explicitly enumerated. The InequalityTerm interface and the Inequality class model inequality constraints over the CPO. The details of the constraints will be discussed later. The InequalitySolver class provides an algorithm to solve a set of constraints. This is used by the Ptolemy II type system, but other uses may arise.

None of the classes in this package is synchronized. If multiple threads access a graph or analysis or data structure concurrently, external synchronization will be needed.

4.2.1 Element and ElementList

A graph element consists of an optional *weight* (an arbitrary object that is associated with the element). We say that an element is *unweighted* if it does not have an assigned weight.

An element list is a list of graph elements. This class manages the storage and weight information associated with a list of unique graph elements. This class is normally for use internally within graph classes. The list is implemented as a HashMap.

4.2.2 Labeled Lists

LabeledList is a type of ElementList that it is used as a support class for graphs in this package and allows one to construct efficient mappings from subsets of nodes and/or edges into arbitrary values. A LabeledList is a list of unique objects (elements) with an assignment from the elements into consecutive integer labels. The labels are consecutive integers between 0 and $N-1$ inclusive, where N is the total number of elements in the list. This list features $O(1)$ list insertion, $O(1)$ testing for membership in the list, $O(1)$ access of a list element from its associated label, and $O(1)$ access of a label from its corresponding element. The element labels are useful, for example, in creating mappings from list elements into elements of arbitrary arrays. More generally, element labels can be used to maintain arbitrary m-dimensional matrices that are indexed by the list elements (via the associated element labels).

Element labels maintain their consistency (remain constant) during periods when no elements are removed from the list. When elements are removed, the labels assigned to the remaining elements may change.

Elements themselves must be non-null and distinct, as determined by the *equals* method.

This class supports all required operations of the list interface, except for the *subList* operation, which results in an *UnsupportedOperationException*.

4.2.3 Node

This class derived from Element models a vertex for inclusion in undirected or directed graphs.

More specifically, all vertices in a graph are Node instances, and each node has an optional *weight* (an arbitrary object that is associated with the node). We say that a node is *unweighted* if it does not have an assigned weight. It is an error to attempt to access the weight of an unweighted node. Node weights must be genuine (non-null) objects.

4.2.4 Edge

This class derived from Element represents a weighted or unweighted edge for a directed or undirected graph. The connectivity of edges is specified by *source* nodes and *sink* nodes. A directed edge is directed *from* its source node *to* its sink node. For an undirected edge, the source node is simply the first node that was specified when the edge was created, and the sink node is the second node. This convention allows undirected edges to later be converted in a consistent manner to directed edges, if desired.

On creation of an edge, an arbitrary object can be associated with the edge as the *weight* of the edge. We say that an edge is *unweighted* if it does not have an assigned weight. It is an error to attempt to access the weight of an unweighted edge.

Self-loop edges (edges whose source and sink nodes are identical) are allowed.

The source node and sink node of an edge cannot be changed.

4.2.5 Graph

This class models a graph with optionally-weighted edges and nodes. Nodes and edges of a graph are instances of Node and Edge, respectively. Thus, each node or edge may have a weight associated with it. The nodes (edges) in a graph are always distinct, but their weights need not be.

Each node (edge) has a unique, integer label associated with it. These labels can be used, for example, to index arrays and matrixes whose rows/columns correspond to nodes (edges). Both directed and undirected graphs can be implemented using this class. In directed graphs, the order of nodes specified to the *addEdge* method is relevant, whereas in undirected graphs, the order is unimportant. Support for both undirected and directed graphs follows from the combined support for these in the underlying Node and Edge classes. The DirectedGraph class provides more thorough support for directed graphs.

The same node can exist in multiple graphs, but any given graph can contain only one instance of the node. Node labels, however, are local to individual graphs. Thus, the same node may have different labels in different graphs. Furthermore, the label assigned in a given graph to a node may change over time (if the set of nodes in the graph changes). The weight of a node is identical for all instances of the node in multiple graphs. All of this holds for edges all well. The same weight may be shared among multiple nodes and edges.

Multiple edges in a graph can connect the same pair of nodes. Thus, multigraphs are supported.

Once assigned, node and edge weights should not be changed in ways that affect comparison under the *equals* method Otherwise, unpredictable behavior may result.

4.2.6 Directed Graphs

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

The computation of transitive closure operations is implemented in this class. The transitive closure is internally stored as a two-dimensional boolean matrix, whose indices correspond to node labels.

The entry (*i*, *j*) is *true* if and only if there exists a path from the node with label *i* to the node with label *j*. This matrix is not exposed at the public interface; instead, it is used by this class and its subclass to do other operations. Once the transitive closure matrix is computed, graph operations like *reachableNodes* can be easily accomplished.

Some methods in this class have two versions, one that operates on graph nodes, and another that operations on node weights. The latter form is called the *weights version*. More specifically, the weights version of an operation takes individual node weights or arrays of weights as arguments, and, when applicable, returns individual weights or arrays of weights.

Multiple edges in a graph can be directed between the same pair of nodes (in the same direction). Thus, directed multigraphs are supported.

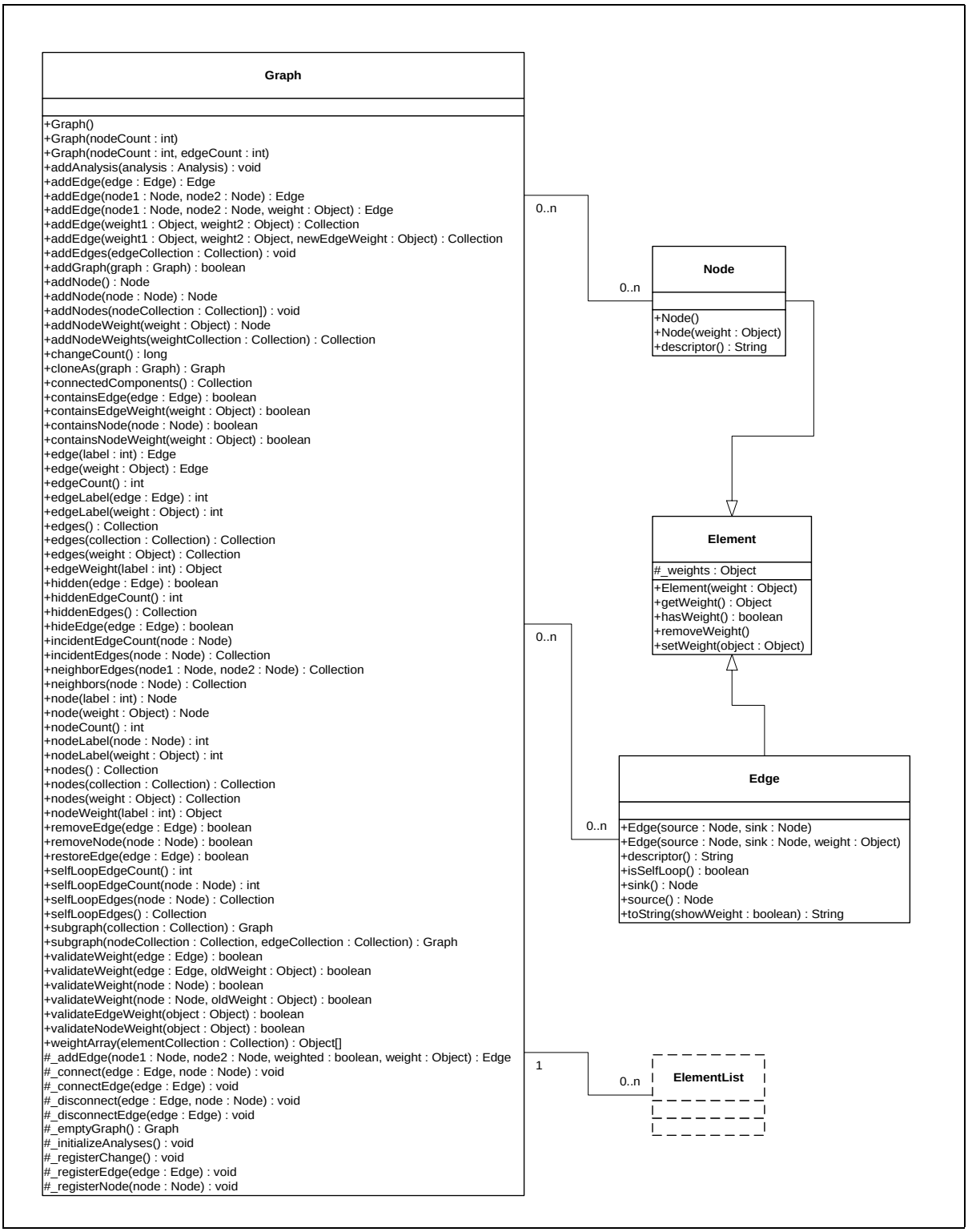


FIGURE 4.1. Core of the graph package.

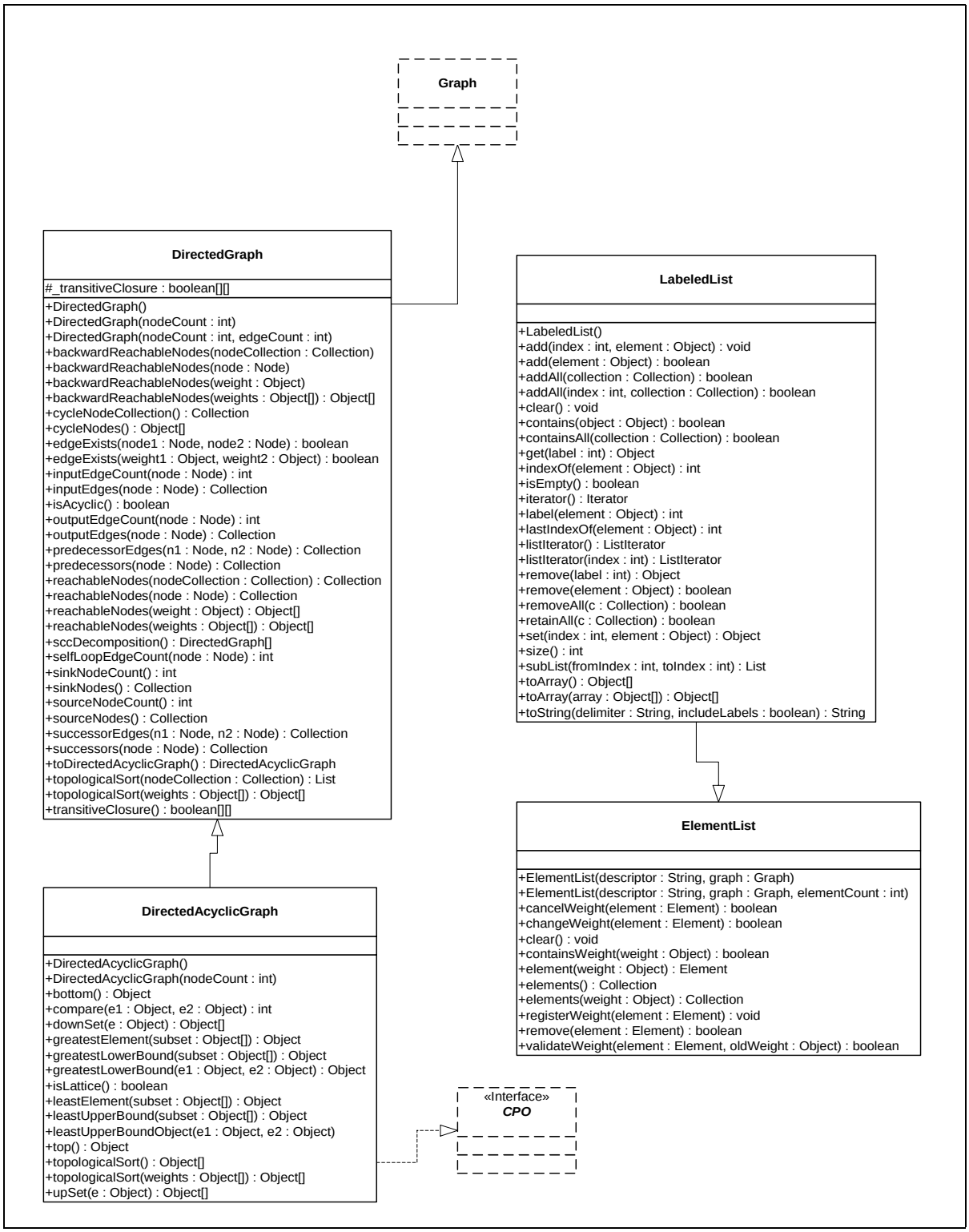


FIGURE 4.2. Core of the graph package (Cont.).

4.2.7 Graph Mappings

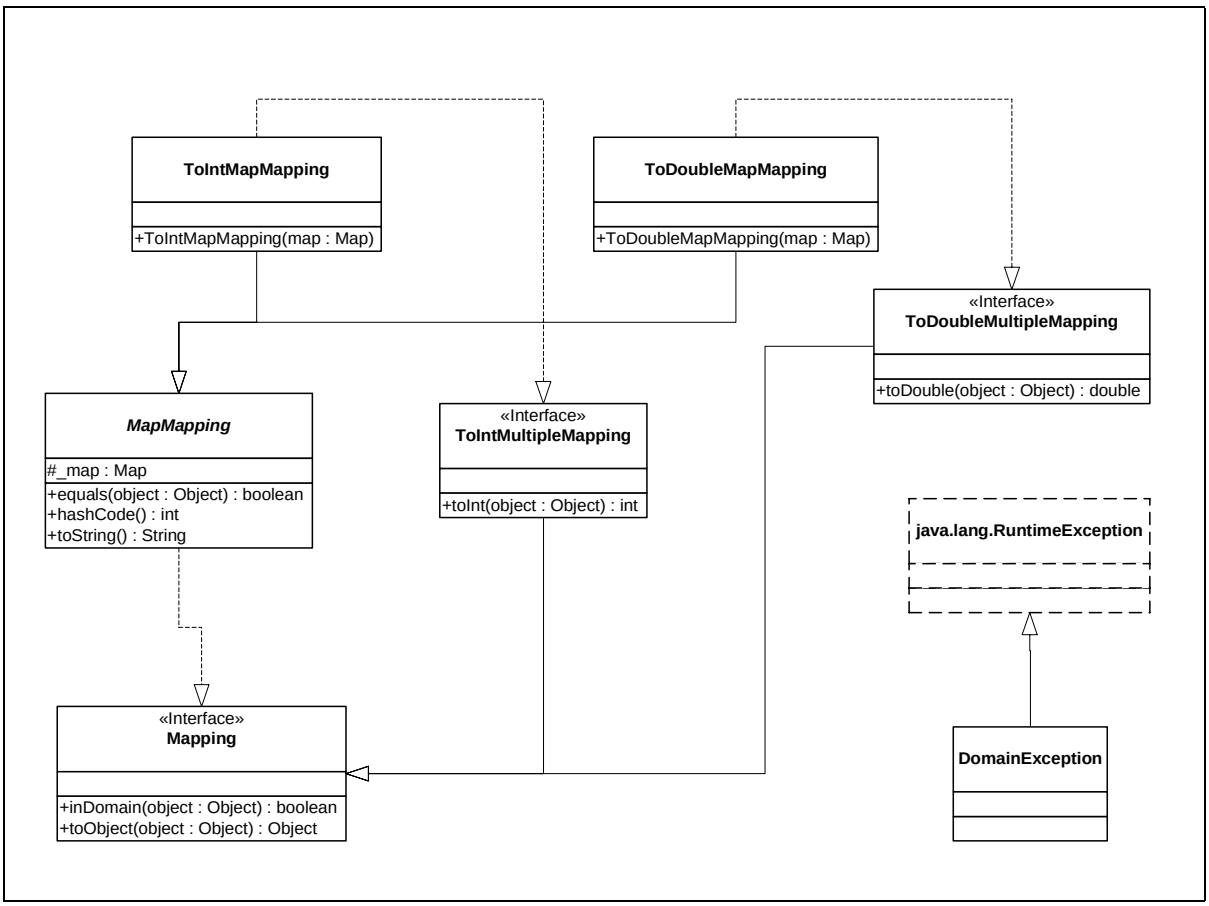


FIGURE 4.3. Classes in the graph.mapping package.

4.2.8 Graph Analysis

The Analysis package implements the Strategy Design Pattern. This design pattern consists of decoupling an algorithm from its host, and encapsulating the algorithm into a separate class. More simply put, an object and its behavior are separated and put into two different classes. This allows the user to switch the algorithm that she/he is using at any time. There are several advantages to doing this. First, if you have several different behaviors that you want an object to perform, it is much simpler to keep track of them if each behavior is a separate class, and not buried in the body of some method. Should you ever want to add, remove, or change any of the behaviors, it is a much simpler task, since each one is its own class. Each such behavior or algorithm encapsulated into its own class is called a Strategy.

In other words strategies define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Classes in ptolemy.graph.analysis consists of different wrappers in which a client can plug a requested strategy/algorithm for an analysis. Strategies for a given analysis implement the same interface defined in ptolemy.graph.analysis.analyzer.

Therefore from now on we will use the name analyzer for all the strategies that implement the same interface and therefore solve the same problem. Analysis classes access the plugged-in strategy class through these interfaces. The strategies classes are defined in ptolemy.graph.analysis.strategy. In addition, the analysis classes provide default constructors which use predefined strategies for those clients who do not want to deal with different strategies. This may introduce some limitations imposed by the used strategy. The documentation of such constructors should reflect the limitations, if any.

Finally, strategies can be instantiated and used independently. In this case the client will lose the possibility of dynamically changing the analyzer for the associated analysis, which would not exist at all, and there will be no default constructor therefore the client need to be familiar with the strategy that she/he is using.

In the base class, methods are provided in order to dynamically change the analyzer of the current analysis and also to check if a given analyzer is applicable to the given analysis.

Analyzers that can be used in these analyses are specialized versions of analyzers of the type ptolemy.graph.analysis.analyzer.GraphAnalyzer

Classes in ptolemy.graph.analysis.analyzer are the interfaces for different strategies (algorithms) used for the analysis. A list of available analyses follows:

Single source longest path, self loop, source node and sink node, transitive closure, cycle mean (maximum and minimum), cycle existence, all-pairs shortest path, negative-weight cycle detection, zero-weight cycle detection, maximum profit to cost ratio, and mirror of a graph.

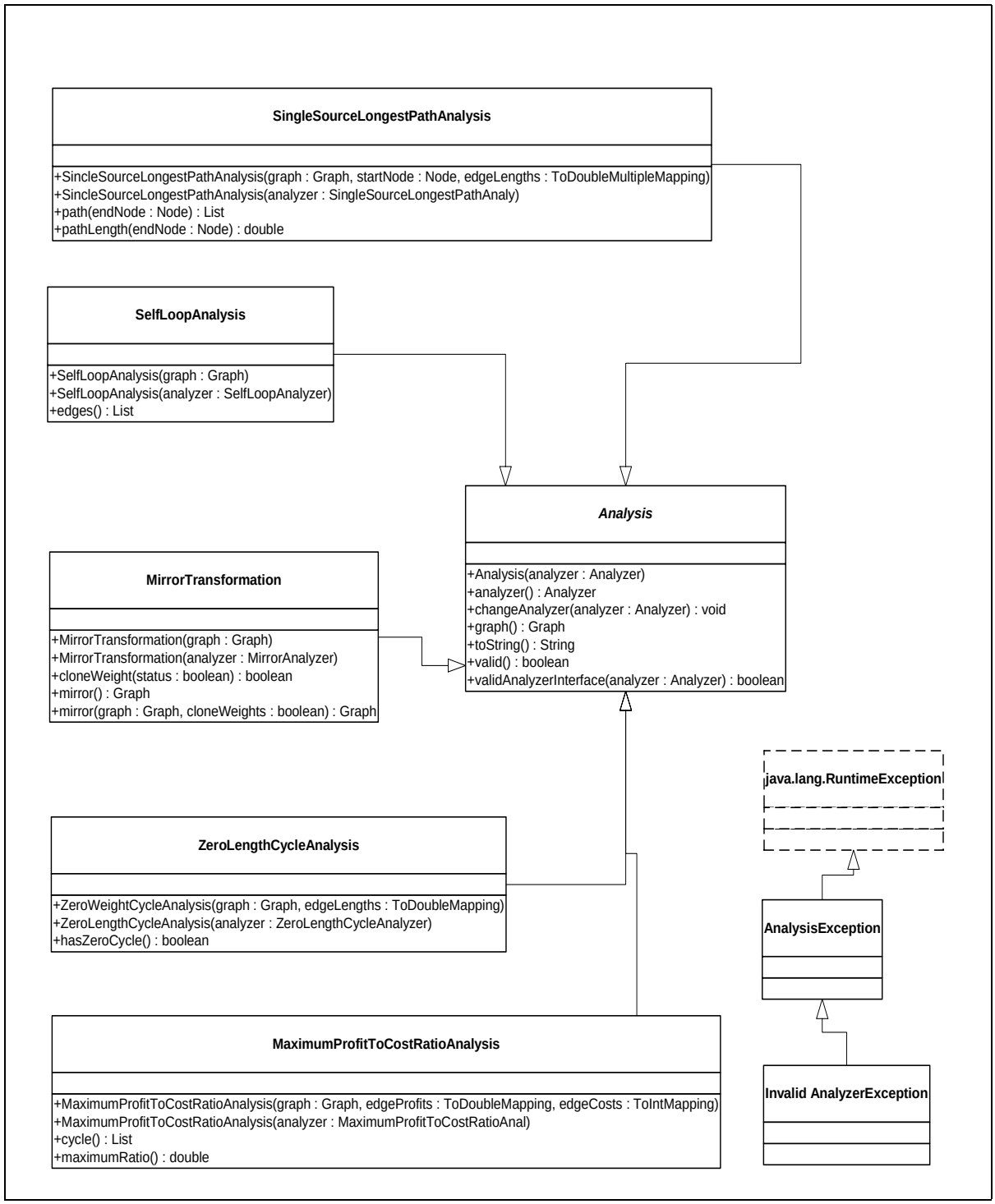


FIGURE 4.4. Classes in the graph.analysis package.

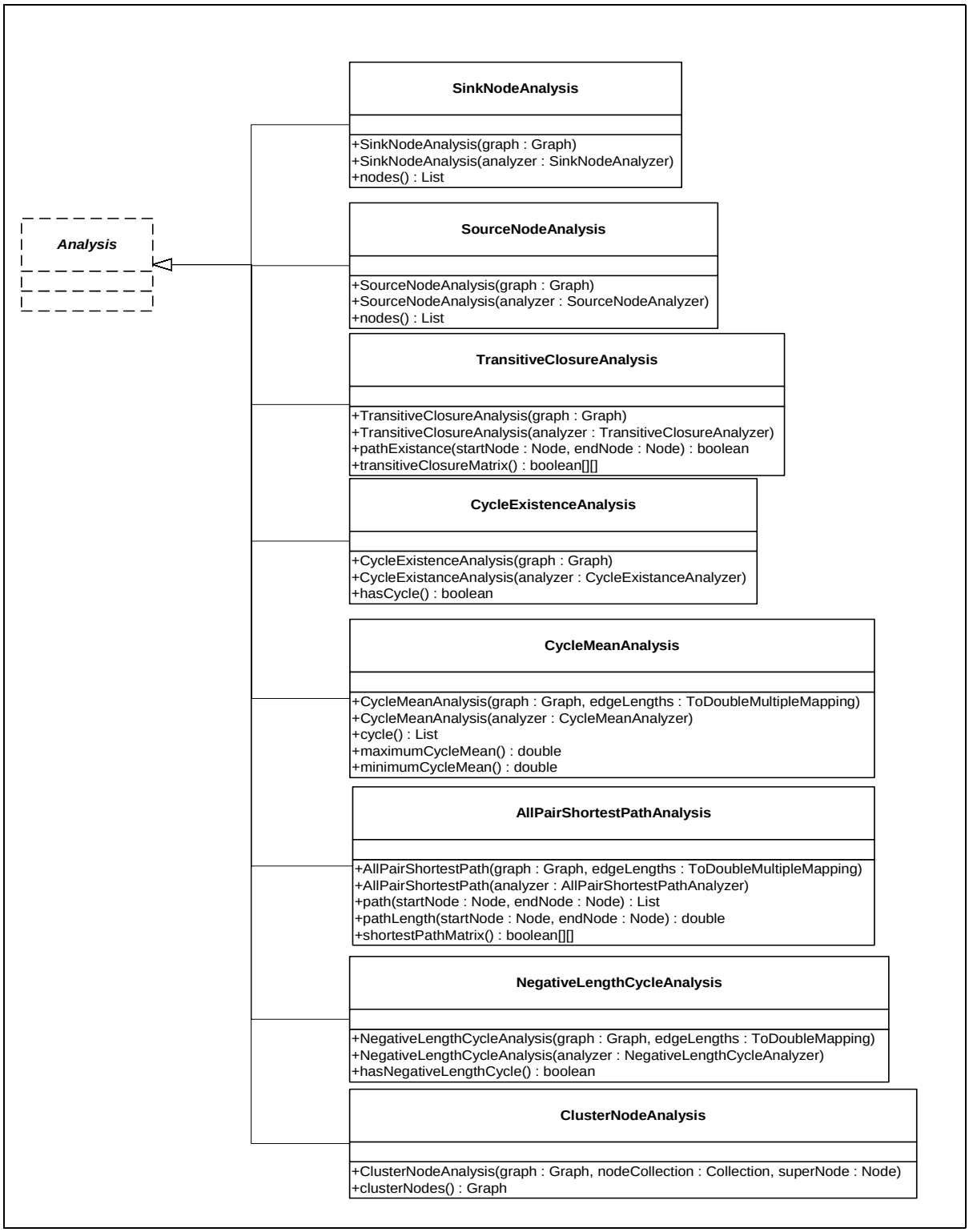


FIGURE 4.5. Classes in the graph.analysis package (Cont.).

4.2.9 Graph Analyzers

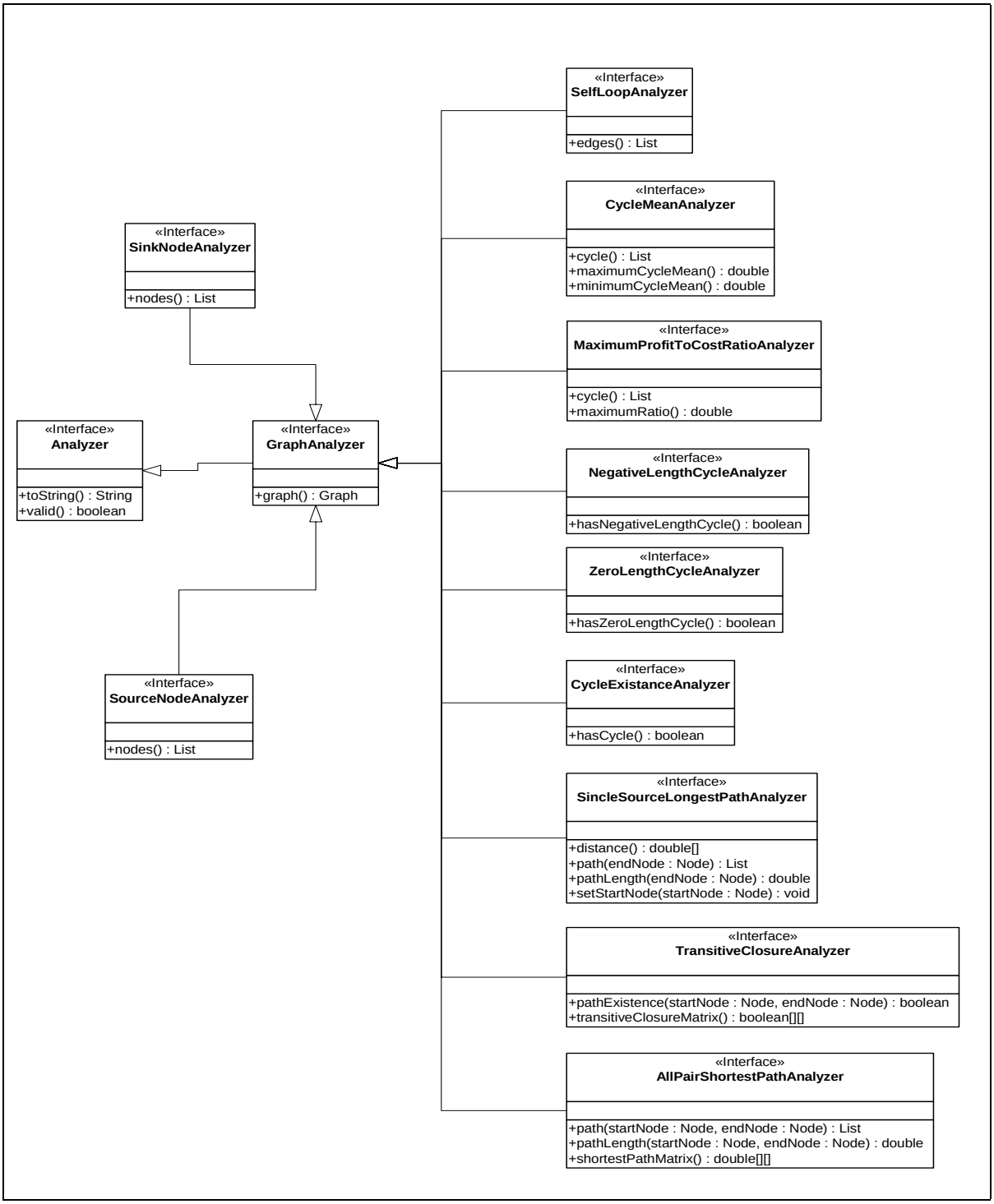


FIGURE 4.6. Classes in the graph.analysis.analyzer package.

4.2.10 Strategies

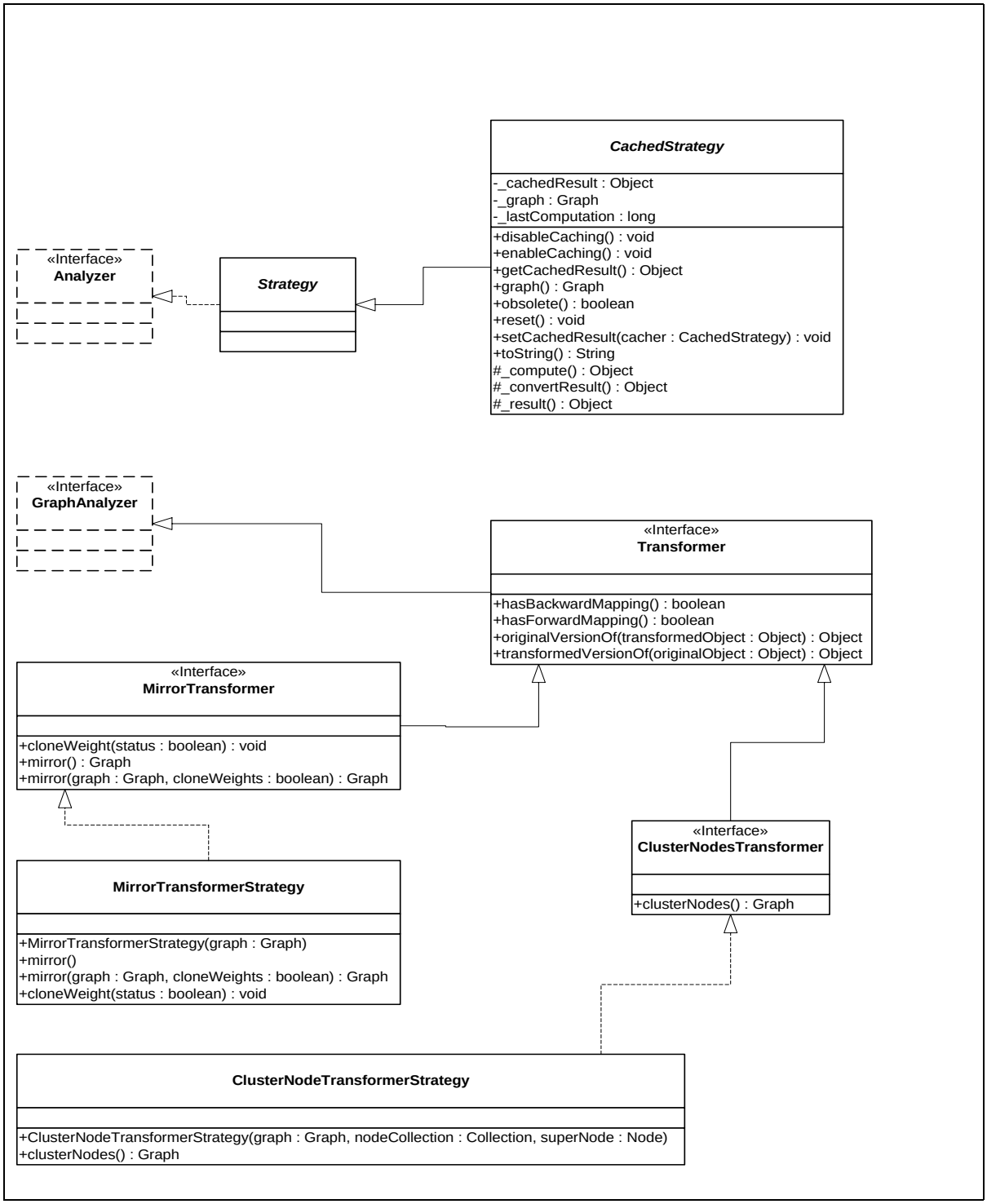


FIGURE 4.7. Classes in the graph.analysis.strategy.

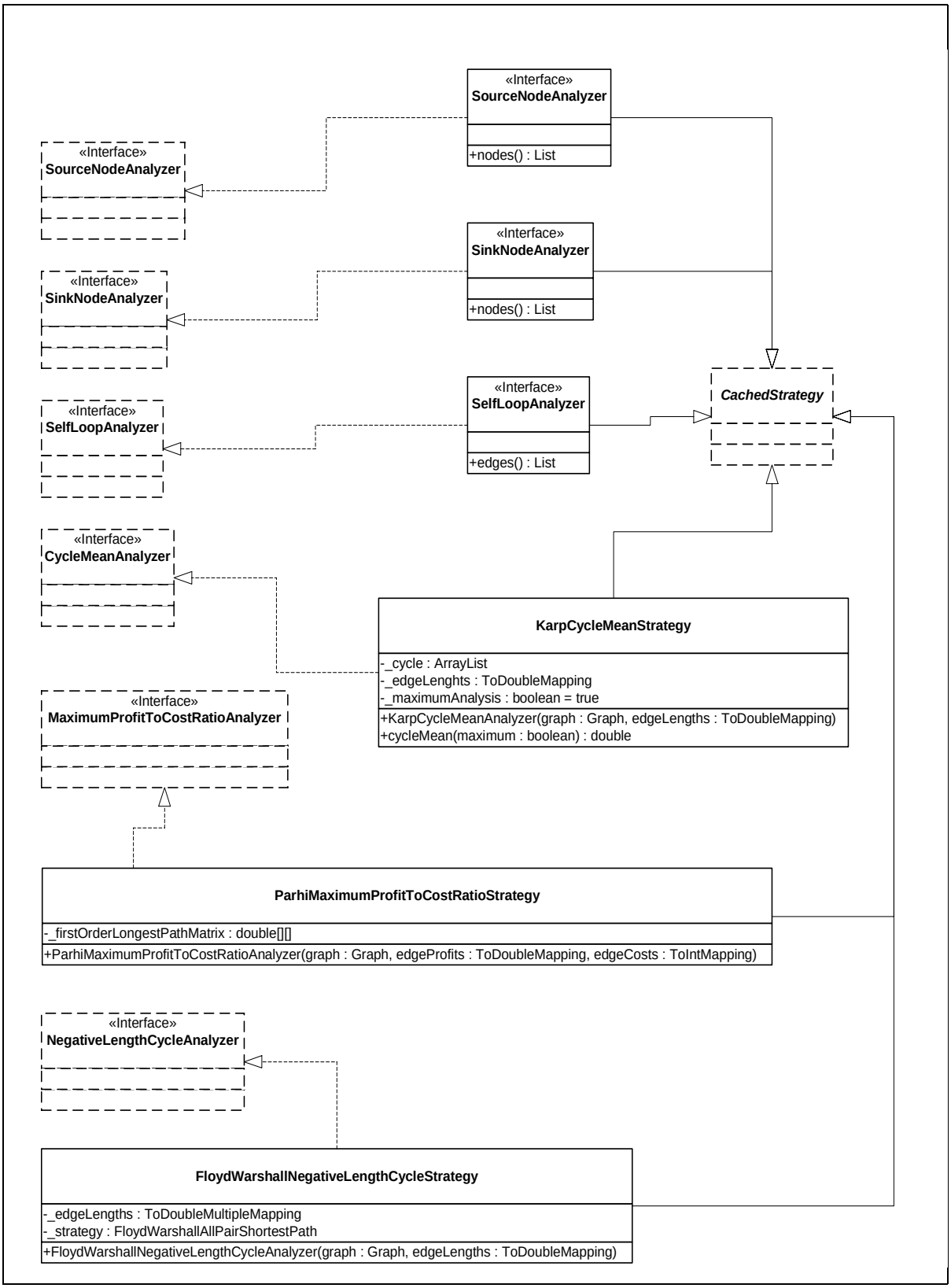


FIGURE 4.8. Classes in the graph.analysis.strategy (Cont.).

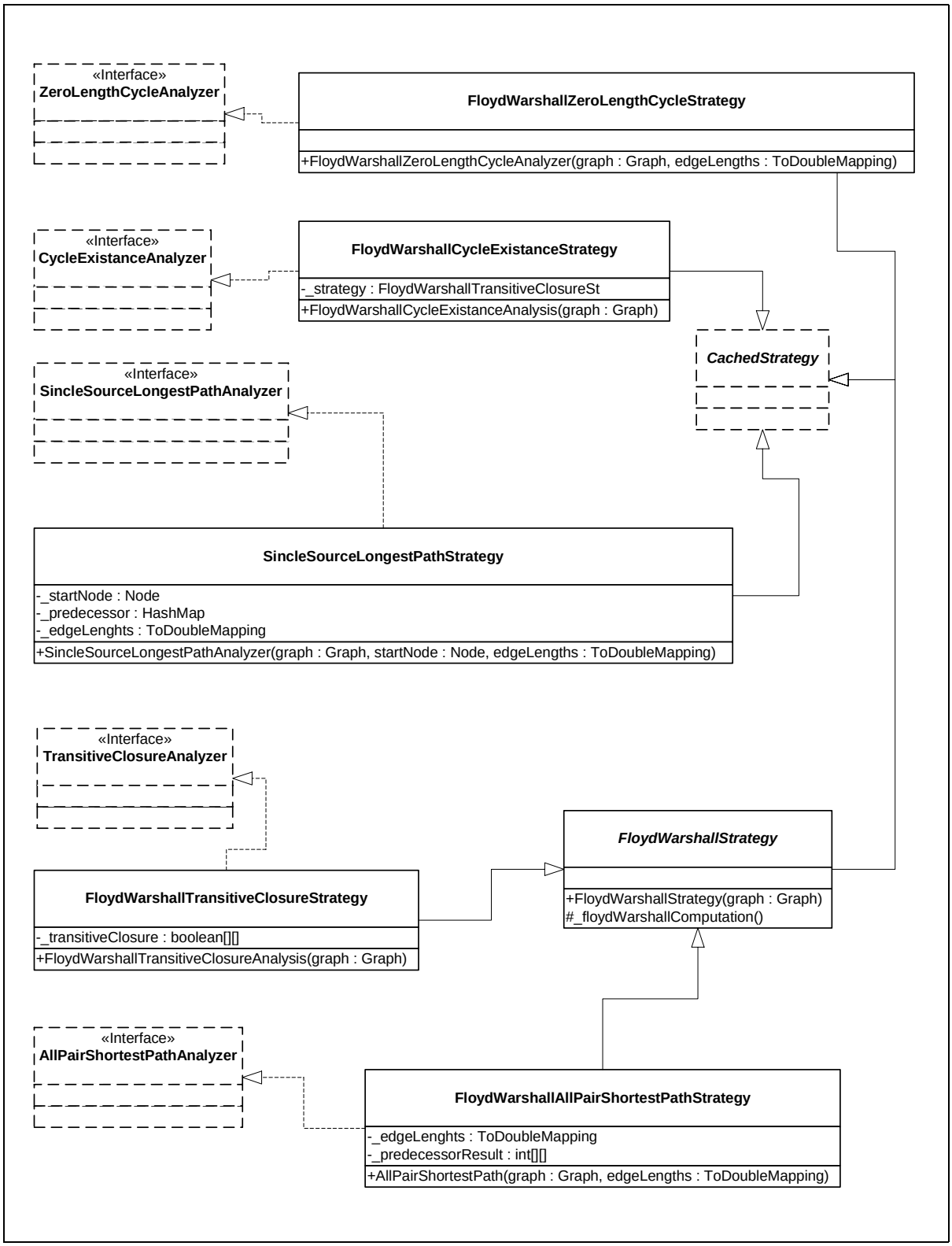


FIGURE 4.9. Classes in the graph.analysis.strategy (Cont.).

4.2.11 Cached Strategies vs. Non-Cached Strategies

To facilitate demand-driven and incremental recomputation of analyzers, the results of those strategies that extend the ptolemy.graph.analysis.CachedStrategy class are cached internally, and are recomputed only when the graph has changed since the last request for the strategy result.

The graph changes tracked by an analyzer are restricted to changes in the graph topology (the set of nodes and edges). For example, changes to edge/node weights that may affect the result of an analysis are not tracked, since analyzers have no specific knowledge of weights. In such cases, it is the responsibility of the client (or derived analyzer class) to invalidate the cached result when changes to graph weights or other non-topology information render the cached result obsolete. For this reason, some caution is generally required when using analyzers whose results depend on more than just the graph topology. In these cases the client should check for data consistency. Refer to the class API for more details.

4.2.12 Graph Schedules

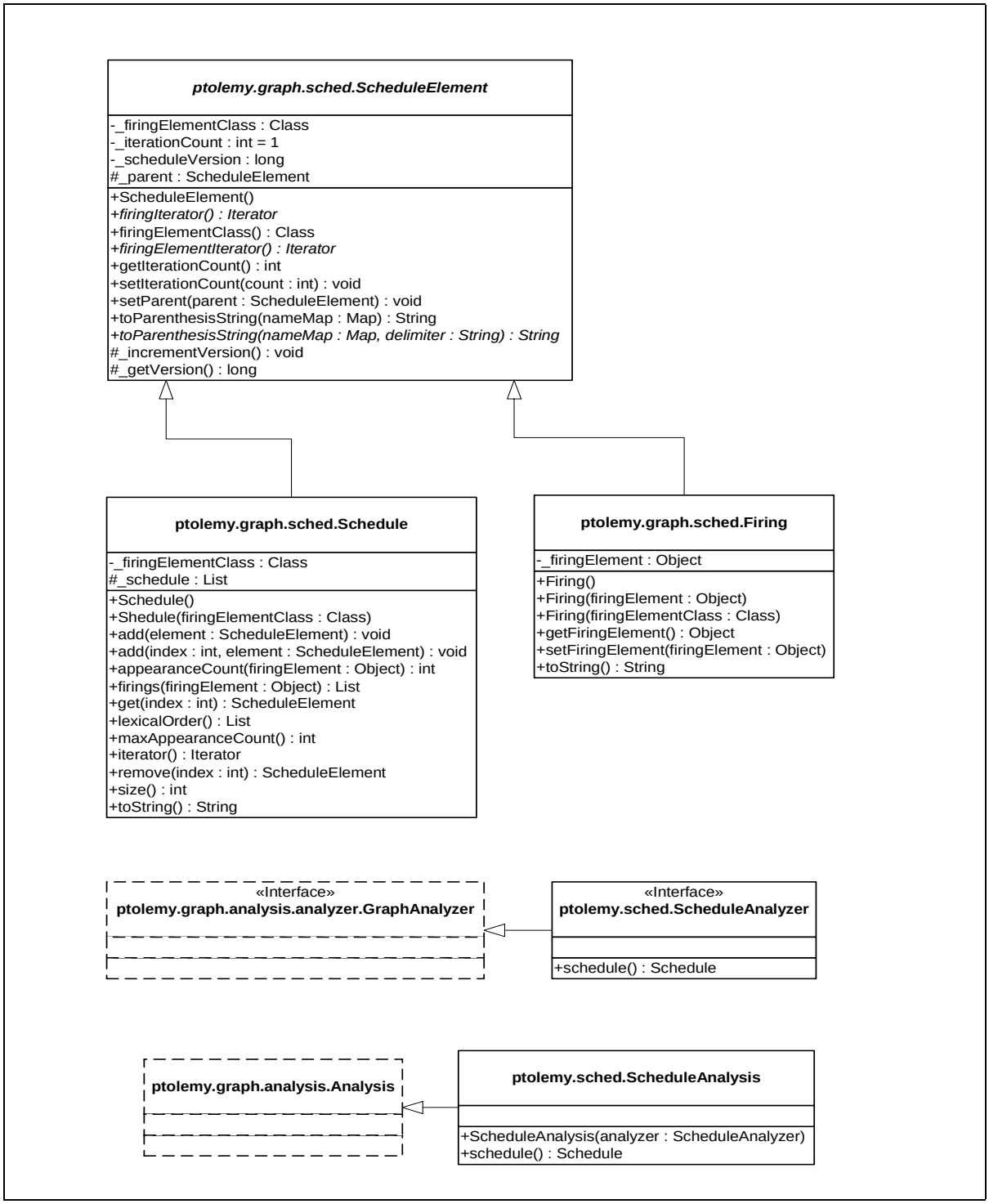


FIGURE 4.10. Classes in the graph.sched package.

4.2.13 Graph Exceptions

The GraphException class is the base class for exceptions for graph errors. This is also an instance of RuntimeException.

GraphElementException is an exception for graph element access errors. The errors are usually due to access of nonexistent elements or invalid element types.

GraphWeightException is an exception for graph element weight access errors. The errors are usually due to access of nonexistent weights or invalid weight types.

GraphConstructionException is an exception for graph structure construction errors. Some examples of the errors are: addition of elements already existing, and addition of an edge where a connection between the ending nodes is built.

GraphStateException is thrown when a functional computation is executed on a graph with incorrect states. Graphs with incorrect states lead to invalid results or even making functions incomputable. Our design should make it impossible for this exception to ever occur, so occurrence is a bug.

GraphTopologyException is an exception for operations on invalid graph topology.

GraphActionException is an exception for invalid actions executed on graphs. The actions refer to operations taking a graph as an argument rather than that in modifying the graph structure. For the latter case, GraphConstructionException should be thrown.

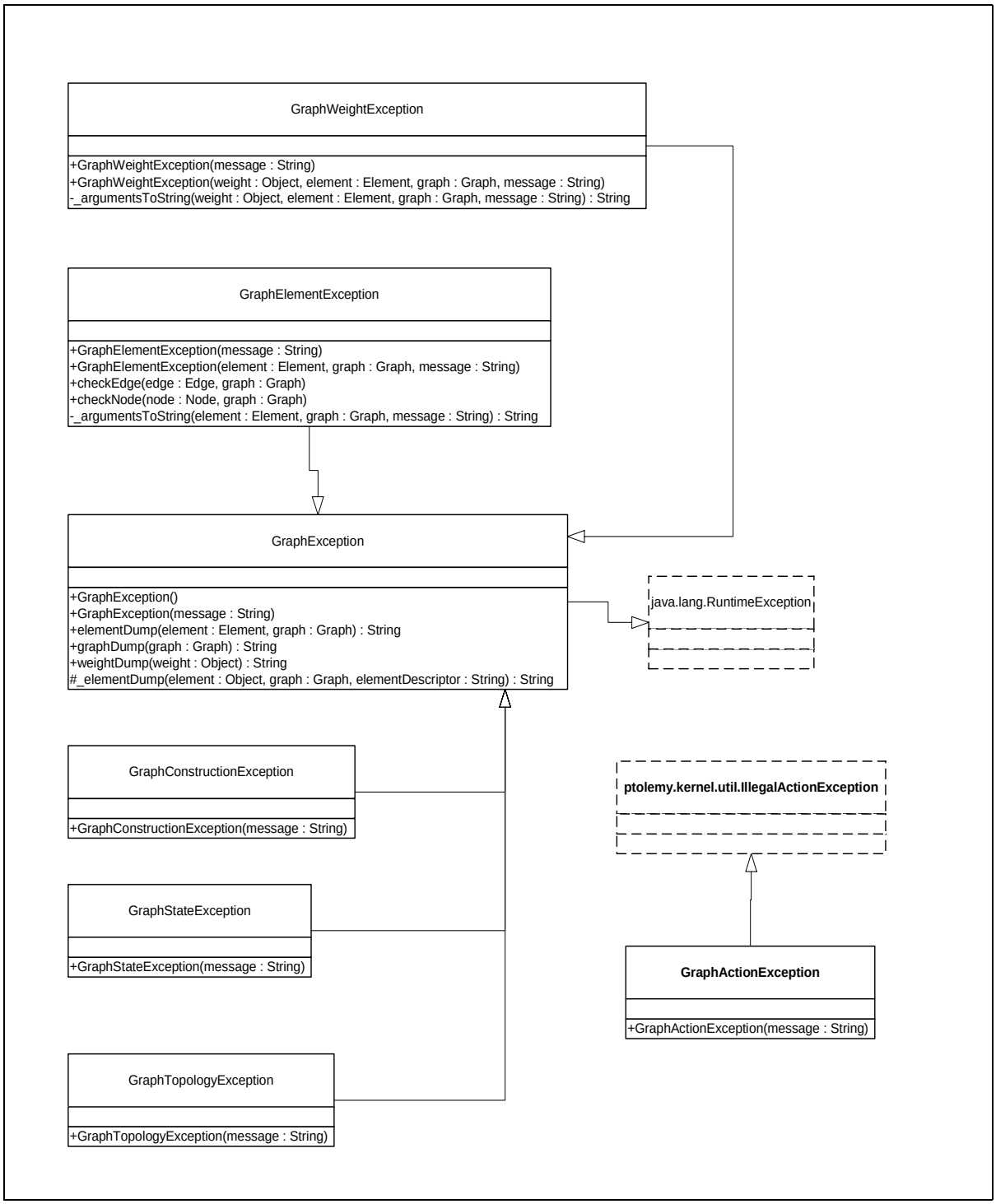


FIGURE 4.11. Classes in the graph exceptions.

4.2.14 Directed Acyclic Graphs and CPO

The DirectedAcyclicGraph class shown in Figure 4.4.2 further restricts DirectedGraph by not allowing cycles. For performance reasons, this requirement is not checked when edges are added to the graph, but is checked when any of the graph operations is invoked. An exception is thrown if the graph is found to be cyclic.

The CPO interface defines the common operations on CPOs. The mathematical definition of these operations can be found in [24]. Informal definitions are given in the class documentation. This interface is implemented by the class DirectedAcyclicGraph.

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

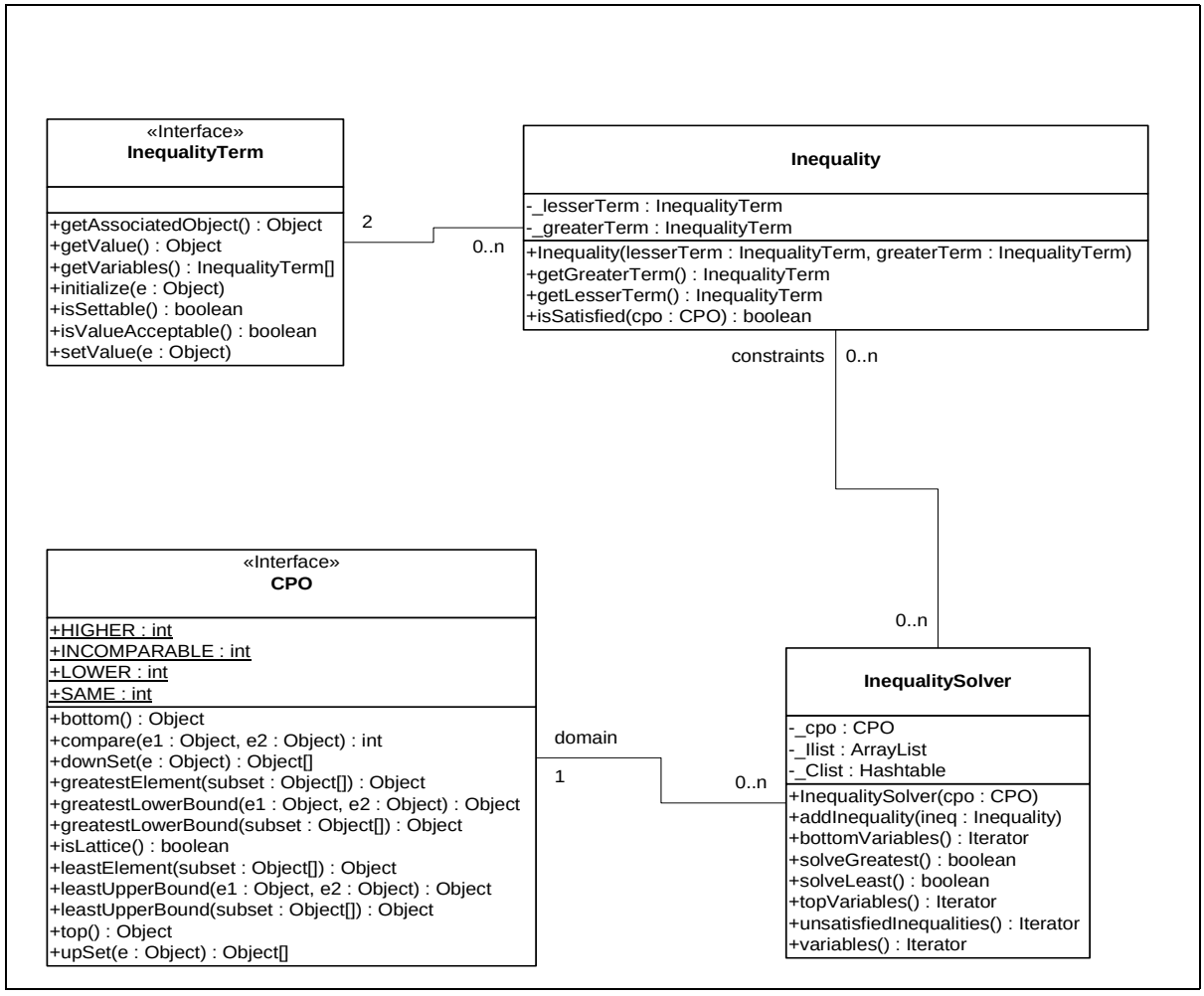


FIGURE 4.12. CPO related classes in the core of the graph package.

4.2.15 Inequality Terms, Inequalities, and the Inequality Solver

The InequalityTerm interface and Inequality and InequalitySolver classes support the construction of a set of inequality constraints over a CPO and the identification of a member of the CPO that satisfies the constraints. A constraint is an inequality defined over a CPO, which can involve constants, variables, and functions. As an example, the following is a set of constraints over the 4-point CPO in Figure 4.4.13:

$\alpha \le w$

$\beta \le x \wedge \alpha$

$\alpha \le \beta$

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

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

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

The InequalitySolver class implements an algorithm to determine satisfiability of a set of inequality constraints and to find the solution to the constraints if they are satisfiable. This algorithm is described in [107]. It is basically an iterative procedure to update the value of variables until all the constraints are satisfied, or until conflicts among the constraints are found. Some limitations on the type of constraints apply for the algorithm to work. The method addInequality() adds an inequality to the set of constraints. Two methods solveLeast() and solveGreatest() can be used to solve the constraints. The former tries to find the least solution, while the latter attempts to find the greatest solution. If a solution is found, these methods return *true* and the current value of the variables is the solution. The method unsatisfiedInequalities() returns an enumeration of the inequalities that are not satisfied based on the current value of the variables. It can be used after solveLeast() or solveGreatest() return *false* to find out which inequalities cannot be satisfied after the algorithm runs. The bottomVariables() and topVariables() methods return enumerations of the variables whose current values are the bottom or the top element of the CPO.

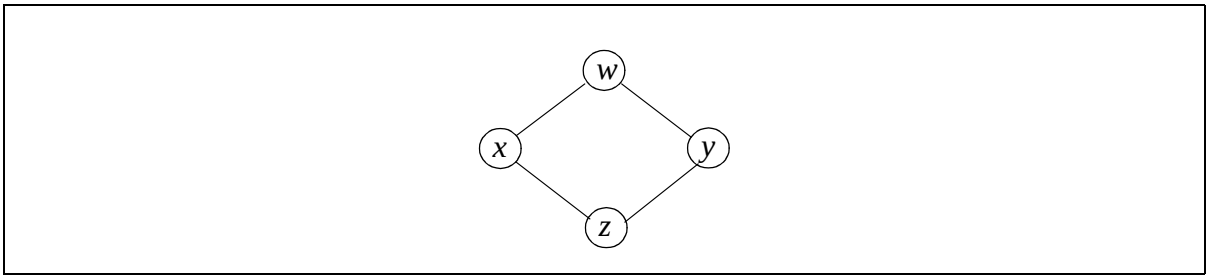


FIGURE 4.13. A 4-point CPO that also happens to be a lattice.

4.3 Example Use

4.3.1 Generating A Schedule for a Composite Actor

Figure 4.4.14 shows an example of using a topological sort to generate a firing schedule for a CompositeActor of the actor package. The connectivity information among the Actors within the composite is translated into a directed acyclic graph, with each node of the graph represented by an Actor. The schedule is stored in an array, where each element of the array is a reference to an Actor.

4.3.2 Forming and Solving Constraints over a CPO

The code in Figure 4.4.15 implements the InequalityTerm interface and models the variable term. The values of these terms are Strings. Inequalities can be formed using these two classes. As another example, the class in Figure 4.4.16 constructs the 4-point CPO of Figure 4.4.13, forms a set of constraints with three inequalities, and solves for both the least and greatest solutions. The inequalities are $a \leq w$; $b \leq a$; $b \leq z$, where w and z are constants in Figure 2.3, and a and b are variables.

```
Object[] generateSchedule(CompositeActor composite) {
   DirectedAcyclicGraph dag = new DirectedAcyclicGraph();
   // Add all the actors contained in the composite to the graph.
   Iterator actors = composite.deepEntityList().iterator();
   while (actors.hasNext()) {
      Actor actor = (Actor)actors.next();
      dag.addNodeWeight(actor);
   }

   // Add all the connection in the composite as graph edges.
   actors =  composite.deepEntityList().iterator();
   while (actors.hasNext()) {
              Actor lowerActor = (Actor)actors.next();
              // Find all the actors "higher" than the current one.
              Iterator outPorts = lowerActor.outputPortList().iterator();
              while (outPorts.hasNext()) {
              IOPort outputPort = (IOPort)outPorts.next();
              Iterator inPorts =
              outputPort.deepConnectedInPortList().iterator();
              while (inPorts.hasNext()) {
              IOPort inputPort = (IOPort)inPorts.next();
              Actor higherActor = (Actor)inputPort.getContainer();
              if (dag.containsNodeWeight(higherActor)) {
              dag.addEdge(lowerActor, higherActor);
              }
              }
              }
              }
              return dag.topologicalSort();
}
```

FIGURE 4.14. An example of using a topological sort to generate a firing schedule for a CompositeActor of the actor package.

```
import ptolemy.graph.*;
import ptolemy.kernel.util.*;

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

   // Construct a constant term with the specified String value.
   public Constant(String value) {
      _value = value;
   }

    // Return the String associated with this term.
   public Object getAssociatedObject() {
      return _value;
   }

   // Return the constant String value of this term.
   public Object getValue() {
      return _value;
   }

   // Constant terms do not contain variables, so return an array of size zero.
   public InequalityTerm[] getVariables() {
      return new InequalityTerm[0];
   }

    // Initialize the value of this term to the specified CPO element.
    public void initialize(Object object) throws IllegalActionException {
      throw new IllegalActionException("Constant inequality term cannot be "
               + "initialized. Its value is set in the constructor.");
    }

   // Constant terms are not settable.
   public boolean isSettable() {
      return false;
   }

    // Check whether the current value of this term is acceptable.
    public boolean isValueAcceptable() {
        return _value != null;  // Any non-null string value is acceptable.
    }

   // Throw an Exception on an attempt to change this constant.
   public void setValue(Object e) throws IllegalActionException {
      throw new IllegalActionException("This term is a constant.");
   }

   // the String value of this term.
   private String _value = null;
}
```

FIGURE 4.15. A class that implements the InequalityTerm interface and models the constant term.

```
import ptolemy.graph.*;

// An example of forming and solving inequality constraints.
public class TestSolver {
   public static void main(String[] argv) {
      // construct the 4-point CPO in figure 2.3.
      CPO cpo = constructCPO();

      // create inequality terms for constants w, z and
      // variables a, b.
      InequalityTerm tw = new Constant("w");
      InequalityTerm tz = new Constant("z");
      InequalityTerm ta = new Variable();
      InequalityTerm tb = new Variable();

      // form inequalities: a<=w; b<=a; b<=z.
      Inequality iaw = new Inequality(ta, tw);
      Inequality iba = new Inequality(tb, ta);
      Inequality ibz = new Inequality(tb, tz);

      // create the solver and add the inequalities.
      InequalitySolver solver = new InequalitySolver(cpo);
      solver.addInequality(iaw);
      solver.addInequality(iba);
      solver.addInequality(ibz);

      // solve for the least solution
      boolean satisfied = solver.solveLleast();

      // The output should be:
      // satisfied=true, least solution: a=z b=z
      System.out.println("satisfied=" + satisfied + ", least solution:"
          + " a=" + ta.getValue() + " b=" + tb.getValue());
      // solve for the greatest solution
      satisfied = solver.solveGreatest();

      // The output should be:
      // satisfied=true, greatest solution: a=w b=z
      System.out.println("satisfied=" + satisfied + ", greatest solution:"
          + " a=" + ta.getValue() + " b=" + tb.getValue());
   }

   public static CPO constructCPO() {
      DirectedAcyclicGraph cpo = new DirectedAcyclicGraph();

      cpo.addNodeWeight("w");
      cpo.addNodeWeight("x");
      cpo.addNodeWeight("y");
      cpo.addNodeWeight("z");

      cpo.addEdge("x", "w");
      cpo.addEdge("y", "w");
      cpo.addEdge("z", "x");
      cpo.addEdge("z", "y");

      return cpo;
   }
}
```

FIGURE 4.16. An example that constructs the 4-point CPO of Figure 4.4.13.

5

Type System

Authors: *Edward A. Lee*
Steve Neuendorffer
Yuhong Xiong
Contributor: *Yang Zhao*

5.1 Introduction

The computation infrastructure provided by the basic actor classes is not statically typed, i.e., the IOPorts on actors do not specify the type of tokens that can pass through them. This can be changed by giving each IOPort a type. One of the reasons for static typing is to increase the level of safety, which means reducing the number of untrapped errors [27].

In a computation environment, two kinds of execution errors can occur, trapped errors and untrapped errors. Trapped errors cause the computation to stop immediately, but untrapped errors may go unnoticed (for a while) and later cause arbitrary behavior. Examples of untrapped errors in a general purpose language are jumping to the wrong address, or accessing data past the end of an array. In Ptolemy II, the underlying language Java is quite safe, so errors rarely, if ever, cause arbitrary behavior.[1] However, errors can certainly go unnoticed for an arbitrary amount of time. As an example, figure 5.1 shows an imaginary application where a signal from a source is downsampled, then fed to a fast Fourier transform (FFT) actor, and the transform result is displayed by an actor. Suppose the FFT actor

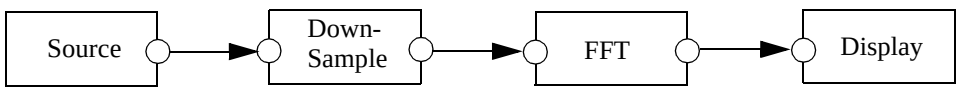


FIGURE 5.1. An imaginary Ptolemy II application

1. Synchronization errors in multi-thread applications are not considered here.

can accept ComplexToken at its input, and the behavior of the DownSample actor is to just pass every second token through regardless of its type. If the Source actor sends instances of ComplexToken, everything works fine. But if, due to an error, the Source actor sends out a StringToken, then the StringToken will pass through the sampler unnoticed. In a more complex system, the time lag between when a token of the wrong type is sent by an actor and the detection of the wrong type may be arbitrarily long.

In languages without static typing, such as Lisp and the scripting language Tcl, safety is achieved by writing defensive code. When safe execution is required, code must check manually at run-time whether the types of values are correct. In Ptolemy II, if we imitated this approach, we would have to require actors to check the type of the received tokens before using them. For example, the FFT actor would have to verify that the every received token is an instance of ComplexToken, or convert it to ComplexToken if possible. This approach places the burden of type checking on actor developers, distracting them from their development effort. It also relies on a policy that cannot be enforced by the system. Furthermore, since type checking is postponed to the last possible moment, the system does not have fail-fast behavior, so a system may generate an error message long after long after the error occurs, as illustrated in figure 5.1. To make matters worse, an actor may receive tokens from multiple sources. If a token with an incompatible type is received, it can be hard to identify the original source of the token. These potential problems can make debugging models unnecessarily difficult.

To address this and other issues discussed later, Ptolemy II includes static type checking. This approach is a significant extension of the simple type mechanism in Ptolemy Classic. In general-purpose statically-typed languages, such as C++ and Java, static type checking done by the compiler can find many potential program errors. However, execution of a model in Ptolemy II is more similar to an interpreted execution, and does not generally involve compilation. Nonetheless, static type checking of the model can still be used to detect modeling errors before actors fire. In figure 5.1, if the Source actor declares that its output port type is *string*, meaning that it will send out StringTokens upon firing, the static type checker will identify this type conflict in the topology.

In Ptolemy II, because actors may contain arbitrary Java code, static typing alone is not enough to ensure type safety at run-time. For example, even if the above Source actor declares its output type to be *complex*, it may still attempt to send out a StringToken at run-time. For instance, the Source actor might contain a bug that incorrectly declares the type of a port. Hence run-time type checking is still necessary for the Ptolemy framework to guarantee that all actors receive tokens of an expected type. Fortunately, with the help of static type checking, run-time type checks can be performed automatically when a token is sent out from a port. The run-time type checker simply compares the type of a produced token against the type of the output port. This way, a type error is detected at the earliest possible time and less reliance on correct actor specifications is needed to ensure type safety. Additionally, actors can safely cast received tokens to the type of the input port without manually checking the type, making actor development easier.

We have found that type checking and type safety conversions can greatly increase our confidence in making use of reusable components. However, static typing does have some drawbacks. For instance, it often requires actor authors to explicitly declare what type(s) of data are allowed, making it more difficult to develop components. Ousterhout [123] also argues that static typing discourages the reuse of existing components.

> *"Typing encourages programmers to create a variety of incompatible interfaces, each interface requires objects of specific type and the compiler prevents any other types of objects from being used with the interface, even if that would be useful".*

In this chapter we will concentrate on two mechanisms for increasing the reusability of actors in the presence of static type checking. The first mechanism, called automatic type conversion, allows a component to receive multiple data types by automatically converting them to a single data type. A second mechanism, called type resolution or type inference, allows constructing data-polymorphic actors. Such actors operate in a similar way on different data types. This chapter will describe how these mechanisms are integrated into the Ptolemy II static type checking framework.

One mechanism that enables polymorphism in Ptolemy II is automatic type conversion. The allowed automatic data type conversions are represented in figure 5.2, called the *type lattice*. In this diagram, a conversion from one type to another is allowed if the first type appears below the second type in the diagram. This relationship implies a partial ordering of types, so we might say that a conversion is allowed if the first type is less than or equal to the second type.

Automatic conversions primarily occur during data transfer from one port to another. When a data token is received, it is automatically converted to the type of the input port receiving it. Along with the run-time type checking of sent data described earlier, this conversion implies that across every connection from an output port to an input, the type of the output must be the same as or lower than the type of the input. This requirement is called the type compatibility rule. For example, an output port with type *int* can be connected to an input port with type *double*, and tokens sent by the output port will be converted to type *double* before being received. On the other hand, a *double* to *int* connection will gen-

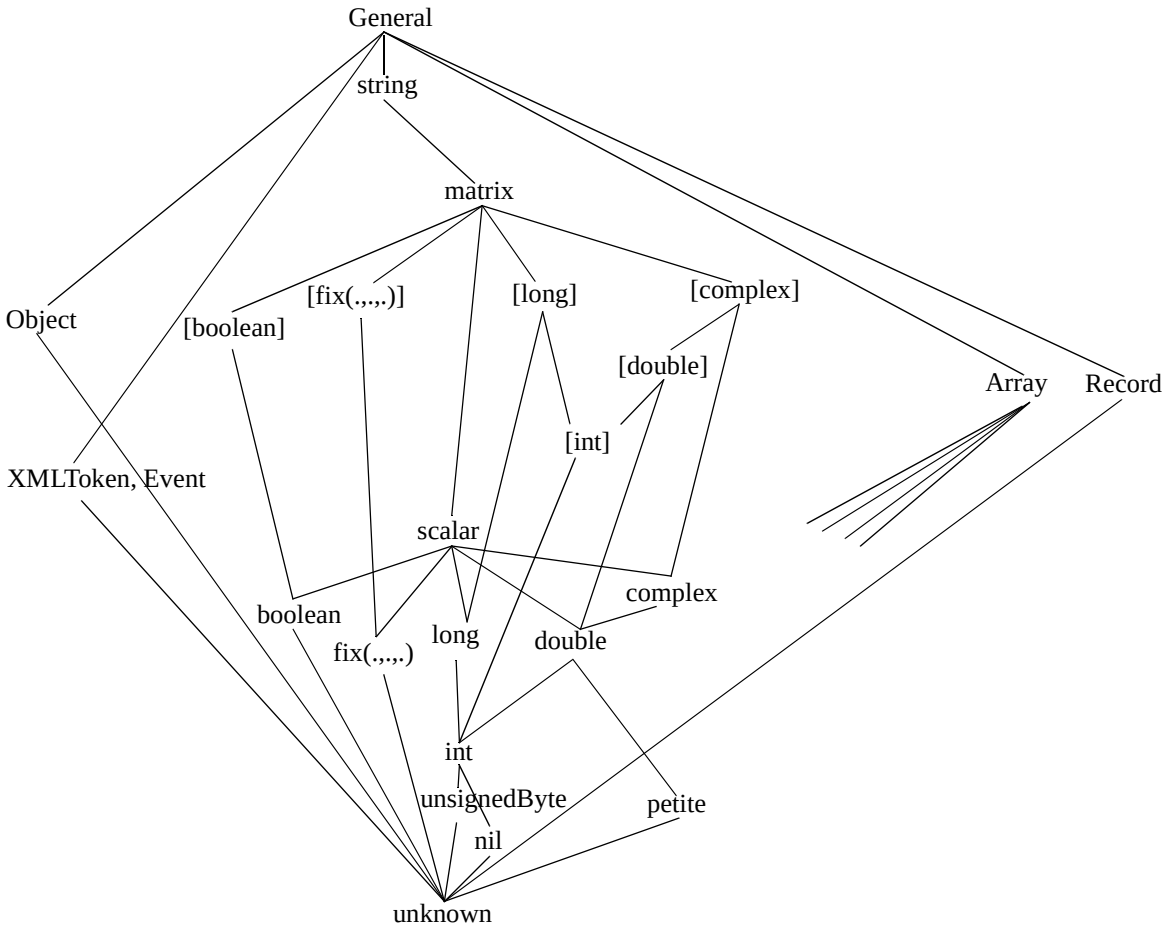


FIGURE 5.2. The Type Lattice

erate a type error during static type checking, since no conversion is possible. These conversions are performed transparently by the Ptolemy II system (actors are not aware it). Automatic conversions are also often performed in the data package when type-polymorphic operations are applied to values of different types.

The type lattice was constructed based on a principle of *lossless conversion*. A conversion is allowed automatically as long as important information about value of data tokens are not lost. Such conversions are referred to as *widening* conversions in Java. For instance, converting a 32-bit signed integer to a 64-bit IEEE double precision floating point number is allowed since every integer can be represented exactly as a floating point number. On the other hand, data type conversions that lose information are not included in the type lattice of automatic conversions. In fact, the concentration on lossless conversions is somewhat arbitrary, but we find that it is relatively easy to use, since it minimizes unintentional loss of numerical precision.

While automatic type conversion allows an actor to receive data of different types, the operation performed by the actor is always performed on the same type of data, determined by the type of the ports. However, There are cases where an actor operates on tokens without regard for the actual types of the tokens. For example, the DownSample in figure 5.1 does not care about the type of token going through it; it works with any type of token. In general, the types on some or all of the ports of a polymorphic actor are not rigidly defined to specific types when the actor is written, so the actor can interact with other actors having different types, increasing reusability.

In Ptolemy Classic, the ports on type-polymorphic actors whose types are not specified are said to have ANYTYPE. ANYTYPE ports were allowed to be connected to ports of any other type. However, in the presence of such ports means that type safety cannot be ensured. Instead, Ptolemy II allows ports to have *undeclared type*, suggesting that the type of those ports has not been determined but cannot be assigned arbitrarily. Instead of being given as constants, the acceptable types on polymorphic actors are described by a set of type constraints. The type checker checks the applicability of a type-polymorphic actor in a model by finding specific types for ports that satisfy the type constraints. This process is called *type resolution* or *type inference*, and the specific types are called the resolved types. Assuming the type constraints of actors are consistent with the actor implementation, this technique can ensure the type safety of actor connections. Type constraints and the type resolution algorithm are described more completely in the next section.

In addition to ports, the parameters which are used to configure actors are also typed objects. By defining a uniform interface for setting up type constraints, Ptolemy II supports type constraints between parameters and ports, as well as between ports. This extends the range of type checking to allow parameters with arbitrary type, such as those that determine the values produced by source actors.

In Ptolemy II, typing does apply some restrictions on the interaction of actors. Particularly, actors cannot be interconnected arbitrarily if the type compatibility rule is violated. However, such models rarely make any sense, so the benefit of typing should far outweigh the inconvenience caused by this restriction. On the other hand, type declarations and type constraints help to clarify the interface of actors and makes them more manageable. Static typing also provide an opportunity for model compiler and circuit synthesis tools to generate type specialized code, when a Ptolemy system is synthesized to hardware, type information can be used for efficient synthesis. If the type checker asserts that a certain polymorphic actor will only receive IntTokens, then only hardware dealing with integers needs to be synthesized.

To summarize, Ptolemy II takes an approach of static typing coupled with run-time type checking. Lossless data type conversions during data transfer are automatically executed. Polymorphic actors are

supported through type resolution.

5.2 Formulation

5.2.1 Type Constraints

In a Ptolemy II topology, the type compatibility rule imposes a type constraint across every connection from an output port to an input port. It requires that the type of the output port, *outType*, be the same as the type of the input port, *inType*, or less than *inType* under the type lattice. This can be written as an inequality:

$$outType \leq inType \tag{5}$$

This constraint guarantees that there is an allowed automatic conversion that can be performed during data transfer. If both the *outType* and *inType* are declared, the static type checker simply checks whether this inequality is satisfied, and reports a type conflict if it is not.

In addition to the above constraint imposed by the topology, actors may also impose constraints. This happens when one or both of the *outType* and *inType* is undeclared, in which case the actor containing the undeclared port needs to describe the acceptable types through type constraints. All the type constraints in Ptolemy II are described in the form of inequalities like the one in (5). If a port or parameter has a declared type, its type appears as a constant in the inequalities. On the other hand, if a port or parameter has an undeclared type, its type is represented in the inequalities by a variable, called a type variable. The value of type variables are allowed to range over the elements of the type lattice. The type resolution algorithm resolves the values of type variables subject to the constraints of the model and the actors. If no solution exists, a type conflict error will be reported. As an example of the inequality constraints, consider figure 5.3.

The port of actor A1 has declared type *int* and the ports of A3 and A4 have declared type *double*. The types of the ports of A2, on the other hand, have been left undeclared. If the type variables of the undeclared types are α, β, and γ, then the type constraints from the topology are:

$int \leq \alpha$

$double \leq \beta$

$\gamma \leq double$

Now, assume A2 is a polymorphic adder, capable of doing addition for integer, double, and complex numbers, and the requirement is that it does not lose precision during the operation. Then the type constraints for the adder can be written as:

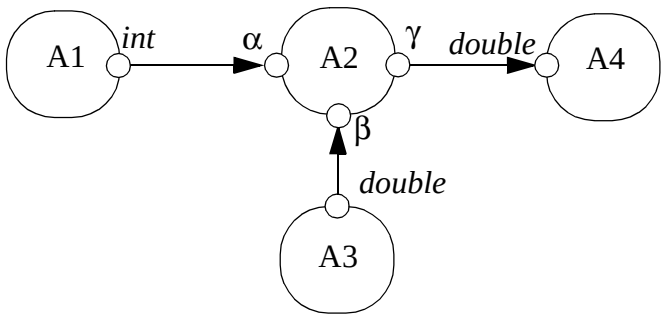


FIGURE 5.3. A topology with types.

$\alpha \leq \gamma$
$\beta \leq \gamma$
$\gamma \leq complex$

The first two inequalities constrain the output precision to be no less than input, the last one requires that the data on the adder ports can be converted to *complex* losslessly. These six inequalities form the complete set of constraints and are used by the type resolution algorithm to solve for α, β, and γ. Hence, the problem has been converted from type resolution into a problem of solving a set of inequalities. An efficient algorithm is available to solve constraints in finite lattices [129], which is described in the appendix through an example. This algorithm finds the set of most specific types for the undeclared types in the topology that satisfy the constraints, if they exist.

This inequality formulation is inspired by the type inference algorithm in ML [109]. There, equalities are used to represent type constraints. In Ptolemy II, the lossless type conversion hierarchy naturally implies inequality relation among the types. In ML, the type constraints are generated from program constructs. In a heterogeneous graphical programming environment like Ptolemy II, the system does not have enough information about the function of the actors, so actors must specify type information either by declaring port types, or by providing type constraints to describe the acceptable types of undeclared ports.

As mentioned earlier, the static type checker flags a type conflict error if the type compatibility rule is violated on a certain connection. There are other kind of type conflicts indicated by one of the following:

- The set of type constraints are not satisfiable.
- Some type variables are resolved to *unknown*.
- Some type variables are resolved to an abstract type, such as *Numerical* in the type hierarchy.

The first case can happen, for example, if the port of actor A1 in figure 5.3 has declared type *complex*. The second case can happen if an actor does not specify any type constraints on an undeclared output port. This is due to the nature of the type resolution algorithm where it assigns all the undeclared types to *unknown* at the beginning. If the type constraints do not restrict a type variable to be greater than *unknown*, it will stay at *unknown* after resolution. The third case is considered a conflict since an abstract type does not correspond to an instantiable token class.

5.2.2 Run-time Type Checking and Lossless Type Conversion

The declared type is a contract between an actor and the Ptolemy II system. If an actor declares an output port to have a certain type, it asserts that it will only send out tokens whose types are less than or equal to that type. If an actor declares an input port to have a certain type, it requires the system to only send tokens that are instances of the class of that type to that input port. Run-time type checking enforces this contract, regardless of whether individual actors respect it. When a token is sent out from an output port, the run-time type checker queries its type and compares the type with the declared type of the output port. If the type of the token is not less than or equal to the declared type, a run-time type error will be generated.

As discussed before, type conversion is performed automatically when a token sent to an input port has a type less than the type of the input port. This conversion enables an actor to safely cast a received token to the type of the port. On the other hand, when an actor sends out tokens, the tokens being sent do not have to have the exact declared output port type. Any type that is less than the declared type is acceptable. For example, if an output port has declared type *double*, the actor can send *IntToken* from that port. As can be seen, the automatic type conversion simplifies the input/output han-

dling of the actors.

Note that even with the convenience provided by the type conversion, actors should still declare the input types to be the most general that they can handle and the output types to be the most specific type that includes all tokens they will send. This maximizes their applications. In the previous example, if the actor only sends out *IntToken*, it should declare the output type to be *int* to allow the port to be connected with an input with type *int*.

If an actor has ports with undeclared types, its type constraints can be viewed as both a requirement and an assertion from the actor. The actor requires the resolved types to satisfy the constraints. Once the resolved types are found, they serve exactly the same role as declared types at run time. The the type checking and type conversion system guarantees the type of tokens received by an actor, and the actor guarantees the types of tokens sent by the actor. These assumptions and guarantees are summarized for all possible types by the type constraints of the actor.

5.3 Structured Types

Structured types include those tokens which aggregate other tokens of arbitrary type, such as array and record types. As described in the Data Package chapter, an ArrayToken contains an array of tokens, and the element tokens can have arbitrary type. For example, an ArrayToken can contain an array of StringTokens, or an array of ArrayTokens. In the latter case, the ArrayToken can be regarded as a two dimensional array. RecordToken contains a set of labeled tokens, like the structure in the C language. It is useful for grouping multiple pieces of related information together. In the type lattice in figure 5.2, record types are incomparable with all the base types, except the top and the bottom elements of the lattice. Array types are a bit more complex because any type is less than an array of that type in the type lattice. This is hinted at in the figure with the disconnected lines at the bottom of the array type. Note that the lattice nodes Array and Record actually represent an infinite number of types, so the type lattice becomes infinite.

The order relation between two array types is that type {*B*} (the type of arrays containing elements of type *B*) is less than type {*A*} if *B* is less than *A*, or if *A* is an array of elements of type *B*. This is a recursive definition if the element types *A* and *B* are themselves structured types. For example, {*int*} ≤ {*double*}, {{*int*}} ≤ {{*double*}}, where {{*int*}} is an array of array. Moreover, {*int*} ≤ {{*double*}}.

The order relation between two record types follows the standard depth subtyping and width subtyping relations [27]. In depth subtyping, a record type *C* is a subtype of a record type *D* if the type of some fields of *C* is a subtype of the corresponding fields in *D*. In width subtyping, a record with more fields is a subtype of a record with less fields. For example, we have:

{x = *string*, y = *int*} ≤ {x = *string*, y = *double*}
{x = *string*, y = *double*, z = *int*} ≤ {x = *string*, y = *double*}

Here, we use the {label = type, label = type, ...} syntax to denote record types. Notice that the width subtyping rule implies a type conversion which loses information, discarding the extra fields of a record.

Another structured type is the union type. It allows the user to create a token that can hold data of various types, but only one at a time. This is like the union construct in C. The union type is also called variant types in the type system literature. The width subtyping relation for union type is the opposite to that of the record type. That is, a shorter union is a subtype of a longer one. This means that there are

an infinite number of types from a particular union type to the top of the type lattice, so the convergence of the type resolution algorithm is not immediately obvious. We are currently addressing this issue and working on the implementation of the union type in Ptolemy II.

One final structured type is the type of function closures. Each function closure is represented by an instance of the FunctionToken class. Function closures take several arguments and return a single value. The type system supports function types where the arguments have declared types, and the return type is known. Function types are related in a way that is contravariant (oppositely related) between inputs and outputs. Namely, if function(x:*int*, y:*int*) *int* is a function that of two integer arguments that returns an integer, then

function(x:*int*, y:*int*) *int* ≤ function(x:*int*, y:*int*) *double*

function(x:*int*, y:*double*) *int* ≤ function(x:*int*, y:*int*) *int*

The contravariant notion here is easiest to think about in terms of the automatic type conversion of one function into another. A function that returns *int* can be converted into a function that returns *double* by adding a conversion of the returned value from *int* to *double*. On the other hand, a function that takes an *int* cannot be converted into a function that takes a *double*, since that would mean that the function is suddenly able to accept *double* arguments when it could not before, and there is no automatic conversion from *double* to *int*. Functions that are lower in the type lattice *assume less* about their inputs and *guarantee more* about their outputs. Note particularly that the names of arguments do not affect the relation between two function types, since argument binding is by the order of arguments only. Additionally, functions with different numbers of arguments are considered incomparable. Eventually, we intend to provide an actor token as well, which would have both contravariance of the types of input and output ports as well as allowing width subtyping, similarly to records. The presence of function types that can be used as any other token results in what is commonly termed a *higher-order* type system.

Type constraints can be specified between the element type of a structured type and the type of a Ptolemy object. For example, a type constraint can specify that the type of a port is no less than the type of the elements of an ArrayToken.

5.3.1 Setting Up Type Constraints

In most cases, type constraints can be set up easily through the methods in the Typeable interface. The setTypeAtMost() method is usually invoked on input ports to declare a requirement that input tokens must satisfy, while the setTypeAtLeast() method is usually invoked on output ports to declare a guarantee of the type of the output. The setTypeEquals() method can also be used to force the type of typeable objects to a particular data type. As an example, the constraint that the type of an input port can be no greater than *double* might be declared as:

```
inputPort.setTypeAtMost(BaseType.DOUBLE);
```

and a constraint that the type of an output port can be no less than the type of a parameter:

```
outputPort.setTypeAtLeast(parameter);
```

This latter type constraint is commonly seen when parameter values are used to compute values produced from an output port. To declare that a parameter is an array of doubles, use

```
parameter.setTypeEquals(new ArrayType(BaseType.DOUBLE));
```

Notice that the argument to setTypeAtMost() and setTypeEquals() is a Type, whereas the argument to setTypeAtLeast() is a Typeable object. This reflects the common usages, where setTypeAtLeast() is declaring a dependency on externally provided types, whereas setTypeAtMost() and setTypeEquals() are declaring constraints on externally defined types.

More complex type constraints arise from structured types, such as arrays and records. The previous example showed how to declare that a parameter or a port has a particular array type. A more flexible parameter might be able to contain an array of any type. This is expresses as follows,

```
parameter.setTypeAtLeast(ArrayType.ARRAY_BOTTOM);
```

In a more elaborate example, we might constrain the type of an output port to be no less than the element type of the array contained by a parameter (or an input port):

```
outputPort.setTypeAtLeast(ArrayType.arrayOf(parameter));
```

To declare that an output port has a type at least that of the elements of input array (or parameter), use

```
outputPort.setTypeAtLeast(ArrayType.elementType(inputPort));
```

The above code implicitly constrains the input port to have an array type, but the constrain the element types of that array.

The above kinds of constraints appear in source actors such as Clock and Pulse, and ArrayToElements and ElementsToArray.

Another common constraint is that an input port of an actor receives a RecordToken with unconstrained fields. This constraint can be declared using the following code:

```
String[] labels = new String[0];
Type[] types = new Type[0];
RecordType declaredType = new RecordType(labels, types);
inputPort.setTypeAtMost(declaredType);
```

Two of the types, *matrix* and *scalar*, are union types. This means that an instance of this type can be any of the types immediately below them in the lattice. An actor may, for example, declare that an input port must be of type no greater than scalar by using the following code,

```
inputPort.setTypeAtMost(BaseType.SCALAR);
```

In this case, inputs of any type immediately below *scalar* in the type lattice will not be converted, except that the type of the input tokens will be reported as *scalar*. This is useful, for example, in actors that need to compare tokens, such as the Limiter actor. The fire() method of that actor contains the code

```
if (input.hasToken(0)) {
    ScalarToken in = (ScalarToken) input.get(0);
```

```
        if ((in.isLessThan((ScalarToken) bottom.getToken()))
                .booleanValue()) {
            output.send(0, bottom.getToken());
        } else if ((in.isGreaterThan((ScalarToken) top.getToken()))
                .booleanValue()) {
            output.send(0, top.getToken());
        } else {
            output.send(0, in);
        }
    }
```

This code relies on the fact that input port *in* and parameter *bottom* have been declared to be at most *scalar* type, and that ScalarToken is a base class for every token with type immediately below *scalar*. It then uses comparison methods defined in the ScalarToken class.

Internally, the class Inequality in the graph package is used to represent type constraints. This class references two objects implementing the InequalityTerm interface, one for each side of the inequality. The InequalityTerm interface is implemented by inner classes of TypedIOPort, Variable, ArrayType, and RecordType, to encapsulate the type of the port, the variable, and the element type of structured types. For some more elaborate type constraints, the actor programmer can use these classes directly. Specifically, in some actors, simple constraints between variables are not capable of representing the type constraints between ports and parameters. In such cases, monotonic functions can be used to specify more complex type constraints. That is, constraints in the form $f(\alpha) \leq \beta$ are admitted, where $f(\alpha)$ is a monotonic function of α, and β can be a constant or a variable. An example of this appears in the AbsoluteValue actor in the actor library. Here, one of the type constraints is: If the input type is not *complex*, the output type is the same as the input type, otherwise, the output type is *double*. This constraint can be expressed as f(inputType) ≤ outputType, where

```
f(inputType)  = inputType,  if inputType ≠ complex
f(inputType)  = double,     if inputType = complex.
```

This function is implemented by an inner class of AbsoluteValue that implements InequalityTerm. The evaluation is done in the getValue() method of InequalityTerm as:

```
public Object getValue() {
   // _port is the input port
   Type inputType = _port.getType();
   return inputType == BaseType.COMPLEX ? BaseType.DOUBLE : inputType;
}
```

Directly implementing the InequalityTerm interface is actually rather complex, and is implemented in the same pattern for all monotonic function constraints. The MonotonicFunction base class, which implements the uninteresting parts of the InequalityTerm interface, allows actors to easily implement new monotonic function constraints. Lastly, if the methods in Typeable are not sufficient for specifying complicated constraints, or the default implementation of the typeConstraints() method in the TypedAtomicActor is not appropriate, this method can be overridden, but this is rarely needed.

5.4 Implementation

5.4.1 Implementation Classes

All the classes for representing the types and the type lattice are under the data.type package, as shown in figure 5.4. The Type interface defines the basic operations on a type. BaseType contains a

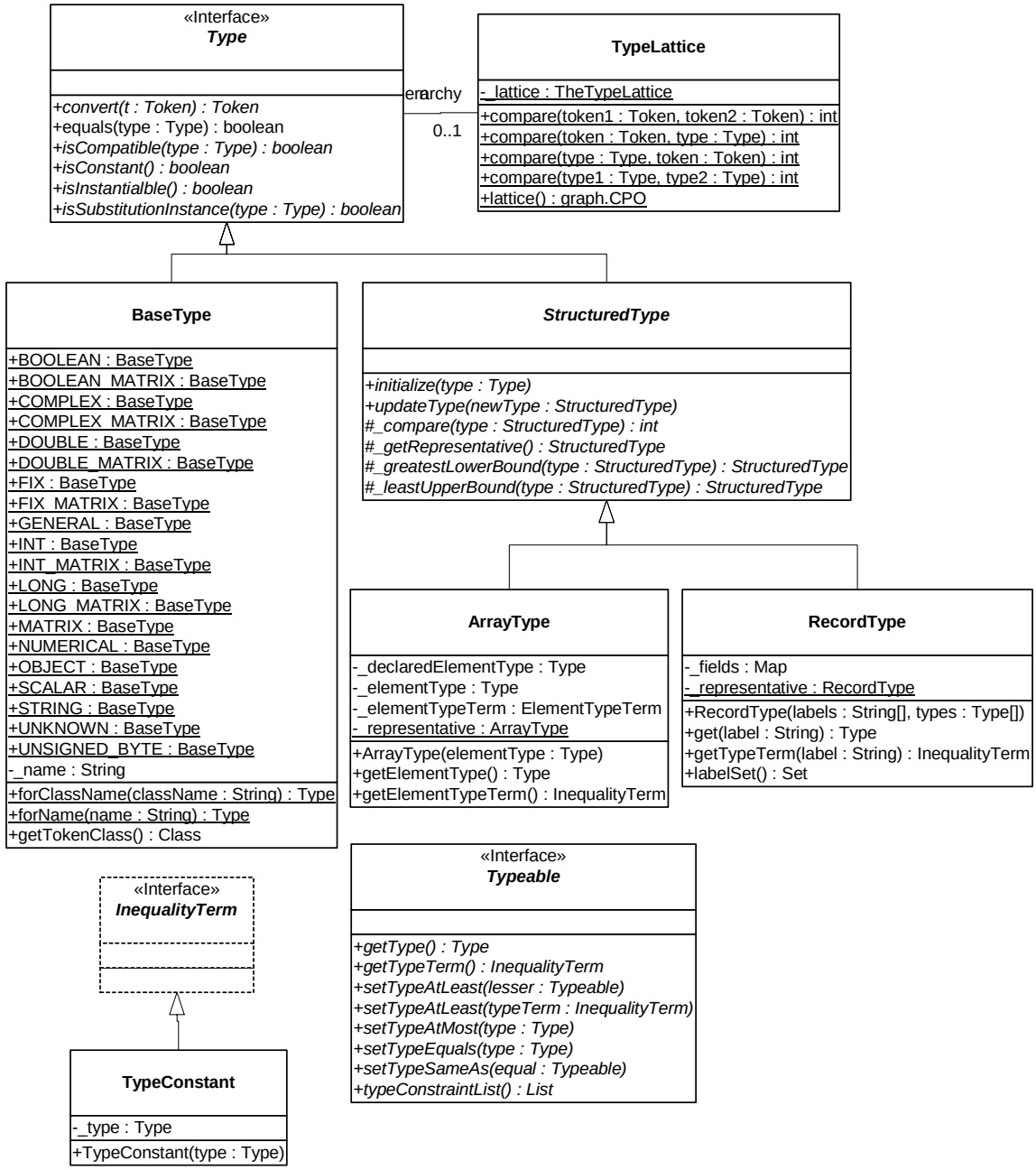


FIGURE 5.4. Classes in the data.type package.

type-safe enumeration of primitive types. For example, *unknown*, the bottom element of the type lattice which can be resolved to any type is represented by the field BaseType.UNKNOWN. ArrayType and RecordType are derived from an abstract class StructuredType. Each type has a convert() method to convert a token lower in the type lattice to one of its type. For base types, this method just calls the same method in the corresponding tokens. For structured types, the conversion is done within the concrete structured type classes.

The Typeable interface defines a set of methods to set type constraints between typed objects. It is implemented by the Variable class in the data.expr package and the TypedIOPort class in the actor package. The TypeConstant class encapsulates a constant type. It implements the InequalityTerm interface and can be used to set up type constraints between a typed object and a constant type.

In the actor package, the Actor interface, the AtomicActor, CompositeActor, IOPort and IORelation classes are extended with TypedActor, TypedAtomicActor, TypedCompositeActor, TypedIOPort and TypedIORelation, respectively, as shown in figure 5.5. The container for TypedIOPort must be a ComponentEntity implementing the TypedActor interface, namely, TypedAtomicActor or TypedCompositeActor. The TypedIORelation class is only able to connect instances of the TypedIOPort. TypedIOPort has a declared type and a resolved type. Declaring a type of BaseType.UNKNOWN allows the type system to infer the resolved type of a port. If a port has a declared type that is not BaseType.UNKNOWN, the resolved type will be the same as the declared type.

5.4.2 Type Checking and Type Resolution

Static type checking and type resolution are performed by the resolveTypes() method of the TypedCompositeActor class. This method finds all connections within the composite by first finding the output ports on deep contained entities, and then finding input ports deeply connected to those output ports. Transparent ports are ignored for type checking. For each connection, if the types on both ends are declared, static type checking is performed using the type compatibility rule. If the model contains other opaque TypedCompositeActors, this method recursively calls the _checkDeclaredTypes() method of the contained actors to perform type checking on the entire hierarchy. Hence, if resolveTypes() is called with the top level TypedCompositeActor, type checking is performed through out the hierarchy.

If a type conflict is detected, i.e., if the declared type at the source end of a connection is greater than or incomparable with the type at the destination end of the connection, then the ports at both ends of the connection are recorded and will be returned in a List at the end of type checking. Note that type checking does not stop after detecting the first type conflict, so the returned List contains all the ports that have type conflicts. This behavior is similar to a regular compiler, where compilation will generally continue after detecting errors in the source code.

The TypedActor interface declares a typeConstraintList() method, which returns the type constraints of this actor. The TypedAtomicActor base class provides a default implementation of this method, which requires that the type of any input port with undeclared type must be less than or equal to the type of any undeclared output port. Ports with declared types are not included in the default constraints. If all of an actor's ports have declared types, no constraints are generated. This default is appropriate for many type-polymorphic actors such as the Commutator actor, the Multiplexer actor, and the DownSample actor in figure 5.1. In addition, the typeConstraintList() method also collects all the constraints from the contained Typeable objects, which are TypedIOPorts and Variables.

The typeConstraintList() method in TypedCompositeActor collects all the constraints for a model, including the constraints for actors and the constraints for connections between actors. It works in a

similar fashion as the _checkDeclaredTypes() method, by recursively traversing the containment hierarchy. It also scans all the connections and forms additional type constraints on connections involving undeclared types. As with _checkDeclaredTypes(), if this method is called on the top level container, all the type constraints within the entire model are returned.

The Manager class has a resolveTypes() method that performs both type checking and resolution. It uses the InequalitySolver class in the graph package to solve the constraints. If type conflicts are

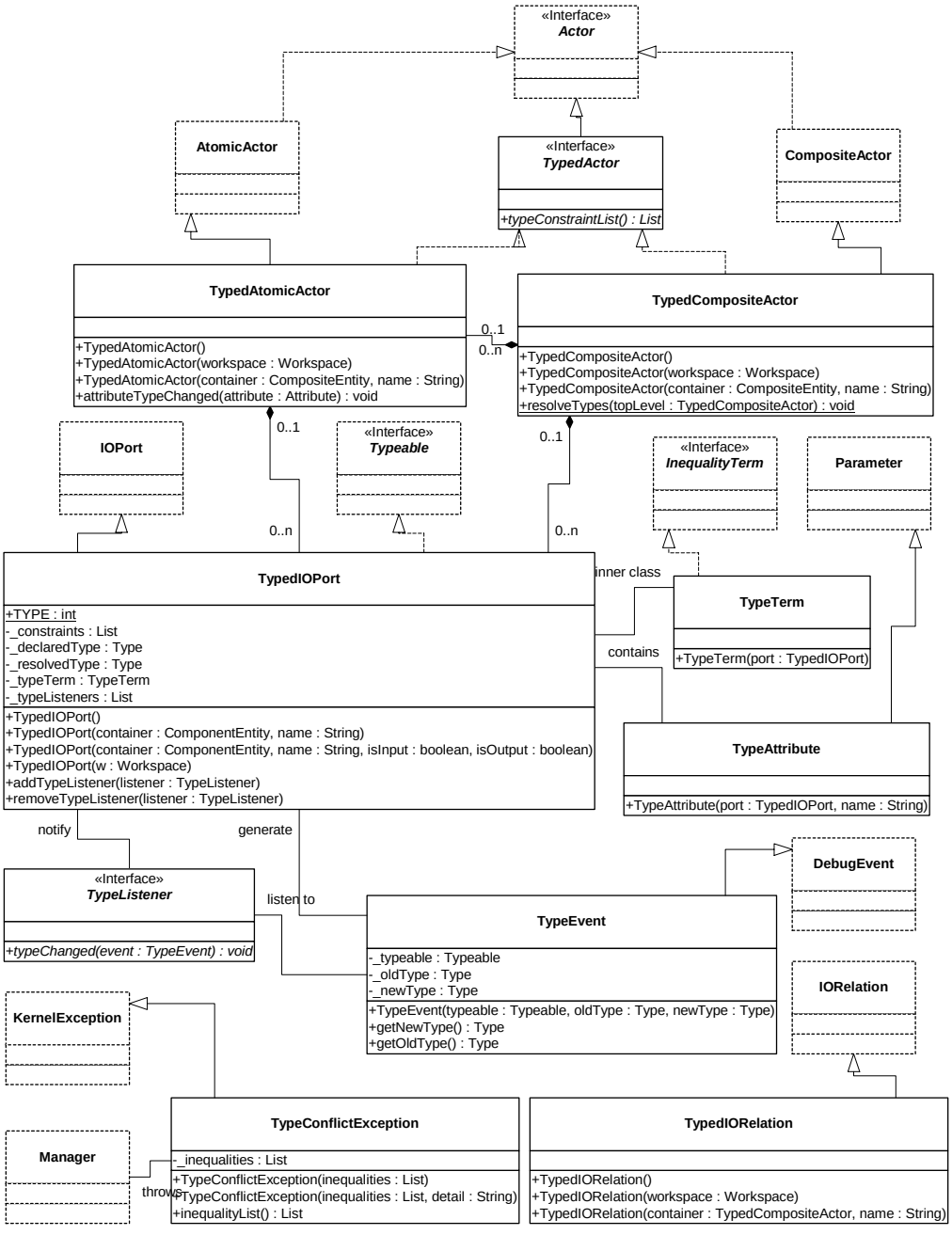


FIGURE 5.5. Classes in the actor package that support type checking.

detected during type checking or after type resolution, this method throws a TypeConflictException. This exception contains a list of inequalities where type conflicts occurred. The resolveTypes() method is invoked by the Manager of a model between the preinitialize() and initialize() phases, and after any mutations are processed.

Run-time type checking is performed in the send() method of TypedIOPort. The checking is simply a comparison of the type of the token being sent with the resolved type of the port. If the type of the token is less than or equal to the resolved type, type checking is passed, otherwise, an IllegalActionException is thrown.

Type conversion, if needed, is also done in the send() method. The type of the destination port is the resolved type of the port containing the receivers that the token is sent to. If the token does not have that type, the convert() method on that type is called to perform the conversion.

5.4.3 Some Implementation Details

The implementation of the structured types is more involved than the base types. This is because the base types are atomic, but structured types that contain type variables are mutable entities. For example, the declared type of a port can be *{unknown}*, meaning that it is an array of undefined element type. After type resolution, that type may be updated to *{double}*. Types that are mutable are variable types. The isConstant() method in Type determines if a type contains a type variable. Type variables are represented by a type initialized to BaseType.UNKNOWN.

When a typed object is cloned, if its type is a variable structured type, that type must be cloned because the original and the cloned Typeable objects may have different types in the future. Similarly, when constructing structured types with variable structured types as element types, the element types must be cloned. However, constant structured types do not need to be cloned. This means that an instance of a constant StructuredType can be shared by many objects, but an instance of a variable StructuredType can only have one user. To ensure this, structured types are always cloned when ports and parameters that contain them are cloned. This incurs some redundant cloning, but the overhead is small.

A variable type can be updated to another type, provided that the new type is compatible with the variable type. For example, a type variable α can be updated to any type, *{α}* can be updated to *{int}*. However, *{α}* cannot be updated to *int*. If a variable type can be updated to a new type, the new type is called a substitution instance of the variable type. This term is borrowed from type literature. Formally, a type is a substitution instance of a variable type if the former can be obtained by substituting the type variables of the latter to another type. The method isSubstitutionInstance() in the Type base class performs this check.

The updateType() method in StructuredType is used to change the variable element type of a structured type. For example, if the types of two ports are *{int}* and *{α}* respectively, and a type constraint is that the second port is no less than the type of the first, that is, *{int}* $\leq$ *{α}*, the type resolution algorithm will change the resolved type of the second port to *{int}*. This step cannot be done by simply changing the type reference in the second port to an instance of *{int}*, since type constraints may be set up between α and another typed objects. Instead, updateType() only changes the type reference for α to *int*.

5.5 Examples

5.5.1 Polymorphic DownSample

In figure 5.1, if the DownSample is designed to do downsampling for any kind of token, its type constraint is just *samplerIn* ≤ *samplerOut*, where *samplerIn* and *samplerOut* are the types of the input and output ports, respectively. The default type constraints works in this case. Assuming the Display actor just calls the *toString()* method of the received tokens and displays the string value in a certain window, the declared type of its port would be *General*. Let the declared types on the ports of FFT be *complex*, the The type constraints of this simple application are:

sourceOut ≤ *samplerIn*

samplerIn ≤ *samplerOut*

samplerOut ≤ *complex*

complex ≤ *General*

Where *sourceOut* represents the declared type of the Source output. The last constraint does not involve a type variable, so it is just checked by the static type checker and not included in type resolution. Depending on the value of *sourceOut*, the ports on the DownSample actor would be resolved to different types. Some possibilities are:

- If *sourceOut* = *complex*, the resolved types would be *samplerIn* = *samplerOut* = *complex*.
- If *sourceOut* = *double*, the resolved types would be *samplerIn* = *samplerOut* = *double*. At runtime, DoubleTokens sent out from the Source will be passed to the DownSample actor unchanged. Before they leave the Downsample actor and are sent to the FFT actor, they are converted to ComplexTokens by the system. The ComplexToken output from the FFT actor are instances of Token, which corresponds to the *General* type, so they are transferred to the input of the Display without change.
- If *sourceOut* = *string*, the set of type constraints do not have a solution, a typeConflictException will be thrown by the static type checker.

5.5.2 Fork Connection

Consider two simple topologies in figure 5.6. where a single output is connected to two inputs in 5.6(a) and two outputs are connected to a single input in 5.6(b). Denote the types of the ports by *a1, a2,*

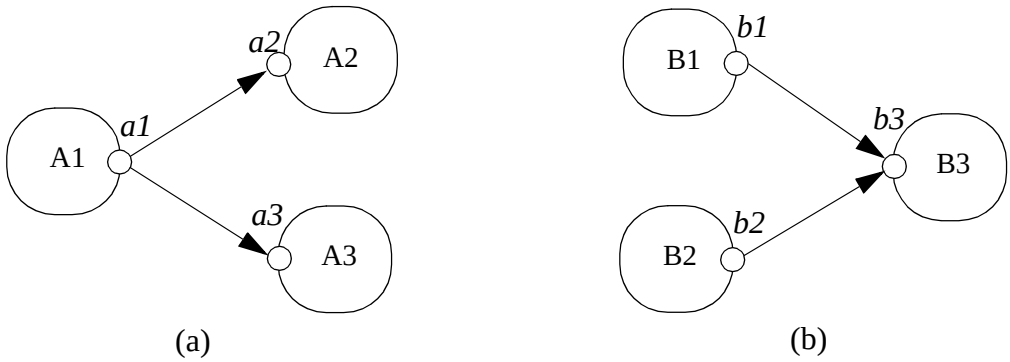


FIGURE 5.6. Two simple topologies with types.

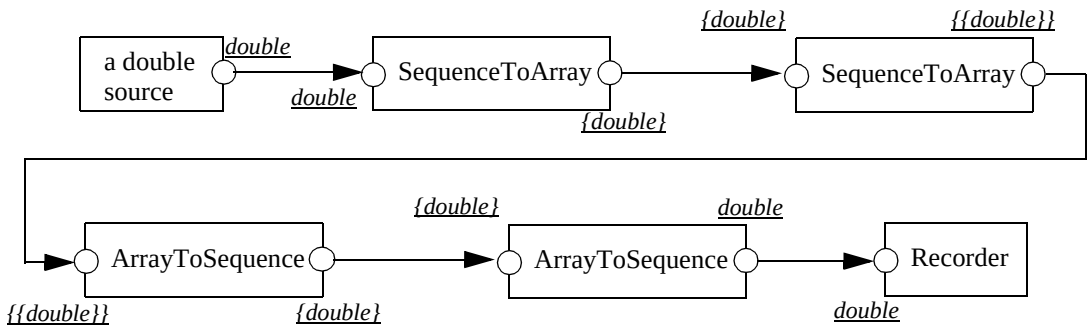


FIGURE 5.7. Conversion between sequence and array.

a3, b1, b2, b3, as indicated in the figure. Some possibilities of legal and illegal type assignments are:

- In 5.6(a), if *a1 = int*, *a2 = double*, *a3 = complex*. The topology is well typed. At run-time, the IntToken sent out from actor A1 will be converted to DoubleToken before transferred to A2, and converted to ComplexToken before transferred to A3. This shows that multiple ports with different types can be interconnected as long as the type compatibility rule is obeyed.
- In 5.6(b), if *b1 = int*, b2 = *double*, and *b3* is undeclared. The the resolved type for *b3* will be *double*. If *b1 = int* and *b2 = boolean*, the resolved type for *b3* will be *string* since it is the lowest element in the type hierarchy that is higher than both *int* and *boolean*. In this case, if the actor B3 has some type constraints that require *b3* to be less than *string*, then type resolution is not possible, a type conflict will be signaled.

5.6 Actors Constructing Tokens with Structured Types

The SDF domain contains two actors that perform conversion between a sequence of tokens and an ArrayToken. Type constraints in these actors ensure that the type of the array element is the same as the type of the sequence tokens. When two SequenceToArray actors are cascaded, the output of the second actor will be an array of array. Cascading ArrayToSequence with SequenceToArray restores the sequence. In these actors, the *arrayLength* parameter determines the size of the produced or consumed array, and also determines the number of tokens produced or consumed in each firing. If the ArrayToken received by ArrayToSequence does not have specified length and the *enforceArrayLength* parameter is true, an exception will be thrown.

The actor.lib package contains two actors that assemble and disassemble RecordTokens: RecordAssembler and RecordDisassembler. The former assembles tokens from multiple input ports into a RecordToken and sends it to the output port, the latter does the reverse. The labels in the RecordToken are the names of the input ports. Type constraints ensure that the type of the record fields is the same as the type of the corresponding ports.

Appendix B: The Type Resolution Algorithm

The type resolution algorithm starts by assigning all the type variables the bottom element of the type hierarchy, *unknown*, then repeatedly updating the variables to a greater element until all the constraints are satisfied, or when the algorithm finds that the set of constraints are not satisfiable. The kind of inequality constraints the algorithm can determine satisfiability are the ones with the greater term (the right side of the inequality) being a variable, or a constant. The algorithm allows the left side of the inequality to contain monotonic functions of the type variables, but not the right side. The first step of the algorithm is to divide the inequalities into two categories, *Cvar* and *Ccnst*. The inequalities in *Cvar* have a variable on the right side, and the inequalities in *Ccnst* have a constant on the right side. In the example of figure 5.3, *Cvar* consists of:

$int \leq \alpha\upsilon\nu\kappa\nu o\omega\nu$

$double \leq \beta$

$\alpha \leq \gamma$

$\beta \leq \gamma$

And *Ccnst* consists of:

$\gamma \leq double$

$\gamma \leq complex$

The repeated evaluations are only done on *Cvar*, *Ccnst* are used as checks after the iteration is finished, as we will see later. Before the iteration, all the variables are assigned the value *unknown*, and *Cvar* looks like:

$int \leq \alpha(unknown)$

$double \leq \beta(unknown)$

$\alpha(unknown) \leq \gamma(unknown)$

$\beta(unknown) \leq \gamma(unknown)$

Where the current value of the variables are inside the parenthesis next to the variable.

At this point, *Cvar* is further divided into two sets: those inequalities that are not currently satisfied, and those that are satisfied:

Not-satisfied	Satisfied
$int \leq \alpha(unknown)$	$\alpha(unknown) \leq \gamma(unknown)$
$double \leq \beta(unknown)$	$\beta(unknown) \leq \gamma(unknown)$

Now comes the update step. The algorithm takes out an arbitrary inequality from the Not-satisfied set, and forces it to be satisfied by assigning the variable on the right side the least upper bound of the values of both sides of the inequality. Assuming the algorithm takes out $int \leq \alpha(unknown)$, then

$$\alpha = int \vee unknown = int \tag{6}$$

After α is updated, all the inequalities in *Cvar* containing it are inspected and are switched to either the Satisfied or Not-satisfied set, if they are not already in the appropriate set. In this example, after this step, *Cvar* is:

Not-satisfied	Satisfied
$double \leq \beta(unknown)$	$int \leq \alpha(int)$
$\alpha(int) \leq \gamma(unknown)$	$\beta(unknown) \leq \gamma(unknown')$

The update step is repeated until all the inequalities in *Cvar* are satisfied. In this example, β and γ

will be updated and the solution is:

$$\alpha = int, \ \beta = \gamma = double$$

Note that there always exists a solution for *Cvar*. An obvious one is to assign all the variables to the top element, *General*, although this solution may not satisfy the constraints in *Ccnst*. The above iteration will find the least solution, or the set of most specific types.

After the iteration, the inequalities in *Ccnst* are checked based on the current value of the variables. If all of them are satisfied, a solution to the set of constraints is found.

This algorithm can be viewed as repeated evaluation of a monotonic function, and the solution is the fixed point of the function. Equation (6) can be viewed as a monotonic function applied to a type variable. The repeated update of all the type variables can be viewed as the evaluation of a monotonic function that is the composition of individual functions like (6). The evaluation reaches a fixed point when a set of type variable assignments satisfying the constraints in *Cvar* is found.

Rehof and Mogensen [129] proved that the above algorithm is linear time in the number of occurrences of symbols in the constraints, and gave an upper bound on the number of basic computations. In our formulation, the symbols are type constants and type variables, and each constraint contains two symbols. So the type resolution algorithm is linear in the number of constraints.

6

Plot Package

Authors: *Christopher Brooks*
Edward A. Lee
Contributors: *Lukito Muliadi*
William Wu
Jun Wu

6.1 Overview

The plot package provides classes, applets, and applications for two-dimensional graphical display of data. It is available in a stand-alone distribution, or as part of the Ptolemy II system.

There are several ways to use the classes in the plot package:

- You can use one of several domain-polymorphic actors in a Ptolemy II model to plot data that is provided as an input to the actor.
- You can invoke an executable, `ptplot`, which is a shell script, to plot data in a local file or on the network (via a URL).
- You can invoke an executable, `histogram`, which is a shell script, to plot histograms of data in a local file or on the network (via a URL)
- You can invoke an executable, `pxgraph`, which is a shell script, to plot data that is stored in an ascii or binary format compatible with the older program `pxgraph`, which is an extension of David Harrison's `xgraph`.
- You can invoke a Java application, such as PlotMLApplication, by using the `java` program that is included in your Java distribution.
- You can use an existing applet class, such as PlotMLApplet, in an HTML file. The applet parameter `dataurl` specifies the source of plot data. You do not even have to have Ptplot installed on your server, since you can always reference the Berkeley installation.
- You can create new classes derived from applet, frame, or application classes to customize your

plots. This allows you to completely control the placement of plots on the screen, and to write Java code that defines the data to be plotted.

The plot data can be specified in any of three data formats:

- *PlotML* is an XML extension for plot data. Its syntax is similar to that of HTML. XML (extensible markup language) is an internet language that is growing rapidly in popularity.
- An older, simpler textual syntax for plot data is also provided, although in the long term, that syntax is unlikely to be maintained (it will not necessarily be expanded to support new features). For simple data plots, however, it is adequate. Using it for applets has the advantage of making it possible to reference a slightly smaller jar file containing the code, which makes for more responsive applets. Also, the data files are somewhat smaller.
- A binary file format used by `pxgraph`, is supported by classes in the compat package. Formatting information in `pxgraph` (and in the compat package) is provided by command-line arguments, rather than being included with the binary plot data, exactly as in the older program. Applets specify these command-line arguments as an applet parameter (pxgraphargs).

6.2 Using Plots

If $PTII represents the home directory of your Ptplot installation (or your Ptolemy II installation), then, $PTII/bin is a directory that contains a number of executables. Three of these invoke plot applications, `ptplot`, `histogram`, and `pxgraph`. We recommend putting this directory into your path so that these executables can be found automatically from the command line. Invoking the command

```
ptplot
```

with no arguments should open a window that looks like that in figure 6.1. You can also specify a file to plot as a command-line argument. To find out about command-line options, type

```
ptplot -help
```

The ptplot command is a shell script that invokes the following equivalent command:

```
java -classpath $PTII ptolemy.plot.plotml.EditablePlotMLApplication
```

Since it is a shell script, it will work on Unix machines and Windows machines that have Cygwin[1] installed. In the same directory are three Windows versions that do not require Cygwin, `ptplot.bat`, `histogram.bat`, and `pxgraph.bat`, which you can invoke by typing into the DOS command prompt, for example,

```
ptplot.bat
```

1. The Cygwin Toolkit is a freely available package available from http://cygwin.com. A Ptolemy II specific subset of Cygwin can be found at http://ptolemy.eecs.berkeley.edu/ptolemyII/ptIIlatest/cygwin.htm

These scripts make three assumptions.

- First, `java` is in your path. Type "`java -version`" to verify that the `java` program is in your path and is working properly. Note that Ptplot 3.x and later require Java 1.3 or later, Java 1.4 is preferred.
- Second, the environment variable PTII is set to point to the home directory of the plot (or Ptolemy II) installation. Type "`echo %PTII%`" in a Windows DOS shell and "`echo $PTII`" in Unix or Windows Cygwin bash shell to check this.
- The directory $PTII/bin is in your path. Under Windows without Cygwin, type "`echo %PATH%`". Type "`type ptplot`" in Windows with Cygwin and "`which ptplot`" in Unix to check this.

In Windows, environment variables and your path are set in the System control panel. You can now explore a number of features of ptplot.

6.2.1 Zooming and filling

To zoom in, drag the left mouse button down and to the right to draw a box around an area that you want to see in detail, as shown in figure 6.2. To zoom out, drag the left mouse button up and to the right. To just fill the drawing area with the available data, type Control-F, or invoke the fill command from the Special menu. In applets, since there is no menu, the fill command is (optionally) made available as a button at the upper right of the plot.

6.2.2 Printing and exporting

The File menu includes a Print and Export command. The Print command works as you expect. The export command produces an encapsulated PostScript file (EPS) suitable for inclusion in word processors. The image in figure 6.3 is such an EPS file imported into FrameMaker.

At this time, the EPS file does not include preview data. This can make it somewhat awkward to

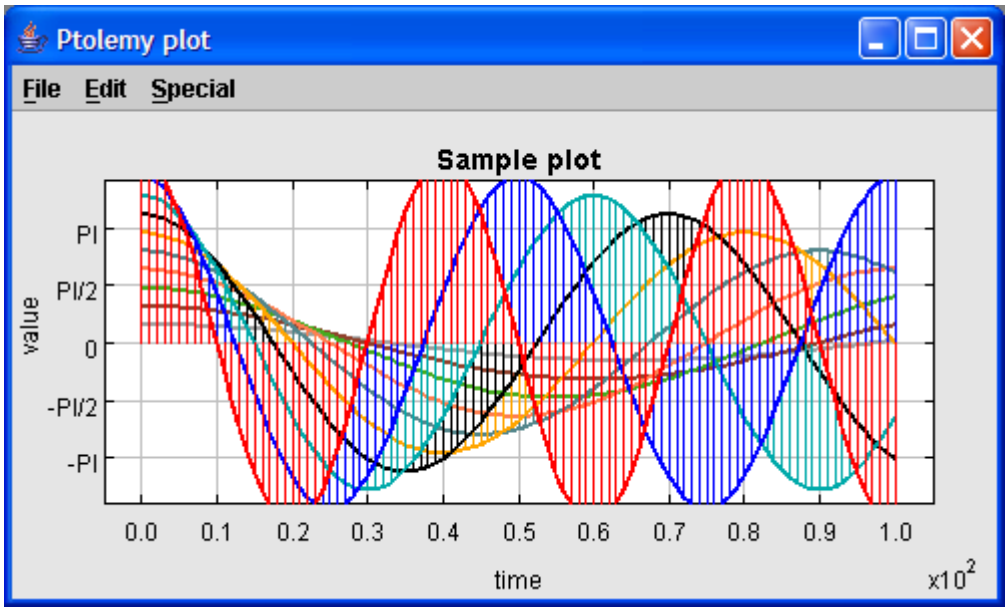


FIGURE 6.1. Result of invoking *ptplot* on the command line with no arguments.

work with in a word processor, since it will not be displayed by the word processor while editing (it will, however, print correctly). It is easy to add the preview data using the freely available program Ghostview[1]. Just open the file using Ghostview and, under the edit menu, select “Add EPS Preview.”

Export facilities are also available from a small set of key bindings, which permits them to be

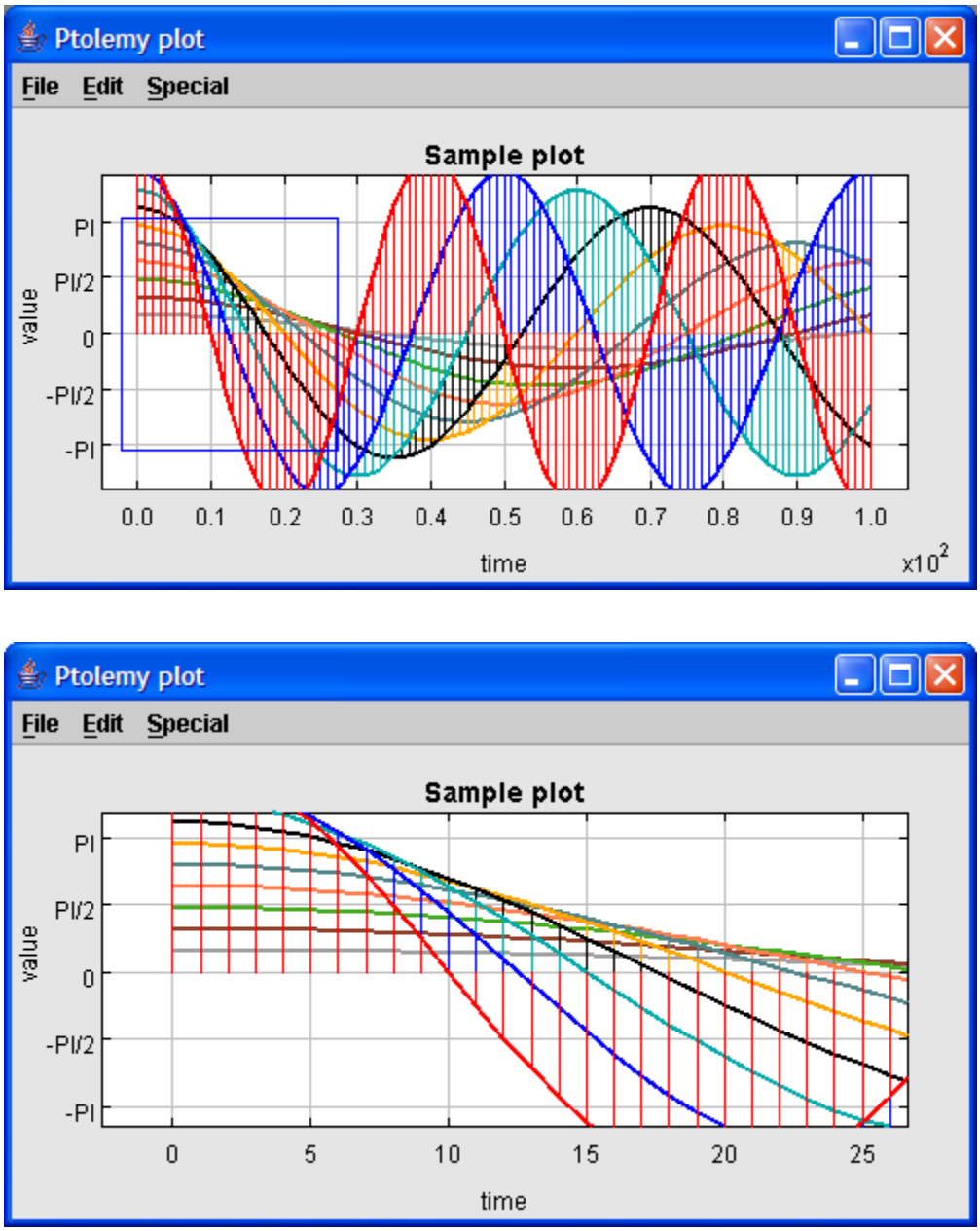


FIGURE 6.2. To zoom in, drag the left mouse button down and to the right to draw a box around the region you wish to see in more detail.

1. Ghostview is available http://www.cs.wisc.edu/~ghost

invoked from applets (which have no menu bar) and from the standalone scripts:

- Control-c: Copy plot to clipboard (EPS format), if permitted.
- D: Dump the plot to standard output in PlotML format.
- E: Export the plot to standard output in EPS format.
- F: Fill the plot.
- H or ?: Display a simple help message.
- Control-d or q: Quit

The encapsulated PostScript (EPS) that is produced is tuned for black-and-white printers. In the future, more formats may supported. Note that with JDK 1.3.0 under Windows 2000, Java's interface the clipboard may not work, so Control-C might not accomplish anything. Note further that with applets, you may find it best to click near the title rather than clicking inside the graph itself and then type the command.

Exporting to the clipboard and to standard output, in theory, is allowed for applets, unlike writing to a file. Thus, these key bindings provide a simple mechanism to obtain a high-resolution image of the plot from an applet, suitable for incorporation in a document. However, in some browsers, exporting to standard out triggers a security violation. You can use Sun's appletviewer instead.

6.2.3 Editing the data

You can modify the data that is plotted by first selecting a data set to modify using the Edit dataset command in the Edit menu, selecting a dataset and then dragging the right mouse button. Figure 6.4 shows the result of modifying one of the datasets (the one in red on a color display). The modification is carried out by freehand drawing, although considerable precision is possible by zooming in. Use the Save or SaveAs command in the File menu to save the modified plot (in PlotML format).

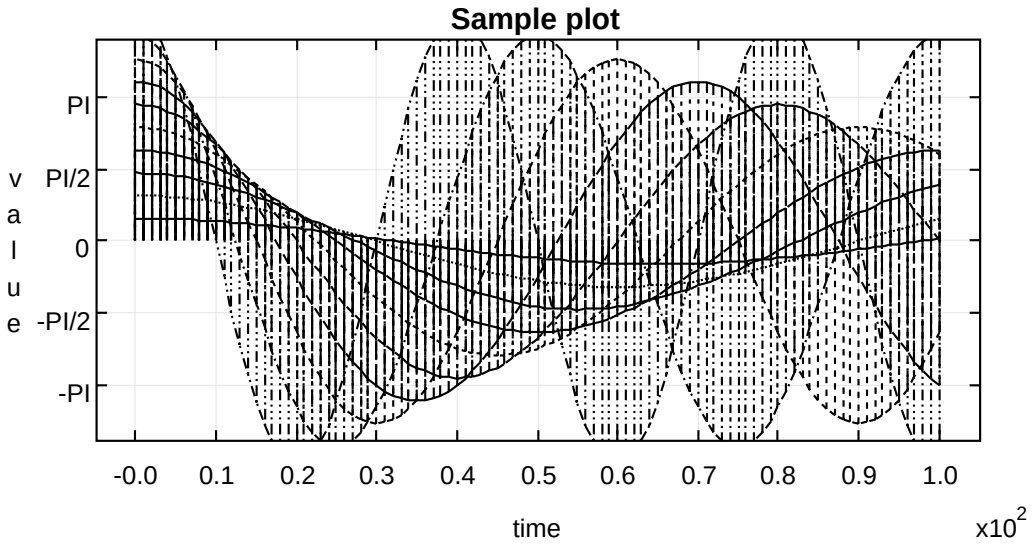


FIGURE 6.3. Encapsulated postscript generated by the Export command in the File menu of *ptplot* can be imported into word processors. This figure was imported into FrameMaker.

6.2.4 Modifying the format

You can control how data is displayed by invoking the Format command in the Edit menu. This brings up a dialog like that at bottom in figure 6.5. At the top is the dialog and the plot before changes are made, and at the bottom is after changes are made. In particular, the grid has been removed, the stems have been removed, the lines connecting the data points have been removed, the data points have been rendered with points, and the color has been removed. Use the Save or SaveAs command in the File menu to save the modified plot (in PlotML format). More sophisticated control over the plot can be had by editing the PlotML file (which is a text file). The PlotML syntax is described below.

The entries in the format dialog are all straightforward to use except the "X Ticks" and "Y Ticks" entries. These are used to specify how the axes are labeled. The tick marks for the axes are usually computed automatically from the ranges of the data. Every attempt is made to choose reasonable positions for the tick marks regardless of the data ranges (powers of ten multiplied by 1, 2, or 5 are used). To change what tick marks are included and how they are labeled, enter into the "X Ticks" or "Y Ticks" entry boxes a string of the following form:

```
label position, label position, ...
```

A *label* is a string that must be surrounded by quotation marks if it contains any spaces. A position is a number giving the location of the tick mark along the axis. For example, a horizontal axis for a frequency domain plot might have tick marks as follows:

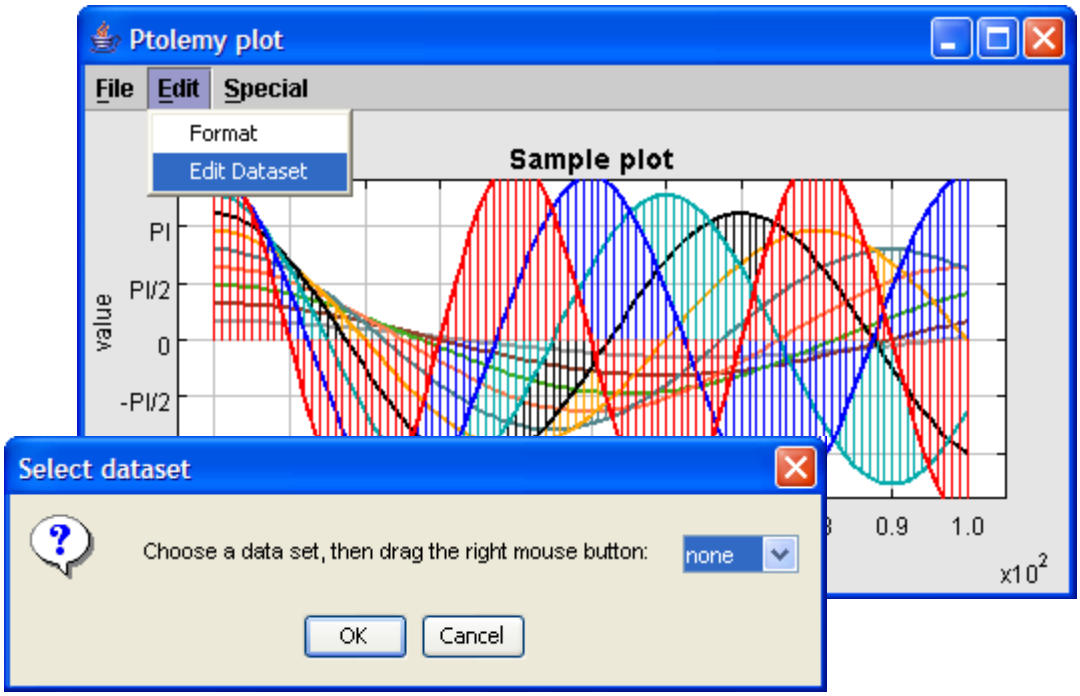


FIGURE 6.4. You can modify the data being plotted by selecting a data set and then dragging the right mouse button. Use the Edit menu to select a data set. Use the Save command in the File menu to save the modified plot (in PlotML format).

```
XTicks: -PI -3.14159, -PI/2 -1.570795, 0 0, PI/2 1.570795, PI 3.14159
```

Tick marks could also denote years, months, days of the week, etc.

6.3 Class Structure

The plot package has two subpackages, plotml and compat. The core package, plot, contains toolkit classes, which are used in Java programs as building blocks. The two subpackages contain classes that are usable by an end-user (vs. a programmer).

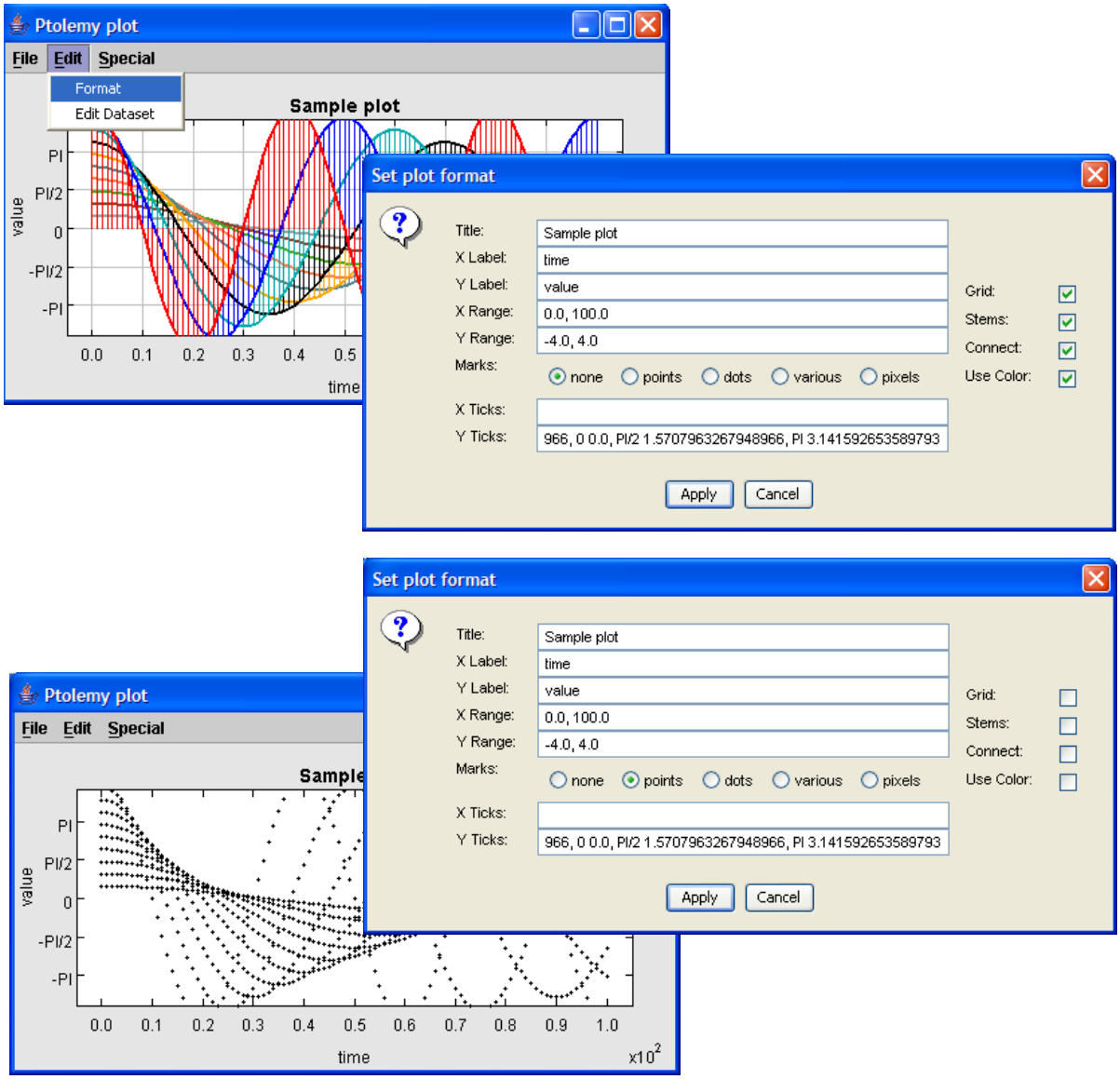


FIGURE 6.5. You can control how data is displayed using the Format command in the Edit menu, which brings up the dialog shown at the right. On the top is before changes are made, and on the bottom is after.

6.3.1 Toolkit classes

The class diagram for the core of the plot package is shown in figure 6.6. These classes provide a toolkit for constructing plotting applications and applets. The base class is PlotBox, which renders the axes and the title. It extends Panel, a basic container class in Java. Consequently, plots can be incorporated into virtually any Java-based user interface.

The Plot class extends PlotBox with data sets, which are collections of instances of PlotPoint. The EditablePlot class extends this further by adding the ability to modify data sets.

Live (animated) data plots are supported by the PlotLive class. This class is abstract; a derived class must be created to generate the data to plot (or collect it from some other application).

The Histogram class extends PlotBox rather than Plot because many of the facilities of Plot are irrelevant. This class computes and displays a histogram from a data file. The same data file can be read by this class and the other plot classes, so you can plot both the histogram and the raw data that is used to generate it from the same file.

6.3.2 Applets and applications

A number of classes are provided to use the plot toolkit classes in common ways, but you should keep in mind that these classes are by no means comprehensive. Many interesting uses of the plot package involve writing Java code to create customized user interfaces that include one or more plots. The most commonly used built-in classes are those in the *plotml* package, which can read PlotML files, as well as the older textual syntax.

Ptplot 5.5, which shipped with Ptolemy II 5.0 requires Swing. The easiest way to get Swing is to install the Java 1.4 (or later) Plug-in, which is part of the JRE and JDK 1.4 installation. Unfortunately, using the Java Plug-in makes the applet HTML more complex. There are two choices:

1. Use fairly complex JavaScript to determine which browser is running and then to properly select one of three different ways to invoke the Java Plug-in. This method works on the most different types of platforms and browsers. The JavaScript is so complex, that rather than reproduce it here, please see one of the demonstration html files.

2. Use the much simpler <applet> ...</applet> tag to invoke the Java Plug-in. This method works on many platforms and browsers, but requires a more recent version of the Java Plug-in, and will not work under Netscape Communicator 4.7x.

   For details about the above two choices, see http://java.sun.com/products/plugin/versions.html.

   We document the much simpler <applet> . . . </applet> tag format below

   The following segment of HTML is an example:

```
<APPLET
 code = "ptolemy.plot.plotml.PlotMLApplet"
 codebase = "../../.."
 archive = "ptolemy/plot/plotmlapplet.jar"
 width = "600"
 height = "400"
 >
<PARAM NAME = "background" VALUE = "#faf0e6" >
<PARAM NAME = "dataurl" VALUE = "plotmlSample.txt" >
  No Java Plug-in support for applet, see
 <a href="http://java.sun.com/products/plugin/"><code>http://java.sun.com/products/plugin/</code></a>
</APPLET>
```

To use this yourself you will probably need to change the *codebase* and *dataurl* entries. The first points

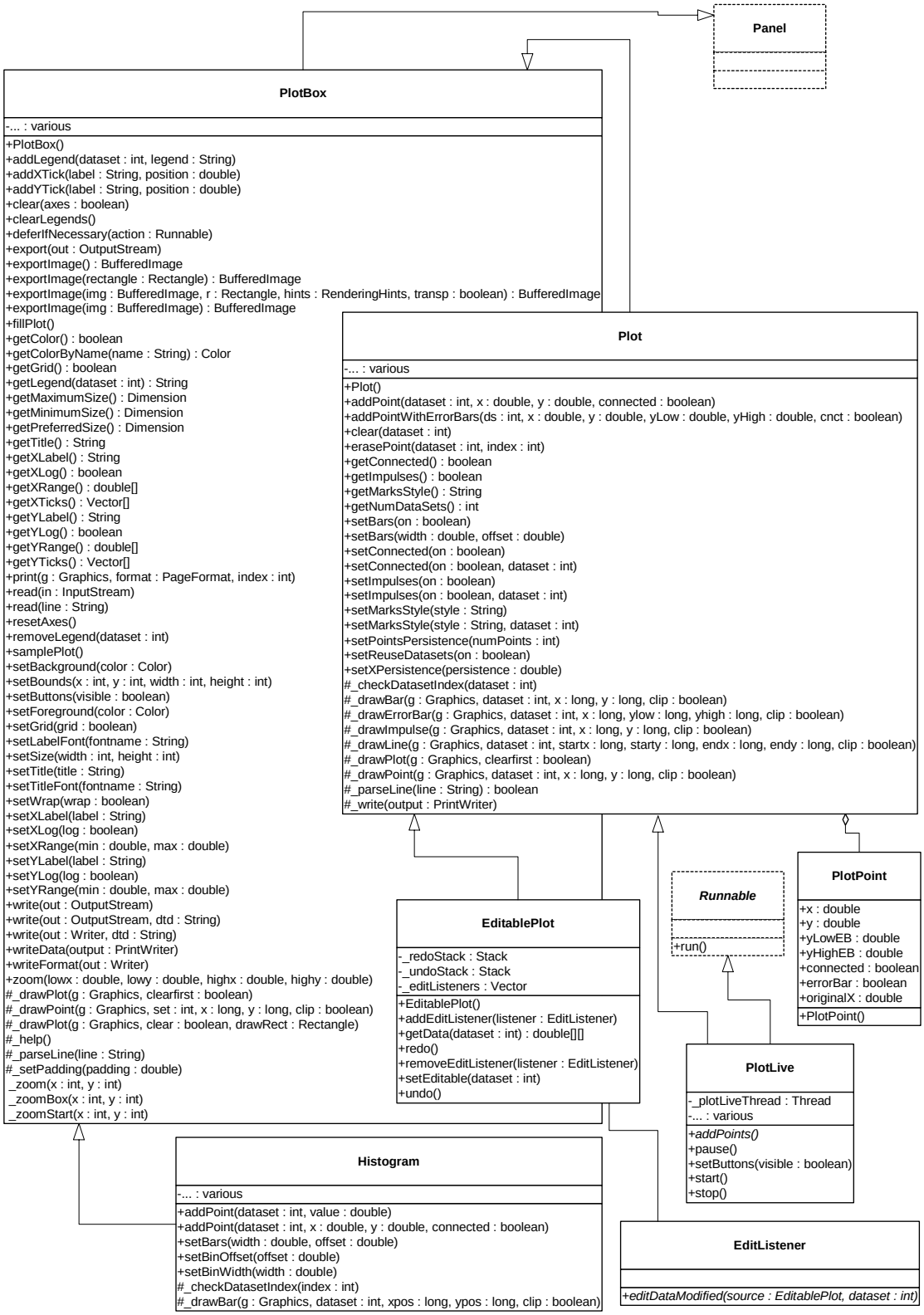


FIGURE 6.6. The core classes of the plot package.

to the root directory of the plot installation (usually, the value of the PTII environment variable). The second points to a file containing data to be plotted, plus optional formatting information. The file format for the data is described in the next section. The applet is created by instantiating the PlotMLApplet class.

The *archive* entry contains the name of the jar file that contains all the classes necessary to run a PlotML applet. The advantage of specifying a jar file is that remote users are likely to experience a faster download because all the classes come over at once, rather than the browser asking for each class from the server. A downside of using jar files in applets is that if you are modifying the source of Ptplot itself, then you must also update the jar file, or your changes will not appear. A common workaround is to remove the archive entry during testing, or remove the jar files themselves.

You can also easily create your own applet classes that include one or more plots. As shown in figure 6.6, the PlotBox class is derived from JPanel, a basic class of the Java Foundation Classes (JFC) toolkit, also known as Swing. It is easy to place a panel in an applet, positioned however you like, and to combine multiple panels into an applet. PlotApplet is a simple class that adds an instance of Plot.

Creating an application that includes one or more plots is also easy. The PlotApplication class, shown in figure 6.7, creates a single top-level window (a JFrame), and places within it an instance of Plot. This class is derived from the PlotFrame class, which provides a menu that contains a set of commands, including opening files, saving the plotted data to a file, printing, etc.

The difference between PlotFrame and PlotApplication is that PlotApplication includes a main() method, and is designed to be invoked from the command line. You can invoke it using commands like the following:

```
java -classpath $PTII ptolemy.plot.PlotApplication args
```

However, the classes shown in figure 6.7, which are in the plot package, are not usually the ones that an end user will use. Instead, use the ones in figure 6.8. These extend the base classes to support the PlotML language, described below. The only motivation for using the base classes in figure 6.7 is to have a slightly smaller jar file to load for applets.

The classes that end users are likely to use, shown in figure 6.8, include:

- PlotMLApplet: An applet that can read PlotML files off the web and render them.
- EditablePlotMLApplet: A version that allows editing of any data set in the plot.
- HistogramMLApplet: A version that uses the Histogram class to compute and plot histograms.
- PlotMLFrame: A top-level window containing a plot defined by a PlotML file.
- PlotMLApplication: An application that can be invoked from the command line and reads PlotML files.
- EditablePlotMLApplication: An extension that allows editing of any data set in the plot.
- HistogramMLApplication: A version that uses the Histogram class to compute and plot histograms.

EditablePlotMLApplication is the class invoked by the `ptplot` command-line script. It can open plot files, edit them, print them, and save them.

6.3.3 Writing applets

A plot can be easily embedded within an applet, although there are some subtleties. The simplest mechanism looks like this:

```
public class MyApplet extends JApplet {
  public void init() {
    super.init();
    Plot myplot = new Plot();
    getContentPane().add(myplot);
```

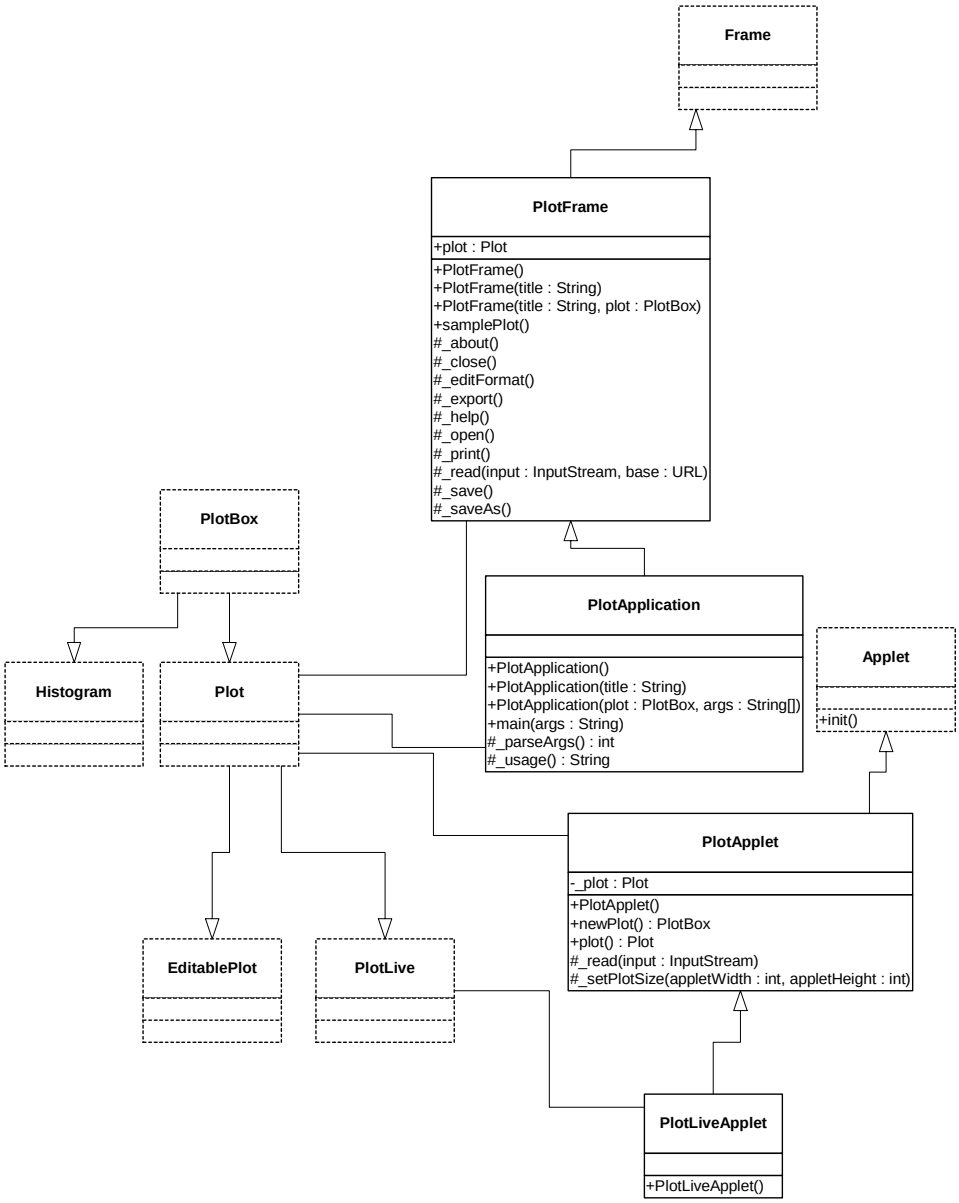


FIGURE 6.7. Core classes supporting applets and applications. Most of the time, you will use the classes in the plotml package, which extend these with the ability to read PlotML files.

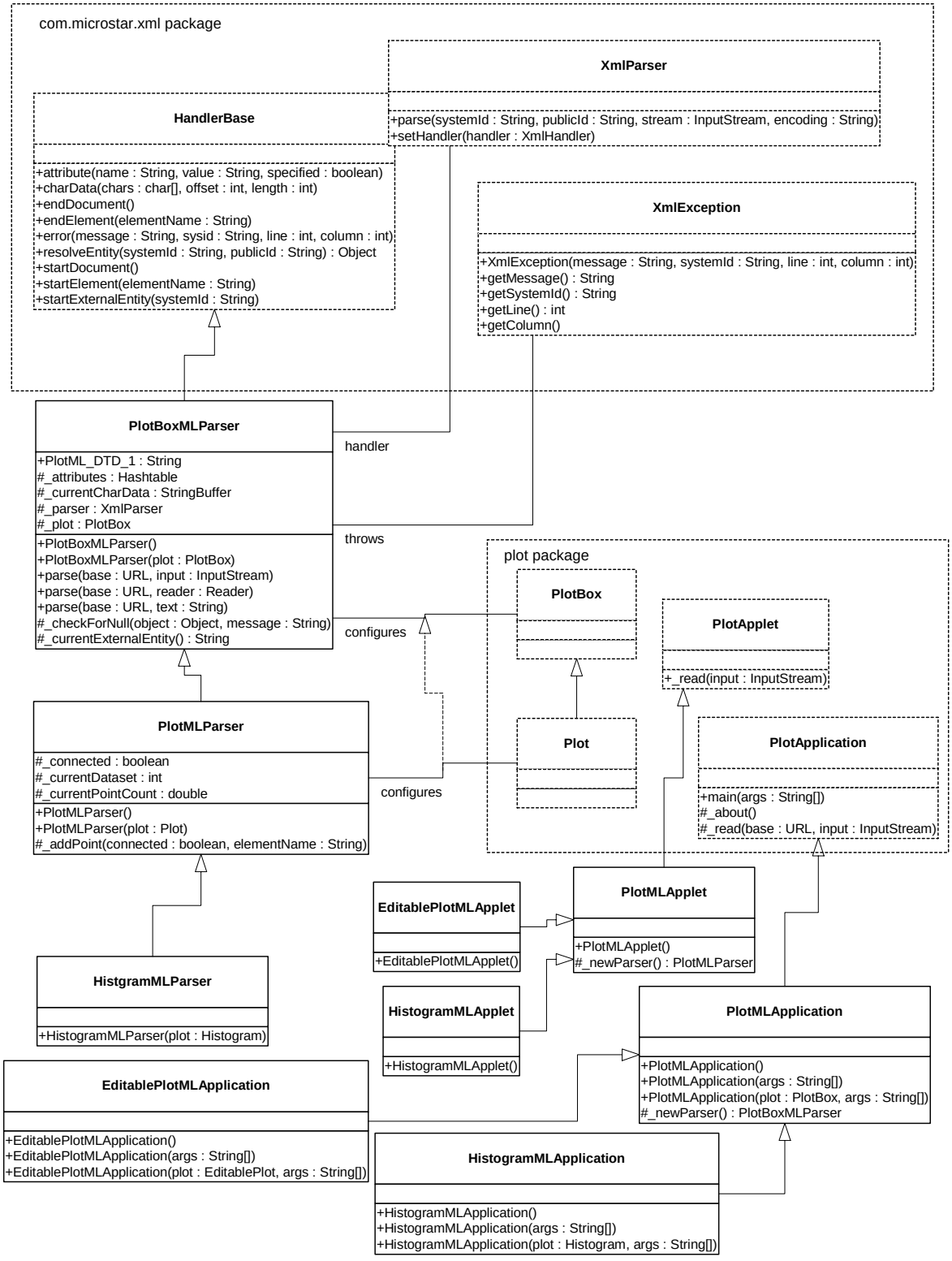


FIGURE 6.8. UML static structure diagram for the plotml package, a subpackage of plot providing classes

```
        myplot.setTitle("Title of plot");
        ...
    }
}
```

This places the plot in the center of the applet space, stretching it to fill the space available. To control the size independently of that of the applet, for some mysterious reason that only Sun can answer, it is necessary to embed the plot in a panel, as follows:

```
public class MyApplet extends JApplet {
  public void init() {
    super.init();
    Plot myplot = new Plot();
    JPanel panel = new JPanel();
    getContentPane().add(panel);
    panel.add(myplot);
    myplot.setSize(500, 300);
    myplot.setTitle("Title of plot");
    ...
  }
}
```

The setSize() method specifies the width and height in pixels. You will probably want to control the background color and/or the border, using statements like:

```
        myplot.setBackground(background color);
        myplot.setBorder(new BevelBorder(BevelBorder.RAISED));
```

Alternatively, you may want to make the plot transparent, which results in the background showing through:

```
        myplot.setOpaque(false);
```

6.4 PlotML File Format

Plots can be specified as textual data in a language called PlotML, which is an XML extension. XML, the popular *extensible markup language*, provides a standard syntax and a standard way of defining the content within that syntax. The syntax is a subset of SGML, and is similar to HTML. It is intended for use on the internet. Plot classes can save data in this format (in fact, the Save operation always saves data in this format), and the classes in the plotml subpackage, shown in figure 6.8, can read data in this format. The key classes supporting this syntax are PlotBoxMLParser, which parses a subset of PlotML supported by the PlotBox class, PlotMLParser, which parses the subset of PlotML supported by the Plot class, and HistogramMLParser, which parses the subset that supports histograms.

6.4.1 Data organization

Plot data in PlotML has two parts, one containing the plot data, including format information (how the plot looks), and the other defining the PlotML language. The latter part is called the *document type definition*, or DTD. This dual specification of content and structure is a key XML innovation.

Every PlotML file must either contain or refer to a DTD. The simplest way to do this is with the following file structure:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC "-//UC Berkeley//DTD PlotML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/PlotML_1.dtd">
<plot>
    format commands...
    datasets...
</plot>
```

Here, "*format commands*" is a set of XML elements that specify what the plot looks like, and "*datasets*" is a set of XML elements giving the data to plot. The syntax for these elements is described below in subsequent sections. The first line above is a required part of any XML file. It asserts the version of XML that this file is based on (1.0) and states that the file includes external references (in this case, to the DTD). The second and third lines declare the document type (plot) and provide references to the DTD.

The references to the DTD above refer to a "public" DTD. The name of the DTD is

```
-//UC Berkeley//DTD PlotML 1//EN
```

which follows the standard naming convention of public DTDs. The leading dash "`-`" indicates that this is not a DTD approved by any standards body. The first field, surrounded by double slashes, in the name of the "owner" of the DTD, "`UC Berkeley`." The next field is the name of the DTD, "`DTD PlotML 1`" where the "`1`" indicates version 1 of the PlotML DTD. The final field, "`EN`" indicates that the language assumed by the DTD is English.

In addition to the name of the DTD, the `DOCTYPE` element includes a URL pointing to a copy of the DTD on the web. If a particular PlotML tool does not have access to a local copy of the DTD, then it finds it at this web site. PtPlot recognizes the public DTD, and uses its own local version of the DTD, so it does not need to visit this website in order to open a PlotML file.

An alternative way to specify the DTD is:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE plot SYSTEM "DTD location">
<plot>
    format commands...
    datasets...
</plot>
```

Here, the DTD location is a relative or absolute URL.

A third alternative is to create a standalone PlotML file that includes the DTD. The result is rather verbose, but has the general structure shown below:

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE plot [
    DTD information
]>
<plot>
    format commands
    datasets
</plot>
```

These latter two methods are useful if you extend the DTD.

The DTD for PlotML is shown in figure 6.9. This defines the PlotML language. However, the DTD is not particularly easy to read, so we define the language below in a more tutorial fashion.

6.4.2 Configuring the axes

The elements described in this subsection are understood by the base class PlotBoxMLParser.

```
<title>Your Text Here</title>
```

The title is bracketed by the start element `<title>` and end element `</title>`. In XML, end elements are always the same as the start element, except for the slash. The DTD for this is simple:

```
<!ELEMENT title (#PCDATA)>
```

This declares that the body consists of *PCDATA, parsed character data.*

Labels for the X and Y axes are similar,

```
<xLabel>Your Text Here</xLabel>
<yLabel>Your Text Here</yLabel>
```

Unlike HTML, in XML, case is important. So the element is `xLabel` not `XLabel`.

The ranges of the X and Y axes can be optionally given by:

```
<xRange min="min" max="max"/>
<yRange min="min" max="max"/>
```

The arguments *min* and *max* are numbers, possibly including a sign and a decimal point. If they are not specified, then the ranges are computed automatically from the data and padded slightly so that datapoints are not plotted on the axes. The DTD for these looks like:

```
<!ELEMENT xRange EMPTY>
    <!ATTLIST xRange min CDATA #REQUIRED
                     max CDATA #REQUIRED>
```

The EMPTY means that the element does not have a separate start and end part, but rather has a final slash before the closing character "`/>`". The two ATTLIST elements declare that min and max

```
<!ELEMENT plot (barGraph | bin | dataset | default | noColor | noGrid | size | title | wrap | xLabel |
   xLog | xRange | xTicks | yLabel | yLog | yRange | yTicks)*>
  <!ELEMENT barGraph EMPTY>
    <!ATTLIST barGraph width CDATA #IMPLIED
                       offset CDATA #IMPLIED>
  <!ELEMENT bin EMPTY>
    <!ATTLIST bin width CDATA #IMPLIED
                  offset CDATA #IMPLIED>
  <!ELEMENT dataset (m | move | p | point)*>
    <!ATTLIST dataset connected (yes | no) #IMPLIED
                      marks (none | dots | points | various | pixels) #IMPLIED
                      name CDATA #IMPLIED
                      stems (yes | no) #IMPLIED>
  <!ELEMENT default EMPTY>
    <!ATTLIST default connected (yes | no) "yes"
                      marks (none | dots | points | various | pixels) "none"
                      stems (yes | no) "no">
  <!ELEMENT noColor EMPTY>
  <!ELEMENT noGrid EMPTY>
  <!ELEMENT reuseDatasets EMPTY>
  <!ELEMENT size EMPTY>
    <!ATTLIST size height CDATA #REQUIRED
                   width CDATA #REQUIRED>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT wrap EMPTY>
  <!ELEMENT xLabel (#PCDATA)>
  <!ELEMENT xLog EMPTY>
  <!ELEMENT xRange EMPTY>
    <!ATTLIST xRange min CDATA #REQUIRED
                     max CDATA #REQUIRED>
  <!ELEMENT xTicks (tick)+>
  <!ELEMENT yLabel (#PCDATA)>
  <!ELEMENT yLog EMPTY>
  <!ELEMENT yRange EMPTY>
    <!ATTLIST yRange min CDATA #REQUIRED
                     max CDATA #REQUIRED>
  <!ELEMENT yTicks (tick)+>
    <!ELEMENT tick EMPTY>
      <!ATTLIST tick label CDATA #REQUIRED
                     position CDATA #REQUIRED>
    <!ELEMENT m EMPTY>
      <!ATTLIST m x CDATA #IMPLIED
                  y CDATA #REQUIRED
                  lowErrorBar CDATA #IMPLIED
                  highErrorBar CDATA #IMPLIED>
    <!ELEMENT move EMPTY>
      <!ATTLIST move x CDATA #IMPLIED
                     y CDATA #REQUIRED
                     lowErrorBar CDATA #IMPLIED
                     highErrorBar CDATA #IMPLIED>
    <!ELEMENT p EMPTY>
      <!ATTLIST p x CDATA #IMPLIED
                  y CDATA #REQUIRED
                  lowErrorBar CDATA #IMPLIED
                  highErrorBar CDATA #IMPLIED>
    <!ELEMENT point EMPTY>
      <!ATTLIST point x CDATA #IMPLIED
                      y CDATA #REQUIRED
                      lowErrorBar CDATA #IMPLIED
                      highErrorBar CDATA #IMPLIED>
```

FIGURE 6.9. The *document type definition* (DTD) for the PlotML language.

attributes are required, and that they consist of character data.

The tick marks for the axes are usually computed automatically from the ranges. Every attempt is made to choose reasonable positions for the tick marks regardless of the data ranges (powers of ten multiplied by 1, 2, or 5 are used). However, they can also be specified explicitly using elements like:

```
<xTicks>
   <tick label="label" position="position"/>
   <tick label="label" position="position"/>
   ...
</xTicks>
```

A *label* is a string that replaces the number labels on the axes. A position is a number giving the location of the tick mark along the axis. For example, a horizontal axis for a frequency domain plot might have tick marks as follows:

```
<xTicks>
   <tick label="-PI" position="-3.14159"/>
   <tick label="-PI/2" position="-1.570795"/>
   <tick label="0" position="0"/>
   <tick label="PI/2" position="1.570795"/>
   <tick label="PI" position="3.14159"/>
</xTicks>
```

Tick marks could also denote years, months, days of the week, etc. The relevant DTD information is:

```
<!ELEMENT xTicks (tick)+>
    <!ELEMENT tick EMPTY>
      <!ATTLIST tick label CDATA #REQUIRED
                     position CDATA #REQUIRED>
```

The notation `(tick)+` indicates that the `xTicks` element contains one or more `tick` elements.

If ticks are not specified, then the X and Y axes can use a logarithmic scale with the following elements:

```
<xLog/>
<yLog/>
```

The tick labels, which are computed automatically, represent powers of 10. The log axis facility has a number of limitations, which are documented in “Limitations” on page 6-156.

By default, tick marks are connected by a light grey background grid. This grid can be turned off with the following element:

```
<noGrid/>
```

Also, by default, the first ten data sets are shown each in a unique color. The use of color can be turned off with the element:

```
<noColor/>
```

Finally, the rather specialized element

```
<wrap/>
```

enables wrapping of the X (horizontal) axis, which means that if a point is added with X out of range, its X value will be modified modulo the range so that it lies in range. This command only has an effect if the X range has been set explicitly. It is designed specifically to support oscilloscope-like behavior, where the X value of points is increasing, but the display wraps it around to left. A point that lands on the right edge of the X range is repeated on the left edge to give a better sense of continuity. The feature works best when points do land precisely on the edge, and are plotted from left to right, increasing in X.

You can also specify the size of the plot, in pixels, as in the following example:

```
<size width="400" height="300">
```

All of the above commands can also be invoked directly by calling the corresponding public methods from Java code.

6.4.3 Configuring data

Each data set has the form of the following example

```
<dataset name="grades" marks="dots" connected="no" stems="no">
  data
</dataset>
```

All of the arguments to the `dataset` element are optional. The `name`, if given, will appear in a legend at the upper right of the plot. The `marks` option can take one of the following values:

- `none`: (the default) No mark is drawn for each data point.
- `points`: A small point identifies each data point.
- `dots`: A larger circle identifies each data point.
- `various`: Each dataset is drawn with a unique identifying mark. There are 10 such marks, so they will be recycled after the first 10 data sets.
- `pixels`: A single pixel identified each data point.

The `connected` argument can take on the values “yes” and “no.” It determines whether successive datapoints are connected by a line. The default is that they are. Finally, the `stems` argument, which can also take on the values “yes” and “no,” specifies whether stems should be drawn. Stems are lines drawn from a plotted point down to the x axis. Plots with stems are often called “stem plots.”

The DTD is:

```
<!ELEMENT dataset (m | move | p | point)*>
    <!ATTLIST dataset connected (yes | no) #IMPLIED
          marks (none | dots | points | various | pixels) #IMPLIED
```

```
            name CDATA #IMPLIED
            stems (yes | no) #IMPLIED>
```

The default values of these arguments can be changed by preceding the `dataset` elements with a `default` element, as in the following example:

```
<default connected="no" marks="dots" stems="yes"/>
```

The DTD for this element is:

```
<!ELEMENT default EMPTY>
<!ATTLIST default connected (yes | no) "yes"
       marks (none | dots | points | various | pixels) "none"
       stems (yes | no) "no">
```

If the following element occurs:

```
<reuseDatasets/>
```

then datasets with the same name will be merged. This makes it easier to combine multiple data files that contain the same datasets into one file. By default, this capability is turned off, so datasets with the same name are not merged.

6.4.4 Specifying data

A dataset has the form

```
<dataset options>
  data
</dataset>
```

The data itself are given by a sequence of elements with one of the following forms:

```
<point y="yValue">
<point x="xValue" y="yValue">
<point y="yValue" lowErrorBar="low" highErrorBar="high">
<point x="xValue" y="yValue" lowErrorBar="low" highErrorBar="high">
```

To reduce file size somewhat, they can also be given as

```
<p y="yValue">
<p x="xValue" y="yValue">
<p y="yValue" lowErrorBar="low" highErrorBar="high">
<p x="xValue" y="yValue" lowErrorBar="low" highErrorBar="high">
```

The first form specifies only a Y value. The X value is implied (it is the count of points seen before in this data set). The second form gives both the X and Y values. The third and fourth forms give low and high error bar positions (error bars are use to indicate a range of values with one data point). Points

given using the syntax above will be connected by lines if the `connected` option has been given value "yes" (or if nothing has been said about it).

Data points may also be specified using one of the following forms:

```
<move y="yValue">
<move x="xValue" y="yValue">
<move y="yValue" lowErrorBar="low" highErrorBar="high">
<move x="xValue" y="yValue" lowErrorBar="low" highErrorBar="high">

<m y="yValue">
<m x="xValue" y="yValue">
<m y="yValue" lowErrorBar="low" highErrorBar="high">
<m x="xValue" y="yValue" lowErrorBar="low" highErrorBar="high">
```

This causes a break in connected points, if lines are being drawn between points. I.e., it overrides the `connected` option for the particular data point being specified, and prevents that point from being connected to the previous point.

6.4.5 Bar graphs

To create a bar graph, use:

```
<barGraph width="barWidth" offset="barOffset"/>
```

You will also probably want the `connected` option to have value "no." The *barWidth* is a real number specifying the width of the bars in the units of the X axis. The *barOffset* is a real number specifying how much the bar of the *i*-th data set is offset from the previous one. This allows bars to "peek out" from behind the ones in front. Note that the front-most data set will be the first one.

6.4.6 Histograms

To configure a histogram on a set of data, use

```
<bin width="binWidth" offset="binOffset"/>
```

The *binWidth* option gives the width of a histogram bin. I.e., all data values within one *binWidth* are counted together. The *binOffset* value is exactly like the *barOffset* option in bar graphs. It specifies by how much successive histograms "peek out."

Histograms work only on Y data; X data is ignored.

6.5 Old Textual File Format

Instances of the PlotBox and Plot classes can read a simple file format that specifies the data to be plotted. This file format predates the PlotML format, and is preserved primarily for backward compatibility. In addition, it is significantly more concise than the PlotML syntax, which can be advantageous, particularly in networked applications.

In this older syntax, each file contains a set of commands, one per line, that essentially duplicate

the methods of these classes. There are two sets of commands currently, those understood by the base class PlotBox, and those understood by the derived class Plot. Both classes ignore commands that they do not understand. In addition, both classes ignore lines that begin with "#", the comment character. The commands are not case sensitive.

6.5.1 Commands Configuring the Axes

The following commands are understood by the base class PlotBox. These commands can be placed in a file and then read via the read() method of PlotBox, or via a URL using the PlotApplet class. The recognized commands include:

- `TitleText: string`
- `XLabel: string`
- `YLabel: string`

These commands provide a title and labels for the X (horizontal) and Y (vertical) axes. A *string* is simply a sequence of characters, possibly including spaces. There is no need here to surround them with quotation marks, and in fact, if you do, the quotation marks will be included in the labels.

The ranges of the X and Y axes can be optionally given by commands like:

- `XRange: min, max`
- `YRange: min, max`

The arguments *min* and *max* are numbers, possibly including a sign and a decimal point. If they are not specified, then the ranges are computed automatically from the data and padded slightly so that datapoints are not plotted on the axes.

The tick marks for the axes are usually computed automatically from the ranges. Every attempt is made to choose reasonable positions for the tick marks regardless of the data ranges (powers of ten multiplied by 1, 2, or 5 are used). However, they can also be specified explicitly using commands like:

- `XTicks: label position, label position, ...`
- `YTicks: label position, label position, ...`

A *label* is a string that must be surrounded by quotation marks if it contains any spaces. A position is a number giving the location of the tick mark along the axis. For example, a horizontal axis for a frequency domain plot might have tick marks as follows:

```
XTicks: -PI -3.14159, -PI/2 -1.570795, 0 0, PI/2 1.570795, PI 3.14159
```

Tick marks could also denote years, months, days of the week, etc.

The X and Y axes can use a logarithmic scale with the following commands:

- `XLog: on`
- `YLog: on`

The tick labels, if computed automatically, represent powers of 10. The log axis facility has a number of limitations, which are documented in "Limitations" on page 6-156.

By default, tick marks are connected by a light grey background grid. This grid can be turned off with the following command:

- `Grid: off`

It can be turned back on with

- `Grid: on`

Also, by default, the first ten data sets are shown each in a unique color. The use of color can be turned off with the command:

- `Color: off`

It can be turned back on with

- `Color: on`

Finally, the rather specialized command

- `Wrap: on`

enables wrapping of the X (horizontal) axis, which means that if a point is added with X out of range, its X value will be modified modulo the range so that it lies in range. This command only has an effect if the X range has been set explicitly. It is designed specifically to support oscilloscope-like behavior, where the X value of points is increasing, but the display wraps it around to left. A point that lands on the right edge of the X range is repeated on the left edge to give a better sense of continuity. The feature works best when points do land precisely on the edge, and are plotted from left to right, increasing in X.

All of the above commands can also be invoked directly by calling the corresponding public methods from some Java code.

6.5.2 Commands for Plotting Data

The set of commands understood by the Plot class support specification of data to be plotted and control over how the data is shown.

The style of marks used to denote a data point is defined by one of the following commands:

- `Marks: none`
- `Marks: points`
- `Marks: dots`
- `Marks: various`
- `Marks: pixels`

Here, `points` are small dots, while `dots` are larger. If `various` is specified, then unique marks are used for the first ten data sets, and then recycled. If `pixels` is specified, then a single pixel is drawn. Using no marks is useful when lines connect the points in a plot, which is done by default. If the above directive appears before any DataSet directive, then it specifies the default for all data sets. If it appears after a DataSet directive, then it applies only to that data set.

To disable connecting lines, use:

- `Lines: off`

To re-enable them, use

- `Lines: on`

You can also specify “impulses”, which are lines drawn from a plotted point down to the x axis. Plots with impulses are often called “stem plots.” These are off by default, but can be turned on with the command:

- `Impulses: on`

or back off with the command

- `Impulses: off`

If that command appears before any DataSet directive, then the command applies to all data sets. Otherwise, it applies only to the current data set.

To create a bar graph, turn off lines and use any of the following commands:

- `Bars: on`
- `Bars: width`
- `Bars: width, offset`

The *width* is a real number specifying the width of the bars in the units of the x axis. The *offset* is a real number specifying how much the bar of the i-th data set is offset from the previous one. This allows bars to "peek out" from behind the ones in front. Note that the front-most data set will be the first one. To turn off bars, use

- `Bars: off`

To specify data to be plotted, start a data set with the following command:

- `DataSet: string`

Here, *string* is a label that will appear in the legend. It is not necessary to enclose the string in quotation marks.

To start a new dataset without giving it a name, use:

- `DataSet:`

In this case, no item will appear in the legend.

If the following directive occurs:

- `ReuseDataSets: on`

then datasets with the same name will be merged. This makes it easier to combine multiple data files that contain the same datasets into one file. By default, this capability is turned off, so datasets with the same name are not merged.

The data itself is given by a sequence of commands with one of the following forms:

- *x*, *y*
- `draw:` *x*, *y*
- `move:` *x*, *y*
- *x*, *y*, *yLowErrorBar*, *yHighErrorBar*
- `draw:` *x*, *y*, *yLowErrorBar*, *yHighErrorBar*
- `move:` *x*, *y*, *yLowErrorBar*, *yHighErrorBar*

The `draw` command is optional, so the first two forms are equivalent. The `move` command causes a break in connected points, if lines are being drawn between points. The numbers *x* and *y* are arbitrary numbers as supported by the Double parser in Java (e.g. "1.2", "6.39e-15", etc.). If there are four numbers, then the last two numbers are assumed to be the lower and upper values for error bars. The numbers can be separated by commas, spaces or tabs.

6.6 Compatibility

Figure 6.10 shows a small set of classes in the compat package that support an older ascii and binary file formats used by the popular `pxgraph` program (an extension of `xgraph` to support binary formats). The PxgraphApplication class can be invoked by the pxgraph executable in $PTII/bin. See the PxgraphParser class documentation for information about the file format.

6.7 Limitations

The plot package is a starting point, with a number of significant limitations.

- A binary file format that includes plot format information is needed. This should be an extension of PlotML, where an external entity is referenced.
- If you zoom in far enough, the plot becomes unreliable. In particular, if the total extent of the plot is more than 2^{32} times extent of the visible area, quantization errors can result in displaying points or lines. Note that 2^{32} is over 4 billion.
- The log axis facility has a number of limitations. Note that if a logarithmic scale is used, then the values must be positive. **Non-positive values will be silently dropped**. Further log axis limitations are listed in the documentation of the _gridInit() method in the PlotBox class.
- Graphs cannot be currently copied via the clipboard.

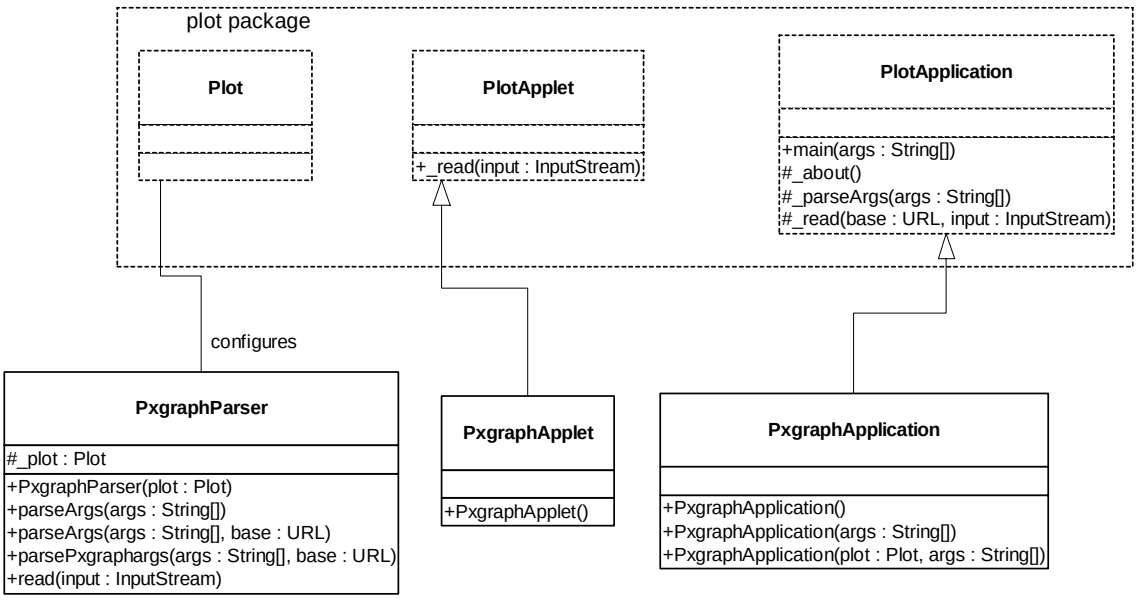


FIGURE 6.10. The compat package provides compatibility with the older pxgraph program.

7

Code Generation

Authors: *Man-Kit Leung*
Gang Zhou
Contributor: *Christopher Brooks*

7.1 Motivation

Ptolemy II is a software lab for experimenting with multiple concurrency formalisms for embedded system design. Many features in Ptolemy II contribute to the ease of its use as a rapid prototyping environment. In particular, modular components make systems more flexible and extensible. Different compositions of the same components can implement different functionality. However, component designs are often slower than custom-built code. The cost of inter-component communication through the component interface introduces overhead, and generic components are highly parameterized for the reusability and thus less efficient.

To regain the efficiency for the implementation, the users could write big monolithic components to reduce inter-component communication, and write highly specialized components rather than general ones. However, manually implementing these solutions is not an option. Partial evaluation [65] provides a mechanism to automate the whole process. In the past, partial evaluation has been mostly used for general purpose software. Recently, partial evaluation has begun to see its use in the embedded world, e.g, see [71]. In our research partial evaluation is used as a code generation technique, which is really a compilation technique for transforming an actor-oriented model into the target code while preserving the model's semantics. However, compared with traditional compiler optimization, partial evaluation for embedded software works at the component level and heavily leverages the domain-specific knowledge. Through model analysis, the tool can discover data type, buffer size, parameter value, model structure and model execution schedule, and then partially evaluate all the known information to reach a very efficient implementation. The end result is that the benefit offered by the high level abstraction comes with (almost) no performance penalty.

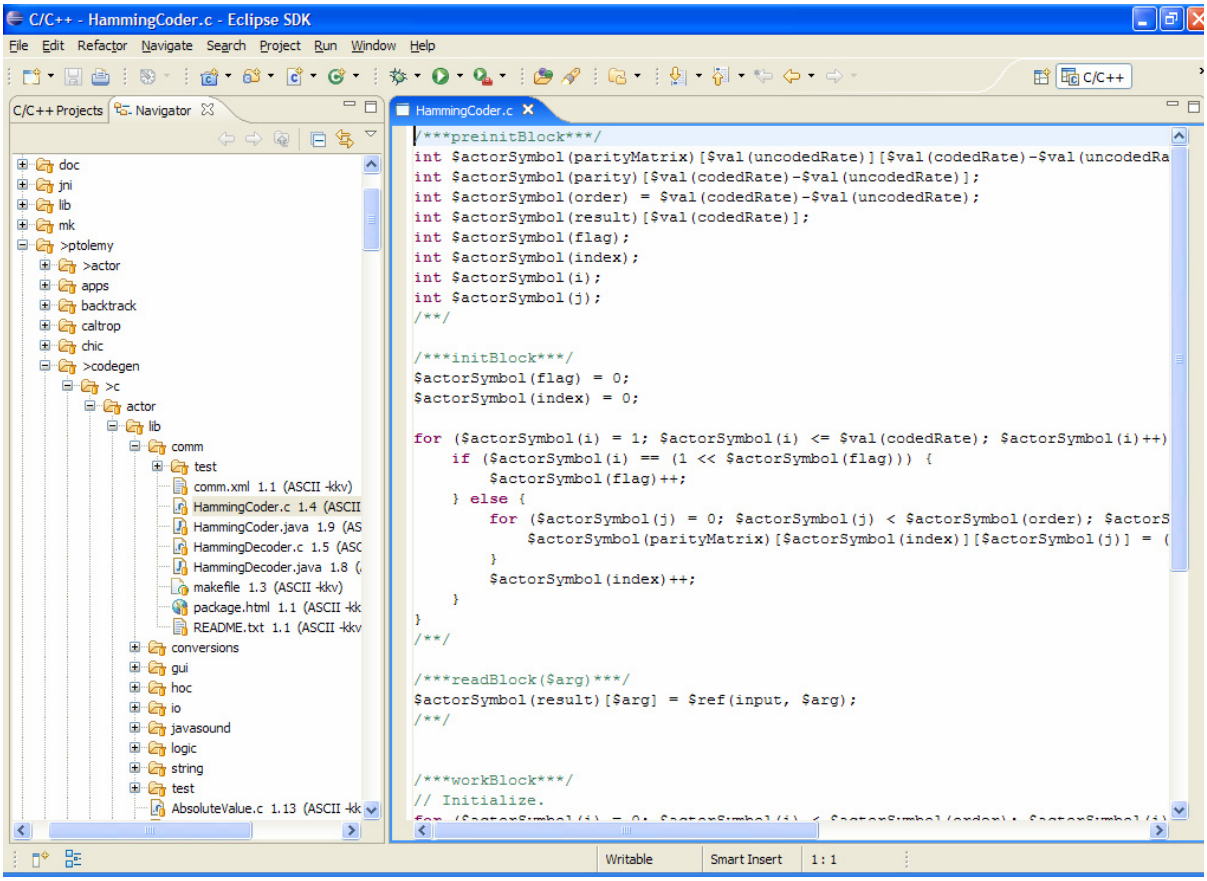


FIGURE 7.1. The C/C++ Development Toolkit (CDT) in the Eclipse Integrated Development Environment.

7.2 A Helper-based Mechanism

Our code generation framework uses a helper-based mechanism. A codegen helper is essentially a component that generates code for a Ptolemy II actor. Each Ptolemy II actor for which code will be generated in a specific language has one associated helper. An actor may have multiple helpers to support multiple target languages (C, VHDL, etc.).

To achieve readability and maintainability in the implementation of helper classes, the target code blocks (for example, the initialize block, fire block, and wrapup block) of each helper are placed in a separate file under the same directory. So a helper essentially consists of two files: a java class file and a code template file. This not only decouples the writing of Java code and target code (otherwise the target code would be wrapped in strings and interspersed with java code), but also allows using a target language specific editor while working on the target language code blocks. For example, in the Eclipse Integrated Development Environment, the C/C++ Development Toolkit (CDT) provides C and C++ extensions to the Eclipse workbench as a set of Eclipse plug-ins, see figure 7.1. The convenient features such as keyword highlights in the C/C++ specific editor could help the writing of C code, resulting in improved productivity.

So the code template file contains code blocks written in the target language. The target code

```
// CountTrues.c

/*** preinitBlock ***/
int $actorSymbol(trueCount);
int $actorSymbol(i);
/**/

/*** fireBlock ***/
$actorSymbol(trueCount) = 0;

for ($actorSymbol(i) = 0; $actorSymbol(i) < $val(blockSize); $actorSymbol(i)++) {
   if ($ref(input, $actorSymbol(i))) {
      $actorSymbol(trueCount)++;
   }
}
$ref(output) = $actorSymbol(trueCount);
/**/
```

FIGURE 7.2. The C code template file for the CountTrues helper.

blocks are hand-coded so users have flexibility in choosing their design styles and algorithms. Hand-coded templates also retain readability in the generated code. The codegen kernel uses the java class of the helper to harvest code blocks from the code template file. The java class of the helper determines which code blocks to harvest based on the actor instance-specific information (e.g., port type, port width, and parameter values). The code template file contains codegen macros that are processed by the codegen kernel. These macros allow the kernel to generate customized code based on the actor instance-specific information.

7.2.1 What is in a C Code Template File?

A C code template file has a .c file extension but it is not C-compilable due to its unique structure. Only the CodeStream object understands how to parse and use these files. Figure 7.2 shows the C code template file for the CountTrues helper, located in $PTII/ptolemy/codegen/c/domains/sdf/lib.

A C code template file consists of one or more C code blocks. Each code block has a header and a footer. The helper uses a CodeStream object to parse the code blocks. Please refer to Appendix C for a detailed documentation of the CodeStream object. The header and footer tags are code block separators that help the CodeStream in parsing. The footer is simply the tag “/**/”. The header starts with the begin tag “/***” and ends with the end tag “***/”. The header also contains a code block name and optionally a parameter list. The parameter list is enclosed by a pair of parentheses “()” and multiple parameters in the list are separated by commas “,”. A code block may have arbitrary number of parameters. Each parameter is prefixed by the dollar sign “$” symbol (i.e. $value, $width, etc.), which allows the CodeStream object to do a straight text substitution with the string value of the parameter. Formally, the signature of a code block is defined as the pair (N, p) where N is the code block name and p is the number of parameters. A code block (N, p) may be overloaded by another code block (N, p’)[1]. Furthermore, different helpers in a class hierarchy may contain code blocks with the same (N, p). So a unique reference to a code block signature is the tuple (H, N, p) where H is the corresponding helper. Defining the uniqueness of a code block prevents unambiguity in referencing a code block.

1. All parameters in a code block are implicitly strings. So unlike the usual overloaded functions with the same name but different types of parameters, we need different number of parameters to have an overload relationship for code blocks.

```
public String generateFireCode() throws IllegalActionException {
    StringBuffer code = new StringBuffer();
    code.append(super.generateFireCode());
    ptolemy.actor.lib.ElementsToArray actor
           = (ptolemy.actor.lib.ElementsToArray) getComponent();

    ArrayList args = new ArrayList();
    args.add(new Integer(0));

    for (int i = 0; i < actor.input.getWidth(); i++) {
        args.set(0, new Integer(i));
        code.append(_generateBlockCode("fillArray", args));
    }
    code.append(_generateBlockCode("sendOutput"));
    return processCode(code.toString());
}
```

FIGURE 7.3. The generateFireCode() function of the ElementsToArray helper.

A code block can also be overridden. A code block (H, N, p) is overridden by a code block (H', N, p) given that H' is a child class of H. This gives rise to code block inheritance. Since Ptolemy II actors are defined within a well-defined class hierarchy, many actors inherits fields and methods from parent actors. The codegen helpers mirror the same class hierarchy. Since code blocks represent functions of actors, the code blocks should be inherited for helpers just as functions are inherited for actors. Given a request for fetching a code block, the CodeStream object searches through all code template files of the helper and its ancestors, starting from the bottom of the class hierarchy. This mirrors the same behavior of invoking an (inherited) function for an actor.

7.2.2 What is in a Helper Java Class File?

A helper java class is a valid Java class that extends the CodeGeneratorHelper class. The CodeGeneratorHelper class implements the code generation interfaces (ComponentCodeGenerator and ActorCodeGenerator) for generating code. The interfaces essentially consist of a set of methods that return code strings for specific parts of the target program (init(), fire(), wrapup(), etc.). The CodeGeneratorHelper class implements the default behavior for these methods: each method fetches and returns a code block (with no parameters) using the default code block name ("initBlock", "fireBlock", "wrapupBlock", etc.) corresponding to the method (generateInitializeCode(), generateFireCode(), generateWrapupCode(), etc.). The child helper java class can either inherit the default behavior or override any method to fetch multiple code blocks with non-default names, give parameters to code blocks, or do special processing on the returned code string.

Figure 7.3 is the ElementsToArray helper's implementation of the generateFireCode() method, found in $PTII/ptolemy/codegen/c/actor/lib. This method uses the channel number of the actor input port as the parameter and fetches the "fillArray" code block from the ElementsToArray.c code template file inside the for-loop. It generates multiple copies of the "fillAarray" code block, each customized with a different channel number. It then fetches and appends a different code block with the name "sendOutput" (with no parameters). Finally, it invokes the processCode() function to process the embedded macros in the code string. Note that a helper java class needs to understand the semantics of its corresponding actor in order to implement these generate methods.

7.2.3 The Macro Language

The macro language allows helpers to be written once, and then used in different context where the

macros are expanded and resolved. All macros in a code block are prefixed with the dollar sign "$" symbol (i.e. $ref(input), $val(width), etc.). The specific macro name follows immediately. The parameters to the macro are enclosed in parentheses "()". Macros can be nested and recursively processed by the codegen helper. The use of the dollar sign as prefix is based on the assumption that it is not a valid identifier in the target language ("$" is not a valid identifier in C). The macro prefix can be configured to correspond to different target languages. The macro names specifies different rules of text substitutions that the code generator helper performs to alter the content of the code block. Since the same set of code blocks may be shared by multiple instances of the helper, the macros mainly serve the purpose of producing unique labels for different instances and generate instance-specific port and parameter information. The following is the documentation of the set of macros used in the C code generation.

The Core Macros:

$ref(name)

Returns a unique reference to a parameter or a port in the global scope. For a multiport, use $ref(name#i) where i is the channel number. During macro expansion, the name is replaced by the full name resulting from the model hierarchy.

$ref(name, offset)

Returns a unique reference to an element in an array parameter or a port with an offset in the global scope. The offset must not be negative. $ref(name, 0) is equivalent to $ref(name).

$val(parameter-name)

Returns the current value of the parameter associated with an actor in the simulation model. The advantage of not using $ref macro in place of $val is that no additional memory needs to be allocated. $val macro is usually used when the parameter is constant during the execution.

$actorSymbol(name)

Returns a unique reference of a user-defined variable in the global scope. This macro is used to define additional variables, for example, to hold internal states of actors between firings. The uniqueness only requires that the name argument be unique within the scope of each actor. The helper writer is responsible for declaring these variables.

$size(name)

If the given name represents an ArrayType parameter, it returns the size of the array. If the given name represents a port of an actor, it returns the width of that port.

The Type Convert Macros (see Appendix C):

$type()

Returns the numeric constant that represents the type of the given port or parameter. The code generator uses the mapping between the token type and the numeric constant to look up the function table.

$targetType()

Returns the corresponding target language type of the given typed parameter or port.

$cgtype()
Returns the name string of the codegen type corresponding to a given typed parameter or port.

$new()
Returns a new Token object of the given type. This macro takes at least one argument: the codegen type name of the Token with the value that are needed by the constructor function of the specific Token type. The code generator keeps track of the different types used through this macro. For example, "$new(Int(2))" creates an Int Token variable in the macro language. It allocates space for the Token; however, the user is responsible for calling the specific delete() function on the Token to deallocate space.

$tokenFunc()
Returns a function call associated with the given token. The function call is translated to a function pointer in a two-dimensional function table. The first index of the table is the different token types, and the second index is the different functions for each type. The type of the given token is used to find the first index, and the name of the function is used to find the second index. The first argument of the function is always the given token, which acts as 'this' in an object-oriented environment. The result is always another Token. For example, the following code illustrates how a user adds two Int tokens together:

```
$tokenFunc($new(Int(2)::add($new(Int(3)))))
```

$typeFunc()
Returns a function call associated with the given token type. Instead of using an associated token, type function uses an associated type class which acts similar to a static class function. The following illustrates how a user converts an Int token to a String token:

```
$typeFunc(TYPE_String::convert($new(Int(2))))
```

7.2.4 The CountTrues Example

Figure 7.4 shows a model in the synchronous dataflow (SDF) domain that counts the true values produced from the data source, which in this case is the Pulse actor. The CountTrues actor has its blockSize parameter set to 2, which means it reads 2 tokens from its input port for each firing. The Pulse actor' parameters are set to the values shown in the figure. When the model is simulated in the Ptolemy II framework, the produced result is also shown in the figure (the model is fired 4 times because the SDFDirector's "iterations" parameter is set to 4).

Let's look at the C code template files of the actors in the model. Macros are used extensively in the code blocks, and we will see how they are processed and changed in the generated code. Figure 7.5 shows the C code template files for the Pulse and CountTrues helpers.

Double clicking on the StaticSchedulingCodeGenerator icon brings up the code generator window. Clicking the "Generate" button in the code generator window starts the code generation for this model. It generates a stand-alone C application program that executes and produces the same result as the simulation model. Figure 7.6 shows the main function of the generated C program.

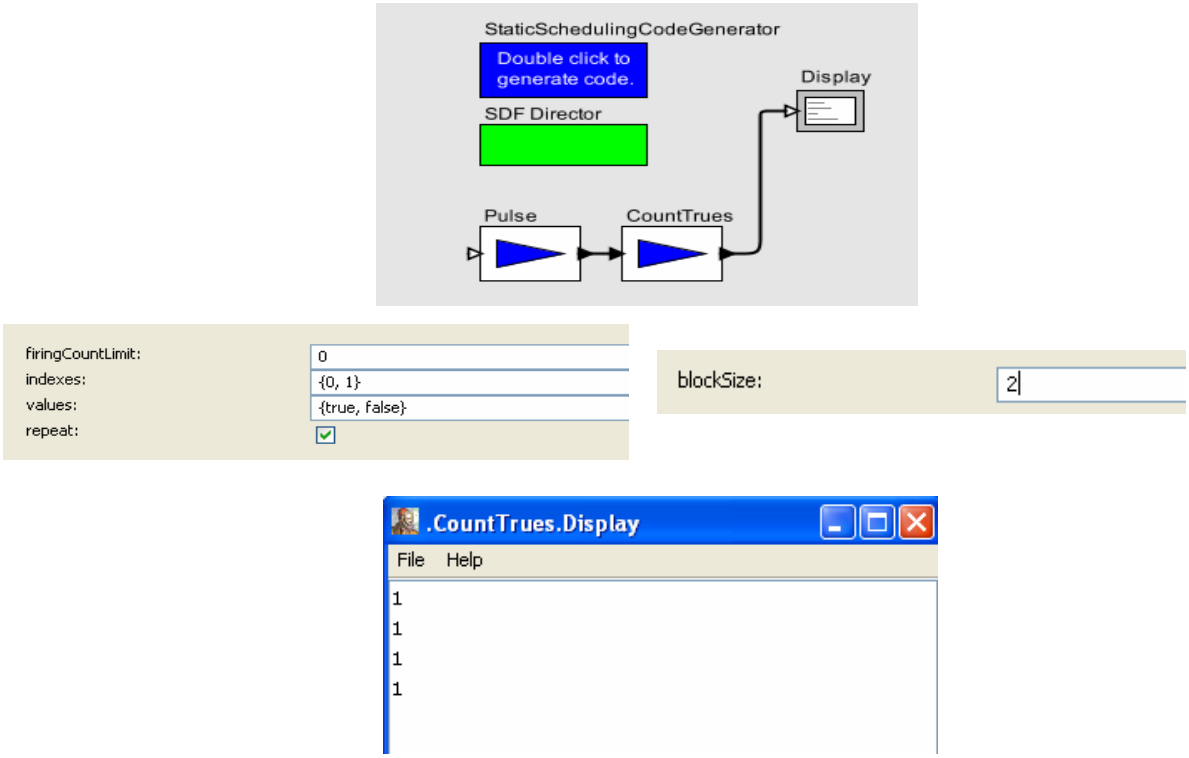


FIGURE 7.4. The model, Pulse's parameters, CountTrues' parameter and the simulation result.

The generated code is essentially the result of combining the helpers' code blocks. The $ref() and $actorSymbol() macro are replaced with unique labels to represent different variable references. The $val() macro in the CountTrues' "fireBlock" code block is replaced by the parameter value of the CountTrue instance in the model. When the generated C program is compiled and executed, the same result is produced as from the Ptolemy II simulation:

```
Display: 1
Display: 1
Display: 1
Display: 1
```

7.3 Overview of The Software Architecture

Our code generation framework has the flavor of CG (i.e., Code Generation) domain and other derived domains in Ptolemy Classic [126]. However, in Ptolemy Classic, code generation domains and simulation domains are separate and so are the actors (called stars in Ptolemy Classic terminology) used in these domains. In ptolemy Classic, the actors in the simulation domains participate in simulation whereas the corresponding actors in the code generation domains participate in code generation. Separate domains (simulation vs. code generation) make it inconvenient to integrate the model design phase with the code generation phase and streamline the whole process. Separate actor libraries make it difficult to maintain a consistent interface for a simulation actor and the corresponding code generation actor.

In Ptolemy II, there are no separate code generations domains. Once a model has been designed,

```
// Pulse.c

/***preinitBlock***/
int $actorSymbol(iterationCount) = 0;
int $actorSymbol(indexColCount) = 0;
unsigned char $actorSymbol(match) = 0;
/**/

/***fireBlock***/
if ($actorSymbol(indexColCount) < $size(indexes)
       && $actorSymbol(iterationCount) == $ref(indexes, $actorSymbol(indexColCount))) {
   $ref(output) = $ref(values, $actorSymbol(indexColCount));
   $actorSymbol(match) = 1;
} else {
   $ref(output) = 0;
}
if ($actorSymbol(iterationCount) <= $ref(indexes, $size(indexes) - 1)) {
   $actorSymbol(iterationCount) ++;
}
if ($actorSymbol(match)) {
   $actorSymbol(indexColCount) ++;
   $actorSymbol(match) = 0;
}
if ($actorSymbol(indexColCount) >= $size(indexes) && $val(repeat)) {
   $actorSymbol(iterationCount) = 0;
   $actorSymbol(indexColCount) = 0;
}
/**/

// CountTrues.c

/*** preinitBlock ***/
int $actorSymbol(trueCount);
int $actorSymbol(i);
/**/

/*** fireBlock ***/
$actorSymbol(trueCount) = 0;

for ($actorSymbol(i) = 0; $actorSymbol(i) < $val(blockSize); $actorSymbol(i)++) {
   if ($ref(input, $actorSymbol(i))) {
      $actorSymbol(trueCount)++;
   }
}
$ref(output) = $actorSymbol(trueCount);
/**/
```

FIGURE 7.5. The C code template files for the Pulse and CountTrues helpers.

simulated and verified to satisfy the given specification in the simulation domain, code can be directly generated from the model. Each helper doesn't have its own interface. Instead, it interrogates the associated actor to find its interface (ports and parameters) during the code generation. Thus the interface consistency is maintained naturally. The generated code, when executed, should present the same behavior as the original model. Compared with the Ptolemy Classic approach, this new approach allows the seamless integration between the model design phase and the code generation phase.

Figure 2.12 shows the UML diagram of key classes to support execution in the ptolemy.actor package. The Executable interface defines how an object can be invoked. The preinitialize() method is assumed to be invoked exactly once during the lifetime of an execution of a model and before the type resolution. The initialize() methods is assumed to be invoked once after the type resolution. It may be invoked again to reinitialize a (sub)model, for example, in a modal model while taking a transition

```
...
...
static int iteration = 0;

main(int argc, char *argv[]) {

   init();

   /* Static schedule: */
   for (iteration = 0; iteration < 4; iteration ++) {

      /* fire Composite Actor CountTrues */
      /* fire Pulse */
      if (_CountTrues_Pulse_indexColCount < 2
            && _CountTrues_Pulse_iterationCount == Array_get(_CountTrues_Pulse_indexes_ ,
            _CountTrues_Pulse_indexColCount).payload.Int) {
         _CountTrues_CountTrues_input[0] = Array_get(_CountTrues_Pulse_values_ ,
             _CountTrues_Pulse_indexColCount).payload.Boolean;
         _CountTrues_Pulse_match = 1;
      } else {
         _CountTrues_CountTrues_input[0] = 0;
      }

      if (_CountTrues_Pulse_iterationCount
            <= Array_get(_CountTrues_Pulse_indexes_ ,2 - 1).payload.Int) {
         _CountTrues_Pulse_iterationCount ++;
      }
      if (_CountTrues_Pulse_match) {
         _CountTrues_Pulse_indexColCount ++;
         _CountTrues_Pulse_match = 0;
      }
      if (_CountTrues_Pulse_indexColCount >= 2 && true) {
         _CountTrues_Pulse_iterationCount = 0;
         _CountTrues_Pulse_indexColCount = 0;
      }

      /* fire Pulse */
      // The code for the second firing of the Pulse actor is omitted here.
      .....
      .....

      /* fire CountTrues */
      _CountTrues_CountTrues_trueCount = 0;

      for (_CountTrues_CountTrues_i = 0; _CountTrues_CountTrues_i < 2;_CountTrues_CountTrues_i++) {
         if (_CountTrues_CountTrues_input[(0 + _CountTrues_CountTrues_i)%2]) {
             _CountTrues_CountTrues_trueCount++;
         }
      }
      _CountTrues_Display_input[0] = _CountTrues_CountTrues_trueCount;

      /* fire Display */
      printf("Display: %d\n", _CountTrues_Display_input[0]);
   }

   wrapup();

   exit(0);
}
```

FIGURE 7.6. The main function of the generated C program for the CountTrues model.

with the reset parameter being true. The prefire(), fire(), and postfire() methods will usually be invoked many times, with each sequence of method invocations defined as one iteration. The stopFire() method is invoked to request suspension of firing. The wrapup() method will be invoked exactly once per execution at the end of the execution. The terminate() method is provided as a last-resort mechanism to interrupt execution based on an external event.

The Executable interface is implemented by the Director class, and is extended by the Actor interface. An actor is an executable entity. There are two types of actors, AtomicActor, which extends ComponentEntity, and CompositeActor, which extends CompositeEntity. An AtomicActor is a single entity, while a CompositeActor is an aggregation of actors.

The classes to support code generation are in the packages under ptolemy.codegen where the helper class hierarchy and package structure parallel those of regular Ptolemy actors. The counterpart of the Executable interface is the ComponentCodeGenerator interface and the ActorCodeGenerator interface. These interfaces define the methods for generating code in different stages corresponding to what happens in the simulation.

The ptolemy.codegen.kernel.CodeGeneratorHelper class is the base class implementing these interfaces and provides common features for all actor helpers. It gives a skeleton implementation for writing a helper class. Each actor has a corresponding helper class for generating functionally equivalent code for this actor in a target language. Actors and their helpers have the same names so that the Java reflection mechanism can be used to load the helper for the corresponding actor during the code generation process. For example, there is a Ramp actor in the package ptolemy.actor.lib. Correspondingly, there is a Ramp helper in the package ptolemy.codegen.c.actor.lib. Here c represents the fact that all the helpers under ptolemy.codegen.c generate C code. Assume we would like to generate code for another target language X, the helpers for generating X code could be implemented under ptolemy.codegen.x. This would result in extendable code generation framework. Developers could not only contribute their own actors and helpers, but also add functionality to generate code for a new target language.

In the generated code, the ports of actors become memory resources in the target language, e.g., global variables in C code. A code template file can also define new variables to specify the need for global resources. A helper, however, does not have the full knowledge of the global resources such as their full names since that would be resolved only during the code generation process. Therefore a set of macros to access the global resources, as defined in the previous section, can be used. The macros are resolved and expanded according to the context in a specific model.

The above approach to create actor helpers achieves modularity, maintainability, portability and efficiency in code generation. The target code for each helper can be verified for correctness and optimized for efficiency individually. The code for the whole model is assembled from the target code for the contained actors plus some extra code serving as glue logic.

To generate code for hierarchically composed models, helpers for composite actors are also created. For example, the most commonly used composite actor is TypedCompositeActor in the package ptolemy.actor. A helper with the same name is created in the package ptolemy.codegen.c.actor. The main function of this helper is to delegate the code generation for the associated composite actor to the helper of the local director (discussed next) or the helpers of the actors contained by the composite actor. Other composite actors include ModalModel, Refinement, etc. and the corresponding helpers are created for each of them (see more details in the Domains section).

In Ptolemy II, a director governs the execution of a composite actor and thus each director needs a

helper to generate the code for the containing composite actor so that the generated code is functionally equivalent to the composite actor. Currently we can generate code for the domains where actor executions can be statically scheduled such as the SDF and HDF domains. The involved directors include SDF director, HDF director, HDFFSM director, MultirateFSM director and FSM director. Indeed there is a helper for each director with the same class name and located in the corresponding package under ptolemy.codegen. These director helpers generate code according to the Models of Computation (MoCs) they represent. They concatenate target code blocks with the help of a scheduler, allocate memory according to the calculated buffer sizes, and also generate the target code for the glue logic specific to their MoCs. The details will be presented in the next section.

Finally the StaticSchedulingCodeGenerator class is used to orchestrate the whole code generation process. An instance of this class is contained by the top level composite actor as an attribute and interacts with it directly. Therefore the code generation starts at the top level composite actor and the code for the whole model is generated hierarchically, much similar to how a model is simulated in Ptolemy II environment.

The flow chart in figure 7.7 sketches the whole code generation process step by step. The details of some steps are MoC-specific and will be presented in the next section. Notice that the steps outlined do not necessarily follow the same order the generated codes are assembled together. For example, only those parameters that are referenced during the code generation will be defined. Therefore those definitions will be generated last but attached to variable definition segment at the beginning of the generated code.

Readers could find out by now that the helper based code generation framework functions as a coordination language for the target code. It not only leverages the huge legacy code repository, but also takes advantage of many years and many researchers' work on compiler optimization techniques for the target language. It is accessible to a huge base of programmers. Oftentimes a new language fails to catch on not because it is technically flawed, but because it is very difficult to penetrate the barrier established by the languages already in widespread use. With the use of the helper class combined with target code template written in a language programmers are familiar with, there is much less of a learning curve to use our design and codegen environment.

7.4 Domains

7.4.1 SDF

The synchronous dataflow (SDF) domain is one of the most mature domains in Ptolemy II. The details of this domain can be found in Volume 3. Under the SDF domain, the execution order of actors is statically determined prior to execution. This opens the door for generating some very efficient code. In fact, the SDF software synthesis has been studied extensively. One book [18], several Ph.D. dissertations [17][112] and numerous papers have been written about it. Many optimization techniques have been designed according to different criteria such as minimization of program size, buffer size [18][113], or actor activation rate [133]. Hardware synthesis from SDF specification has also been studied by many researchers, e.g., see [62].

In parallel with the software architecture in the simulation domain, the helper for the SDFDirector is inherited from the helper for the StaticSchedulingDirector, which is further inherited from the helper for the Director. The helper for the StaticSchedulingDirector is responsible for generating the target code semantically equivalent to a predetermined sequence of actor firings in one complete static

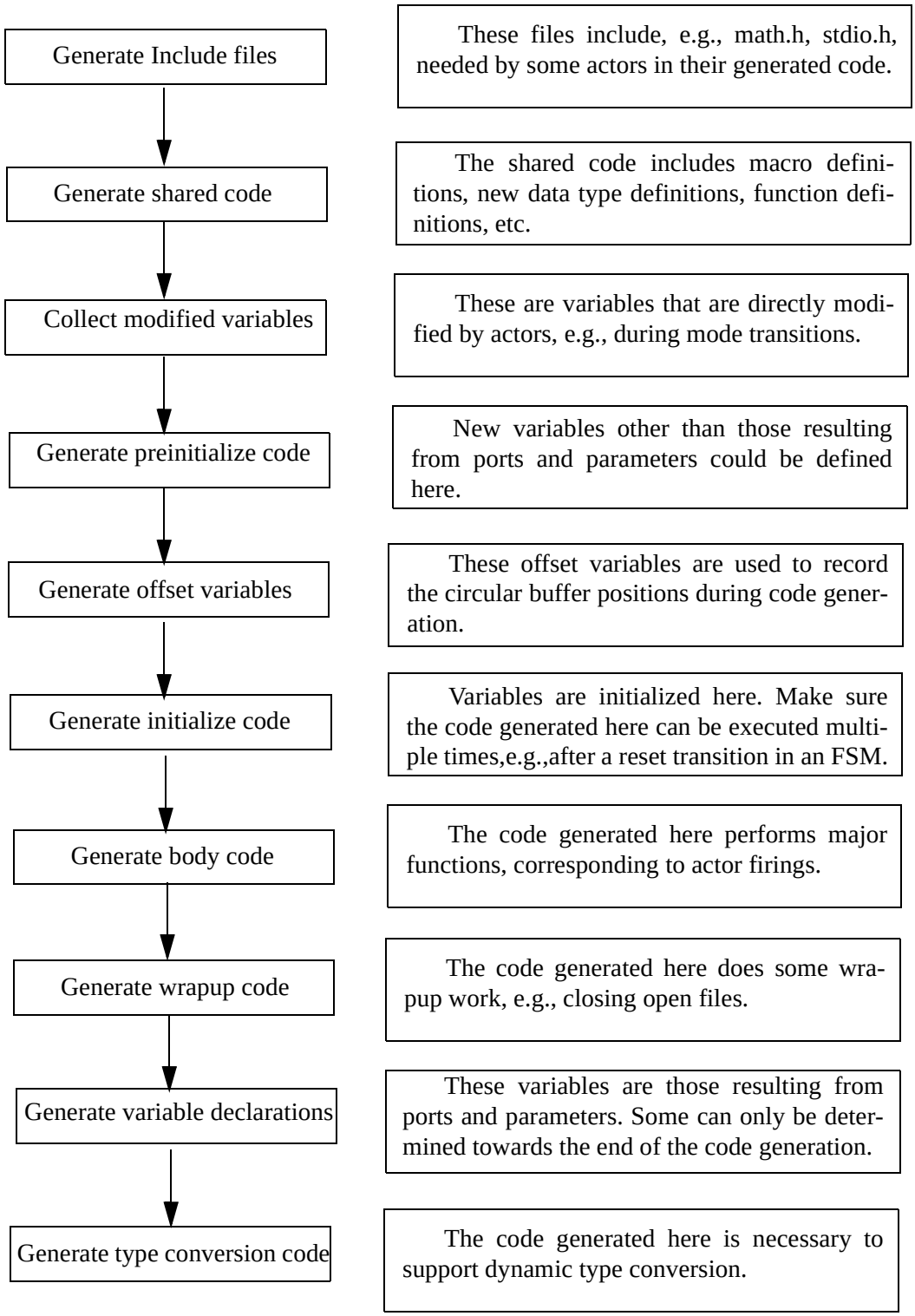


FIGURE 7.7. The flow chart of the code generation process.

schedule. The helper for the SDFDirector utilizes the SDFScheduler implemented in the current SDF domain, but the framework allows to plug in any optimizing scheduler and then generates the corresponding code for that scheduler.

There are several points to notice in the implementation of the helper for the SDFDirector. First, the minimum buffer size of each receiver is determined by the SDFScheduler and the helper for the SDFDirector uses that information to determine the size for the buffer array in the generated code. For a minimum buffer size of one, only a simple variable instead of an array is used. For a multiport, when the minimum buffer size of each receiver contained by the multiport is one, one dimensional array is used where each element of the array corresponds to a different receiver contained by the multiport; when the minimum buffer size of any receiver contained by the multiport exceeds one, a general two dimensional array is used. Second, the firing code for each actor is inlined by default, i.e., during the code generation, the firing code of each actor is expanded whenever the actor needs to be fired as dictated by the schedule, resulting in a monolithic body of the code for the whole model. When the inline option is switched off, the firing code for each actor is wrapped inside its own function and the generated code calls these functions in the order dictated by the schedule. The inline version may run faster without the call-return overhead. The non-inline version may reduce the memory footprint of the generated code when there is no single appearance schedule (a single appearance schedule is a looped schedule where each actor shows up at most once [18]) or when the reduction of the buffer size using multiple appearance schedule is more effective than the reduction of program size using single appearance schedule. We are experimenting with different options to generate code with better performance or better (smaller) size.

The code generation framework has been under active development and we can generate code for a lot of models. But still we cannot generate code for all SDF models, mostly due to the lack of helpers for the actors contained in these models, but sometimes due to other reasons such as the lack of codegen support for the data types used in the models. We are continuously adding more helpers and more functionalities.

Several interesting demos for the SDF code generation are presented next. Figure 7.8 shows the Butterfly demo in $PTII/ptolemy/domains/sdf/demo/Butterfly. During code generation, the helper for the Expression actor uses a PtParser to parse the Ptolemy expression specified in the actor and then uses a CParseTreeCodeGenerator to generate the corresponding C code. The C code generated by the helper for the XYPlotter actor invokes JVM (Java Virtual Machine) through JNI (Java Native Interface) and then calls the methods of the classes in the plot package for two-dimensional graphical display. Notice the generated C code does not need the Ptolemy framework to run. It merely uses the plot utilities (which happen to be written in Java) for displaying data. One could write a different helper for the XYPlotter actor to generate the customized code for displaying data in a specific target system.

The Case demo in $PTII/ptolemy/actor/lib/hoc/demo/Case shows a model with a switch/case-type structure. The interesting part about this model is the use of a Case actor controlled by a CaseDirector. Correspondingly, there is a helper for the Case actor and a helper for the CaseDirector for code generation.

7.4.2 FSM

Finite state machines (FSMs) have been the subject of a long history of research work. In ptolemy II, an FSM actor serves two purposes: traditional FSM modeling and modal models. In traditional FSM modeling, an FSM actor reacts to the inputs by making state transitions and sending tokens to the output ports like a regular Ptolemy actor. In the *charts formalism with modal models [46], modes are

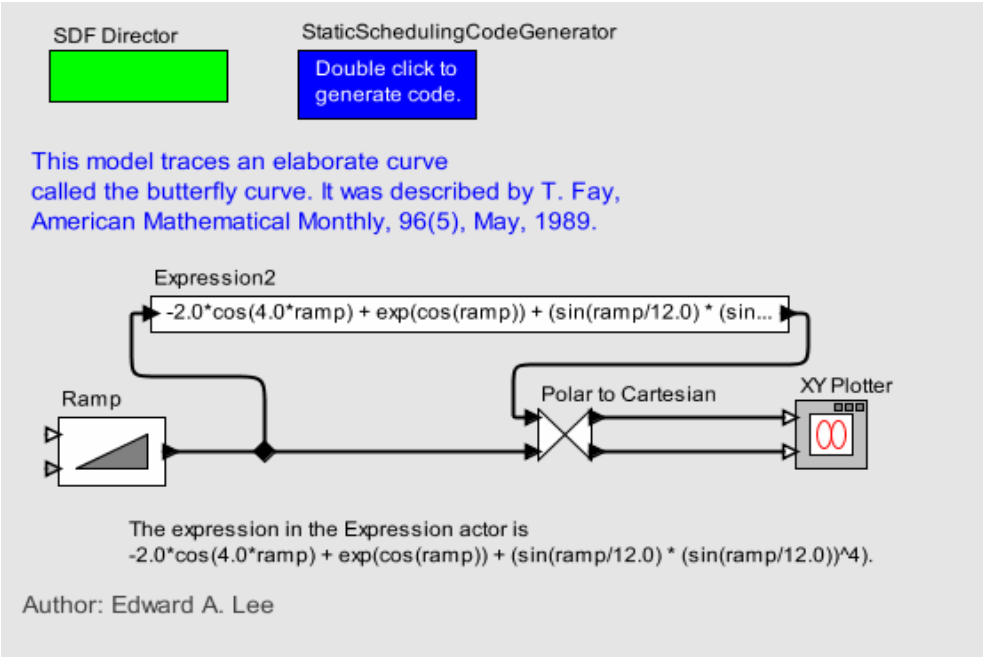


FIGURE 7.8. SDF Code generation for the Butterfly demo.

represented by states of an FSM actor that controls mode switching. Each mode has one or more refinements that specify the behavior of the mode. A modal model is constructed in a ModalModel actor having the FSM director as local director. The ModalModel actor contains a ModalController (essentially an FSM actor) and a set of Refinement actors that model the refinements associated with states and possibly a set of TransitionRefinement actors that model the refinements associated with transitions. The FSM director mediates the interaction with the outside domain, and coordinates the execution of the refinements with the ModalController. The details of this domain can be found in Volume 3.

Correspondingly, the helper for the FSMActor is designed to generate appropriate target code according to its context. When the FSMActor is used as a standalone actor, the helper generates the code for all outgoing transitions for each state; when it is used as a ModalController in a ModalModel, the helper generates the code for preemptive transitions and the code for non-preemptive transitions separately. Regardless of the context, for each state and each outgoing transition from that state, the generated code follows exactly the same execution sequence as in the original model--first, the guard expression is translated into the target code using a ParseTreeCodeGenerator; then all the choice actions contained by the transition are transformed into the target code; if there is any refinement associated with the transition (represented by a TransitionRefinement actor which is different from a Refinement actor associated with a state), the corresponding code for the refinement is generated; next, all the commit actions contained by the transition are transformed into the target code; the code for updating new current state follows; finally the code for (re)initializing the refinement associated with the new state is generated when the reset parameter of the transition is true.

The internal execution of a ModalModel is controlled by a local director, which can be an FSMDirector, a MultirateFSMDirector, or an HDFFSMDirector. A user can choose a director for a ModalModel. Usually only one specific director makes sense for the behavior the user wants for the model. For

example, an FSMDirector is used when the rate for all the ports of a ModalModel is 1 and the ModalModel tries a transition from the current state during every firing of the ModalModel. A MultirateFSMDirector is inherited from FSMDirector to model multirate behavior. To guarantee static schedulability under SDF, all state refinements must present the same rate to the outside for all the ports mirrored to the same port in the ModalModel. In addition, a MultirateFSMDirector makes only non-preemptive transitions so that the refinement for the current state gets fired before trying a transition and tokens are consumed from input ports and sent to output ports according to the rate of each port. If the refinement associated with different state presents a different rate, an HDFFSMDirector must be used. It is further inherited from MultirateFSMDirector and only tries a transition after the whole model finishes the execution of one complete schedule. Together with HDFDirector, it implements the HDF domain (see section 7.4.3). No matter which FSM director the user chooses, the corresponding helper generates the target code which preserves the original model behavior.

To support code generation for modal models, four helpers are also created including ModalController, ModalModel, Refinement and TransitionRefinement. They are inherited from the helpers for FSMActor or TypedCompositeActor but add no new functionality because their associated actor classes are used to build modal model structures (such as mirroring ports). These helpers inherit their functionality from their base classes.

The VariableScope inner class in CodeGeneratorHelper handles code generation for expressions contained by parameters. To support code generation for modal models, the derived PortScope inner class in the helper for the FSMActor handles code generation for guard expressions with multi-channel and multi-rate syntax. Both inner classes would expand any expression recursively until any variable in the expression is either a constant (in this case the constant is substituted for the variable) or a modifiable variable, e.g., modified in a mode transition or in a SetVariable actor (in this case a variable which is defined at the beginning of the generated code is used).

Several interesting demos for the FSM code generation are presented next. The Blending demo in $PTII/ptolemy/domains/fsm/demo/Blending models a controller with two major control modes and two transition modes. Each major mode has one Refinement. Each transition mode has three Refinements. Some Refinements are shared by different modes. After the controller is designed and simulated to meet the given specification in the Ptolemy environment, the generated C code can then run on some embedded platform.

The Modal Binary Symmetric Channel demo in $PTII/ptolemy/domains/fsm/demo/ModalBSC models a channel with two states, each with different probabilities of error. The model not only has Refinement associated with state, but also has TransitionRefinement associated with transition. These modeling constructs are automatically realized in the generated code.

7.4.3 HDF

The Heterochronous Dataflow (HDF) domain extends the SDF domain by allowing changes in port rates (called rate signature) between iterations of the whole model. Within each iteration, rate signatures are fixed and an HDF model behaves like an SDF model. This guarantees that a schedule can be completely executed. Between iterations, any modal model can make a state transition and therefore derives its rate signature from the refinement associated with the new state. The HDF domain recomputes the schedule when necessary. The details of this domain can be found in Volume 3.

Since it’s expensive to compute the schedule during the run time, all possible schedules are precomputed during the compilation time (i.e., code generation time). The structure of the generated code is hard-coded in such a way that it reflects all possible execution paths for different schedules.

Since an HDF model can have arbitrary levels of hierarchy, there must be a systematic way to find out the number of schedules and enumerate all schedules for any HDF model. The approach taken here uses the following definition.

For each opaque actor (i.e., atomic actor or opaque composite actor), let N be the number of configurations of this actor. For each configuration of the actor, it has a corresponding local SDF schedule. (An atomic actor has a degenerate form of local SDF schedule: fire itself once.)

Let r_{ij} be the rate of the jth port of this actor in the ith configuration, where $i = 0, \ldots, N-1$, $j = 0, \ldots, P-1$ and P is the number of ports of this actor.

Let $R_i = \{r_{ij} | j = 0, \ldots, P-1\}$ be the rate signature of this actor in the ith configuration.

During code generation, N and R_i, $i = 0, \ldots, N-1$ for each opaque actor are derived in a recursive bottom-up fashion:

1. ***Atomic actor***: $N = 1$, R_0 = the rate signature of the atomic actor.

2. ***Composite actor with a local SDFDirector***: $N = 1$, R_0 = the rate signature of the composite actor inferred from the local SDF schedule. *Precondition*: each contained actor has only one rate signature (i.e., one configuration).

3. ***Modal model with a local FSMDirector***: $N = 1$, $R_0 = \{r_{0j}\}$ where $r_{0j} = 1$ for all j. *Precondition*: all the refinements have only one rate signature and all the port rates are 1.

4. ***Modal model with a local MultirateFSMDirector***: $N = 1$, R_0 = the rate signature of any refinement. *Precondition*: all the refinements have only one and same rate signature.

5. ***Modal model with a local HDFFSMDirector***: $N = \sum_{j=0}^{M-1} N_j$ where N_j is the number of configurations of the jth refinement, M is the number of refinements. For $i = 0, \ldots, N-1$, R_i = the rate signature of the kth refinement in its qth configuration, where k is derived from $i - \sum_{j=0}^{k-1} N_j \geq 0$ and $i - \sum_{j=0}^{k} N_j < 0$, q is derived from $q = i - \sum_{j=0}^{k-1} N_j$.

6. ***Composite actor with a local HDFDirector***: $N = \prod_{j=0}^{M-1} N_j$ where N_j is the number of configurations of the jth contained actor, M is the number of contained actors. For $i = 0, \ldots, N-1$, R_i = the rate signature of the composite actor inferred from the local SDF schedule when the jth contained actor, for $j = 0, \ldots, M-1$, presents its rate signature in its k_jth configuration , where k_j is derived from $i = \sum_{j=0}^{M-1} \left(k_j \prod_{q=j+1}^{M-1} N_q \right)$.

The above computation is performed in the generatePreinitializeCode() method of each director

helper. After the computation is complete, the helper for each actor has recorded its number of configurations, the rate signature presented to the outside in each configuration, and the local SDF schedule in each configuration. During this step, the maximum capacity of each receiver under all schedules is also recorded, except the receivers of modal controllers, whose maximum capacity must be computed in the next step, because a modal controller may potentially access all received data during one global iteration (of the whole model) and the number of times the modal model containing the modal controller is fired in one global iteration is not available in the bottom-up traversal.

The next step is to traverse the model structure in a top-down fashion in the createOffsetVariablesIfNeeded() method. The helper for each composite actor contains an integer array: _firingsPerGlobalIteration, the length of which is equal to the number of configurations of the actor. Each element in the array represents the maximum number of times the actor is fired in each configuration during one global iteration. It is computed in the following way. Each element of the array is initialized to be 1 for the top composite actor. If a composite actor is internally controlled by an HDFDirector, each contained actor derives its _firingsPerGlobalIteration from the _firingsPerGlobalIteration of its container together with the number of times this actor is fired in a local SDF schedule. If a composite actor is internally controlled by an HDFFSMDirector, each contained refinement derives its _firingsPerGlobalIteration directly from the _firingsPerGlobalIteration of its container. Imagine a controller receives a large number of tokens in one global iteration and correspondingly a large chunk of memory must be allocated in the generated code. One way to get around this is to directly analyze the guard expression and find out how far back the controller needs to access the received tokens and only allocate that amount of memory. This should be easy to do if *constant* array indexes are used to access the received tokens. However, if that is not the case, then the optimizing approach might not work. The current implementation allocates the maximum amount of memory ever needed in one global iteration, therefore correctness is guaranteed.

Upon completing the previous two traversals, the HDF model has been analyzed and it's ready to generate the target code. The code for variable declarations and initializations is generated as usual. The bulk of the code is generated in the generateFireCode() method. The interesting part is where the code for making mode transitions is generated. In a modal model containing a MultirateFSMDirector or an FSMDirector, the mode transition code is generated in the generateFireCode() method and therefore executed at the end of each firing of the modal model. In a modal model containing an HDFFSMDirector, only the code for setting some "fired" boolean variables is generated in the generateFireCode() method; the actual code for making mode transitions is generated in the generateModeTransitionCode() method and appended after the code corresponding to one global iteration. When the generated code is executed, those "fired" variables essentially record a trace about what part of the model is executed in each global iteration. The mode transition phase follows this trace and executes the code for making mode transitions and updating the variables representing the current state and the current configuration.

Several interesting demos for the HDF code generation are presented next. The Merge demo in $PTII/ptolemy/domains/hdf/demo/Merge shows an interesting way to merge two increasing sequences of numbers into one increasing sequence. The AdaptiveCoding demo in $PTII/ptolemy/domains/hdf/demo/AdaptiveCoding shows the modeling of two modes of Hamming codec, a (7, 4) Hamming code and a (3,1) Hamming code. The model switches between the two coding/decoding schemes depending on the channel condition.

Appendix C: CodeStream and CodeGen Types

C.1 The CodeStream Mechanism

C.1.1 Code Block Structure

For a helper, a code block is uniquely defined by its signature which consists of its code block name and the number of parameters it takes. A proper code block should have the following grammar:

```
/*** CodeBlockName [($parameter1, $parameter2, ...)] ***/
CodeBlockBody
/**/
```

Parameterized code blocks can contain parameters which the user can specify. Parameter substitution syntax is straight-forward string pattern substitution, so the user is responsible for declaring unique parameter names. For example, a code block is declared to be the following:

```
/*** initBlock ($arg) ***/
if ($ref(input) != $arg) {
   $ref(output) = $arg;
}
/**/
```

If the helper invokes the appendCodeBlock() method with a single parameter, which is the integer 3,

```
ArrayList args = new ArrayList();
args.add(Integer.toString(3));
appendCodeBlock("initBlock", args);
```

then after parameter substitution, the code block would become:

```
if ($ref(input) != 3) {
   $ref(output) = 3;
}
```

C.1.2 Overriding and Overloading

CodeStream supports overriding superclass code blocks with same signature. However, it does not support call to superclass code blocks that have been overridden (i.e., there is not a function call like super()), so far. It also supports overloading code blocks with different number of parameters.

C.2 Type Conversion: CodeGen Types

Ptolemy II supports a variety of types that are different from the target language. It sometimes dynamically converts tokens in the execution of the model. Thus, the code generator has to deal with some type conversion issues. First, it has to generate code that represents the different PTII token types and functionality of the tokens in the target language. Second, it has to be able to convert one type to another that is higher in the type lattice. Thirdly, the code generator has to know where type conversions need to occur in order to generate compilable code and code that produces the correct result.

For each Ptolemy token type, there is an associated codegen Token type. In the $PTII/ptolemy/codegen/kernel/type directory, there is a target-language specific file which contains functionality code for each Ptolemy type. The code generator uses the code stream mechanism to harvest the code blocks in these files. E.g. The Int.c file contains code to operate on an integer token.

```
/***negateBlock***/
Token Int_negate(Token this, ...) {
   this.payload.Int = -this.payload.Int;
   return this;
}
/**/
```

The code generator handles three kinds of type conversion. The first case is converting between primitive (non-Token) types. The target languages often support some of the Ptolemy types as primitive types. The code generator can take advantage of the language constructs to avoid storage and processing time overhead by using these primitive types. For example, the c code generator uses int, double and char array (char*) to represent the Ptolemy int, double and string types. The c code generator generates functions to convert int and double to char* type (because string is higher than int and double in the type lattice).

```
char* InttoString (int i) {
   char* string = (char*) malloc(sizeof(char) * 12);
   sprintf((char*) string, "%d", i);
   return string;
}
```

The second case is to upgrade a primitive type to a Token type. This happens often in a multiport connection where the sink port (multiport) is resolved to type ‘general’ and the source ports are resolved to concrete types, like int, string, etc. The code generator takes care of this by using the $new() macro to create a new Token for the given primitive value (there is an associated codegen Token type for each Ptolemy type including the primitive type).

The third case is to convert between different Token types. The functionality code for each codegen type has a convert function that converts the given token. For example, in $PTII/ptolemy/codegen/kernel/type/String.c, the convertBlock looks like[1]:

```
/***convertBlock***/
Token String_convert(Token token, ...) {
   char* stringPointer;

   switch (token.type) {
      #ifdef TYPE_Boolean
      case TYPE_Boolean:
         stringPointer = BooleantoString(token.payload.Boolean);
         break;
      #endif

      #ifdef TYPE_Int
      case TYPE_Int:
         stringPointer = InttoString(token.payload.Int);
         break;
      #endif

      #ifdef TYPE_Double
      case TYPE_Double:
         stringPointer = DoubletoString(token.payload.Double);
         break;
      #endif

      default:
         // FIXME: not finished
```

1. The user should use the $typeFunc() (rather than $tokenFunc()) macro to call the convert() function because convert() is a static/class function. If the $tokenFunc() macro is used, the convert function of the given token’s type is invoked rather than the String type. Since the token is, of course, the same type of its own type; thus, no conversion occurs.

```
            fprintf(stderr, "String_convert():
                  Conversion from an unsupported type. (%d)\n", token.type);
            break;
      }
      token.payload.String = stringPointer;
      token.type = TYPE_String;
      return token;
}
/**/
```

C.3 Examples

C.3.1 How to write the helper for a polymorphic actor (AddSubtract)

The following is the code generation fire method in the AddSubtract helper (AddSubtract.java):

```
public String generateFireCode() throws IllegalActionException {
   super.generateFireCode();

   ptolemy.actor.lib.AddSubtract actor =
      (ptolemy.actor.lib.AddSubtract) getComponent();

   Type type = actor.output.getType();
   boolean minusOnly = actor.plus.getWidth() == 0;

   ArrayList args = new ArrayList();
   args.add(new Integer(0));

   if (type == BaseType.STRING) {
      _codeStream.appendCodeBlock("StringPreFireBlock");
      for (int i = 0; i < actor.plus.getWidth(); i++) {
         args.set(0, new Integer(i));
         _codeStream.appendCodeBlock("StringLengthBlock", args);
      }
      _codeStream.appendCodeBlock("StringAllocBlock");
   } else {
      String blockType = isPrimitive(type) ? "" : "Token";
      String blockPort = (minusOnly) ? "Minus" : "";

      _codeStream.appendCodeBlock(blockType + blockPort + "PreFireBlock");
   }

   String blockType = isPrimitive(type) ? codeGenType(type) : "Token";

   for (int i = 1; i < actor.plus.getWidth(); i++) {
      args.set(0, new Integer(i));
      _codeStream.appendCodeBlock(blockType + "AddBlock", args);
   }

   for (int i = minusOnly ? 1 : 0; i < actor.minus.getWidth(); i++) {
      args.set(0, new Integer(i));
      _codeStream.appendCodeBlock(blockType + "MinusBlock", args);
   }

   _codeStream.appendCodeBlock("PostFireBlock");

   return processCode(_codeStream.toString());
}
```

These are the corresponding code blocks in the code template (AddSubtract.c):

```
/***PreFireBlock***/
$actorSymbol(result) = $ref(plus#0);
/**/
```

```
/***MinusPreFireBlock***/
$actorSymbol(result) = -$ref(minus#0);
/**/

/***TokenPreFireBlock***/
$actorSymbol(result) = $ref(plus#0);
/**/

/***TokenMinusPreFireBlock***/
$actorSymbol(result) = $tokenFunc($ref(minus#0)::negate());
/**/

/***IntAddBlock($channel)***/
$actorSymbol(result) += $ref(plus#$channel);
/**/

/***IntMinusBlock($channel)***/
$actorSymbol(result) -= $ref(minus#$channel);
/**/

/***DoubleAddBlock($channel)***/
$actorSymbol(result) += $ref(plus#$channel);
/**/

/***DoubleMinusBlock($channel)***/
$actorSymbol(result) -= $ref(minus#$channel);
/**/


/***BooleanAddBlock($channel)***/
$actorSymbol(result) |= $ref(plus#$channel);
/**/

/***StringPreFireBlock***/
$actorSymbol(length) = 1;// null terminator.
/**/

/***StringLengthBlock($channel)***/
$actorSymbol(length) += strlen($ref(plus#$channel));
/**/

/***StringAllocBlock***/
$actorSymbol(result) = (char*) realloc($ref(output), $actorSymbol(length));
strcpy($actorSymbol(result), $ref(plus#0));
/**/

/***StringAddBlock($channel)***/
strcat($actorSymbol(result), $ref(plus#$channel));
/**/

/***TokenAddBlock($channel)***/
$actorSymbol(result) = $tokenFunc($actorSymbol(result)::add($ref(plus#$channel)));
/**/

/***TokenMinusBlock($channel)***/
$actorSymbol(result) = $tokenFunc($actorSymbol(result)::substract($ref(minus#$channel)));
/**/

/***PostFireBlock***/
$ref(output) = $actorSymbol(result);
/**/
```

C.3.2 How to write the helper that extends another concrete helper

The following is the Uniform helper, which extends the RandomSource helper:

```
// Uniform.java
```

```
public class Uniform extends RandomSource {

   public Uniform(ptolemy.actor.lib.Uniform actor) {
      super(actor);
   }

   protected String _generateRandomNumber() throws IllegalActionException {
      return _generateBlockCode("randomBlock");
   }
}
```

This is the Uniform helper's code template file (Uniform.c):

```
/*** randomBlock ***/
$ref(output) = (RandomSource_nextDouble(&$actorSymbol(seed))
      * ($val(upperBound) - $val(lowerBound))) + $val(lowerBound);
/**/
```

It references the function RandomSource_nextDouble from its parent's code template. The following is the code template for the RandomSource helper (RandomSource.c):

```
/*** sharedBlock ***/
int RandomSource_next(int bits, double* seed) {
   *seed = (((long long) *seed * 0x5DEECE66DLL) + 0xBLL) & ((1LL << 48) - 1);
   return (int)((signed long long) *seed >> (48 - bits));
}

double RandomSource_nextDouble(double* seed) {
   return (((long long)RandomSource_next(26, seed) << 27)
         + RandomSource_next(27, seed)) / (double)(1LL << 53);
}
/**/

/*** gaussianBlock ***/
double RandomSource_nextGaussian(double* seed, boolean* haveNextNextGaussian, double* nextNextGaussian) {
   double multiplier;
   double v1;
   double v2;
   double s;

   if (*haveNextNextGaussian) {
      *haveNextNextGaussian = false;
      return *nextNextGaussian;
   } else {
      do {
         v1 = 2 * RandomSource_nextDouble(seed) - 1;   // between -1.0 and 1.0
         v2 = 2 * RandomSource_nextDouble(seed) - 1;   // between -1.0 and 1.0
         s = v1 * v1 + v2 * v2;
      } while (s >= 1 || s == 0);

      multiplier = sqrt(-2 * log(s)/s);
      *nextNextGaussian = v2 * multiplier;
      *haveNextNextGaussian = true;
      return v1 * multiplier;
   }
}
/**/

/*** setSeedBlock0($hashCode) ***/
$actorSymbol(seed) = $actorSymbol(seed) = time (NULL) + $hashCode;
/**/

/*** setSeedBlock1 ***/
/* see documentation from http://java.sun.com/j2se/1.4.2/docs/api/java/util/Random.html#setSeed(long) */
//this.seed = (seed ^ 0x5DEECE66DL) & ((1L << 48) - 1);
```

```
$actorSymbol(seed) = ((long long) $val(seed) ^ 0x5DEECE66DLL)  & ((1LL << 48) - 1);
/**/

/*** preinitBlock ***/
double $actorSymbol(seed);
/**/
```

Since the Uniform helper does not override any of code generation methods, the parent's methods are called instead. The sharedBlock, setSeedBlock and preinitBlock code blocks are harvested from the RandomSource code template. The child helper may override its parent's code blocks while using its parent's code generation methods. A child helper can also override the code generation method. Neither way of overriding the super class's code harvesting increases run-time overhead in the generated code. The code generator handles method /code block overrides during compile time and add no extra logic in the final code.

8

Copernicus

Authors: Steve Neuendorffer
Christopher Brooks
Ankush Varma
Shuvra S. Bhattacharyya

8.1 Introduction

The copernicus package is an infrastructure for building code generators for Ptolemy II models by analyzing the Java byte code by using the Soot[1] package. The primary idea behind copernicus was to reuse preexisting actors written in Java thus relieving the developer from the task of rewriting actors for code generation. In contrast to copernicus, Ptolemy II also includes the codegen package, documented in chapter 7, which is a template based code generation system that requires the developer to rewrite actors. For further information about the various Ptolemy code generators, see the Ptolemy II FAQ at http://ptolemy.eecs.berkeley.edu/ptolemyII/ptIIfaq.htm.

The basic design goal of the copernicus package was to provide a common interface to different Soot based code generators and consolidate some of the basic argument handling and default parameters. Several different code generators of varying complexity have been implemented. There is also quite a bit of testing infrastructure for integrating code generation with the nightly build system.

The basic infrastructure is implemented in the copernicus.kernel package. The Copernicus class contains a main function suitable for invocation from the command line. The GeneratorAttribute class represents code generation parameters that can be persistently added to a model for unusual configurations of a code generator. The KernelMain class is a base class from which classes for various code generators can be derived. Instances of these subclasses are instantiated and executed in a code generation job.

The Copernicus class itself is invoked to begin code generation from a model. If invoked from the command line, it reads command line arguments to determine various code generation options and an

1. Soot is a Java Optimization Framework that works on .class files, see http://www.sable.mcgill.ca/soot/

MoML file to load a model from. If invoked via various static methods, code generation options are assumed to be passed in through a GeneratorAttribute. Default code generation options are specified in the Generator.xml file. One of the code generation parameters determines the code generator to execute. The class representing the code generator is loaded through reflection and invoked through the KernelMain base class.

The KernelMain base class provides default behavior for most code generators. It performs static analysis, such as type resolution and scheduling by invoking the Manager.preinitializeAndResolveTypes() method and then passes control to the specific code generator.

8.1.1 Default options

Most code generators share common options. The following options are defined by default in generator.xml.

codeGenerator: The code generator to run.

codeGeneratorClassName: The class that is instantiated to execute a particular code generator. This class is expected to be a subclass of ptolemy.copernicus.kernel.KernelMain.

compile: If true, compile the generated code. The default is true.

show: If true, then show the generated code. The default is true.

run: If true, then run the generated code. The default is true.

ptII: The location of the Ptolemy II classes. The default is the value of the ptolemy.ptII.dir Java system property

ptIIUserDirectory: The top level directory to write the code in. The default is the value of the ptII parameter. The code will appear in 'ptIIUserDirectory/targetpath'.

targetPackage: The package to generate code in. The default is the model name

targetPath: The path relative to the ptIIUserDirectory to generate code in. The default is the “cg” subdirectory of the particular code generator.

outputDirectory: The directory that code will be generated in. By default this is the targetPath parameter appended to the ptIIUserDirectory path.

modelPath: The path to the model, including the .xml extension. The modelPath parameter is converted to a URL internally before use.

compileOptions: User supplied arguments to be passed to the code generator. Defaults to the empty string.

javaClassPath: The Java class path, converted to a string.

runCommandTemplateFile: The template file that contains the command to run the generated code.

runOptions: User supplied arguments to be passed to the command that will run the generated code. Defaults to the empty string.

sootDir: The directory that contains the soot jar files. Defaults to the value of the ptII parameter + "/lib"

sootClasses: The location of sootclasses.jar, jasminclasses.jar and the Java system jar (usually rt.jar). The necessaryClassPath parameter may end up duplicating some of the elements of this parameter.

watchDogTimeout: The number of milliseconds that code generation will run for. Defaults to 720000, which is 12 minutes. The watchdog is used to prevent the code generator and the generated code from hanging the nightly build.

output: The filename to redirect the standard output stream of the code generator to. This is used, for example, in the nightly build to provide easily parseable error messages. If the value is not set, then the output will not be redirected.

8.2 Copernicus Java Code Generator

The Copernicus Java code generator is implemented by the copernicus.java package. This code generator targets the generation of self-contained Java code optimized for code size, memory usage and execution speed. The Java code generator leverages the Soot compiler framework to parse the bytecode for each atomic actor in the model. The actors are then specialized according to their context in the model.

The Copernicus Java code generator operates in several phases, and the output of each phase is a partially specialized model. The output from the intermediate phases can be generated by setting the *snapshots* parameter to be true. The first snapshot consists of self-contained code specialized to the domains in the model. The second snapshot is additionally specialized to the parameter values in the model, while the third is specialized to the structure of the model. The fourth snapshot eliminates all references to Ptolemy named objects in the model, resulting in self-contained code without component interfaces. The final generated code has also been specialized for data types and contains no references to Ptolemy tokens.

One of the goals of the Copernicus Java code generator was to avoid separate specifications for simulation and code generation wherever possible. The Copernicus Java code generator operates by transforming Java actor specifications (actor classes) and on Java data type specifications (token classes). In most cases, new actor and token classes will be leveraged transparently by the code gener-

ator. Unfortunately, domain specifications are not as easily reused and the Copernicus Java code generator contains “re-implementations” of domains for code generation. This allows for more efficient code to be generated, at the expense of duplicating aspects of existing Director and Receiver code, and making it more difficult for new domains to be implemented in code generation.

In order for existing actor code to be leveraged by the code generator, it assumes that the code is written according to the Ptolemy style for writing actors. This style assumes naming conventions for the public fields of an actor class that refer to parameters and ports of the actor. The code generator also assumes that the ports and parameters of an actor are created in the class constructor and not modified later. Some actors do not fit these constraints and cannot be used directly in the code generator. Such actor classes cannot be used directly by the code generator, although in some cases we have been able to have the code generator deal specially with such actors. In other cases, the actor class fits the constraints but cannot be effectively specialized using generic techniques. Such actors can also be dealt with specially by the code generator to more effectively generate code.

8.2.1 Software Architecture

The Copernicus Java code generator consists of a large number of individual transformation steps, which will not be described here. These transformation steps are implemented by classes extending the SceneTransformer class, or the BodyTransformer class. Two key points of extensibility are provided for generating domain code and for generating actor code to replace unspecializable actor classes.

Code generation for specific domains is handled by various implementations of the DomainCodeGenerator interface. An implementation of this interface is responsible for generating domain interaction code for a particular composite actor in the hierarchy, including code to invoke the methods of various actors and domain-specific communication structures. Currently, the following domains are handled:

Synchronous Dataflow (SDF): The SDF implementation transforms the SDF schedule into Java code that invokes the actors in a model. Fixed size arrays are generated for communication buffers dedicated to each relation in the model, and communication methods are replaced with circularly indexed addressing into the communication buffers.

Hybrid Systems (HS): The hybrid systems director deals with modal models. Currently, only the subset that is useful in modal SDF models is implemented.

Giotto: The Giotto implementation interfaces directly with the Java output of the Giotto compiler. It generates classes with static methods used for communication by the Giotto compiler and generates a .giotto file that describes the classes implementing the various Giotto tasks. The Giotto compiler compiles this file into a class that implements the Giotto task scheduling model.

Code Generation for actors is handled various implementations of the AtomicActorCreator interface. An implementation of this interface generates a self-contained class for a particular actor. The default implementation of this interface, the GenericAtomicActorCreator class, simply copies the existing actor specification code. Other implementations of the interface deal with generating code for specific actors. Currently the following actors are handled specially:

Expression: The standard implementation of this actor builds a parse tree for the expression and traverses the parse tree to evaluate it at run time. This actor is handled specifically by the ExpressionCreator class for two reasons. Primarily, the parse tree is convenient way of representing an arbitrary expression, but much simpler code can be generated for a specific expression. Secondarily, the parse tree complicates other transformations that specialize communication between actors and data types.

FSMActor: The standard implementation of this actor builds parse trees for every expression in a

model, and suffers from the same drawbacks as the Expression actor. The FSMActor also attempts to deal with run-time modifications of the finite-state machine in an efficient manner, which is not necessary in generated code. This actor is handled specifically by the FSMCreator class.

Many actors are not handled specifically, but should be. Here is a short list:

MathFunction: This actor creates and deletes its ports based on a parameter value. In generated code will likely not happen (since the parameter value is not likely to change), but the GenericAtomicActorCreator is not smart enough to deal with ports that are not created in the constructor. It is likely easiest to handle this actor by handling it specially and checking that the parameter value does not change using reconfiguration analysis.

TypeTest: This actor tests the type system, but has no run-time behavior. It is problematic because it iterates over all of the actors in a model, which is currently not supported by the code generation mechanism. It could probably be checked statically and ignored in generated code.

RecordAssembler and RecordDisassembler: These actors iterator over their input and output ports to construct a record. They could either be dealt with specially, or the code generator could be improved to unroll iterators over ports.

ExpressionToToken and ExpressionReader: These actors operate in a similar way to the expression actor, except that the expression is received from an input port. Because of this, code generation will not work. It is not clear how to make this actor work nicely with type specialization.

8.2.2 Generated Code

The code generated from the Copernicus Java code generator is a set of self-contained Java .class files with a command-line interface. A makefile is automatically generated with a large number of rules for manipulating the generated code. The makefile rules are:

runJava: Run the generated code.

compareAll: Run a series of comparisons between the simulation model, the generated code, and the obfuscated version of the generated code comparing code size, execution speed, and memory usage.

treeShake: Generate a self-contained .jar file containing only code necessary for the generated code. This rule uses reachable method information gained through static analysis in the code generator, if possible.

treeShakeByRunning: Generate a self-contained .jar file containing only code necessary for the generated code. This rule executes the generated code and extracts information from the virtual machine about which classes were loaded at runtime.

runTreeShake: Run the generated code from the self-contained .jar file.

profileTreeShake: Run the generated code from the self-contained .jar file with profiling options to report runtime memory usage. An average of several runs is reported.

treeShakeWithoutCodegen: Generate a self-contained .jar file containing only code necessary for executing the simulation model. This rule executes the generated code and extracts information from the virtual machine about which classes were loaded at runtime.

runTreeShakeWithoutCodegen: Run the original simulation model from the self-contained .jar file.

profileTreeShakeWithoutCodegen: Run the original simulation model from the self-contained .jar file with profiling options to report runtime memory usage. An average of several runs is reported.

obfuscate: Run the Jode obfuscator on the generated code, to minimize the size of the generated

.jar file.

runObfuscate: Run the obfuscated version of the generated code.

profileObfuscate: Run the obfuscated version of the generated code from the self-contained .jar file with profiling options to report runtime memory usage. An average of several runs is reported.

gcj: Compile the generated code into a native executable using gcj. Note: this will likely only work for simple models, as the gcj standard Java libraries are far from complete.

8.2.3 Java Code Generation Demonstrations

Below are several demonstrations of the Copernicus Java code generator. Our canonical Java code generation model is the OrthogonalCom model located in $PTII/ptolemy/domains/sdf/demo/OrthogonalCom/OrthogonalCom.xm, see Figure 8.1l.

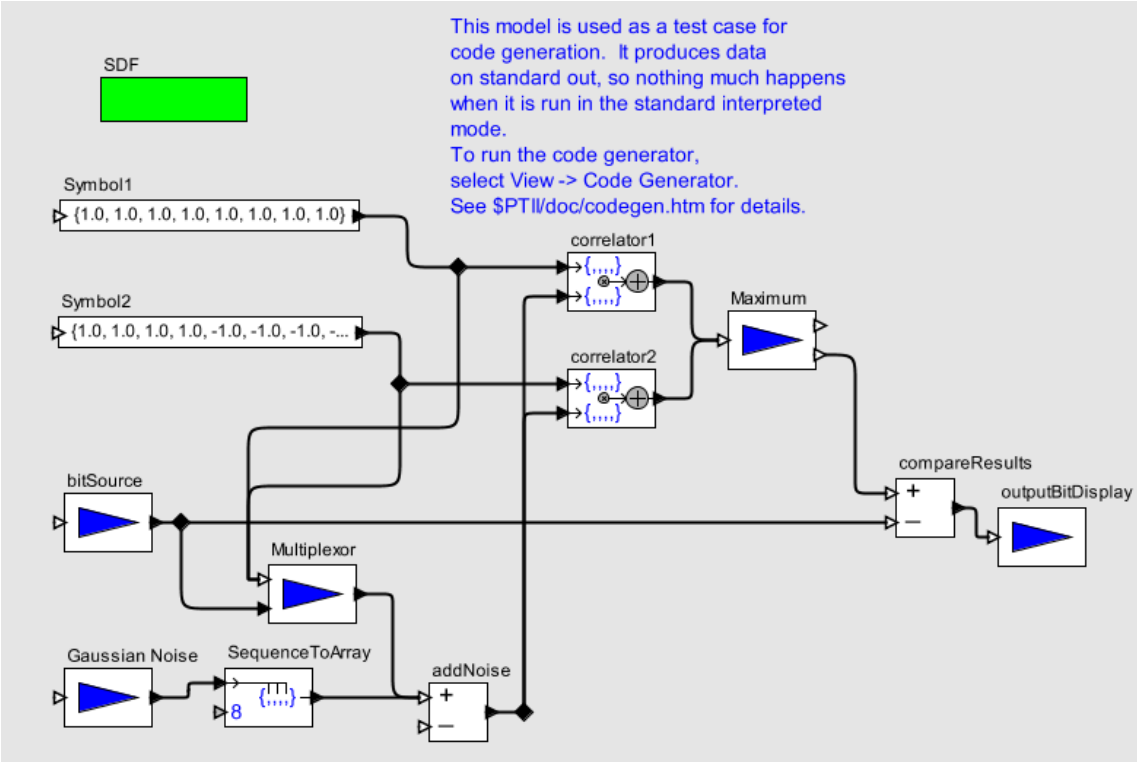


FIGURE 8.1. The Orthogonal Communication model, used as a Java code generation example.

We use the copernicus shell script, located at $PTII/bin/copernicus to run the Copernicus Java code generator:

```
$PTII/bin/copernicus -codeGenerator java $PTII/ptolemy/domains/sdf/demo/OrthogonalCom/OrthogonalCom.xml
```

The above command will generate voluminous output and eventually create .class files in $PTII/ptolemy/copernicus/java/cg/OrthogonalCom and run the generated code.

Treeshaking.

The copernicus script also generates a makefile in the output directory that contain rules to perform operations like treeshaking and obfuscation. Treeshaking is an optimization where we run the model, note what .class files are loaded and then place those .class files in a jar file. Treeshaking is not perfect, since if the model is running from the jar file and later throws an exception the error handlers and other .class files might not be present. Obfuscation is an optimization that shortens class and method names so as to decrease the size of the jar file.

We use Jode to obfuscate the code. Jode is available from http://jode.sourceforge.net/. Unfortunately, Jode is distributed under the GNU General Public License (GPL), so we do not include it in the Ptolemy release. In addition Jode might not work with Java 1.5. To set up Jode:

1. Download Jode so that $PTII/vendors/jode/1.1.1/jode.jar is present.

2. Re run configure with:
   ```
   cd $PTII
   ./configure
   ```
   Then go back to the output directory and run make:
   ```
   $PTII/ptolemy/copernicus/java/cg/OrthogonalCom
   make compareAll
   ```

Setting the iterations.

The truly observant will have noticed that the various versions of the model ran very quickly and that it is difficult to compare the time performance of the different versions. The solution is to increase the number of iterations so that we can see differences in time performance. $PTII/ptolemy/copernicus/kernel/KernelMain.java describes how to set the iterations:

```
*  If the director is an SDF director, then the number of
* iterations is handled specially.  If the director is an SDF
* director and a parameter called "copernicus_iterations" is
* present, then the value of that parameter is used as the
* number of iterations.  If the director is an SDF director, and
* there is no "copernicus_iterations" parameter but the
* "ptolemy.ptII.copernicusIterations" Java property is set, then
* the value of that property is used as the number of
* iterations.
```

So, we can either edit the model and add a copernicus_iterations parameter, or else we can run copernicus with the ptolemy.ptII.copernicusIterations property set. In this example, we set the property and rerun copernicus

```
export USERJAVAPROPERTIES=-Dptolemy.ptII.copernicusIterations=100
$PTII/bin/copernicus -codeGenerator java $PTII/ptolemy/domains/sdf/demo/OrthogonalCom/OrthogonalCom.xml
```

Then, we cd to the generated directory and re run the comparison:

```
cd $PTII/ptolemy/copernicus/java/cg/OrthogonalCom
make compareAll
```

The results is that the different versions of the models run for 1000 iterations and we can compare

the times:

Table 12: Jar file sizes and elapsed time

Optimizations	Jar file size	Time
interpreted code with treeshaking	~750 k bytes	4787 ms.
copernicus generated code with treeshaking	~77 k bytes	230 ms.
copernicus generated code with treeshaking and obfuscation	~40 k bytes	217 ms.

Note that these results are not particularly rigorous, see [118] for a more formal analysis.

8.3 Copernicus C Code Generator

The Copernicus C code generator [147] is implemented by the copernicus.c package. This code generator targets the generation of self-contained C code by post-processing the result of the Copernicus Java code generator, and performing further code size optimizations. The Copernicus C code generator leverages the Soot compiler framework to parse the bytecode representation for each class to be compiled. Note that work on the Copernicus C code generator has stopped, the Ptolemy group is working on a template based code generator.

Within the Ptolemy II framework, the C code generator takes the class files generated by copernicus.java as input.The C code generator can also be used as a stand-alone Java-to-C compiler to generate C code for arbitrary Java programs.

8.3.1 Code Generation

The main steps in the code generation algorithm are as follows:

1. Read in main class file using Soot.
2. Use CallGraphPruner to compute the set of required methods, classes and fields.
3. Generate .c and .h files for the main class(es).
4. Generate a .c file containing code for initialization and setup.
5. Generate .c and .h files for all required Java library classes in a separate directory (named j2c_lib by default). Note that these only contain code for required methods and fields, to minimize code size. The code for each method is generated by converting the jimple statements for the method's body atomically into the appropriate C constructs.
6. Generate a makefile for compiling the code into an executable.

8.3.2 The Code Pruning Algorithm

The Soot framework is used to create a Call Graph of the application. This is a graph with methods as the nodes, and calls from one method to another as directed edges.

At first glance, it seems that the transitive closure of the methods in the main class should represent all methods that can be called. However, this is not so, because the first time the field or method of a class is referenced, its class initialization method is also invoked, and this can reference other methods or fields in turn.

The method call graph also contains an edge from a method to every possible target of method calls in it. The number of such targets can be large for polymorphic method calls. A more sophisticated analysis can trim the method call graph by removing some of the edges corresponding to polymorphic invocations.

We use Soot's Variable Type Analysis (VTA) to perform this call graph trimming. This analysis computes the possible runtime types of each variable using a reaching type analysis, and uses this information to remove spurious edges.

Computing the Set of Required Entities.

From the analysis mentioned above, the set of all possible required classes, methods and fields (collectively grouped as *entities*) can be statically computed. We use a set of rules to determine which classes are required.

1. A set of compulsory entities is always required. This includes the `System.initializeSystemClass()` method, all methods and fields of the `java.lang.Object` class (since it is the global superclass) and the main method of the main class to be compiled.
2. If a method *m* is required, the following also become required: the class declaring *m*, all methods that may possibly be called by *m*, all fields accessed in the body of *m*, the classes of all local variables and arguments of m, the classes corresponding to all exceptions that may be caught or thrown by *m*, and the method corresponding to *m* in all required subclasses of the class declaring *m*.
3. If a field *f* is required, the following also become required: the class declaring *f*, the class corresponding to the type of *f* (if any) and the field corresponding to *f* in all required subclasses of the class declaring it.
4. If a class *c* is required, the following also become required: all superclasses of *c*, the class initialization method of *c*, and the instance initialization method of *c*.

Interfaces are treated as classes. A worklist-based algorithm can be used to add to the set of required entities until no additional entities can be found by application of these rules. Together, rules 2, 3 and 4 encapsulate all possible dependencies between entities. This makes the set of required entities self-contained.

8.3.3 Limitations

The restrictions imposed C-based static compilation strategy are:

- Dynamic Loading and Reflection are not supported.
- The generated executable runs as a user process, so applications that rely on a JVM as a buffer between them and the platform for security cannot be guaranteed to run correctly.

The further limitations of the current implementation are:

- No support for threads.
- GUI-based functions are currently not implemented.
- Certain java classes are not currently supported, because the native methods for them need to be

coded. The list of these is maintained in the OverriddenMethodGenerator class.

8.3.4 Options

There are a number of command-line options available:

- verbose: *true/false* Turns verbose mode on or off.
- compileMode: *singleClass* compiles only the given class, *full* generates all required files.
- pruneLevel: *0* no code pruning done, *1* code pruning done by CallGraphPruner.
- vta: *true/false* Whether or not to perform Variable Type Analysis.
- lib: the path to the directory where library of generated files should be stored.
- gcDir: stores the path to the directory containing the garbage collector. Not using this option turns the collector off.
- target: The target platform. A blank refers to a generic POSIX-like system including Cygwin installations. *C6000* The TMS320C6xxx series of processors.
- runtimeDir: The path to the runtime directory.
- ptII: The path to the ptII directory.
- compulsoryMethods: A semicolon-separated list of methods for which code must always be generated. If more than one such entity is to be specified, the entire list may be enclosed within double quotes. The complete method subsignature of the form *returnType class.method(arg1, arg2, …)* must be specified.
- cFlags: The GCC flags to be used in the makefile.
- reportEntities: *true/false* whether to output a summary of the number of classes, methods and fields (entities) generated.

8.3.5 Directory structure

The main subdirectories in copernicus.c are:

- runtime: Contains a small amount of C code that provides basic functionality. This is linked in while generating the executables.
- runtime/native_bodies: C code for native methods.
- runtime/over_bodies: C code for methods with custom code.
- test: Various test programs.
- testOutput: Auto-generated C code.

8.3.6 Code Flow

The following UML diagram shows the various classes that populate copernicus.c. This is a relatively complex package, so many implementation details have been abstracted out. Complete descriptions of all classes and their members are available in the API, and we attempt to provide an insight into the higher-level structure of the package here.

- Protected and private methods are not shown, unless they are central to the functionality of the class.
- Unimportant external superclasses are not shown here.
- Public methods that are not central to the operation of the class are omitted.
- CSwitch has a caseXXX method for each kind of Jimple statement XXX. These are shown as a

single entry in the figure.

- The methods in ExceptionTracker are omitted. This class tracks the current exceptions and works closely with MethodCodeGenerator. However, the C implementation of Java exceptions is complex and is not discussed here.

The dashed arrows in the UML diagram represent the coarse-grained code flow in copernicus.c. The entry class is JavaToC when used in stand-alone mode, and Main when copernicus.c is used as a ptolemy code-generation back-end.

JavaToC reads in a Java class file and implicitly converts it to the Soot Jimple format. Then it calls RequiredFileGenerator, MakeFileGenerator and MainFileGenerator.

RequiredFileGenerator uses CallGraphPruner to compute the set of required classes, methods and fields. Then it calls ClassFileGenerator, HeaderFileGenerator and StubFileGenerator on each required class.

ClassFileGenerator creates the .c file containing all the function definitions. Each of these function definitions is created by MethodCodeGenerator. MethodCodeGenerator calls CSwitch on each Jimple statement to find its C equivalent.

HeaderFileGenerator creates the .h file corresponding to the class. This consists of a class-specific C structure for the class (created by ClassStructureGenerator), an instance-specific C structure for the class (created by InstanceStructureGenerator) and various function declarations. MethodListGenerator is the class that “understands” inheritance to create the lists of constructors, inherited methods, new methods, private methods, etc.

StubFileGenerator creates a small “stub” of prototype declarations useful for breaking circular dependencies between classes.

MakeFileGenerator creates a makefile proving rules for compiling the generated C code into an executable.

MainFileGenerator creates a file that contains the C “main” method, which performs initialization functions, wraps the command-line arguments into the C equivalent of a Java string array and passes them to the “java” main method.

In addition, CNames converts Java names into unique legal C names, InterfaceLookupGenerator takes care of resolving interface method invocations, FileHandler provides file I/O utilities, Options stores configuration information, Context handles useful global information, NativeMethodGenerator handles native methods and OverriddenMethodGenerator allows user-defined code to override the compiler.

8.3.7 HOW TOs

Generating an executable from a Java ClassFile.

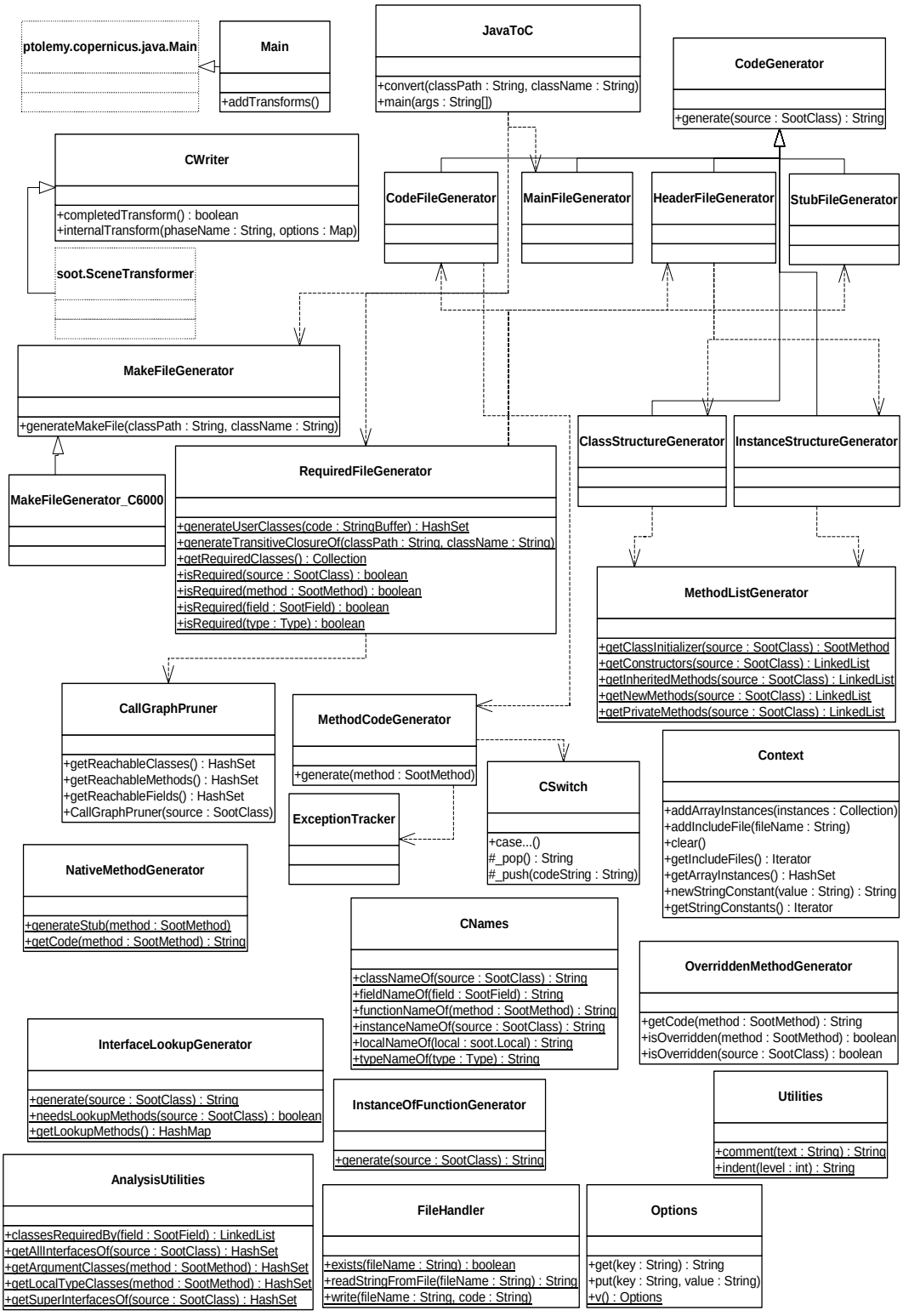
Put the classfile in c/test

```
cd test
java -classpath $classpath ptolemy.copernicus.c.JavaToC $classpath  className
```

(note that the classpath has to be specified twice)

```
make -s -f classname.make
```

Generating Code from a MoML model.


ptolemy.copernicus.java.Main
Main
+addTransforms()
JavaToC
+convert(classPath : String, className : String)
+main(args : String[])
CodeGenerator
+generate(source : SootClass) : String
CWriter
+completedTransform() : boolean
+internalTransform(phaseName : String, options : Map)
soot.SceneTransformer
CodeFileGenerator
MainFileGenerator
HeaderFileGenerator
StubFileGenerator
MakeFileGenerator
+generateMakeFileClassPath : String, className : String)
MakeFileGenerator_C6000
RequiredFileGenerator
+generateUserClasses(code : StringBuffer) : HashSet
+generateTransitiveClosureOf(classPath : String, className : String)
+getRequiredClasses() : Collection
+isRequired(source : SootClass) : boolean
+isRequired(method : SootMethod) : boolean
+isRequired(field : SootField) : boolean
+isRequired(type : Type) : boolean
ClassStructureGenerator
InstanceStructureGenerator
MethodListGenerator
+getClassInitializer(source : SootClass) : SootMethod
+getConstructors(source : SootClass) : LinkedList
+getInheritedMethods(source : SootClass) : LinkedList
+getNewMethods(source : SootClass) : LinkedList
+getPrivateMethods(source : SootClass) : LinkedList
CallGraphPruner
+getReachableClasses() : HashSet
+getReachableMethods() : HashSet
+getReachableFields() : HashSet
+CallGraphPruner(source : SootClass)
MethodCodeGenerator
+generate(method : SootMethod)
CSwitch
+case...()
#_pop() : String
#_push(codeString : String)
ExceptionTracker
Context
+addArrayInstances(instances : Collection)
+addIncludeFile(fileName : String)
+clear()
+getIncludeFiles() : Iterator
+getArrayInstances() : HashSet
+newStringConstant(value : String) : String
+getStringConstants() : Iterator
NativeMethodGenerator
+generateStub(method : SootMethod)
+getCode(method : SootMethod) : String
CNames
+classNameOf(source : SootClass) : String
+fieldNameOf(field : SootField) : String
+functionNameOf(method : SootMethod) : String
+instanceNameOf(source : SootClass) : String
+localNameOf(local : soot.Local) : String
+typeNameOf(type : Type) : String
OverriddenMethodGenerator
+getCode(method : SootMethod) : String
+isOverridden(method : SootMethod) : boolean
+isOverridden(source : SootClass) : boolean
InterfaceLookupGenerator
+generate(source : SootClass) : String
+needsLookupMethods(source : SootClass) : boolean
+getLookupMethods() : HashMap
InstanceOfFunctionGenerator
+generate(source : SootClass) : String
Utilities
+comment(text : String) : String
+indent(level : int) : String
AnalysisUtilities
+classesRequiredBy(field : SootField) : LinkedList
+getAllInterfacesOf(source : SootClass) : HashSet
+getArgumentClasses(method : SootMethod) : HashSet
+getLocalTypeClasses(method : SootMethod) : HashSet
+getSuperInterfacesOf(source : SootClass) : HashSet
FileHandler
+exists(fileName : String) : boolean
+readStringFromFile(fileName : String) : String
+write(fileName : String, code : String)
Options
+get(key : String) : String
+put(key : String, value : String)
+v() : Options

Move the xml model to c/test/simple

```
java ptolemy.copernicus.kernel.Copernicus -codeGenerator c model.xml
```

Writing Code for a Native Method.

Java requires certain *native* methods, which are methods implemented in platform-dependent code, typically written in another programming language such as C. The C code generator allows the user to specify C code for the body of any native method. At compile-time, this is integrated with the generated C code, allowing any C native methods to be fully supported. To do this:

1. Find the C name of that method (say *f00xx_abc*).
2. Create a file by this name (*f00xx_abc.c*) in runtime/native_bodies, containing the code for that method.
3. Add this method to the list of native methods in NativeMethodGenerator.

Overriding Code for an Existing Method.

It is also possible to override the C code generator and write custom C code for a given method instead. To do this:

1. Find the C name of that method (say *f00xx_abc*).
2. Create a file by this name (*f00xx_abc.c*) in runtime/over_bodies, containing the code for that method.
3. Add this method to the list of overridden methods in OverriddenMethodGenerator.

Note that the term *overridden* in this context does not refer to methods that are overridden through inheritance in Java classes.

Suppressing Code Generation for a Method, Class or Package.

To “turn off” code generation for a method, override it without creating code for it in runtime/over_bodies. For an entire class or package, list it in OverriddenMethodGenerator.isOverriddenClass(). This will generate methods with blank bodies and trivial return statements which will return 0 or NULL.

CAVEAT: Make sure that the returned values are not used. Referencing a NULL pointer will cause the executable to throw a segmentation fault.

8.4 Applet Code Generator

The Applet code generator takes a model and creates HTML files for use as a web based applet.

The applet generator reads template files that end in .in from $PTII/ptolemy/copernicus/applet substitutes keywords and writes out the files in the destination directory. Users may modify the template files to match their local setup

Making an applet available via the web is somewhat complex because the Java Plugin has two sections, one for Netscape, the other for Internet Explorer, so changes to the htm files must be replicated in both sections. The codebase and the location of the jar files also add to the problems.

If a model is named MyModel, and the user selects foo.bar as the package, then saving the model as an applet will create a directory called $PTII/foo/bar/MyModel and create the following files for that model:

makefile
: make demo will run appletviewer on the HTML files

MyModel.xml
: A local copy of the model

MyModel.htm
: An HTML file containing the code necessary to MyModel.xml

MyModelVergil.htm
: An HTML file containing the code necessary to display MyModel.xml graphically, using ptolemy.vergil.VergilApplet and in text format

8.4.1 Applet Code Generation demonstrations

Below are several demonstrations of the applet code generator. The code generator graphical user interface is difficult to use, so we recommend using the copernicus command instead of using the code generator GUI.

The OrthogonalCom model generates prints output to standard out, so when this model is run as an applet, the output will appear in the Java Plugin console. Instead, we generate an applet for the Butterfly model, which will generate display a nice plot. Note that the Butterfly model uses the Expression actor so that while we cannot use deep code generation on the Butterfly actor, we can generate an applet for this model.

Copernicus command - create applet within the Ptolemy tree.

To create the html file and open it with the browser:

```
cd $PTII/ptolemy/domains/sdf/demo/Butterfly
$PTII/bin/copernicus -codeGenerator applet Butterfly.xml
```

The HTML can be found in $PTII/ptolemy/copernicus/applet/cg/Butterfly.

Applet code generator GUI - create applet within the Ptolemy Tree.

If you would like to generate an applet in a directory within the Ptolemy tree using the experimental code generation GUI, follow these steps:

1. Open up the SDF Butterfly Model at $PTII/ptolemy/domains/sdf/demo/Butterfly/Butterfly.xml.
2. In the left hand actor tree, select More Libraries -> Codegen -> Copernicus and drag a SDF Codegenerator into the main window.
3. Double click on the SDF Codegenerator.
4. Change the CodeGenerator combo box from java to applet
5. Hit the Generate Button
6. The code generator will invoke an separate java process that generates code in $PTII/ptolemy/

copernicus/applet/cg/Butterfly and then opens the generated file with the browser.

Copernicus command - create applet outside the Ptolemy tree.

Usually, one wants to put an applet on a website. Ptolemy applets require jar files for the runtime environment, so the applet code generator will copy the necessary jar files if the value of the ptIIUserDirectory parameter is outside the $PTII directory.

1. If you built Ptolemy II from source, generate Ptolemy II jar files by running
   ```
   cd $PTII
   make install
   ```
2. Create the target directory:
   ```
   mkdir c:/tmp/ptIIapplet/Butterfly
   ```
3. Invoke copernicus:
   ```
   cd $PTII/ptolemy/domains/sdf/demo/Butterfly
   $PTII/bin/copernicus -codeGenerator applet -ptIIUserDirectory \
          c:/tmp/ptIIapplet -targetPath Butterfly Butterfly.xml
   ```
   Note that the copernicus command should be typed in on one line.

Applet code generator GUI - create applet outside the Ptolemy tree.

If you would like to generate an applet in a directory outside of the Ptolemy tree using the experimental code generation GUI, follow these steps:

1. If you built Ptolemy II from source, generate Ptolemy II jar files by running
   ```
   cd $PTII
   make install
   ```
2. Create the target directory:
   ```
   mkdir c:/tmp/ptIIapplet/Butterfly
   ```
3. Open up the SDF Butterfly Model at $PTII/ptolemy/domains/sdf/demo/Butterfly/Butterfly.xml
4. Select View -> Code Generator
5. Change the CodeGenerator combo box from java to applet
6. Change the ptIIUserDirectory to the directory where you would like the applet to be created, for example
   ```
   c:/tmp/ptapplet
   ```
   Note that the directory must already exist. If it does not exist, then the default directory will automatically be used.
7. Change the targetPath to the string
   ```
   $modelName
   ```
8. Change the modelName parameter to
   ```
   Butterfly
   ```
9. Hit the Parameters button, which will update the parameters and display their values.
10. Hit the Generate Button
11. The code generator will invoke an separate java process that generates an applet and then invokes

the browser on the generated HTML code.

8.4.2 Applet Limitations

Under Web Start, you may need to add classes to the necessaryClasses parameter so that the necessaryClassPath parameter will get updated with the appropriate jar files and passed to the subprocess that invokes the applet code generator. The reason this is necessary is because Web Start is invoked using a special class loader that accesses separate jar files in the Web Start cache. The applet code generator does not have direct access to the Web Start class loader, so we tell it what classes we need so that they can be added to the class path.

- It would be nice if the applet code generator would bundle up the necessary class files in a single jar file so that it was easier to install an applet.
- The applet code generator could use tree shaking to create a much smaller jar file that contains only the classes that are used. One issue is that the user would need to exercise the applet by invoking all the features of the GUI, such as the plot format window.
- The applet code generator should grab the top level text annotations from the MoML file and use them as comments.

References

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] G. Agha, "Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems," in *Formal Methods for Open Object-based Distributed Systems*, IFIP Transactions, E. Najm and J.-B. Stefani, Eds., Chapman & Hall, 1997.

[3] G. Agha, "Concurrent object-oriented programming," *Communications of the ACM*, 33(9):125–140, Sept. 1990.

[4] G. Agha, S. Frolund, W. Kim, R. Panwar, A. Patterson, and D. Sturman, "Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3–14, May 1993.

[5] G. Agha, I. A. Mason, S. F.Smith, and C. L. Talcott, "A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[6] R. Allen and D. Garlan, "Formalizing Architectural Connection," in *Proc. of the 16th International Conference on Software Engineering* (ICSE 94), May 1994, pp. 71-80, IEEE Computer Society Press.

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

[8] R. L. Bagrodia, "Parallel Languages for Discrete Event Simulation Models," *IEEE Computational Science & Engineering*, vol. 5, no. 2, April-June 1998, pp 27-38.

[9] R. Bagrodia, R. Meyer, *et al.*, "Parsec: A Parallel Simulation Environment for Complex Systems," *IEEE Computer*, vol. 31, no. 10, October 1998, pp 77-85.

[10] P. Baldwin, S. Kohli, E. A. Lee, X. Liu and Y. Zhao, "Modeling of Sensor Nets in Ptolemy II," In *Proceedings of Information Processing in Sensor Networks* (IPSN), Berkeley, CA, USA, April 26-27, 2004.

[11] P. Baldwin, S. Kohli, E. A. Lee, X. Liu and Y. Zhao, "Visualsense: Visual Modeling for Wireless and Sensor Network Systems," Technical Memorandum UCB/ERL M05/25, University of California, Berkeley, July 15, 2005.

[12] M. von der Beeck, "A Comparison of Statecharts Variants," in *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, LNCS 863, pp. 128-148, Springer-Verlag, 1994.

[13] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1270-1282.

[14] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Tr. on Automatic Control*, Vol. 35, No. 5, pp. 525-546, May 1990.

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

[16] S. Bhatt, R. M. Fujimoto, A. Ogielski, and K. Perumalla, “Parallel Simulation Techniques for Large-Scale Networks,” *IEEE Communications Magazine*, Vol. 36, No. 8, August 1998, pp. 42-47.

[17] S. S. Bhattacharyya, "Compiling Dataflow Programs for Digital Signal Processing," Tech. Report UCB/ERL 94/52, Ph.D. Thesis, Dept. of EECS, University of California, Berkeley, CA 94720, July 12, 1994.

[18] S. S. Bhattacharyya, P. K. Murthy and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwell, Mass, 1996.

[19] J. Bier, E. Goei, W. Ho, P. Lapsley, M. O'Reilly, G. Sih and E. A. Lee, “Gabriel: A Design Environment for DSP,” *IEEE Micro Magazine*, October 1990, vol. 10, no. 5, pp. 28-45.

[20] C. H. Brooks and E. A. Lee, “Ptolemy II Coding Style,” Technical Memorandum UCB/ERL M03/44, University of California at Berkeley, November 24, 2003. (http://ptolemy.eecs.berkeley.edu/publications/papers/03/codingstyle/)

[21] Randy Brown, “CalendarQueue: A Fast Priority Queue Implementation for The Simulation Event Set Problem”, Communications of the ACM, October 1998, Volume 31, Number 10.

[22] V. Bryant, “Metric Spaces,” Cambridge University Press, 1985.

[23] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *Int. Journal of Computer Simulation*, special issue on “Simulation Software Development,” vol. 4, pp. 155-182, April, 1994. (http://ptolemy.eecs.berkeley.edu/publications/papers/94/JEurSim)

[24] A. Burns, *Programming in OCCAM 2*, Addison-Wesley, 1988.

[25] James C. Candy, “A Use of Limit Cycle Oscillations to Obtain Robust Analog-to-Digital Converters,” *IEEE Tr. on Communications*, Vol. COM-22, No. 3, pp. 298-305, March 1974.

[26] A. Cataldo, C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer and H. Zheng, “Hyvisual: A Hybrid System Visual Modeler,” Technical Memorandum UCB/ERL M03/30, University of California, Berkeley, July 17, 2003.

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

[28] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, “LUSTRE: A Declarative Language for Programming Synchronous Systems,” *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, Munich, Germany, January, 1987.

[29] K. M. Chandy and J. Misra, “Asynchronous Distributed Simulation Via a Sequence of Parallel Computations,” *Communications of the ACM*, vol. 24, no. 11, November 1981, pp. 198-205.

[30] I. Craig, *The Interpretation of Object-Oriented Programming Languages*, Springer-Verlag, 2001.

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

[32] John Davis II, "Order and Containment in Concurrent System Design," **Ph.D. thesis**, Memorandum UCB/ERL M00/47, Electronics Research Laboratory, University of California, Berkeley, September 8, 2000.(http://ptolemy.eecs.berkeley.edu/publications/papers/00/concsys/)

[33] S. A. Edwards and E. A. Lee, “The Semantics and Execution of a Synchronous Block-Diagram Language,” *Science of Computer Programming*, Vol. 48, no. 1, July 2003.

[34] S. A. Edwards, “The Specification and Execution of Heterogeneous Synchronous Reactive Systems,” **Ph.D. thesis**, University of California, Berkeley, May 1997. Available as UCB/ERL M97/31. (http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/)

[35] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, “Taming Heterogeneity-the Ptolemy Approach,” *Proceedings of the IEEE*, V. 91, No 1, January 2003.

[36] J. Eker and J. W. Janneck, “Cal Language Report: Specification of the Cal Actor Language,” Technical Memorandum No. UCB/ERL M03/48, University of California, Berkeley, CA, December 1, 2003.

[37] P. H. J. van Eijk, C. A. Vissers, M. Diaz, *The formal description technique LOTOS*, Elsevier Science, B.V., 1989. (http://wwwtios.cs.utwente.nl/lotos)

[38] R. Esser, “An Object Oriented Petri Net Approach to Embedded System Design,” Ph.D. Thesis, ETH, Zurich, 1996.

[39] P. A. Fishwick, *Simulation Model Design and Execution: Building Digital Worlds*, Prentice Hall, 1995.

[40] C. Fong, "Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II," Master's Report, Memorandum UCB/ERL M01/9, Electronics Research Laboratory, University of California, Berkeley, January 2001.(http://ptolemy.eecs.berkeley.edu/publications/papers/00/dt/)

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

[42] R. M. Fujimoto, “Parallel Discrete Event Simulation,” *Communications of the ACM*, vol. 33, no. 10, October 1990, pp 30-53.

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

[44] C. W. Gear, “Numerical Initial Value Problems in Ordinary Differential Equations,” Prentice Hall Inc. 1971.

[45] A. J. C. van Gemund, *“Performance Prediction of Parallel Processing Systems: The PAMELA Methodology,”* Proc. 7th Int. Conf. on Supercomputing, pages 418-327, Tokyo, July 1993.

[46] A. Girault, B. Lee, and E. A. Lee, “Hierarchical Finite State Machines with Multiple Concurrency Models,” April 13, 1998 (revised from Memorandum UCB/ERL M97/57, Electronics Research Laboratory, University of California, Berkeley, CA 94720, August 1997). (http://ptolemy.eecs.berkeley.edu/publications/papers/98/starcharts)

[47] M. Goel, *Process Networks in Ptolemy II*, MS Report, ERL Technical Report UCB/ERL No. M98/69, University of California, Berkeley, CA 94720, December 16, 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/PNinPtolemyII)

[48] G. Goessler and A. Sangiovanni-Vincentelli, “Compositional Modeling in Metropolis,” In *Proceedings of Second International Workshop on Embedded Software* (EMSOFT), Grenoble, France, Springer-Verlag, October 7-9, 2002.

[49] M. Grand, *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML*, John Wiley & Sons, 1998.

[50] C. Hansen, “Hardware logic simulation by compilation,” In *Proceedings of the Design Automation Conference* (DAC). SIGDA, ACM, 1988.

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

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

[53] T. A. Henzinger, B. Horowitz and C. M. Kirsch, “Giotto: A Time-Triggered Language for Embedded Programming,” EMSOFT 2001, Tahoe City, CA, Springer-Verlag.

[54] T. A. Henzinger, “The theory of hybrid automata,” in *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1996, pp. 278-292, invited tutorial.

[55] T.A. Henzinger, and O. Kupferman, and S. Qadeer, “From *pre*historic to *post*modern symbolic model checking,” in *CAV 98: Computer-aided Verification*, pp. 195-206, eds. A.J. Hu and M.Y. Vardi, Lecture Notes in Computer Science 1427, Springer-Verlag, 1998.

[56] T. A. Henzinger and C. M. Kirsch, “The Embedded Machine: Predictable, portable real-time code,” In *Proceedings of Conference on Programming Language Design and Implementation* (PLDI). SIGPLAN, ACM, June 2002.

[57] C. Hewitt, “Viewing control structures as patterns of passing messages,” *Journal of Artificial Intelligence*, 8(3):323–363, June 1977.

[58] M. G. Hinchey and S. A. Jarvis, *Concurrent Systems: Formal Developments in CSP*, McGraw-Hill, 1995.

[59] C. W. Ho, A. E. Ruehli, and P. A. Brennan, “The Modified Nodal Approach to Network Analysis,” IEEE Tran. on Circuits and Systems, Vol. CAS-22, No. 6, 1975, pp. 504-509.

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

[61] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[62] Jens Horstmannshoff and Heinrich Meyr, "Efficient Building Block Based RTL Code Generation from Synchronous Data Flow Graphs," Proceedings of the 37th conference on Design automation, Los Angeles, California, United States, pp. 552 - 555, 2000.

[63] IEEE DASC 1076.1 Working Group, “VHDL-A Design Objective Document, version 2.3,” http://www.vhdl.org/analog/ftp_files/requirements/DOD_v2.3.txt

[64] D. Jefferson, Brian Beckman, et al, “Distributed Simulation and the Time Warp Operating System,” UCLA Computer Science Department: 870042, 1987.

[65] Neil D. Jones, Carsten K. Gomard and Peter Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, June 1993.

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

[67] G. Kahn and D. B. MacQueen, “Coroutines and Networks of Parallel Processes,” *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[68] G. Karsai, M. Maroti, Á. Lédeczi, J. Gray and J. Sztipanovits, “Type Hierarchies and Composition in Modeling and Meta-Modeling Languages,” *IEEE Transactions on Control System Technology*, Vol. 12, No. 2, March 2004.

[69] G. Karsai, “A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming,” *IEEE Computer*: 36-44, March 1995.

[70] E. Kohler, *The Click Modular Router*, Ph.D. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 2001.

[71] E. Kohler, R. Morris R and B. Chen, "Programming language optimizations for modular router configurations," ACM. SIGPLAN Notices, vol.37, no.10, Oct. 2002, pp. 251-63.

[72] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.

[73] P. Laramie, R.S. Stevens, and M.Wan, “Kahn process networks in Java,” ee290n class project report, Univ. of California at Berkeley, 1996.

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

[75] B. Lee and E. A. Lee, “Interaction of Finite State Machines with Concurrency Models,” *Proc. of Thirty Second Annual Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/Interaction-FSM/)

[76] B. Lee and E. A. Lee, “Hierarchical Concurrent Finite State Machines in Ptolemy,” *Proc. of International Conference on Application of Concurrency to System Design*, p. 34-40, Fukushima, Japan, March 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/HCFSMinPtolemy/)

[77] E. A. Lee and S. Neuendorffer, “Classes and Subclasses in Actor-Oriented Design,” In Proceedings of Conference on Formal Methods and Models for Codesign (MEMOCODE), San Diego, CA, USA, June 22-25, 2004.

[78] E. A. Lee and Y. Xiong, “A Behavioral Type System and Its Application in Ptolemy II,” *Formal Aspects of Computing Journal*, special issue on Semantic Foundations of Engineering Design Languages, Volume 16, Number 3, August 2004.

[79] E. A. Lee, S. Neuendorffer and M. J. Wirthlin, “Actor-Oriented Design of Embedded Hardware and Software Systems,” *Journal of Circuits, Systems, and Computers*, 12(3): 231-260, 2003, 2003.

[80] E. A. Lee, “Embedded Software,” in *Advances in Computers* (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

[81] E. A. Lee and T. M. Parks, “Dataflow Process Networks,” in *Readings in Hardware/Software Co-Design*, G. De Micheli, R. Ernst, and W. Wolf, eds., Morgan Kaufmann, San Francisco, 2002 (reprinted from 86).

[82] E. A. Lee, “What's Ahead for Embedded Software?” *IEEE Computer*, September 2000, pp. 18-26.

[83] E. A. Lee, “Modeling Concurrent Real-time Processes Using Discrete Events,” Invited paper to *Annals of Software Engineering*, Special Volume on Real-Time Software Engineering, Volume 7, 1999, pp 25-45. Also UCB/ERL Memorandum M98/7, March 4th 1998.(http://ptolemy.eecs.berkeley.edu/publications/papers/98/realtime)

[84] E. A. Lee and Y. Xiong, “System-Level Types for Component-Based Design,” *First Workshop on Embedded Software*, EMSOFT 2001, Lake Tahoe, CA, USA, Oct. 8-10, 2001. (also Technical Memorandum UCB/ERL M00/8, Electronics Research Lab, University of California, Berkeley, CA 94720, USA, February 29, 2000. http://ptolemy.eecs.berkeley.edu/publications/papers/01/systemLevelType/).

[85] E. A. Lee, “Computing for Embedded Systems,” invited paper, *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 21-23, 2001.

[86] E. A. Lee and T. M. Parks, “Dataflow Process Networks,”, *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May, 1995. (http://ptolemy.eecs.berkeley.edu/publications/papers/95/processNets)

[87] E. A. Lee and A. Sangiovanni-Vincentelli, “A Framework for Comparing Models of Computation,”, *IEEE Transactions on CAD*, Vol 17, No. 12, December 1998 (Revised from ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997). (http://ptolemy.eecs.berkeley.edu/publications/papers/97/denotational/)

[88] E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing,” *IEEE Trans. on Computers*, January, 1987.

[89] M. A. Lemkin, *Micro Accelerometer Design with Digital Feedback Control*, Ph.D. dissertation, University of California, Berkeley, Fall 1997.

[90] S. Y. Liao, S. Tjiang, and R. Gupta, “An efficient implementation of reactivity for modeling hardware in the Scenic design environment,” In *Proceedings of the 34th Design Automation Conference* (DAC’1997). SIGDA, ACM, 1997.

[91] J. Liu, J. Eker, J. W. Janneck and E. A. Lee, “Realistic Simulations of Embedded Control Systems,” *International Federation of Automatic Control, 15th IFAC World Congress*, Barcelona, Spain, July 21-26, 2002.

[92] J. Liu, X. Liu, and E. A. Lee, “Modeling Distributed Hybrid Systems in Ptolemy II,” invited embedded tutorial in *American Control Conference*, Arlington, VA, June 25-27, 2001.

[93] J. Liu, S. Jefferson, and E. A. Lee, “Motivating Hierarchical Run-Time Models in Measurement and Control Systems,” *American Control Conference*, Arlington, VA, pp. 3457-3462, June 25-27, 2001.

[94] J. Liu and E. A. Lee, “A Component-Based Approach to Modeling and Simulating Mixed-Signal and Hybrid Systems,” *ACM Trans. on Modeling and Computer Simulation*, special issue on computer automated multi-paradigm modeling, Volume 12, Issue 4, pp. 343-368, October 2002.

[95] J. Liu and E. A. Lee, “On the Causality of Mixed-Signal and Hybrid Models,” *6th International Workshop on Hybrid Systems: Computation and Control* (HSCC '03), April 3-5, Prague, Czech Republic, 2003.

[96] J. Liu and E. A. Lee, “Timed Multitasking for Real-Time Embedded Software,” *IEEE Control Systems Magazine*: 65-75, February, 2003.

[97] J. Liu, "Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems," **Ph.D. thesis**, Technical Memorandum UCB/ERL M01/41, University of California, Berkeley, CA 94720, December 20th, 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/responsibleFrameworks/)

[98] J. Liu, *Continuous Time and Mixed-Signal Simulation in Ptolemy II*, MS Report, UCB/ERL Memorandum M98/74, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/MixedSignalinPtII/)

[99] J. Liu and E. A. Lee, "Component-based Hierarchical Modeling of Systems with Continuous and Discrete Dynamics," *Proc. of the 2000 IEEE International Conference on Control Applications and IEEE Symposium on Computer-Aided Control System Design* (CCA/CACSD'00), Anchorage, AK, September 25-27, 2000. pp. 95-100.

[100]J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S. Sastry, and E. A. Lee, "A Hierarchical Hybrid System and Its Simulation", 1999 38th IEEE Conference on Decision and Control (CDC'99), Phoenix, Arizona.

[101]X. Liu, J. Liu, J. Eker, and E. A. Lee, “Heterogeneous Modeling and Design of Control Systems,” in *Software-Enabled Control: Information Technology for Dynamical Systems*, T. Samad and G. Balas (eds.), New York City: IEEE Press, 2003.

[102]D. C. Luckham and J. Vera, “An Event-Based Architecture Definition Language,” *IEEE Transactions on Software Engineering*, 21(9), pp. 717-734, September, 1995.

[103]F. Maraninchi, “The Argos Language: Graphical Representation of Automata and Description of Reactive Systems,” in *Proc. of the IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.

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

[105]K. Mehlhorn and Stefan Naher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1997.

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

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

[108]R. Milner, *“A Calculus of Communicating Systems”*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

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

[110]J. Misra, “Distributed Discrete-Event Simulation,” *Computing Surveys*, vol. 18, no. 1, March 1986, pp. 39-65.

[111]L. Muliadi, "Discrete Event Modeling in Ptolemy II," MS Report, Dept. of EECS, University of California, Berkeley, CA 94720, May 1999. (http://ptolemy.eecs.berkeley.edu/publications/papers/99/deModeling/)

[112]Praveen K. Murthy, "Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow," Technical Memorandum UCB/ERL M96/79, Ph.D. Thesis, EECS Department, University of California, Berkeley, CA 94720, December 1996.

[113]P. Murthy, S. S. Bhattacharyya and E. A. Lee, “Joint Minimization of Code and Data for Synchronous Dataflow Programs,” Journal of Formal Methods in System Design, vol. 11, No. 1, July 1997.

[114]P. K. Murthy and E. A. Lee, “Multidimensional Synchronous Dataflow,” *IEEE Transactions on Signal Processing*, volume 50, no. 8, pp. 2064 -2079, August 2002.

[115]L. W. Nagal, “SPICE2: A Computer Program to Simulate Semiconductor Circuits,” ERL Memo No. ERL-M520, Electronics Research Laboratory, University of California, Berkeley, CA 94720.

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

[117]S. Neuendorffer, “Automatic Specialization of Actor-Oriented Models in Ptolemy II,” Master's Report, Technical Memorandum UCB/ERL M02/41, University of California, Berkeley, CA 94720, December 25, 2002. (http://ptolemy.eecs.berkeley.edu/papers/02/actorSpecialization)

[118]S. Neuendorffer, “Actor-Oriented Metaprogramming,” Ph.D. Thesis, University of California, Berkeley, December 21, 2004.(http://ptolemy.eecs.berkeley.edu/publications/papers/04/Steves-Thesis)

[119]A. R. Newton and A. L. Sangiovanni-Vincentelli, “Relaxation-Based Electrical Simulation,” *IEEE Tr. on Electronic Devices*, Vol. ed-30, No. 9, Sept. 1983.

[120]S. Oaks and H. Wong, *Java Threads*, O’Reilly, 1997.

[121]OMG, *Unified Modeling Language: Superstructure*, version 2.0, 3rd revised submission to RFP ad/00-09-02, April 10, 2003.

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

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

[124]T. M. Parks, *Bounded Scheduling of Process Networks*, Technical Report UCB/ERL-95-105. **Ph.D. Dissertation**. EECS Department, University of California. Berkeley, CA 94720, December 1995. (http://ptolemy.eecs.berkeley.edu/publications/papers/95/parksThesis/)

[125]J. K. Peacock, J. W. Wong and E. G. Manning, “Distributed Simulation Using a Network of Processors,” *Computer Networks*, vol. 3, no. 1, February 1979, pp. 44-56.

[126]José L. Pino, "Software Synthesis for Single-Processor DSP Systems Using Ptolemy," Master's Report, UCB/ERL M93/35, Dept. of EECS, University of California, Berkeley, CA 94720, May 1993.

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

[128]J. Reekie, S. Neuendorffer, C. Hylands and E. A. Lee, “Software Practice in the Ptolemy Project,” Technical Report Series, GSRC-TR-1999-01, Gigascale Silicon Research Center, University of California, Berkeley, CA 94720, April 1999. (http://ptolemy.eecs.berkeley.edu/publications/papers/99/sftwareprac/)

[129]J. Rehof and T. Mogensen, “Tractable Constraints in Finite Semilattices,” *Third International Static Analysis Symposium*, pp. 285-301, Volume 1145 of Lecture Notes in Computer Science, Springer, Sept., 1996.

[130]J. H. Reppy, “CML: A Higher-Order Concurrent Language,” SIGPLAN Notices, 26(6): 293-305, June, 1991.

[131]C. Rettig, “Automatic Units Tracking,” *Embedded System Programming*, March, 2001.

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

[133]S. Ritz, S. Pankert, and H. Meyr, “Optimum Vectorization of Scalable Synchronous Dataflow Graphs”, Technical Report IS2/DSP93.1a, Aachen University of Technology, Germany, January,1993.

[134]R. C. Rosenberg and D.C. Karnopp, *Introduction to Physical System Dynamics*, McGraw-Hill, NY, 1983.

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

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

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

[138]S. Saracco, J. R. W. Smith, and R. Reed, *Telecommunications Systems Engineering Using SDL*, North-Holland - Elsevier, 1989.

[139]B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, NY 1994.

[140]N. Smyth, *Communicating Sequential Processes Domain in Ptolemy II*, MS Report, UCB/ERL Memorandum M98/70, Dept. of EECS, University of California, Berkeley, CA 94720, December 1998. (http://ptolemy.eecs.berkeley.edu/publications/papers/98/CSPinPtolemyII/)

[141]I. E. Sutherland, “Sketchpad - a Man-Machine Graphical Communication System,” Technical Report 296, MIT Lincoln Laboratory, January, 1963.

[142]W. R. Sutherland, "The on-Line Graphical Specification of Computer Procedures," Ph.D. Thesis, MIT, Cambridge, MA, 1966.

[143]J. Teich, E. Zitzler, and S. Bhattacharyya, “3D exploration of software schedules for DSP algorithms,” In *Proceedings of International Symposium on Hardware/Software Codesign* (CODES). SIGDA, ACM, May 1999.

[144]J. Tsay, “A Code Generation Framework for Ptolemy II,” ERL Technical Report UCB/ERL No. M00/25, Dept. EECS, University of California, Berkeley, CA 94720, May 19, 2000. (http://ptolemy.eecs.berkeley.edu/publications/papers/00/codegen)

[145]J. Tsay, C. Hylands and E. A. Lee, "A Code Generation Framework for Java Component-Based Designs," *CASES '00*, November 17-19, 2000, San Jose, CA.

[146]P. Whitaker, "The Simulation of Synchronous Reactive Systems In Ptolemy II," Master's Report, Memorandum UCB/ERL M01/20, Electronics Research Laboratory, University of California, Berkeley, May 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/sr/)

[147]World Wide Web Consortium, *XML 1.0 Recommendation*, October 2000, http://www.w3.org/XML/

[148]World Wide Web Consortium, *Overview of SGML Resources*, August 2000, http://www.w3.org/MarkUp/SGML/

[149]Y. Xiong and E. A. Lee, “An Extensible Type System for Component-Based Design,” *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. LNCS 1785.

[150]Y. Xiong, “An Extensible Type System for Component-Based Design,” **Ph.D. thesis**, Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1, 2002. (http://ptolemy.eecs.berkeley.edu/papers/02/typeSystem).

[151]Y. Zhao, “A Model of Computation with Push and Pull Processing,” Masters Thesis, Technical Memorandum No. UCB/ERL M03/51, University of California, Berkeley, December 16, 2003.

Index

D

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z