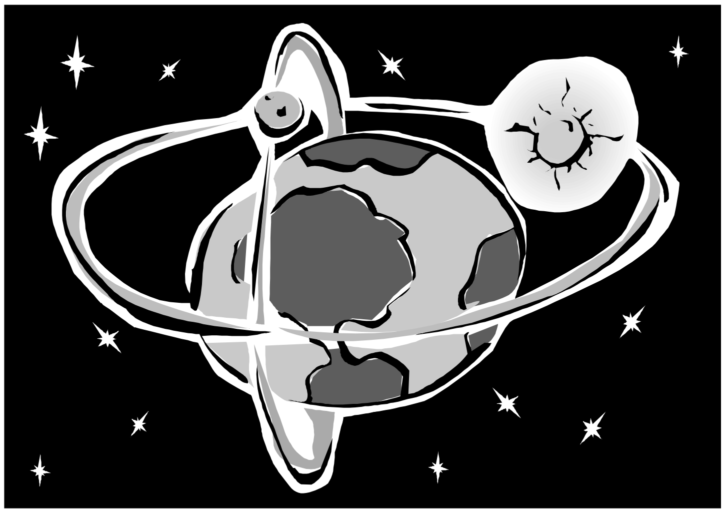
-PTOLEMY II -



TOOL INTEGRATION INFORMATION

John Davis, II
Christopher Hylands
Bart Kienhuis
Edward A. Lee
Jie Liu
Xiaojun Liu
Lukito Muliadi
Steve Neuendorffer
Jeff Tsay
Brian Vogel
Yuhong Xiong

Previous Authors:
Mudit Goel
Lukito Muliadi
John Reekie
Neil Smyth

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

Based on Memorandum UCB/ERL M01/12
Originally published as Memorandum UCB/ERL M99/40
Document Version 1.0
for use with Ptolemy II 1.0
September 24, 2001

Mobies Project
DARPA ITO

Principal Contact: Edward A. Lee, eal@eecs.berkeley.edu

1

MoML

Authors: *Edward A. Lee*
Steve Neuendorffer

1.1 Introduction

Ptolemy II models might be *simulations* (executable models of some other system) or *implementations* (the system itself). They might be classical computer programs (applications), or any of a number of network-integrated programs (applets, servlets, or CORBA services, for example).

Models can be specified in a number of ways. You can write Java code that instantiates components, parameterizes them, and interconnects them. Or you can use Vergil (see the Vergil chapter above) to graphically construct models. Vergil stores models in ASCII files using an XML schema called MoML. MoML (which stands for Modeling Markup Language) is the primary persistent file format for Ptolemy II models. It is also the primary mechanism for constructing models whose definition and execution is distributed over the network.

This chapter explains MoML. Most users will not need to edit MoML files directly. Use Vergil instead. Occasionally, however, it is useful to examine and/or edit MoML files directly.

MoML is a modeling markup schema in the Extensible Markup Language (XML). It is intended for specifying interconnections of parameterized components. A MoML file can be executed as an application using any of the following commands,

```
ptolemy filename.xml
ptexecute filename.xml
vergil filename.xml
moml configuration.xml filename.xml
```

These commands are defined in the directory `$PTII/bin`, which must be in your path[1], where `$PTII` is the location of the Ptolemy II installation. In all cases, the filename can be replaced by a URL. The `ptolemy` command assumes that the file defines an executable Ptolemy II model, and opens a control panel to execute it. The `ptexecute` command executes it without a control panel. The `vergil` command opens a graphical editor to edit and execute the model. The `moml` command uses the specified configuration file (a MoML file containing a Ptolemy II configuration) to invoke some set of customized views or editors on the model. The filename extension can be ".xml" or ".moml" for MoML files. And the same XML file can be used in an applet[2].

To get a quick start, try entering the following into a file called `test.xml` (This file is also available as $PTII/ptolemy/moml/demo/test.xml):

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="test" class="ptolemy.actor.TypedCompositeActor">
  <property name="director"
            class="ptolemy.domains.sdf.kernel.SDFDirector"/>
  <entity name="ramp" class="ptolemy.actor.lib.Ramp"/>
  <entity name="plot" class="ptolemy.actor.gui.SequencePlotter"/>
  <relation name="r" class="ptolemy.actor.TypedIORelation"/>
  <link port="ramp.output" relation="r"/>
  <link port="plot.input" relation="r"/>
</entity>
```

This code defines a model in a top-level entity called "test". By convention, we use the same name for the top-level model and the file in which it resides. The top-level model is an instance of the Ptolemy II class `ptolemy.actor.TypedCompositeActor`. It contains a director, two entities, a relation, and two links. The model is depicted in figure 1.1, where the director is not shown. It can be run using the command

```
ptolemy test.xml
```

You should get a window looking like that in figure .1.2. Enter "10" in the iterations box and hit the "Go" button to execute the model for 10 iterations (leaving the default "0" in the iterations box executes it forever, until you hit the "Stop" button).

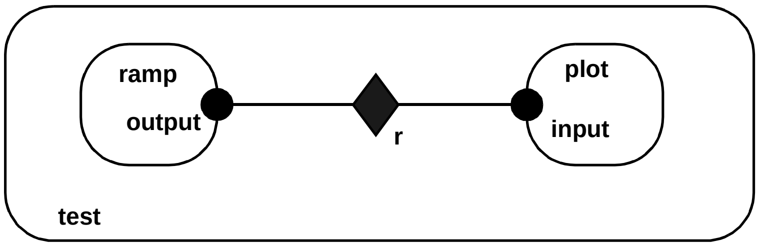


FIGURE 1.1. Simple example in the file $PTII/ptolemy/moml/demo/test.xml.

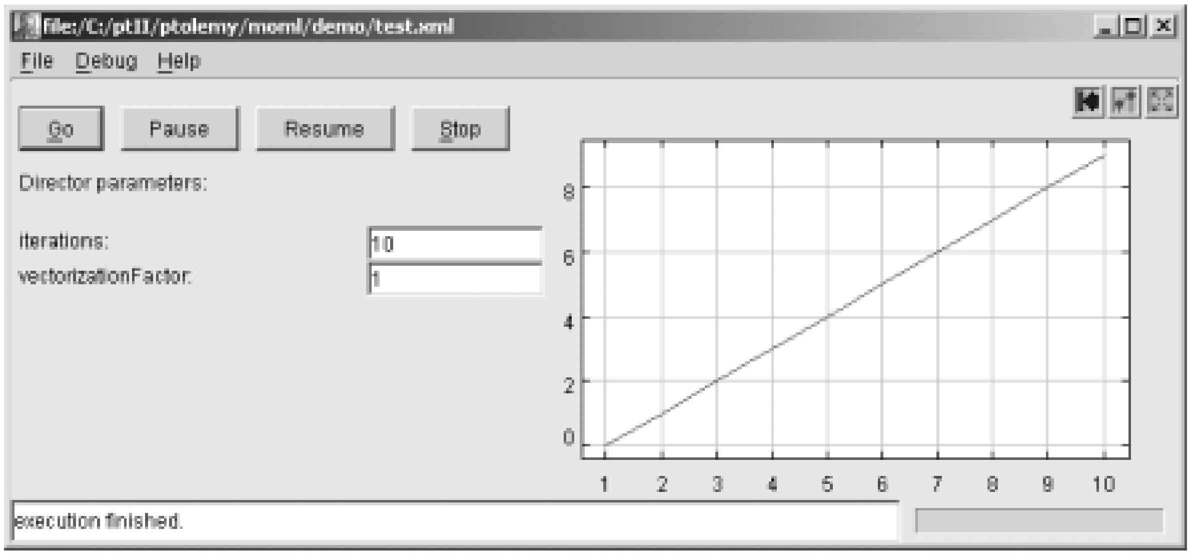
1. These commands are executed this way on Unix systems and on Windows systems with Cygwin installed. On other configurations, the equivalent commands are invoked in some other way.
2. An *applet* is a Java program that is downloaded from a web server by a browser and executed in the client's computer (usually within a plug-in for the browser).

The structure of the above MoML text is explained in detail in this chapter. A more interesting example is given in the appendix to this chapter. You may wish to refer to that example as you read about the details. The next chapter explains how to bypass MoML and write applets directly. The chapter after that describes the actors libraries that are included in the current Ptolemy II version.

1.2 MoML Principles

The key features of MoML include:

- *Web integration*. MoML is an XML schema. XML, the popular *extensible markup language*[89], provides a standard syntax and a standard way of defining the content within that syntax. The syntax is a subset of SGML[90], and is similar to HTML. It is intended for use on the Internet, and is intended for precisely this sort of specialization into schemas. File references are via URIs (in practice, URLs), both relative and absolute, so MoML is equally comfortable working in applets and applications.
- *Implementation independence*. MoML is designed to work with a variety of tools. A modeling tool that reads MoML files is expected to provide a class loader in some form. Given the name of a class, and possibly a URL for the class definition, the class loader must be able to instantiate it. Classes might be defined in MoML or in some base language such as Java. In Java, the class loader could be that built in to the JVM. In C++ or other languages, the class loader would have to be implemented by the modeling tool. Ptolemy II can be viewed as a reference implementation of a MoML tool that uses Java as its base language.
- *Extensibility.* Components can be parameterized in two ways. First, they can have named properties with string values. Second, they can be associated with an external configuration file that can be in any format understood by the component. Typically, the configuration will be in some other XML schema, such as PlotML or SVG (scalable vector graphics).
- *Classes and inheritance*. Components can be defined in MoML as classes which can then be instantiated in a model. Components can extend other components through an object-oriented inheritance mechanism.



Director parameters:

iterations: 10
vectorizationFactor: 1

execution finished.

FIGURE 1.2. Simple example of a Ptolemy II model execution control window.

- *Semantics independence*. MoML defines no semantics for an interconnection of components. It represents only the hierarchical containment relationships between entities with properties, their ports, and the connections between their ports. In Ptolemy II, the meaning of a connection (the semantics of the model) is defined by the director for the model, which is a property of the top-level entity. The director defines the semantics of the interconnection. MoML knows nothing about directors except that they are instances of classes that can be loaded by the class loader and assigned as properties.

1.2.1 Clustered Graphs

A model is given as a clustered graph, which is an *abstract syntax* for representing netlists, state transition diagrams, block diagrams, etc. An abstract syntax is a conceptual data organization. A particular clustered graph configuration is called a *topology*. A topology is a collection of *entities* and *relations*. Furthermore, entities have *ports* and relations connect the ports. We consistently use the term *connection* to denote the association between connected ports (or their entities), and the term *link* to denote the association between ports and relations. Thus, a connection consists of a relation and two or more links.

The concept of an abstract syntax can be contrasted with a *concrete syntax*, which is a persistent, readable representation of the data. For example, EDIF is a concrete syntax for representing netlists. MoML is a concrete syntax for the clustered graph abstract syntax. Furthermore, we use the visual notation shown in figure 1.3, where entities are depicted as rounded boxes, relations as diamonds, and entities as filled circles.

The use of ports and hierarchy distinguishes our topologies from mathematical graphs. In a mathematical graph, an entity would be a vertex, and an arc would be a connection between entities. A vertex could be represented in our schema using entities that always contain exactly one port. In a directed graph, the connections are divided into two subsets, one consisting of incoming arcs, and the other of outgoing arcs. The vertices in such a graph could be represented by entities that contain two ports, one for incoming arcs and one for outgoing arcs. Thus, in mathematical graphs, entities always have one or two ports, depending on whether the graph is directed. Our schema generalizes this by permitting an entity to have any number of ports, thus dividing its connections into an arbitrary number of subsets.

A second difference between our graphs and mathematical graphs is that our relations are multi-way associations, whereas an arc in a graph is a two-way association. A third difference is that mathe-

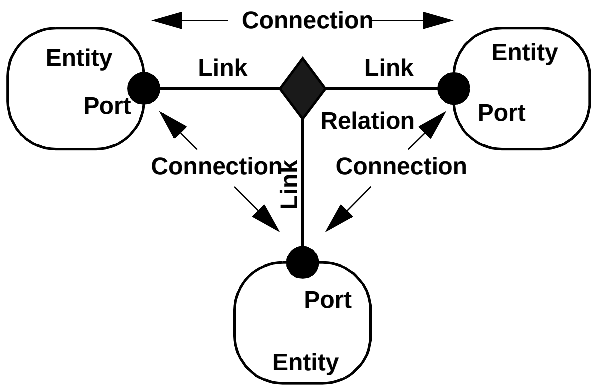


FIGURE 1.3. Visual notation and terminology.

matical graphs normally have no notion of hierarchy (clustering).

Relations are intended to serve a mediators, in the sense of the Mediator design pattern[28]. “Mediator promotes loose coupling by keeping objects from referring to each other explicitly...” For example, a relation could be used to direct messages passed between entities. Or it could denote a transition between states in a finite state machine, where the states are represented as entities. Or it could mediate rendezvous between processes represented as entities. Or it could mediate method calls between loosely associated objects, as for example in remote method invocation over a network.

1.2.2 Abstraction

Composite entities (clusters) are entities that can contain a topology (entities and relations). Clustering is illustrated by the example in figure 1.4. A port contained by a composite entity has inside as well as outside links. 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. Ports within an entity are hidden by default, and must be explicitly exposed to be visible (linkable) from outside the entity[3]. The composite entity with ports thus provides an abstraction of the contents of the composite.

1.3 Specification of a Model

In this section, we describe the XML elements that are used to define MoML models.

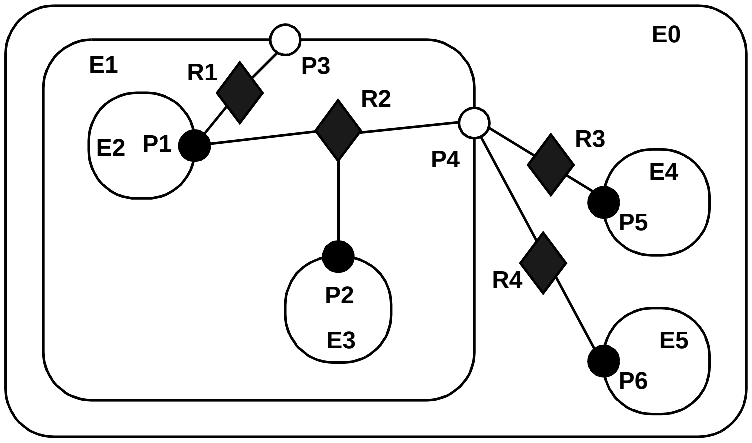


FIGURE 1.4. Ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

3. Unless level-crossing links are allowed. MoML supports these, but they are discouraged.

1.3.1 Data Organization

As with all XML files, MoML files have two parts, one defining the MoML language and one containing the model data. The first part is called the *document type definition*, or DTD. This dual specification of content and structure is a key XML innovation. The DTD for MoML is given in figure 1.5. If you are adept at reading these, it is a complete specification of the schema. However, since it is not particularly easy to read, we explain its key features here.

Every MoML 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 entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="modelname" class="classname">
  model definition ...
</entity>
```

Here, “*`model definition`*” is a set of XML elements that specify a clustered graph. The syntax for these elements is described in subsequent sections. The first line above is required in 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 (model) 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 MoML 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, is the name of the “owner” of the DTD, “`UC Berkeley`.” The next field is the name of the DTD, “`DTD MoML 1`” where the “`1`” indicates version 1 of the MoML DTD. The final field, “`EN`” indicates that the language assumed by the DTD is English. The Ptolemy II MoML parser requires that the public DTD be given exactly as shown, or it will not recognize the file as MoML.

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 MoML tool does not have access to a local copy of the DTD, then it finds it at this web site.

The “entity” element may be replaced by a “class” element, as in:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE class PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<class name="modelname" class="classname">
  class definition ...
</class>
```

We will say more about class definitions below.

```
<!ELEMENT class (class | configure | deleteEntity | deletePort | deleteRelation | director |
  doc | entity | group | import | input | link | port | property | relation | rename |
  rendition | unlink)*>
<!ATTLIST class name CDATA #REQUIRED
              extends CDATA #IMPLIED
              source CDATA #IMPLIED>

<!ELEMENT configure (#PCDATA)>
<!ATTLIST configure source CDATA #IMPLIED>

<!ELEMENT deleteEntity EMPTY>
<!ATTLIST deleteEntity name CDATA #REQUIRED>

<!ELEMENT deletePort EMPTY>
<!ATTLIST deletePort name CDATA #REQUIRED>

<!ELEMENT deleteProperty EMPTY>
<!ATTLIST deleteProperty name CDATA #REQUIRED>

<!ELEMENT deleteRelation EMPTY>
<!ATTLIST deleteRelation name CDATA #REQUIRED>

<!ELEMENT doc (#PCDATA)>
<!ATTLIST doc name CDATA #IMPLIED>

<!ELEMENT entity (class | configure | deleteEntity | deletePort | deleteRelation | director |
  doc | entity | group | import | input | link | port | property | relation | rename |
  rendition | unlink)*>
<!ATTLIST entity name CDATA #REQUIRED
               class CDATA #IMPLIED
               source CDATA #IMPLIED>

<!ELEMENT group ANY>
<!ATTLIST group name CDATA #IMPLIED>

<!ELEMENT input EMPTY>
<!ATTLIST input source CDATA #REQUIRED>

<!ELEMENT link EMPTY>
<!ATTLIST link insertAt CDATA #IMPLIED
               port CDATA #REQUIRED
               relation CDATA #REQUIRED
               vertex CDATA #IMPLIED>
<!ELEMENT port (configure | doc | property | rename)*>
<!ATTLIST port class CDATA #IMPLIED
               name CDATA #REQUIRED>
<!ELEMENT property (configure | doc | property | rename)*>
<!ATTLIST property class CDATA #IMPLIED
                    name CDATA #REQUIRED
                    value CDATA #IMPLIED>
<!ELEMENT relation (configure | doc | property | rename | vertex)*>
<!ATTLIST relation name CDATA #REQUIRED
                   class CDATA #IMPLIED>
<!ELEMENT rename EMPTY>
<!ATTLIST rename name CDATA #REQUIRED>
<!ELEMENT unlink EMPTY>
<!ATTLIST unlink index CDATA #IMPLIED
               insideIndex CDATA #IMPLIED
               insideIndex CDATA #IMPLIED
               port CDATA #REQUIRED
               relation CDATA #REQUIRED>
<!ELEMENT vertex (configure | doc | location | property | rename)*>
<!ATTLIST vertex name CDATA #REQUIRED
                 pathTo CDATA #IMPLIED
                 value CDATA #IMPLIED>
```

FIGURE 1.5. MoML version 1.2 DTD.

1.3.2 Overview of XML

An XML document consists of the header tags "`<?xml ... ?>`" and "`<!DOCTYPE ... >`" followed by exactly one *element*. The element has the structure:

```
start tag
body
end tag
```

where the start tag has the form

```
<elementName attributes>
```

and the end tag has the form

```
</elementName>
```

The body, if present, can contain additional elements as well as arbitrary text. If the body is not present, then the element is said to be *empty*; it can optionally be written using the shorthand:

```
<elementName attributes/>
```

where the body and end tag are omitted.

The attributes are given as follows:

```
<elementName attributeName="attributeValue" .../>
```

Which attributes are legal in an element is defined by the DTD. The quotation marks delimit the value of the attributes, so if the attribute value needs to contain quotation marks, then they must be given using the special XML entity "`&quot;`" as in the following example:

```
<elementName attributeName="&quot;foo&quot;"/>
```

The value of the attribute will be

```
"foo"
```

(with the quotation marks).

In XML "`&quot;`" is called an *entity*, creating possible confusion with our use of entity in Ptolemy II. In XML, an entity is a named storage unit of data. Thus, "`&quot;`" references an entity called "`quot`" that stores a double quote character.

1.3.3 Names and Classes

Most MoML elements have *name* and *class* attributes. The name is a handle for the object being defined or referenced by the element. In MoML, the same syntax is used to reference a pre-existing object as to create a new object. If a new object is being created, then the class attribute (usually) must be given. If a pre-existing object is being referenced, or if the MoML reader has a built-in default class for the element, then the class attribute is optional. If the class attribute is given, then the pre-existing object must be an instance of the specified class.

A name is either absolute or relative. Absolute names begin with a period "." and consist of a series of name fields separated by periods, as in ".x.y.z". Each name field can have alphanumeric characters, spaces, or the underscore "_" character. The first field is the name of the top-level model or class object. The second field is the name of an object immediately contained by that top-level.

Any name that does not begin with a period is relative to the current context, the object defined or referenced by an enclosing element. The first field of such a name refers to or defines an object immediately contained by that object. For example, inside of an object with absolute name ".x" the name "y.z" refers to an object with absolute name ".x.y.z".

A name is required to be unique within its container. That is, in any given model, the absolute names of all the objects must be unique. There can be two objects named "z", but they must not be both contained by ".x.y".

Not much more will be said about classes. Particular implementations of MoML can use this field as necessary to specify different variations of the basic syntactic objects. The class names that are used in the Ptolemy II implementation of MoML are always fully qualified Java class names. In addition, in Ptolemy II a MoML file can be referenced as a class in the same way

1.3.4 Model Element

A very simple MoML file looks like this:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="modelname" class="classname">
</entity>
```

A *model* element has name and class attributes. This value of the class attribute must be a class that instantiable by the MoML tool. For example, in Ptolemy II, we can define a model with:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="ptIImodel" class="ptolemy.actor.TypedCompositeActor">
</entity>
```

Here, `ptolemy.actor.TypedCompositeActor` is a class that a Java class loader can find and that the MoML parser can instantiate. In Ptolemy II, it is a container class for clustered graphs representing executable models or libraries of instantiable model classes. A model can be an instance of `ptolemy.kernel.util.NamedObj` or any derived class, although most useful models will be instances of `ptolemy.kernel.CompositeEntity` or a derived class. `TypedCompositeActor`, as in the above example, is derived from `CompositeEntity`.

1.3.5 Entity Element

A model typically contains entities, as in the following Ptolemy II example:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="ptIImodel" class="ptolemy.actor.TypedCompositeActor">
  <entity name="source" class="ptolemy.actor.lib.Ramp"/>
  <entity name="sink" class="ptolemy.actor.lib.SequencePlotter"/>
</entity>
```

Notice the common XML shorthand here of writing "`<entity ... />`" rather than "`<entity ...></entity>.`" Of course, the shorthand only works if there is nothing in the body of the entity element.

An entity can contain other entities, as shown in this example:

```
<entity name="ptIImodel" class="ptolemy.actor.TypedCompositeActor">
  <entity name="container" class="ptolemy.actor.TypedCompositeActor">
    <entity name="source" class="ptolemy.actor.lib.Ramp"/>
  </entity>
</entity>
```

An entity must specify a class unless the entity already exists in the containing entity or model. The name of the entity reflects the container hierarchy. Thus, in the above example, the *source* entity has the full name "`.ptIImodel.container.source`".

The definition of an entity can be distributed in the MoML file. Once created, it can be referred to again by name as follows:

```
<entity name="top" class="classname">
  <entity name="x" class="classname"/>
  ...
  <entity name="x">
    <property name="y">
  </entity>
</entity>
```

The property element (see section 1.3.6 below) is added to the pre-existing entity with name "x" when the second entity element is encountered.

In principle, MoML supports multiple containment, as in the following:

```
<entity name="top" class="classname">
  <entity name="x" class="classname"/>
  ...
  <entity name="y" class="classname">
    <entity name=".top.x"/>
  </entity>
</entity>
```

Here, the element named "x" appears both in "top" and in ".top.y", i.e. the same instance appears in two different places. Thus, it would have two full names, ".top.x" and ".top.y.x". However, Ptolemy II does not support this, as it implements a strict container relationship, where an object can have only one container. Thus, attempting to parse the above MoML will result in an exception being thrown.

1.3.6 Properties

Entities (and some other elements) can be parameterized. There are two mechanisms. The simplest one is to use the *property* element:

```
<entity name="source" class="ptolemy.actor.lib.Ramp">
  <property name="init"
            value="5"
            class="ptolemy.data.expr.Parameter"/>
</entity>
```

The property element has a name, at minimum (the value and class are optional). It is common for the enclosing class to already contain properties, in which case the property element is used only to set the value. For example:

```
<entity name="source" class="ptolemy.actor.lib.Ramp">
  <property name="init" value="5"/>
</entity>
```

In the above, the enclosing object (*source*, an instance of `ptolemy.actor.lib.Ramp`) must already contain a property with the name *init*. This is typically how library components are parameterized. In Ptolemy II, the value of a property may be an expression, as in “`PI/50`”. The expression may refer to other properties of the containing entity or of its container. Note that the expression language is not part of MoML, but is rather part of Ptolemy II. In MoML, a property value is simply an uninterpreted string. It is up to a MoML tool, such as Ptolemy II, to interpret that string.

A property can be declared without a class and without a pre-existing property if it is a *pure property*, one with only a name and no value. For example:

```
<entity name="source" class="ptolemy.actor.lib.Ramp">
  <property name="abc"/>
</entity>
```

A property can also contain a property, as in

```
<property name="x" value="5">
  <property name="y" value="10"/>
</property>
```

A second, much more flexible mechanism is provided for parameterizing entities. The *configure* element can be used to specify a relative or absolute URL pointing to a file that configures the entity, or it can be used to include the configuration information in line. That information need not be MoML information. It need not even be XML, and can even be binary encoded data (although binary data cannot be in line; it must be in an external file). For example,

```
<entity name="sink" class="ptolemy.actor.lib.SequencePlotter">
  <configure source="url"/>
```

```
        </entity>
```

Here, *url* can give the name of a file containing data, or a URL for a remote file. (For the SequencePlotter actor, that external data will have PlotML syntax; PlotML is another XML schema for configuring plotters.) Configure information can also be given in the body of the MoML file as follows:

```
        <entity name="sink" class="ptolemy.actor.lib.SequencePlotter">
          <configure>
            configure information
          </configure>
        </entity>
```

With the above syntax, the configure information must be textual data. It can contain XML markup with only one restriction: if the tag "`</configure>`" appears in the textual data, then it must be preceeded by a matching "`<configure>`". That is, any configure elements in the markup must have balanced start and end tags.[4]

You can give both a source attribute and in-line configuration information, as in the following:

```
        <entity name="sink" class="ptolemy.actor.lib.SequencePlotter">
          <configure source="url">
            configure information
          </configure>
        </entity>
```

In this case, the file data will be passed to the application first, followed by the in-line configuration data.

In Ptolemy II, the configure element is supported by any class that implements the Configurable interface. That interface defines a configure() method that accepts an input stream. Both external file data and in-line data are provided to the class as a character stream by calling this method.

There is a subtle limitation with using markup within the configure element. If any of the elements within the configure element match MoML elements, then the MoML DTD will be applied to assign default values, if any, to their attributes. Thus, this mechanism works best if the markup within the configure element is not using an XML schema that happens to have element names that match those in MoML. Alternatively, if it does use MoML element names, then those elements are used with their MoML meaning. This limitation can be fixed using XML namespaces, something we will eventually implement.

1.3.7 Doc Element

Some elements can be documented using the *doc* element. For example,

```
        <entity name="source" class="ptolemy.actor.lib.Ramp">
          <property name="init" value="5">
```

4. XML allow markup to be included in arbitrary data as long as it appears within either a processing instruction or a CDATA body. However, for reasons that would only baffle anyone familiar with modern programming languages, processing instructions and CDATA bodies cannot be nested within one another. The MoML configure element can be nested, so it offers a much more flexible mechanism than the standard ones in XML.

```
         <doc>Initialize the ramp above the default because... </doc>
      </property>
      <doc>
      This actor produces an increasing sequence beginning with 5.
      </doc>
   </entity>
```

With the above syntax, the documentation information must be textual data. It can include markup, as in the following example, which uses XHTML[5] formatting within the doc element:

```
   <entity name="source" class="ptolemy.actor.lib.Ramp">
      <doc><H1>Using HTML</H1>Text with <I>markup</I>.</doc>
   </entity>
```

An alternative method is to use an XML processing instruction as follows:

```
   <entity name="source" class="ptolemy.actor.lib.Ramp">
      <doc><?xhtml <H1>Using HTML</H1>Text with <I>markup</I>.?></doc>
   </entity>
```

This requires that any utility that uses the documentation information be able to handle the `xhtml` processing instruction, but it makes it very clear that the contents are XHTML. However, for reasons we do not understand, XML does not allow processing instructions to be nested, so this technique has its limitations.

More than one doc element can be included in an element. To do this, give each doc element a name, as follows:

```
   <entity name="entityname" class="classname">
      <doc name="docname">
         doc contents
      </doc>
   </entity>
```

The name must not conflict with any preexisting property. If a doc element or a property with the specified name exists, then it is removed and replaced with the property. If no name is given, then the doc element is assigned the name "_doc".

A common convention, used in Ptolemy II, is to add doc elements with the name "tooltip" to define a tooltip for GUI views of the component. A tooltip is a small window with short documentation that pops up when the mouse lingers on the graphical component.

Note that the same limitation of using markup within configure elements also applies to doc elements.

5. XHTML is HTML with additional constraints so that it conforms with XML syntax rules. In particular, every start tag must be matched by an end tag, something that ordinary HTML does not require (but fortunately, does allow).

1.3.8 Ports

An entity can declare a port:

```
<entity name="A" class="classname">
  <port name="out"/>
</entity>
```

In the above example, no class is given for the port. If a port with the specified name already exists in the class for entity A, then that port is the one referenced. Otherwise, a new port is created in Ptolemy II by calling the newPort() method of the container. Alternatively, we can specify a class name, as in

```
<entity name="A" class="classname">
  <port name="out" class="classname"/>
</entity>
```

In this case, a port will be created if one does not already exist. If it does already exist, then its class is checked for consistency with the declared class (the pre-existing port must be an instance of the declared class). In Ptolemy II, the typical classname for a port would be

```
ptolemy.actor.TypedIOPort
```

In Ptolemy II, the container of a port is required to be an instance of ptolemy.kernel.Entity or a derived class.

It is often useful to declare a port to be an input, an output, or both. To do this, enclose in the port a property named “input” or “output” or both, as in the following example:

```
<port name="out" class="ptolemy.actor.IOPort">
  <property name="output"/>
</port>
```

This is an example of a pure property. Optionally, the property can be given a boolean value, as in

```
<port name="out" class="ptolemy.actor.IOPort">
  <property name="output" value="true"/>
</port>
```

The value can be either “true” or “false”, where the latter will define the port to not be an output. A port can be defined to be both an input and an output, as follows

```
<port name="out" class="ptolemy.actor.IOPort">
  <property name="output" value="true"/>
  <property name="input" value="true"/>
</port>
```

It is also sometimes necessary to declare that a port is a multiport. To do this, enclose in the port a property named “multiport” as in the following example:

```
<port name="out" class="ptolemy.actor.IOPort">
  <property name="multiport"/>
</port>
```

The enclosing port must be an instance of IOPort (or a derived class such as TypedIOPort), or else the property is treated as an ordinary property. As with the input and output attribute, the multiport property can be given a boolean value, as in

```
<port name="out" class="ptolemy.actor.IOPort">
  <property name="multiport" value="true"/>
</port>
```

If a port is an instance of TypedIOPort (for library actors, most are), then you can set the type of the port in MoML as follows:

```
<port name="out" class="ptolemy.actor.IOPort">
  <property name="type"
            value="double"
            class="ptolemy.actor.TypeAttribute"/>
</port>
```

This is occasionally useful when you need to constrain the types beyond what the built-in type system takes care of. The names of the built-in types are (currently) boolean, booleanMatrix, complex, complexMatrix, double, doubleMatrix, fix, fixMatrix, int, intMatrix, long, longMatrix, object, string, and general. These are defined in the class ptolemy.data.type.BaseType.

1.3.9 Relations and Links

To connect entities, you create relations and links. The following example describes the topology shown in figure 1.6:

```
<entity name="top" class="*classname*">
```

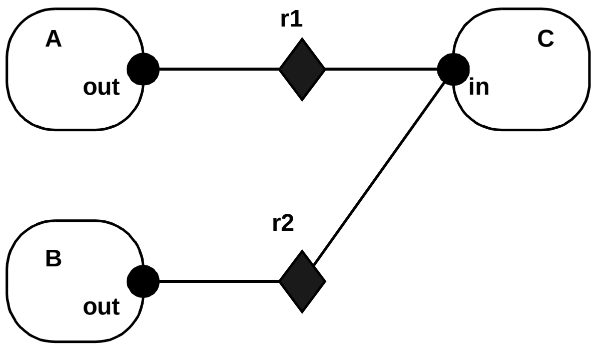


FIGURE 1.6. Example topology.

```
    <entity name="A" class="classname">
      <port name="out"/>
    </entity>
    <entity name="B" class="classname">
      <port name="out"/>
    </entity>
    <entity name="C" class="classname">
      <port name="in">
        <property name="multiport"/>
      </port>
    </entity>
    <relation name="r1" class="classname"/>
    <relation name="r2" class="classname"/>
    <link port="A.out" relation="r1"/>
    <link port="B.out" relation="r2"/>
    <link port="C.in" relation="r1"/>
    <link port="C.in" relation="r2"/>
  </entity>
```

In Ptolemy II, the typical classname for a relation would be `ptolemy.actor.TypedIORelation`. The classname may be omitted, in which case the newRelation() method of the container is used to create a new relation. The container is required to be an instance of ptolemy.kernel.CompositeEntity, or a derived class. As usual, the class attribute may be omitted if the relation already exists in the containing entity.

Notice that this example has two distinct links to `C.in` from two different relations. The order of these links may be important to a MoML tool, so any MoML tool must preserve the order in which they are specified, as Ptolemy II does. We say that C has two links, indexed 0 and 1.

The `link` element can explicitly give the index number at which to insert the link. For example, we could have achieved the same effect above by saying

```
    <link port="C.in" relation="r1" insertAt="0"/>
    <link port="C.in" relation="r2" insertAt="1"/>
```

Whenever the insertAt option is not specified, the link is always appended to the end of the list of links.

When the insertAt option is specified, the link is inserted at that position, so any pre-existing links with larger indices will have their index numbers incremented. For example, if we do

```
    <link port="C.in" relation="r1" insertAt="0"/>
    <link port="C.in" relation="r2" insertAt="1"/>
    <link port="C.in" relation="r3" insertAt="1"/>
```

then there will be a link to r1 with index 0, a link to r2 with index 2 (note! not 1), and a link to r3 with index 1.

If the specified index is beyond the existing number of links, then null links (i.e. links to nothing) are created to fill in. So for example, if the first link we create is given by

```
    <link port="C.in" relation="r2" insertAt="1"/>
```

then the port will have *two* links, not one, but the first one will be an empty link. If we then say

```
<link port="C.in" relation="r2"/>
```

then the port will have *three* links, with the first one being empty. If we then say

```
<link port="C.in" relation="r2" insertAt="0"/>
```

then there will be *four* links, with the *second* one being empty.

Note that the index number is not the same thing as the channel number in Ptolemy II. In Ptolemy II, a relation may have a width greater than one, so a single link may represent more than one channel (actually, it could even represent zero channels if that relation is not linked to another ports).

1.3.10 Classes

So far, entities have been instances of externally defined classes accessed via a class loader. They can also be instances of classes defined in MoML. To define a class in MoML, use the *class* element, as in the following example:[6]

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE class PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
  "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<class name="Gen" extends="ptolemy.actor.TypedCompositeActor">
  <entity name="ramp" class="ptolemy.actor.lib.Ramp">
    <port name="output"/>
    <property name="step" value="2*PI/50"/>
  </entity>
  <entity name="sine" class="ptolemy.actor.lib.TrigFunction">
    <port name="input"/>
    <port name="output"/>
  </entity>
  <port name="output" class="ptolemy.actor.TypedIOPort"/>
  <relation name="r1" class="ptolemy.actor.TypedIORelation"/>
  <relation name="r2" class="ptolemy.actor.TypedIORelation"/>
  <link port="ramp.output" relation="r1"/>
  <link port="sine.input" relation="r1"/>
  <link port="sine.output" relation="r2"/>
  <link port="output" relation="r2"/>
</class>
```

The class element may be the top-level element in a file, in which case the `DOCTYPE` should be declared as “class” as done above. It can also be nested within a model. The above example specifies the topology shown in figure 1.7. Once defined, it can be instantiated as if it were a class loaded by the class loader:

6. This is a simplified version of the Sinewave class, whose complete definition is given in the appendix.

```
<entity name="instancename" class="classname"/>
```

or

```
<entity name="instancename" class="classname" source="url"/>
```

The first form can be used if the class definition can be found from the *classname*. There are two ways that this could happen. First, the *classname* might be an absolute name for a class defined within the same top level entity that this entity element is in. Second, the *classname* might be sufficient to find the class definition in a file, much the way Java classes are found. For example, if the classname is `ptolemy.actor.lib.Sinewave` and the class is defined in the file `$PTII/ptolemy/actor/lib/Sinewave.xml`, then there is no need to use the second form to specify the URL where the class is defined. Specifically, the CLASSPATH[7] is searched for a file matching the classname. By convention, the file defining the class has the same name as the class, with the extension "`.xml`" or "`.moml`".

In the first of these techniques, the class name follows the same convention as entity names, except that a classname referring to a class defined within the same MoML top-level must be absolute. In fact, a class *is* an entity with the additional feature that one can create new instances of it with the entity element. Consider for example,

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="top" extends="ptolemy.kernel.CompositeEntity">
  <class name="Gen" extends="ptolemy.actor.TypedCompositeActor">
    class definition ...
  </class>
  <entity name="derived" class=".top.Gen"/>
</entity>
```

Here, the entity `derived` is an instance of `.top.Gen`, which is defined within the same MoML top level. The absolute class name is "`.top.Gen`".

The ability to give a URL as the source of a class definition is very powerful. It means that a model may be build from component libraries that are defined worldwide. There is no need to localize these.

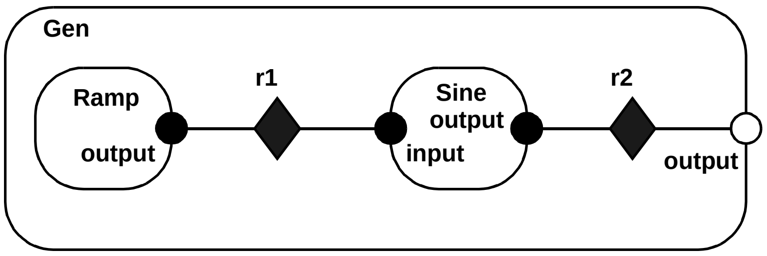


FIGURE 1.7. Sine wave generator topology.

7. CLASSPATH is an environment variable that Java uses to find Java classes. The Ptolemy II implementation of MoML simply leverages this so that MoML classes can also be found if they are on the CLASSPATH.

Of course, referencing a URL means the usual risks that the link will become invalid. It is our hope that reliable and trusted sources of components will emerge who will not allow this to happen.

The `Gen` class given at the beginning of this subsection generates a sine wave with a period of 50 samples. It is not all that useful without being parameterized. Let us extend it and add properties:[8]

```
<class name="Sinegen" extends="Gen">
  <property name="samplingFrequency"
            value="8000.0"
            class="ptolemy.data.expr.Parameter">
    <doc>The sampling frequency in Hertz.</doc>
  </property>
  <property name="frequency"
            value="440.0"
            class="ptolemy.data.expr.Parameter">
    <doc>The frequency in Hertz.</doc>
  </property>
  <property name="ramp.step"
            value="frequency*2*PI/samplingFrequency">
    <doc>Formula for the step size.</doc>
  </property>
  <property name="ramp.init"
            value="phase">
  </property>
</class>
```

This class extends Gen by adding two properties, and then sets the properties of the component entities to have values that are expressions.

1.3.11 Inheritance

MoML supports inheritance by permitting you to extend existing classes. For example, consider the following MoML file:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
  "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="top" class="ptolemy.kernel.CompositeEntity">
  <class name="base" extends="ptolemy.kernel.CompositeEntity">
    <entity name="e1" class="ptolemy.kernel.ComponentEntity">
    </entity>
  </class>
  <class name="derived" extends=".top.base">
    <entity name="e2" class="ptolemy.kernel.ComponentEntity"/>
  </class>
  <entity name="instance" extends=".top.derived"/>
</entity>
```

8. This is still not quite as elaborate as the Sinewave class defined in the appendix, which is why we give it a slightly different name, Sinegen.

Here, the “derived” class extends the “base” class by adding another entity to it, and “instance” is an instance of derived. The class “derived” can also give a source attribute, which gives a URL for the source definition.

1.3.12 Directors

Recall that a clustered graph in MoML has no semantics. However, a particular model has semantics. It may be a dataflow graph, a state machine, a process network, or something else. To give it semantics, Ptolemy II requires the specification of a director associated with a model, an entity, or a class. The director is a property of the model. The following example gives discrete-event semantics to a Ptolemy II model:

```
<entity name="top" class="ptolemy.actor.TypedCompositeActor">
  <property name="director"
            class="ptolemy.domains.de.kernel.DEDirector">
    <property name="stopTime" value="100.0"/>
  </director>
  ...
</entity>
```

This example also sets a property of the director. The name of the director is not important, except that it cannot collide with the name of any other property in the model.

1.3.13 Input Element

It is possible to insert MoML from another file or URL into a particular point in your model. For example:

```
<entity name="top" class="...">
    <entity name="a" class="...">
        <input source="url"/>
    </entity>
</entity>
```

This takes the contents of the URL specified in the source attribute of the input element and places them inside the entity named “a”. The base of the current document (the one containing the import statement) is used to interpret a relative URL, or if the current document has no base, then the current working directory is used, or if that fails, the current CLASSPATH.

1.3.14 Annotations for Visual Rendering

The abstract syntax of MoML, clustered graphs, is amenable to visual renditions as bubble and arc diagrams or as block diagrams. To support tools that display and/or edit MoML files visually, MoML allows a relation to have multiple vertices that form a path. Links can then be made to individual vertices. Consider the following example:

```
<relation name="r" class="ptolemy.actor.TypedIORelation">
  <vertex name="v1" class="classname" value="location"/>
  <vertex name="v2" class="classname" value="location" pathTo="v1"/>
```

```
</relation>
<link port="A.out" relation="r" vertex="v1"/>
<link port="B.in" relation="r" vertex="v1"/>
<link port="C.in" relation="r" vertex="v2"/>
```

This assumes that there are three entities named *A*, *B*, and *C*. The relation is annotated with a set of vertices, `v1` and `v2`, which will normally be rendered as graphical objects. The vertices are linked together with paths, which in a simple visual tool might be straight lines, or in a more sophisticated tool might be autorouted paths. In the above example, `v1` and `v2` are linked by a path. The link elements specify not just a relation, but also a vertex within that relation. This tells the visual rendering tool to draw a path from the specified port to the specified vertex.

Figure 1.8 illustrates how the above fragment might be rendered. The square boxes are icons for the three entities. They have ports with arrowheads suggesting direction. There is a single relation, which shows up visually only as a set of lines and two vertices. The vertices are shown as small diamonds.

A vertex is exactly like a property, except that it has an additional attribute, pathTo, used to link vertices, and it can be referenced in a link element. Like any other property, it has a class attribute, which specifies the class implementing the vertex. In Ptolemy II, the class for a vertex is typically ptolemy.moml.Vertex. Like other properties, a vertex can have a value. This value will typically specify a location for a visual rendition. For example, in Ptolemy II, the first vertex above might be given as

```
<vertex name="v1"
        class="ptolemy.moml.Vertex"
        value="184.0, 93.0"/>
```

This indicates that the vertex should be rendered at the location 184.0, 93.0.

Ptolemy II uses ordinary MoML properties to specify other visual aspects of a model. First, an entity can contain a location property, which is a hint to a visual renderer, as follows:

```
<entity name="ramp" class="ptolemy.actor.lib.Ramp">
  <property name="location"
            class="ptolemy.moml.Location"
            value="50.0, 50.0"/>
</entity>
```

This suggests to the visual renderer that the Ramp actor should be drawn at location 50.0, 50.0.

Ptolemy II also supports a powerful and extensible mechanism for specifying the visual rendition

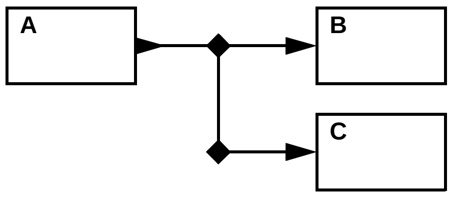


FIGURE 1.8. Example showing how MoML might be visually rendered.

of an entity. Consider the following example:

```
<entity name="ramp" class="ptolemy.actor.lib.Ramp">
  <property name="location"
           class="ptolemy.moml.Location"
           value="50.0, 50.0"/>
  <property name="iconDescription"
           class="ptolemy.kernel.util.SingletonAttribute">
     <configure><svg>
       <rect x="0" y="0" width="80" height="20"
         style="fill:green;stroke:black;stroke-width:5"/>
     </svg></configure>
  </property>
</entity>
```

The SingletonAttribute class is used to attach an XML description of the rendition, which in this case is a wide box filled with green. The XML schema used to define the icon is SVG (scalable vector graphics), which can be found at http://www.w3.org/TR/SVG/.[9]

The rendering of the icon is done by another property of class XMLIcon, which need not be explicitly specified because the visual renderer will create it if it isn't present. However, it is possible to create totally customized renditions by defining classes derived from XMLIcon, and attaching them to entities as properties. This is beyond the scope of this chapter.

1.4 Incremental Parsing

MoML may be used as a command language to modify existing models, as well as being used to specify complete models. This technique is known as *incremental parsing*.

1.4.1 Adding Entities

Consider for example the simple model created as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/moml.dtd">
<entity name="top" class="ptolemy.actor.TypedCompositeActor">
    ... contents of the model ...
</entity>
```

Later, the following MoML element can be used to add an entity to the model:

```
<entity name=".top">
    <entity name="inside" class="ptolemy.actor.TypedCompositeActor"/>
</entity>
```

9. Currently, the Diva graphics infrastructure, which is used by Vergil to render these icons, only supports a small subset of SVG. Eventually, we hope it will support the full specification.

The name of the outer entity "`.top`" is the name of the top-level model created by the first segment of MoML. (Recall that the leading period means that the name is absolute.) The line

```
<entity name=".top">
```

defines the context for evaluation of the element

```
    <entity name="inside" class="ptolemy.actor.TypedCompositeActor"/>
```

Any entity constructed in a previous parsing phase can be specified as the context for evaluation of a new MoML element.

Of course, the MoML parser must have a reference to the context in order to later parse this incremental element. This is accomplished by either using the same parser, which keeps track of the top-level entity in the last model it parsed, or by calling the setTopLevel() or setContext() methods of the parser, passing as an argument the model.

1.4.2 Using Absolute Names

Above, we have used the fact that an entity element can refer to a pre-existing element by name. That name can be relative to the context in which the entity element exists, or it can be absolute. If it is absolute, then it must nonetheless be properly contained by the enclosing entity. The following example is incorrect, and will trigger an exception:

```
<entity name="top" class="ptolemy.actor.TypedCompositeActor">
  <entity name="a" class="ptolemy.actor.TypedCompositeActor"/>
  <entity name="b" class="ptolemy.actor.TypedCompositeActor">
     <entity name=".top.a"/>
  </entity>
</entity>
```

The "`.top.a`" cannot be specified within "b" because it is already contained within "`top`."

1.4.3 Adding Ports, Relations, and Links

A port or relation can be added to an entity that has been previously constructed by the parser. For example, assuming that `.top.inside` has been constructed as before, we can add a port to it with the following MoML segment:

```
<entity name=".top.inside">
    <port name="input" class="ptolemy.actor.TypedIOPort"/>
</entity>
```

A relation and link can then be added as follows:

```
<entity name=".top">
    <relation name="r" class="ptolemy.actor.TypedIORelation"/>
    <link port="inside.input" relation="r"/>
</entity>
```

1.4.4 Changing Port Configurations

A port that is an input can be converted to an output with the following MoML segment:

```
<port name="portname">
    <property name="input" value="false"/>
    <property name="output" value="true"/>
</port>
```

A port can be made into a multiport as follows:

```
<port name="portname">
    <property name="multiport" value="true"/>
</port>
```

1.4.5 Deleting Entities, Relations, and Ports

An entity that has been previously constructed by a parser can be deleted by evaluating MoML. For example, assuming that `.top.inside` has been constructed as before, we can delete it with the following MoML segment:

```
<entity name=".top">
    <deleteEntity name="inside"/>
</entity>
```

Any links to ports of the entity will also be deleted. Similarly, relations can be deleted using the `deleteRelation` element, and ports can be deleted using the `deletePort` element.

1.4.6 Renaming Objects

A previously existing entity can be renamed using the `rename` element, as follows:

```
<entity name="entityName">
    <rename name="newName"/>
</entity>
```

The new name is required to not have any periods in it. It consists of alphanumeric characters, the underscore, and spaces.

1.4.7 Changing Documentation, Properties, and Directors

Documentation is attached to entities using the doc element (see section 1.3.7). A doc element can optionally be given a name; if no name is given, then the name is implicitly “_doc”. To replace a doc element, just give a new doc element with the same name. To remove a doc element, give a doc element with the same name and an empty body, as in

```
<doc name="docname"></doc>
```

or

```
<doc name="docname"/>
```

Properties can have their value changed using the property element (see section 1.3.6) with a new value, for example:

```
<property name="propertyname" value="propertyvalue"/>
```

A property can be deleted using the deleteProperty element

```
<deleteProperty name="propertyname"/>
```

Since a director is a property, this same mechanism can be used to remove a director.

1.4.8 Removing Links

To remove individual links, use the `unlink` element. This element has three forms. The first is

```
<unlink port="portname" relation="relationname"/>
```

This unlinks a port from the specified relation. If the port is linked more than once to the specified relation, then all links to this relation are removed. It makes no difference whether the link is an inside link or an outside link, since this can be determined from the containers of the port and the relation.

The second and third forms are

```
<unlink port="portname" index="linknumber"/>
<unlink port="portname" insideIndex="linknumber"/>
```

These both remove a link by index number. The first is used for an outside link, and the second for an inside link. The valid indices range from 0 to one less than the number of links that the port has. If the port is not a multiport, then there is at most one valid index, number 0. If an invalid index is given then the element is ignored. Note that the indexes of links above that of the removed link will be decremented by one.

The unlink element can also be used to remove null links. For example, if we have created a link with

```
<link port="portname" relation="r" insertAt="1"/>
```

where there was previously no link on this port, then this leaves a null link (not linked to anything) with index 0 (see section 1.3.9), and of course a link to relation `r` with index 1. The null link can be removed with

```
<unlink port="portname" insideIndex="0"/>
```

which leaves the link to `r` as the sole link, having index 0.

Note that the index is not the same thing as the channel number. A relation may have a width greater than one, so a single link may represent more than one channel (actually, it could even represent zero channels if that relation is not linked to other suitable ports).

1.4.9 Grouping Elements

Occasionally, you may wish to incrementally parse a set of elements. For example, in the Ptolemy II implementation, the parser has a method for setting the context, so you could set the context to a CompositeEntity and then create several entities by parsing the following MoML:

```
        <entity name="firstEntity" class="classname"/>
        <entity name="firstEntity" class="classname"/>
        <entity name="firstEntity" class="classname"/>
```

However, the XML parser will fail to parse this because it requires that there be a single top-level element. The group element is provided for this purpose:

```
    <group>
        <entity name="firstEntity" class="classname"/>
        <entity name="firstEntity" class="classname"/>
        <entity name="firstEntity" class="classname"/>
    </group>
```

This element is ignored by the parser, in that it does not define a new container for the enclosed entities. It simply aggregates them, leaving the context the same as it is for the group element itself.

The group element may be given a name attribute, in which case it defines a *namespace*. All named objects (such as entities) that are immediately inside the group will have their names modified by prepending them with the name of the group and a colon. For example,

```
    <group name="a">
        <entity name="b" class="classname">
            <entity name="c" class="classname"/>
        </entity>
    </group>
```

The entity “b” will actually be named “a:b”. The entity “c” will not be affected by the group name. Its full name, however, will be “a:b.c”.

1.5 Parsing MoML

MoML is intended to be a generic modeling markup language, not one that is specialized to Ptolemy II. As such, Ptolemy II may be viewed as a reference implementation of a MoML tool. In Ptolemy II, MoML is supported primarily by the moml package.

The moml package contains the classes shown in figure 1.9 (see appendix A of chapter 1 for UML syntax). The basis for the MoML parser is the parser distributed by Microstar. The parse() methods of the MoMLParser class read MoML data and construct a Ptolemy II model. They return the top-level model. The same parser can then be used to incrementally parse MoML segments to modify that model.

The EntityLibrary class takes particular advantage of MoML. This class extends CompositeEntity, and is designed to contain a library of entities. But it is carefully designed to avoid instantiating those entities until there is some request for them. Instead, it maintains a MoML representation of the library. This allows for arbitrarily large libraries without the overhead of instantiating components in the library that might not be needed.

Incremental parsing is when a MoML parser is used to modify a pre-existing model (see section 1.4). A MoML parser that was used to create the pre-existing model can be used to modify it. If there is no such parser, then it is necessary to call the `setToplevel()` method of MoMLParser to associate the parser with the pre-existing model.

Incremental parsing should (usually) be done using a change request. A change request is an active object that makes a modification to a Ptolemy model. They are queued with a composite entity con-

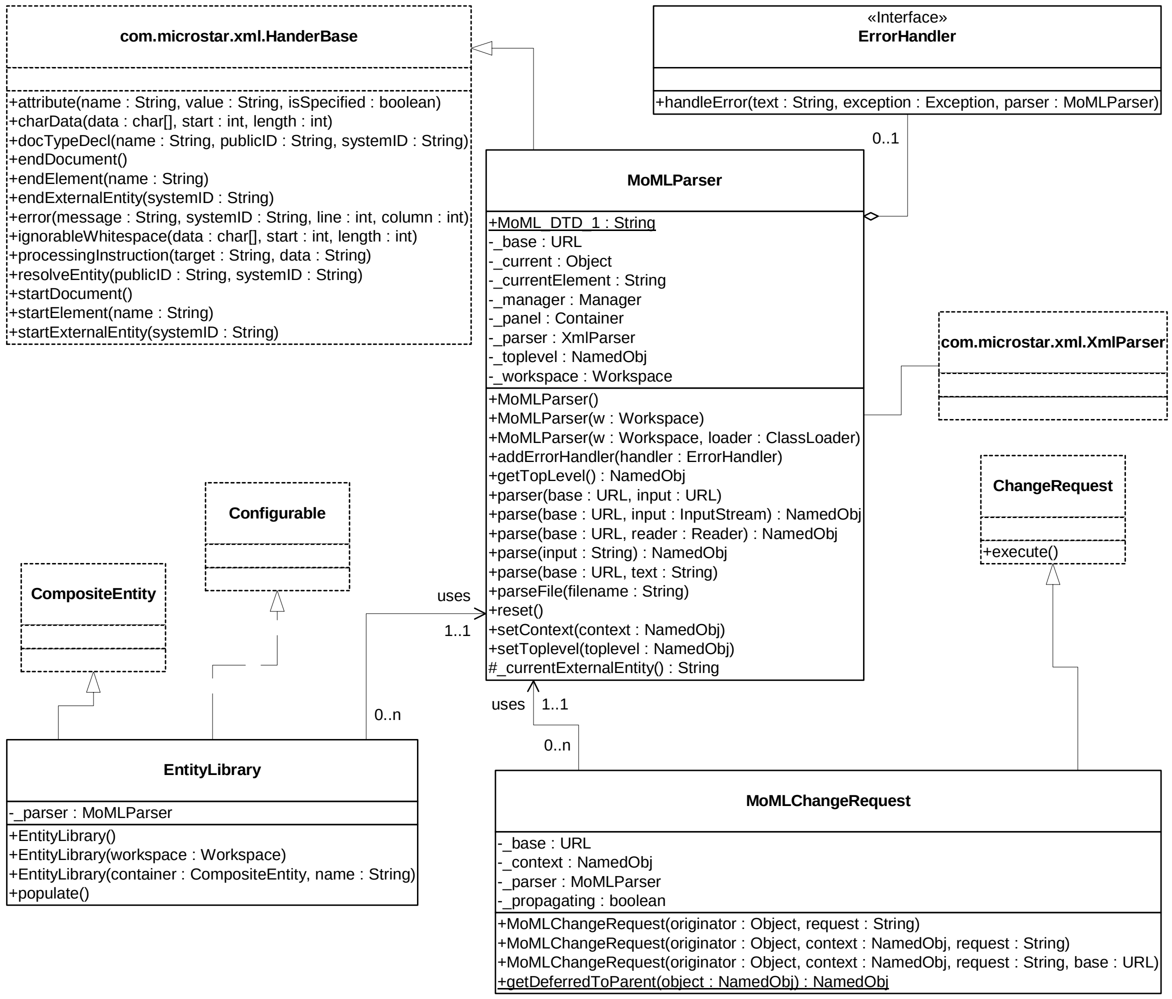

com.microstar.xml.HanderBase
+attribute(name : String, value : String, isSpecified : boolean)
+charData(data : char[], start : int, length : int)
+docTypeDecl(name : String, publicID : String, systemID : String)
+endDocument()
+endElement(name : String)
+endExternalEntity(systemID : String)
+error(message : String, systemID : String, line : int, column : int)
+ignorableWhitespace(data : char[], start : int, length : int)
+processingInstruction(target : String, data : String)
+resolveEntity(publicID : String, systemID : String)
+startDocument()
+startElement(name : String)
+startExternalEntity(systemID : String)
«Interface»
ErrorHandler
+handleError(text : String, exception : Exception, parser : MoMLParser)
0..1
MoMLParser
+MoML_DTD_1 : String
-_base : URL
-_current : Object
-_currentElement : String
-_manager : Manager
-_panel : Container
-_parser : XmlParser
-_toplevel : NamedObj
-_workspace : Workspace
+MoMLParser()
+MoMLParser(w : Workspace)
+MoMLParser(w : Workspace, loader : ClassLoader)
+addErrorHandler(handler : ErrorHandler)
+getTopLevel() : NamedObj
+parser(base : URL, input : URL)
+parse(base : URL, input : InputStream) : NamedObj
+parse(base : URL, reader : Reader) : NamedObj
+parse(input : String) : NamedObj
+parse(base : URL, text : String)
+parseFile(filename : String)
+reset()
+setContext(context : NamedObj)
+setToplevel(toplevel : NamedObj)
#_currentExternalEntity() : String
com.microstar.xml.XmlParser
ChangeRequest
+execute()
Configurable
CompositeEntity
uses
1..1
0..n
uses
1..1
0..n
EntityLibrary
-_parser : MoMLParser
+EntityLibrary()
+EntityLibrary(workspace : Workspace)
+EntityLibrary(container : CompositeEntity, name : String)
+populate()
MoMLChangeRequest
-_base : URL
-_context : NamedObj
-_parser : MoMLParser
-_propagating : boolean
+MoMLChangeRequest(originator : Object, request : String)
+MoMLChangeRequest(originator : Object, context : NamedObj, request : String)
+MoMLChangeRequest(originator : Object, context : NamedObj, request : String, base : URL)
+getDeferredToParent(object : NamedObj) : NamedObj

FIGURE 1.9. Classes supporting MoML parsing in the moml package.

tainer by calling its requestChange() method. This ensures that the mutation is executed only when it is safe to modify the structure of the model. The class MoMLChangeRequest (see figure 1.9) can be used for this purpose. Simply create an instance of this class, providing the constructor with a string containing the MoML code that specifies the change.

The exportMoML() methods of Ptolemy II objects can be used to produce a MoML file given a model. Thus, MoML can be used as the persistent file format for Ptolemy II models

1.6 Exporting MoML

Almost any Ptolemy II object can export a MoML description of itself. The following methods of NamedObj (and derived classes) are particularly useful:

```
exportMoML(): String
exportMoML(output: Writer)
exportMoML(output: Writer, depth: int)
exportMoML(output: Writer, depth: int, name: String)
_exportMoMLContents(output: Writer, depth: int)
```

Since any object derived from NamedObj can export MoML, MoML becomes an effective persistent format for Ptolemy II models. Almost everything in Ptolemy II is derived from NamedObj. It is much more compact than serializing the objects, and the description is much more durable (since serialized objects are not guaranteed to load properly into future versions of the Java virtual machine).

There is one significant subtlety that occurs when an entity is instantiated from a class defined in MoML. Consider the example:

```
<entity name="top" class="ptolemy.kernel.CompositeEntity">
    <class name="master" extends="ptolemy.kernel.ComponentEntity">
        <port name="p" class="ptolemy.kernel.ComponentPort"/>
    </class>
    <entity name="derived" class=".top.master"/>
</entity>
```

This model defines one class and one entity that instantiates that class. When we export MoML for this top-level model, we get:

```
<entity name="top" class="ptolemy.kernel.CompositeEntity">
    <class name="master" extends="ptolemy.kernel.ComponentEntity">
        <port name="p" class="ptolemy.kernel.ComponentPort">
        </port>
    </class>
    <entity name="derived" class=".top.master">
    </entity>
</entity>
```

Aside from some minor differences in syntax, this is identical to our specification above. In particular, note that the entity “derived” does not describe its port “p” even though it certainly has such a port. That port is implied because the entity instantiates the class “`.top.master`”.

Suppose that using incremental parsing we subsequently modify the model as follows:

```
<entity name=".top.derived">
    <port name="q" class="ptolemy.kernel.ComponentPort"/>
</entity>
```

That is, we add a port to the instantiated entity. Then the added port *is* exported when we export MoML. That is, we get:

```
<entity name="top" class="ptolemy.kernel.CompositeEntity">
    <class name="master" extends="ptolemy.kernel.ComponentEntity">
        <port name="p" class="ptolemy.kernel.ComponentPort">
        </port>
    </class>
    <entity name="derived" class=".top.master">
        <port name="q" class="ptolemy.kernel.ComponentPort">
        </port>
    </entity>
</entity>
```

This is what we would expect. The entity is based on the specified class, but actually extends it with additional features. Those features are persistent.

Properties are treated more simply. They are always described when MoML is exported, regardless of whether they are defined in the class on which an entity is based. The reason for this is that properties are usually modified in instances, for example by giving them new values.

There is an additional subtlety. If a topology is modified by making direct kernel calls, then exportMoML() will normally export the modified topology. However, if a derived component is modified by direct kernel calls, then exportMoML() will fail to catch the changes. In fact, only if the changes are made by evaluating MoML will the modifications be exported. This actually can prove to be convenient. It means that if a model mutates during execution, and is later saved, that a user interface can ensure that only the original model, before mutations, is saved.

1.7 Special Attributes

The moml package also includes a set of attribute classes that decorate the objects in a model with MoML-specific information, as shown in figure 1.10. These classes are used to decorate a Ptolemy II object with additional information that is relevant to a GUI or other user interface. For example, the Location class is used to specify the location of visual rendition of a component in a visual editor. A Vertex decorates a relation with one of several visual handles to which connections can be made. A MoMLAttribute decorates an object with a property that can describe itself with arbitrary MoML.

1.8 Acknowledgements

Many thanks to Ed Willink of Racal Research Ltd. and Simon North of Synopsys for many helpful suggestions, only some of which have made it into this version of MoML. Also, thanks to Tom Henzinger, Alberto Sangiovanni-Vincentelli, and Kees Vissers for helping clarify issues of abstract syntax.

Appendix A: Example

Figures 1.11 and 1.12 show a simple Ptolemy II model in the SDF domain. Figure 1.13 shows the execution window for this model. This model generates two sinusoidal waveforms and multiplies them together. This appendix gives the complete MoML code. The MoML code is divided into two files. The first of these defined a component, a sinewave generator. The second creates two instances of this sinewave generator and multiplies their outputs. The code listings are (hopefully) self-explanatory.

A.1 Sinewave Generator

The Sinewave component is defined in the file $PTII/ptolemy/actor/lib/Sinewave.xml, which is listed below. This file defines a MoML class, which can then be referenced by the class name ptolemy.actor.lib.Sinewave. The Vergil rendition of this model is shown in figure 1.11.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE class PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<class name="Sinewave" extends="ptolemy.actor.TypedCompositeActor">
    <doc>This composite actor generates a sine wave.</doc>
    <property name="samplingFrequency" class="ptolemy.data.expr.Parameter" value="8000.0">
        <doc>The sampling frequency, in the same units as the frequency.</doc>
    </property>
    <property name="frequency" class="ptolemy.data.expr.Parameter" value="440.0">
        <doc>The frequency of the sinusoid, in the same units as the sampling frequency.</doc>
```

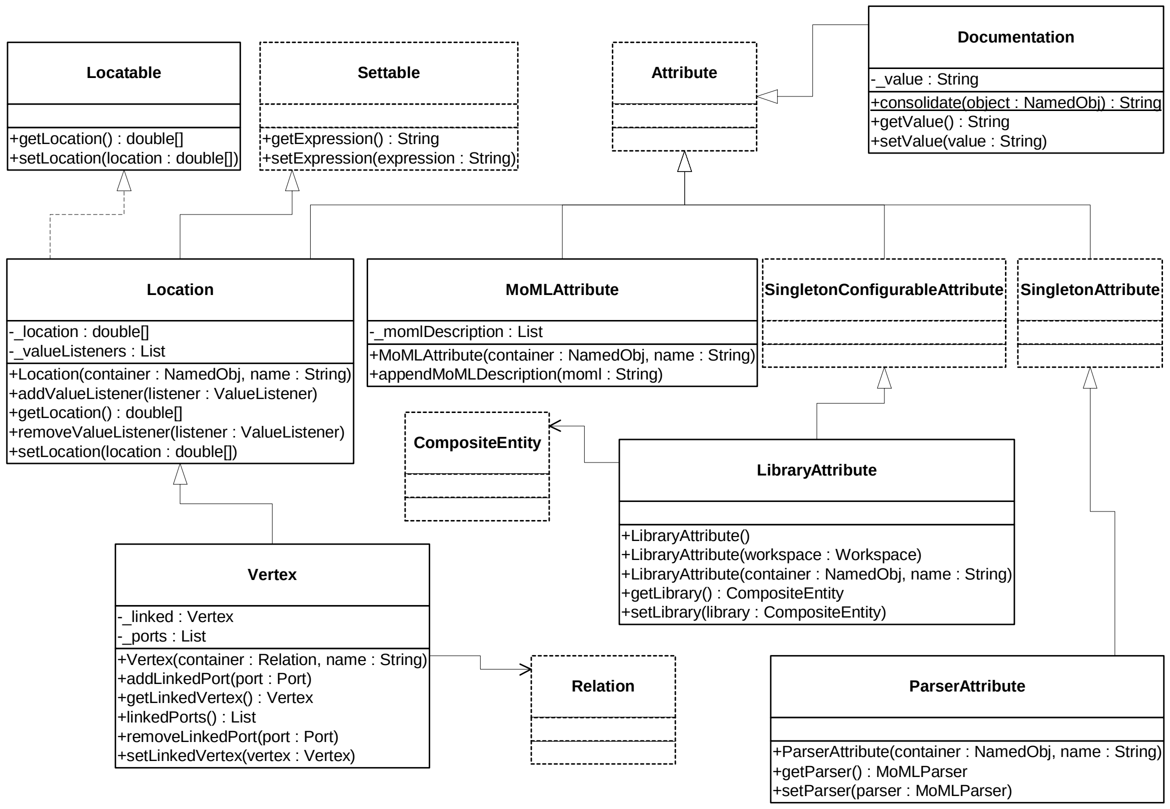

Locatable
+getLocation() : double[]
+setLocation(location : double[])
Settable
+getExpression() : String
+setExpression(expression : String)
Attribute
Documentation
-_value : String
+consolidate(object : NamedObj) : String
+getValue() : String
+setValue(value : String)
Location
-_location : double[]
-_valueListeners : List
+Location(container : NamedObj, name : String)
+addValueListener(listener : ValueListener)
+getLocation() : double[]
+removeValueListener(listener : ValueListener)
+setLocation(location : double[])
MoMLAttribute
-_momlDescription : List
+MoMLAttribute(container : NamedObj, name : String)
+appendMoMLDescription(moml : String)
SingletonConfigurableAttribute
SingletonAttribute
CompositeEntity
LibraryAttribute
+LibraryAttribute()
+LibraryAttribute(workspace : Workspace)
+LibraryAttribute(container : NamedObj, name : String)
+getLibrary() : CompositeEntity
+setLibrary(library : CompositeEntity)
Vertex
-_linked : Vertex
-_ports : List
+Vertex(container : Relation, name : String)
+addLinkedPort(port : Port)
+getLinkedVertex() : Vertex
+linkedPorts() : List
+removeLinkedPort(port : Port)
+setLinkedVertex(vertex : Vertex)
Relation
ParserAttribute
+ParserAttribute(container : NamedObj, name : String)
+getParser() : MoMLParser
+setParser(parser : MoMLParser)

FIGURE 1.10. Attributes in the moml package.

```
    </property>
    <property name="phase" class="ptolemy.data.expr.Parameter" value="0.0">
        <doc>The phase, in radians.</doc>
    </property>
    <port name="output" class="ptolemy.actor.TypedIOPort">
        <property name="output"/>
        <doc>Sinusoidal waveform output.</doc>
        <property name="_location" class="ptolemy.moml.Location" value="460.0, 225.0">
        </property>
    </port>
    <entity name="Ramp" class="ptolemy.actor.lib.Ramp">
        <property name="firingCountLimit" class="ptolemy.data.expr.Parameter" value="0">
        </property>
        <property name="init" class="ptolemy.data.expr.Parameter" value="phase">
        </property>
        <property name="step" class="ptolemy.data.expr.Parameter"
                value="frequency*2*PI/samplingFrequency">
        </property>
        <property name="_location" class="ptolemy.moml.Location" value="140.0, 225.0">
        </property>
        <port name="output" class="ptolemy.actor.TypedIOPort">
            <property name="output"/>
        </port>
        <port name="trigger" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="multiport"/>
        </port>
    </entity>
    <entity name="Sine Function" class="ptolemy.actor.lib.TrigFunction">
        <property name="function" class="ptolemy.kernel.util.StringAttribute" value="sin">
            <property name="style" class="ptolemy.actor.gui.style.ChoiceStyle">
                <property name="acos" class="ptolemy.kernel.util.StringAttribute" value="acos">
                </property>
                <property name="asin" class="ptolemy.kernel.util.StringAttribute" value="asin">
                </property>
                <property name="atan" class="ptolemy.kernel.util.StringAttribute" value="atan">
                </property>
                <property name="cos" class="ptolemy.kernel.util.StringAttribute" value="cos">
                </property>
                <property name="sin" class="ptolemy.kernel.util.StringAttribute" value="sin">
                </property>
                <property name="tan" class="ptolemy.kernel.util.StringAttribute" value="tan">
                </property>
            </property>
        </property>
```

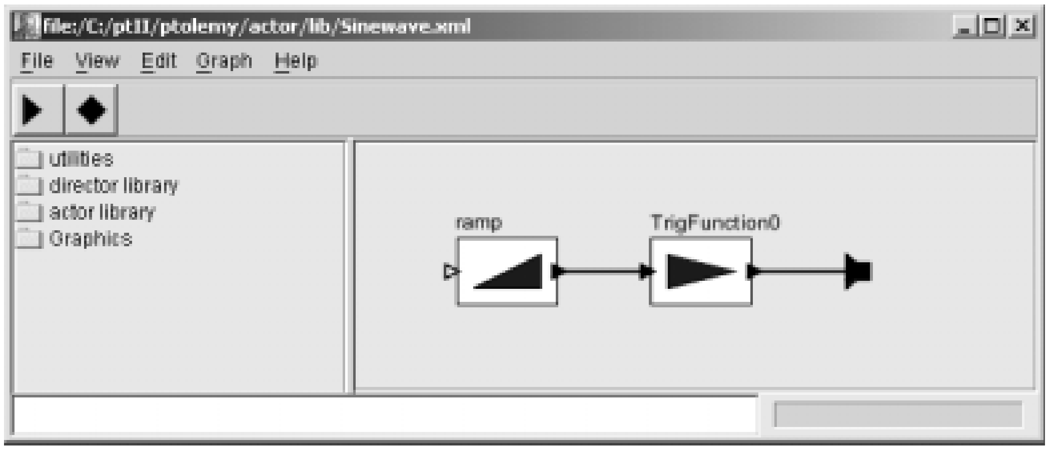


FIGURE 1.11. Rendition of the Sinewave class in Vergil 1.0.

```
        <property name="_location" class="ptolemy.moml.Location" value="300.0, 225.0">
        </property>
        <port name="input" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
        </port>
        <port name="output" class="ptolemy.actor.TypedIOPort">
            <property name="output"/>
        </port>
    </entity>
    <relation name="relation1" class="ptolemy.actor.TypedIORelation">
    </relation>
    <relation name="relation2" class="ptolemy.actor.TypedIORelation">
    </relation>
    <link port="output" relation="relation1"/>
    <link port="Ramp.output" relation="relation2"/>
    <link port="Sine Function.input" relation="relation2"/>
    <link port="Sine Function.output" relation="relation1"/>
</class>
```

A.2 Modulation

The top-level is defined in the file $PTII/ptolemy/moml/demo/modulation.xml, which is listed below. The Vergil rendition of this model is shown in figure 1.12, and its execution is shown in figure 1.13.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="modulation" class="ptolemy.actor.TypedCompositeActor">
    <doc>Multiply a low-frequency sine wave (the signal) by a higher frequency one (the carrier).</doc>
    <property name="frequency1" class="ptolemy.data.expr.Parameter" value="PI*0.2">
        <doc>Frequency of the carrier</doc>
    </property>
    <property name="frequency2" class="ptolemy.data.expr.Parameter" value="PI*0.02">
        <doc>Frequency of the sinusoidal signal</doc>
    </property>
    <property name="director" class="ptolemy.domains.sdf.kernel.SDFDirector">
        <property name="iterations" class="ptolemy.data.expr.Parameter" value="100">
            <doc>Number of iterations in an execution.</doc>
        </property>
```

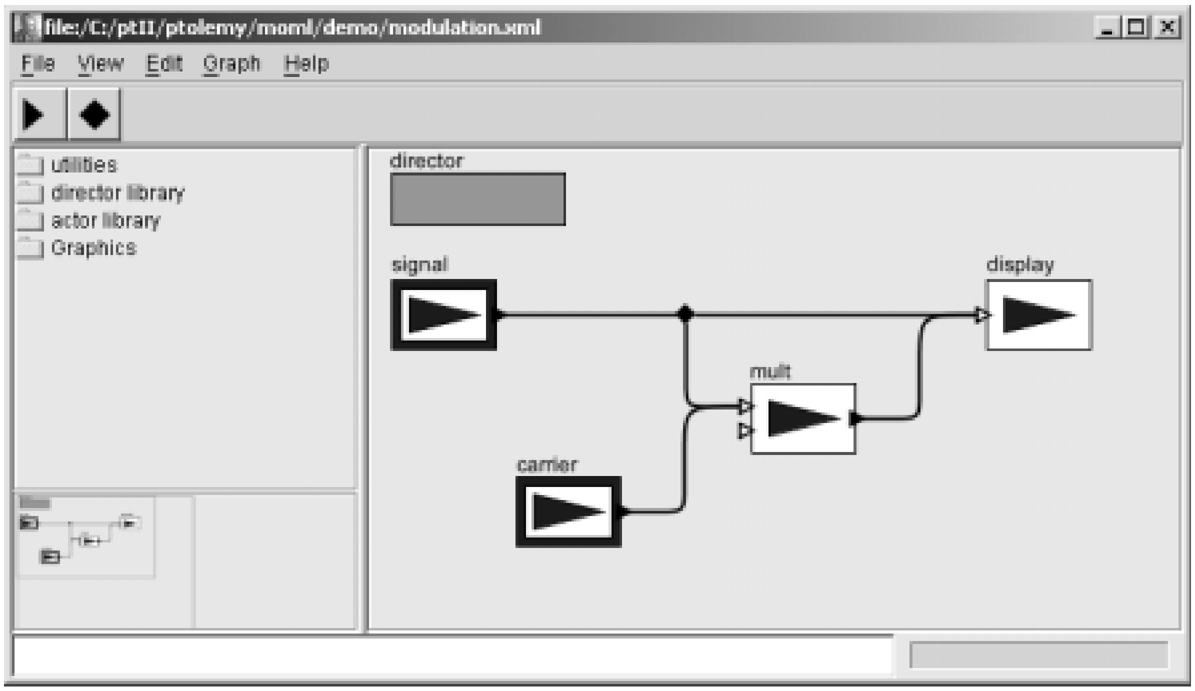
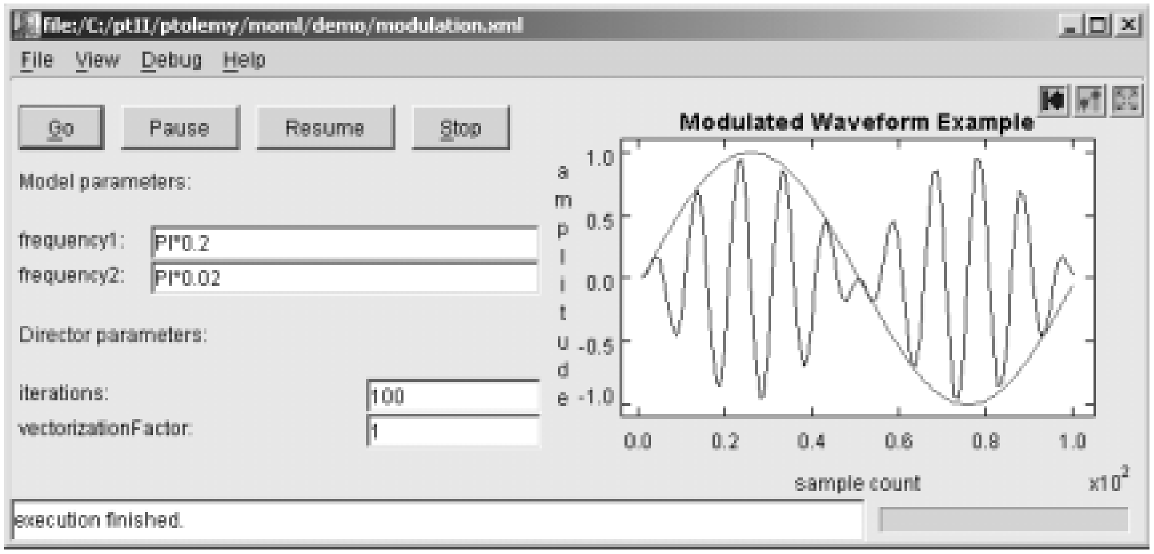


FIGURE 1.12. Rendition of the modulation model in Vergil 1.0.

```
    <property name="vectorizationFactor" class="ptolemy.data.expr.Parameter" value="1">
    </property>
    <property name="_location" class="ptolemy.moml.Location" value="62.0, 23.0">
    </property>
</property>
<entity name="carrier" class="ptolemy.actor.lib.Sinewave">
    <doc>This composite actor generates a sine wave.</doc>
    <property name="samplingFrequency" class="ptolemy.data.expr.Parameter" value="2*PI">
        <doc>The sampling frequency, in the same units as the frequency.</doc>
    </property>
    <property name="frequency" class="ptolemy.data.expr.Parameter" value="frequency1">
        <doc>The frequency of the sinusoid, in the same units as the sampling frequency.</doc>
    </property>
    <property name="phase" class="ptolemy.data.expr.Parameter" value="0.0">
        <doc>The phase, in radians.</doc>
    </property>
    <property name="_location" class="ptolemy.moml.Location" value="215.0, 250.0">
    </property>
</entity>
<entity name="signal" class="ptolemy.actor.lib.Sinewave">
    <doc>This composite actor generates a sine wave.</doc>
    <property name="samplingFrequency" class="ptolemy.data.expr.Parameter" value="2*PI">
        <doc>The sampling frequency, in the same units as the frequency.</doc>
    </property>
    <property name="frequency" class="ptolemy.data.expr.Parameter" value="frequency2">
        <doc>The frequency of the sinusoid, in the same units as the sampling frequency.</doc>
    </property>
    <property name="phase" class="ptolemy.data.expr.Parameter" value="0.0">
        <doc>The phase, in radians.</doc>
    </property>
    <property name="_location" class="ptolemy.moml.Location" value="143.0, 135.0">
    </property>
</entity>
<entity name="mult" class="ptolemy.actor.lib.MultiplyDivide">
    <property name="_location" class="ptolemy.moml.Location" value="347.0, 196.0">
    </property>
    <port name="multiply" class="ptolemy.actor.TypedIOPort">
        <property name="input"/>
        <property name="multiport"/>
    </port>
    <port name="divide" class="ptolemy.actor.TypedIOPort">
        <property name="input"/>
```



Model parameters:

frequency1: PI*0.2

frequency2: PI*0.02

Director parameters:

iterations: 100

vectorizationFactor: 1

execution finished.

FIGURE 1.13. Execution window for the modulation model.

```
            <property name="multiport"/>
        </port>
        <port name="output" class="ptolemy.actor.TypedIOPort">
            <property name="output"/>
        </port>
    </entity>
    <entity name="display" class="ptolemy.actor.lib.gui.SequencePlotter">
        <property name="fillOnWrapup" class="ptolemy.data.expr.Parameter" value="true">
        </property>
        <property name="startingDataset" class="ptolemy.data.expr.Parameter" value="0">
        </property>
        <property name="xInit" class="ptolemy.data.expr.Parameter" value="0.0">
        </property>
        <property name="xUnit" class="ptolemy.data.expr.Parameter" value="1.0">
        </property>
        <property name="_location" class="ptolemy.moml.Location" value="479.99998474121094, 135.0">
        </property>
        <port name="input" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="multiport"/>
        </port>
        <configure><?plotml
<!DOCTYPE plot PUBLIC "-//UC Berkeley//DTD PlotML 1//EN"
"http://ptolemy.eecs.berkeley.edu/xml/dtd/PlotML_1.dtd">
<plot>
<title>Modulated Waveform Example</title>
<xLabel>sample count</xLabel>
<yLabel>amplitude</yLabel>
<xRange min="1.0" max="100.0"/>
<yRange min="-1.0" max="1.0"/>
<noGrid/>
</plot>?>
        </configure>
    </entity>
    <relation name="r1" class="ptolemy.actor.TypedIORelation">
    </relation>
    <relation name="r2" class="ptolemy.actor.TypedIORelation">
        <vertex name="vertex0" class="ptolemy.moml.Vertex" value="279.0, 141.0">
        </vertex>
    </relation>
    <relation name="r3" class="ptolemy.actor.TypedIORelation">
    </relation>
    <link port="carrier.output" relation="r1"/>
    <link port="signal.output" relation="r2"/>
    <link port="mult.multiply" relation="r1"/>
    <link port="mult.multiply" relation="r2"/>
    <link port="mult.output" relation="r3"/>
    <link port="display.input" relation="r2"/>
    <link port="display.input" relation="r3"/>
</entity>
```

2

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

2.1 Abstract Syntax

The kernel defines a small set of Java classes that implement a data structure supporting a general form of uninterpreted clustered graphs, plus methods for accessing and manipulating such graphs. These graphs provide an abstract syntax for netlists, state transition diagrams, block diagrams, etc. An *abstract syntax* is a conceptual data organization. It can be contrasted with a *concrete syntax*, which is a syntax for a persistent, readable representation of the data, such as EDIF for netlists. A particular graph configuration is called a *topology.*

Although this idea of an uninterpreted abstract syntax is present in the original Ptolemy kernel [14], in fact the original Ptolemy kernel has more semantics than we would like. It is heavily biased towards dataflow, the model of computation used most heavily. Much of the effort involved in implementing models of computation that are very different from dataflow stems from having to work around certain assumptions in the kernel that, in retrospect, proved to be particular to dataflow.

A topology is a collection of *entities* and *relations*. We use the graphical notation shown in figure 2.1, where entities are depicted as rounded boxes and relations as diamonds. Entities have *ports*, shown as filled circles, and relations connect the ports. We consistently use the term *connection* to denote the association between connected ports (or their entities), and the term *link* to denote the association between ports and relations. Thus, a connection consists of a relation and two or more links.

The use of ports and hierarchy distinguishes our topologies from mathematical graphs. In a mathematical graph, an entity would be a vertex, and an arc would be a connection between entities. A vertex could be represented in our schema using entities that always contain exactly one port. In a directed graph, the connections are divided into two subsets, one consisting of incoming arcs, and the other of outgoing arcs. The vertices in such a graph could be represented by entities that contain two ports, one for incoming arcs and one for outgoing arcs. Thus, in mathematical graphs, entities always have one or two ports, depending on whether the graph is directed. Our schema generalizes this by permitting an entity to have any number of ports, thus dividing its connections into an arbitrary number of subsets.

A second difference between our graphs and mathematical graphs is that our relations are multiway associations whereas an arc in a graph is a two-way association. A third difference is that mathematical graphs normally have no notion of hierarchy (clustering).

Relations are intended to serve as mediators, in the sense of the Mediator design pattern of Gamma, *et al*. [28]. “Mediator promotes loose coupling by keeping objects from referring to each other explicitly...” For example, a relation could be used to direct messages passed between entities. Or it could denote a transition between states in a finite state machine, where the states are represented as entities. Or it could mediate rendezvous between processes represented as entities. Or it could mediate method calls between loosely associated objects, as for example in remote method invocation over a network.

2.2 Non-Hierarchical Topologies

The classes shown in figure 2.2 support non-hierarchical topologies, like that shown in figure 2.1. Figure 2.2 is a UML static structure diagram (see appendix A of chapter 1).

2.2.1 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 is uses the NamedList class shown at the bottom of figure 2.2 and defined fully in figure 2.3. There is exactly one

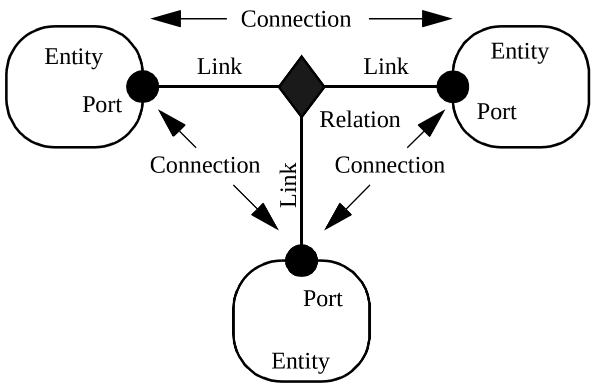


FIGURE 2.1. Visual notation and terminology.

instance of NamedList associated with Entity, and it aggregates the ports.

A Port is associated with any number of Relations (the association is called a *link*), and a Relation is associated with any number of Ports. Link associations use CrossRefList, shown in figure 2.3. There is exactly one instance of CrossRefList associated with each port and each relation. The links define a web of interconnected entities.

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.

2.2.2 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,

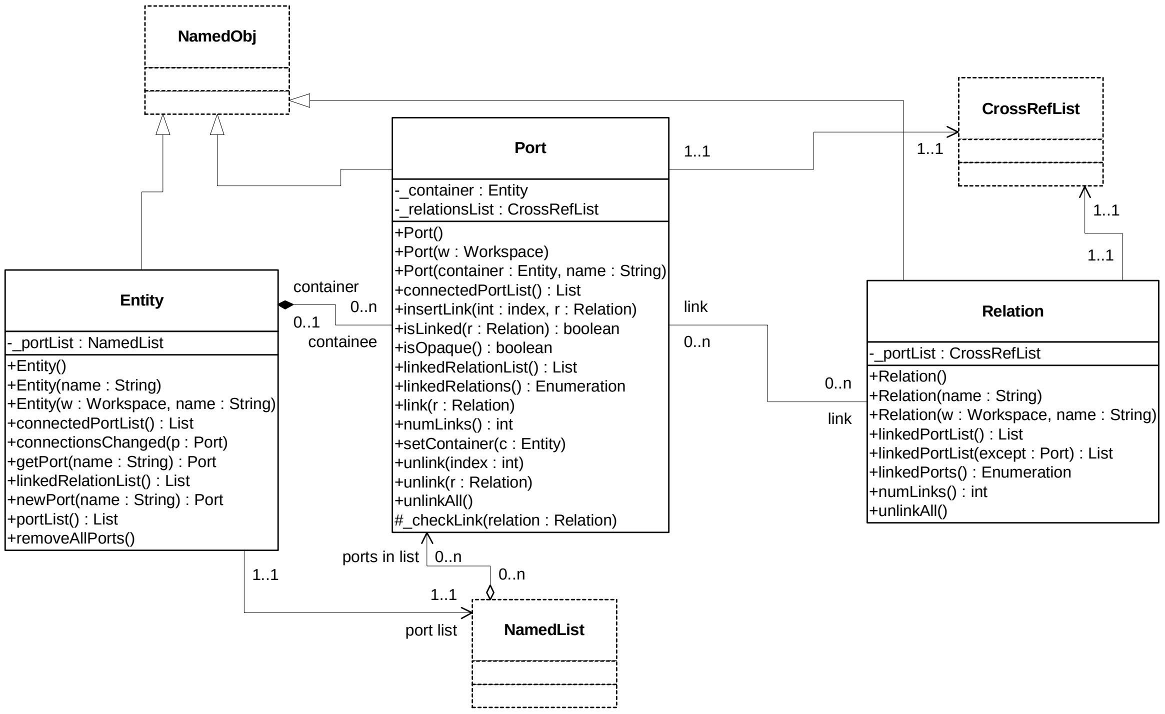

NamedObj
CrossRefList
Port
-_container : Entity
-_relationsList : CrossRefList
+Port()
+Port(w : Workspace)
+Port(container : Entity, name : String)
+connectedPortList() : List
+insertLink(int : index, r : Relation)
+isLinked(r : Relation) : boolean
+isOpaque() : boolean
+linkedRelationList() : List
+linkedRelations() : Enumeration
+link(r : Relation)
+numLinks() : int
+setContainer(c : Entity)
+unlink(index : int)
+unlink(r : Relation)
+unlinkAll()
#_checkLink(relation : Relation)
Entity
-_portList : NamedList
+Entity()
+Entity(name : String)
+Entity(w : Workspace, name : String)
+connectedPortList() : List
+connectionsChanged(p : Port)
+getPort(name : String) : Port
+linkedRelationList() : List
+newPort(name : String) : Port
+portList() : List
+removeAllPorts()
Relation
-_portList : CrossRefList
+Relation()
+Relation(name : String)
+Relation(w : Workspace, name : String)
+linkedPortList() : List
+linkedPortList(except : Port) : List
+linkedPorts() : Enumeration
+numLinks() : int
+unlinkAll()
NamedList
container
0..n
0..1
containee
link
0..n
0..n
link
1..1
1..1
1..1
1..1
ports in list
0..n
0..n
1..1
1..1
port list

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

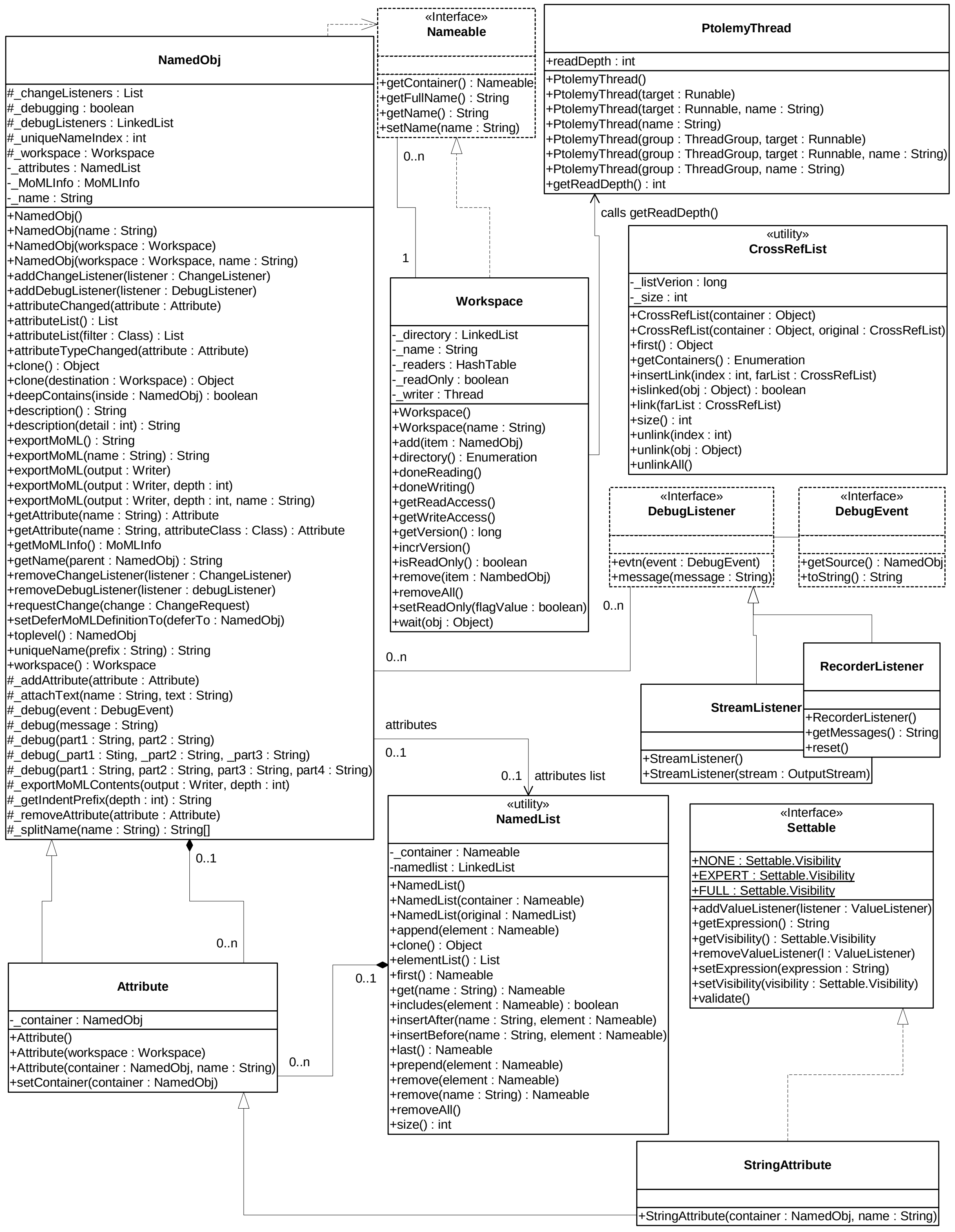

NamedObj
#_changeListeners : List
#_debugging : boolean
#_debugListeners : LinkedList
#_uniqueNameIndex : int
#_workspace : Workspace
-_attributes : NamedList
-_MoMLInfo : MoMLInfo
-_name : String
+NamedObj()
+NamedObj(name : String)
+NamedObj(workspace : Workspace)
+NamedObj(workspace : Workspace, name : String)
+addChangeListener(listener : ChangeListener)
+addDebugListener(listener : DebugListener)
+attributeChanged(attribute : Attribute)
+attributeList() : List
+attributeList(filter : Class) : List
+attributeTypeChanged(attribute : Attribute)
+clone() : Object
+clone(destination : Workspace) : Object
+deepContains(inside : NamedObj) : boolean
+description() : String
+description(detail : int) : String
+exportMoML() : String
+exportMoML(name : String) : String
+exportMoML(output : Writer)
+exportMoML(output : Writer, depth : int)
+exportMoML(output : Writer, depth : int, name : String)
+getAttribute(name : String) : Attribute
+getAttribute(name : String, attributeClass : Class) : Attribute
+getMoMLInfo() : MoMLInfo
+getName(parent : NamedObj) : String
+removeChangeListener(listener : ChangeListener)
+removeDebugListener(listener : debugListener)
+requestChange(change : ChangeRequest)
+setDeferMoMLDefinitionTo(deferTo : NamedObj)
+toplevel() : NamedObj
+uniqueName(prefix : String) : String
+workspace() : Workspace
#_addAttribute(attribute : Attribute)
#_attachText(name : String, text : String)
#_debug(event : DebugEvent)
#_debug(message : String)
#_debug(part1 : String, part2 : String)
#_debug(_part1 : Sting, _part2 : String, _part3 : String)
#_debug(part1 : String, part2 : String, part3 : String, part4 : String)
#_exportMoMLContents(output : Writer, depth : int)
#_getIndentPrefix(depth : int) : String
#_removeAttribute(attribute : Attribute)
#_splitName(name : String) : String[]
«Interface»
Nameable
+getContainer() : Nameable
+getFullName() : String
+getName() : String
+setName(name : String)
0..n
1
PtolemyThread
+readDepth : int
+PtolemyThread()
+PtolemyThread(target : Runable)
+PtolemyThread(target : Runnable, name : String)
+PtolemyThread(name : String)
+PtolemyThread(group : ThreadGroup, target : Runnable)
+PtolemyThread(group : ThreadGroup, target : Runnable, name : String)
+PtolemyThread(group : ThreadGroup, name : String)
+getReadDepth() : int
calls getReadDepth()
«utility»
CrossRefList
-_listVerion : long
-_size : int
+CrossRefList(container : Object)
+CrossRefList(container : Object, original : CrossRefList)
+first() : Object
+getContainers() : Enumeration
+insertLink(index : int, farList : CrossRefList)
+islinked(obj : Object) : boolean
+link(farList : CrossRefList)
+size() : int
+unlink(index : int)
+unlink(obj : Object)
+unlinkAll()
Workspace
-_directory : LinkedList
-_name : String
-_readers : HashTable
-_readOnly : boolean
-_writer : Thread
+Workspace()
+Workspace(name : String)
+add(item : NamedObj)
+directory() : Enumeration
+doneReading()
+doneWriting()
+getReadAccess()
+getWriteAccess()
+getVersion() : long
+incrVersion()
+isReadOnly() : boolean
+remove(item : NambedObj)
+removeAll()
+setReadOnly(flagValue : boolean)
+wait(obj : Object)
«Interface»
DebugListener
+evtn(event : DebugEvent)
+message(message : String)
«Interface»
DebugEvent
+getSource() : NamedObj
+toString() : String
0..n
0..n
RecorderListener
+RecorderListener()
+getMessages() : String
+reset()
StreamListener
+StreamListener()
+StreamListener(stream : OutputStream)
attributes
0..1
0..1
attributes list
«utility»
NamedList
-_container : Nameable
-namedlist : LinkedList
+NamedList()
+NamedList(container : Nameable)
+NamedList(original : NamedList)
+append(element : Nameable)
+clone() : Object
+elementList() : List
+first() : Nameable
+get(name : String) : Nameable
+includes(element : Nameable) : boolean
+insertAfter(name : String, element : Nameable)
+insertBefore(name : String, element : Nameable)
+last() : Nameable
+prepend(element : Nameable)
+remove(element : Nameable)
+remove(name : String) : Nameable
+removeAll()
+size() : int
«Interface»
Settable
+NONE : Settable.Visibility
+EXPERT : Settable.Visibility
+FULL : Settable.Visibility
+addValueListener(listener : ValueListener)
+getExpression() : String
+getVisibility() : Settable.Visibility
+removeValueListener(l : ValueListener)
+setExpression(expression : String)
+setVisibility(visibility : Settable.Visibility)
+validate()
0..1
0..n
0..1
Attribute
-_container : NamedObj
+Attribute()
+Attribute(workspace : Workspace)
+Attribute(container : NamedObj, name : String)
+setContainer(container : NamedObj)
0..n
StringAttribute
+StringAttribute(container : NamedObj, name : String)

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

then adding it to the new container's port list. A port is removed from an entity by calling setContainer() with a null argument.

A change in a containment association involves several distinct objects, and therefore must be atomic, in the sense that other threads must not be allowed to intervene and modify or access relevant attributes halfway through the process. This is ensured by synchronization on the workspace, as explained below in section 2.6. Moreover, if an exception is thrown at any point during the process of changing a containment association, any changes that have been made must be undone so that a consistent state is restored.

2.3 Support Classes

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

2.3.1 Containers

Although these classes do not provide support for constructing clustered graphs, they provide rudimentary support for *container* associations. An instance of these classes can have at most one container. That container is viewed as the owner of the object, and "managed ownership" [47] is used as a central tool in thread safety, as explained in section 2.6 below.

In the base classes shown in figure 2.2, only an instance of Port can have a non-null container. It is the only class with a setContainer() method. Instances of all other classes have no container, and their getContainer() method will return null. In the classes of figure 2.3, only Attribute has a setContainer() method.

Every object is associated with exactly one instance of Workspace, as shown in figure 2.3, but the workspace is not viewed as a container. The workspace is defined when an object is constructed, and no methods are provided to change it. It is said to be *immutable*, a critical property in its use for thread safety.

2.3.2 Name and Full Name

The Nameable interface supports hierarchy in the naming so that individual named objects in a hierarchy can be uniquely identified. By convention, the *full name* of an object is a concatenation of the full name of its container, if there is one, 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 2.3.4.

Names of objects are only required to be unique within a container. Thus, even the full name is not assured of being globally unique.

Here, names are a property of the instances themselves, rather than properties of an association between entities. As argued by Rumbaugh in [84], this is not always the right choice. Often, a name is more properly viewed as a property of an association. For example, a file name is a property of the association between a directory and a file. A file may have multiple names (through the use of symbolic links). Our design takes a stronger position on names, and views them as properties of the object, much as we view the name of a person as a property of the person (vs. their employee number, for example, which is a property of their association with an employer).

2.3.3 Workspace

Workspace is a concrete class that implements the Nameable interface, as shown in figure 2.3. All objects in a topology are associated with a workspace, and almost all operations that involve multiple objects are only supported for objects in the same workspace. This constraint is exploited to ensure thread safety, as explained in section 2.6 below.

2.3.4 Attributes

In almost all applications of Ptolemy II, entities, ports, and relations need to be parameterized. The base classes shown in figure 2.3 provide for these objects to have any number of instances of the Attribute class attached to them. Attribute is a NamedObj that can be contained by another NamedObj, and serves as a base class for parameters.

Attributes are added to a NamedObj by calling their setContainer() method and passing it a reference to the container. They are removed by calling setContainer() with a null argument. The NamedObj class provides the getAttribute() method, which takes an attribute name as an argument and returns the attribute, and the attributeList() method, which returns a list of the attributes contained by the object.

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 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.

An attribute that is not an instance of Settable 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 2.1:Names of special attributes

name	class	use
_doc	ptolemy.actor.gui.Documentation	Default documentation attribute name.
_generator	ptolemy.codegen.saveasjava.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.

Table 2.1:Names of special attributes

name	class	use
_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.

2.3.5 List Classes

Figure 2.3 shows two list classes that are used extensively in Ptolemy II. NamedList implements an ordered list of objects with the Nameable interface. It is unlike a hash table in that it maintains an ordering of the entries that is independent of their names. It is unlike a vector or a linked list in that it supports accesses by name. It is used in figure 2.3 to maintain a list of attributes, and in figure 2.2 to maintain the list of ports contained by an entity.

The class CrossRefList is a bit more interesting. It mediates bidirectional links between objects that contain CrossRefLists, in this case, ports and relations. It provides a simple and efficient mechanism for constructing a web of objects, where each object maintains a list of the objects it is linked to. That list is an instance of CrossRefList. The class ensures consistency. That is, if one object in the web is linked to another, then the other is linked back to the one. CrossRefList also handles efficient modification of the cross references. In particular, if a link is removed from the list maintained by one object, the back reference in the remote object also has to be deleted. This is done in O(1) time. A more brute force solution would require searching the remote list for the back reference, increasing the time required and making it proportional to the number of links maintained by each object.

2.4 Clustered Graphs

The classes shown in figure 2.2 provide only partial support for hierarchy, through the concept of a container. Subclasses, shown in figure 2.4, extend these with more complete support for hierarchy. ComponentEntity, ComponentPort, and ComponentRelation are used whenever a clustered graph is used. All ports of a ComponentEntity are required to be instances of ComponentPort. CompositeEntity extends ComponentEntity with the capability of containing ComponentEntity and ComponentRelation objects. Thus, it contains a subgraph. The association between ComponentEntity and CompositeEntity is the classic Composite design pattern [28].

2.4.1 Abstraction

Composite entities are non-atomic (isAtomic() return false). They can contain a graph (entities and relations). By default, a CompositeEntity is transparent (isOpaque() returns false). Conceptually, this means that its contents are visible from the outside. The hierarchy can be ignored (flattened) by algorithms operating on the topology. Some subclasses of CompositeEntity are opaque (see the Actor 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

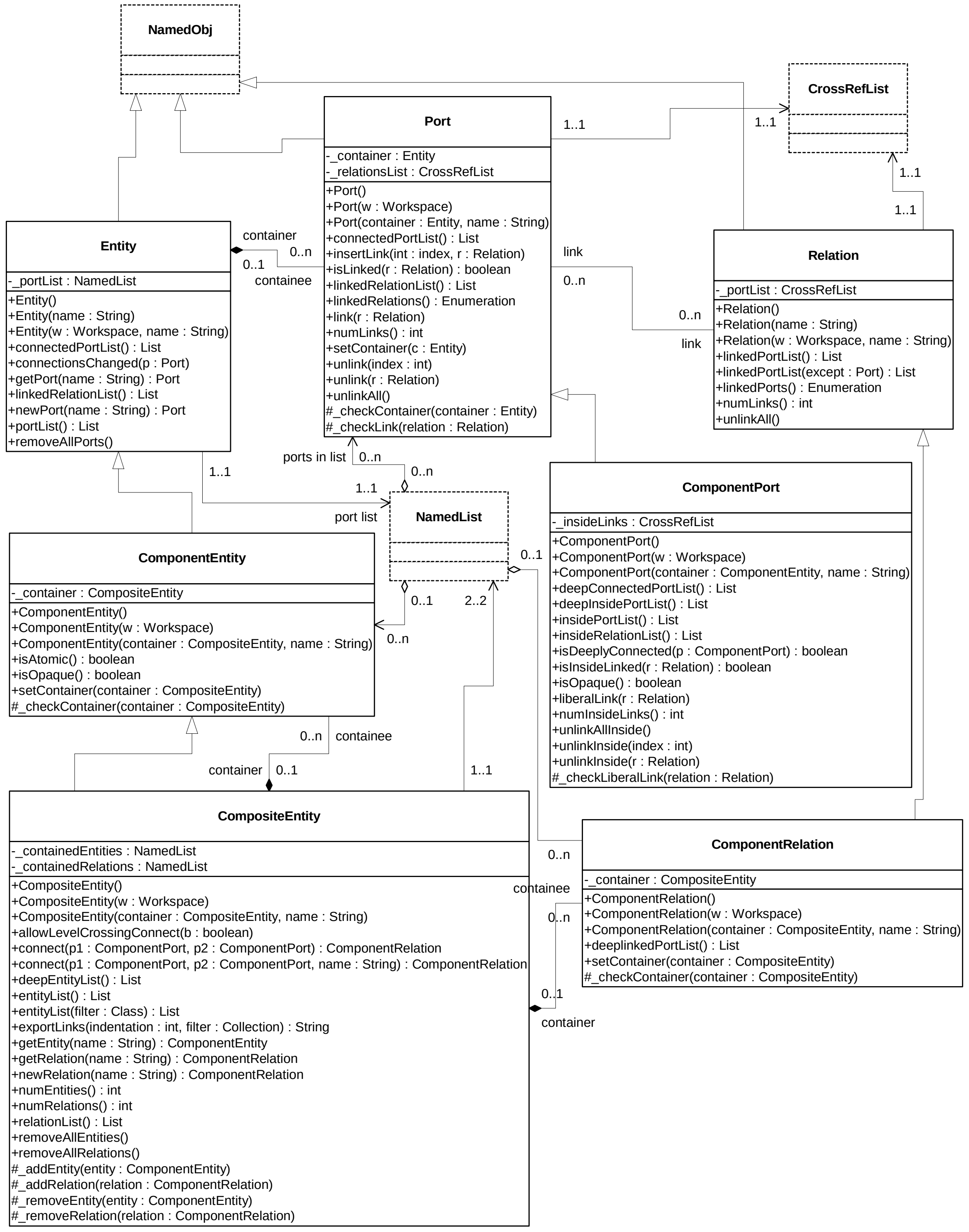

NamedObj
CrossRefList
Port
-_container : Entity
-_relationsList : CrossRefList
+Port()
+Port(w : Workspace)
+Port(container : Entity, name : String)
+connectedPortList() : List
+insertLink(int : index, r : Relation)
+isLinked(r : Relation) : boolean
+linkedRelationList() : List
+linkedRelations() : Enumeration
+link(r : Relation)
+numLinks() : int
+setContainer(c : Entity)
+unlink(index : int)
+unlink(r : Relation)
+unlinkAll()
#_checkContainer(container : Entity)
#_checkLink(relation : Relation)
1..1
1..1
1..1
1..1
Entity
-_portList : NamedList
+Entity()
+Entity(name : String)
+Entity(w : Workspace, name : String)
+connectedPortList() : List
+connectionsChanged(p : Port)
+getPort(name : String) : Port
+linkedRelationList() : List
+newPort(name : String) : Port
+portList() : List
+removeAllPorts()
container
0..n
0..1
containee
link
0..n
0..n
link
Relation
-_portList : CrossRefList
+Relation()
+Relation(name : String)
+Relation(w : Workspace, name : String)
+linkedPortList() : List
+linkedPortList(except : Port) : List
+linkedPorts() : Enumeration
+numLinks() : int
+unlinkAll()
ports in list
0..n
0..n
1..1
1..1
port list
NamedList
0..1
0..1
2..2
0..n
ComponentPort
-_insideLinks : CrossRefList
+ComponentPort()
+ComponentPort(w : Workspace)
+ComponentPort(container : ComponentEntity, name : String)
+deepConnectedPortList() : List
+deepInsidePortList() : List
+insidePortList() : List
+insideRelationList() : List
+isDeeplyConnected(p : ComponentPort) : boolean
+isInsideLinked(r : Relation) : boolean
+isOpaque() : boolean
+liberalLink(r : Relation)
+numInsideLinks() : int
+unlinkAllInside()
+unlinkInside(index : int)
+unlinkInside(r : Relation)
#_checkLiberalLink(relation : Relation)
ComponentEntity
-_container : CompositeEntity
+ComponentEntity()
+ComponentEntity(w : Workspace)
+ComponentEntity(container : CompositeEntity, name : String)
+isAtomic() : boolean
+isOpaque() : boolean
+setContainer(container : CompositeEntity)
#_checkContainer(container : CompositeEntity)
0..n
containee
container
0..1
1..1
CompositeEntity
-_containedEntities : NamedList
-_containedRelations : NamedList
+CompositeEntity()
+CompositeEntity(w : Workspace)
+CompositeEntity(container : CompositeEntity, name : String)
+allowLevelCrossingConnect(b : boolean)
+connect(p1 : ComponentPort, p2 : ComponentPort) : ComponentRelation
+connect(p1 : ComponentPort, p2 : ComponentPort, name : String) : ComponentRelation
+deepEntityList() : List
+entityList() : List
+entityList(filter : Class) : List
+exportLinks(indentation : int, filter : Collection) : String
+getEntity(name : String) : ComponentEntity
+getRelation(name : String) : ComponentRelation
+newRelation(name : String) : ComponentRelation
+numEntities() : int
+numRelations() : int
+relationList() : List
+removeAllEntities()
+removeAllRelations()
#_addEntity(entity : ComponentEntity)
#_addRelation(relation : ComponentRelation)
#_removeEntity(entity : ComponentEntity)
#_removeRelation(relation : ComponentRelation)
0..n
containee
0..n
0..1
container
ComponentRelation
-_container : CompositeEntity
+ComponentRelation()
+ComponentRelation(w : Workspace)
+ComponentRelation(container : CompositeEntity, name : String)
+deeplinkedPortList() : List
+setContainer(container : CompositeEntity)
#_checkContainer(container : CompositeEntity)

FIGURE 2.4. Key classes supporting clustered graphs.

ports in the contained entities as ports of the composite. This is the converse of the "hiding" operator often found in process algebras [64]. Ports within an entity are hidden by default, and must be explicitly exposed to be visible (linkable) from outside the entity[1]. The composite entity with ports thus provides an abstraction of the contents of the composite.

A port of a composite entity may be opaque or transparent. It is defined to be *opaque* if its container is opaque. Conceptually, if it is opaque, then its inside links are not visible from the outside, and the outside links are not visible from the inside. If it is opaque, it appears from the outside to be indistinguishable from a port of an atomic entity.

The transparent port mechanism is illustrated by the example in figure 2.5[2]. Some of the ports in figure 2.5 are filled in white rather than black. These ports are said to be *transparent*. Transparent ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

ComponentPort, ComponentRelation, and CompositeEntity have a set of methods with the prefix "deep," as shown in figure 2.4. These methods flatten the hierarchy by traversing it. Thus, for example, the ports that are "deeply" connected to port P1 in figure 2.5 are P2, P5, and P6. No transparent port is included, so note that P3 is not included.

Deep traversals of a graph follow a simple rule. If a transparent port is encountered from inside, then the traversal continues with its outside links. If it is encountered from outside, then the traversal continues with its inside links. Thus, for example, the ports deeply connected to P5 are P1 and P2. Note that P6 is not included. Similarly, the 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 2.2, the Workspace class includes a getVersion() and incrVer-

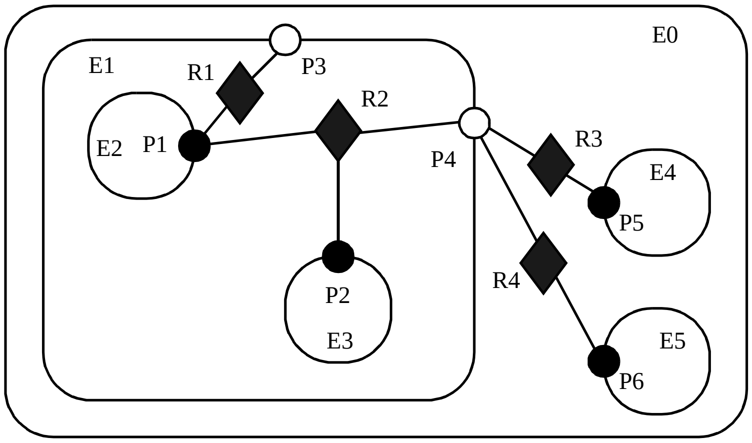


FIGURE 2.5. Transparent ports (P3 and P4) are linked to relations (R1 and R2) below their container (E1) in the hierarchy. They may also be linked to relations at the same level (R3 and R4).

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

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

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

2.4.2 Level-Crossing Connections

For a few applications, such as Statecharts [34], level-crossing links and connections are needed. The example shown in figure 2.6 has three level-crossing connections that are slightly different from one another. The links in these connections are created using the liberalLink() method of ComponentPort. The link() method prohibits such links, throwing an exception if they are attempted (most applications will prohibit level-crossing connections by using only the link() method).

An alternative that may be more convenient for a user interface is to use the connect() methods of CompositeEntity rather than the link() or liberalLink() method of ComponentPort. To allow level-crossing links using connect(), first call allowLevelCrossingConnect() with a *true* argument.

The simplest level-crossing connection in figure 2.6 is at the bottom, connecting P2 to P7 via the relation R5. The relation is contained by E1, but the connection would be essentially identical if it were contained by any other entity. Thus, the notion of composite entities containing relations is somewhat

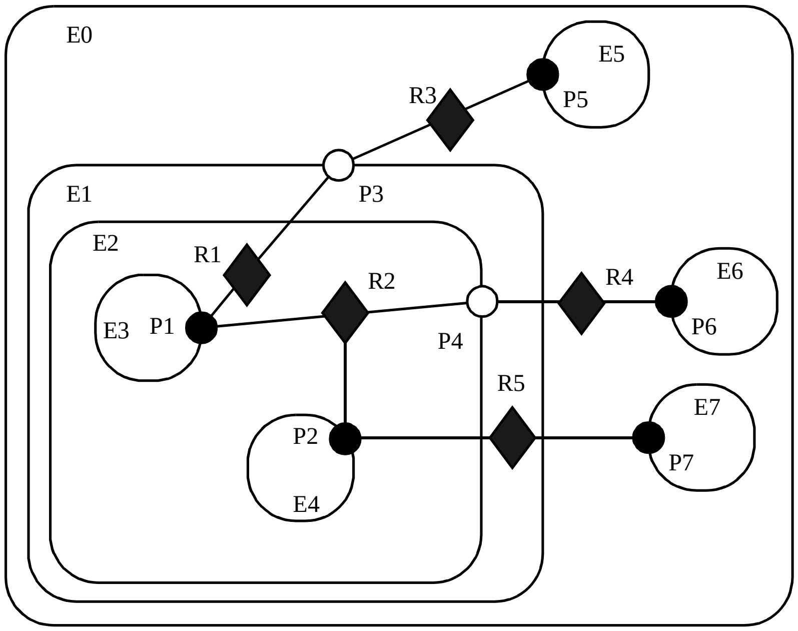


FIGURE 2.6. An example with level-crossing transitions.

weaker when level-crossing connections are allowed.

The other two level-crossing connections in figure 2.6 are mediated by transparent ports. This sort of hybrid could come about in heterogeneous representations, where level-crossing connections are permitted in some parts but not in others. It is important, therefore, for the classes to support such hybrids.

To support such hybrids, we have to modify slightly the algorithm by which a port recognizes an inside link. Given a relation and a port, the link is an inside link if the relation is contained by an entity that is either the same as or is deeply contained (i.e. directly or indirectly contained) by the entity that contains the port. The deepContains() method of NamedObj supports this test.

2.4.3 Tunneling Entities

The transparent port mechanism we have described supports connections like that between P1 and P5 in figure 2.7. That connection passes through the entity E2. The relation R2 is linked to the inside of each of P2 and P4, in addition to its link to the outside of P3. Thus, the ports deeply connected to P1 are P3 and P5, and those deeply connected to P3 are P1 and P5, and those deeply connected to P5 are P1 and P3.

A *tunneling entity* is one that contains a relation with links to the inside of more than one port. It may of course also contain more standard links, but the term “tunneling” suggests that at least some deep graph traversals will see right through it.

Support for tunneling entities is a major increment in capability over the previous Ptolemy kernel [14] (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 [54] was made much more complicated. These higher-order functions had to be careful to avoid mutations that created tunneling.

2.4.4 Cloning

The kernel classes are all capable of being *cloned*, with some restrictions. Cloning means that an identical but entirely independent object is created. Thus, if the object being cloned contains other objects, then those objects are also cloned. If those objects are linked, then the links are replicated in

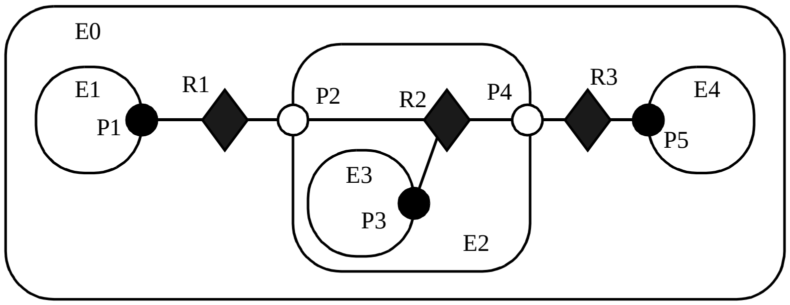


FIGURE 2.7. A tunneling entity contains a relation with inside links to more than one port.

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.

2.4.5 An Elaborate Example

An elaborate example of a clustered graph is shown in figure 2.8. This example includes instances of all the capabilities we have discussed. The top-level entity is named “E0.” All other entities in this example have containers. A Java class that implements this example is shown in figure 2.9. A script in the Tcl language [73] that constructs the same graph is shown in figure 2.10. This script uses Tcl Blend, an interface between Tcl and Java that is distributed by Scriptics.

The order in which links are constructed matters, in the sense that methods that return lists of objects preserve this order. The order implemented in both figures 2.9 and 2.10 is top-to-bottom and left-to-right in figure 2.8. A graphical syntax, however, does not generally have a particularly convenient way to completely control this order.

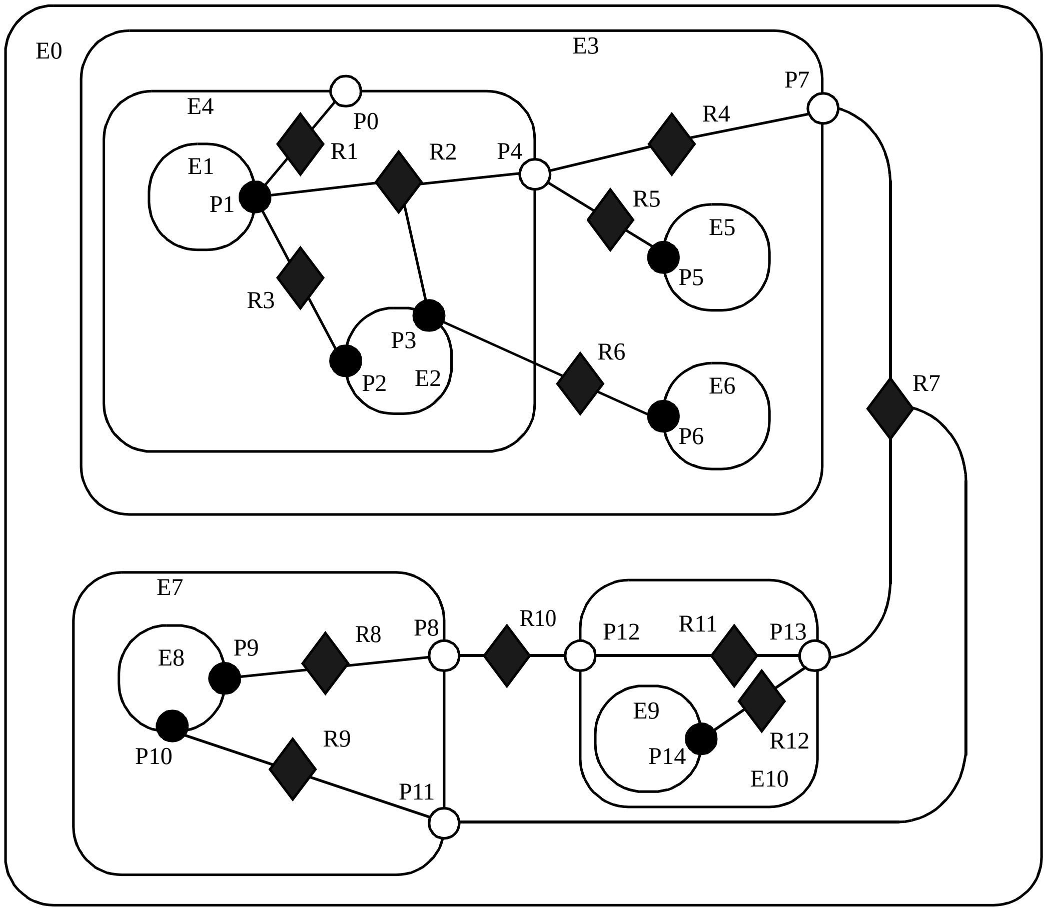


FIGURE 2.8. An example of a clustered graph.

```
public class ExampleSystem {
    private CompositeEntity e0, e3, e4, e7, e10;
    private ComponentEntity e1, e2, e5, e6, e8, e9;
    private ComponentPort p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p4;
    private ComponentRelation r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12;

    public ExampleSystem() throws IllegalActionException, NameDuplicationException {
        e0 = new CompositeEntity();
        e0.setName("E0");
        e3 = new CompositeEntity(e0, "E3");
        e4 = new CompositeEntity(e3, "E4");
        e7 = new CompositeEntity(e0, "E7");
        e10 = new CompositeEntity(e0, "E10");

        e1 = new ComponentEntity(e4, "E1");
        e2 = new ComponentEntity(e4, "E2");
        e5 = new ComponentEntity(e3, "E5");
        e6 = new ComponentEntity(e3, "E6");
        e8 = new ComponentEntity(e7, "E8");
        e9 = new ComponentEntity(e10, "E9");

        p0 = (ComponentPort) e4.newPort("P0");
        p1 = (ComponentPort) e1.newPort("P1");
        p2 = (ComponentPort) e2.newPort("P2");
        p3 = (ComponentPort) e2.newPort("P3");
        p4 = (ComponentPort) e4.newPort("P4");
        p5 = (ComponentPort) e5.newPort("P5");
        p6 = (ComponentPort) e5.newPort("P6");
        p7 = (ComponentPort) e3.newPort("P7");
        p8 = (ComponentPort) e7.newPort("P8");
        p9 = (ComponentPort) e8.newPort("P9");
        p10 = (ComponentPort) e8.newPort("P10");
        p11 = (ComponentPort) e7.newPort("P11");
        p12 = (ComponentPort) e10.newPort("P12");
        p13 = (ComponentPort) e10.newPort("P13");
        p14 = (ComponentPort) e9.newPort("P14");

        r1 = e4.connect(p1, p0, "R1");
        r2 = e4.connect(p1, p4, "R2");
        p3.link(r2);
        r3 = e4.connect(p1, p2, "R3");
        r4 = e3.connect(p4, p7, "R4");
        r5 = e3.connect(p4, p5, "R5");
        e3.allowLevelCrossingConnect(true);
        r6 = e3.connect(p3, p6, "R6");
        r7 = e0.connect(p7, p13, "R7");
        r8 = e7.connect(p9, p8, "R8");
        r9 = e7.connect(p10, p11, "R9");
        r10 = e0.connect(p8, p12, "R10");
        r11 = e10.connect(p12, p13, "R11");
        r12 = e10.connect(p14, p13, "R12");
        p11.link(r7);
    }
    ...
}
```

FIGURE 2.9. The same topology as in figure 2.8 implemented as a Java class.

The results of various method accesses on the graph are shown in figure 2.11. This table can be studied to better understand the precise meaning of each of the methods.

2.5 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.

```
# Create composite entities
   set e0 [java::new pt.kernel.CompositeEntity E0]
   set e3 [java::new pt.kernel.CompositeEntity $e0 E3]
   set e4 [java::new pt.kernel.CompositeEntity $e3 E4]
   set e7 [java::new pt.kernel.CompositeEntity $e0 E7]
   set e10 [java::new pt.kernel.CompositeEntity $e0 E10]

   # Create component entities.
   set e1 [java::new pt.kernel.ComponentEntity $e4 E1]
   set e2 [java::new pt.kernel.ComponentEntity $e4 E2]
   set e5 [java::new pt.kernel.ComponentEntity $e3 E5]
   set e6 [java::new pt.kernel.ComponentEntity $e3 E6]
   set e8 [java::new pt.kernel.ComponentEntity $e7 E8]
   set e9 [java::new pt.kernel.ComponentEntity $e10 E9]

   # Create ports.
   set p0 [$e4 newPort P0]
   set p1 [$e1 newPort P1]
   set p2 [$e2 newPort P2]
   set p3 [$e2 newPort P3]
   set p4 [$e4 newPort P4]
   set p5 [$e5 newPort P5]
   set p6 [$e6 newPort P6]
   set p7 [$e3 newPort P7]
   set p8 [$e7 newPort P8]
   set p9 [$e8 newPort P9]
   set p10 [$e8 newPort P10]
   set p11 [$e7 newPort P11]
   set p12 [$e10 newPort P12]
   set p13 [$e10 newPort P13]
   set p14 [$e9 newPort P14]

   # Create links
   set r1 [$e4 connect $p1 $p0 R1]
   set r2 [$e4 connect $p1 $p4 R2]
   $p3 link $r2
   set r3 [$e4 connect $p1 $p2 R3]
   set r4 [$e3 connect $p4 $p7 R4]
   set r5 [$e3 connect $p4 $p5 R5]
   $e3 allowLevelCrossingConnect true
   set r6 [$e3 connect $p3 $p6 R6]
   set r7 [$e0 connect $p7 $p13 R7]
   set r8 [$e7 connect $p9 $p8 R8]
   set r9 [$e7 connect $p10 $p11 R9]
   set r10 [$e0 connect $p8 $p12 R10]
   set r11 [$e10 connect $p12 $p13 R11]
   set r12 [$e10 connect $p14 $p13 R12]
   $p11 link $r7
```

FIGURE 2.10. The same topology as in figure 2.8 described by the Tcl Blend commands to create it.

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*.

2.6 Concurrency

We expect concurrency. Topologies often represent the structure of computations. Those computations themselves may be concurrent, and a user interface may be interacting with the topologies while they execute their computation. Moreover, Ptolemy II objects may interact with other objects concurrently over the network via RMI or CORBA.

Both computations within an entity and the user interface are capable of modifying the topology. Thus, extra care is needed to make sure that the topology remains consistent in the face of simultaneous modifications (we defined consistency in section 2.2.2).

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

Table 2.1:Methods of ComponentRelation

Method Name	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
getLinkedPorts	P1 P0	P1 P4 P3	P1 P2	P4 P7	P4 P5	P3 P6	P7 P13 P11	P9 P8	P10 P11	P8 P12	P12 P13	P14 P13
deepGetLinkedPorts	P1	P1 P9 P14 P10 P5 P3	P1 P2	P1 P3 P9 P14 P10	P1 P3 P5	P3 P6	P1 P3 P9 P14 P10	P9 P1 P3 P10	P10 P1 P3 P9 P14	P9 P1 P3 P10	P9 P1 P3 P10	P14 P1 P3 P10

Table 2.2:Methods of ComponentPort

Method Name	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
getConnectedPorts		P0 P4 P3 P2	P1	P1 P4 P6	P7 P5	P4	P3	P13 P11	P12	P8	P11	P7 P13	P8	P7 P11	P13
deepGetConnectedPorts		P9 P14 P10 P5 P3 P2	P1	P1 P9 P14 P10 P5 P6	P9 P14 P10 P5	P1 P3	P3	P9 P14 P10	P1 P3 P10	P1 P3 P10	P1 P3 P9 P14	P1 P3 P9 P14	P9	P1 P3 P10	P1 P3 P10

FIGURE 2.11. Key methods applied to figure 2.8.

occurring. Such enforced consistency is called *thread safety.*

2.6.1 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 2.12. It indicates that an exclusive lock should be obtained on a specific object before executing the body of code. If the keyword annotates a method, as in figure 2.12(a), then the method’s object is locked (an instance of class A in the figure). The keyword can also be associated with an arbitrary body of code and can acquire a lock on an arbitrary object. In figure 2.12(b), the code body represented by ellipses (...) can be executed only after a lock has been acquired on object *obj*.

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

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

(a)

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

(b)

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

(c)

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

(d)

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

(e)

FIGURE 2.12. Using monitors for thread safety. The method used in Ptolemy II is in (d) and (e).

One possible solution is to ensure that locks are always acquired in the same order [47]. 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.

2.6.2 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 only on instances of the class Workspace. As shown in figure 2.3, every instance of NamedObj (or derived classes) is associated with a single instance of Workspace. Each body of code that alters or depends on the topology must acquire a lock on its workspace. Moreover, the workspace associated with an object is immutable. It is set in the constructor and never modified. This is enforced by a very simple mechanism: a reference to the workspace is stored in a private variable of the base class NamedObj, as shown in figure 2.3, and no methods are provided to modify it. Moreover, in instances of these kernel classes, a container and its containees must share the same workspace (derived classes may be more liberal in certain circumstances). This “managed ownership” [47] is our central strategy in thread safety.

As shown in figure 2.12(c), a conservative approach would be to acquire a monitor on the workspace for each body of code that reads or modified objects in the workspace. However, this approach is too conservative. Instead, Ptolemy II allows any number of readers to simultaneously access a workspace. Only one writer can access the workspace, however, and only if no readers are concurrently accessing the workspace.

The code for readers and writers is shown in figure 2.12(d) and (e). In (d), a reader first calls the getReadAccess() method of the Workspace class. That method does not return until it is safe to read data anywhere in the workspace. It is safe if there is no other thread concurrently holding (or requesting) a write lock on the workspace (the thread calling getReadAccess() may safely hold both a read and a write lock). When the user is finished reading the workspace data, it must call doneReading(). Failure to do so will result in no writer ever again gaining write access to the workspace. Because it is so important to call this method, it is enclosed in the finally clause of a try statement. That clause is executed even if an exception occurs in the body of the try statement.

The code for writers is shown in figure 2.12(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 done-Writing() be called after writing is complete.

This solution, while not as conservative as the single monitor of figure 2.12(c), is still conservative in that mutual exclusion is applied even on write actions that are independent of one another if they share the same workspace. This effectively serializes some modifications that might otherwise occur in parallel. However, there is no constraint in Ptolemy II on the number of workspaces used, so subclasses of these kernel classes could judiciously use additional workspaces to increase the parallelism. But they must do so carefully to avoid deadlock. Moreover, most of the methods in the kernel refuse to operate on multiple objects that are not in the same workspace, throwing an exception on any attempt to do so. Thus, derived classes that are more liberal will have to implement their own mechanisms supporting interaction across workspaces.

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

The way to avoid this situation is to use the wait() method of Workspace, passing as an argument the object on which you wish to wait (see Workspace methods in figure 2.3). That method first relinquishes all read permissions before calling wait on the target object. When wait() returns, notice that it is possible that the topology has changed, so callers should be sure to re-read any topology-dependent information. In general, this technique should be used whenever a thread suspends while it holds read permissions.

2.6.3 Making a Workspace Read Only

Acquiring read and write access permissions on the workspace is not free, and it is performed so often in a typical application that it can significantly degrade performance. In some situations, an application may simply wish to prohibit all modifications to the topology for some period of time. This can be done by calling setReadOnly() on the workspace (see Workspace methods in figure 2.3). Once the workspace is read only, requests for read permission are routinely (and very quickly) granted, and requests for write permission trigger an exception. Thus, making a workspace read only can significantly improve performance, at the expense of denying changes to the topology.

2.7 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. During these segments, the workspace can be made read-only (see section 2.6.3), significantly improving performance.

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 2.13. 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 suc-

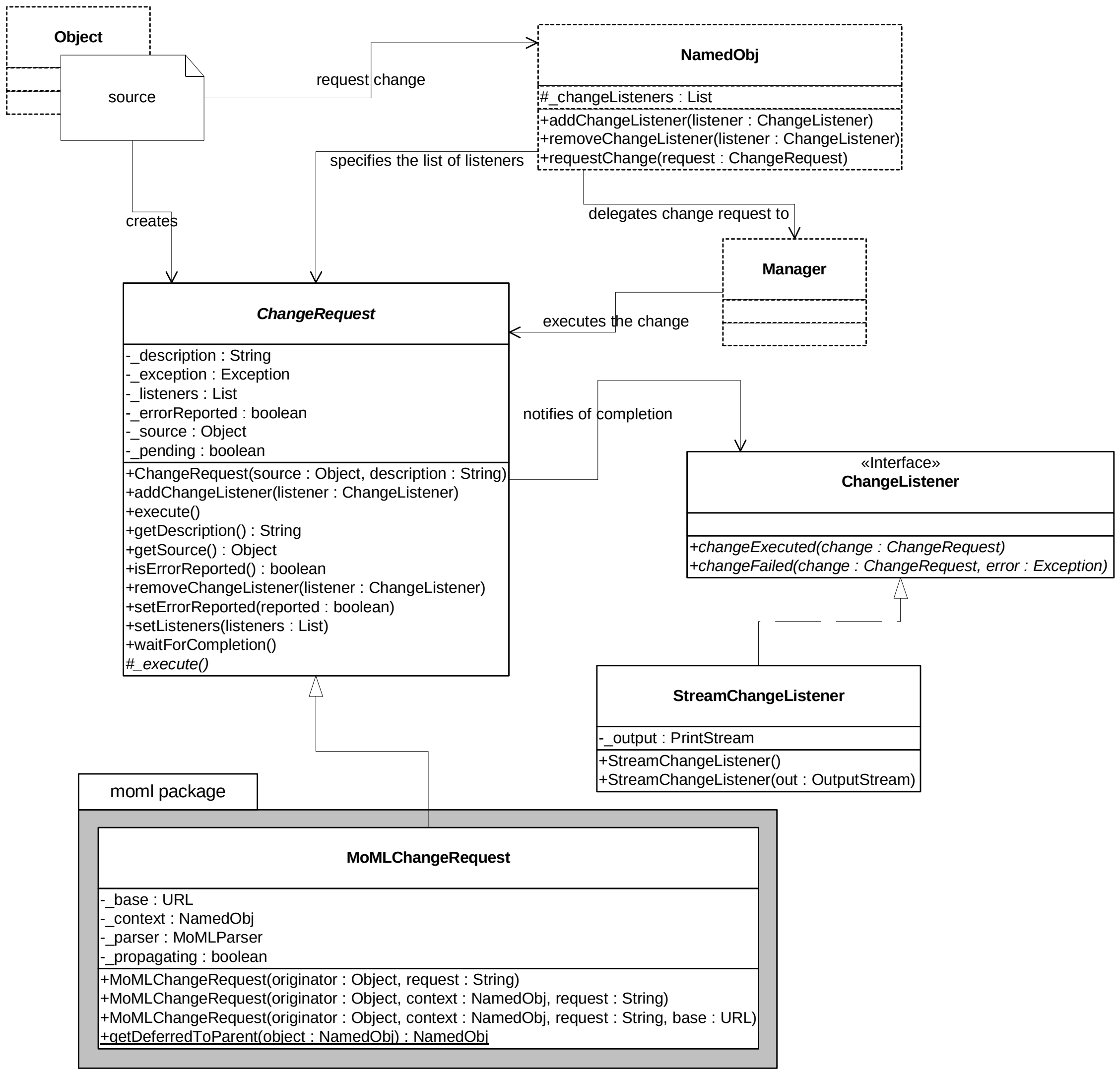

Object
source
request change
NamedObj
#_changeListeners : List
+addChangeListener(listener : ChangeListener)
+removeChangeListener(listener : ChangeListener)
+requestChange(request : ChangeRequest)
specifies the list of listeners
creates
delegates change request to
Manager
ChangeRequest
executes the change
-_description : String
-_exception : Exception
-_listeners : List
-_errorReported : boolean
-_source : Object
-_pending : boolean
+ChangeRequest(source : Object, description : String)
+addChangeListener(listener : ChangeListener)
+execute()
+getDescription() : String
+getSource() : Object
+isErrorReported() : boolean
+removeChangeListener(listener : ChangeListener)
+setErrorReported(reported : boolean)
+setListeners(listeners : List)
+waitForCompletion()
#_execute()
notifies of completion
«Interface»
ChangeListener
+changeExecuted(change : ChangeRequest)
+changeFailed(change : ChangeRequest, error : Exception)
StreamChangeListener
-_output : PrintStream
+StreamChangeListener()
+StreamChangeListener(out : OutputStream)
moml package
MoMLChangeRequest
-_base : URL
-_context : NamedObj
-_parser : MoMLParser
-_propagating : boolean
+MoMLChangeRequest(originator : Object, request : String)
+MoMLChangeRequest(originator : Object, context : NamedObj, request : String)
+MoMLChangeRequest(originator : Object, context : NamedObj, request : String, base : URL)
+getDeferredToParent(object : NamedObj) : NamedObj

FIGURE 2.13. Classes and interfaces that support controlled topology mutations. A source requests topology changes and a manager performs them at a safe time.

ceeds, 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.

2.7.1 Change Requests

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

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

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

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

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

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

A key concrete class extending ChangeRequest is implemented in the moml package, as shown in figure 2.13. 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);
```

2.7.2 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.

2.8 Exceptions

Ptolemy II includes a set of exception classes that provide a uniform mechanism for reporting errors that takes advantage of the identification of named objects by full name. These exception are summarized in the class diagram in figure 2.14.

2.8.1 Base Class

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

2.8.2 Less Severe Exceptions

These exceptions generally indicate that an operation failed to complete. These can result in a topology that is not what the caller expects, since the caller’s modifications to the topology did not succeed. However, they should *never* result in an inconsistent or contradictory topology.

IllegalActionException. Thrown on an attempt to perform an action that is disallowed. For example, the action would result in an inconsistent or contradictory data structure if it were allowed to complete. E.g., attempt to set the container of an object to be another object that cannot contain it because it is of the wrong class.

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

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

2.8.3 More Severe Exceptions

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

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

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

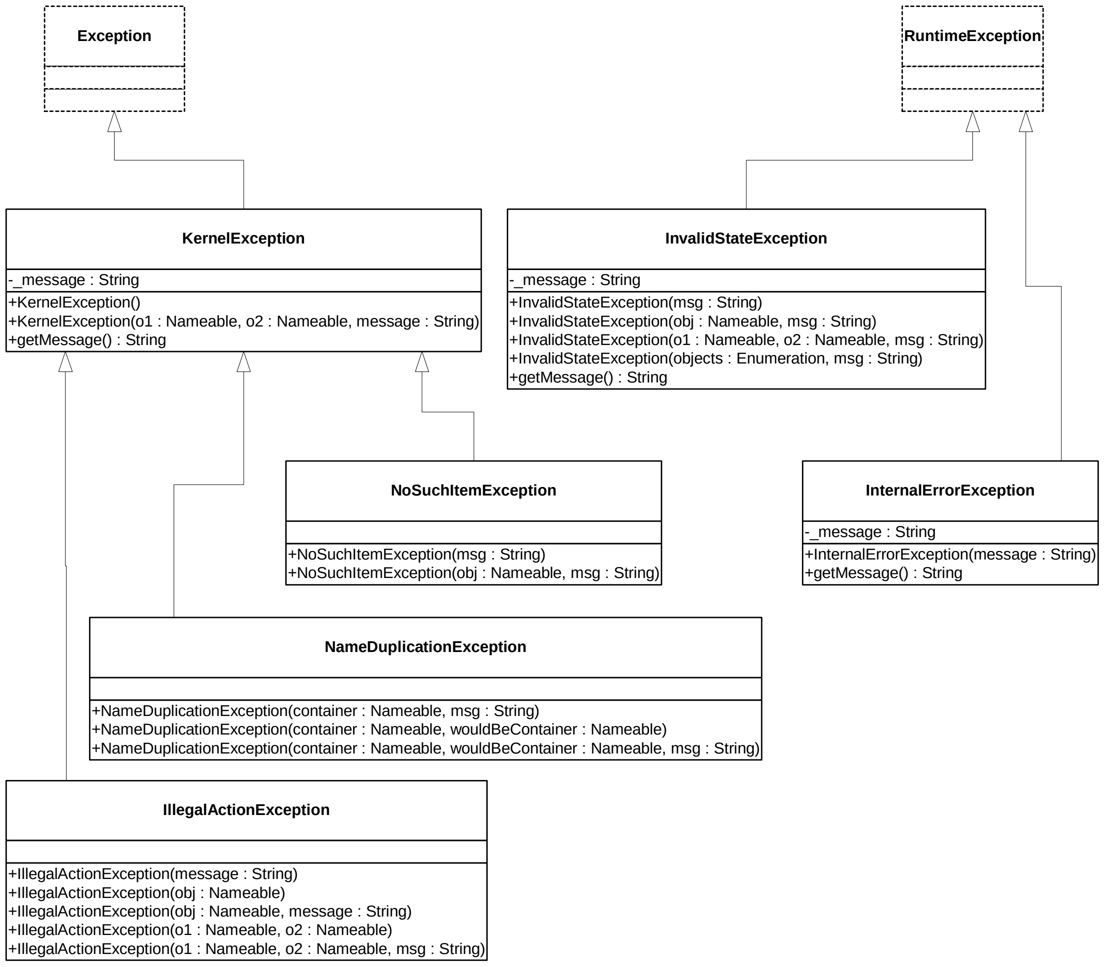

Exception
RuntimeException
KernelException
-_message : String
+KernelException()
+KernelException(o1 : Nameable, o2 : Nameable, message : String)
+getMessage() : String
InvalidStateException
-_message : String
+InvalidStateException(msg : String)
+InvalidStateException(obj : Nameable, msg : String)
+InvalidStateException(o1 : Nameable, o2 : Nameable, msg : String)
+InvalidStateException(objects : Enumeration, msg : String)
+getMessage() : String
NoSuchItemException
+NoSuchItemException(msg : String)
+NoSuchItemException(obj : Nameable, msg : String)
InternalErrorException
-_message : String
+InternalErrorException(message : String)
+getMessage() : String
NameDuplicationException
+NameDuplicationException(container : Nameable, msg : String)
+NameDuplicationException(container : Nameable, wouldBeContainer : Nameable)
+NameDuplicationException(container : Nameable, wouldBeContainer : Nameable, msg : String)
IllegalActionException
+IllegalActionException(message : String)
+IllegalActionException(obj : Nameable)
+IllegalActionException(obj : Nameable, message : String)
+IllegalActionException(o1 : Nameable, o2 : Nameable)
+IllegalActionException(o1 : Nameable, o2 : Nameable, msg : String)

FIGURE 2.14. 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.

exception is derived from the Java RuntimeException.

3

Actor Package

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

3.1 Concurrent Computation

In the kernel package, entities have no semantics. They are syntactic placeholders. In many of the uses of Ptolemy II, entities are executable. The actor package provides basic support for executable entities. It makes a minimal commitment to the semantics of these entities by avoiding specifying the order in which actors execute (or even whether 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.

3.2 Message Passing

The actor package provides templates for executable entities called *actors* that communicate with one another via message passing. Messages are encapsulated in *tokens* (see the Data Package chapter). Messages are sent and received via ports. IOPort is the key class supporting message transport, and is shown in figure 3.2. An IOPort can only be connected to other IOPort instances, and only via IORelations. The IORelation class is also shown in figure 3.2. TypedIOPort and TypedIORelation are subclasses that manage type resolution. 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.

3.2.1 Data Transport

Data transport is depicted in figure 3.1. The originating actor E1 has an output port P1, indicated in the figure with an arrow in the direction of token flow. The destination actor E2 has an input port P2, indicated in the figure with another arrow. E1 calls the send() method of P1 to send a token *t* to a remote actor. The port obtains a reference to a remote receiver (via the IORelation) and calls the put() method of the receiver, passing it the token. The destination actor retrieves the token by calling the get() method of its input port, which in turn calls the get() method of the designated receiver.

Domains typically provide specialized receivers. These receivers override get() and put() to implement the communication protocol pertinent to that domain. A domain that uses asynchronous message passing, for example, can usually use the QueueReceiver shown in figure 3.2. A domain that uses synchronous message passing (rendezvous) has to provide a new receiver class.

In figure 3.1 there is only a single channel, indexed 0. The “0” argument of the send() and get() methods refer to this channel. A port can support more than one channel, however, as shown in figure 3.3. This can be represented by linking more than one relation to the port, or by linking a relation that has a width greater than one. A port that supports this is called a *multiport*. The channels are indexed $0, \ldots, N-1$, where N is the number of channels. An actor distinguishes between channels using this index in its send() and get() methods. By default, an IOPort is not a multiport, and thus supports only one channel (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 3.4. The *width of a relation* is the number of channels supported by the relation. If the relation is not a bus, then its width is one.

The *width of a port* is the sum of the widths of the relations linked to it. In figure 3.4, both the sending and receiving ports are multiports with width two. This is indicated by the “2” adjacent to each

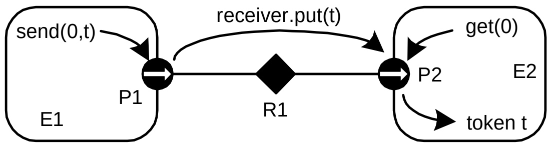


FIGURE 3.1. Message passing is mediated by the IOPort class. Its send() method obtains a reference to a remote receiver, and calls the put() method of the receiver, passing it the token *t*. The destination actor retrieves the token by calling the get() method of its input port.

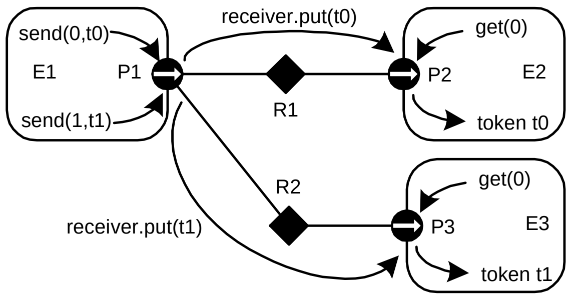


FIGURE 3.3. A port can support more than one channel, permitting an entity to send distinct data to distinct destinations via the same port. This feature is typically used when the number of destinations varies in different instances of the source actor.

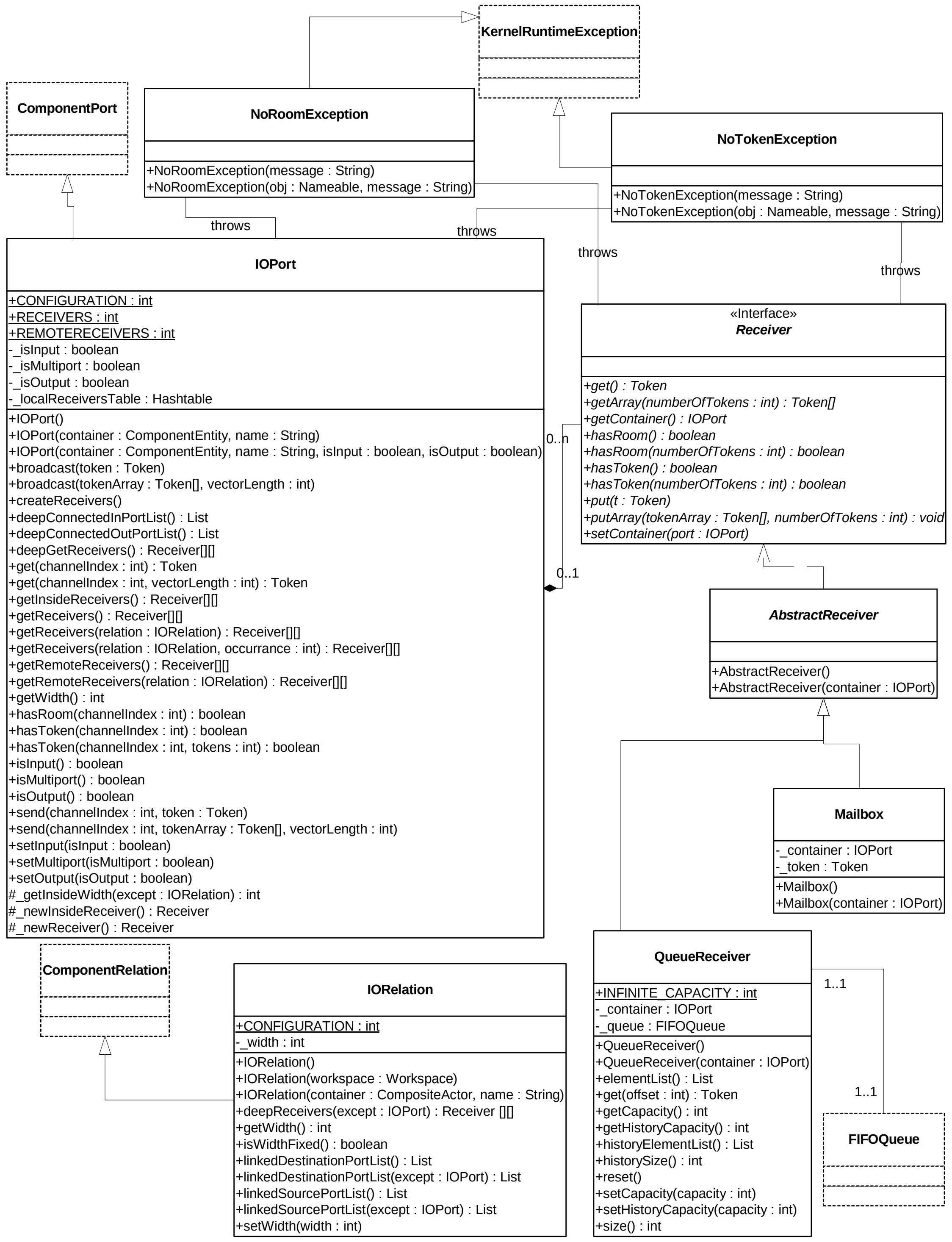

KernelRuntimeException
ComponentPort
NoRoomException
+NoRoomException(message : String)
+NoRoomException(obj : Nameable, message : String)
NoTokenException
+NoTokenException(message : String)
+NoTokenException(obj : Nameable, message : String)
throws
throws
throws
throws
IOPort
+CONFIGURATION : int
+RECEIVERS : int
+REMOTERECEIVERS : int
-_isInput : boolean
-_isMultiport : boolean
-_isOutput : boolean
-_localReceiversTable : Hashtable
+IOPort()
+IOPort(container : ComponentEntity, name : String)
+IOPort(container : ComponentEntity, name : String, isInput : boolean, isOutput : boolean)
+broadcast(token : Token)
+broadcast(tokenArray : Token[], vectorLength : int)
+createReceivers()
+deepConnectedInPortList() : List
+deepConnectedOutPortList() : List
+deepGetReceivers() : Receiver[][]
+get(channelIndex : int) : Token
+get(channelIndex : int, vectorLength : int) : Token
+getInsideReceivers() : Receiver[][]
+getReceivers() : Receiver[][]
+getReceivers(relation : IORelation) : Receiver[][]
+getReceivers(relation : IORelation, occurrance : int) : Receiver[][]
+getRemoteReceivers() : Receiver[][]
+getRemoteReceivers(relation : IORelation) : Receiver[][]
+getWidth() : int
+hasRoom(channelIndex : int) : boolean
+hasToken(channelIndex : int) : boolean
+hasToken(channelIndex : int, tokens : int) : boolean
+isInput() : boolean
+isMultiport() : boolean
+isOutput() : boolean
+send(channelIndex : int, token : Token)
+send(channelIndex : int, tokenArray : Token[], vectorLength : int)
+setInput(isInput : boolean)
+setMultiport(isMultiport : boolean)
+setOutput(isOutput : boolean)
#_getInsideWidth(except : IORelation) : int
#_newInsideReceiver() : Receiver
#_newReceiver() : Receiver
0..n
0..1
«Interface»
Receiver
+get() : Token
+getArray(numberOfTokens : int) : Token[]
+getContainer() : IOPort
+hasRoom() : boolean
+hasRoom(numberOfTokens : int) : boolean
+hasToken() : boolean
+hasToken(numberOfTokens : int) : boolean
+put(t : Token)
+putArray(tokenArray : Token[], numberOfTokens : int) : void
+setContainer(port : IOPort)
AbstractReceiver
+AbstractReceiver()
+AbstractReceiver(container : IOPort)
Mailbox
-_container : IOPort
-_token : Token
+Mailbox()
+Mailbox(container : IOPort)
ComponentRelation
IORelation
+CONFIGURATION : int
-_width : int
+IORelation()
+IORelation(workspace : Workspace)
+IORelation(container : CompositeActor, name : String)
+deepReceivers(except : IOPort) : Receiver [][]
+getWidth() : int
+isWidthFixed() : boolean
+linkedDestinationPortList() : List
+linkedDestinationPortList(except : IOPort) : List
+linkedSourcePortList() : List
+linkedSourcePortList(except : IOPort) : List
+setWidth(width : int)
QueueReceiver
+INFINITE_CAPACITY : int
-_container : IOPort
-_queue : FIFOQueue
+QueueReceiver()
+QueueReceiver(container : IOPort)
+elementList() : List
+get(offset : int) : Token
+getCapacity() : int
+getHistoryCapacity() : int
+historyElementList() : List
+historySize() : int
+reset()
+setCapacity(capacity : int)
+setHistoryCapacity(capacity : int)
+size() : int
1..1
1..1
FIFOQueue

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

port. Note that the width of a port could be zero, if there are no relations linked to a port (such a port is said to be *disconnected*). Thus, a port may have width zero, even though a relation cannot. By convention, in Ptolemy II, if a token is sent from such a port, the token goes nowhere. Similarly, if a token is sent via a relation that is not linked to any input ports, then the token goes nowhere. Such a relation is said to be *dangling*.

A given channel may reach multiple ports, as shown in figure 3.5. This is represented by a relation that is linked to multiple input ports. In the default implementation, in class IOPort, a 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.

3.2.2 Example

An elaborate example showing all of the above features is shown in figure 3.6. In that example, we assume that links are constructed in top-to-bottom order. The arrows in the ports indicate the direction of the flow of tokens, and thus specify whether the port is an input, an output, or both. Multiports are indicated by adjacent numbers larger than one.

The top relation is a bus with width two, and the rest are not busses. The width of port *P1* is four.

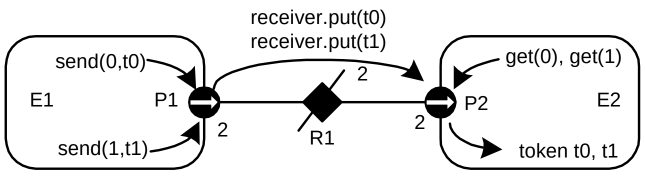


FIGURE 3.4. A bus is an IORelation that represents multiple channels. It is indicated by a relation with a slash through it, and the number adjacent to the bus is the width of the bus.

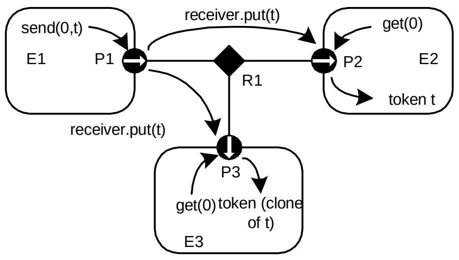


FIGURE 3.5. Channels may reach multiple destinations. This is represented by relations linking multiple input ports to an output port.

Its first two outputs (channels zero and one) go to *P4* and to the first two inputs of *P5*. The third output of *P1* goes nowhere. The fourth becomes the third input of *P5*, the first input of *P6*, and the only input of *P8*, which is both an input and an output port. Ports *P2* and *P8* send their outputs to the same set of destinations, except that *P8* does not send to itself. Port *P3* has width zero, so its send() method 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.

3.2.3 Transparent Ports

Recall that a port is transparent if its container is transparent (isOpaque() returns *false*). A CompositeActor is transparent unless it has a local director. Figure 3.7 shows an elaborate example where busses, input, and output ports are combined with transparent ports. The transparent ports are filled in white, and again arrows indicate the direction of token flow. The Jacl code to construct this example is shown in figure 3.8.

By definition, a transparent port is an input if either

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

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

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

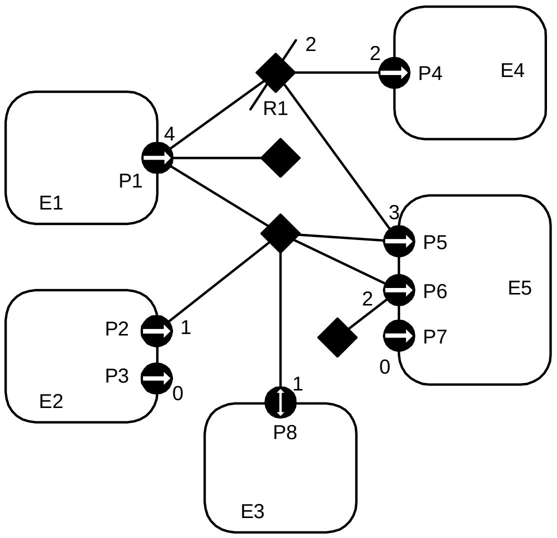


FIGURE 3.6. An elaborate example showing several features of the data transport mechanism.

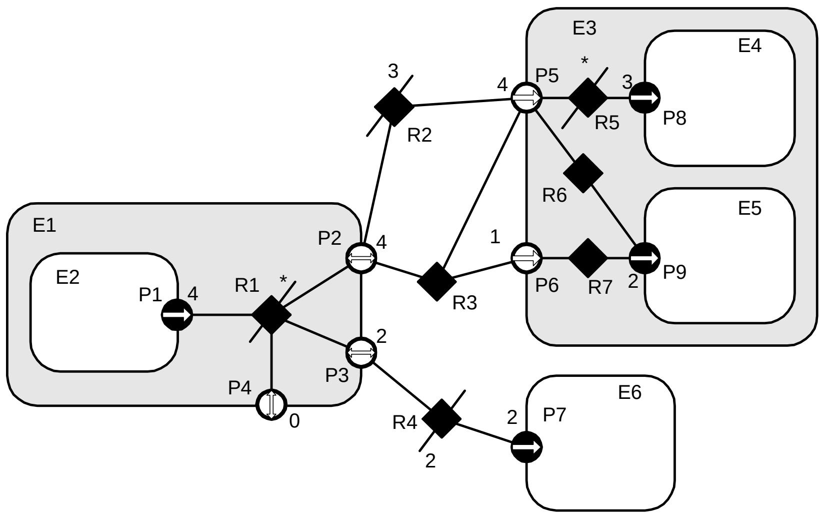


FIGURE 3.7. An example showing busses combined with input, output, and transparent ports.

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

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

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

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

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

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

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

FIGURE 3.8. Tcl Blend code to construct the example in figure 3.7.

the inside. Each such port is allowed to have at most one inside relation with an unspecified width, or an exception is thrown. If this inference yields a width of zero, then the width is defined to be one. Thus, R1 will have width 4 and R5 will have width 3 in this example. The width of a transparent port is the sum of the widths of the relations it is linked to on the outside (just like an ordinary port). Thus, P4 has width 0, P3 has width 2, and P2 has width 4. Recall that a port can have width 0, but a relation cannot have width less than one.

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

By default, an IORelation is not a bus, so its width is one. To turn it into a bus with unspecified width, call setWidth() with a zero argument. Note that getWidth() will nonetheless never return zero (it returns at least one). To find out whether setWidth() has been called with a zero argument, call isWidthFixed() (see figure 3.2). If a bus with unspecified width is not linked on the inside to any transparent ports, then its width is one. It is not allowed for a transparent port to have more than one bus with unspecified width linked on the inside (an exception will be thrown on any attempt to construct such a topology). Note further that a bus with unspecified width is still a bus, and so can only be linked to multiports.

In general, bus widths inside and outside a transparent port need not agree. For example, if $M < N$ in figure 3.9, then first M channels from P1 reach P3, and the last $N - M$ channels are dangling. If $M > N$, then all N channels from P1 reach P3, but the last $M - N$ channels at P3 are dangling. Attempting to get a token from these channels will trigger an exception. Sending a token to these channels just results in loss of the token.

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

3.2.4 Data Transfer in Various Models of Computation

The receiver used by an input port determines the communication protocol. This is closely bound to the model of computation. The IOPort class creates a new receiver when necessary by calling its _newReceiver() protected method. That method delegates to the director returned by getDirector(), calling its newReceiver() method (the Director class will be discussed in section 3.3 below). Thus, the director controls the communication protocol, in addition to its primary function of determining the flow of control. Here we discuss the receivers that are made available in the actor package. This should not be viewed as an exhaustive set, but rather as a particularly useful set of receivers. These receivers

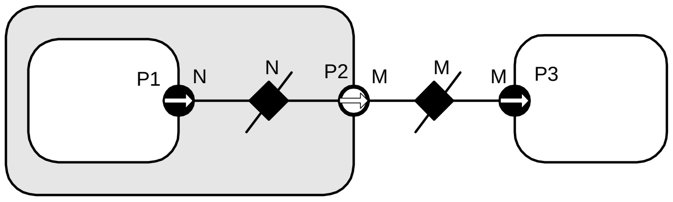


FIGURE 3.9. Bus widths inside and outside a transparent port need not agree..

are shown in figure 3.2.

Mailbox Communication. The Director base class by default returns a simple receiver called a Mailbox. A *mailbox* is a receiver 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 [44], which is a generalization of dataflow [54], 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 [75].

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

FIFOQueue is a support class that implements a first-in, first-out queue. It is part of the actor.util package, shown in figure 3.10. This class has two specialized features that make it particularly useful in this context. First, its capacity can be constrained or unconstrained. Second, it can record a finite or infinite history, the sequence of objects previously removed from the queue. The history mechanism is useful both to support tracing and debugging and to provide access to a finite buffer of previously consumed tokens.

An example of an actor definition is shown in figure 3.11. This actor has a multiport output. It

```
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 3.11. An actor that distributes successive input tokens to a set of output channels.

reads successive input tokens from the input port and distributes them to the output channels. This actor is written in a domain-polymorphic way, and can operate in any of a number of domains. If it is used in the PN domain, then its input will have a QueueReceiver and the output will be connected to ports with instances QueueReceiver.

Rendezvous Communications. Rendezvous, or synchronous communication, requires that the originator of a token and the recipient of a token both be simultaneously ready for the data transfer. As with process networks, the originator and the recipient are separate threads. The originating thread indicates 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

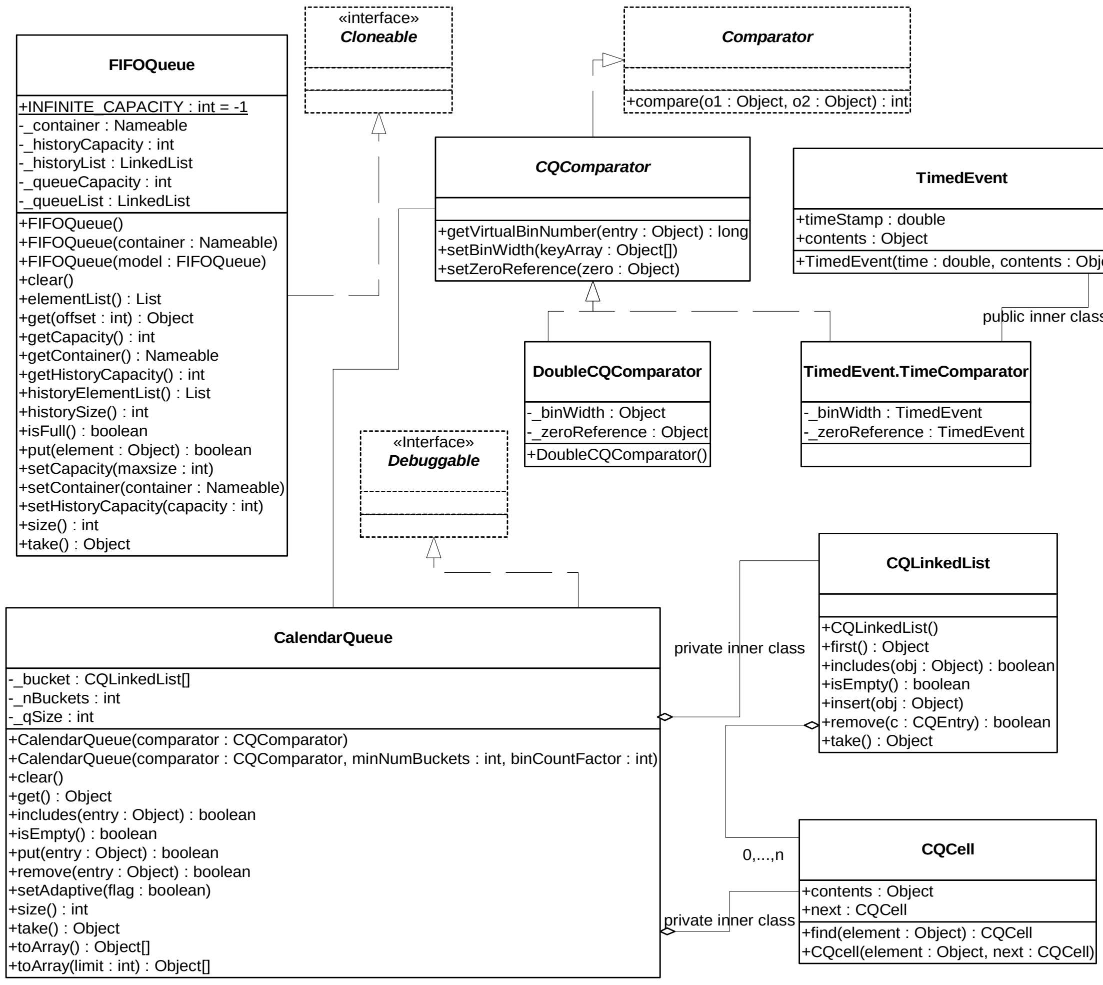

FIFOQueue
+INFINITE_CAPACITY : int = -1
-_container : Nameable
-_historyCapacity : int
-_historyList : LinkedList
-_queueCapacity : int
-_queueList : LinkedList
+FIFOQueue()
+FIFOQueue(container : Nameable)
+FIFOQueue(model : FIFOQueue)
+clear()
+elementList() : List
+get(offset : int) : Object
+getCapacity() : int
+getContainer() : Nameable
+getHistoryCapacity() : int
+historyElementList() : List
+historySize() : int
+isFull() : boolean
+put(element : Object) : boolean
+setCapacity(maxsize : int)
+setContainer(container : Nameable)
+setHistoryCapacity(capacity : int)
+size() : int
+take() : Object
«interface»
Cloneable
Comparator
+compare(o1 : Object, o2 : Object) : int
CQComparator
+getVirtualBinNumber(entry : Object) : long
+setBinWidth(keyArray : Object[])
+setZeroReference(zero : Object)
TimedEvent
+timeStamp : double
+contents : Object
+TimedEvent(time : double, contents : Object)
public inner class
DoubleCQComparator
-_binWidth : Object
-_zeroReference : Object
+DoubleCQComparator()
TimedEvent.TimeComparator
-_binWidth : TimedEvent
-_zeroReference : TimedEvent
«Interface»
Debuggable
CQLinkedList
+CQLinkedList()
+first() : Object
+includes(obj : Object) : boolean
+isEmpty() : boolean
+insert(obj : Object)
+remove(c : CQEntry) : boolean
+take() : Object
private inner class
CalendarQueue
-_bucket : CQLinkedList[]
-_nBuckets : int
-_qSize : int
+CalendarQueue(comparator : CQComparator)
+CalendarQueue(comparator : CQComparator, minNumBuckets : int, binCountFactor : int)
+clear()
+get() : Object
+includes(entry : Object) : boolean
+isEmpty() : boolean
+put(entry : Object) : boolean
+remove(entry : Object) : boolean
+setAdaptive(flag : boolean)
+size() : int
+take() : Object
+toArray() : Object[]
+toArray(limit : int) : Object[]
0,...,n
CQCell
+contents : Object
+next : CQCell
+find(element : Object) : CQCell
+CQcell(element : Object, next : CQCell)
private inner class

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

3.11 works unchanged in the CSP domain, although its behavior is different in that input and output actions involve rendezvous with another thread.

Nondeterministic transfers can be easily implemented using this mechanism. Suppose for example that a recipient is willing to rendezvous with any of several originating threads. It could spawn a thread for each. These threads should each call get(), which will suspend the thread until the originator is willing to rendezvous. When one of the originating threads is willing to rendezvous with it, it will call put(). The multiple recipient threads will all be awakened, but only 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 [4] can also be implemented.

Discrete-Event Communication. In the discrete-event model of computation, tokens that are transferred between actors have a *time stamp*, which specifies the order in which tokens should be processed by the recipients. The order is chronological, by increasing time stamp. To implement this, a discrete-event system will normally use a single, global, sorted queue rather than an instance of FIFOQueue in each input port. The kernel.util package, shown in figure 3.10, provides the CalendarQueue class, which gives an efficient and flexible implementation of such a sorted queue.

3.2.5 Discussion of the Data Transfer Mechanism

This data transfer mechanism has a number of interesting features. First, note that the actual transfer of data does not involve relations, so a model of computation could be defined that did not rely on relations. For example, a global name server might be used to address recipient 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.

3.3 Execution

The Executable interface, shown in figure 3.12, is implemented by the Director class, and is extended by the Actor interface. An *actor* is an executable entity. There are two types of actors, AtomicActor, which extends ComponentEntity, and CompositeActor, which extends CompositeEntity. As the names imply, an AtomicActor is a single entity, while a CompositeActor is an aggregation of actors. 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

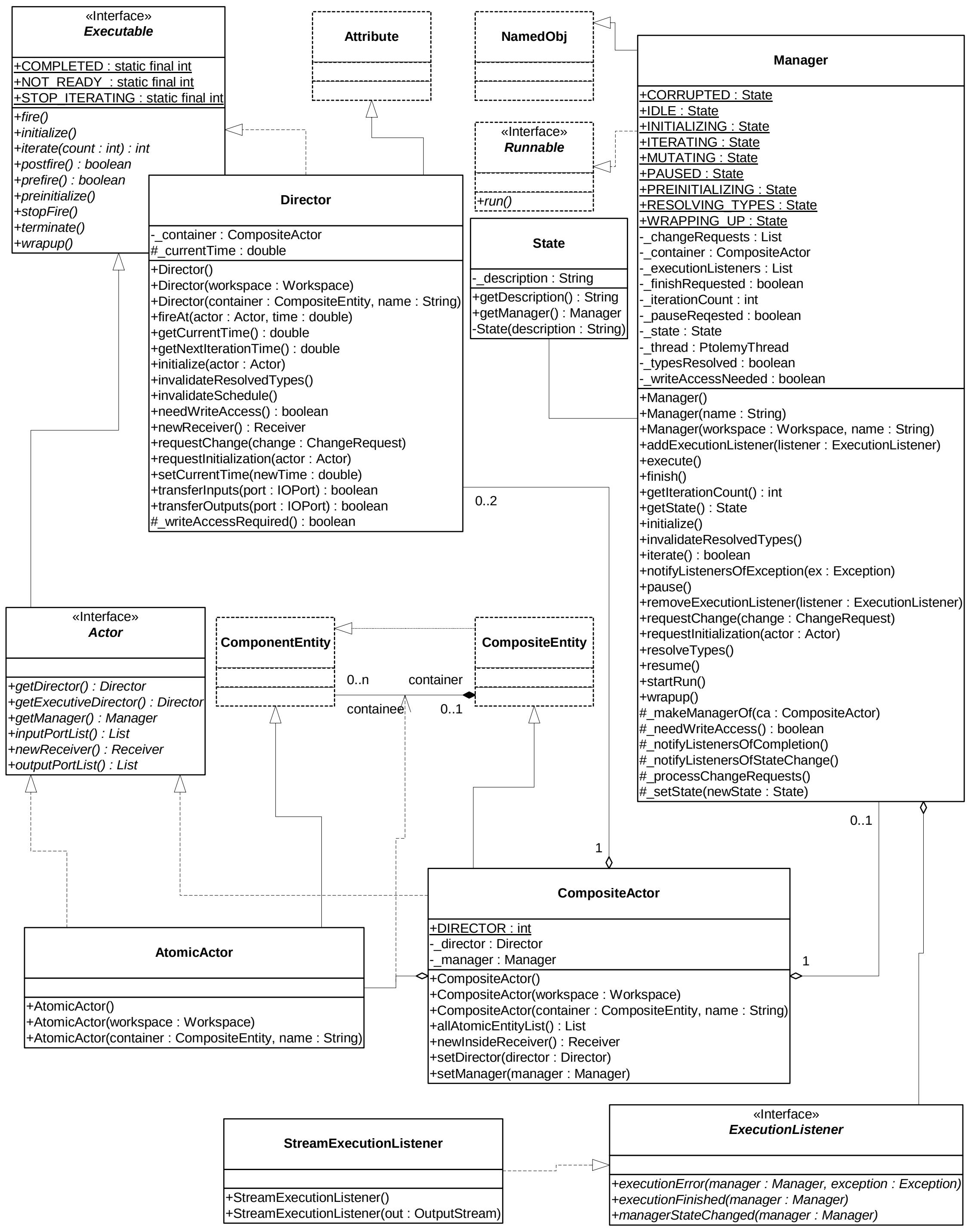

«Interface»
Executable
+COMPLETED : static final int
+NOT_READY : static final int
+STOP_ITERATING : static final int
+fire()
+initialize()
+iterate(count : int) : int
+postfire() : boolean
+prefire() : boolean
+preinitialize()
+stopFire()
+terminate()
+wrapup()
Attribute
NamedObj
Manager
+CORRUPTED : State
+IDLE : State
+INITIALIZING : State
+ITERATING : State
+MUTATING : State
+PAUSED : State
+PREINITIALIZING : State
+RESOLVING_TYPES : State
+WRAPPING_UP : State
-_changeRequests : List
-_container : CompositeActor
-_executionListeners : List
-_finishRequested : boolean
-_iterationCount : int
-_pauseReqested : boolean
-_state : State
-_thread : PtolemyThread
-_typesResolved : boolean
-_writeAccessNeeded : boolean
+Manager()
+Manager(name : String)
+Manager(workspace : Workspace, name : String)
+addExecutionListener(listener : ExecutionListener)
+execute()
+finish()
+getIterationCount() : int
+getState() : State
+initialize()
+invalidateResolvedTypes()
+iterate() : boolean
+notifyListenersOfException(ex : Exception)
+pause()
+removeExecutionListener(listener : ExecutionListener)
+requestChange(change : ChangeRequest)
+requestInitialization(actor : Actor)
+resolveTypes()
+resume()
+startRun()
+wrapup()
#_makeManagerOf(ca : CompositeActor)
#_needWriteAccess() : boolean
#_notifyListenersOfCompletion()
#_notifyListenersOfStateChange()
#_processChangeRequests()
#_setState(newState : State)
«Interface»
Runnable
+run()
Director
-_container : CompositeActor
#_currentTime : double
+Director()
+Director(workspace : Workspace)
+Director(container : CompositeEntity, name : String)
+fireAt(actor : Actor, time : double)
+getCurrentTime() : double
+getNextIterationTime() : double
+initialize(actor : Actor)
+invalidateResolvedTypes()
+invalidateSchedule()
+needWriteAccess() : boolean
+newReceiver() : Receiver
+requestChange(change : ChangeRequest)
+requestInitialization(actor : Actor)
+setCurrentTime(newTime : double)
+transferInputs(port : IOPort) : boolean
+transferOutputs(port : IOPort) : boolean
#_writeAccessRequired() : boolean
State
-_description : String
+getDescription() : String
+getManager() : Manager
-State(description : String)
0..2
«Interface»
Actor
+getDirector() : Director
+getExecutiveDirector() : Director
+getManager() : Manager
+inputPortList() : List
+newReceiver() : Receiver
+outputPortList() : List
ComponentEntity
CompositeEntity
0..n
container
containee
0..1
0..1
1
CompositeActor
+DIRECTOR : int
-_director : Director
-_manager : Manager
+CompositeActor()
+CompositeActor(workspace : Workspace)
+CompositeActor(container : CompositeEntity, name : String)
+allAtomicEntityList() : List
+newInsideReceiver() : Receiver
+setDirector(director : Director)
+setManager(manager : Manager)
1
AtomicActor
+AtomicActor()
+AtomicActor(workspace : Workspace)
+AtomicActor(container : CompositeEntity, name : String)
StreamExecutionListener
+StreamExecutionListener()
+StreamExecutionListener(out : OutputStream)
«Interface»
ExecutionListener
+executionError(manager : Manager, exception : Exception)
+executionFinished(manager : Manager)
+managerStateChanged(manager : Manager)

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

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. 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 3.3.4 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.

3.3.1 Director

A *director* governs the execution of a composite entity. A *manager* governs the overall execution of a model. An example of the use of these classes is shown in figure 3.13. In that example, a top-level entity, E0, has an instance of Director, D1, that serves the role of its local director. A *local director* is responsible for execution of the components within the composite. It will perform any scheduling that might be necessary, dispatch threads that need to be started, generate code that needs to be generated, etc. In the example, D1 also serves as an executive director for E2. The *executive director* associated with an actor is the director that is responsible for firing the actor.

A composite actor that is not at the top level may or may not have its own local director. If it has a local director, then it defined to be opaque (isOpaque() returns *true*). In figure 3.13, E2 has a local director and E3 does not. The contents of E3 are directly under the control of D1, as if the hierarchy were flattened. By contrast, the contents of E2 are under the control of D2, which in turn is under the control of D1. In the terminology of the previous generation, Ptolemy 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 3.3.4.

We define the *director* (vs. local director or executive director) of an actor to be either its local director (if it has one) or its executive director (if it does not). A composite actor that is not at the top level has as its executive director the director of the container. Every executable actor has a director except the top-level composite actor, and that director is what is returned by the getDirector() method of the Actor interface (see figure 3.12).

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

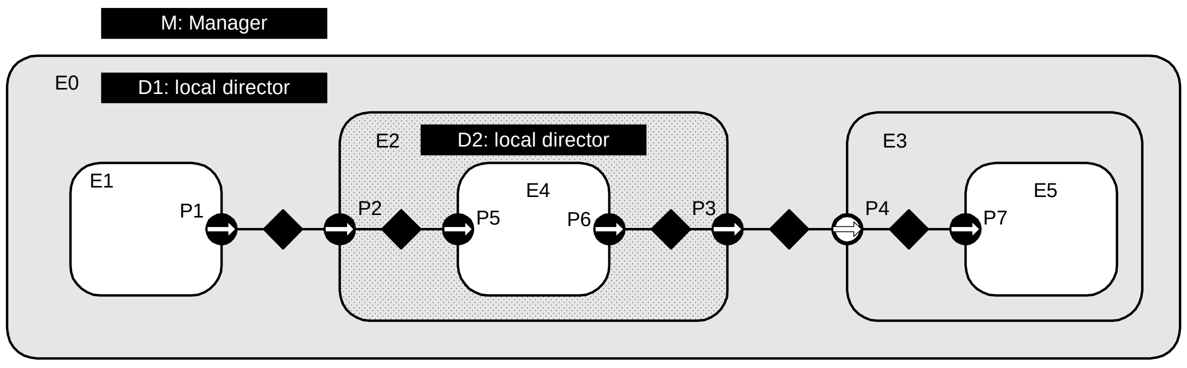


FIGURE 3.13. Example application, showing a typical arrangement of actors, directors, and managers.

ping condition for the model of computation is reached.

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

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

Two common sequences of method calls between actors and directors are shown in figure 3.14 and 3.15. These differ in the shaded areas, which define the domain-specific sequencing of actor firings. In figure 3.14, the fire() method of the director selects an actor, invokes its prefire() method, and if that returns true, invokes its fire() method some number of times (domain dependent) followed by its postfire() method. In figure 3.15, the fire() method of the director invokes the prefire() method of all the actors before invoking any of their fire() methods.

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

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

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

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.

3.3.2 Manager

While a director implements a model of computation, a *manager* controls the overall execution of a model. The manager interacts with a single composite actor, known as a *top level composite actor*. The Manager class is shown in figure 3.12. Execution of a model is implemented by three methods, execute(), run() and startRun(). The startRun() method spawns a thread that calls run(), and then imme-

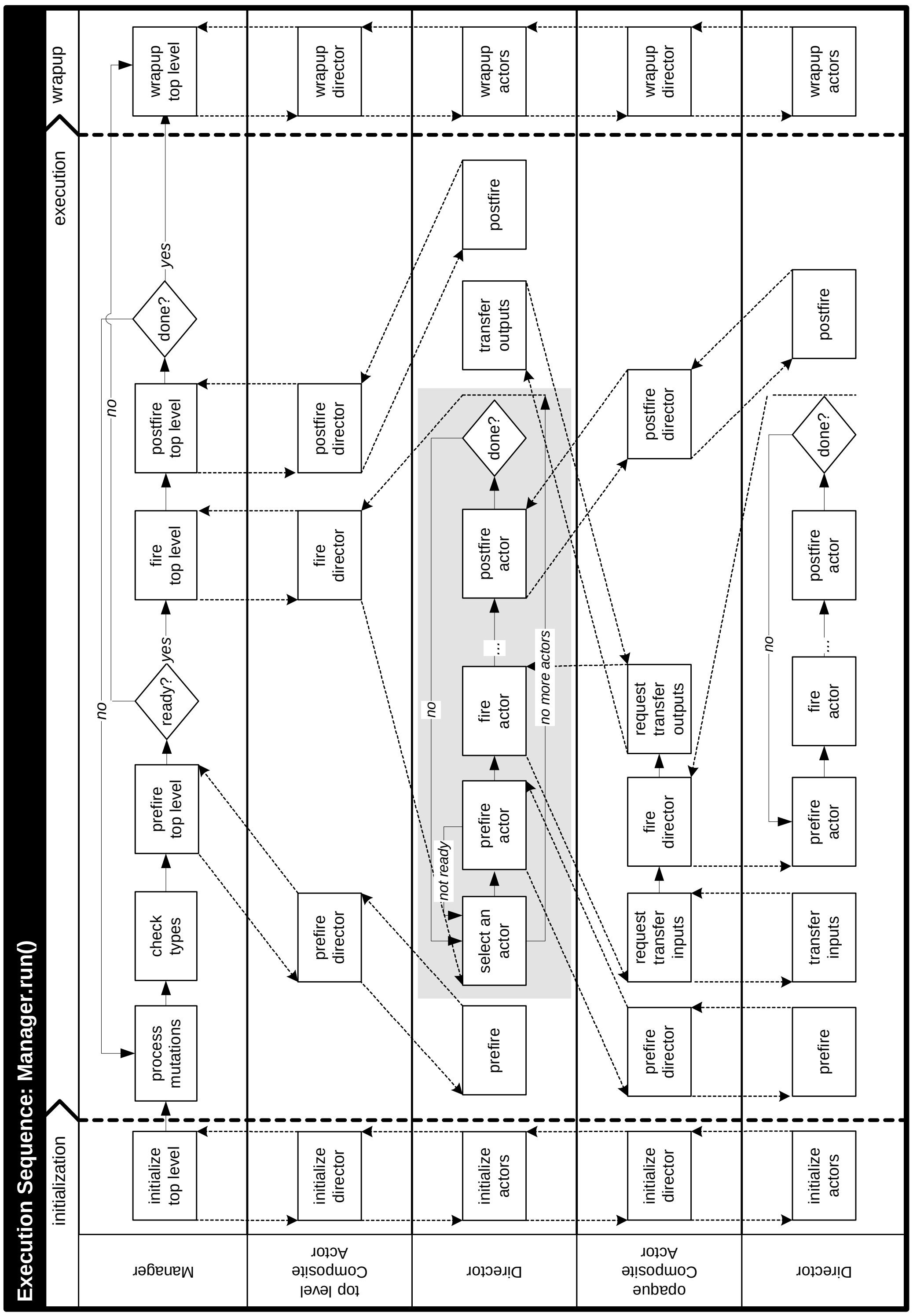


FIGURE 3.14. Example execution sequence implemented by run() method of the Director class.

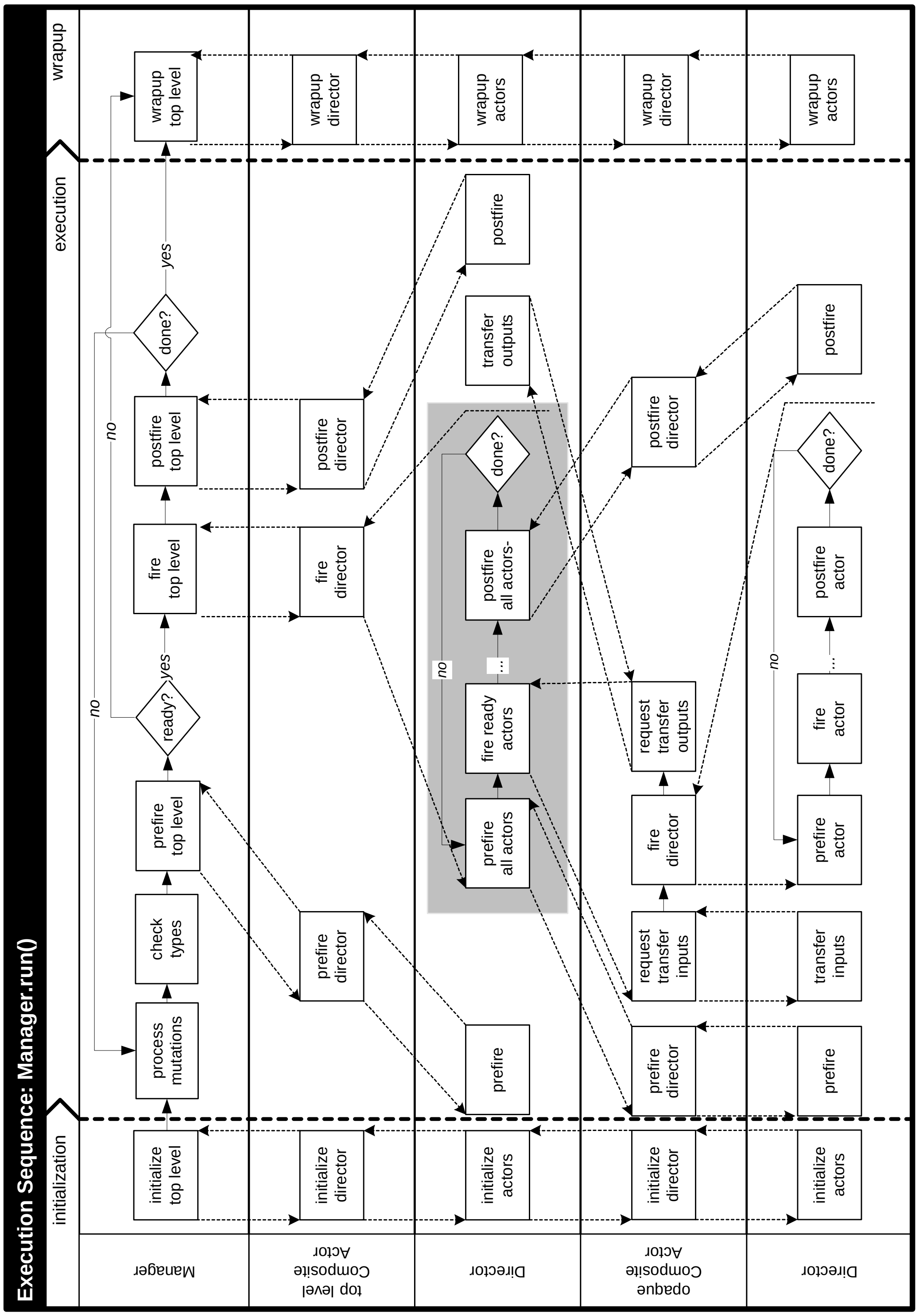


FIGURE 3.15. Alternative execution sequence implemented by run() method of the Director class.

diately 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.

3.3.3 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 StreamExecutionListener class. This class reports all events on the standard output.

3.3.4 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 3.16. Assume that both E0 and E2 have local directors (D1 and D2), so E2 is opaque. The ports of E2 therefore are opaque, as indicated in the figure by their solid fill. Since its ports are opaque, when a token is sent from the output port P1, it is deposited in P2, not P5.

In the execution sequences of figures 3.14 and 3.15, E2 is treated as an atomic actor by D1; i.e. D1 acts as an executive director to E2. Thus, the fire() method of D1 invokes the prefire(), fire(), and postfire() methods of E1, E2, and E3. The fire() method of E2 is responsible for transferring the token from P2 to P5. It does this by delegating to its local director, invoking its transferInputs() method. It then invokes the fire() method of D2, which in turn invokes the prefire(), fire(), and postfire() methods of E4.

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

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

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

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

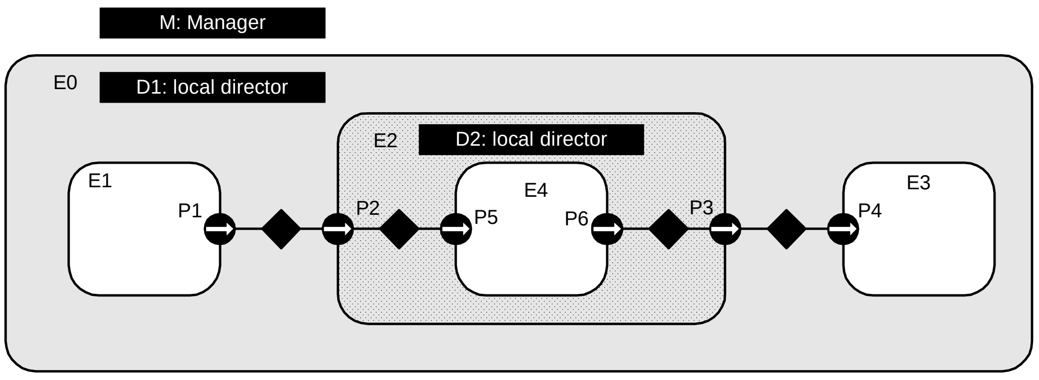


FIGURE 3.16. An example of an opaque composite actor. E0 and E2 both have local directors, not necessarily implementing the same model of computation.

3.4 Scheduler and Process Support

The actor package has two 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 are shown in figure 3.17 and figure 3.18.

3.4.1 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 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

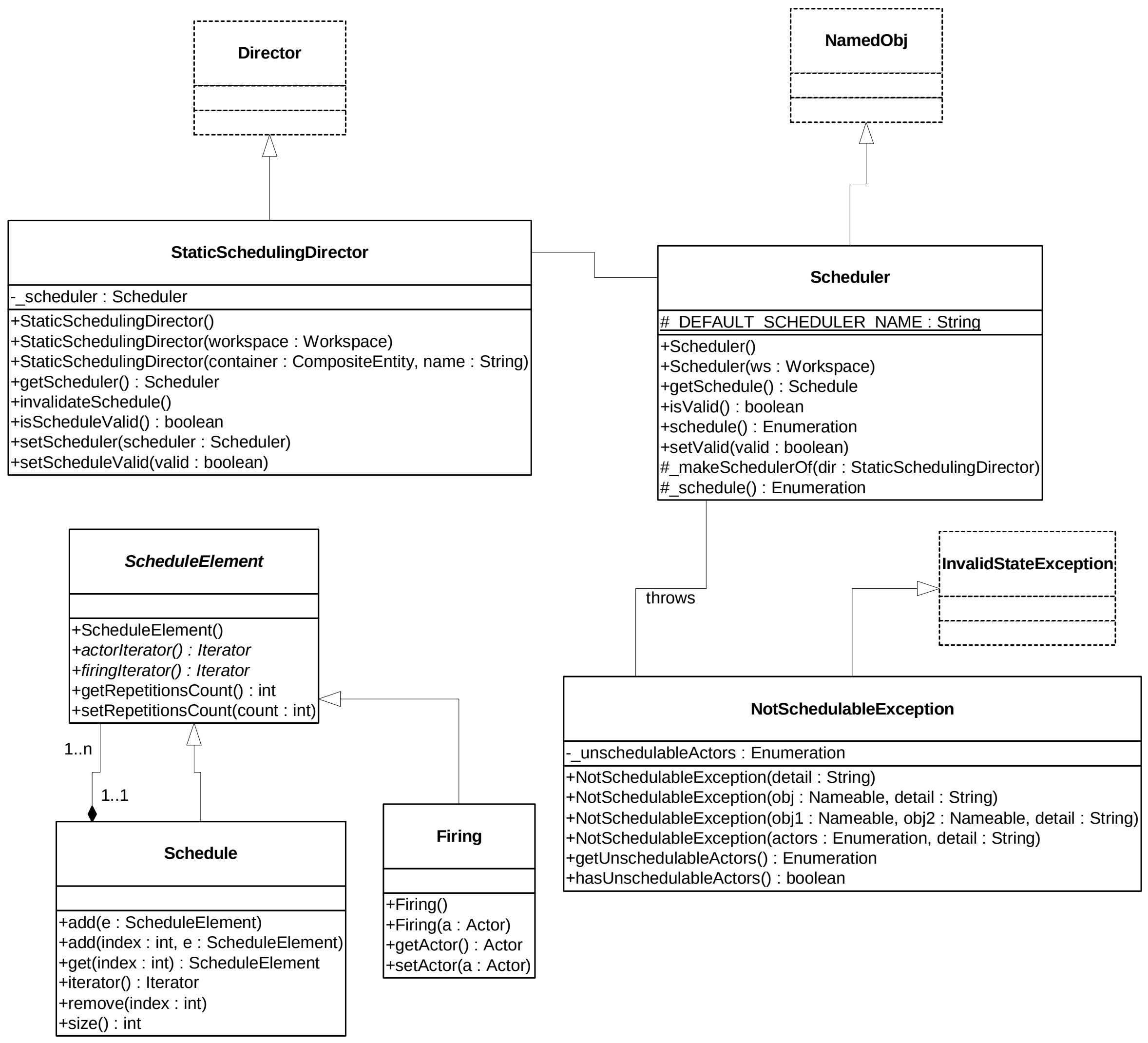

Director
NamedObj
StaticSchedulingDirector
-_scheduler : Scheduler
+StaticSchedulingDirector()
+StaticSchedulingDirector(workspace : Workspace)
+StaticSchedulingDirector(container : CompositeEntity, name : String)
+getScheduler() : Scheduler
+invalidateSchedule()
+isScheduleValid() : boolean
+setScheduler(scheduler : Scheduler)
+setScheduleValid(valid : boolean)
Scheduler
#_DEFAULT_SCHEDULER_NAME : String
+Scheduler()
+Scheduler(ws : Workspace)
+getSchedule() : Schedule
+isValid() : boolean
+schedule() : Enumeration
+setValid(valid : boolean)
#_makeSchedulerOf(dir : StaticSchedulingDirector)
#_schedule() : Enumeration
ScheduleElement
+ScheduleElement()
+actorIterator() : Iterator
+firingIterator() : Iterator
+getRepetitionsCount() : int
+setRepetitionsCount(count : int)
InvalidStateException
throws
NotSchedulableException
-_unschedulableActors : Enumeration
+NotSchedulableException(detail : String)
+NotSchedulableException(obj : Nameable, detail : String)
+NotSchedulableException(obj1 : Nameable, obj2 : Nameable, detail : String)
+NotSchedulableException(actors : Enumeration, detail : String)
+getUnschedulableActors() : Enumeration
+hasUnschedulableActors() : boolean
1..n
1..1
Schedule
+add(e : ScheduleElement)
+add(index : int, e : ScheduleElement)
+get(index : int) : ScheduleElement
+iterator() : Iterator
+remove(index : int)
+size() : int
Firing
+Firing()
+Firing(a : Actor)
+getActor() : Actor
+setActor(a : Actor)

FIGURE 3.17. UML static structure diagram for the actor.sched package.

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 element, 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

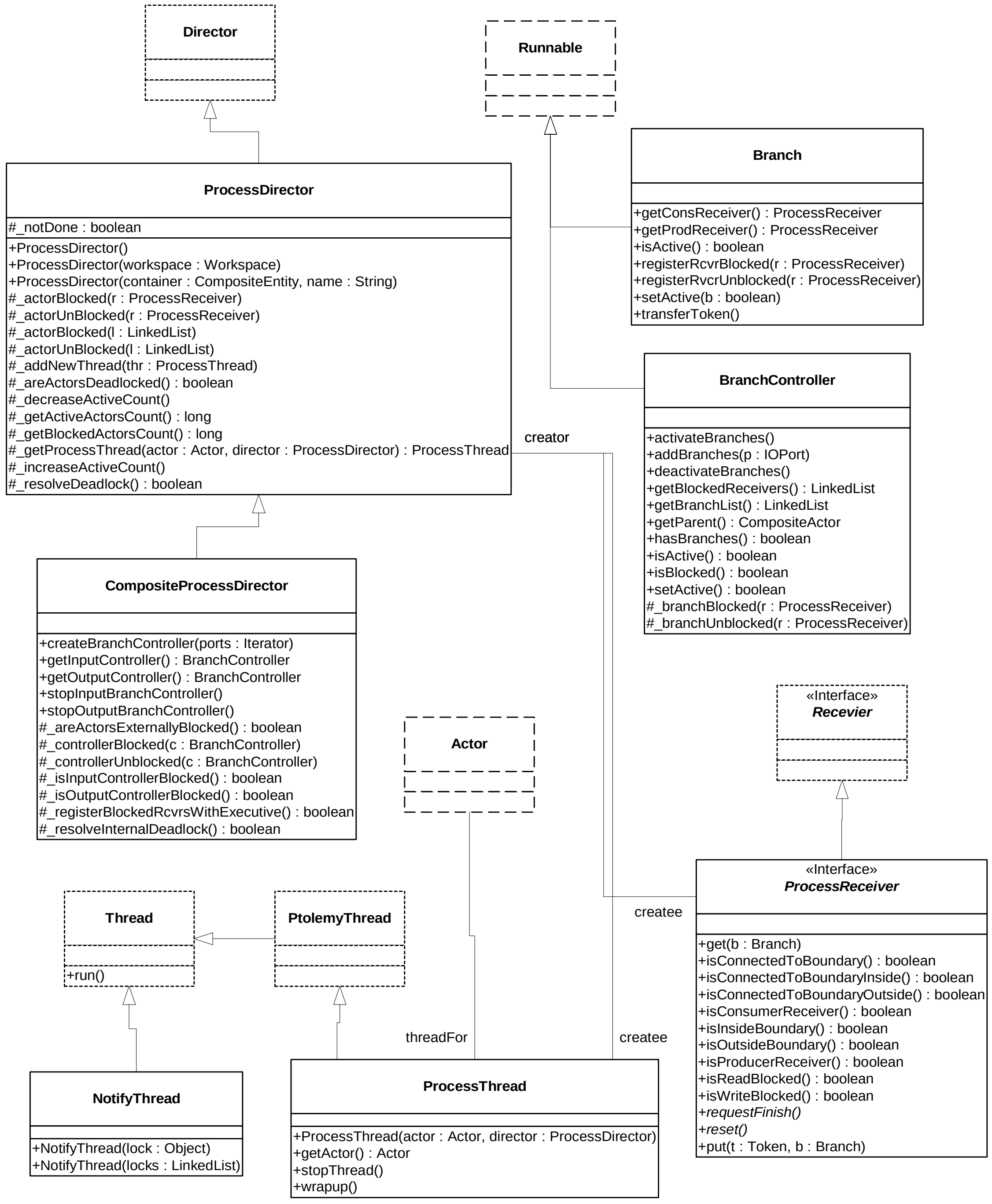

Director
Runnable
Branch
+getConsReceiver() : ProcessReceiver
+getProdReceiver() : ProcessReceiver
+isActive() : boolean
+registerRcvrBlocked(r : ProcessReceiver)
+registerRvcrUnblocked(r : ProcessReceiver)
+setActive(b : boolean)
+transferToken()
ProcessDirector
#_notDone : boolean
+ProcessDirector()
+ProcessDirector(workspace : Workspace)
+ProcessDirector(container : CompositeEntity, name : String)
#_actorBlocked(r : ProcessReceiver)
#_actorUnBlocked(r : ProcessReceiver)
#_actorBlocked(l : LinkedList)
#_actorUnBlocked(l : LinkedList)
#_addNewThread(thr : ProcessThread)
#_areActorsDeadlocked() : boolean
#_decreaseActiveCount()
#_getActiveActorsCount() : long
#_getBlockedActorsCount() : long
#_getProcessThread(actor : Actor, director : ProcessDirector) : ProcessThread
#_increaseActiveCount()
#_resolveDeadlock() : boolean
creator
BranchController
+activateBranches()
+addBranches(p : IOPort)
+deactivateBranches()
+getBlockedReceivers() : LinkedList
+getBranchList() : LinkedList
+getParent() : CompositeActor
+hasBranches() : boolean
+isActive() : boolean
+isBlocked() : boolean
+setActive() : boolean
#_branchBlocked(r : ProcessReceiver)
#_branchUnblocked(r : ProcessReceiver)
CompositeProcessDirector
+createBranchController(ports : Iterator)
+getInputController() : BranchController
+getOutputController() : BranchController
+stopInputBranchController()
+stopOutputBranchController()
#_areActorsExternallyBlocked() : boolean
#_controllerBlocked(c : BranchController)
#_controllerUnblocked(c : BranchController)
#_isInputControllerBlocked() : boolean
#_isOutputControllerBlocked() : boolean
#_registerBlockedRcvrsWithExecutive() : boolean
#_resolveInternalDeadlock() : boolean
Actor
«Interface»
Recevier
«Interface»
ProcessReceiver
+get(b : Branch)
+isConnectedToBoundary() : boolean
+isConnectedToBoundaryInside() : boolean
+isConnectedToBoundaryOutside() : boolean
+isConsumerReceiver() : boolean
+isInsideBoundary() : boolean
+isOutsideBoundary() : boolean
+isProducerReceiver() : boolean
+isReadBlocked() : boolean
+isWriteBlocked() : boolean
+requestFinish()
+reset()
+put(t : Token, b : Branch)
createe
Thread
+run()
PtolemyThread
threadFor
createe
NotifyThread
+NotifyThread(lock : Object)
+NotifyThread(locks : LinkedList)
ProcessThread
+ProcessThread(actor : Actor, director : ProcessDirector)
+getActor() : Actor
+stopThread()
+wrapup()

FIGURE 3.18. UML static structure diagram for the actor.process package.

firingIterator() method on the schedule. In the iterator, the schedule is expanded recursively, with each firing repeated the appropriate number of times.[1]

3.4.2 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 [72][47]. 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 [46]. 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. Returning 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

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

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 3.19. 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 [47].

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

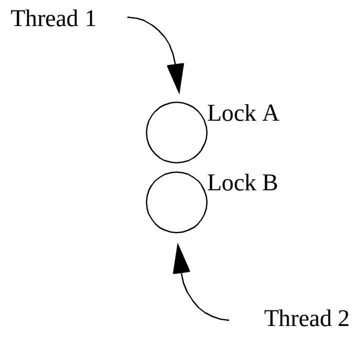


FIGURE 3.19. Deadlock Due to Unordered Locking.

locks be acquired in the following order:

$$\begin{tikzcd} \textbf{Workspace} \arrow[r] & \textbf{ProcessReceiver} \arrow[r] & \textbf{ProcessDirector} \arrow[r] & \textbf{AtomicActor} \end{tikzcd}$$

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 obj, and regains read access on the workspace before returning.

4

Data Package

Authors: *Bart Kienhuis*
Edward A. Lee
Xiaojun Liu
Neil Smyth
Yuhong Xiong

4.1 Introduction

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

4.2 Data Encapsulation

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

Tokens in Ptolemy II, except ObjectToken, are immutable. This means that their value cannot be changed after the instance of Token is constructed. The value of a token must therefore be specified as a constructor argument, and there must be no other mechanism for setting the value. If the value must be changed, a new instance of Token must be constructed.

There are several reasons for making tokens immutable.

- First, when a token is to be sent to several receivers, we want to be sure that all receivers get the same data. Each receiver is sent a reference to the same token. If the Token were not immutable,

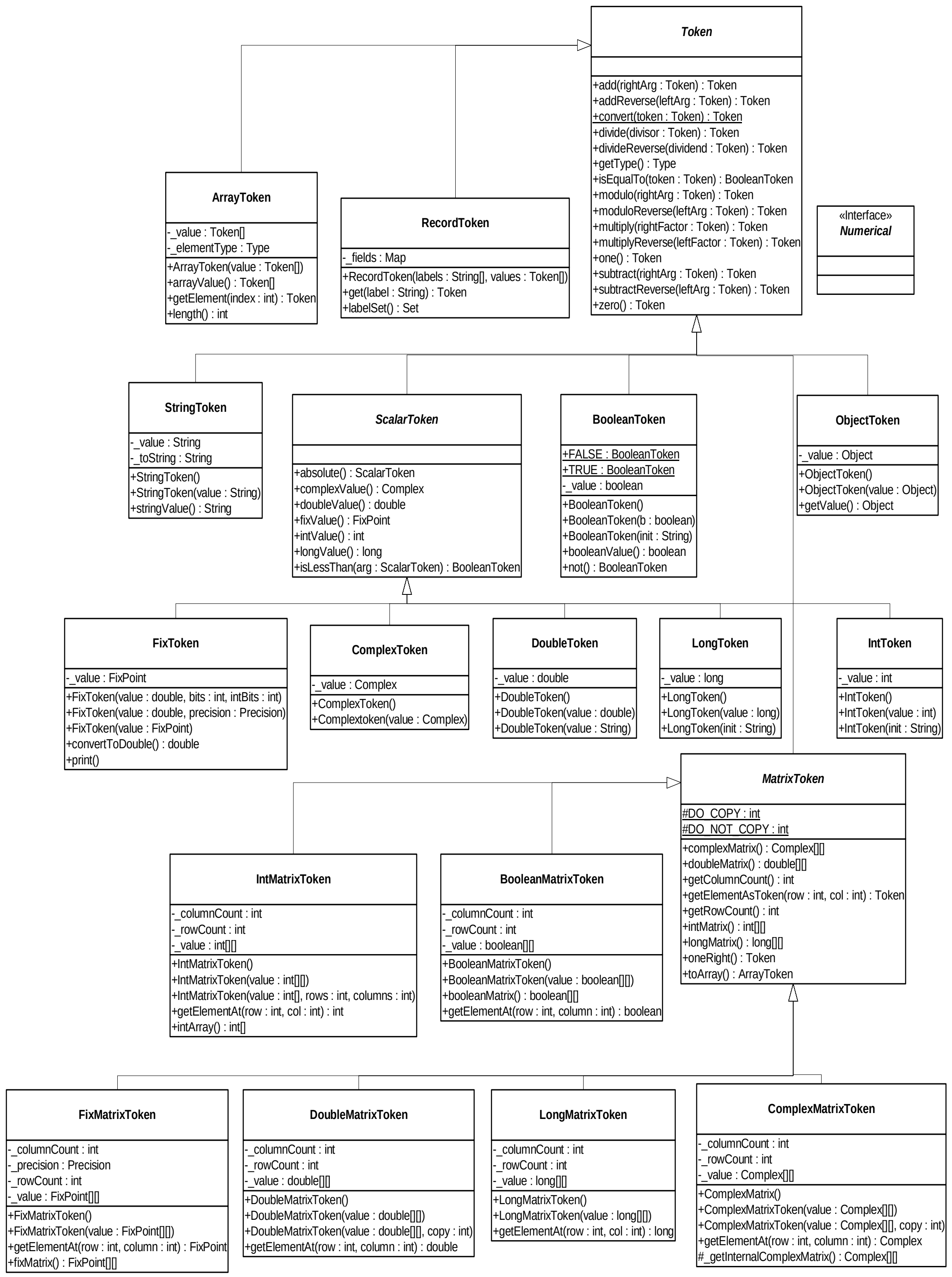

Token
+add(rightArg : Token) : Token
+addReverse(leftArg : Token) : Token
+convert(token : Token) : Token
+divide(divisor : Token) : Token
+divideReverse(dividend : Token) : Token
+getType() : Type
+isEqualTo(token : Token) : BooleanToken
+modulo(rightArg : Token) : Token
+moduloReverse(leftArg : Token) : Token
+multiply(rightFactor : Token) : Token
+multiplyReverse(leftFactor : Token) : Token
+one() : Token
+subtract(rightArg : Token) : Token
+subtractReverse(leftArg : Token) : Token
+zero() : Token
«Interface»
Numerical
ArrayToken
-_value : Token[]
-_elementType : Type
+ArrayToken(value : Token[])
+arrayValue() : Token[]
+getElement(index : int) : Token
+length() : int
RecordToken
-_fields : Map
+RecordToken(labels : String[], values : Token[])
+get(label : String) : Token
+labelSet() : Set
StringToken
-_value : String
-_toString : String
+StringToken()
+StringToken(value : String)
+stringValue() : String
ScalarToken
+absolute() : ScalarToken
+complexValue() : Complex
+doubleValue() : double
+fixValue() : FixPoint
+intValue() : int
+longValue() : long
+isLessThan(arg : ScalarToken) : BooleanToken
BooleanToken
+FALSE : BooleanToken
+TRUE : BooleanToken
-_value : boolean
+BooleanToken()
+BooleanToken(b : boolean)
+BooleanToken(init : String)
+booleanValue() : boolean
+not() : BooleanToken
ObjectToken
-_value : Object
+ObjectToken()
+ObjectToken(value : Object)
+getValue() : Object
FixToken
-_value : FixPoint
+FixToken(value : double, bits : int, intBits : int)
+FixToken(value : double, precision : Precision)
+FixToken(value : FixPoint)
+convertToDouble() : double
+print()
ComplexToken
-_value : Complex
+ComplexToken()
+Complextoken(value : Complex)
DoubleToken
-_value : double
+DoubleToken()
+DoubleToken(value : double)
+DoubleToken(value : String)
LongToken
-_value : long
+LongToken()
+LongToken(value : long)
+LongToken(init : String)
IntToken
-_value : int
+IntToken()
+IntToken(value : int)
+IntToken(init : String)
MatrixToken
#DO_COPY : int
#DO_NOT_COPY : int
+complexMatrix() : Complex[][]
+doubleMatrix() : double[][]
+getColumnCount() : int
+getElementAsToken(row : int, col : int) : Token
+getRowCount() : int
+intMatrix() : int[][]
+longMatrix() : long[][]
+oneRight() : Token
+toArray() : ArrayToken
IntMatrixToken
-_columnCount : int
-_rowCount : int
-_value : int[][]
+IntMatrixToken()
+IntMatrixToken(value : int[][])
+IntMatrixToken(value : int[], rows : int, columns : int)
+getElementAt(row : int, col : int) : int
+intArray() : int[]
BooleanMatrixToken
-_columnCount : int
-_rowCount : int
-_value : boolean[][]
+BooleanMatrixToken()
+BooleanMatrixToken(value : boolean[][])
+booleanMatrix() : boolean[][]
+getElementAt(row : int, column : int) : boolean
FixMatrixToken
-_columnCount : int
-_precision : Precision
-_rowCount : int
-_value : FixPoint[][]
+FixMatrixToken()
+FixMatrixToken(value : FixPoint[][])
+getElementAt(row : int, column : int) : FixPoint
+fixMatrix() : FixPoint[][]
DoubleMatrixToken
-_columnCount : int
-_rowCount : int
-_value : double[][]
+DoubleMatrixToken()
+DoubleMatrixToken(value : double[][])
+DoubleMatrixToken(value : double[][], copy : int)
+getElementAt(row : int, column : int) : double
LongMatrixToken
-_columnCount : int
-_rowCount : int
-_value : long[][]
+LongMatrixToken()
+LongMatrixToken(value : long[][])
+getElementAt(row : int, col : int) : long
ComplexMatrixToken
-_columnCount : int
-_rowCount : int
-_value : Complex[][]
+ComplexMatrix()
+ComplexMatrixToken(value : Complex[][])
+ComplexMatrixToken(value : Complex[][], copy : int)
+getElementAt(row : int, column : int) : Complex
#_getInternalComplexMatrix() : Complex[][]

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

then it would be necessary to clone the token for all receivers after the first one.

- Second, we use tokens to parameterize objects, and parameters have mutual dependencies. That is, the value of a parameter may depend on the value of other parameters. The value of a parameter is represented by an instance of Token. If that token were allowed to change value without notifying the parameter, then the parameter would not be able to notify other parameters that depend on its value. Thus, a mutable token would have to implement a publish-and-subscribe mechanism so that parameters could subscribe and thus be notified of any changes. By making tokens immutable, we greatly simplify the design.
- Finally, having our Tokens immutable makes them similar in concept to the data wrappers in Java, like Double, Integer, etc., which are also immutable.

An ObjectToken contains a reference to an arbitrary Java object created by the user. Since the user may modify the object after the token is constructed, ObjectToken is an exception to immutability. Moreover, the getValue() method returns a reference to the object. That reference can be used to modify the object. Although ObjectToken could clone the object in the constructor and return another clone in getValue(), this would require the object to be cloneable, which severely limits the use of the ObjectToken. In addition, even if the object is cloneable, since the default implementation of clone() only makes a shallow copy, it is still not enough to enforce immutability. In addition, cloning a large object could be expensive. For these reasons, the ObjectToken does not enforce immutability, but rather relies on the cooperation from the user. Violating this convention could lead to unintended non-determinism.

For matrix tokens, immutability requires the contained matrix (Java array) to be copied when the token is constructed, and when the matrix is returned in response to queries such as intMatrix(), doubleMatrix(), etc. This is because arrays are objects in Java. Since the cost of copying large matrices is non-trivial, the user should not make more queries than necessary. The getElementAt() method should be used to read the contents of the matrix.

ArrayToken is a token that contains an array of tokens. All the element tokens must have the same type, but that type can be any token type, including the type of the ArrayToken itself. That is, we can have an array of arrays. ArrayToken is different from the MatrixTokens in that MatrixTokens contain primitive data, such as int, double, while ArrayToken contains Ptolemy Tokens. MatrixTokens are very efficient for storing two dimensional primitive data, while ArrayToken offers more flexibility in type specifications.

RecordToken contains a set of labeled values, like the structure in the C language. The values can be arbitrary tokens, and they are not required to have the same type. ArrayToken and RecordToken will be discussed in more detail in the Type System chapter.

4.3 Polymorphism

4.3.1 Polymorphic Arithmetic Operators

One of the goals of the data package is to support polymorphic operations between tokens. For this, the base Token class defines methods for the 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

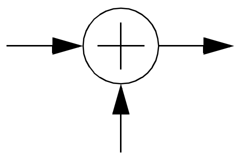
```
a.modulo(c)
```

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 in detail below.

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

4.3.2 Lossless Type Conversion

For the above arithmetic operations, if the two tokens being operated on have different types, type conversion is needed. In Ptolemy II, only conversions that do not lose information are implicitly performed. Lossy conversions must be explicitly done by the user, either through casting or by other means. The lossless type conversion relation among different token types is modeled as a partially ordered set called the *type lattice*, shown in figure 4.2. In that diagram, type *A* is *greater than* type *B* if there is a path upwards from *B* to *A*. Thus, ComplexMatrix is greater than Int. Type *A* is *less than* type *B* if there is a path downwards from *B* to *A*. Thus, Int is less than ComplexMatrix. Otherwise, types *A* and *B* are *incomparable*. Complex and Long, for example, are incomparable.

In the type lattice, a type can be losslessly converted to any type greater than it. This hierarchy is related to the inheritance hierarchy of the token classes in that a subclass is always less than its super class in the type lattice. However, some adjacent types in the lattice are not related by inheritance.

This hierarchy is realized by the TypeLattice class 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, *Numerical* and *Scalar*, are abstract. They cannot be instantiated. This is indicated

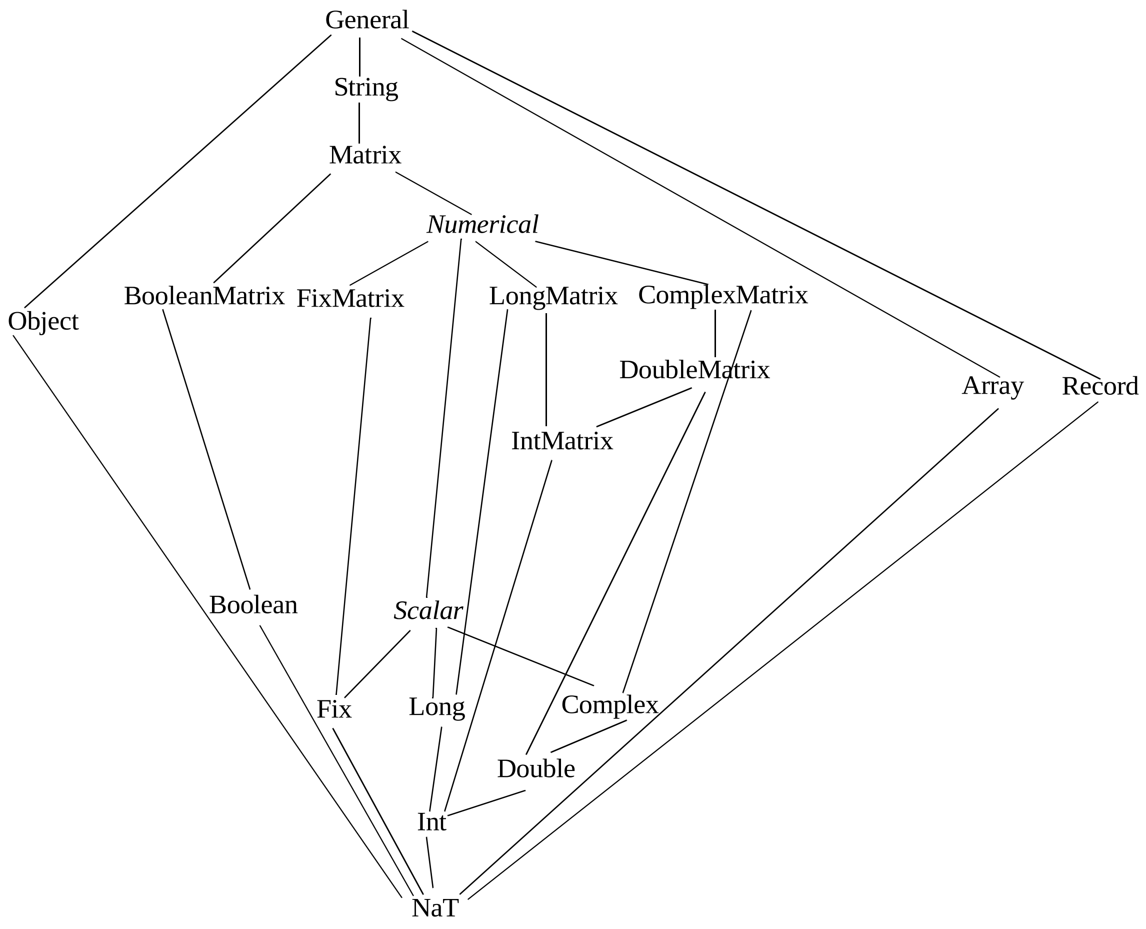


FIGURE 4.2. The type lattice.

in the type lattice by italics.

Type conversion is done by the static method convert() in the token classes. This method converts the argument into an instance of the class implementing this method. For example, DoubleToken.convert(Token token) converts the specified token into an instance of DoubleToken. The convert() method can convert any token immediately below it in the type hierarchy into an instance of its own class. If the argument is higher in the type hierarchy, or is incomparable with its own class, convert() throws an exception. If the argument to convert() is already an instance of its own class, it is returned without any change.

The implementation of the add(), subtract(), multiply(), divide(), modulo(), and equals() methods requires that the type of the argument and the implementing class be comparable in the type hierarchy. If this condition is not met, these methods will throw an exception. If the type of the argument is lower than the type of the implementing class, then the argument is converted to the type of the implementing class before the operation is carried out.

The implementation is more involved if the type of the argument is higher than the implementing class, in which case, the conversion must be done in the other direction. Since the `convert()` method only knows how to convert types lower in the type hierarchy up, the operation must take place in the class of the argument. Furthermore, since many of the supported operations are not commutative, for example, `"Hello" + "world"` is not the same as `"world" + "Hello"`, and 3-2 is not the same as 2-3, the implementation of the arithmetic operations cannot simply call the same method on the class of the argument. Instead, a separate set of methods must be used. These methods are addReverse(), subtractReverse(), multiplyReverse(), divideReverse(), and moduloReverse(). The equality check is always commutative so no equalsReverse() is needed. Under this setup, a.add(b) means $a+b$, and a.addReverse(b) means $b+a$, where a and b are both tokens. If, for example, when a.add(b) is invoked and the type of b is higher than a, the add() method of a will automatically call b.addReverse(a) to carry out the addition.

For scalar and matrix tokens, methods are also provided to convert the content of the token into another numeric type. In ScalarToken, these methods are intValue(), longValue(), doubleValue(), fixValue(), and complexValue(). In MatrixToken, the methods are intMatrix(), longMatrix(), doubleMatrix(), fixMatrix(), and complexMatrix(). The default implementation in these two base classes just throw an exception. Derived classes override the methods if the corresponding conversion is lossless, returning a new instance of the appropriate class. For example, IntToken overrides all the methods defined in ScalarToken, but DoubleToken does not override intValue(). A double cannot, in general, be losslessly converted to an integer.

4.3.3 Limitations

As of this writing, the following issues remain open:

- For numerical matrix tokens, only the add() and addReverse() methods are supported; other arithmetic operations are not implemented yet.

4.4 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* is identical to a variable, but realized by instances of the Parameter class, which is derived from Variable and adds no functionality. See figure 4.3.

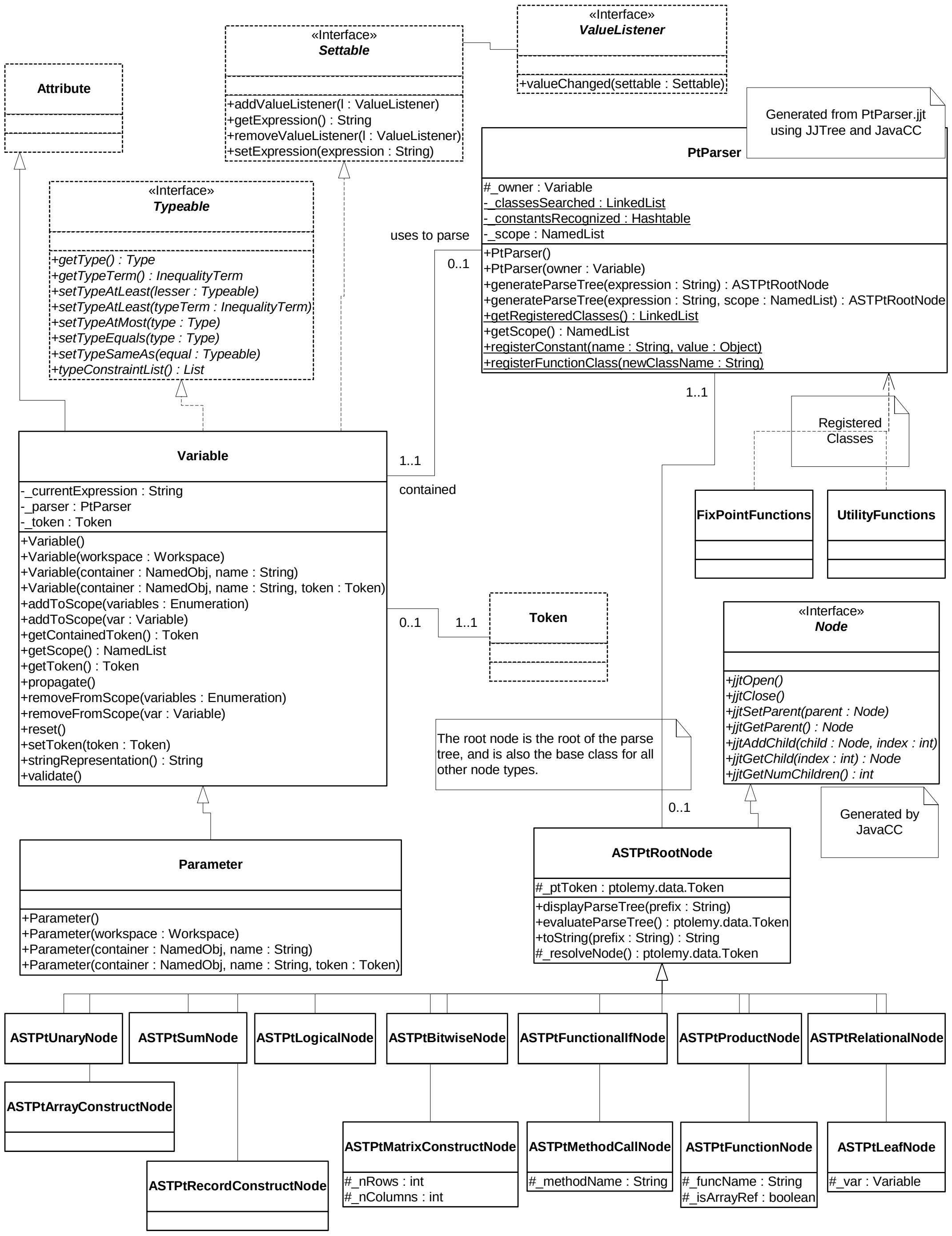

«Interface»
Settable
+addValueListener(l : ValueListener)
+getExpression() : String
+removeValueListener(l : ValueListener)
+setExpression(expression : String)
«Interface»
ValueListener
+valueChanged(settable : Settable)
Attribute
Generated from PtParser.jjt using JJTree and JavaCC
PtParser
#_owner : Variable
-_classesSearched : LinkedList
-_constantsRecognized : Hashtable
-_scope : NamedList
+PtParser()
+PtParser(owner : Variable)
+generateParseTree(expression : String) : ASTPtRootNode
+generateParseTree(expression : String, scope : NamedList) : ASTPtRootNode
+getRegisteredClasses() : LinkedList
+getScope() : NamedList
+registerConstant(name : String, value : Object)
+registerFunctionClass(newClassName : String)
«Interface»
Typeable
+getType() : Type
+getTypeTerm() : InequalityTerm
+setTypeAtLeast(lesser : Typeable)
+setTypeAtLeast(typeTerm : InequalityTerm)
+setTypeAtMost(type : Type)
+setTypeEquals(type : Type)
+setTypeSameAs(equal : Typeable)
+typeConstraintList() : List
uses to parse
0..1
1..1
contained
1..1
Registered Classes
Variable
-_currentExpression : String
-_parser : PtParser
-_token : Token
+Variable()
+Variable(workspace : Workspace)
+Variable(container : NamedObj, name : String)
+Variable(container : NamedObj, name : String, token : Token)
+addToScope(variables : Enumeration)
+addToScope(var : Variable)
+getContainedToken() : Token
+getScope() : NamedList
+getToken() : Token
+propagate()
+removeFromScope(variables : Enumeration)
+removeFromScope(var : Variable)
+reset()
+setToken(token : Token)
+stringRepresentation() : String
+validate()
FixPointFunctions
UtilityFunctions
0..1
1..1
Token
«Interface»
Node
+jjtOpen()
+jjtClose()
+jjtSetParent(parent : Node)
+jjtGetParent() : Node
+jjtAddChild(child : Node, index : int)
+jjtGetChild(index : int) : Node
+jjtGetNumChildren() : int
The root node is the root of the parse tree, and is also the base class for all other node types.
0..1
Generated by JavaCC
Parameter
+Parameter()
+Parameter(workspace : Workspace)
+Parameter(container : NamedObj, name : String)
+Parameter(container : NamedObj, name : String, token : Token)
ASTPtRootNode
#_ptToken : ptolemy.data.Token
+displayParseTree(prefix : String)
+evaluateParseTree() : ptolemy.data.Token
+toString(prefix : String) : String
#_resolveNode() : ptolemy.data.Token
ASTPtUnaryNode
ASTPtSumNode
ASTPtLogicalNode
ASTPtBitwiseNode
ASTPtFunctionalIfNode
ASTPtProductNode
ASTPtRelationalNode
ASTPtArrayConstructNode
ASTPtMatrixConstructNode
#_nRows : int
#_nColumns : int
ASTPtMethodCallNode
#_methodName : String
ASTPtFunctionNode
#_funcName : String
#_isArrayRef : boolean
ASTPtLeafNode
#_var : Variable
ASTPtRecordConstructNode

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

The reason for having two classes with identical interfaces and functionality, Variable and Parameter, is that their intended uses are different. Parameters are meant to be visible to the end user of a component, whereas variables are meant to operate behind the scenes, unseen. A GUI, for example, might present parameters for editing, but not variables.

4.4.1 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 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.

A variable can also be reset. If the variable was originally set from a token, then this token is placed again in the variable, and the type of the variable is set to equal that of the token. If the variable was originally given an expression, then this expression is placed again in the variable (but not evaluated), and the type is reset to null. The type will be determined when the expression is evaluated or when type resolution is done.

4.4.2 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. You can constrain the allowable types of a parameter or variable using the following mechanisms:

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

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().

4.4.3 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.

4.5 Expressions

Ptolemy II includes a simple but extensible expression language. This language permits operations on tokens to be specified in a scripting fashion, without requiring compilation of Java code. The expression language can be used to define parameters in terms of other parameters, for example. It can also be used to provide end-users with actors that compute a user-specified expression that refers to inputs and parameters of the actor.

4.5.1 The Ptolemy II Expression Language

Arithmetic operators. The arithmetic operators are +, –, *, /, ^, and %. Most of these operators, along with ==, are overloaded[1], so their implementation depends on the types being operated on. Operator overloading is achieved using the methods in the Token classes. These methods are add(), subtract(), multiply(), divide(), modulo(), and equals(). The ^ operator computes “to the power of” where the power can only be an integer.

Bit manipulation. The bitwise operators are &, |, and ~. They operate on integers, where & is bitwise and, ~ is bitwise not, and | is bitwise or.

Relational operators. The relational operators are <, <=, >, >=, == and !=. They return booleans.

Logical operators. The logical boolean operators are &&, ||, !, & and |. They operate on booleans and return booleans. Note that the difference between logical && and logical & is that & evaluates all the operands regardless of whether their value is now irrelevant. Similarly for logical || and |. This

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.

approach is borrowed from Java.

Conditionals. The language is an expression language, not an imperative language with sequentially executed statements. Thus, it makes no sense to have the usual `if...then...else...` construct. Such a construct in Java (and most imperative languages) depends on side effects. However, Java does have a functional version of this construct (one that returns a value). The syntax for this is

```
    boolean ? value1 : value2
```

If the boolean is true, `value1` is returned, else `value2` is returned. The Ptolemy II expression language uses this same syntax.

Comments. Anything inside **/*...*/** is ignored.

Variables. Expressions can contain references by name to parameters within the ***scope*** of the expression. Consider a parameter *P* with container *X* which is in turn contained by *Y*. The scope of an expression for *P* includes all the parameters contained by *X* and *Y*, plus those of the container of *Y*, its container, etc. The scope is implemented as an instance of NamedList, which provides a symbol table. Note that a class derived from Parameter may define scope differently.

Constants. If an identifier is encountered in an expression that does not match a parameter in the scope, then it might be a constant that has been registered as part of the language. By default, the constants *PI*, *pi*, *E*, *e*, *true*, *false*, *i*, and *j* are registered, but as we will see later, this can easily be extended. (The constants *i* and *j* are complex numbers with value equal to 0.0 + 1.0i). In addition, literal string constants are supported. Anything between quotes, "...", is interpreted as a string constant. Numerical values without decimal points, such as "10" or "-3" are integers. Numerical values with decimal points, such as "10.0" or "3.14159" are doubles. Integers followed by the character "l" (el) or "L" are long integers.

Arrays. Arrays are specified with curly brackets. E.g., "{1, 2, 3}" is an array of integers, while "`{"x", "y", "z"}`" is an array of strings. An array is an ordered list of tokens of any type, with the only constraint being that the elements all have the same type. Thus, for example, "{1, 2.3}" is illegal because the first element is an integer and the second is a double. The elements of the array can be given by expressions, as in the example "{2*pi, 3*pi}." Arrays can be nested; for example, "{{1, 2}, {3, 4, 5}}" is an array of arrays of integers.

Matrices. Matrices are specified with square brackets, using commas to separate row elements and semicolons to separate rows. E.g., "[1, 2, 3; 4, 5, 5+1]" gives a two by three integer matrix (2 rows and 3 columns). Note that an array or matrix element can be given by an expression. A row vector can be given as "[1, 2, 3]" and a column vector as "[1; 2; 3]". Some Matlab-style array constructors are supported. For example, "[1:2:9]" gives an array of odd numbers from 1 to 9, and is equivalent to "[1, 3, 5, 7, 9]." Similarly, "[1:2:9; 2:2:10]" is equivalent to "[1, 3, 5, 7, 9; 2, 4, 6, 8, 10]."

Matrix references. Reference to matrices have the form "*name*(*n*, *m*)" where *name* is the name of a matrix variable in scope (or a constant matrix), *n* is the row index, and *m* is the column index. Index numbers start with zero, as in Java, not 1, as in Matlab.

Records. A record token is a composite type where each element is named, and each element can have a distinct type. Records are delimited by curly braces, with each element given a name. For example, "`{a=1, b="foo"}`" is a record with two elements, named "a" and "b", with values 1 (an integer) and "foo" (a string), respectively. The value of a record element can be an arbitrary expression, and records can be nested (an element of a record token may be a record token).

Functions. The language includes an extensible set of functions, such as sin(), cos(), etc. The functions that are built in include all static methods of the java.lang.Math class and the ptolemy.data.expr.UtilityFunctions class. This can easily be extended by registering another class that includes static methods. The functions currently available are shown in figures 4.4 and 4.5, with the argument types and return types[1].

One slightly subtle function is the random() function. It takes no arguments, and hence is written "`random()`". It returns a random number. However, this function is evaluated only when the expression within which it appears is evaluated. The result of the expression may be used repeatedly without re-evaluating the expression. The random() function is not called again. Thus, for example, if the *value* parameter of the Const actor is set to "`random()`", then its output will be a random constant; i.e., it will not change on each firing.

Methods. Every element and subexpression in an expression represents an instance of Token (or more likely, a class derived from Token). The expression language supports invocation of any method of a given token, as long as the arguments of the method are of type Token and the return type is Token (or a class derived from Token, or something that the expression parser can easily convert to a token, such as a string, double, int, etc.). The syntax for this is (*token*).*name*(*args*), where *name* is the name of the method and *args* is a comma-separated set of arguments. Each argument can itself be an expression. Note that the parentheses around the *token* are not required, but might be useful for clarity. As an example, the ArrayToken class has a getElement(int) method, which can be used as follows:

```
{1, 2, 3}.getElement(1)
```

This returns the integer 2. Another useful function of array token is illustrated by the following example:

```
{1, 2, 3}.length()
```

which returns the integer 3.

The MatrixToken classes have three particularly useful methods, illustrated in the following examples:

function	argument type(s)	return type	description
gaussian	double, double	double	Gaussian random variable with the specified mean, and standard deviation
gaussian	double, double, int, int	double matrix	Gaussian random matrix with the specified mean, standard deviation, rows, and columns

FIGURE 4.5. Functions available to the expression language from the ptolemy.data.expr.UtilityFunctions class. This class is still at a preliminary stage, and the function it provides will grow over time.

1. At this time, in release 1.0, the types must match exactly for the expression evaluator to work. Thus, "sin(1)" fails, because the argument to the sin() function is required to be a double.

function	argument type(s)	return type	description
abs	double	double	absolute value
abs	int	int	absolute value
abs	long	long	absolute value
acos	double	double	arc cosine
asin	double	double	arc sine
atan	double	double	arc tangent
atan2	double, double	double	angle of a vector
ceil	double	double	ceiling function
cos	double	double	cosine
exp	double	double	exponential function (e^argument)
floor	double	double	floor function
IEEEremainder	double, double	double	remainder after division
lob	double	double	natural logarithm
max	double, double	double	maximum
max	int, int	int	maximum
max	long, long	long	maximum
min	double, double	double	minimum
min	int, int	int	minimum
min	long, long	long	minimum
pow	double, double	double	first argument to the power of the second
random		double	random number between 0.0 and 1.0
rint	double	double	round to the nearest integer
round	double	long	round to the nearest integer
sin	double	double	sine function
sqrt	double	double	square root
tan	double	double	tangent function
toDegrees	double	double	convert radians to degrees
toRadians	double	double	convert degrees to radians

FIGURE 4.4. Functions available to the expression language from the java.lang.Math class.

```
[1, 2; 3, 4; 5, 6].getRowCount()
```

which returns 3, and

```
[1, 2; 3, 4; 5, 6].getColumnCount()
```

which returns 2, and

```
[1, 2; 3, 4; 5, 6].toArray()
```

which returns {1, 2, 3, 4, 5, 6}. The latter function can be particularly useful for creating arrays using Matlab-style syntax. For example, to obtain an array with the integers from 1 to 100, you can enter:

```
[1:1:100].toArray()
```

The get() method of RecordToken accesses a record field, as in the following example:

```
{a=1, b=2}.get("a")
```

which returns 1.

Types. The types currently supported in the language are boolean, complex, fixed point, double, int, long, arrays, matrices, records, and string. Note that there is no float or byte. Use double or int instead. A long is defined by appending an integer with “l” (lower case L) or “L”, as in Java. A complex is defined by appending an “i” or a “j” to a double for the imaginary part. This gives a purely imaginary complex number which can then leverage the polymorphic operations in the Token classes to create a general complex number. Thus “`2 + 3i`” will result in the expected complex number. A fixed point number is defined using the “fix” function, as will be explained below in section 4.6.4.

The Token classes from the data package form the primitives of the language. For example the number 10 becomes an IntToken with the value 10 when evaluating an expression. Normally this is invisible to the user. The expression language is object-oriented, of course, so methods can be invoked on these primitives. A sophisticated user, therefore, can make use of the fact that “10” is in fact an object to invoke methods of that object.

In particular, the convert() method of the Token class might be useful, albeit a bit subtle in how it is used. For example:

```
(1.2).convert(1)
```

creates a DoubleToken with value 1.2, and then invokes its convert() method with argument 1, which is an IntToken. The convert() method of DoubleToken converts the argument to a DoubleToken, so the result of this expression is (somewhat surprisingly) 1.0. The convert() method supports only lossless type conversion (see section 4.3.2). Lossy conversion has to be done explicitly via a function call.

The expression language is extensible. The basic mechanism for extension is object-oriented. The reflection package in Java is used to recognize method invocations and user-defined constants. We also expect the language to grow over time, so this description should be viewed as a snapshot of its capabilities.

4.5.2 Limitations

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

- The class ptolemy.data.util.UtilityFunctions containing the utility functions has not yet been fully written.
- Functions in the math package need to be supported in much the same way that java.lang.Math is supported.
- Method calls are currently only allowed on tokens in the ptolemy.data package.
- Statements are not supported. It is not clear that they ever will be, since currently the expression language is strictly functional, and converting it to imperative semantics could drastically change its flavor.

4.6 Fixed Point Data Type

Ptolemy II includes a preliminary fixed point data type. The FixPoint class in the math package represents fixed point numbers. The FixToken class encapsulates fixed point data for exchange between Ptolemy II actors. The precision of fixed point data is denoted 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.

We represent a fixed point value in the expression language using the following format:

```
fix(value, integerBits, fractionBits)
```

Thus, a fixed point value of 5.375 that uses 8 bit precision of which 4 bits are used to represent the integer part can be represented as:

```
fix(5.375, 8, 4)
```

These functions are implemented by the FixPointFunctions class in the ptolemy.data.expr package. The value can also be a matrix of doubles. The values are rounded, yielding the nearest value representable with the specified precision. If the value to represent is out of range, then it is saturated, meaning that the maximum or minimum fixed point value is returned, depending on the sign of the specified value. For example,

```
fix(5.375, 8, 3)
```

will yield 3.968758, the maximum value possible with the (8/3) precision.

In addition to the fix() function, the expression language offers a quantize() function. The arguments are the same as those of the fix() function, but the return type is a DoubleToken or DoubleMatrixToken instead of a FixToken or FixMatrixToken. This function can therefore be used to quantize double-precision values without ever explicitly working with the fixed-point representation.

4.6.1 FixPoint Implementation

We will now discuss how the FixPoint data type is implemented in Ptolemy II, and how it interacts with the Token types and expression parser. The overall UML diagram showing classes involved in the definition of the FixPoint data type is shown in Figure 4.6

4.6.2 FixPoint

The FixPoint type is written from scratch and it uses at it's core the Java package *BigInteger* to represent the finite precision value that is captured in a FixPoint. 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 FixPoint data type uses an innerclass to represent the BigInteger. The innerclass is used to keep track of errors as they may occur. These errors are that an overflow or rounding condition occurred. The innerclass keeps the BigInteger and error messages together. Besides the BigInteger package, the FixPoint class also relies on the *BigDecimal* package when converting values from FixPoints to doubles and vice versa.

The precision used in the FixPoint data type is represented by class Precision. 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.

A FixPoint is created by supplying a BigInteger and a Precision. This seems to be an odd way of creating FixPoints. That is because the preferred way to create a FixPoint is to use one of the static quantizer functions in class *Quantizer*. By selecting either the *round* or the *truncate* method, a different quantizer is chosen to convert a double into a FixPoint.

To change the precision of a FixPoint, you have to use the specific implementation of round and truncate. If the change of precision can be accommodated, the FixPoint value isn't changed. If the change cannot be accommodated, then precision is changed and an overflow or quantization error may occur. The way the overflow error is handled is determined by a mode switch.

mode = 0, **Saturate**: The fixed point value is set, depending on its sign, equal to the Maximum or Minimum value possible with the new given precision.

mode = 1, **Zero Saturate**: The fixed point value is set equal to zero.

4.6.3 FixToken

A FixToken is realized by encapsulating a value of the FixPoint type and by implementing all methods of super class Token using the methods available for FixPoint. Because FixToken is derived from Token and ScalarToken, it can consequently be used in every data type polymorphic actors. In a similar way data type FixMatrixToken is created. It encapsulates an two-dimensional array of fixed point values.

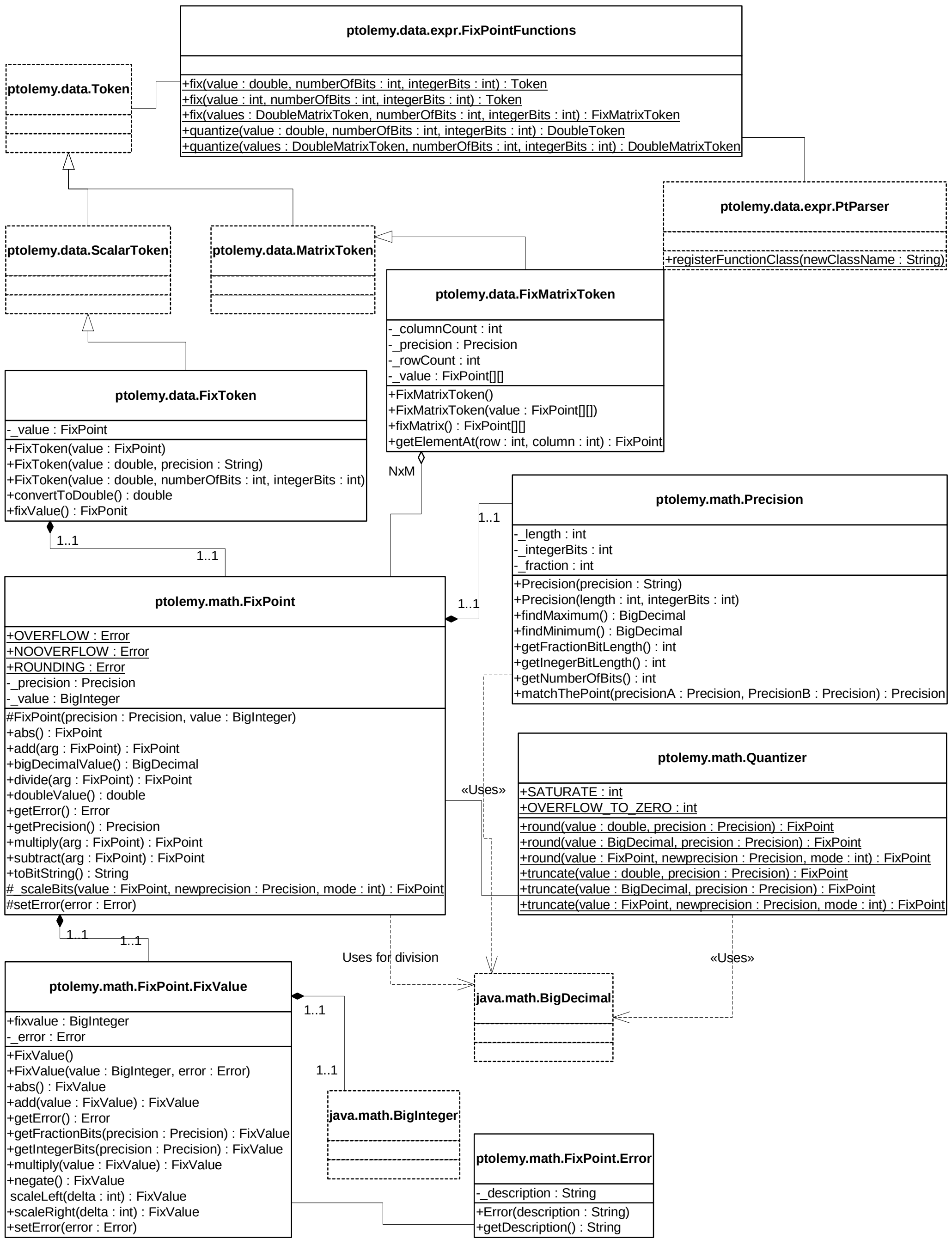

ptolemy.data.expr.FixPointFunctions
+fix(value : double, numberOfBits : int, integerBits : int) : Token
+fix(value : int, numberOfBits : int, integerBits : int) : Token
+fix(values : DoubleMatrixToken, numberOfBits : int, integerBits : int) : FixMatrixToken
+quantize(value : double, numberOfBits : int, integerBits : int) : DoubleToken
+quantize(values : DoubleMatrixToken, numberOfBits : int, integerBits : int) : DoubleMatrixToken
ptolemy.data.Token
ptolemy.data.expr.PtParser
+registerFunctionClass(newClassName : String)
ptolemy.data.ScalarToken
ptolemy.data.MatrixToken
ptolemy.data.FixMatrixToken
-_columnCount : int
-_precision : Precision
-_rowCount : int
-_value : FixPoint[][]
+FixMatrixToken()
+FixMatrixToken(value : FixPoint[][])
+fixMatrix() : FixPoint[][]
+getElementAt(row : int, column : int) : FixPoint
ptolemy.data.FixToken
-_value : FixPoint
+FixToken(value : FixPoint)
+FixToken(value : double, precision : String)
+FixToken(value : double, numberOfBits : int, integerBits : int)
+convertToDouble() : double
+fixValue() : FixPonit
NxM
1..1
1..1
ptolemy.math.Precision
-_length : int
-_integerBits : int
-_fraction : int
+Precision(precision : String)
+Precision(length : int, integerBits : int)
+findMaximum() : BigDecimal
+findMinimum() : BigDecimal
+getFractionBitLength() : int
+getInegerBitLength() : int
+getNumberOfBits() : int
+matchThePoint(precisionA : Precision, PrecisionB : Precision) : Precision
1..1
1..1
ptolemy.math.FixPoint
+OVERFLOW : Error
+NOOVERFLOW : Error
+ROUNDING : Error
-_precision : Precision
-_value : BigInteger
#FixPoint(precision : Precision, value : BigInteger)
+abs() : FixPoint
+add(arg : FixPoint) : FixPoint
+bigDecimalValue() : BigDecimal
+divide(arg : FixPoint) : FixPoint
+doubleValue() : double
+getError() : Error
+getPrecision() : Precision
+multiply(arg : FixPoint) : FixPoint
+subtract(arg : FixPoint) : FixPoint
+toBitString() : String
#_scaleBits(value : FixPoint, newprecision : Precision, mode : int) : FixPoint
#setError(error : Error)
«Uses»
ptolemy.math.Quantizer
+SATURATE : int
+OVERFLOW_TO_ZERO : int
+round(value : double, precision : Precision) : FixPoint
+round(value : BigDecimal, precision : Precision) : FixPoint
+round(value : FixPoint, newprecision : Precision, mode : int) : FixPoint
+truncate(value : double, precision : Precision) : FixPoint
+truncate(value : BigDecimal, precision : Precision) : FixPoint
+truncate(value : FixPoint, newprecision : Precision, mode : int) : FixPoint
1..1
1..1
Uses for division
«Uses»
ptolemy.math.FixPoint.FixValue
+fixvalue : BigInteger
-_error : Error
+FixValue()
+FixValue(value : BigInteger, error : Error)
+abs() : FixValue
+add(value : FixValue) : FixValue
+getError() : Error
+getFractionBits(precision : Precision) : FixValue
+getIntegerBits(precision : Precision) : FixValue
+multiply(value : FixValue) : FixValue
+negate() : FixValue
scaleLeft(delta : int) : FixValue
+scaleRight(delta : int) : FixValue
+setError(error : Error)
1..1
1..1
java.math.BigDecimal
java.math.BigInteger
ptolemy.math.FixPoint.Error
-_description : String
+Error(description : String)
+getDescription() : String

FIGURE 4.6. Organization of the FixPoint Data Type.

The FixToken class implements all the methods of Token and ScalarToken. However, one specific methods has been added: *convertToDouble.* The convertToDouble method converts a fixed point value into a double representation. The *getDouble* method defined by Token cannot be used since the conversion from a FixPoint to a double is not lossless and an exception will be thrown when tried.

4.6.4 Expression Language

To make the FixToken accessible within the expression language, we have added a number of fixed point specific utility functions. These functions have been registered with the expression parser, using its function registration mechanism.

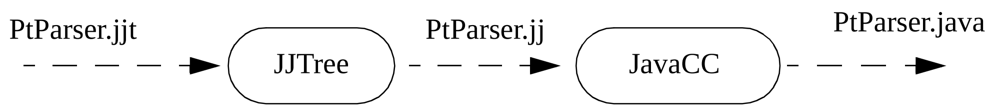
- To create a single FixPoint Token using the expression language:
  ```
  fix(5.34, 10, 4)
  ```
  This will create a FixToken. In this case, we try to fit the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.
- To create a Matrix with FixPoint values using the expression language:
  ```
  fix([ -.040609, -.001628, .17853 ], 10,  2)
  ```
  This will create a FixMatrixToken with 1 row and 3 columns, in which each element is a FixPoint value with precision (10/2). The resulting FixMatrixToken will try to fit each element of the given double matrix into a 10 bit representation with 2 bits used for the integer part. It uses by default the round quantizer.
- To create a single DoubleToken, which is the quantized version of the double value given, using the expression language:
  ```
  quantize(5.34, 10, 4)
  ```
  This will create a DoubleToken. The resulting DoubleToken contains the double value obtained by fitting the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.
- To create a Matrix with doubles quantized to a particular precision using the expression language:
  ```
  quantize([ -.040609, -.001628, .17853 ], 10,  2)
  ```
  This will create a DoubleMatrixToken with 1 row and 3 columns. The elements of the token are obtained by fitting the given matrix elements into a 10 bit representation with 2 bits used for the integer part. Instead of being a fixed point value, the values are converted back to their double representation and by default the round quantizer is used.

Appendix B: 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.

B.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 an AST. The parser definition is stored in the file PtParser.jjt, and the generated file is PtParser.java. Thus the procedure is



Note that JavaCC generates top-down parsers, or LL(k) in parser terminology. This is different from yacc (or bison) which 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.

The parser takes as input a string, and optionally a NamedList of parameters to which the input expression can refer. That NamedList is the symbol table. If the parse is successful, it returns the root node of the abstract syntax tree (AST) for the given string. Each node object can contain a token, which represents both the type and value information for that node. The type of the token stored in a node, e.g. DoubleToken, IntToken etc., represents the type of the node. The data value contained by the token is the value information for the node. In the AST as it is returned from PtParser, the token types and values are only resolved for the leaf nodes of the tree.

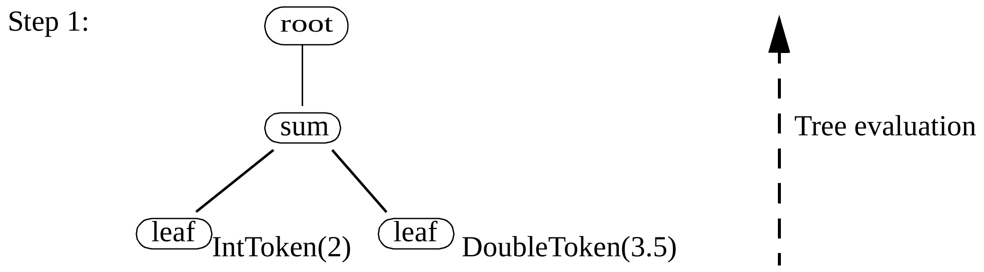
One of the key properties of the expression language is the ability to refer to other parameters by name. Since an expression that refers to other parameters may need to be evaluated several times (when the referred parameter changes), it is important that the parse tree does not need to be recreated every time. When an identifier is parsed, the parser first checks whether it refers to a parameter within the current scope. If it does it creates a ASTPtLeafNode with a reference to that parameter. Note that a leaf node can have a parameter or a token. If it has a parameter then when the token to be stored in this node is evaluated, it is set to the token contained by the parameter. Thus the AST tree does not need to be recreated when a referenced parameter changes as upon evaluation it will just get the new token stored in the referenced parameter. If the parser was created by a parameter, the parameter passes in a reference to itself in the constructor. Then upon parsing a reference to another parameter, the parser takes care of registering the parameter that created it as a listener with the referred parameter. This is how dependencies between parameters get registered. There is also a mechanism built into parameters to detect dependency loops.

If the identifier does not refer to a parameter, the parser then checks if it refers to a constant registered with the parser. If it does it creates a node with the token associated with the identifier. If the identifier is neither a reference to a parameter or a constant, an exception is thrown.

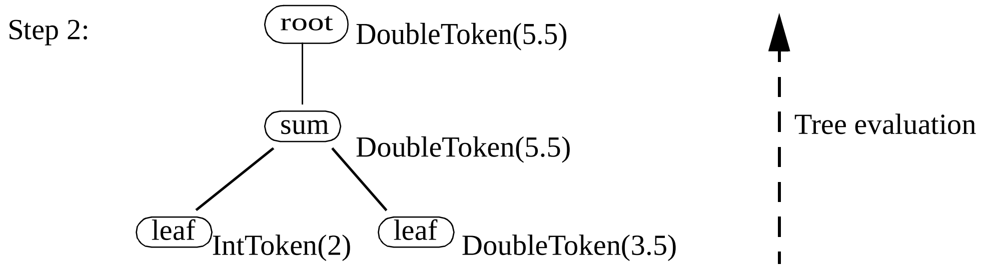
B.2 Evaluating the parse tree

The AST can be evaluated by invoking the method evaluateParseTree() on the root node. The AST is evaluated in a bottom up manner as each node can only determine its type after the types of all its children have been resolved. When the type of the token stored in the root node has been resolved, this token is returned as the result of evaluating the parse tree.

As an example consider the input string 2 + 3.5. The parse tree returned from the parser will look like this:



which will then get evaluated to this:



and DoubleToken(5.5) will be returned as the result.

As seen in the above example, when evaluateParseTree() is invoked on the root node, the type and value of the tokens stored at each node in the tree is resolved, and finally the token stored in the root node is returned. If an error occurs during either the creation of the parse tree or the evaluation of the parse tree, an IllegalArgumentException is thrown with a error message about where the error occurred.

If a node has more than two children, type resolution is done pairwise from the left. Thus “2 + 3 + “hello”” resolves to 5hello. This is the same approach that Java follows.

Each time the parser encounters a function call, it creates an ASTPtFunctionNode. When this node is being evaluated, it uses reflection to look for that function in the list of classes registered with the parser for that purpose. The classes automatically searched are java.lang.Math and ptolemy.data.expr.UtilityFunctions. To register another class to be searched when a function call is parsed, call registerFunctionClass() on the parser with the full name of the class to be added to the function search path.

When a parameter has been informed that another parameter it references changed, the parameter re-evaluates the parse tree for the expression to obtain the new value when getToken() is called on the parameter. It is not necessary to parse the expression again as the relevant leaf node stores a reference to the referenced parameter, not the token contained in that parameter. Thus at any use, the value of a parameter is up to date.

B.2.1 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.

ASTPtBitwiseNode. This is created when a bitwise operation (&, |, ^) happens. Type resolution occurs in the node. The & and | operators are only valid between two booleans, or two integer types. The ^ operator is only valid between two integer types.

ASTPtLeafNode. This represents the leaf nodes in the AST. The parser will always place either a token of the appropriate type (e.g. IntToken if “2” is what is parsed) or a parameter in a leaf node. A parameter is placed so that the parse tree can be reevaluated without reparsing whenever the value of the parameter changes. No type resolution is necessary in this node.

ASTPtRootNode. Parent class of all the other nodes. As its name suggests, it is the root node of the AST. It always has only one child, and its type and value is that of its child.

ASTPtFunctionNode. This is created when a function is called. Type resolution occurs in the node. It uses reflection to call the appropriate function with the arguments supplied. It searches the classes registered with the parser for the function. By default it only looks in java.lang.Math and ptolemy.data.expr.UtilityFunctions.

ASTPtFunctionalIfNode. This is created when a functional if is parsed. Type resolution occurs in the node. For a functional if, the first child node must contain a BooleanToken, which is used to chose which of the other two tokens of the child nodes to store at this node.

ASTPtMethodCallNode. This is created when a method call is parsed. Method calls are currently only allowed on tokens in the ptolemy.data package. All of the arguments to the method, and the return type, must be of type Token (or a subclass).

ASTPtProductNode. This is created when a *, / or % is parsed. Type resolution does not occur in the node. It uses the multiply(), divide() and modulo() methods in the token classes to resolve the nodes type.

ASTPtSumNode. This is created when a + or - is parsed. Type resolution does not occur in the node. It uses the add() and subtract() methods in the token classes to resolve the nodes type.

ASTPtLogicalNode. This is created when a && or || is parsed. Type resolution occurs in the node. All children nodes must have tokens of type BooleanToken. The resolved type of the node is also BooleanToken.

ASTPtRelationalNode. This is created when one of the relational operators(!=, ==, >, >=, <, <=) is parsed. The resolved type of the token of this node is BooleanToken. The "==" and "!=" operators are overloaded via the equals() method in the token classes. The other operators are only valid on ScalarTokens. Currently the numbers are converted to doubles and compared, this needs to be adjusted to take account of Longs.

ASTPtUnaryNode. This is created when a unary negation operator(!, ~, -) is parsed. Type resolution occurs in the node, with the resulting type being the same as the token in the only child of the node.

ASTPtArrayConstructNode. This is created when an array construction sub-expression is parsed.

ASTPtMatrixConstructNode. This is created when a matrix construction sub-expression is parsed.

ASTPtRecordConstructNode. This is created when a record construct sub-expression is parsed.

B.2.2 Extensibility

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

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

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

5 Graph Package

Authors: *Jie Liu*
Yuhong Xiong

5.1 Introduction

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

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

The graphs constructed using this package are lightweight, designed for fast implementation of complex algorithms more than for generality. This makes them maximally complementary to the clustered graphs of the Ptolemy II kernel, which emphasize generality. A typical use of this package is to construct a graph that represents the topology of a CompositeEntity, run a graph algorithm, and extract useful information from the result. For example, a graph might be constructed that represents data precedences, and a topological sort might be used to generate a schedule. In this kind of application, the hierarchy of the original clustered graph is flattened, so nodes in the graph represent only opaque entities.

The architecture of this package is somewhat different from LEDA, in part because of the existence of the complementary kernel package. Unlike LEDA, there are no dedicated classes representing nodes and edges in the graph. The nodes in this package are represented by arbitrary instances of the Java Object class, and the graph topology is stored in a structure similar to an adjacency list.

The facilities that currently exist in this package are those that we have had most immediate need for. Since the type system of Ptolemy II requires extensive operations on lattices and CPOs, support for these is better developed than for other types of graphs.

5.2 Classes and Interfaces in the Graph Package

Figure 5.1 shows the class diagram of the graph package. The classes Graph, DirectedGraph and DirectedAcyclicGraph support graph construction and provide graph algorithms. Currently, only topological sort and transitive closure are implemented; other algorithms will be added as needed. The CPO interface defines the basic CPO operations, and the class DirectedAcyclicGraph implements this interface. An instance of DirectedAcyclicGraph is also a finite CPO where all the elements and order relations are explicitly specified. Defining the CPO operations in an interface allows future expansion to support infinite CPOs and finite CPOs where the elements are not explicitly enumerated. The Ine-

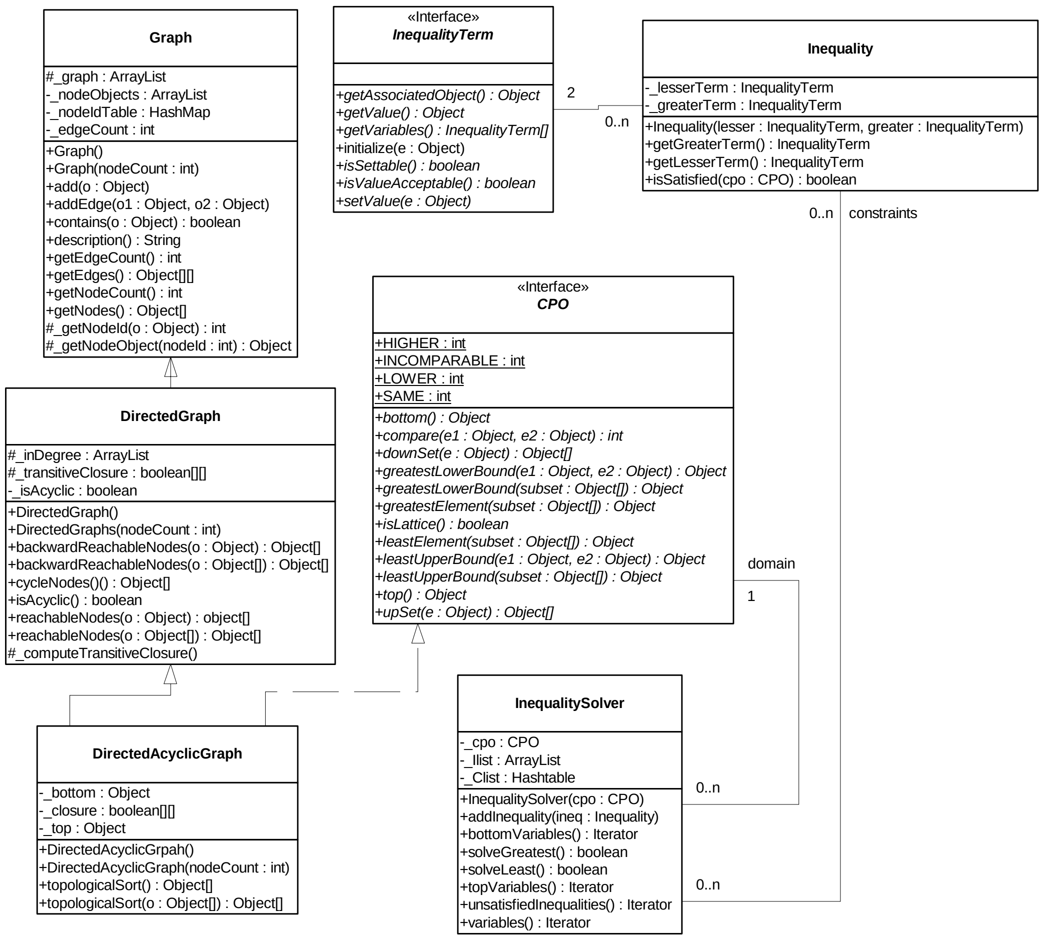

Graph
#_graph : ArrayList
-_nodeObjects : ArrayList
-_nodeIdTable : HashMap
-_edgeCount : int
+Graph()
+Graph(nodeCount : int)
+add(o : Object)
+addEdge(o1 : Object, o2 : Object)
+contains(o : Object) : boolean
+description() : String
+getEdgeCount() : int
+getEdges() : Object[][]
+getNodeCount() : int
+getNodes() : Object[]
#_getNodeId(o : Object) : int
#_getNodeObject(nodeId : int) : Object
«Interface»
InequalityTerm
+getAssociatedObject() : Object
+getValue() : Object
+getVariables() : InequalityTerm[]
+initialize(e : Object)
+isSettable() : boolean
+isValueAcceptable() : boolean
+setValue(e : Object)
2
0..n
Inequality
-_lesserTerm : InequalityTerm
-_greaterTerm : InequalityTerm
+Inequality(lesser : InequalityTerm, greater : InequalityTerm)
+getGreaterTerm() : InequalityTerm
+getLesserTerm() : InequalityTerm
+isSatisfied(cpo : CPO) : boolean
0..n
constraints
«Interface»
CPO
+HIGHER : int
+INCOMPARABLE : int
+LOWER : int
+SAME : int
+bottom() : Object
+compare(e1 : Object, e2 : Object) : int
+downSet(e : Object) : Object[]
+greatestLowerBound(e1 : Object, e2 : Object) : Object
+greatestLowerBound(subset : Object[]) : Object
+greatestElement(subset : Object[]) : Object
+isLattice() : boolean
+leastElement(subset : Object[]) : Object
+leastUpperBound(e1 : Object, e2 : Object) : Object
+leastUpperBound(subset : Object[]) : Object
+top() : Object
+upSet(e : Object) : Object[]
domain
1
DirectedGraph
#_inDegree : ArrayList
#_transitiveClosure : boolean[][]
-_isAcyclic : boolean
+DirectedGraph()
+DirectedGraphs(nodeCount : int)
+backwardReachableNodes(o : Object) : Object[]
+backwardReachableNodes(o : Object[]) : Object[]
+cycleNodes()() : Object[]
+isAcyclic() : boolean
+reachableNodes(o : Object) : object[]
+reachableNodes(o : Object[]) : Object[]
#_computeTransitiveClosure()
InequalitySolver
-_cpo : CPO
-_Ilist : ArrayList
-_Clist : Hashtable
+InequalitySolver(cpo : CPO)
+addInequality(ineq : Inequality)
+bottomVariables() : Iterator
+solveGreatest() : boolean
+solveLeast() : boolean
+topVariables() : Iterator
+unsatisfiedInequalities() : Iterator
+variables() : Iterator
0..n
0..n
DirectedAcyclicGraph
-_bottom : Object
-_closure : boolean[][]
-_top : Object
+DirectedAcyclicGrpah()
+DirectedAcyclicGraph(nodeCount : int)
+topologicalSort() : Object[]
+topologicalSort(o : Object[]) : Object[]

FIGURE 5.1. Classes in the graph package

qualityTerm interface and the Inequality class model inequality constraints over the CPO. The details of the constraints will be discussed later. The InequalitySolver class provides an algorithm to solve a set of constraints. This is used by the Ptolemy II type system, but other uses may arise.

The implementation of the above classes is not synchronized. If multiple threads access a graph or a set of constraints concurrently, external synchronization will be needed.

5.2.1 Graph

This class models a simple undirected graph. Each node in the graph is represented by an arbitrary Java object. The method add() is used to add a node to the graph, and addEdge() is used to connect two nodes in the graph. The arguments of addEdge() are two Objects representing two nodes already added to the graph. To mirror a topology constructed in the kernel package, multiple edges between two nodes are allowed. Each node is assigned a node ID based on the order the nodes are added. The translation from the node ID to the node Object is done by the _getNodeObject() method, and the translation in the other direction is done by _getNodeId(). Both methods are protected. The node ID is only used by this class and the derived classes, it is not exposed in any of the public interfaces. The topology is stored in the Vector _graph. The indexes of this Vector correspond to node IDs. Each entry of _graph is also a Vector, in which a list of node IDs are stored. When an edge is added by calling addEdge() with the first argument having node ID *i* and the second having node ID *j*, an Integer containing *j* is added to the Vector at the *i*-th entry of _graph. For example, if the graph in figure 5.2(a) is connected using the sequence of calls: addEdge(n0, n1); addEdge(n0, n2); addEdge(n2, n1), where n0, n1, n2 are Objects representing the nodes with IDs 0, 1, 2, respectively, then the data structure will be in the form of 5.2(b).

Note that in this undirected graph, the data format is dependent on the order of the two arguments in the addEdge() calls. Since each edge is stored only once, this data structure is not exactly the same as the adjacency list for undirected graphs, but it is quite similar. This structure is designed to be used by subclasses that model directed graphs, as well as by this base class. If it appears awkward when adding algorithms for undirected graph, a new class that derives from Graph may be added in the future to model undirected graph exclusively, in which case, Graph will provide the basic support for both undirected and directed graphs.

5.2.2 Directed Graphs

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

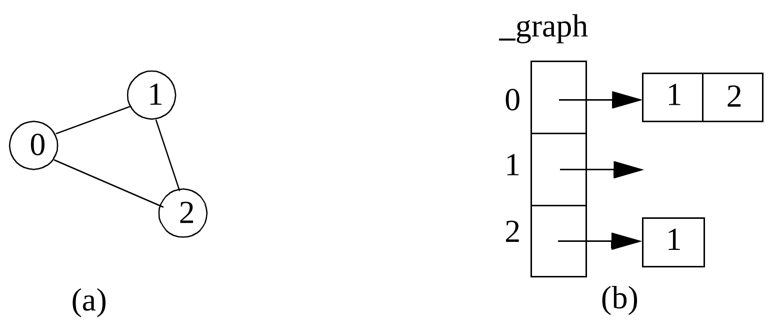


FIGURE 5.2. An undirected graph

a *higher* node, as opposed to from *source* to *sink*, *head* to *tail*, etc. The terms lower and higher conforms with the convention of the graphical representation of CPOs and lattices (the Hasse diagram), so they can be consistently used on both directed graphs and CPOs.

The computation of transitive closure is implemented in this class. The transitive closure is internally stored as a 2-D boolean matrix, whose indexes correspond to node IDs. The entry (i, j) is *true* if and only if there exists a path from the node with ID i to the node with ID j. This matrix is not exposed at the public interface; instead, it is used by this class and its subclass to do other operations. Once the transitive closure matrix is computed, graph operations like *reachableNodes* can be easily accomplished.

5.2.3 Directed Acyclic Graphs and CPO

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

The CPO interface defines the common operations on CPOs. The mathematical definition of these operations can be found in [20]. Informal definitions are given in the class documentation. This interface is implemented by the class DirectedAcyclicGraph.

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

5.2.4 Inequality Terms, Inequalities, and the Inequality Solver

The InequalityTerm interface and Inequality and InequalitySolver classes supports the construction of a set of inequality constraints over a CPO and the identification of a member of the CPO that satisfies the constraints. A constraint is an inequality defined over a CPO, which can involve constants, variables, and functions. As an example, the following is a set of constraints over the 4-point CPO in figure 5.3:

$\alpha \leq w$

$\beta \leq x \wedge \alpha$

$\alpha \leq \beta$

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

An inequality term is either a constant, a variable, or a function over a CPO. The InequalityTerm

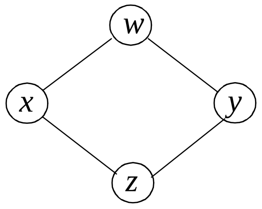


FIGURE 5.3. A 4-point CPO that also happens to be a lattice.

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

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

The InequalitySolver class implements an algorithm to determine satisfiability of a set of inequality constraints and to find the solution to the constraints if they are satisfiable. This algorithm is described in [79]. It is basically an iterative procedure to update the value of variables until all the constraints are satisfied, or until conflicts among the constraints are found. Some limitations on the type of constraints apply for the algorithm to work. The method addInequality() adds an inequality to the set of constraints. Two methods solveLeast() and solveGreatest() can be used to solve the constraints. The former tries to find the least solution, while the latter attempts to find the greatest solution. If a solution is found, these methods return *true* and the current value of the variables is the solution. The method unsatisfiedInequalities() returns an enumeration of the inequalities that are not satisfied based on the current value of the variables. It can be used after solveLeast() or solveGreatest() return *false* to find out which inequalities cannot be satisfied after the algorithm runs. The bottomVariables() and topVariables() methods return enumerations of the variables whose current values are the bottom or the top element of the CPO.

5.3 Example Use

5.3.1 Generating A Schedule for A Composite Actor

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

```
Object[] generateSchedule(CompositeActor composite) {
    DirectedAcyclicGraph 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.add(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.contains(higherActor)) {
               dag.addEdge(lowerActor, higherActor);
            }
         }
      }
   }
   return dag.topologicalSort();
}
```

5.3.2 Forming and Solving Constraints over a CPO

The code below uses two classes implementing the InequalityTerm interface. They model constant and variable terms, respectively. The values of these terms are Strings. Inequalities can be formed using these two classes.

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

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

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

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

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

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

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

// A variable InequalityTerm with a String value.
class Variable implements InequalityTerm {

   // Construct a variable InequalityTerm with a null initial value.
      public Variable() {
   }

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

```
    // Return an array containing this variable term.
    public InequalityTerm[] getVariables() {
        InequalityTerm[] variable = new InequalityTerm[1];
        variable[0] = this;
        return variable;
    }

    // Variable terms are settable.
    public boolean isSettable() {
        return true;
    }

    // Set the value of this variable to the specified String.
    // Not checking the type of the specified Object before casting for simplicity.
    public void setValue(Object e) throws IllegalActionException {
        _value = (String)e;
    }

    private String _value = null;
}
```

As a simple example, the following Java class constructs the 4-point CPO of figure 5.3, forms a set of constraints with three inequalities, and solves for both the least and greatest solutions. The inequalities are $a \leq w$; $b \leq a$; $b \leq z$, where w and z are constants in figure 2.3, and a and b are variables.

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

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

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

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

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

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

        // solve for the greatest solution
        satisfied = solver.solveGreatest();

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

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

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

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

        return cpo;
    }
}
```

6 Type System

Authors: *Edward A. Lee*
Yuhong Xiong
Contributors: *Steve Neuendorffer*

6.1 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 [17].

In a computation environment, two kinds of execution errors can occur, trapped errors and untrapped errors. Trapped errors cause the computation to stop immediately, but untrapped errors may go unnoticed (for a while) and later cause arbitrary behavior. Examples of untrapped errors in a general purpose language are jumping to the wrong address, or accessing data past the end of an array. In Ptolemy II, the underlying language Java is quite safe, so errors rarely, if ever, cause arbitrary behavior.[1] However, errors can certainly go unnoticed for an arbitrary amount of time. As an example, figure 6.1 shows an imaginary application where a signal from a source is downsampled, then fed to a fast Fourier transform (FFT) actor, and the transform result is displayed by an actor. Suppose the FFT actor can accept ComplexToken at its input, and the behavior of the DownSample actor is to just pass every

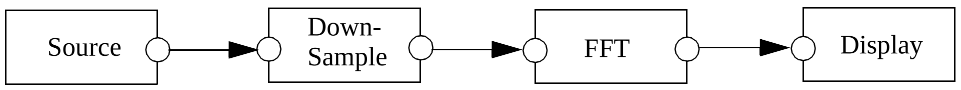


FIGURE 6.1. An imaginary Ptolemy II application

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

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 extensive run-time checking. In Ptolemy II, if we imitated this approach, we would have to require actors to check the type of the received tokens before using them. For example, the FFT actor would have to verify that the every received token is an instance of ComplexToken, or convert it to ComplexToken if possible. This approach gives the burden of type checking to the actor developers, distracting them from their development effort. It also relies on a policy that cannot be enforced by the system. Furthermore, since type checking is postponed to the last possible moment, the system does not have fail-stop behavior, so a system may generate an error only after running for an extended period of time, as figure 6.1 shows. To make things worse, an actor may receive tokens from multiple sources. If a token with the wrong type is received, it might be hard to identify from which source the token comes. All these make debugging difficult.

To address this and other issues discussed later, we added static typing to Ptolemy II. This approach is consistent with Ptolemy Classic. In general-purpose statically-typed languages, such as C++ and Java, static type checking done by the compiler can find a large fraction of program errors. In Ptolemy II, execution of a model does not involve compilation. Nonetheless, static type checking can correspondingly detect problems before any actors fire. In figure 6.1, if the Source actor declares that its output port type is *String*, meaning that it will send out StringTokens upon firing, the static type checker will identify this type conflict in the topology.

In Ptolemy II, because models are not compiled, static typing alone is not enough to ensure type safety at run-time. For example, even if the above Source actor declares its output type to be *Complex*, nothing prevents it from sending out a StringToken at run-time. So run-time type checking is still necessary. With the help of static typing, run-time type checking can be done when a token is sent out from a port. I.e., the run-time type checker checks the token type against the type of the output port. This way, a type error is detected at the earliest possible time, and run-time type checking (as well as static type checking) can be performed by the system instead of by the actors.

One design principle of Ptolemy II is that data type conversions that lose information are not implicitly performed by the system. In the data package, a lossless data type conversion hierarchy, called the type lattice, is defined (see figure 4.2). In that hierarchy, the conversion from a lower type to a higher type is lossless, and is supported by the token classes. This lossless conversion principle also applies to data transfer. This means that across every connection from an output port to an input, the type of the output must be the same as or lower than the type of the input. This requirement is called the type compatibility rule. For example, an output port with type *Int* can be connected to an input port with type *Double*, but a *Double* to *Int* connection will generate a type error during static type checking. This behavior is different from Ptolemy Classic, but it should be useful in many applications where the users do not want lossy conversion to take place without their knowledge.

As can be seen from above examples, when a system runs, the type of a token sent out from an output port may not be the same as the type of the input port the token is sent to. If this happens, the token must be converted to the input port type before it is used by the receiving actor. This kind of run-time type conversion is done transparently by the Ptolemy II system (actors are not aware it). So the actors can safely cast the received tokens to the type of the input port. This makes the actor development easier.

Ousterhout [74] argues that static typing discourages reuse.

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

In Ptolemy II, typing does apply some restrictions on the interaction of actors. Particularly, actors cannot be interconnected arbitrarily if the type compatibility rule is violated. However, the benefit of typing should far outweigh the inconvenience caused by this restriction. In addition, the automatic run-time type conversion provided by the system permits ports of different types to be connected (under the type compatibility rule), which partly relaxes the restriction caused by static typing. Furthermore, there is one important component in Ptolemy that brings much flexibility to the actor interface, the type-polymorphic actors.

Type-polymorphic actors (called polymorphic actors in the rest of this chapter) are actors that can accept multiple types on their ports. For example, the DownSample in figure 6.1 does not care about the type of token going through it; it works with any type of token. In general, the types on some or all of the ports of a polymorphic actor are not rigidly defined to specific types when the actor is written, so the actor can interact with other actors having different types, increasing reusability. In Ptolemy Classic, the ports on polymorphic actors whose types are not specified are said to have ANYTYPE, but Ptolemy II uses the term *undeclared type*, since the type on those ports cannot be arbitrary in general. The acceptable types on polymorphic actors are described by a set of type constraints. The static type checker checks the applicability of a polymorphic actor in a topology by finding specific types for them that satisfy the type constraints. This process is called *type resolution*, and the specific types are called the resolved types.

In addition to ports, Parameters, which are often 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 among ports. This extends the range of type checking to some of the internal states of actors.

Static typing and type resolution have other benefits in addition to the ones mentioned above. Static typing helps to clarify the interface of actors and makes them more manageable. Just as typing may improve run-time efficiency in a general-purpose language by allowing the compiler to generate specialized code, when a Ptolemy system is synthesized to hardware, type information can be used for efficient synthesis. For example, if the type checker asserts that a certain polymorphic actor will only receive IntTokens, then only hardware dealing with integers needs to be synthesized.

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

6.2 Formulation

6.2.1 Type Constraints

In a Ptolemy II topology, the type compatibility rule imposes a type constraint across every connection from an output port to an input port. It requires that the type of the output port, *outType*, be the same as the type of the input port, *inType*, or less than *inType* under the type lattice in figure 4.2. I.e.,

$$outType \le inType \qquad (1)$$

This guarantees that information is not lost during data transfer. If both the *outType* and *inType* are declared, the static type checker simply checks whether this inequality is satisfied, and reports a type conflict if it is not.

In addition to the above constraint imposed by the topology, actors may also impose constraints. This happens when one or both of the *outType* and *inType* is undeclared, in which case the actor containing the undeclared port needs to describe the acceptable types through type constraints. All the type constraints in Ptolemy II are described in the form of inequalities like the one in (1). If a port has a declared type, its type appears as a constant in the inequalities. On the other hand, if a port has an undeclared type, its type is represented by a variable, called the type variable, in the inequalities. The domain of the type variable is the elements of the type lattice. The type resolution algorithm resolves the undeclared types subject to the constraints. If resolution is not possible, a type conflict error will be reported. As an example of the inequality constraints, consider figure 6.2.

The port on actors A1 has declared type *int*; the ports on A3 and A4 have declared type *double*; and the ports on A2 have their types undeclared. Let the type variables for the undeclared types be α, β, and γ, the type constraints from the topology are:

$$\mathit{int} \le \alpha$$
$$\mathit{double} \le \beta$$
$$\gamma \le \mathit{double}$$

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

$$\alpha \le \gamma$$
$$\beta \le \gamma$$
$$\gamma \le \mathit{Complex}$$

The first two inequalities constrain the output precision to be no less than input, the last one requires that the data on the adder ports can be converted to *Complex* losslessly.

These six inequalities form the complete set of constraints and are used by the type resolution algorithm to solve for α, β, and γ.

This inequality formulation is inspired by the type inference algorithm in ML [66]. There, equalities are used to represent type constraints. In Ptolemy II, the lossless type conversion hierarchy naturally implies inequality relation among the types. In ML, the type constraints are generated from program constructs. In a heterogeneous graphical programming environment like Ptolemy II, the system does not have enough information about the function of the actors, so the actors must present their

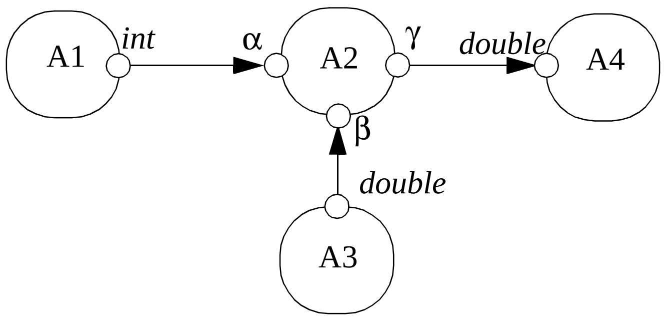


FIGURE 6.2. A topology with types.

type information by either declaring the type on their port, or specify a set of type constraints to describe the acceptable types on the undeclared ports.

This formulation converts type resolution into a problem of solving a set of inequalities. An efficient algorithm is available to solve constraints in finite lattices [79], which is described in the appendix through an example and in figure 6.3. This algorithm finds the set of most specific types for the undeclared types in the topology that satisfy the constraints, if they exist.

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

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

The first case can happen, for example, if the port on actor A1 in figure 6.2 has declared type *Complex*. The second case can happen if an actor does not specify any type constraints on an undeclared output port. This is due to the nature of the type resolution algorithm where it assigns all the undeclared types to *NaT* at the beginning. If the type constraints do not restrict a type variable to be greater than *NaT*, it will stay at *NaT* after resolution. The third case is considered a conflict since an abstract

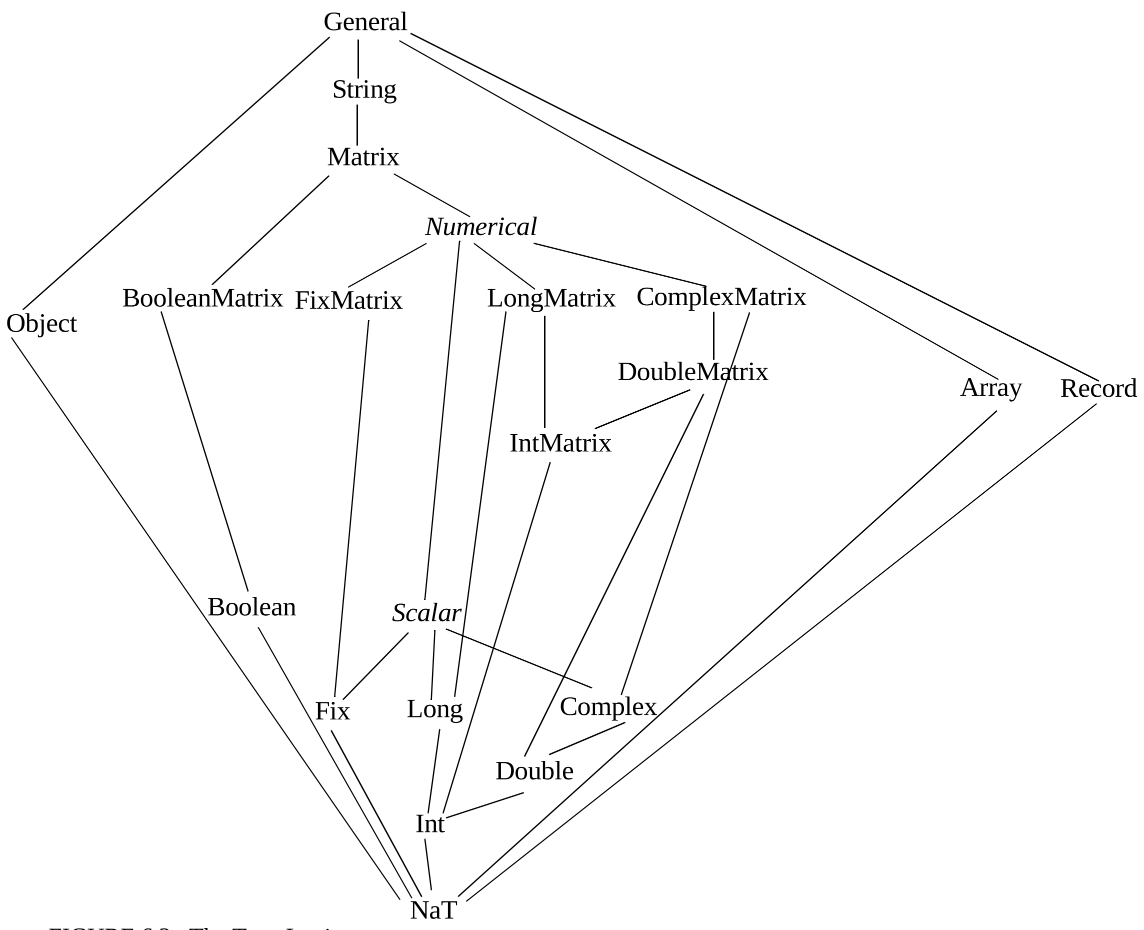


FIGURE 6.3. The Type Lattice

type does not correspond to an instantiable token class.

To avoid the second case above, any output port must either have a declared type, or some constraints to force its type to be greater than *NaT*. This requirement should be easily satisfied on most actors. A situation that needs some attention is the source actor. A source actor cannot leave its output port type unconstrained. One way to cope with this is to declare the type at a time after the type information is known, but prior to type resolution. For example, if the output data is determined by a parameter set by the user, the parameter can be evaluated during the initialization phase of the execution and the port type can be declared at the end of the initialization, which precedes type resolution.

6.2.2 Run-time Type Checking and Lossless Type Conversion

The declared type is a contract between an actor and the Ptolemy II system. If an actor declares an output port to have a certain type, it asserts that it will only send out tokens whose types are less than or equal to that type. If an actor declares an input port to have a certain type, it requires the system to only send tokens that are instances of the class of that type to that input port. Run-time type checking is the component in the system that enforces this contract. When a token is sent out from an output port, the run-time type checker finds its type using the run-time type identification (RTTI) capability of the underlying language (Java), and compares the type with the declared type of the output port. If the type of the token is not less than or equal to the declared type, a run-time type error will be generated.

As discussed before, type conversion is needed when a token sent to an input port has a type less than the type of the input port but is not an instance of the class of that type. Since this kind of lossless conversion is done automatically, an actor can safely cast a received token to the declared type. On the other hand, when an actor sends out tokens, the tokens being sent do not have to have the exact declared output port type. Any type that is less than the declared type is acceptable. For example, if an output port has declared type *double*, the actor can send *IntToken* from that port. As can be seen, the automatic type conversion simplifies the input/output handling of the actors.

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

If an actor has ports with undeclared types, its type constraints can be viewed as both a requirement and an assertion from the actor. The actor requires the resolved types to satisfy the constraints. Once the resolved types are found, they serve the role of declared types at run time. I.e., the type checking and type conversion system guarantees to only put tokens that are instances of the class of the resolved type to input ports, and the actor asserts to only send tokens whose types are less than or equal to the resolved type from output ports.

6.3 Structured Types

Structured types include array and record types. The Array type is implemented by ArrayToken. As described in the Data Package chapter, 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 6.3, array and record types are incomparable with all the base types, except the top and the bottom elements of the lattice. 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* is less than type *A* if the element type of *B* is less than the element type of *A*. This is a recursive definition if the element types are structured types. For example, *Int Array* ≤ *Double Array*, *Int Array Array* ≤ *Double Array Array*, where *Int Array Array* is an array of array. And *Int Array* and *Double Array Array* are incomparable.

The order relation between two record types follow the standard depth subtyping and width subtyping relations [17]. 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:

{name: *String*, value: *Int*} ≤ {name: *String*, value: *Double*}

{name: *String*, value: *Double*, id: *Int*} ≤ {name: *String*, value: *Double*}

Here, we use the {label: type, label: type, ...} syntax to denote record types.

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.

6.4 Implementation

6.4.1 Implementation Classes

All the classes for representing the types and the type lattice are under the data.type package, as shown in figure 6.4. The Type interface defines the basic operations on a type. BaseType contains a type-safe enumeration of all the primitive types. The type UNKNOWN corresponds to the bottom element of the type lattice, it represents a type variable that can be resolved to any type. 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. TypeConstant encapsulate 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 6.5. The container for TypedIOPort must be a ComponentEntity implementing the TypedActor interface, namely, TypedAtomicActor or TypedCompositeActor. The container for TypedAtomicActor and TypedCompositeActor must be a TypedCompositeActor. TypedIORelation constrains that TypedIOPort can only be connected with TypedIOPort. TypedIOPort has a declared type and a resolved type. Undeclared type is represented by BaseType.UNKNOWN. If a port has a declared type that is not BaseType.UNKNOWN, the resolved type will be the same as the declared type.

6.4.2 Type Checking and Type Resolution

Static type checking is done in the checkTypes() method of TypedCompositeActor. This method finds all the connection within the composite by first finding the output ports on deep contained entities, and then finding the deeply connected input ports to those output ports. Transparent ports are ignored for type checking. For each connection, if the types on both ends are declared, static type

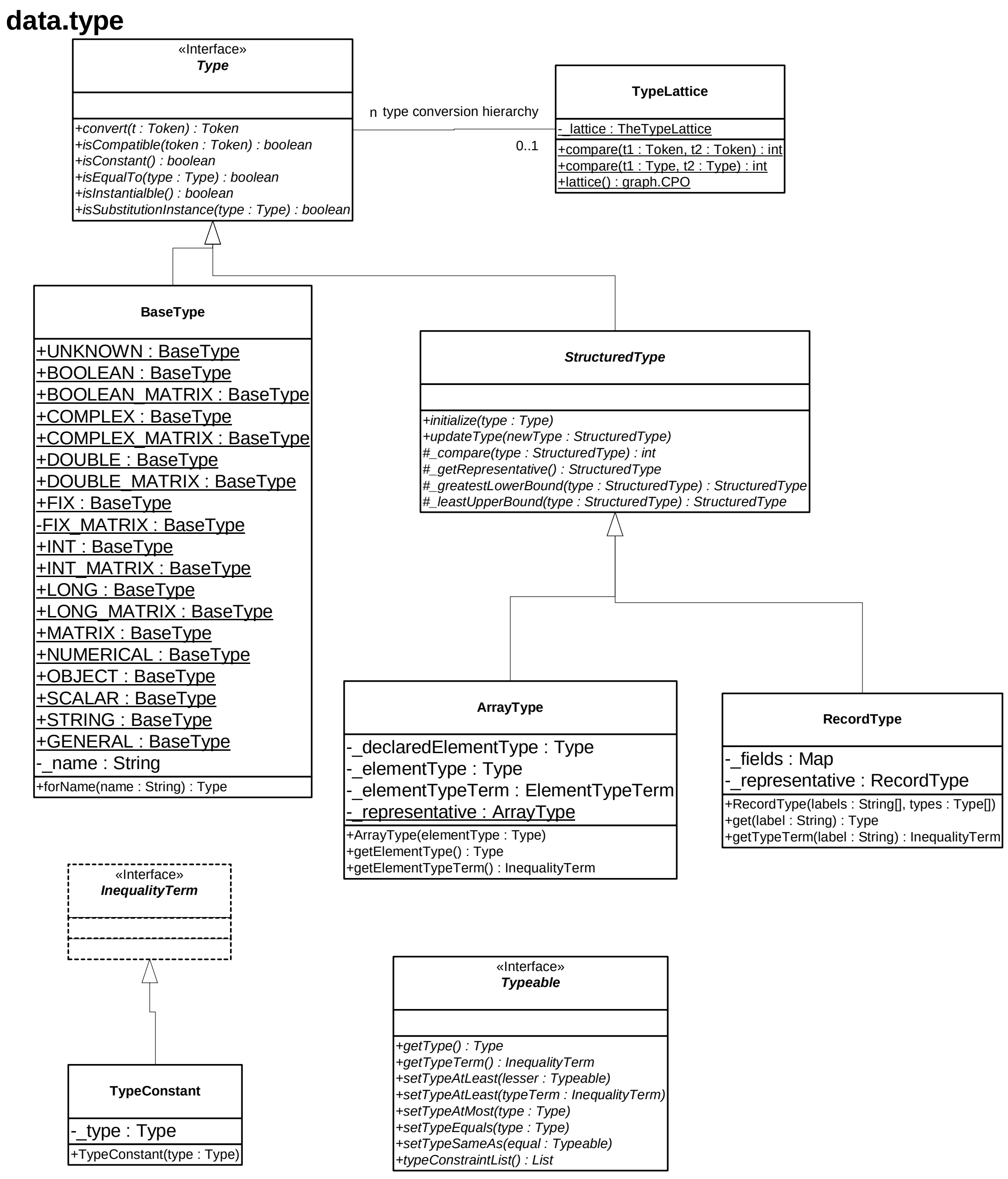
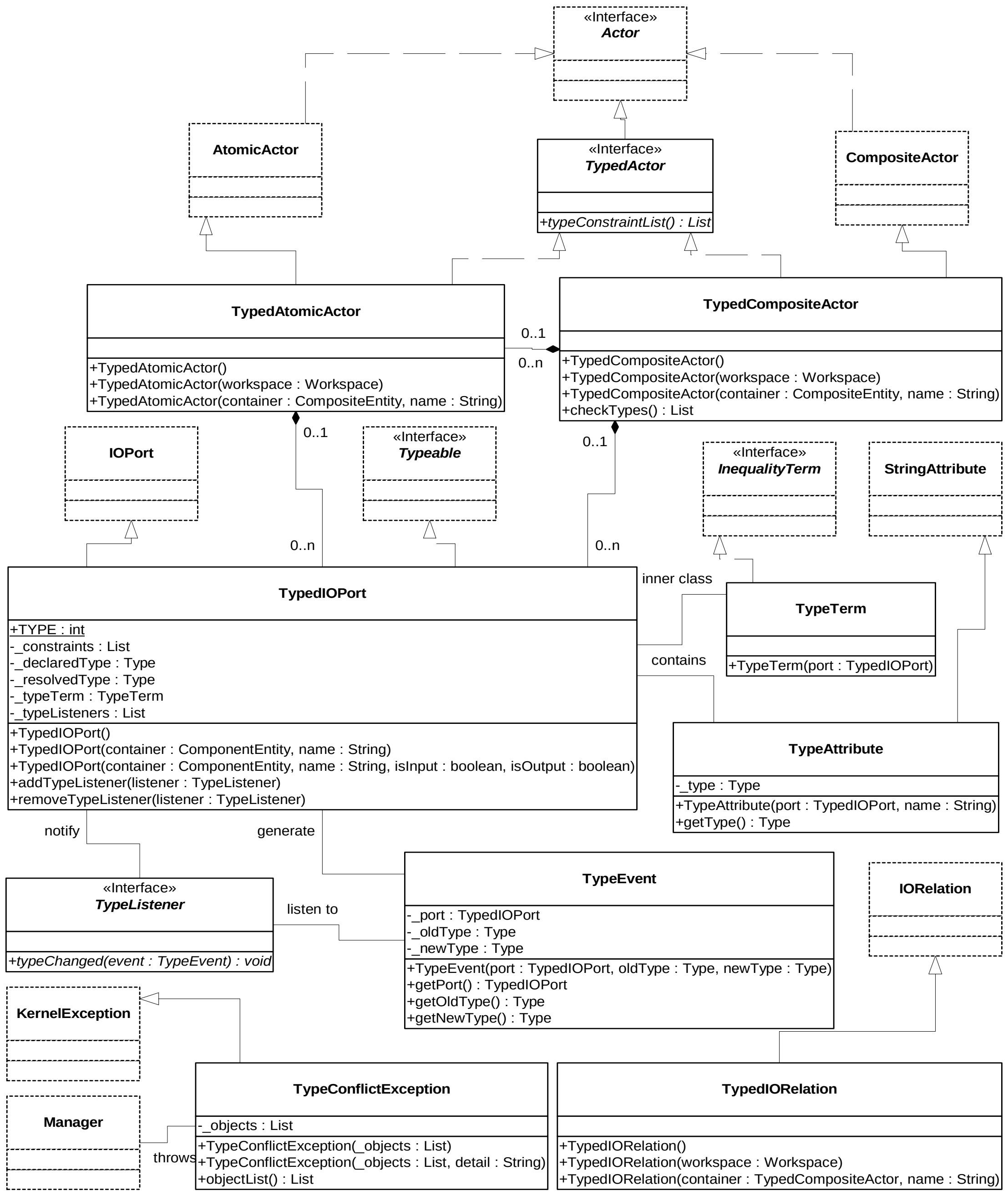

data.type
«Interface»
Type
+convert(t : Token) : Token
+isCompatible(token : Token) : boolean
+isConstant() : boolean
+isEqualTo(type : Type) : boolean
+isInstantialble() : boolean
+isSubstitutionInstance(type : Type) : boolean
n type conversion hierarchy
0..1
TypeLattice
-_lattice : TheTypeLattice
+compare(t1 : Token, t2 : Token) : int
+compare(t1 : Type, t2 : Type) : int
+lattice() : graph.CPO
BaseType
+UNKNOWN : BaseType
+BOOLEAN : BaseType
+BOOLEAN_MATRIX : BaseType
+COMPLEX : BaseType
+COMPLEX_MATRIX : BaseType
+DOUBLE : BaseType
+DOUBLE_MATRIX : BaseType
+FIX : BaseType
-FIX_MATRIX : BaseType
+INT : BaseType
+INT_MATRIX : BaseType
+LONG : BaseType
+LONG_MATRIX : BaseType
+MATRIX : BaseType
+NUMERICAL : BaseType
+OBJECT : BaseType
+SCALAR : BaseType
+STRING : BaseType
+GENERAL : BaseType
-_name : String
+forName(name : String) : Type
StructuredType
+initialize(type : Type)
+updateType(newType : StructuredType)
#_compare(type : StructuredType) : int
#_getRepresentative() : StructuredType
#_greatestLowerBound(type : StructuredType) : StructuredType
#_leastUpperBound(type : StructuredType) : StructuredType
ArrayType
-_declaredElementType : Type
-_elementType : Type
-_elementTypeTerm : ElementTypeTerm
-_representative : ArrayType
+ArrayType(elementType : Type)
+getElementType() : Type
+getElementTypeTerm() : InequalityTerm
RecordType
-_fields : Map
-_representative : RecordType
+RecordType(labels : String[], types : Type[])
+get(label : String) : Type
+getTypeTerm(label : String) : InequalityTerm
«Interface»
InequalityTerm
TypeConstant
-_type : Type
+TypeConstant(type : Type)
«Interface»
Typeable
+getType() : Type
+getTypeTerm() : InequalityTerm
+setTypeAtLeast(lesser : Typeable)
+setTypeAtLeast(typeTerm : InequalityTerm)
+setTypeAtMost(type : Type)
+setTypeEquals(type : Type)
+setTypeSameAs(equal : Typeable)
+typeConstraintList() : List

FIGURE 6.4. Classes in the data.type package.

checking is performed using the type compatibility rule. If the composite contains other opaque Typed-CompositeActors, this method recursively calls the checkTypes() method of the contained actors to perform type checking down the hierarchy. Hence, if this method is called on the top level TypedCompositeActor, type checking is performed through out the hierarchy.

If a type conflict is detected, i.e., if the declared type at the source end of a connection is greater than or incomparable with the type at the destination end of the connection, the ports at both ends of



«Interface»
Actor
AtomicActor
«Interface»
TypedActor
+typeConstraintList() : List
CompositeActor
TypedAtomicActor
+TypedAtomicActor()
+TypedAtomicActor(workspace : Workspace)
+TypedAtomicActor(container : CompositeEntity, name : String)
0..1
0..n
TypedCompositeActor
+TypedCompositeActor()
+TypedCompositeActor(workspace : Workspace)
+TypedCompositeActor(container : CompositeEntity, name : String)
+checkTypes() : List
IOPort
0..1
«Interface»
Typeable
0..1
«Interface»
InequalityTerm
StringAttribute
0..n
0..n
TypedIOPort
+TYPE : int
-_constraints : List
-_declaredType : Type
-_resolvedType : Type
-_typeTerm : TypeTerm
-_typeListeners : List
+TypedIOPort()
+TypedIOPort(container : ComponentEntity, name : String)
+TypedIOPort(container : ComponentEntity, name : String, isInput : boolean, isOutput : boolean)
+addTypeListener(listener : TypeListener)
+removeTypeListener(listener : TypeListener)
inner class
TypeTerm
+TypeTerm(port : TypedIOPort)
contains
TypeAttribute
-_type : Type
+TypeAttribute(port : TypedIOPort, name : String)
+getType() : Type
notify
generate
«Interface»
TypeListener
+typeChanged(event : TypeEvent) : void
listen to
TypeEvent
-_port : TypedIOPort
-_oldType : Type
-_newType : Type
+TypeEvent(port : TypedIOPort, oldType : Type, newType : Type)
+getPort() : TypedIOPort
+getOldType() : Type
+getNewType() : Type
IORelation
KernelException
Manager
throws
TypeConflictException
-_objects : List
+TypeConflictException(_objects : List)
+TypeConflictException(_objects : List, detail : String)
+objectList() : List
TypedIORelation
+TypedIORelation()
+TypedIORelation(workspace : Workspace)
+TypedIORelation(container : TypedCompositeActor, name : String)

FIGURE 6.5. Classes in the actor package that support type checking.

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 has a typeConstraints() method, which returns the type constraints of this actor. For atomic actors, the type constraints are different in different actors, but the TypedAtomicActor class provides a default implementation, which is that the type of any input port with undeclared type must be less than or equal to the type of any undeclared output port. Ports with declared types are not included in the default constraints. If all the ports have declared type, no constraints are generated. This default works for most of the control actors such as commutator, multiplexer, and the DownSample actor in figure 6.1. In addition, the typeConstraints() method also collects all the constraints from the contained Typeable objects, which are TypedIOPorts and Variables.

The typeConstraints() method in TypedCompositeActor collects all the constraints within the composite. It works in a similar fashion as the checkTypes() method, where it recursively goes down the containment hierarchy to collect type constraints of the contained actors. It also scans all the connections and forms type constraints on connections involving undeclared types. As with checkTypes(), if this method is called on the top level container, all the type constraints within the composite are returned.

The Manager class has a resolveTypes() method that invokes type checking and resolution. It uses the InequalitySolver class in the graph package to solve the constraints. If type conflicts are detected during type checking or after type resolution, this method throws TypeConflictException. This exception contains a List of Typeable objects where type conflicts occur. The resolveTypes() method is called inside Manager after all the mutations are processed. If TypeConflictException is thrown, it is caught within the Manager and an KernelException is generated to pass the exception information to the user interface.

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

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.

6.4.3 Setting Up Type Constraints

The class Inequality in the graph package is used to represent type constraints. This class contains two objects implementing the InequalityTerm interface, which represent the lesser and greater terms. InequalityTerm 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. In most cases, type constraints can be set up easily through the methods in the Typeable interface. For example, to constrain that the type of a port to be no greater than *Double*:

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

to constrain that the type of a port to be no less than the type of a parameter:

```
    port.setTypeAtLeast(parameter);
```

to specify that a parameter can only contain an ArrayToken, and to constrain the type of a port to be no less than the element type of that array:

```
parameter.setTypeEquals(new ArrayType(BaseType.UNKNOWN));
ArrayType arrayType = (ArrayType)parameter.getType();
InequalityTerm elementTerm = arrayType.getElementTypeTerm();
port.setTypeAtLeast(elementTerm);
```

These kinds of constraints appear in source actors such as Clock and Pulse, where the actor outputs a sequence of values specified by an ArrayToken.

In some actors, monotonic functions can help specify less straightforward constraints. The type resolution algorithm allows the lesser term to be a monotonic function when searching for the most specific types. That is, constraints in the form $f(\alpha) \leq b$ are admitted, where $f(\alpha)$ is a monotonic function of α and b 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 FunctionTerm 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;
}
```

Finally, 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.

6.4.4 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 Array*, meaning that it is an array of undefined element type. After type resolution, that type may be updated to *Double Array*. 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. One way to support this is to have bidirectional references between a variable structured type and its user, and only allow the type to have one user. But the bidirectional references make the implementation complicated, and consistency is hard to maintain. A better way is to always clone the structured type when its container is cloned, or when constructing a new instance of StructuredType. This is done in the data package. This implementation 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 *Array* can be updated to *Int Array*. However, α *Array* 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 Type does 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 Array* and α *Array* respectively, and a type constraint is that the second port is no less than the type of the first, that is, *Int Array* $\le \alpha$ *Array*, the type resolution algorithm will change the type of the second port to *Int Array*. This step cannot be done by simply changing the type reference in the second port to an instance of *Int Array*, since type constraints may be set up between α and another typed objects. Instead, updateType() only changes the type reference for α to *Int*.

6.5 Examples

6.5.1 Polymorphic DownSample

In figure 6.1, if the DownSample is designed to do downsampling for any kind of token, its type constraint is just *samplerIn* $\le$ *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 $\le$ *samplerIn*

samplerIn $\le$ *samplerOut*

samplerOut $\le$ *Complex*

Complex $\le$ *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.

6.5.2 Fork Connection

Consider two simple topologies in figure 6.6. where a single output is connected to two inputs in 6.6(a) and two outputs are connected to a single input in 6.6(b). Denote the types of the ports by *a1, a2, a3, b1, b2, b3,* as indicated in the figure. Some possibilities of legal and illegal type assignments are:

- In 6.6(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 6.6(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.

6.6 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 SequenceToArray, the parameter TokenConsumptionRate of the input port determines the length of the output array, while in ArrayToSequence, the parameter tokenProductionRate of the output port specifies the length of the input array. If the ArrayToken received by ArrayToSequence does not have the correct length, an exception will be thrown.

The actor.lib package contains two actors that assembles and disassembles 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.

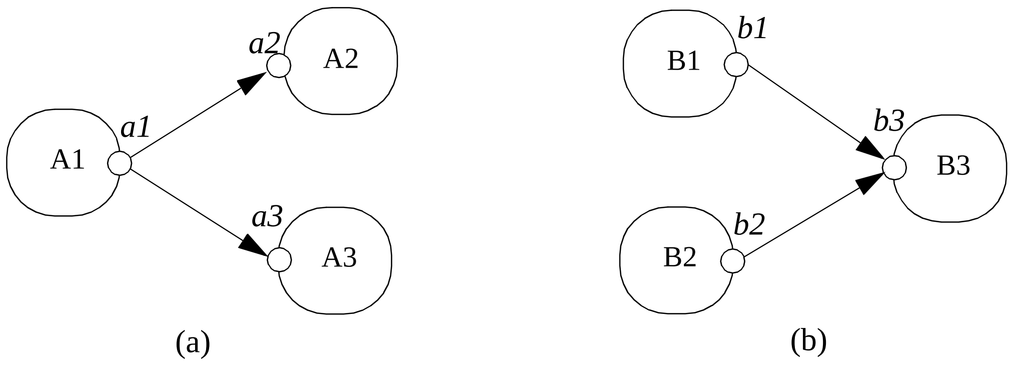


FIGURE 6.6. Two simple topologies with types.

Appendix C: The Type Resolution Algorithm

The type resolution algorithm starts by assigning all the type variables the bottom element of the type hierarchy, *NaT*, then repeatedly updating the variables to a greater element until all the constraints are satisfied, or when the algorithm finds that the set of constraints are not satisfiable. The kind of inequality constraints the algorithm can determine satisfiability are the ones with the greater term (the right side of the inequality) being a variable, or a constant. The algorithm allows the left side of the inequality to contain monotonic functions of the type variables, but not the right side. The first step of the algorithm is to divide the inequalities into two categories, *Cvar* and *Ccnst*. The inequalities in *Cvar* have a variable on the right side, and the inequalities in *Ccnst* have a constant on the right side. In the example of figure 6.2, *Cvar* consists of:

$Int \le \alpha$
$Double \le \beta$
$\alpha \le \gamma$
$\beta \le \gamma$

And *Ccnst* consists of:

$\gamma \le Double$
$\gamma \le Complex$

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

$Int \le \alpha(NaT)$
$Double \le \beta(NaT)$
$\alpha(NaT) \le \gamma(NaT)$
$\beta(NaT) \le \gamma(NaT)$

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

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

Not-satisfied	Satisfied
$Int \le \alpha(NaT)$	$\alpha(NaT) \le \gamma(NaT)$
$Double \le \beta(NaT)$	$\beta(NaT) \le \gamma(NaT)$

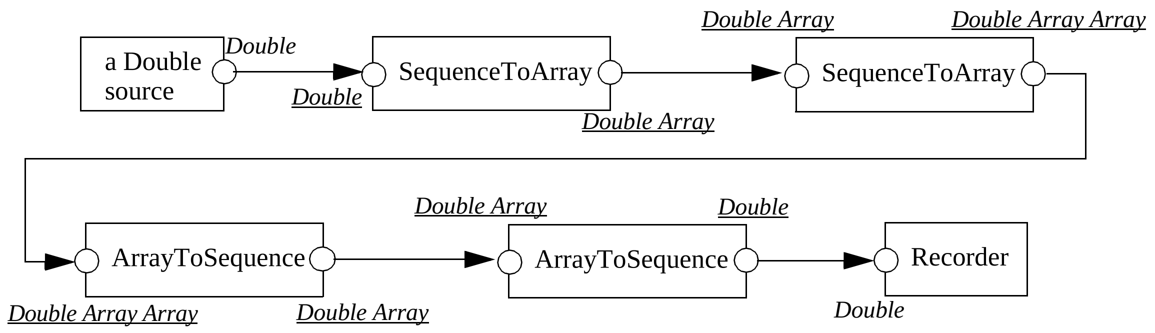


FIGURE 6.7. Conversion between sequence and array.

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 $int \leq \alpha(NaT)$, then

$$\alpha = Int \vee NaT = Int \tag{2}$$

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

Not-satisfied	Satisfied
$Double \leq \beta(NaT)$	$Int \leq \alpha(Int)$
$\alpha(Int) \leq \gamma(NaT)$	$\beta(NaT) \leq \gamma(NaT)$

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

$$\alpha = Int,\ \ \beta = \gamma = Double$$

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

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

This algorithm can be viewed as repeated evaluation of a monotonic function, and the solution is the fixed point of the function. Equation (2) can be viewed as a monotonic function applied to a type variable. The repeated update of all the type variables can be viewed as the evaluation of a monotonic function that is the composition of individual functions like (2). The evaluation reaches a fixed point when a set of type variable assignments satisfying the constraints in *Cvar* is found.

Rehof and Mogensen [79] proved that the above algorithm is linear time in the number of occurrences of symbols in the constraints, and gave an upper bound on the number of basic computations. In our formulation, the symbols are type constants and type variables, and each constraint contains two symbols. So the type resolution algorithm is linear in the number of constraints.

7 CT Domain

Author: *Jie Liu*

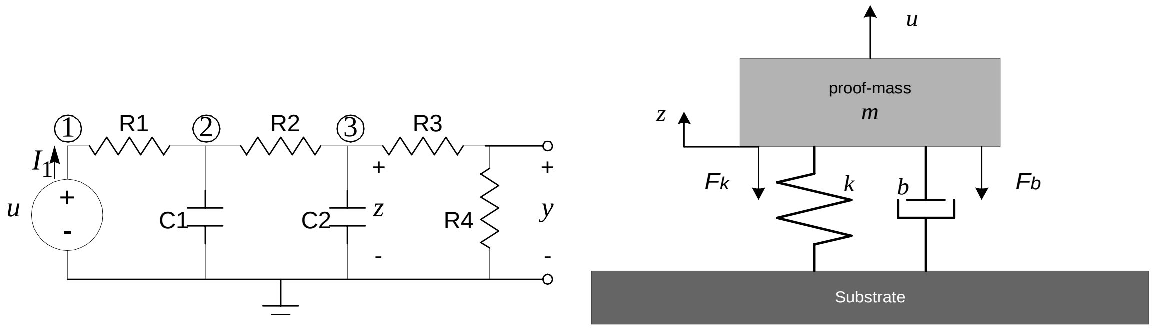
7.1 Introduction

The continuous-time (CT) domain in Ptolemy II aims to help the design and simulation of systems that can be modeled using ordinary differential equations (ODEs). ODEs are often used to model analog circuits, plant dynamics in control systems, lumped-parameter mechanical systems, lumped-parameter heat flows and many other physical systems.

Let's start with an example. Consider a second order differential system,

$$\begin{aligned} m\ddot{z}(t) + b\dot{z}(t) + kz(t) &= u(t) \quad . \\ y(t) &= c \cdot z(t) \\ z(0) = 10, \dot{z}(0) &= 0. \end{aligned} \tag{1}$$

The equations could be a model for an analog circuit as shown in figure 7.1(a), where *z* is the voltage



(a) A circuit implementation. (b) A mechanical implementation.

FIGURE 7.1. Possible implementations of the system equations.

of node 3, and

$$m = R1 \cdot R2 \cdot C1 \cdot C2 \tag{2}$$

$$k = R1 \cdot C1 + R2 \cdot C2$$

$$b = 1$$

$$c = \frac{R4}{R3 + R4}.$$

Or it could be a lumped-parameter spring-mass mechanical model for the system shown in figure 7.1(b), where z is the position of the mass, m is the mass, k is the spring constant, b is the damping parameter, and $c = 1$.

In general, an ODE-based continuous-time system has the following form:

$$\dot{x} = f(x, u, t) \tag{3}$$

$$y = g(x, u, t) \tag{4}$$

$$x(t_0) = x_0, \tag{5}$$

where, $t \in \Re$, $t \geq t_0$, a real number, is *continuous time*. At any time t, $x \in \Re^n$, an n-tuple of real numbers, is the *state* of the system; $u \in \Re^m$ is the m-dimensional *input* of the system; $y \in \Re^l$ is the l-dimensional *output* of the system; $\dot{x} \in \Re^n$ is the derivative of x with respect to time t, i.e.

$$\dot{x} = \frac{dx}{dt}. \tag{6}$$

Equations (3), (4), and (5) are called the *system dynamics*, the *output map*, and the *initial condition* of the system, respectively.

For example, for the mechanical system above, if we define a vector

$$x(t) = \begin{bmatrix} z(t) \\ \dot{z}(t) \end{bmatrix}, \tag{7}$$

then system (1) can be written in form of (3)-(5), like

$$\dot{x}(t) = \frac{1}{m}\begin{bmatrix} 0 & 1 \\ -k & -b \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 1/m \end{bmatrix} u(t) \tag{8}$$

$$y(t) = \begin{bmatrix} c & 0 \end{bmatrix} x(t)$$

$$x(0) = \begin{bmatrix} 10 \\ 0 \end{bmatrix}.$$

The solution, $x(t)$, of the set of ODE (3)-(5), is a continuous function of time, also called a *waveform*, which satisfies the equation (3) and initial condition (5). The output of the system is then defined as a function of $x(t)$ and $u(t)$, which satisfies (4). The precise solution of a set of ODEs is usually impossible to be found using digital computers. Numerical solutions are approximations of the precise solution. A numerical solution of ODEs are usually done by integrating the right-hand side of (3) on a discrete set of time points. Using digital computers to simulate continuous-time systems has been studied for more than three decades. One of the most well-known tools is Spice [69]. The CT domain differs from Spice-like continuous-time simulators in two ways — the system specification is somewhat different, and it is designed to interact with other models of computation.

7.1.1 System Specification

There are usually two ways to specify a continuous-time system, the conservation-law model and the signal-flow model [42]. The conservation-law models, like the nodal analysis in circuit simulation [39] and bond graphs [81] in mechanical models, define systems by their physical components, which specify relations of *cross* and *through* variables, and *conservation laws* are used to compile the component relations into global system equations. For example, in circuit simulation, the cross variables are voltages, the through variables are currents, and the conservation laws are Kirchhoff's laws. This model directly reflects the physical components of a system, thus is easy to construct from a potential implementation. The actual mathematical representation of the system is hidden. In signal-flow models, entities in a system are maps that define the mathematical relation between their input and output signals. Entities communicate by passing signals. This kind of models directly reflects the mathematical relations among signals, and is more convenient for specifying systems that do not have an explicit physical implementation yet.

In the CT domain of Ptolemy II, the signal-flow model is chosen as the interaction semantics. The conservation-law semantics may be used within an entity to define its I/O relation. There are four major reasons for this decision:

1. *The signal-flow model is more abstract*. Ptolemy II focuses on system-level design and behavior simulation. It is usually the case that, at this stage of a design, users are working with abstract mathematical models of a system, and the implementation details are unknown or not cared about.
2. *The signal flow model is more flexible and extensible*, in the sense that it is easy to embed components that are designed using other models. For example, a discrete controller can be modeled as a component that internally follows a discrete event model of computation but exposes a continuous-time interface.
3. *The signal flow model is consistent with other models of computation in Ptolemy II*. Most models of computation in Ptolemy use message-passing as the interaction semantics. Choosing the signal-flow model for CT makes it consistent with other domains, so the interaction of heterogeneous systems is easy to study and implement. This also allows domain polymorphic actors to be used in the CT domain.
4. *The signal flow model is compatible with the conservation law model*. For physical systems that are based on conservation laws, it is usually possible to wrap them into an entity in the signal flow model. The inputs of the entity are the excitations, like the current on ideal current sources, and the outputs are the variables that the rest of the system may be interested in.

The signal flow block diagram of the system (3) - (5) is shown in figure 7.2. The system dynamics (3) is built using integrators with feedback. In this figure, u, $\dot{x}$, x, and y, are continuous signals flowing from one block to the next. Notice that this diagram is only conceptual, most models may involve multiple integrators[1]. Time is shared by all components, so it is not considered as an input. At any fixed time t, if the "snapshot" values $x(t)$ and $u(t)$ are given, then $\dot{x}(t)$ and $y(t)$ can be found by evaluating f

1. Ptolemy II does not support vectorization in the CT domain yet.

and g, which can be achieved by firing the respective blocks. The “snapshot” of all the signals at t is called the *behavior* of the system at time t.

The signal-flow model for the example system (1) is shown in figure 7.3. For comparison purpose, the conservation-law model (modified nodal analysis) of the system shown in figure 7.1(a) is shown in (9).

$$\begin{bmatrix} \frac{1}{R1} & -\frac{1}{R1} & 0 & 0 & -1 \\ -\frac{1}{R1} & \frac{1}{R1}+\frac{1}{R2}+C1\frac{d}{dt} & -\frac{1}{R2} & 0 & 0 \\ 0 & -\frac{1}{R2} & \frac{1}{R2}+\frac{1}{R3}+C2\frac{d}{dt} & -\frac{1}{R3} & 0 \\ 0 & 0 & -\frac{1}{R3} & \frac{1}{R3}+\frac{1}{R4} & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ y \\ I_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ u \end{bmatrix} \qquad (9)$$

By doing some math, we can see that (9) and (8) are in fact equivalent. Equation (9) can be easily assembled from the circuit, but it is more complicated than (8). Notice that in (9) d/dt is the derivative operator, which is replaced by an integration algorithm at each time step, and the system equations reduce to a set of algebraic equations. Spice software is known to have a very good simulation engine for models in form of (9).

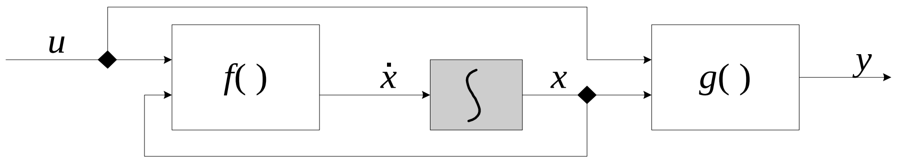


FIGURE 7.2. A conceptual block diagram for continuous time systems.

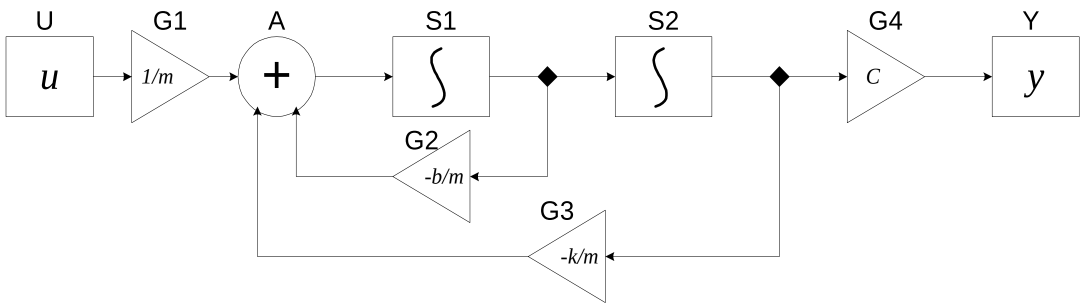


FIGURE 7.3. The block diagram for the example system.

7.1.2 Time

One distinct characterization of the CT model is the continuity of time. This implies that a continuous-time system have a behavior at any time instance. The simulation engine of the CT model should be able to compute the behavior of the system at any time point, although it may march discretely in time. In order to achieve an accurate simulation, time should be carefully discretized. The discretization of time, which appears as integration step sizes, may be determined by time points of interest (e.g. discontinuities), by the numerical error of integration, and by the convergence in solving algebraic equations.

Time is also global, which means that all components in the system share the same notion of time.

7.2 Solving ODEs numerically

We outline some basic terminologies on numerical ODE solving techniques that are used in this chapter. This is not a summary of numerical ODE solving theory. For a detailed treatment for ODEs and their numerical solutions, please refer to books on numerical solutions for ODEs, e.g. [29].

Not all ODEs have a solution, and some ODEs have more than one solution. In such situations, we say that the solution is not well defined. This is usually a result of errors in the system modeling. We restrict our discussion to systems that have unique solutions. Theorem 1 in Appendix D states the conditions for the existence and uniqueness of solutions of ODEs. Roughly speaking, we denote by D a set in $\Re$ which contains at most a finite number of points per unit interval, and let u be piecewise-continuous on $\Re - D$. Then, for any fixed $u(t)$, if f is also piecewise-continuous on $\Re - D$, and f satisfies the Lipschitz condition (see e.g. [29]), then the ODE (3) with the initial condition (5) has a unique solution. The solution is called the *state trajectory* of the system. The key of simulating a continuous-time system numerically is to find an accurate numerical approximation of the state trajectory.

7.2.1 Basic Notations

Usually, only the solution on a finite time interval $[t_0, t_f]$ is needed. A simulation of the system is performed on discrete time points in this interval. We denote by

$$Tc = \{t_0, t_1, t_2, \ldots t_n, \ldots t_f\}, Tc \subset [t_0, t_f], \tag{10}$$

where

$$t_0 < t_1 < t_2 < \ldots < t_n < \ldots < t_f, \tag{11}$$

the set of the discrete time points of interest. To explicitly illustrate the discretization of time and the difference between the precise solution and the numerical solution, we use the following notation in the rest of the chapter:

- t_n : the n-th time point, to explicitly show the discretization of time. However, we write t, if the index n is not important.
- $x[t_i, t_j]$: the *precise* (continuous) state trajectory from time t_i to t_j;
- $x(t_n)$: the *precise* solution of (3) at time t_n;
- x_{t_n} : the *numerical* solution of (3) at time t_n;
- $h_n = t_n - t_{n-1}$: step size of numerical integration. We also write h if the index n in the sequence

is not important. For accuracy reason, h may not be uniform.

- $\|x(t_n) - x_{t_n}\|$: the 2-normed difference between the precise solution and the numerical solution at step *n* is called the (*global*) *error* at step *n*; the difference, when we assume $x_{t_0} \dots x_{t_{n-1}}$ are precise, is called the *local error* at step *n*. Local errors are usually easy to estimate and the estimation can be used for controlling the accuracy of numerical solutions.

A general way of numerically simulating a continuous-time system is to compute the state and the output of the system in an increasing order of t_n. Such algorithms are called the *time-marching* algorithms, and, in this chapter, we only consider these algorithms. There are variety of time marching algorithms that differ on how x_{t_n} is computed given $x_{t_0} \dots x_{t_{n-1}}$. The choice of algorithms is application dependent, and usually reflects the speed, accuracy, and numerical stability trade-offs.

7.2.2 Fixed-Point Behavior

Numerical ODE solving algorithms approximate the derivative operator in (3) using the history and the current knowledge on the state trajectory. That is, at time t_n, the derivative of *x* is approximated by a function of $x_{t_0}, \dots, x_{t_{n-1}}, x_{t_n}$, i.e.

$$\dot{x}_{t_n} = p(x_{t_0}, \dots, x_{t_{n-1}}, x_{t_n}) . \tag{12}$$

Plugging (3) in this, we get

$$p(x_{t_0} \dots x_{t_{n-1}}, x_{t_n}) = f(x_{t_n}, u(t_n), t_n) \tag{13}$$

Depending on whether x_{t_n} explicitly appears in (13), the algorithms are called *explicit integration algorithms* or *implicit integration algorithms*. That is, we end up solving a set of algebraic equations in one of the two forms:

$$x_{t_n} = F_E(x_{t_0}, \dots, x_{t_{n-1}}) \tag{14}$$

or

$$F_I(x_{t_0}, \dots, x_{t_n}) = 0 , \tag{15}$$

where F_E and F_I are derived from the time t_n, the input $u(t_n)$, the function *f*, and the history of *x* and $\dot{x}$. Solving (14) or (15) at a particular time t_n is called an *iteration* of the CT simulation at t_n.

Equation (14) can be solved simply by a function evaluation and an assignment. But the solution of (15) is the *fixed point* of F_I, which may not exist, may not be unique, or may not be able to be found. The *contraction mapping theorem* [13] shows the existence and uniqueness of the fixed-point solution, and provides one way to find it. Given the map F_I that is a local contraction map (generally true for small enough step sizes) and let an initial guess σ_0 be in the contraction radius, then a unique fixed point exists and can be found by iteratively computing:

$$\sigma_1 = F_E(\sigma_0), \sigma_2 = F_E(\sigma_1), \sigma_3 = F_E(\sigma_2), \dots \tag{16}$$

Solving both (14) and (15) should be thought of as finding the fixed-point behavior of the system at a particular time. This means both functions F_E and F_I should be smooth w.r.t. time, during one iteration of the simulation. This further implies that the topology of the system, all the parameters, and all the internal states that the firing functions depend on should be kept unchanged. We require that domain polymorphic actors to update internal states only in the `postfire()` method exactly for this reason.

7.2.3 ODE Solvers Implemented

The following solvers has been implemented in the CT domain.

1. Forward Euler solver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + h_{n+1} \cdot \dot{x}_{t_n} \\ &= x_{t_n} + h_{n+1} \cdot f(x_{t_n}, u_{t_n}, t_n) \end{aligned} \tag{17}$$

2. Backward Euler solver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + h_{n+1} \cdot \dot{x}_{t_{n+1}} \\ &= x_{t_n} + h_{n+1} \cdot f(x_{t_{n+1}}, u_{t_{n+1}}, t_{n+1}) \end{aligned} \tag{18}$$

3. 2(3)-order Explicit Runge-Kutta solver

$$\begin{aligned} K_0 &= h_{n+1} \cdot f(x_{t_n}, u_{t_n}, t_n) \\ K_1 &= h_{n+1} \cdot f(x_{t_n} + K_0/2, u_{t_n + h_{n+1}/2}, t_n + h_{n+1}/2) \\ K_2 &= h_{n+1} \cdot f(x_{t_n} + 3K_1/4, u_{t_n + 3h_{n+1}/4}, t_n + 3h_{n+1}/4) \\ \tilde{x}_{t_{n+1}} &= x_{t_n} + \frac{2}{9}K_0 + \frac{1}{3}K_1 + \frac{4}{9}K_2 \end{aligned} \tag{19}$$

with error control:

$$\begin{aligned} K_3 &= h_{n+1} \cdot f(\tilde{x}_{t_{n+1}}, u_{t_{n+1}}, t_{n+1}) \\ LTE &= -\frac{5}{72}K_0 + \frac{1}{12}K_1 + \frac{1}{9}K_2 - \frac{1}{8}K_3 \end{aligned} \tag{20}$$

if $|LTE| < ErrorTolerance$, $x_{t_{n+1}} = \tilde{x}_{t_{n+1}}$, otherwise, fail. If this step is successful, the next integration step size is predicted by:

$$h_{n+2} = h_{n+1} \cdot max(0.5, 0.8 \cdot \sqrt[3]{(ErrorTolerance)/|LTE|}) \tag{21}$$

4. Trapezoidal Rule solver:

$$\begin{aligned} x_{t_{n+1}} &= x_{t_n} + \frac{h_{n+1}}{2}(\dot{x}_{t_n} + \dot{x}_{t_{n+1}}) \qquad . \\ &= x_{t_n} + \frac{h_{n+1}}{2}(\dot{x}_{t_n} + f(x_{t_{n+1}}, u_{t_{n+1}}, t_{n+1})) \end{aligned} \tag{22}$$

Among these solvers, 1) and 3) are explicit; 2) and 4) are implicit. Also, 1) and 2) do not perform step size control, so are called fixed-step-size solvers; 3) and 4) change step sizes according to error estimation, so are called variable-step-size solvers. Variable-step-size solvers adapt the step sizes according to changes of the system flow, thus are “smarter” than fixed-step-size solvers.

7.2.4 Discontinuity

The existence and uniqueness of the solution of an ODE (Theorem 1 in Appendix D) allows the right-hand side of (3) to be discontinuous at a countable number of discrete points *D*, which are called the *breakpoints* (also called the *discontinuous points* in some literature). These breakpoints may be caused by the discontinuity of input signal *u*, or by the intrinsic flow of *f*. In theory, the solutions at these points are not well defined. But the left and right limits are. So, instead of solving the ODE at those points, we would actually try to find the left and right limits.

One impact of breakpoints on ODE solvers is that history solutions are useless when approximating the derivative of *x* after the breakpoints. The solver should resolve the new initial conditions and start the solving process as if it is at a starting point. So, the discretization of time should step exactly on breakpoints for the left limit, and start at the breakpoint again after finding the right limit.

A breakpoint may be known beforehand, in which case it is called a *predictable breakpoint*. For example, a square wave source actor knows its next flip time. This information can be used to control the discretization of time. A breakpoint can also be *unpredictable*, which means it is unknown until the time it occurs. For example, an actor that varies its functionality when the input signal crosses a threshold can only report a “missed” breakpoint after an integration step is finished. How to handle breakpoints correctly is a big challenge for integrating continuous-time models with discrete models like DE and FSM.

7.2.5 Breakpoint ODE Solvers

Breakpoints in the CT domain are handled by adjusting integration steps. We use a table to handle predictable breakpoints, and use the step size control mechanism to handle unpredictable breakpoints. The breakpoint handling are transparent to users, and the implementation details (provided in section 7.7.4) are only needed when developing new directors, solvers, or event generators.

Since the history information is useless at breakpoints, special ODE solvers are designed to restart the numerical integration process. In particular, we have implemented the following breakpoint ODE solvers.

1. DerivativeResolver:

It calculates the derivative of the current state, i.e. $\frac{dx}{dt}$. This is simply done by evaluation the right-hand side of (3). At breakpoints, this solver is used for the first step to generate history information for explicit methods or one step methods.

2. ImpulseBESolver:

$$\tilde{x}_{t_{n+1}} = x_{t_n} + h_{n+1} \cdot \tilde{\dot{x}}_{t_{n+1}} \tag{23}$$

$$x_{t_n^+} = \tilde{x}_{t_{n+1}} - h_{n+1} \cdot \dot{x}_{t_n^+}$$

The two time points t_n and t_n^+ have the same time value. This solver is used for breakpoints at which a Dirac impulse signal appears.

Notice that none of these solvers advance time. They can only be used at breakpoints.

7.3 CT Actors

A CT system can be built up using actors in the ptolemy.domains.ct.lib package and domain polymorphic actors that have continuous behaviors (i.e. all actors that do not implement the SequenceActor interface). The key actor in CT is the integrator. It serves the unique role of wiring up ODEs. Other actors in a CT system are usually stateless. A general understanding is that, in a pure continuous-time model, all the information — the state of the system— is stored in the integrators.

7.3.1 CT Actor Interfaces

In order to schedule the execution of actors in a CT model and to support the interaction between CT and other domains (which are usually discrete), we provide the following interfaces.

- **CTDynamicActor**. Dynamic actors are actors that contains continuous dynamics in their I/O path. An integrator is a dynamic actor, and so are all actors that have integration relations from their inputs to their outputs.
- **CTEventGenerator**. *Event generators* are actors that convert continuous time input signals to discrete output signals.
- **CTStatefulActor**. Stateful actors are actors that have internal states. The reason to classify this kind of actor is to support rollback, which may happen when a CT model is embedded in a discrete event model.
- **CTStepSizeControlActor**. Step size control actors influence the integration step size by telling the director whether the current step is accurate. The accuracy is in the sense of both tolerable numerical errors and absence of unpredictable breakpoints. It may also provide information about refining a step size for an inaccurate step and suggesting the next step size for an accurate step.
- **CTWaveformGenerator**. *Waveform generators* are actors that convert discrete input signals to continuous-time output signals.

Strictly speaking, event generators and waveform generators do not belong to any domain, but the CT domain is design to handle them intrinsically. When building systems, CT parts can always provide discrete interface to other domains.

Neither a loop of dynamic actors nor a loop of non-dynamic actors are allowed in a CT model. They introduce problems about the order that actors be executed. A loop of dynamic actors can be easily broken by a Scale actor with scale 1. A loop of non-dynamic actors builds an algebraic equation. The CT domain does not support modeling algebraic equations, yet.

7.3.2 Actor Library

1. **CTPeriodicalSampler**. This event generator periodically samples the input signal and generates events with the value of the input signal at these time points. The sampling rate is given by the *samplePeriod* parameter, which has default value 0.1. The sampling time points, which are known beforehand, are examples of predictable breakpoints.
2. **CTTriggeredSampler**. This actor samples the continuous input signal when there is a discrete event present at the "trigger" input.
3. **ContinuousTransferFunction**. A transfer function in the continuous time domain. This actor implements a transfer function where the single input (u) and single output (y) can be expressed in (Laplace) transfer function form as the following equation:

$$\frac{Y(s)}{U(s)} = \frac{b_1 s^{m-1} + b_2 s^{m-2} + \ldots + b_m}{a_1 s^{n-1} + a_2 s^{n-2} + \ldots + a_n} \tag{24}$$

where *m* and *n* are the number of numerator and denominator coefficients, respectively. This actors has two parameters – *numerator* and *denominator* – containing the coefficients of the numerator and denominator in descending powers of *s*. The parameters are double arrays. The order of the denominator (*n*) must be greater than or equal to the order of the numerator (*m*).

4. **DifferentialSystem**. The differential system model implements a system whose behavior is defined by:

$$\begin{aligned} \dot{x} &= f(x, u, t) \\ y &= g(x, u, t) \\ x(t_0) &= x_0 \end{aligned} \tag{25}$$

where *x* is the state vector, *u* is the input vector, and *y* is the output vector, *t* is the time. Users must give the name of the variables by filling in the parameter and add ports with proper names. The actor, upon creation, has no inputs and no outputs. After creating proper ports, their names can be used in the expressions of state equations and output equations. The name of the state variables are manually added by filling in the *stateVariableNames* parameter.

The state equations and output maps must be manually created by users as parameters. If there are *n* state variables $x_1 \ldots x_n$ then users need to create *n* additional parameters, one for each state equation. And the parameters must be named as x_1*_dot*, ..., x_n*_dot*, respectively. Similarly, if the output ports have names $y_1 \ldots y_r$, then users must create additional *r* parameters for output maps. These parameters should be named y_1, ..., y_r, respectively.

5. **Integrator**: The integrator for continuous-time simulation. An integrator has one input port and one output port. Conceptually, the input is the derivative of the output, and an ordinary differential equation is modeled as an integrator with feedback.

   An integrator is a dynamic, step-size-control, and stateful actor. To help resolve new states from previous states, a set of variables are used:

- *state and its derivative*: These are the new state and its derivative at a time point, which have been confirmed by all the step size control actors.
- *tentative state and tentative derivative*: These are the state and derivative which have not been confirmed. It is a starting point for other actors to estimate the accuracy of this integration step.
- *history*: The previous states and derivatives. An integrator remembers the history states and their derivatives for the past several steps. The history is used by multistep methods.

   An integrator has one parameter: *initialState*. At the initialization stage of the simulation, the state of the integrator is set to the initial state. Changes of *initialState* will be ignored after the simulation starts, unless the `initialize()` method of the integrator is called again. The default value of this parameter is 0.0. An integrator can possibly have several auxiliary variables. These auxiliary variables are used by ODE solvers to store intermediate states for individual integrators.

6. **LinearStateSpace**. The State-Space model implements a system whose behavior is defined by:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \\ x(t_0) &= x_0 \end{aligned} \tag{26}$$

where x is the state vector, u is the input vector, and y is the output vector. The matrix coefficients must have the following characteristics:

- A must be an n-by-n matrix, where n is the number of states.
- B must be an n-by-m matrix, where m is the number of inputs.
- C must be an r-by-n matrix, where r is the number of outputs.
- D must be an r-by-m matrix.

The actor accepts m inputs and generates r outputs through a multiple input port and a multiple output port. The widths of the ports must match the number of rows and columns in corresponding matrices, otherwise, an exception will be thrown.

7. **ZeroCrossingDetector**. This is an event generator that monitors the signal coming in from an input port – trigger. If the trigger is zero, then output the token from the input port. Otherwise, there is no output. This actor controls the integration step size to accurately resolve the time that the zero crossing happens. It has a parameter, *errorTolerance*, which controls how accurately the zero crossing is determined.

8. **ZeroOrderHold**. This is a waveform generator that converts discrete events into continuous signals. This actor acts as a zero-order hold. It consumes the token when the `consumeCurrentEvent()` is called. This value will be held and emitted every time it is fired, until the next time `consumeCurrentEvent()` is called. This actor has one single input port ,one single output port, and no parameters.

9. **ThresholdMonitor**. This actor controls the integration steps so that the given threshold (on the input) is not crossed in one step. This actor has one input port and one output port. It has two parameters *thresholdWidth* and *thresholdCenter*, which have default value 1e-2 and 0, respectively. If the input is within the range defined by the threshold center and threshold width, then a true token is emitted from the output.

7.3.3 Domain Polymorphic Actors

Not all domain polymorphic actors can be used in the CT domain. Whether an actor can be used depends on how the internal states of the actor evolve when executing.

- **Stateless actors**: All stateless actors can be used in CT. In fact, most CT systems are built by integrators and stateless actors.
- **Timed actors**: Timed actors change their states according to the notion of time in the model. All actors that implement the TimedActor interface can be used in CT, as long as they do not also implement SequenceActor. Timed actors that can be used in CT include plotters that are designed to plot timed signals.
- **Sequence actors**: Sequence actors change their states according to the number of input tokens received by the actor and the number of times that the actor is postfired. Since CT is a time driven model, rather than a data driven model, the number of received tokens and the number of postfires do not have a significant semantic meaning. So, none of the sequence actors can be used in the CT domain. For example, the Ramp actor in Ptolemy II changes its state –– the next token to emit ––

corresponding to the number of times that the actor is postfired. In CT, the number of times that the actor is postfired depends on the discretization of time, which further depend on the choice of ODE solvers and setting of parameters. As a result, the slope of the ramp may not be a constant, and this may lead to very counterintuitive models. The same functionality is replaced by a CurrentTime actor and a Scale actor. If sequence behaviors are indeed required, event generators and waveform generators may be helpful to convert continuous and discrete signals.

7.4 CT Directors

There are three CT directors –– CTMultiSolverDirector, CTMixedSignalDirector, and CTEmbeddedDirector. The first one can only serve as a top-level director, a CTMixedSignalDirector can be used both at the top-level or inside a composite actor, and a CTEmbeddedDirector can only be contained in a CTCompositeActor. In terms of mixing models of computation, all the directors can execute composite actors that implement other models of computation, as long as the composite actors are properly connected (see section 7.5). Only CTMixedSignalDirector and CTEmbeddedDirector can be contained by other domains. The outside domain of a composite actor with CTMixedSignalDirector can be any discrete domain, such as DE, DT, etc. The outside domain of a composite actor with CTEmbeddedDirector must also be CT or FSM, if the outside domain of the FSM model is CT. (See also the HSDirector in the FSM domain.)

7.4.1 ODE Solvers

There are six ODE solvers implemented in the ptolemy.domains.ct.kernel.solver package. Some of them are specific for handling breakpoints. These solvers are ForwardEulerSolver, BackwardEulerSolver, ExplicitRK23Solver, TrapezoidalRuleSolver, DerivativeResolver, and ImpulseBESolver. They implement the ODE solving algorithms in section 7.2.3 and section 7.2.5, respectively.

7.4.2 CT Director Parameters

The CTDirector base class maintains a set of parameters which controls the execution. These parameters, shared by all CT directors, are listed in Table 14 on page 12. Individual directors may have their own (additional) parameters, which will be discussed in the appropriate sections.

Table 14: CTDirector Parameters

Name	Description	Type	Default Value
errorTolerance	The upper bound of local errors. Actors that perform integration error control (usually integrators in variable step size ODE solving methods) will compare the estimated local error to this value. If the local error estimation is greater than this value, then the integration step is considered inaccurate, and should be restarted with a smaller step sizes.	double	1e-4
initStepSize	This is the step size that users specify as the desired step size. For fixed step size solvers, this step size will be used in all non-breakpoint steps. For variable step size solvers, this is only a suggestion.	double	0.1
maxIterations	This is used to avoid the infinite loops in (implicit) fixed-point iterations. If the number of fixed-point iterations exceeds this value, but the fixed point is still not found, then the fixed-point procedure is considered failed. The step size will be reduced by half and the integration step will be restarted.	int	20

Table 14: CTDirector Parameters

Name	Description	Type	Default Value
maxStepSize	The maximum step size used in a simulation. This is the upper bound for adjusting step sizes in variable step-size methods. This value can be used to avoid sparse time points when the system dynamic is simple.	double	1.0
minStepSize	The minimum step size used in a simulation. This is the lower bound for adjusting step sizes. If this step size is used and the errors are still not tolerable, the simulation aborts. This step size is also used for the first step after breakpoints.	double	1e-5
startTime	The start time of the simulation. This is only applicable when CT is the top level domain. Otherwise, the CT director follows the time of its executive director.	double	0.0
stopTime	The stop time of the simulation. This is only applicable when CT is the top level domain. Otherwise, the CT director follows the time of its executive director.	double	Double. MAX_ VALUE
timeResolution	This controls the comparison of time. Since time in the CT domain is a double precision real number, it is sometimes impossible to reach or step at a specific time point. If two time points are within this resolution, then they are considered identical.	double	1e-10
valueResolution	This is used in (implicit) fixed-point iterations. If in two successive iterations the difference of the states is within this resolution, then the integration step is called converged, and the fixed point is considered reached.	double	1e-6

7.4.3 CTMultiSolverDirector

A CTMultiSolverDirector has two ODE solvers — one for ordinary use and one specifically for breakpoints. Thus, besides the parameters in the CTDirector base class, this class adds two more parameters as shown in Table 15 on page 13.

Table 15: Additional Parameter for CTMultiSolverDirector

Name	Description	Type	Default Value
ODESolver	The fully qualified class name for the ODE solver class.	string	“ptolemy.domains.ct.kernel.solver.ForwardEulerSolver”
breakpointODESolver	The fully qualified class name for the breakpoint ODE solver class.	string	“ptolemy.domains.ct.kernel.solver.DerivativeResolver”

A CTMultiSolverDirector can direct a model that has composite actors implementing other models of computation. One simulation iteration is done in two phases: the continuous phase and the discrete phase. Let the current iteration be *n*. In the continuous phase, the differential equations are integrated from time t_{n-1} to t_n. After that, in the discrete phase, all (discrete) events which happen at t_n are processed. The step size control mechanism will assure that no events will happen between t_{n-1} and t_n.

7.4.4 CTMixedSignalDirector

This director is designed to be the director when a CT subsystem is contained in an event-based system, like DE or DT. As proved in [57], when a CT subsystem is contained in the DE domain, the CT subsystem should run ahead of the global time, and be ready for rollback. This director implements this optimistic execution.

Since the outside domain is event-based, each time the embedded CT subsystem is fired, the input data are events. In order to convert the events to continuous signals, breakpoints have to be introduced. So this director extends CTMultiSolverDirector, which always has two ODE solvers. There is one more parameter used by this director — the *runAheadLength*, as shown in Table 16 on page 14.

Table 16: Additional Parameter for CTMixedSignalDirector

Name	Description	Type	Default Value
runAheadLength	The maximum length of time for the CT subsystem to run ahead of the global time.	double	1.0

When the CT subsystem is fired, the CTMixedSignalDirector will get the current time τ and the next iteration time τ' from the outer domain, and take the min($\tau - \tau'$, l) as the fire end time, where l is the value of the parameter *maxRunAheadLength*. The execution lasts as long as the fire end time is not reached or an output event is not detected.

This director supports rollback; that is when the state of the continuous subsystem is confirmed (by knowing that no events with a time earlier than the CT current time will be present), the state of the system is marked. If an optimistic execution is known to be wrong, the state of the CT subsystem will roll back to the latest marked state.

7.4.5 CTEmbeddedDirector

This director is used when a CT subsystem is embedded in another continuous time system, either directly or through a hierarchy of finite state machines, like in the hybrid system scenario [59]. This director can pass step size control information up to its executive director. To achieve this, the director must be contained in a CTCompositeActor, which implements the CTStepSizeControlActor interface and can pass the step size control information from the inner domain to the outer domain.

This director extends CTMultiSolverDirector, with no additional parameters. A major difference between this director and the CTMixedSignalDirector is that this director does not support rollback. In fact, when a CT subsystem is embedded in a continuous-time environment, rollback is not necessary.

7.5 Interacting with Other Domains

The CT domain can interact with other domains in Ptolemy II. In particular, we consider interaction among the CT domain, the discrete event (DE) domain and the finite state machine (FSM) domain. Following circuit design communities, we call a composition of CT and DE a *mixed-signal model*; following control and computation communities, we call a composition of CT and FSM a *hybrid system model*.

There are two ways to put CT and DE models together, depending on the containment relation. In either case, event generators and waveform generators are used to convert the two types of signals. Figure 7.4 shows a DE component wrapped by an event generator and a waveform generator. From the input/output point of view, it is a continuous time component. Figure 7.5 shows a CT subsystem wrapped by a waveform generator and an event generator. From the input/output point of view, it is a discrete event component. Notice that event generators and waveform generators always stay in the CT domain.

A hierarchical composition of FSM and CT is shown in figure 7.6. A CT component, by adopting the event generation technique, can have both continuous and discrete signals as its output. The FSM can use predicates on these signals, as well as its own input signals, to build trigger conditions. The actions associated with transitions are usually setting parameters in the destination state, including the initial conditions of integrators.

7.6 CT Domain Demos

Here are some demos in the CT domain showing how this domain works and the interaction with other domains.

7.6.1 Lorenz System

The Lorenz System (see, for example, pp. 213-214 in [24]) is a famous nonlinear dynamic system that shows chaotic attractors. The system is given by:

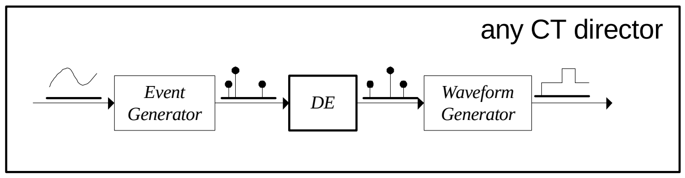


FIGURE 7.4. Embedding a DE component in a CT system.

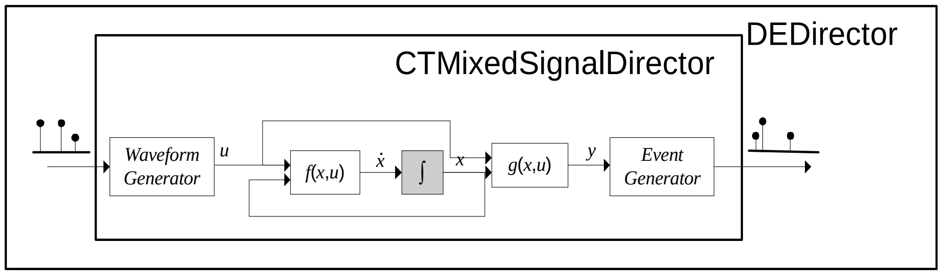


FIGURE 7.5. Embedding a CT component in a DE system.

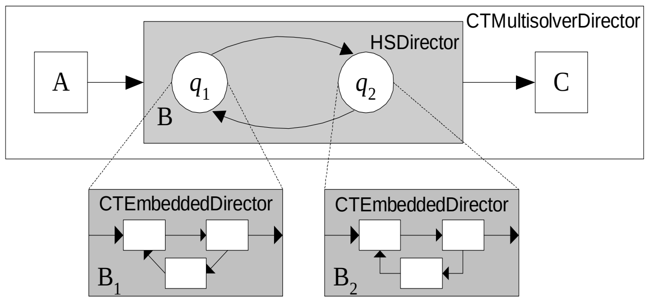


FIGURE 7.6. Hybrid system modeling.

$$\begin{aligned}\dot{x}_1 &= \sigma(x_2 - x_1) \\ \dot{x}_2 &= (\lambda - x_3)x_1 - x_2 \\ \dot{x}_3 &= x_1 \cdot x_2 - b \cdot x_3\end{aligned} \tag{27}$$

The system is built by integrators and stateless domain polymorphic actors, as shown in figure 7.7.

The result of the state trajectory projecting onto the (x_1, x_2) plane is shown in figure 7.8. The initial conditions of the state variables are all 1.0. The default value of the parameters are: $\sigma = 1, \lambda = 25, b = 2.0$.

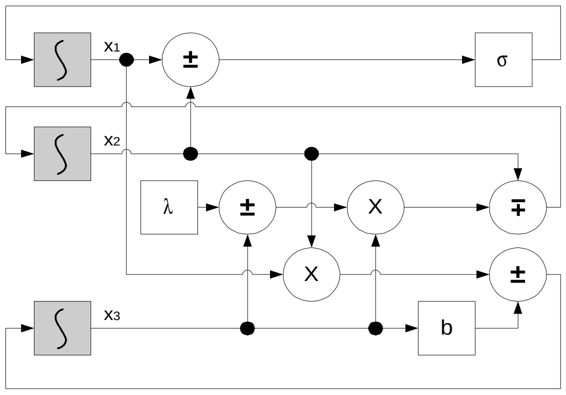
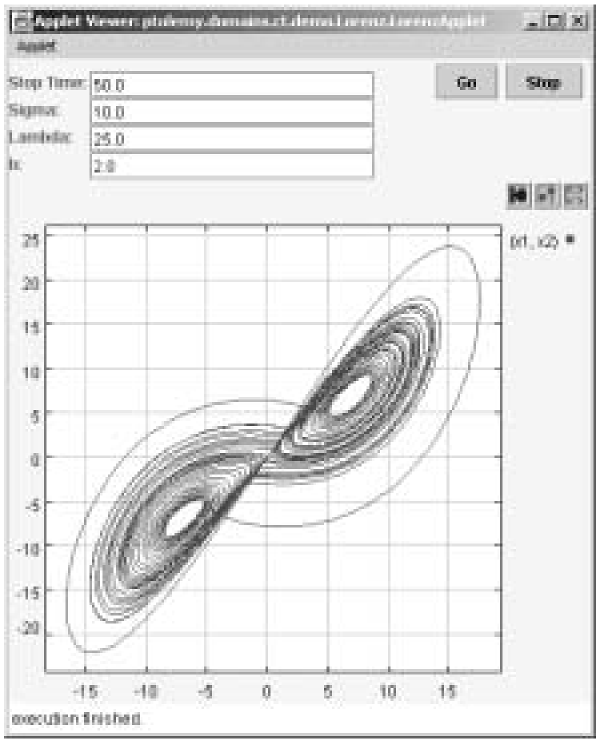


FIGURE 7.7. Block diagram for the Lorenz system.



Stop Time: 50.0
Sigma: 10.0
Lambda: 25.0
b: 2.0

execution finished

FIGURE 7.8. The simulation result of the Lorenz system.

7.6.2 Microaccelerometer with Digital Feedback.

Microaccelerometers are MEMS devices that use beams, gaps, and electrostatics to measure acceleration. Beams and anchors, separated by gaps, form parallel plate capacitors. When the device is accelerated in the sensing direction, the displacement of the beams causes a change of the gap size, which further causes a change of the capacitance. By measuring the change of capacitance (using a capacitor bridge), the acceleration can be obtained accurately. Feedback can be applied to the beams by charging the capacitors. This feedback can reduce the sensitivity to process variations, eliminate mechanical resonances, and increase sensor bandwidth, selectivity, and dynamic range.

Sigma-delta modulation [16], also called pulse density modulation or a bang-bang control, is a digital feedback technique, which also provides the A/D conversion functionality. Figure 7.9 shows the conceptual diagram of system. The central part of the digital feedback is a one-bit quantizer.

We implemented the system as Mark Alan Lemkin designed [56]. As shown in the figure 7.10, the second order CT subsystem is used to model the beam. The voltage on the beam-gap capacitor is sampled every *T* seconds (much faster than the required output of the digital signal), then filtered by a lead compensator (FIR filter), and fed to an one-bit quantizer. The outputs of the quantizer are converted to force and fed back to the beams. The outputs are also counted and averaged every *NT* seconds to produce the digital output. In our example, the external acceleration is a sine wave.

The execution result of the microaccelerometer system is shown in figure 7.11. The upper plot in the figure plots the continuous signals, where the low frequency (blue) sine wave is the acceleration

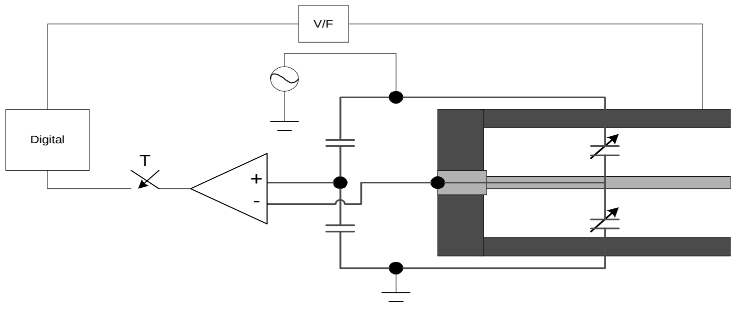


FIGURE 7.9. Micro-accelerator with digital feedback

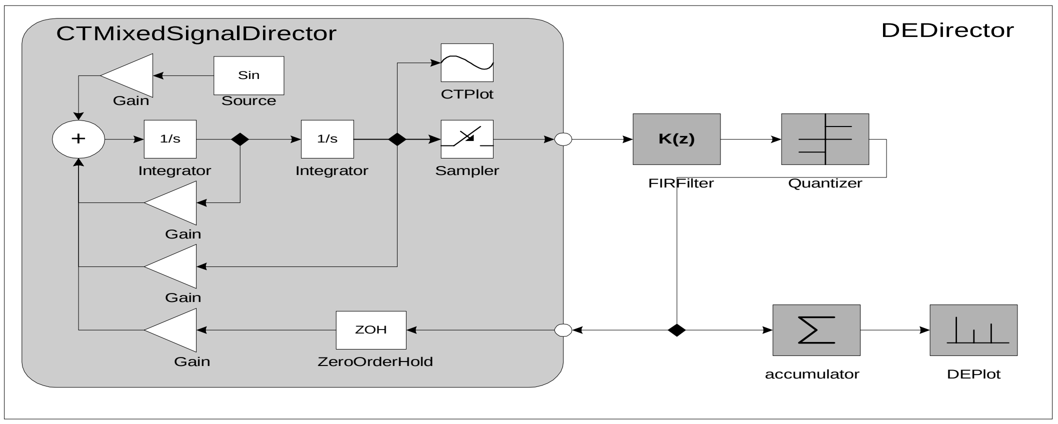
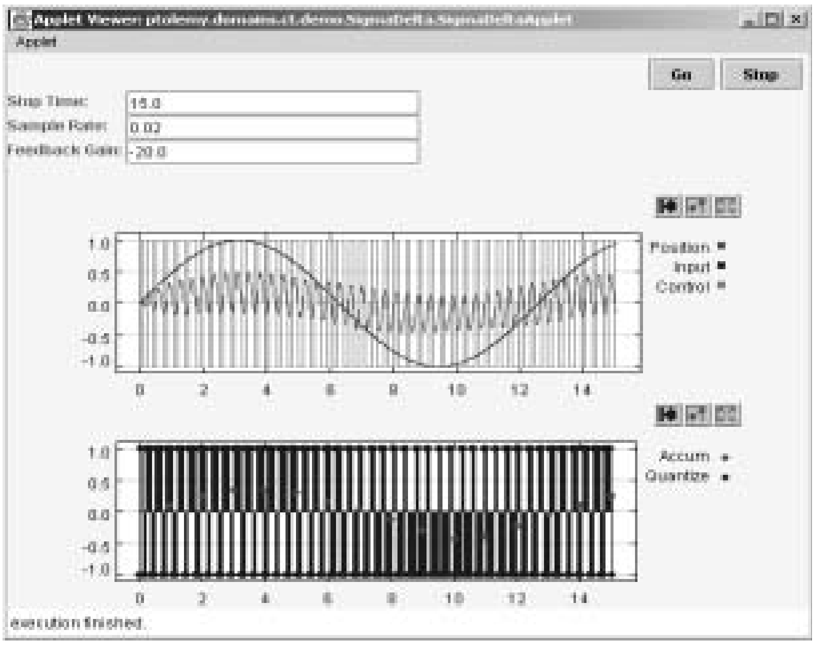


FIGURE 7.10. Block diagram for the micro-accelerator system.

input, the high frequency waveform (red) is the capacitance measurement, and the squarewave (green) is the zero-order hold of the feedback from the digital part. In the lower plot, the dense events (blue) are the quantized samples of the capacitance measurements, which has value +1 or -1, and the sparse events (red) are the accumulation and average of the previous 64 quantized samples. The sparse events are the digital output, and as expected, they have a sinsoidal shape.

7.6.3 Sticky Point Masses System

This sticky point mass demo shows a simple hybrid system. As shown in figure 7.12, there are two point masses on a frictionless table with two springs attaching them to fixed walls. Given initial positions other than the equilibrium points, the point masses oscillate. The distance between the two walls are close enough that the two point masses may collide. The point masses are sticky, in the way so that when they collide, they will sticky together and become one point mass with two springs attached to it. We also assume that the stickiness decays exponentially after the collision, such that eventually the



Applet Viewer: ptolemy.domains.ct.demo.SigmaDelta.SigmaDeltaApplet

Stop Time: 15.0
Sample Rate: 0.02
Feedback Gain: -20.0

execution finished.

FIGURE 7.11. Execution result of the microaccelerometer system.

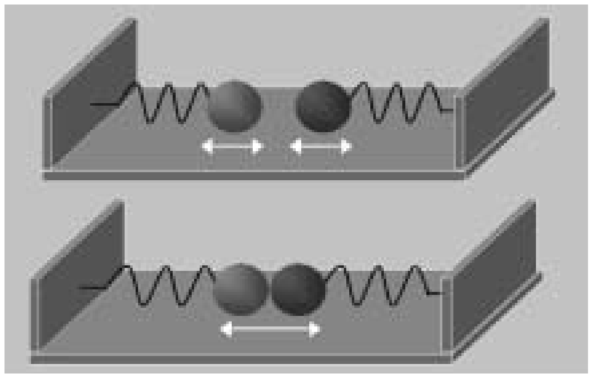


FIGURE 7.12. Sticky point masses system

pulling force between the two springs is big enough to pull the point masses apart. This separation gives the two point masses a new set of initial positions, and they oscillate freely until they collide again.

The system model, as shown in figure 7.13, has three levels of hierarchy –– CT, FSM, and CT. The top level is a continuous time model with two actors, a CTCompositeActor that outputs the position of the two point masses, and a plotter that simply plots the trajectories. The composite actor is a finite state machine with two modes, *separated* and *together.*

In the separated state, there are two differential equations modeling two independently oscillating point masses. There is also an event detection mechanism, implemented by subtracting one position from another and comparing the result to zero. If the positions are equal, within a certain accuracy, then the two point masses collide, and a collision event is generated. This event will trigger a transition from the separated state to the together state. And the actions on the transition set the velocity of the stuck point mass based on Law of Conservation of Momentum.

In the together state, there is one differential equation modeling the stuck point masses, and another first order differential equation modeling the exponentially decaying stickiness. There is another expression computing the pulling force between the two springs. The guard condition from the together state to the separated state compares the pulling force to the stickiness. If the pulling force is bigger than the stickiness, then the transition is taken. The velocities of the two separated point masses equal to their velocities before the separation. The simulation result is shown in figure 7.14, where the position of the two point masses are plotted.

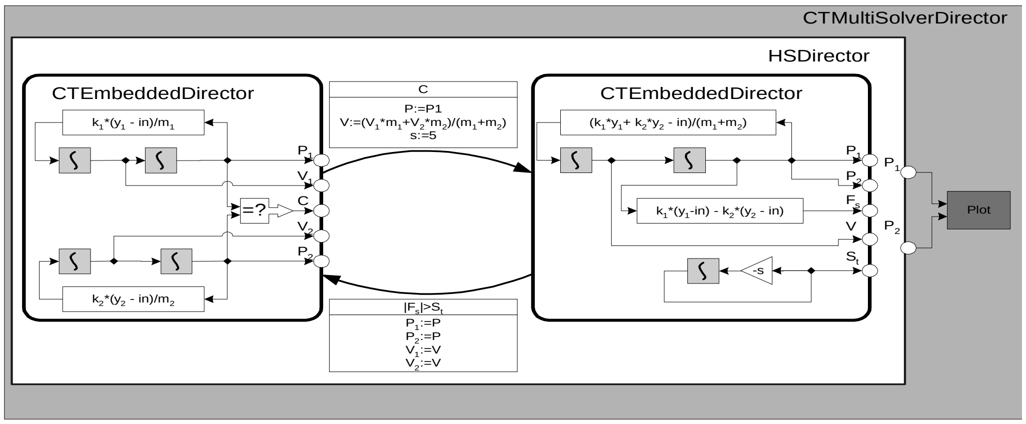


FIGURE 7.13. Modeling sticky point masses.

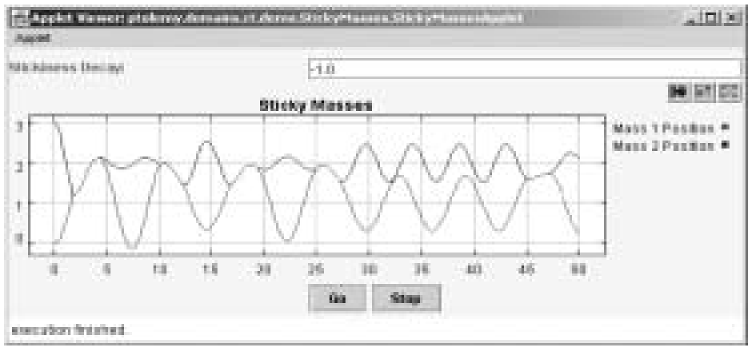


FIGURE 7.14. The simulation result of the sticky point masses system.

7.7 Implementation

The CT domain consists of the following packages, ct.kernel, ct.kernel.util, ct.kernel.solver, and ct.lib, as shown in figure 7.15.

7.7.1 ct.kernel.util package

The ct.kernel.util package provides a basic data structure — `TotallyOrderedSet`, which is used to store breakpoints. The UML for this package is shown in figure 7.16. A totally ordered set is a set (i.e. no duplicated elements) in which the elements are totally comparable. This data structure is used to store breakpoints since breakpoints are processed in their chronological order.

7.7.2 ct.kernel package

The ct.kernel package is the key package of the CT domain. It provides interfaces to classify actors, scheduler, director, and a base class for ODE solvers. The interfaces are used by the scheduler to generate schedules. The classes, including the `CTBaseIntegrator` class and the `ODESolver`

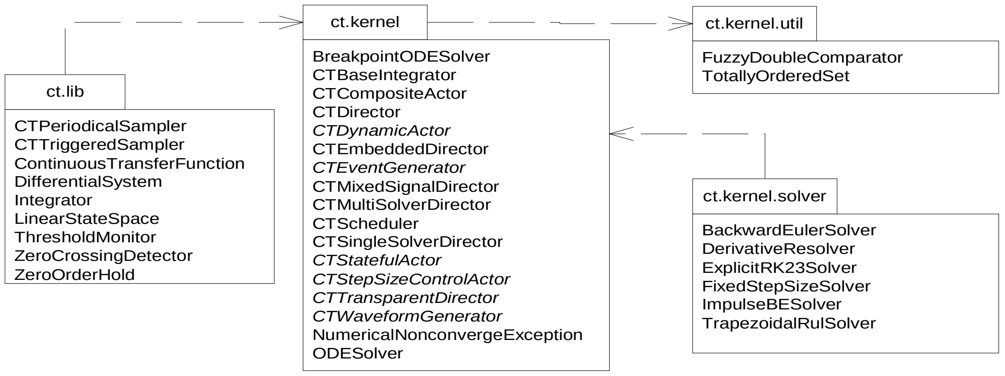


FIGURE 7.15. The packages in the CT domain.

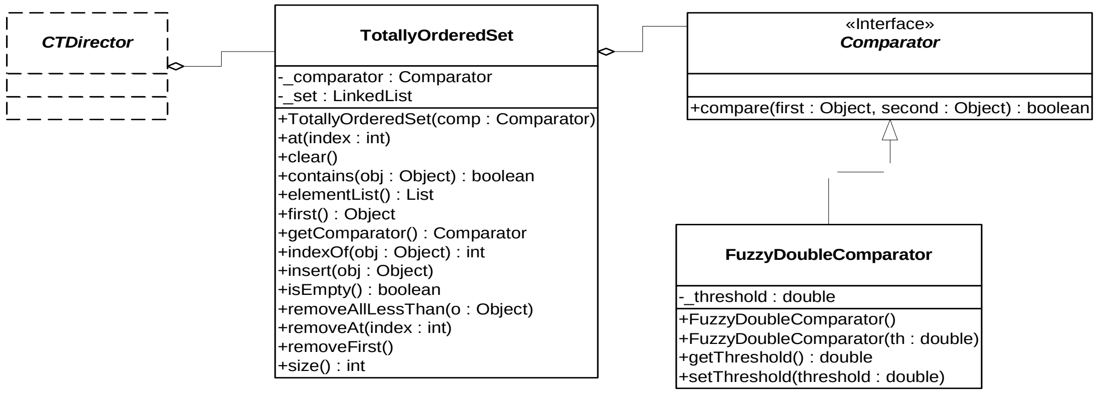


FIGURE 7.16. UML for ct.kernel.util package.

class, are shown in figure 7.17. Here, we use the delegation and the strategy design patterns [33][28] in the `CTBaseIntegrator` and the `ODESolver` classes to support seamlessly changing ODE solvers without reconstructing integrators. The execution methods of the `CTBaseIntegrator` class are delegated to the `ODESolver` class, and subclasses of `ODESolver` provide the concrete implementations of these methods, depending on the ODE solving algorithms.

CT directors implement the semantics of the continuous time execution. As shown in figure 7.18, directors that are used in different scenarios derive from the `CTDirector` base class. The `CTScheduler` class provides schedules for the directors.

The ct.kernel.solver package provides a set of ODE solvers. The classes are shown in figure 7.19. In order for the directors to choose among ODE solvers freely during the execution, the strategy design pattern is used again. A director class talks to the abstract `ODESolver` base class and individual ODE solver classes extend the `ODESolver` to provide concrete strategies.

7.7.3 Scheduling

This section and the following three sections provide technical details and design decisions made in the implementation of the CT domain. These details are only necessary if the readers want to implement new directors or ODE solvers.

In general, simulating a continuous-time system (3)-(5) by a time-marching ODE solver involves

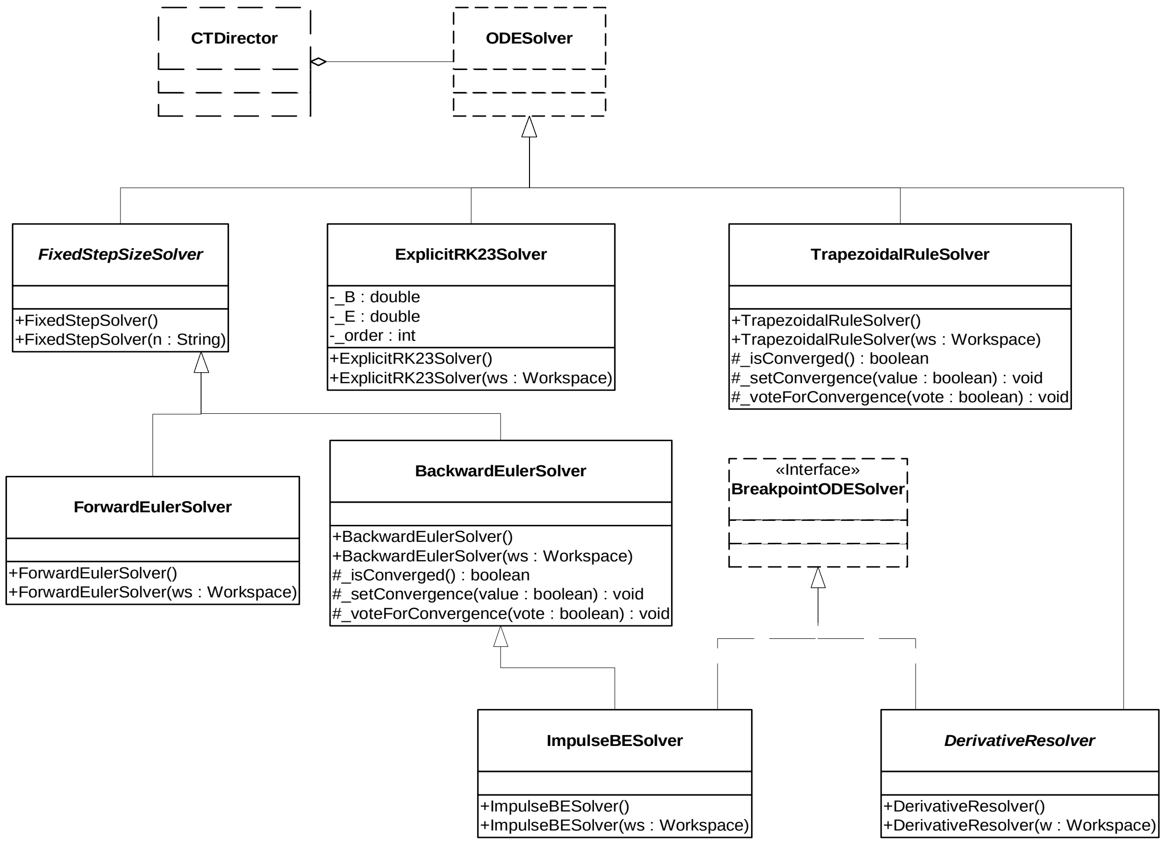

CTDirector
ODESolver
FixedStepSizeSolver
+FixedStepSolver()
+FixedStepSolver(n : String)
ExplicitRK23Solver
-_B : double
-_E : double
-_order : int
+ExplicitRK23Solver()
+ExplicitRK23Solver(ws : Workspace)
TrapezoidalRuleSolver
+TrapezoidalRuleSolver()
+TrapezoidalRuleSolver(ws : Workspace)
#_isConverged() : boolean
#_setConvergence(value : boolean) : void
#_voteForConvergence(vote : boolean) : void
ForwardEulerSolver
+ForwardEulerSolver()
+ForwardEulerSolver(ws : Workspace)
BackwardEulerSolver
+BackwardEulerSolver()
+BackwardEulerSolver(ws : Workspace)
#_isConverged() : boolean
#_setConvergence(value : boolean) : void
#_voteForConvergence(vote : boolean) : void
«Interface»
BreakpointODESolver
ImpulseBESolver
+ImpulseBESolver()
+ImpulseBESolver(ws : Workspace)
DerivativeResolver
+DerivativeResolver()
+DerivativeResolver(w : Workspace)

FIGURE 7.19. UML for ct.kernel.solver package.

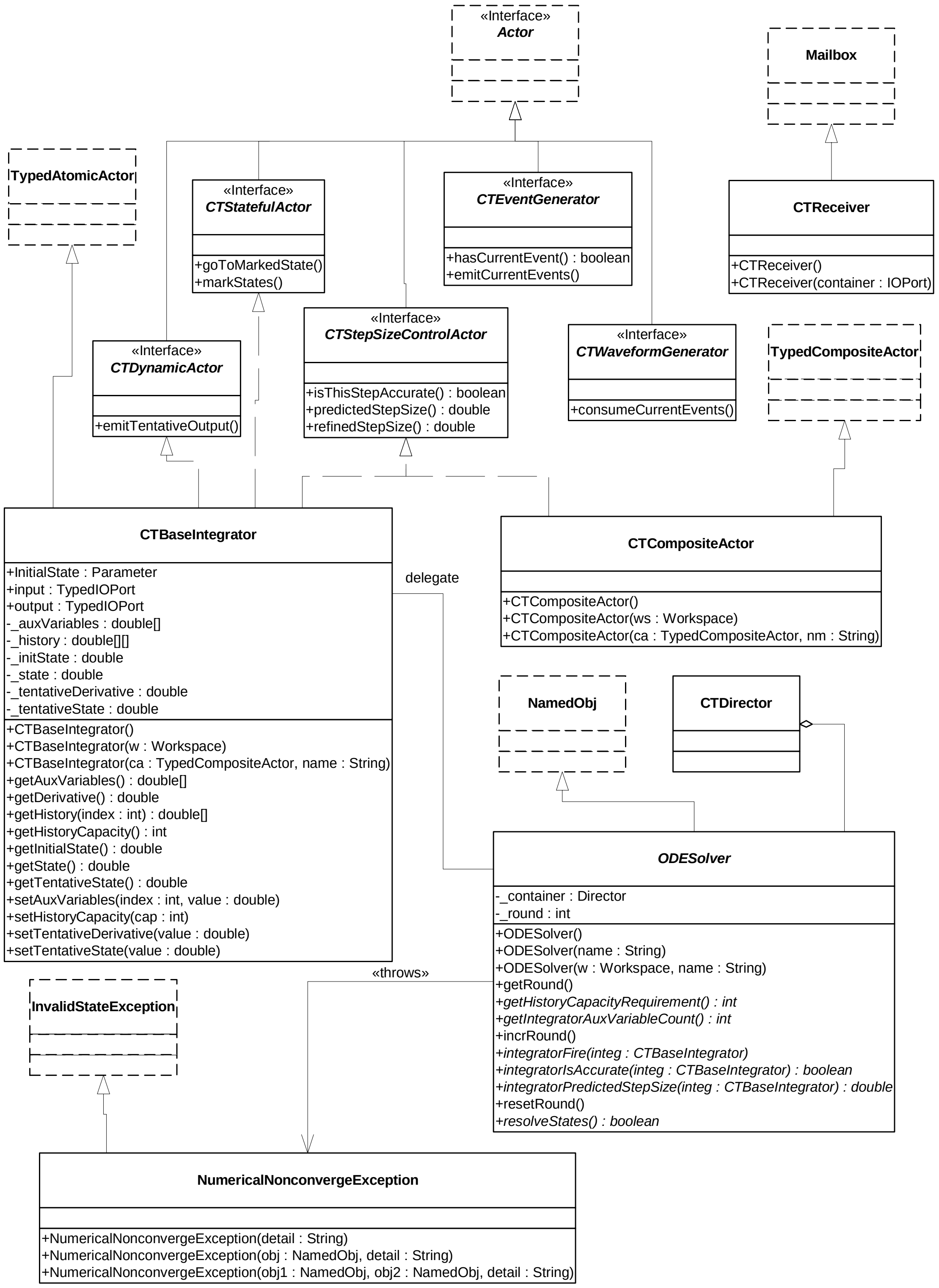

«Interface»
Actor
Mailbox
TypedAtomicActor
«Interface»
CTStatefulActor
+goToMarkedState()
+markStates()
«Interface»
CTEventGenerator
+hasCurrentEvent() : boolean
+emitCurrentEvents()
CTReceiver
+CTReceiver()
+CTReceiver(container : IOPort)
«Interface»
CTDynamicActor
+emitTentativeOutput()
«Interface»
CTStepSizeControlActor
+isThisStepAccurate() : boolean
+predictedStepSize() : double
+refinedStepSize() : double
«Interface»
CTWaveformGenerator
+consumeCurrentEvents()
TypedCompositeActor
CTBaseIntegrator
+InitialState : Parameter
+input : TypedIOPort
+output : TypedIOPort
-_auxVariables : double[]
-_history : double[][]
-_initState : double
-_state : double
-_tentativeDerivative : double
-_tentativeState : double
+CTBaseIntegrator()
+CTBaseIntegrator(w : Workspace)
+CTBaseIntegrator(ca : TypedCompositeActor, name : String)
+getAuxVariables() : double[]
+getDerivative() : double
+getHistory(index : int) : double[]
+getHistoryCapacity() : int
+getInitialState() : double
+getState() : double
+getTentativeState() : double
+setAuxVariables(index : int, value : double)
+setHistoryCapacity(cap : int)
+setTentativeDerivative(value : double)
+setTentativeState(value : double)
delegate
CTCompositeActor
+CTCompositeActor()
+CTCompositeActor(ws : Workspace)
+CTCompositeActor(ca : TypedCompositeActor, nm : String)
NamedObj
CTDirector
ODESolver
-_container : Director
-_round : int
+ODESolver()
+ODESolver(name : String)
+ODESolver(w : Workspace, name : String)
+getRound()
+getHistoryCapacityRequirement() : int
+getIntegratorAuxVariableCount() : int
+incrRound()
+integratorFire(integ : CTBaseIntegrator)
+integratorIsAccurate(integ : CTBaseIntegrator) : boolean
+integratorPredictedStepSize(integ : CTBaseIntegrator) : double
+resetRound()
+resolveStates() : boolean
«throws»
InvalidStateException
NumericalNonconvergeException
+NumericalNonconvergeException(detail : String)
+NumericalNonconvergeException(obj : NamedObj, detail : String)
+NumericalNonconvergeException(obj1 : NamedObj, obj2 : NamedObj, detail : String)

FIGURE 7.17. UML for ct.kernel package, actor related classes.

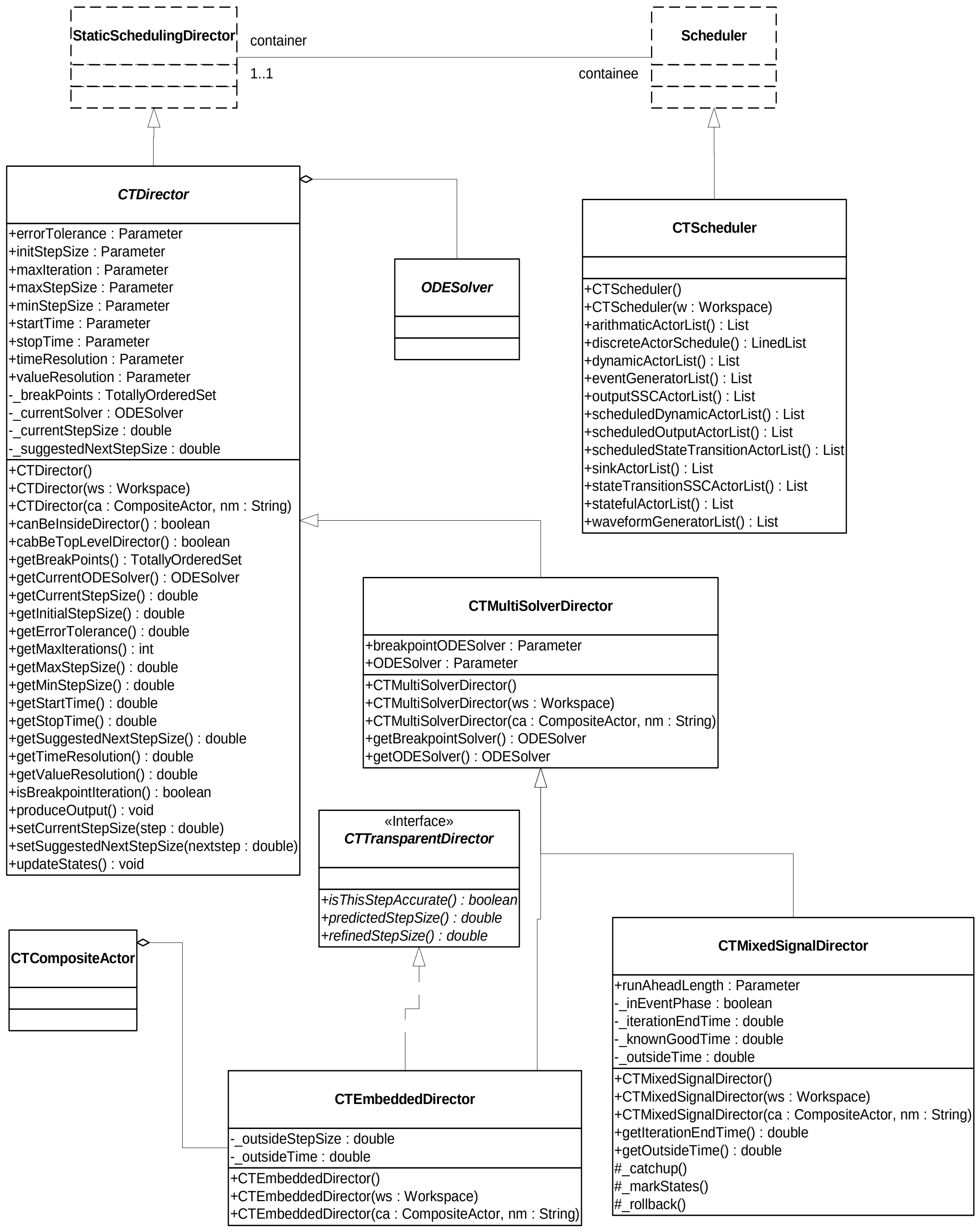

StaticSchedulingDirector
container
1..1
containee
Scheduler
CTDirector
+errorTolerance : Parameter
+initStepSize : Parameter
+maxIteration : Parameter
+maxStepSize : Parameter
+minStepSize : Parameter
+startTime : Parameter
+stopTime : Parameter
+timeResolution : Parameter
+valueResolution : Parameter
-_breakPoints : TotallyOrderedSet
-_currentSolver : ODESolver
-_currentStepSize : double
-_suggestedNextStepSize : double
+CTDirector()
+CTDirector(ws : Workspace)
+CTDirector(ca : CompositeActor, nm : String)
+canBeInsideDirector() : boolean
+cabBeTopLevelDirector() : boolean
+getBreakPoints() : TotallyOrderedSet
+getCurrentODESolver() : ODESolver
+getCurrentStepSize() : double
+getInitialStepSize() : double
+getErrorTolerance() : double
+getMaxIterations() : int
+getMaxStepSize() : double
+getMinStepSize() : double
+getStartTime() : double
+getStopTime() : double
+getSuggestedNextStepSize() : double
+getTimeResolution() : double
+getValueResolution() : double
+isBreakpointIteration() : boolean
+produceOutput() : void
+setCurrentStepSize(step : double)
+setSuggestedNextStepSize(nextstep : double)
+updateStates() : void
ODESolver
CTScheduler
+CTScheduler()
+CTScheduler(w : Workspace)
+arithmaticActorList() : List
+discreteActorSchedule() : LinedList
+dynamicActorList() : List
+eventGeneratorList() : List
+outputSSCActorList() : List
+scheduledDynamicActorList() : List
+scheduledOutputActorList() : List
+scheduledStateTransitionActorList() : List
+sinkActorList() : List
+stateTransitionSSCActorList() : List
+statefulActorList() : List
+waveformGeneratorList() : List
CTMultiSolverDirector
+breakpointODESolver : Parameter
+ODESolver : Parameter
+CTMultiSolverDirector()
+CTMultiSolverDirector(ws : Workspace)
+CTMultiSolverDirector(ca : CompositeActor, nm : String)
+getBreakpointSolver() : ODESolver
+getODESolver() : ODESolver
«Interface»
CTTransparentDirector
+isThisStepAccurate() : boolean
+predictedStepSize() : double
+refinedStepSize() : double
CTCompositeActor
CTMixedSignalDirector
+runAheadLength : Parameter
-_inEventPhase : boolean
-_iterationEndTime : double
-_knownGoodTime : double
-_outsideTime : double
+CTMixedSignalDirector()
+CTMixedSignalDirector(ws : Workspace)
+CTMixedSignalDirector(ca : CompositeActor, nm : String)
+getIterationEndTime() : double
+getOutsideTime() : double
#_catchup()
#_markStates()
#_rollback()
CTEmbeddedDirector
-_outsideStepSize : double
-_outsideTime : double
+CTEmbeddedDirector()
+CTEmbeddedDirector(ws : Workspace)
+CTEmbeddedDirector(ca : CompositeActor, nm : String)

FIGURE 7.18. UML for ct.kernel package, director related classes.

the following execution steps:

1. Given the state of the system $x_{t_0} \ldots x_{t_{n-1}}$ at time points $t_0 \ldots t_{n-1}$, if the current integration step size is h, i.e. $t_n = t_{n-1} + h$, compute the new state x_{t_n} using the numerical integration algorithms. During the application of an integration algorithm, each evaluation of the *f*(*a*, *b*, *t*) function is achieved by the following sequence:
- Integrators emit tokens corresponding to *a*;
- Source actors emit tokens corresponding to *b*;
- The current time is set to *t*;
- The tokens are passed through the topology (in a data-driven way) until they reach the integrators again. The returned tokens are $\dot{x}|_{x=a} = f(a, b, t)$.

2. After the new state x_{t_n} is computed, test whether this step is successful. Local truncation error and unpredictable breakpoints are the issues to be concerned with, since those could lead to an unsuccessful step.

3. If the step is successful, predict the next step size. Otherwise, reduce the step size and try again.

Due to the signal-flow representation of the system, the numerical ODE solving algorithms are implemented as actor firings and token passings under proper scheduling.

The scheduler partitions a CT system into two clusters: the *state transition cluster* and the *output cluster.* In a particular system, these clusters may overlap.

The state transition cluster includes all the actors that are in the signal flow path for evaluating the *f* function in (3). It starts from the source actors and the outputs of the integrators, and ends at the inputs of the integrators. In other words, integrators, and in general dynamic actors, are used to break causality loops in the model. A topological sort of the cluster provides an enumeration of actors in the order of their firings. This enumeration is called the *state transition schedule*. After the integrators produce tokens representing x_t, one iteration of the state transition schedule gives the tokens representing $\dot{x}_t = f(x_t, u(t), t)$ back to the integrators.

The output cluster consists of actors that are involved in the evaluation of the output map *g* in (4). It is also similarly sorted in topological order. The *output schedule* starts from the source actors and the integrators, and ends at the sink actors.

For example, for the system shown in figure 7.3, the state transition schedule is

```
U-G1-G2-G3-A
```

where the order of `G1`, `G2`, and `G3` are interchangeable. The output schedule is

```
G4-Y
```

The event generating schedule is empty.

A special situation that must be taken care of is the firing order of a chain of integrators, as shown in figure 7.20. For the implicit integration algorithms, the order of firings determines two distinct kinds of fixed point iterations. If the integrators are fired in the topological order, namely $x_1 \rightarrow x_2$ in our example, the iteration is called the *Gauss-Seidel iteration*. That is, x_2 always uses the new guess from

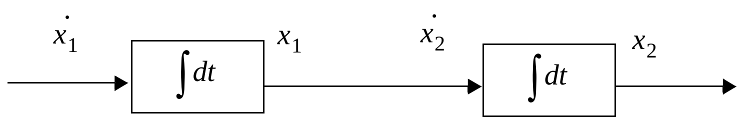


FIGURE 7.20. A chain of integrators.

x_1 in this iteration for its new guess. On the other hand, if they are fired in the reverse topological order, the iteration is called the *Gauss-Jacobi iteration*, where x_2 uses the tentative output from x_1 in the last iteration for its new estimation. The two iterations both have their pros and cons, which are thoroughly discussed in [71]. Gauss-Seidel iteration is considered faster in the speed of convergence than Gauss-Jacobi. For explicit integration algorithms, where the new states x_{t_n} are calculated solely from the history inputs up to $\dot{x}_{t_{n-1}}$, the integrators must be fired in their reverse topological order. For simplicity, the scheduler of the CT domain, at this time, always returns the reversed topological order of a chain of integrators. This order is considered safe for all integration algorithms.

7.7.4 Controlling Step Sizes

Choosing the right time points to approximate a continuous time system behavior is one of the major tasks of simulation. There are three factors that may impact the choice of the step size.

- *Error control*. For all integration algorithms, the *local error* at time t_n is defined as a vector norm (say, the 2-norm) of the difference between the actual solution $x(t_n)$ and the approximation x_{t_n} calculated by the integration method, given that the last step is accurate. That is, assuming $x_{t_{n-1}} = x(t_{n-1})$ then

$$E_{t_n} = \left\| x_{t_n} - x(t_n) \right\| . \qquad (28)$$

  It can be shown that by carefully choosing the parameters in the integration algorithms, the local error is approximately of the *p*-th order of the step size, where *p*, an integer closely related to the number of *f* function evaluations in one integration step, is called the *order* of the integration algorithm, i.e. $E_{t_n} \sim O((t_n - t_{n-1})^p)$. Therefore, in order to achieve an accurate solution, the step size should be chosen to be small. But on the other hand, small step sizes means long simulation time. In general, the choice of step size reflects the trade-off between speed and accuracy of a simulation.
- *Convergence*. The local contraction mapping theorem (Theorem 2 in Appendix D) shows that for implicit ODE solvers, in order to find the fixed point at t_n, the map $F_I(\ )$ in (15) must be a (local) contraction map, and the initial guess must be within an ε ball (the contraction radius) of the solution. It can be shown that $F_I(\ )$ can be made contractive if the step size is small enough. (The choice of the step size is closely related to the Lipschitz constant). So the general approach for resolving the fixed point is that if the iterating function $F_I(\ )$ does not converge at one step size, then reduce the step size by half and try again.
- *Discontinuity.* At discontinuous points, the derivatives of the signals are not continuous, so the integration formula is not applicable. That means the discontinuous points can not be crossed by one integration step. In particular, suppose the current time is *t* and the intended next time point is *t+h.* If there is a discontinuous point at $t + \delta$, where $\delta < h$, then the next step size should be reduced to $t + \delta$. For a predictable breakpoint, the director can adjust the step size accordingly before starting an integration step. However for an unpredictable breakpoint, which is reported "missed" after an integration step, the director should be able to discard its last step and restart with a smaller step size to locate the actual discontinuous point.

Notice that convergence and accuracy concerns only apply to some ODE solvers. For example, explicit algorithms do not have the convergence problem, and fixed step size algorithms do not have the error control capability. On the other hand, discontinuity control is a generic feature that is independent on the choice of ODE solvers.

7.7.5 Mixed-Signal Execution

DE inside CT.

Since time advances monotonically in CT and events are generated chronologically, the DE component receives input events monotonically in time. In addition, a composition of causal DE components is causal [50], so the time stamps of the output events from a DE component are always greater than or equal to the global time. From the view point of the CT system, the events produced by a DE component are predictable breakpoints.

Note that in the CT model, finding the numerical solution of the ODE at a particular time is semantically an instantaneous behavior. During this process, the behavior of all components, including those implemented in a DE model, should keep unchanged. This implies that the DE components should not be executed during one integration step of CT, but only between two successive CT integration steps.

CT inside DE.

When a CT component is contained in a DE system, the CT component is required to be causal, like all other components in the DE system. Let the CT component have local time t, when it receives an input event with time stamp τ. Since time is continuous in the CT model, it will execute from its local time t, and may generate events at any time greater or equal to t. Thus we need

$$t \geq \tau \tag{29}$$

to ensure causality. This means that the local time of the CT component should always be greater than or equal to the global time whenever it is executed.

This ahead-of-time execution implies that the CT component should be able to remember its past states and be ready to rollback if the input event time is smaller than its current local time. The state it needs to remember is the state of the component after it has processed an input event. Consequently, the CT component should not emit detected events to the outside DE system before the global time reaches the event time. Instead, it should send a pure event to the DE system at the event time, and wait until it is safe to emit it.

7.7.6 Hybrid System Execution

Although FSM is an untimed model, its composition with a timed model requires it to transfer the notion of time from its external model to its internal model. During continuous evolution, the system is simulated as a CT system where the FSM is replaced by the continuous component refining the current FSM state. After each time point of CT simulation, the triggers on the transitions starting from the current FSM state are evaluated. If a trigger is enabled, the FSM makes the corresponding transition. The continuous dynamics of the destination state is initialized by the actions on the transition. The simulation continues with the transition time treated as a breakpoint.

Appendix D: Brief Mathematical Background

Theorem 1. [Existence and uniqueness of the solution of an ODE] Consider the initial value ODE problem

$$\begin{aligned} \dot{x} &= f(x, t) \quad . \\ x(t_0) &= x_0 \end{aligned} \tag{30}$$

If *f* satisfies the conditions:

1. [*Continuity Condition*] Let *D* be the set of possible discontinuity points; it may be empty. For each fixed $x \in \Re^n$ and $u \in \Re^m$, the function $f{:}\Re \setminus \mathrm{D} \to \Re^n$ in (30) is continuous. And $\forall \tau \in D$, the left-hand and right-hand limit $f(x, u, \tau^-)$ and $f(x, u, \tau^+)$ are finite.
2. [*Lipschitz Condition*] There is a piecewise continuous bounded function $k:\Re \to \Re^+$, where $\Re^+$ is the set of non-negative real numbers, such that $\forall t \in \Re, \forall \zeta, \xi \in \Re^n, \forall u \in \Re^m$

$$\| f(\xi, u, t) - f(\zeta, u, t) \| \le k(t) \| \xi - \zeta \| \, . \tag{31}$$

Then, for each initial condition $(t_0, x_0) \subseteq \Re \times \Re^n$ there exists a *unique* continuous function $\psi\colon \Re \to \Re^n$ such that,

$$\psi(t_0) = x_0 \tag{32}$$

and

$$\dot{\psi}(t) = f(\psi(t), u(t), t) \qquad \forall t \in \Re \setminus \mathrm{D} \, . \tag{33}$$

This function $\psi(t)$ is called the *solution* through (t_0, x_0) of the ODE (30).

◆

Theorem 2. [Contraction Mapping Theorem.] If $F{:}\Re^n \to \Re^n$ is a local contraction map at *x* with contraction radius ε, then there exists a unique fixed point of *F* within the ε ball centered at *x*. I.e. there exists a unique $\sigma \in \Re^n$, $\|\sigma - x\| \le \varepsilon$, such that $\sigma = F(\sigma)$. And $\forall \sigma_0 \in \Re^n$, $\|\sigma_0 - x\| \le \varepsilon$, the sequence

$$\sigma_1 = F(\sigma_0), \sigma_2 = F(\sigma_1), \sigma_3 = F(\sigma_2), \ldots \tag{34}$$

converges to σ.

8 DE Domain

Authors: *Lukito Muliadi*
Edward A. Lee

8.1 Introduction

The discrete-event (DE) domain supports time-oriented models of systems such as queueing systems, communication networks, and digital hardware. In this domain, actors communicate by sending *events*, where an event is a data value (a token) and a *time stamp*. A DE scheduler ensures that events are processed chronologically according to this time stamp by firing those actors whose available input events are the oldest (having the earliest time stamp of all pending events).

A key strength in our implementation is that simultaneous events (those with identical time stamps) are handled systematically and deterministically. A second key strength is that the global event queue uses an efficient structure that minimizes the overhead associated with maintaining a sorted list with a large number of events.

8.1.1 Model Time

In the DE model of computation, time is *global*, in the sense that all actors share the same global time. The *current time* of the model is often called the *model time* or *simulation time* to avoid confusion with current real time.

As in most Ptolemy II domains, actors communicate by sending tokens through ports. Ports can be input ports, output ports, or both. Tokens are sent by an output port and received by all input ports connected to the output port through relations. When a token is sent from an output port, it is packaged as an event and stored in a global event queue. By default, the time stamp of an output is the model time, although specialized DE actors can produce events with future time stamps.

Actors may also request that they be fired at some time in the future by calling the fireAt() method of the director. This places a *pure event* (one with a time stamp, but no data) on the event queue. A pure event can be thought of as setting an alarm clock to be awakened in the future. Sources (actors with no inputs) are thus able to be fired despite having no inputs to trigger a firing. Moreover, actors that introduce *delay* (outputs have larger time stamps than the inputs) can use this mechanism to schedule a firing in the future to produce an output.

In the global event queue, events are sorted based on their time stamps. An event is removed from the global event queue when the *model time* reaches its time stamp, and if it has a data token, then that token is put into the destination input port.

At any point in the execution of a model, the events stored in the global event queue have time stamps greater than or equal to the model time. The DE director is responsible for advancing (i.e. incrementing) the model time when all events with time stamps equal to the current model time have been processed (i.e. the global event queue only contains events with time stamps strictly greater than the current time). The current time is advanced to the smallest time stamp of all events in the global event queue.

8.1.2 Simultaneous events

An important aspect of a DE domain is the prioritizing of simultaneous events. This gives the domain a dataflow-like behavior for events with identical time stamps. It is done by assigning a *depth* to each actor and a *microstep* to each phase of execution within a given time stamp. Each depth is a non-negative integer, uniquely assigned; i.e. no two actors are assigned the same depth.

The depth of an actor determines the *priority* of events destined to that actor, relative to other events with the same time stamp and the same microstep. The highest priority events are those destined to actors with the lowest depth.

Consider the simple topology shown in figure 8.1. Assume that actor *Y* is not a delay actor, meaning that its output events have the same time stamp and microstep as its input events (this is suggested by the dotted arrow). Suppose that actor *X* produces an event with time stamp τ. That event is available at ports *B* and *D*, so the scheduler could choose to fire actors *Y* or *Z*. Which should it fire? Intuition tells us it should fire the upstream one first, *Y*, because that firing may produce another event with time stamp τ at port *D* (which is presumably a multiport). It seems logical that if actor *Z* is going to get one event on each input channel with the same time stamp, then it should see those events in the same firing. Thus, if there are simultaneous events at *B* and *D*, then the one at *B* will have higher priority.

The depths are determined by a *topological sort* of a *directed acyclic graph* (DAG) of the actors. The DAG of actors follows the topology of the graph, except when there are declared delays. Once the DAG is constructed, it is sorted topologically. This simply means that an ordering of actors is assigned such that an upstream actor in the DAG is earlier in the ordering than a downstream actor. The depth of an actor is defined to be its position in this topological sort, starting with zero. For example, in figure 8.1, *X* will have depth 0, *Y* will have depth 1, and *Z* will have depth 2.

In general, a DAG has several correct topological sorts. The topological sort is not unique, mean-

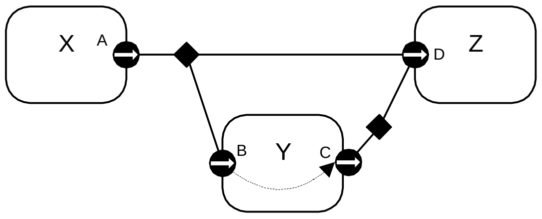


FIGURE 8.1. If there are simultaneous events at B and D, then the one at B will have higher priority because it may trigger another simultaneous event at D.

ing that the depths assigned to actors are somewhat arbitrary. But an upstream actor will always have a lower depth than a downstream actor, unless there is an intervening delay actor. Thus, given simultaneous input events with the same microstep, an upstream actor will always fire before a downstream actor. Such a strategy ensures that the execution is *deterministic*, assuming the actors only communicate via events. In other words, even though there are several possible choices that a scheduler could make for an ordering of firings, all choices that respect the priorities yield the same results.

There are situations where constructing a DAG following the topology is not possible. Consider the topology shown in figure 8.2. It is evident from the figure that the topology is not acyclic. Indeed, figure 8.2 depicts a *zero-delay loop* where topological sort cannot be done. The director will refuse to run the model, and will terminate with an error message.

The TimedDelay actor in DE is a domain-specific actor that asserts a delay relationship between its input and output. Thus, if we insert a TimedDelay actor in the loop, as shown in figure 8.3, then constructing the DAG becomes once again possible. The TimedDelay actor breaks the precedences.

Note in particular that the TimedDelay actor breaks the precedences *even if its delay parameter is set to zero*. Thus, the DE domain is perfectly capable of modeling feedback loops with zero time delay, but the model builder has to specify the order in which events should be processed by placing a TimedDelay actor with a zero value for its parameter.

8.1.3 Iteration

At each iteration, after advancing the current time, the director chooses all events in the global event queue that have the smallest time stamps, microstep, and depth (tested in that order). The chosen events are then removed from the global event queue and their data tokens are inserted into the appropriate input ports of the destination actor. Then, the director iterates the destination actor; i.e. it invokes prefire(), fire(), and postfire(). All of these events are destined to the same actor, since the depth is unique for each actor.

A firing may produce additional events at the current model time (the actor reacts *instantaneously*, or has *zero delay*). There also may be other events with time stamp equal to the current model time still pending on the event queue. The DE director repeats the above procedure until there are no more

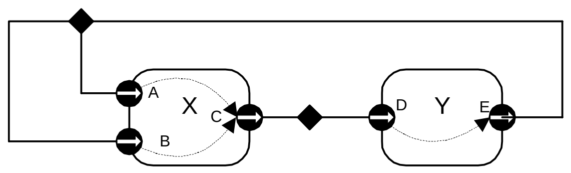


FIGURE 8.2. An example of a directed zero-delay loop.

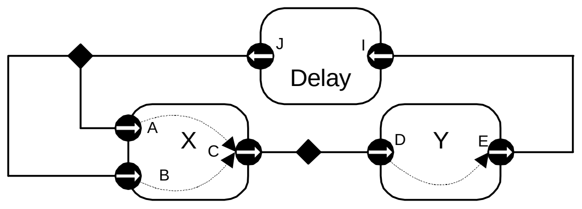


FIGURE 8.3. A Delay actor can be used to break a zero-delay loop.

events with time stamp equal to the current time. This concludes one iteration of the model. An iteration, therefore, processes all events on the event queue with the smallest time stamp.

8.1.4 Getting a Model Started

Before one of the iterations described above can be run, there have to be initial events in the global event queue. Actors may produce initial pure events or regular output events in their initialize() method. Thus, to get a model started, at least one actor must produce events. All the domain-polymorphic timed sources described in the Actor Libraries chapter produce pure events, so these can be used in DE. We can define the *start time* to be the smallest time stamp of these initial events.

8.1.5 Pure Events at the Current Time

An actor calls fireAt() to schedule a pure event. The pure event is a request to the scheduler to fire the actor sometime in the future. However, the actor may choose to call fireAt() with the time argument equal to the current time. In fact, the preferred method for domain-polymorphic source actors to get started is to have code like the following in their initialize() method:

```
    Director director = getDirector();
    director.fireAt(this, director.getCurrentTime());
```

This will schedule a pure event on the event queue with microstep zero and depth equal to that of the calling actor.

An actor may also call fireAt() with the current time in its fire() method. This is a request to be refired later *in the current iteration*. This is managed by queueing a pure event with microstep one greater than the current microstep. In fact, this is only situation in which the microstep is incremented beyond zero.

8.1.6 Stopping Execution

Execution stops when one of these conditions become true:

- The current time reaches the *stop time*, set by calling the setStopTime() method of the DE director.
- The global event queue becomes empty.

Events at the stop time are processed before stopping the model execution. The execution ends by calling the wrapup() method of all actors.

It is also possible to explicitly invoke the iterate() method of the manager for some fixed number of iterations. Recall that an iteration processes all events with a given time stamp, so this will run the model through a specified number of discrete time steps.

8.2 Overview of The Software Architecture

The UML static structure diagram for the DE kernel package is shown in figure 8.4. For model builders, the important classes are DEDirector, DEActor and DEIOPort. At the heart of DEDirector is a global event queue that sorts events according to their time stamps and priorities.

The DEDirector uses an efficient implementation of the global event queue, a calendar queue data

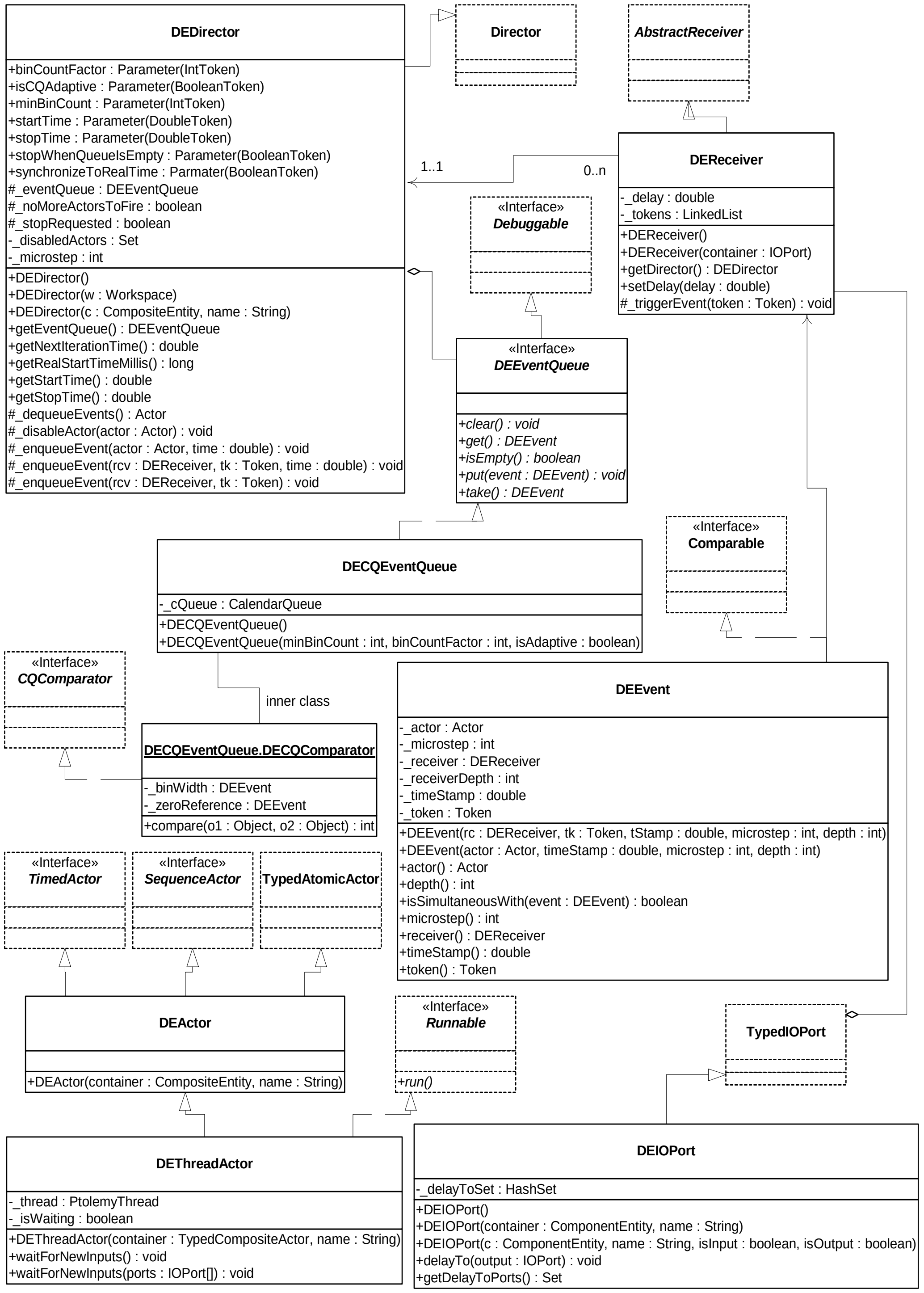

DEDirector
+binCountFactor : Parameter(IntToken)
+isCQAdaptive : Parameter(BooleanToken)
+minBinCount : Parameter(IntToken)
+startTime : Parameter(DoubleToken)
+stopTime : Parameter(DoubleToken)
+stopWhenQueueIsEmpty : Parameter(BooleanToken)
+synchronizeToRealTime : Parmater(BooleanToken)
#_eventQueue : DEEventQueue
#_noMoreActorsToFire : boolean
#_stopRequested : boolean
-_disabledActors : Set
-_microstep : int
+DEDirector()
+DEDirector(w : Workspace)
+DEDirector(c : CompositeEntity, name : String)
+getEventQueue() : DEEventQueue
+getNextIterationTime() : double
+getRealStartTimeMillis() : long
+getStartTime() : double
+getStopTime() : double
#_dequeueEvents() : Actor
#_disableActor(actor : Actor) : void
#_enqueueEvent(actor : Actor, time : double) : void
#_enqueueEvent(rcv : DEReceiver, tk : Token, time : double) : void
#_enqueueEvent(rcv : DEReceiver, tk : Token) : void
Director
AbstractReceiver
1..1
0..n
DEReceiver
-_delay : double
-_tokens : LinkedList
+DEReceiver()
+DEReceiver(container : IOPort)
+getDirector() : DEDirector
+setDelay(delay : double)
#_triggerEvent(token : Token) : void
«Interface»
Debuggable
«Interface»
DEEventQueue
+clear() : void
+get() : DEEvent
+isEmpty() : boolean
+put(event : DEEvent) : void
+take() : DEEvent
«Interface»
Comparable
DECQEventQueue
-_cQueue : CalendarQueue
+DECQEventQueue()
+DECQEventQueue(minBinCount : int, binCountFactor : int, isAdaptive : boolean)
«Interface»
CQComparator
inner class
DECQEventQueue.DECQComparator
-_binWidth : DEEvent
-_zeroReference : DEEvent
+compare(o1 : Object, o2 : Object) : int
DEEvent
-_actor : Actor
-_microstep : int
-_receiver : DEReceiver
-_receiverDepth : int
-_timeStamp : double
-_token : Token
+DEEvent(rc : DEReceiver, tk : Token, tStamp : double, microstep : int, depth : int)
+DEEvent(actor : Actor, timeStamp : double, microstep : int, depth : int)
+actor() : Actor
+depth() : int
+isSimultaneousWith(event : DEEvent) : boolean
+microstep() : int
+receiver() : DEReceiver
+timeStamp() : double
+token() : Token
«Interface»
TimedActor
«Interface»
SequenceActor
TypedAtomicActor
DEActor
+DEActor(container : CompositeEntity, name : String)
«Interface»
Runnable
+run()
TypedIOPort
DEThreadActor
-_thread : PtolemyThread
-_isWaiting : boolean
+DEThreadActor(container : TypedCompositeActor, name : String)
+waitForNewInputs() : void
+waitForNewInputs(ports : IOPort[]) : void
DEIOPort
-_delayToSet : HashSet
+DEIOPort()
+DEIOPort(container : ComponentEntity, name : String)
+DEIOPort(c : ComponentEntity, name : String, isInput : boolean, isOutput : boolean)
+delayTo(output : IOPort) : void
+getDelayToPorts() : Set

FIGURE 8.4. UML static structure diagram for the DE kernel package.

structure [12]. The time complexity for this particular implementation is O(1) in both enqueue and dequeue operations, in theory. This means that the time complexity for enqueue and dequeue operations is independent of the number of pending events in the global event queue. However, to realize this performance, it is necessary for the distribution of events to match certain assumptions. Our calendar queue implementation observes events as they are dequeued and adapts the structure of the queue according to their statistical properties. Nonetheless, the calendar queue structure will not prove optimal for all models. For extensibility, alternative implementations of the global event queue can be realized by implementing the DEEventQueue interface and specifying the event queue using the appropriate constructor for DEDirector.

The DEEvent class carries tokens through the event queue. It contains their time stamp, their microstep, and the depth of the destination actor, as well as a reference to the destination actor. It implements the java.lang.Comparable interface, meaning that any two instances of DEEvent can be compared. The private inner class DECQEventQueue.DECQComparator, which is provided to the calendar queue at the time of its construction, performs the requisite comparisons of events.

The DEActor class provides convenient methods to access time, since time is an essential part of a timed domain like DE. Nonetheless, actors in a DE model are not required to be derived from the DEActor class. Simply deriving from TypedAtomicActor gives you the same capability, but without the convenience. In the latter case, time is accessible through the director.

The DEIOPort class is be used by actors that are specialized to the DE domain. It supports annotations that inform the scheduler about delays through the actor. It also provides two additional methods, overloaded versions of broadcast() and send(). The overloaded versions have a second argument for the time delay, allowing actors to send output data with a time delay (relative to current time).

Domain polymorphic actors, such as those described in the Actor Libraries chapter, have as ports instances of TypedIOPort, not DEIOPort, and therefore cannot produce events in the future directly by sending it through output ports. Note that tokens sent through TypedIOPort are treated as if they were sent through DEIOPort with the time delay argument equal to zero. Domain polymorphic actors can produce events in the future indirectly by using the fireAt() method of the director. By calling fireAt(), the actor requests a refiring in the future. The actor can then produce a delayed event during the refiring.

8.3 The DE Actor Library

The DE domain has a small library of actors in the ptolemy.domains.de.lib package, shown in figure 8.5. These actors are particularly characterized by implementing both the TimedActor and SequenceActor interfaces. These actors use the current model time, and in addition, assume they are dealing with sequences of discrete events. Some of them use domain-specific infrastructure, such as the convenience class DEActor and the base class DETransformer. The DETransformer class provides in input and output port that are instances of DEIOPort. The Delay and Server actors use facilities of these ports to influence the firing priorities. The Merge actor merges events sequences in chronological order.

8.4 Mutations

The DE director tolerates changes to the model during execution. The change should be queued using requestChange(). While invoking those changes, the method invalidateSchedule() is expected to be called, notifying the director that the topology it used to calculate the priorities of the actors is no longer valid. This will result in the priorities being recalculated the next time prefire() is invoked.

An example of a mutation is shown in figures 8.6 and 8.7. Figure 8.7 defines a class that constructs a simple model in its constructor. The model consists of a clock connected to a recorder. The method insertClock() creates an anonymous inner class that extends ChangeRequest. Its execute() method disconnects the two existing actors, creates a new clock and a merge actor, and reconnects the actors as shown in figure 8.6.

When the insertClock() method is called, a change request is queue with the top-level composite actor, which delegates the request to the manager. The manager executes the request after the current iteration completes. Thus, the change will always be executed between non-equal time stamps, since an iteration consists of processing all events at the current time stamp.

Actors that are added in the change request are automatically initialized. Note, however, one sub-

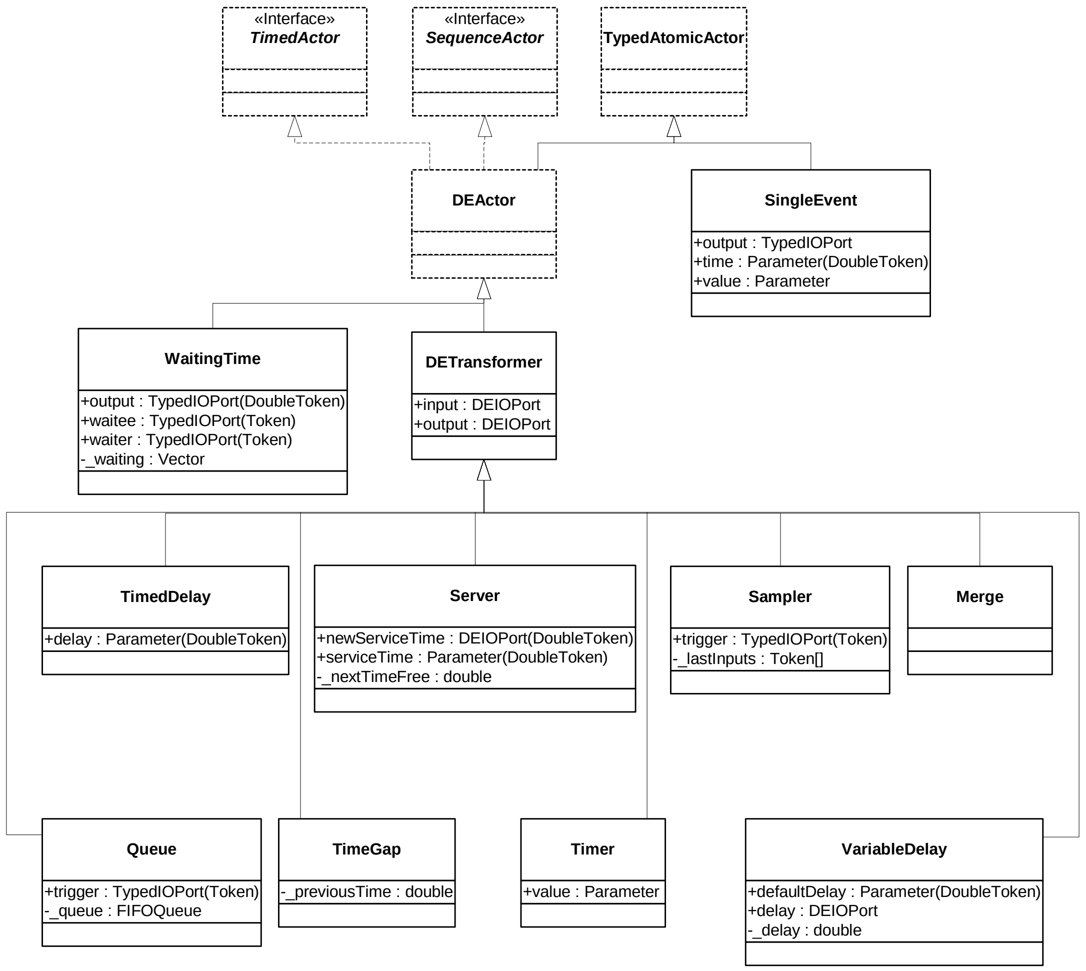


FIGURE 8.5. The library of DE-specific actors.

tlety. The next to last line of the insertClock() method is:

```
    _rec.input.createReceivers();
```

This method call is necessary because the connections of the recorder actor have changed, but since the actor is not new, it will *not* be reinitialized. Recall that the preinitialize() and initialize() methods are guaranteed to be called only once, and one of the responsibilities of the preinitialize() method is to create the receivers in all the input ports of an actor. Thus, whenever connections to an input port change during a mutation, the mutation code itself must call createReceivers() to reconstruct the receivers. Note that this will result in the loss of any tokens that might already be queued in the preexisting receivers of the ports. It is because of this possible loss of data that the creation of receivers is not done automatically. The designer of the mutation should be aware of the possible loss of data.

There is one additional subtlety about mutations. If an actor produces events in the future via DEIOPort, then the destination actor will be fired even if it has been removed from the topology by the time the execution reaches that future time. This may not always be the expected behavior. The Delay actor in the DE library behaves this way, so if its destination is removed before processing delayed events, then it may be invoked at a time when it has no container. Most actors will tolerate this and will not cause problems. But some might have unexpected behavior. To prevent this behavior, the mutation that removes the actor should also call the disableActor() method of the director.

8.5 Writing DE Actors

It is very common in DE modeling to include custom-built actors. No pre-defined actor library seems to prove sufficient for all applications. For the most part, writing actors for the DE domain is no different than writing actors for any other domain. Some actors, however, need to exercise particular control over time stamps and actor priorities. Such actors use instances of DEIOPort rather than TypedIOPort. The first section below gives general guidelines for writing DE actors and domain-polymorphic actors that work in DE. The second section explains in detail the priorities, and in particular, how to write actors that declare delays. The final section discusses actors that operate as a Java thread.

8.5.1 General Guidelines

The points to keep in mind are:

- When an actor fires, not all ports have tokens, and some ports may have more than one token. The

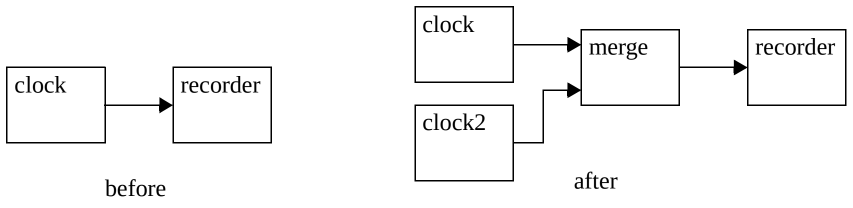


FIGURE 8.6. Topology before and after mutation for the example in figure 8.7.

time stamps of the events that contained these tokens are no longer explicitly available. The current model time is assumed to be the time stamp of the events.

- If the actor leaves unconsumed tokens on its input ports, then it will be iterated again before model

```
package ptolemy.domains.de.lib.test;

import ptolemy.kernel.util.*;
import ptolemy.kernel.*;
import ptolemy.actor.*;
import ptolemy.actor.lib.*;
import ptolemy.domains.de.kernel.*;
import ptolemy.domains.de.lib.*;

public class Mutate {

    public Manager manager;

    private Recorder _rec;
    private Clock _clock;
    private TypedCompositeActor _top;
    private DEDirector _director;

    public Mutate() throws IllegalActionException,
            NameDuplicationException {
       _top = new TypedCompositeActor();
       _top.setName("top");
       manager = new Manager();
       _director = new DEDirector();
       _top.setDirector(_director);
       _top.setManager(manager);

       _clock = new Clock(_top, "clock");
       _clock.values.setExpression("[1.0]");
       _clock.offsets.setExpression("[0.0]");
       _clock.period.setExpression("1.0");
       _rec = new Recorder(_top, "recorder");
       _top.connect(_clock.output, _rec.input);
    }

    public void insertClock() {
       // Create an anonymous inner class
       ChangeRequest change = new ChangeRequest(_top, "test2") {
          public void _execute() throws IllegalActionException,
                NameDuplicationException {
             _clock.output.unlinkAll();
             _rec.input.unlinkAll();
             Clock clock2 = new Clock(_top, "clock2");
             clock2.values.setExpression("[2.0]");
             clock2.offsets.setExpression("[0.5]");
             clock2.period.setExpression("2.0");
             Merge merge = new Merge(_top, "merge");
             _top.connect(_clock.output, merge.input);
             _top.connect(clock2.output, merge.input);
             _top.connect(merge.output, _rec.input);
             // Any pre-existing input port whose connections
             // are modified needs to have this method called.
             _rec.input.createReceivers();
             _director.invalidateSchedule();
          }
       };
       _top.requestChange(change);
    }
}
```

FIGURE 8.7. An example of a class that constructs a model and then mutates it.

time is advanced. This ensures that the current model time is in fact the time stamp of the input events. However, occasionally, an actor will want to leave unconsumed tokens on its input ports, and not be fired again until there is some other new event to be processed. To get this behavior, it should return *false* from prefire(). This indicates to the DE director that it does not wish to be iterated.

- If the actor returns *false* from postfire(), then the director will not fire that actor again. Events that are destined for that actor are discarded.
- When an actor produces an output token, the time stamp for the output event is taken to be the current model time. If the actor wishes to produce an event at a future model time, one way to accomplish this is to call the director's fireAt() method to schedule a future firing, and then to produce the token at that time. A second way to accomplish this is to use instances of DEIOPort and use the overloaded send() or broadcast() methods that take a time delay argument.
- The DEIOPort class (see figure 8.4) can produce events in the future, but there is an important subtlety with using these methods. Once an event has been produced, it cannot be retracted. In particular, even if the actor is deleted before model time reaches that of the future event, the event will be delivered to the destination. If you use fireAt() instead to generate delayed events, then if the actor is deleted (or returns *false* from postfire()) before the future event, then the future event will not be produced.
- By convention in Ptolemy II, actors update their state only in the postfire() method. In DE, the fire() method is only invoked once per iteration, so there is no particular reason to stick to this convention. Nonetheless, we recommend that you do in case your actor becomes useful in other domains. The simplest way to ensure this is follow the following pattern. For each state variable, such as a private variable named *_count*,

  ```
  private int _count;
  ```

  create a shadow variable

  ```
  private int _countShadow;
  ```

  Then write the methods as follows:

  ```
  public void fire() {
     _countShadow = _count;
     ... perform some computation that may modify _countShadow ...
  }
  public boolean postfire() {
     _count = _countShadow;
     return super.postfire();
  }
  ```

  This ensures that the state is updated only in postfire().

  In a similar fashion, delayed outputs (produced by either mechanism) should be produced only in the postfire() method, since a delayed outputs are persistent state. Thus, fireAt() should be called in postfire() only, as should the overloaded send() and broadcast() of DEIOPort.

8.5.2 Examples

Simplified Delay Actor. An example of a domain-specific actor for DE is shown in figure 8.8. This actor delays input events by some amount specified by a parameter. The domain-specific features of the actor are shown in bold. They are:

- It uses DEIOPort rather than TypedIOPort.
- It has the statement:

  ```
  input.delayTo(output);
  ```

  This statement declares to the director that this actor implements a delay from input to output. The actor uses this to break the precedences when constructing the DAG to find priorities.
- It uses an overloaded send() method, which takes a delay argument, to produce the output. Notice that the output is produced in the postfire() method, since by convention in Ptolemy II, persistent state is not updated in the fire() method, but rather is updated in the postfire() method.

```
package ptolemy.domains.de.lib.test;

import ptolemy.actor.TypedAtomicActor;
import ptolemy.domains.de.kernel.DEIOPort;
import ptolemy.data.DoubleToken;
import ptolemy.data.Token;
import ptolemy.data.expr.Parameter;
import ptolemy.actor.TypedCompositeActor;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;
import ptolemy.kernel.util.Workspace;

public class SimpleDelay extends TypedAtomicActor {

    public SimpleDelay(TypedCompositeActor container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        input = new DEIOPort(this, "input", true, false);
        output = new DEIOPort(this, "output", false, true);
        delay = new Parameter(this, "delay", new DoubleToken(1.0));
        delay.setTypeEquals(DoubleToken.class);
        input.delayTo(output);
    }

    public Parameter delay;
    public DEIOPort input;
    public DEIOPort output;
    private Token _currentInput;

    public void fire() throws IllegalActionException {
        _currentInput = input.get(0);
    }

    public boolean postfire() throws IllegalActionException {
        output.send(0, _currentInput,
                ((DoubleToken)delay.getToken()).doubleValue());
        return super.postfire();
    }
}
```

FIGURE 8.8. A domain-specific actor in DE.

Server Actor. The Server actor in the DE library (see figure 8.5) uses a rich set of behavioral properties of the DE domain. A server is a process that takes some amount of time to serve "customers." While it is serving a customer, other arriving customers have to wait. This actor can have a fixed service time (set via the parameter *serviceTime*, or a variable service time, provided via the input port *newServiceTime*). A typical use would be to supply random numbers to the *newServiceTime* port to generate random service times. These times can be provided at the same time as arriving customers to get an effect where each customer experiences a different, randomly selected service time.

The (compacted) code is shown in figure 8.9. This actor extends DETransformer, which has two public members, *input* and *output*, both instances of DEIOPort. The constructor makes use of the delayTo() method of these ports to indicate that the actor introduces delay between its inputs and its output.

The actor keeps track of the time at which it will next be free in the private variable _nextTimeFree. This is initialized to minus infinity to indicate that whenever the model begins executing, the server is free. The prefire() method determines whether the server is free by comparing this private variable against the current model time. If it is free, then this method returns true, indicating to the scheduler that it can proceed with firing the actor. If the server is not free, then the prefire() method checks to see whether there is a pending input, and if there is, requests a firing when the actor will become free. It then returns false, indicating to the scheduler that it does not wish to be fired at this time. Note that the prefire() method uses the methods getCurrentTime() and fireAt() of DEActor, which are simply convenient interfaces to methods of the same name in the director.

The fire() method is invoked only if the server is free. It first checks to see whether the newServiceTime port is connected to anything, and if it is, whether it has a token. If it does, the token is read and used to update the *serviceTime* parameter. No more than one token is read, even if there are more in the input port, in case one token is being provided per pending customer.

The fire() method then continues by reading an input token, if there is one, and updating _nextTimeFree. The input token that is read is stored temporarily in the private variable _currentInput. The postfire() method then produces this token on the output port, with an appropriate delay. This is done in the postfire() method rather than the fire() method in keeping with the policy in Ptolemy II that persistent state is not updated in the fire() method. Since the output is produced with a future time stamp, then it is persistent state.

Note that when the actor will not get input tokens that are available in the fire() method, it is essential that prefire() return false. Otherwise, the DE scheduler will keep firing the actor until the inputs are all consumed, which will never happen if the actor is not consuming inputs!

Like the SimpleDelay actor in figure 8.8, this one produces outputs with future time stamps, using the overloaded send() method of DEIOPort that takes a delay argument. There is a subtlety associated with this design. If the model mutates during execution, and the Server actor is deleted, it cannot retract events that it has already sent to the output. Those events will be seen by the destination actor, even if by that time neither the server nor the destination are in the topology! This could lead to some unexpected results, but hopefully, if the destination actor is no longer connected to anything, then it will not do much with the token.

8.5.3 Thread Actors

In some cases, it is useful to describe an actor as a thread that waits for input tokens on its input ports. The thread suspends while waiting for input tokens and is resumed when some or all of its input ports have input tokens. While this description is functionally equivalent to the standard description

```
package ptolemy.domains.de.lib;
import statements ...
public class Server extends DETransformer {

   public DEIOPort newServiceTime;
   public Parameter serviceTime;

   private Token _currentInput;
   private double _nextTimeFree = Double.NEGATIVE_INFINITY;

   public Server(TypedCompositeActor container, String name)
          throws NameDuplicationException, IllegalActionException  {
       super(container, name);
       serviceTime = new Parameter(this, "serviceTime", new DoubleToken(1.0));
       serviceTime.setTypeEquals(BaseType.DOUBLE);
       newServiceTime = new DEIOPort(this, "newServiceTime", true, false);
       newServiceTime.setTypeEquals(BaseType.Double);
       output.setTypeAtLeast(input);
       input.delayTo(output);
       newServiceTime.delayTo(output);
    }

   ... attributeChanged(), clone() methods ...

   public void initialize() throws IllegalActionException {
       super.initialize();
       _nextTimeFree = Double.NEGATIVE_INFINITY;
   }

   public boolean prefire() throws IllegalActionException {
      DEDirector director = (DEDirector)getDirector();DEDirector dir = (DEDirector)getDirector();

      if (director.getCurrentTime() >= _nextTimeFree) {
           return true;
       } else {
           // Schedule a firing if there is a pending token so it can be served.
           if (input.hasToken(0)) {
               director.fireAt(this,_nextTimeFree);
           }
           return false;
       }
    }

   public void fire() throws IllegalActionException {
       if (newServiceTime.getWidth() > 0 && newServiceTime.hasToken(0)) {
           DoubleToken time = (DoubleToken)(newServiceTime.get(0));
           serviceTime.setToken(time);
       }
       if (input.getWidth() > 0 && input.hasToken(0)) {
           _currentInput = input.get(0);
           double delay = ((DoubleToken)serviceTime.getToken()).doubleValue();
           _nextTimeFree = ((DEDirector)getDirector()).getCurrentTime() + delay;
       } else {
           _currentInput = null;
       }
   }

   public boolean postfire() throws IllegalActionException {
       if (_currentInput != null) {
           double delay = ((DoubleToken)serviceTime.getToken()).doubleValue();
           output.send(0, _currentInput, delay);
       }
       return super.postfire();
   }
}
```

FIGURE 8.9. Code for the Server actor. For more details, see the source code.

explained above, it leverages on the Java multi-threading infrastructure to save the state information.

Consider the code for the ABRecognizer actor shown in figure 8.10. The two code listings implement two actors with equivalent behavior. The left one implements it as a threaded actor, while the right one implements it as a standard actor. We will from now on refer to the left one as the threaded description and the right one as the standard description. In both description, the actor has two input ports, `inportA` and `inportB`, and one output port, `outport`. The behavior is as follows.

Produce an output event at outport as soon as events at inportA and inportB occurs in that particular order, and repeat this behavior.

Note that the standard description needs a state variable `state`, unlike the case in the threaded description. In general the threaded description encodes the state information in the position of the code, while the standard description encodes it explicitly using state variables. While it is true that the context switching overhead associated with multi-threading application reduces the performance, we argue that the simplicity and clarity of writing actors in the threaded fashion is well worth the cost in some applications.

The infrastructure for this feature is shown in figure 8.4. To write an actor in the threaded fashion, one simply derives from the DEThreadActor class and implements the run() method. In many cases, the content of the run() method is enclosed in the infinite '`while(true)`' loop since many useful threaded actors do not terminate.

The waitForNewInputs() method is overloaded and has two flavors, one that takes no arguments and another that takes an IOPort array as argument. The first suspends the thread until there is at least one input token in at least one of the input ports, while the second suspends until there is at least one input token in any one of the specified input ports, ignoring all other tokens.

In the current implementation, both versions of waitForNewInputs() clear all input ports before the thread suspends. This guarantees that when the thread resumes, all tokens available are new, in the sense that they were not available before the waitForNewInput() method call.

```
public class ABRecognizer extends DEThreadActor {
   StringToken msg = new StringToken("Seen AB");

   // the run method is invoked when the thread
   // is started.
   public void run() {
      while (true) {
         waitForNewInputs();
         if (inportA.hasToken(0)) {
            IOPort[] nextinport = {inportB};
            waitForNewInputs(nextinport);
            outport.broadcast(msg);
         }
      }
   }
```

```
public class ABRecognizer extends DEActor {
   StringToken msg = new StringToken("Seen AB");

   // We need an explicit state variable in
   // this case.
   int state = 0;

   public void fire() {
      switch (state) {
         case 0:
            if (inportA.hasToken(0)) {
               state = 1;
               break;
            }
         case 1:
            if (inportB.hasToken(0)) {
               state = 0;
               outport.broadcast(msg);
            }
      }
   }
}
```

FIGURE 8.10. Code listings for two style of writing the ABRecognizer actor.

The implementation also guarantees that between calls to the waitForNewInputs() method, the rest of the DE model is suspended. This is equivalent to saying that the section of code between calls to the waitForNewInput() method is a critical section. One immediate implication is that the result of the method calls that check the configuration of the model (e.g. hasToken() to check the receiver) will not be invalidated during execution in the critical section. It also means that this should not be viewed as a way to get parallel execution in DE. For that, consider the DDE domain.

It is important to note that the implementation serializes the execution of threads, meaning that at any given time there is only one thread running. When a threaded actor is running (i.e. executing inside its run() method), all other threaded actors and the director are suspended. It will keep running until a waitForNewInputs() statement is reached, where the flow of execution will be transferred back to the director. Note that the director thread executes all non-threaded actors. This serialization is needed because the DE domain has a notion of global time, which makes parallelism much more difficult to achieve.

The serialization is accomplished by the use of monitor in the DEThreadActor class. Basically, the fire() method of the DEThreadActor class suspends the calling thread (i.e. the director thread) until the threaded actor suspends itself (by calling waitForNewInputs()). One key point of this implementation is that the threaded actors appear just like an ordinary DE actor to the DE director. The DEThreadActor base class encapsulates the threaded execution and provides the regular interfaces to the DE director. Therefore the threaded description can be used whenever an ordinary actor can, which is everywhere.

The code shown in figure 8.11 implements the run method of a slightly more elaborate actor with the following behavior:

> *Emit an output O as soon as two inputs A and B have occurred. Reset this behavior each time the input R occurs.*

Future work in this area may involve extending the infrastructure to support various concurrency constructs, such as preemption, parallel execution, etc. It might also be interesting to explore new concurrency semantics similar to the threaded DE, but without the ‘forced’ serialization.

8.6 Composing DE with Other Domains

One of the major concepts in Ptolemy II is modeling heterogeneous systems through the use of hierarchical heterogeneity. Actors on the same level of hierarchy obey the same set of semantics rules. Inside some of these actors may be another domain with a different model of computation. This mechanism is supported through the use of opaque composite actors. An example is shown in figure 8.12. The outermost domain is DE and it contains seven actors, two of them are opaque and composite. The opaque composite actors contain subsystems, which in this case are in the DE and CT domains.

8.6.1 DE inside Another Domain

The DE subsystem completes one iteration whenever the opaque composite actor is fired by the outer domain. One of the complications in mixing domains is in the synchronization of time. Denote the current time of the DE subsystem by t_{inner} and the current time of the outer domain by t_{outer}. An iteration of the DE subsystem is similar to an iteration of a top-level DE model, except that prior to the iteration tokens are transferred from the ports of the opaque composite actors into the ports of the contained DE subsystem, and after the end of the iteration, the director requesting a refire at the smallest time stamp in the event queue of the DE subsystem.

The first of these is done in the transferInputs() method of the DE director. This method is extended from its default implementation in the Director class. The implementation in the DEDirector class advances the current time of the DE subsystem to the current time of the outer domain, then calls super.transferInputs(). It is done in order to correctly associate tokens seen at the input ports of the opaque composite actor, if any, with events at the current time of the outer domain, t_{outer}, and put these events into the global event queue. This mechanism is, in fact, how the DE subsystem synchronize its current time, t_{inner}, with the current time of the outer domain, t_{outer}.(Recall that the DE director advances time by looking at the smallest time stamp in the event queue of the DE subsystem). Specifically, before the advancement of the current time of the DE subsystem t_{inner} is less than or equal to the t_{outer}, and after the advancement t_{inner} is equal to the t_{outer}.

Requesting a refiring is done in the postfire() method of the DE director by calling the fireAt() method of the executive director. Its purpose is to ensure that events in the DE subsystem are processed

```
public void run() {
   try {
      while (true) {
         // In initial state..
         waitForNewInputs();
         if (R.hasToken(0)) {
            // Resetting..
            continue;
         }
         if (A.hasToken(0)) {
            // Seen A..
            IOPort[] ports = {B,R};
            waitForNewInputs(ports);
            if (!R.hasToken(0)) {
               // Seen A then B..
               O.broadcast(new DoubleToken(1.0));
               IOPort[] ports2 = {R};
               waitForNewInputs(ports2);
            } else {
               // Resetting
               continue;
            }
         } else if (B.hasToken(0)) {
            // Seen B..
            IOPort[] ports = {A,R};
            waitForNewInputs(ports);
            if (!R.hasToken(0)) {
            // Seen B then A..
            O.broadcast(new DoubleToken(1.0));
            IOPort[] ports2 = {R};
            waitForNewInputs(ports2);
         } else {
            // Resetting
            continue;
         }
      } // while (true)
   } catch (IllegalActionException e) {
      getManager().notifyListenersOfException(e);
   }
}
```

FIGURE 8.11. The run() method of the ABRO actor.

on time with respect to the current time of the outer domain, t_{outer}.

Note that if the DE subsystem is fired due to the outer domain processing a refire request, then there may not be any tokens in the input port of the opaque composite actor at the beginning of the DE subsystem iteration. In that case, no new events with time stamps equal to t_{outer} will be put into the global event queue. Interestingly, in this case, the time synchronization will still work because t_{inner} will be advanced to the smallest time stamp in the global event queue which, in turn, has to be equal t_{outer} because we always request a refire according to that time stamp.

8.6.2 Another Domain inside DE

Due to its nature, the opaque composite actor is opaque and therefore, as far as the DE Director is concerned, behaves exactly like a domain polymorphic actor. Recall that domain polymorphic actors are treated as functions with zero delay in computation time. To produce events in the future, domain polymorphic actors request a refire from the DE director and then produce the events when it is refired.

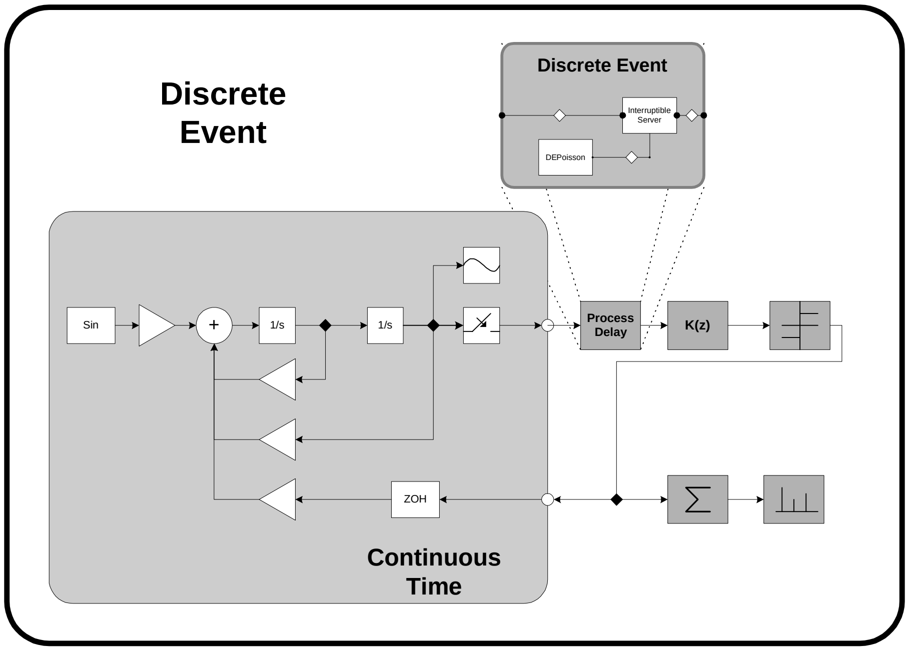


FIGURE 8.12. An example of heterogeneous and hierarchical composition. The CT subsystem and DE subsystem are inside an outermost DE system. This example is developed by Jie Liu [57].

9 SDF Domain

Author: Steve Neuendorffer
Contributor: Brian Vogel

9.1 Purpose of the Domain

The synchronous dataflow (SDF) domain is useful for modeling simple dataflow systems without complicated flow of control, such as signal processing systems. Under the SDF domain, the execution order of actors is statically determined prior to execution. This results in execution with minimal overhead, as well as bounded memory usage and a guarantee that deadlock will never occur. This domains is specialized, and may not always be suitable. Applications that require dynamic scheduling could use the process networks (PN) domain instead, for example.

9.2 Using SDF

There are three issues that must be addressed when using the SDF domain:

- Deadlock
- Consistency of data rates
- The value of the iterations parameter

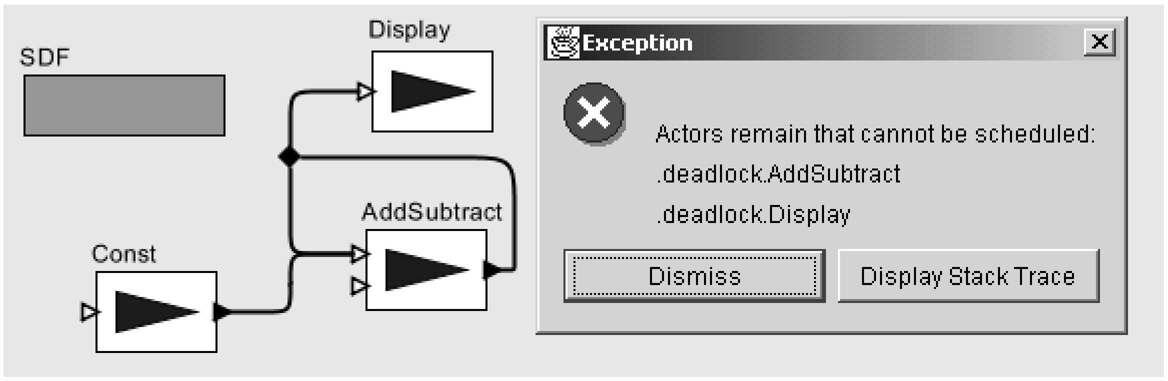
This section will present a short description of these issues. For a more complete description, see section 9.3.

9.2.1 Deadlock

Consider the SDF model shown in figure 9.1. This actor has a feedback loop from the output of the AddSubtract actor back to its own input. Attempting to run the model results in the exception shown at the right in the figure. The director is unable to schedule the model because the input of the AddSubtract actor depends on data from its own output. In general, feedback loops can result in such conditions.

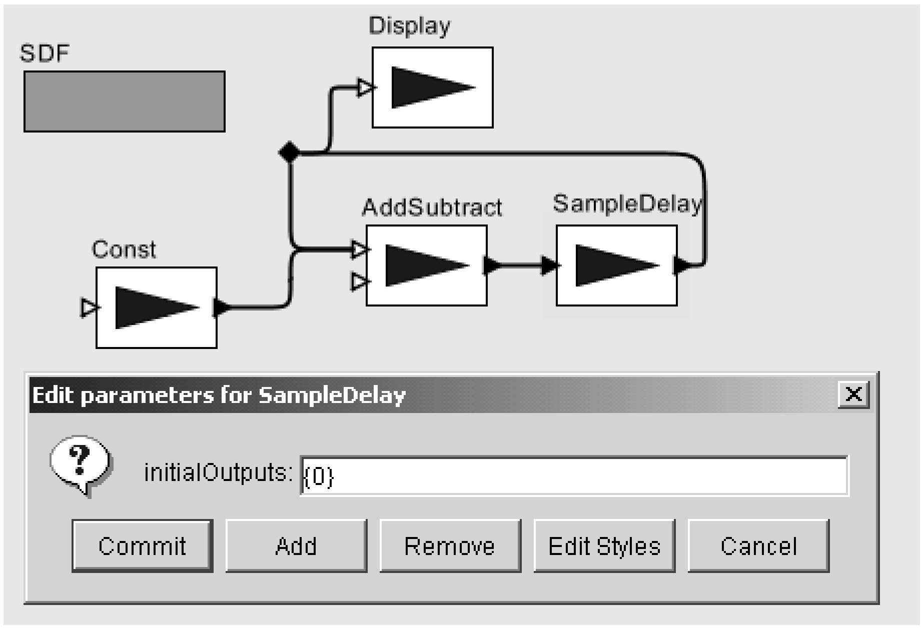
The fix for such deadlock conditions is to use the SampleDelay actor, shown highlighted in figure 9.2. This actor injects into the feedback loop an initial token, the value of which is given by the *initialOutputs* parameter of the actor. In the figure, this parameter has the value {0}. This is an array with a single token, an integer with value 0. A double delay with initial values 0 and 1 can be specified using a two element array, such as {0, 1}.

It is important to note that it is occasionally necessary to add a delay that is not in a feedback loop to match the delay of an in input with the delay around a feedback loop. It can sometimes be tricky to see exactly where such delays should be placed without fully considering the flow of the initial tokens described above.



Actors remain that cannot be scheduled:
.deadlock.AddSubtract
.deadlock.Display

FIGURE 9.1. An SDF model that deadlocks.



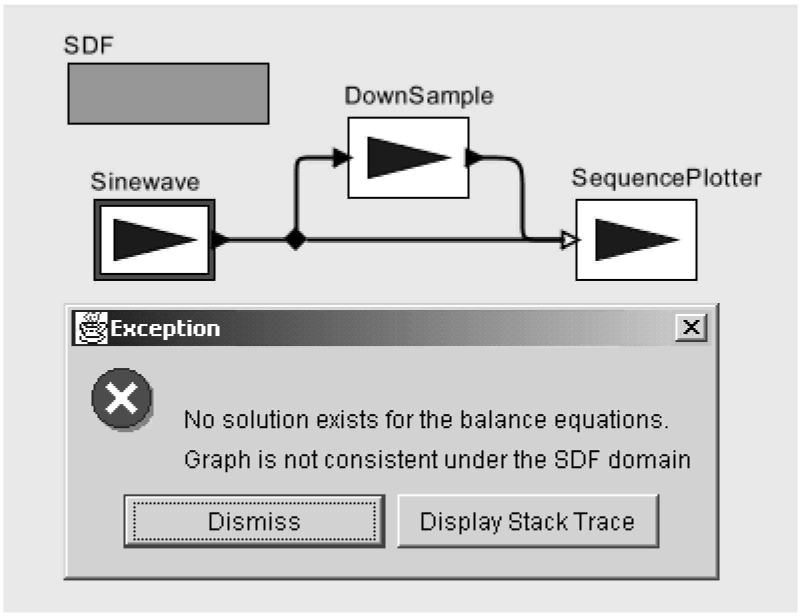
initialOutputs: {0}

FIGURE 9.2. The model of figure 9.1 corrected with an instance of SampleDelay in the feedback loop.

9.2.2 Consistency of data rates

Consider the SDF model shown in figure 9.3. The model is attempting to plot a sinewave and its downsampled counterpart. However, there is an error because the number of tokens on each channel of the input port of the plotter can never be made the same. The DownSample actor declares that it consumes 2 tokens using the *tokenConsumptionRate* parameter of its input port. Its output port similarly declares that it produces only one token, so there will only be half as many tokens being plotted from the DownSample actor as from the Sinewave.

The fixed model is shown in figure 9.4, which uses two separate plotters. When the model is executed, the plotter on the bottom will fire twice as often as the plotter on the top, since must consume twice as many tokens. Notice that the problem appears because one of the actors (in this case, the DownSample actor) produces or consumes more than one token on one of its ports. One easy way to ensure rate consistency is to use actors that only produce and consume one token at a time. This special case is known as *homogeneous* SDF. Note that actors like the Sequence plotter which do not specify rate parameters are assumed to be homogeneous. For more specific information about the rate parame-



No solution exists for the balance equations.
Graph is not consistent under the SDF domain

FIGURE 9.3. An SDF model with inconsistent rates.

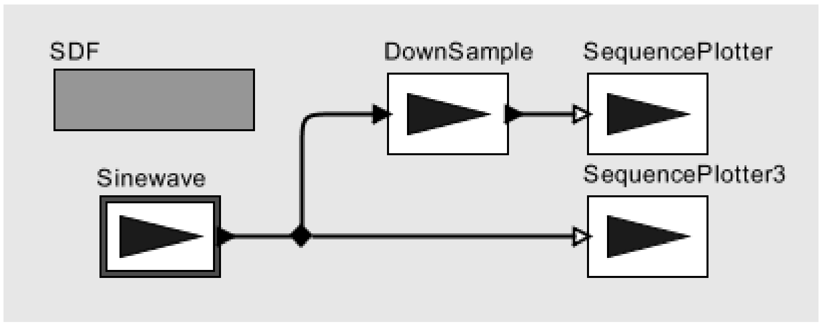


FIGURE 9.4. Figure 9.3 modified to have consistent rates.

ters and how they are used for scheduling, see section 9.3.1.

9.2.3 How many iterations?

One final issue when using the SDF domain concerns the value of the *iterations* parameter of the SDF director. In homogeneous models one token is usually produced for every iteration. However, when token rates other than one are used, more than one interesting output value may be created for each iteration. For example, consider figure 9.5 which contains a model that plots the Fast Fourier Transform of the input signal. The important thing to realize about this model is that the FFT actor declares that it consumes 256 tokens from its input port and produces 256 tokens from its output port, corresponding to an order 8 FFT. This means that only one iteration is necessary to produce all 256 values of the FFT.

Contrast this with the model in figure 9.6. This model plots the individual values of the signal. Here 256 iterations are necessary to see the entire input signal, since only one output value is plotted in each iteration.

9.3 Properties of the SDF domain

SDF is an untimed model of computation. All actors under SDF consume input tokens, perform their computation and produce outputs in one atomic operation. If an SDF model is embedded within a timed model, then the SDF model will behave as a zero-delay actor.

In addition, SDF is a statically scheduled domain. The firing of a composite actor corresponds to a

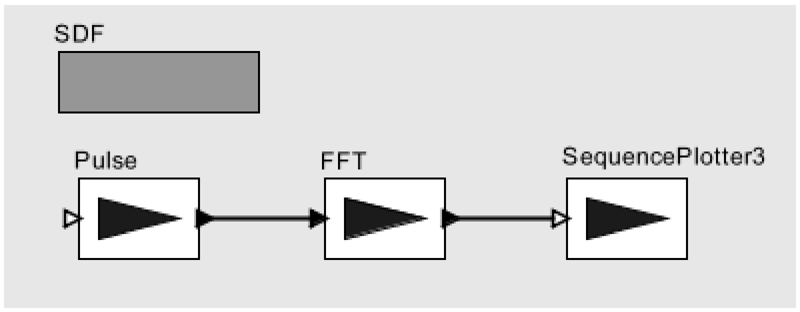


FIGURE 9.5. A model that plots the Fast Fourier Transform of a signal. Only one iteration must be executed to plot all 256 values of the FFT, since the FFT actor produces and consumes 256 tokens each firing.

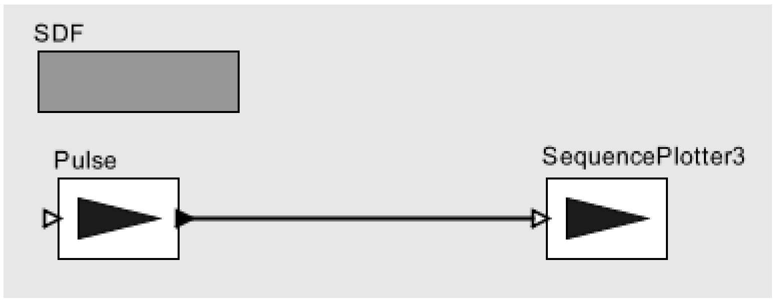


FIGURE 9.6. A model that plots the values of a signal. 256 iterations must be executed to plot the entire signal.

single iteration of the contained(9.3.1) model. An SDF iteration consists of one execution of the pre-calculated SDF schedule. The schedule is calculated so that the number of tokens on each relation is the same at the end of an iteration as at the beginning. Thus, an infinite number of iterations can be executed, without deadlock or infinite accumulation of tokens on each relation.

Execution in SDF is extremely efficient because of the scheduled execution. However, in order to execute so efficiently, some extra information must be given to the scheduler. Most importantly, the data rates on each port must be declared prior to execution. The data rate represents the number of tokens produced or consumed on a port during every firing[1]. In addition, explicit data delays must be added to feedback loops to prevent deadlock. At the beginning of execution, and any time these data rates change, the schedule must be recomputed. If this happens often, then the advantages of scheduled execution can quickly be lost.

9.3.1 Scheduling

The first step in constructing the schedule is to solve the *balance equations* [53]. These equations determine the number of times each actor will fire during an iteration. For example, consider the model in figure 9.7. This model implies the following system of equations, where *ProductionRate* and *ConsumptionRate* are declared properties of each port, and *Firings* is a property of each actor that will be solved for:

$$\mathit{Firings}(\mathrm{A}) \times \mathit{ProductionRate}(\mathrm{A1}) = \mathit{Firings}(\mathrm{B}) \times \mathit{ConsumptionRate}(\mathrm{B1})$$

$$\mathit{Firings}(\mathrm{A}) \times \mathit{ProductionRate}(\mathrm{A2}) = \mathit{Firings}(\mathrm{C}) \times \mathit{ConsumptionRate}(\mathrm{C1})$$

$$\mathit{Firings}(\mathrm{C}) \times \mathit{ProductionRate}(\mathrm{C2}) = \mathit{Firings}(\mathrm{B}) \times \mathit{ConsumptionRate}(\mathrm{B2})$$

These equations express constraints that the number of tokens created on a relation during an iteration is equal to the number of tokens consumed. These equations usually have an infinite number of linearly dependent solutions, and the least positive integer solution for *Firings* is chosen as the *firing vector,* or the repetitions vector.

The second step in constructing an SDF schedule is dataflow analysis. Dataflow analysis orders the firing of actors, based on the relations between them. Since each relation represents the flow of data, the actor producing data must fire before the consuming actor. Converting these data dependencies to a sequential list of properly scheduled actors is equivalent to topologically sorting the SDF graph, if the graph is acyclic[2]. Dataflow graphs with cycles cause somewhat of a problem, since such

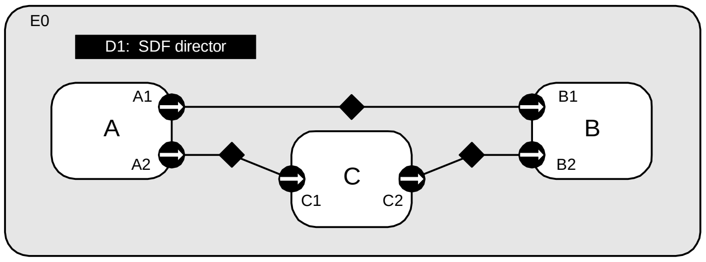


FIGURE 9.7. An example SDF model.

1. This is known as *multirate* SDF, where arbitrary rates are allowed. Not to be confused with *homogeneous* SDF, where the data rates are fixed to be one.

graphs cannot be topologically sorted. In order to determine which actor of the loop to fire first, a *data delay* must be explicitly inserted somewhere in the cycle. This delay is represented by an initial token created by one of the output ports in the cycle during initialization of the model. The presence of the delay allows the scheduler to break the dependency cycle and determine which actor in the cycle to fire first. In Ptolemy II, the initial token (or tokens) can be sent from any port, as long as the port declares an *initProduction* property. However, because this is such a common operation in SDF, the Delay actor (see section 9.5) is provided that can be inserted in a feedback look to break the cycle. Cyclic graphs not properly annotated with delays cannot be executed under SDF. An example of a cyclic graph properly annotated with a delay is shown in figure 9.8.

In some cases, a non-zero solution to the balance equations does not exist. Such models are said to be *inconsistent*, and cannot be executed under SDF. Inconsistent graphs inevitably result in either deadlock or unbounded memory usage for any schedule. As such, inconsistent graphs are usually bugs in the design of a model. However, inconsistent graphs can still be executed using the PN domain, if the behavior is truly necessary. Examples of consistent and inconsistent graphs are shown in figure 9.9.

9.3.2 Hierarchical Scheduling

So far, we have assumed that the SDF graph is not hierarchical. The simplest way to schedule a hierarchical SDF model is flatten the model to remove the hierarchy, and then schedule the model as usual. This technique allows the most efficient schedule to be constructed for a model, and avoids certain composability problems when creating hierarchical models. In Ptolemy II, a model created using a transparent composite actor to define the hierarchy is scheduled in exactly this way.

Ptolemy II also supports a stronger version of hierarchy, in the form of opaque composite actors. In this case, the hierarchical actor appears to be no different from the outside than an atomic actor with no hierarchy. The SDF domain does not have any information about the contained model, other than the

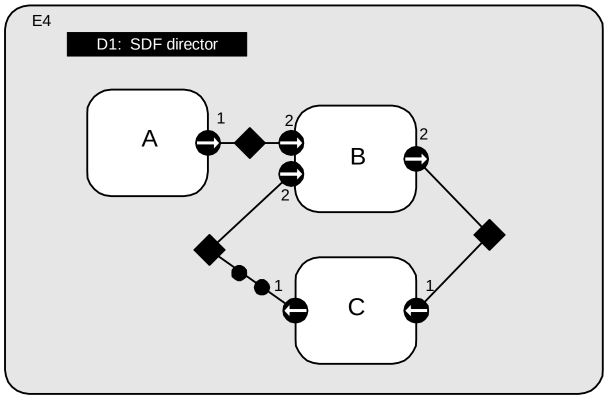


FIGURE 9.8. A consistent cyclic graph, properly annotated with delays. A one token delay is represented by a black circle. E3 is responsible for setting the *tokenInitProduction* parameter on its output port, and creating the two tokens during initialization. This graph can be executed using the schedule E1, E1, E2, E3, E3.

2. Note that the topological sort does not correspond to a unique total ordering over the actors. Furthermore, especially in multirate models it may be possible to interleave the firings of actors that fire more than once. This can result in many possible schedules that represent different performance trade-offs. We anticipate that future schedulers will be implemented to take advantage of these trade-offs. For more information about these trade-offs, see [47].

rate parameters that may be specified on the ports of the composite actor. The SDF domain is designed so that it automatically sets the rates of external ports when the schedule is computed. Most other domains are designed (conveniently enough) so that their models are compatible with default rate properties assumed by the SDF domain. For a complete description of these defaults, see the description of the SDFScheduler class in section 9.4.2.

9.4 Software Architecture

The SDF kernel package implements the SDF model of computation. The structure of the classes in this package is shown in figure 9.10.

9.4.1 SDF Director

The SDFDirector class extends the StaticSchedulingDirector class. When an SDF director is created, it is automatically associated with an instance of the default scheduler class, SDFScheduler. This scheduler is intended to be relatively fast, but not designed to optimize for any particular performance goal. The SDF director does not currently restrict the schedulers that may be used with it. For more information about SDF schedulers, see section 9.4.2.

The director has a parameter, *iterations*, which determines a limit on the number of times the director wishes to be fired[1]. After the director has been fired the given number of times, it will always return false in its postfire() method, indicating that it does not wish to be fired again. The *iterations* parameter must contain a non-negative integer value. The default value is an IntToken with value 0, indicating that there is no preset limit for the number of times the director will fire. Users will likely specify a non-zero value in the director of the toplevel composite actor as the number of toplevel itera-

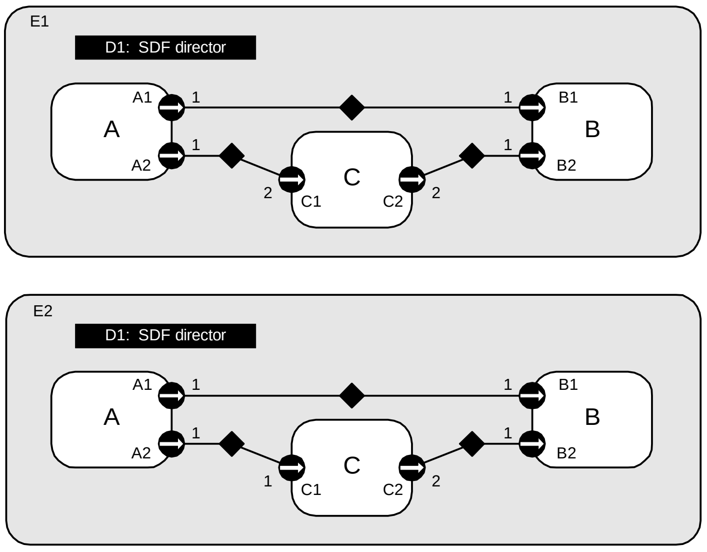


FIGURE 9.9. Two models, with each port annotated with the appropriate rate properties. The model on the top is consistent, and can be executed using the schedule A, A, C, B, B. The model on the bottom is inconsistent because tokens will accumulate between ports C2 and B2.

1. This parameter acts similarly to the Time-to-Stop parameter in Ptolemy Classic.

tions of the model.

The SDF director also has a *vectorizationFactor* parameter that can be used to request vectorized execution of a model. This parameter suggests that the director modify the schedule so that instead of firing each actor only once, it is fired *vectorizationFactor* times using the vectorized iterate method. The specified factor serves only as a suggestion, and the director is free to ignore it or to use a different factor. The *vectorizationFactor* parameter must contain a positive integer value. The default value is an IntToken with value one, indicating that no vectorization should be done. Note that vectorizing the execution of a model is not necessarily possible if the model contains feedback cycles. At the very least, it is likely that the data delay specified for any cycle must be increased (possibly changing the meaning of the model).

The newReceiver() method in SDF directors is overloaded to return instances of the SDFReceiver

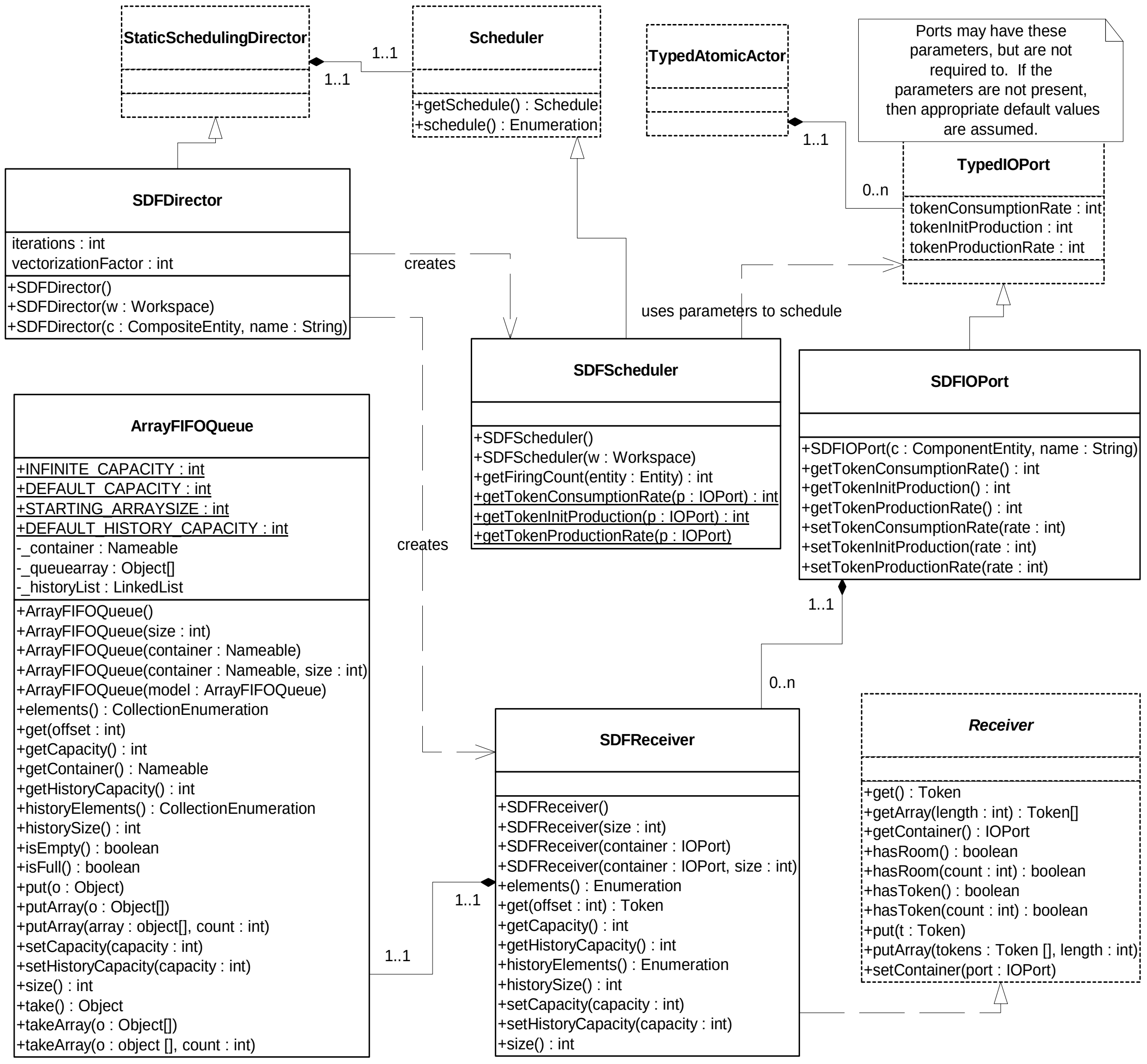

StaticSchedulingDirector
Scheduler
+getSchedule() : Schedule
+schedule() : Enumeration
1..1
1..1
TypedAtomicActor
Ports may have these parameters, but are not required to. If the parameters are not present, then appropriate default values are assumed.
1..1
0..n
TypedIOPort
tokenConsumptionRate : int
tokenInitProduction : int
tokenProductionRate : int
SDFDirector
iterations : int
vectorizationFactor : int
+SDFDirector()
+SDFDirector(w : Workspace)
+SDFDirector(c : CompositeEntity, name : String)
creates
uses parameters to schedule
SDFScheduler
+SDFScheduler()
+SDFScheduler(w : Workspace)
+getFiringCount(entity : Entity) : int
+getTokenConsumptionRate(p : IOPort) : int
+getTokenInitProduction(p : IOPort) : int
+getTokenProductionRate(p : IOPort)
SDFIOPort
+SDFIOPort(c : ComponentEntity, name : String)
+getTokenConsumptionRate() : int
+getTokenInitProduction() : int
+getTokenProductionRate() : int
+setTokenConsumptionRate(rate : int)
+setTokenInitProduction(rate : int)
+setTokenProductionRate(rate : int)
ArrayFIFOQueue
+INFINITE_CAPACITY : int
+DEFAULT_CAPACITY : int
+STARTING_ARRAYSIZE : int
+DEFAULT_HISTORY_CAPACITY : int
-_container : Nameable
-_queuearray : Object[]
-_historyList : LinkedList
+ArrayFIFOQueue()
+ArrayFIFOQueue(size : int)
+ArrayFIFOQueue(container : Nameable)
+ArrayFIFOQueue(container : Nameable, size : int)
+ArrayFIFOQueue(model : ArrayFIFOQueue)
+elements() : CollectionEnumeration
+get(offset : int)
+getCapacity() : int
+getContainer() : Nameable
+getHistoryCapacity() : int
+historyElements() : CollectionEnumeration
+historySize() : int
+isEmpty() : boolean
+isFull() : boolean
+put(o : Object)
+putArray(o : Object[])
+putArray(array : object[], count : int)
+setCapacity(capacity : int)
+setHistoryCapacity(capacity : int)
+size() : int
+take() : Object
+takeArray(o : Object[])
+takeArray(o : object [], count : int)
creates
1..1
0..n
SDFReceiver
+SDFReceiver()
+SDFReceiver(size : int)
+SDFReceiver(container : IOPort)
+SDFReceiver(container : IOPort, size : int)
+elements() : Enumeration
+get(offset : int) : Token
+getCapacity() : int
+getHistoryCapacity() : int
+historyElements() : Enumeration
+historySize() : int
+setCapacity(capacity : int)
+setHistoryCapacity(capacity : int)
+size() : int
1..1
1..1
Receiver
+get() : Token
+getArray(length : int) : Token[]
+getContainer() : IOPort
+hasRoom() : boolean
+hasRoom(count : int) : boolean
+hasToken() : boolean
+hasToken(count : int) : boolean
+put(t : Token)
+putArray(tokens : Token [], length : int)
+setContainer(port : IOPort)

FIGURE 9.10. The static structure of the SDF kernel classes.

class. This receiver contains optimized method for reading and writing blocks of tokens. For more information about SDF receivers, see section 9.4.3.

9.4.2 SDF Scheduler

The basic SDFScheduler derives directly from the Scheduler class. This scheduler provides unlooped, sequential schedules suitable for use on a single processor. No attempt is made to optimize the schedule by minimizing data buffer sizes, minimizing the size of the schedule, or detecting parallelism to allow execution on multiple processors. We anticipate that more elaborate schedulers capable of these optimizations will be added in the future.

The scheduling algorithm is based on the simple multirate algorithm in [53]. Currently, only single processor schedules are supported. The multirate scheduling algorithm relies on the actors in the system to declare the data rates of each port. The data rates of ports are specified using three parameters on each port named *tokenConsumptionRate*, *tokenProductionRate*, and *tokenInitProduction*. The production parameters are valid only for output ports, while the consumption parameter is valid only for input ports. If a parameter exists that is not valid for a given port, then the value of the parameter must be zero, or the scheduler will throw an exception. If a valid parameter is not specified when the scheduler runs, then default values of the parameters will be assumed, however the parameters are not then created[1].

After scheduling, the SDF scheduler will set the rate parameters on any external ports of the composite actor. This allows a containing actor, which may represent an SDF model, to properly schedule the contained model, as long as the contained model is scheduled first. To ensure this, the SDF director forces the creation of the schedule after initializing all the actors in the model. This mechanism is illustrated in the sequence diagram in figure 9.11.

Disconnected graphs. SDF graphs should generally be connected. If an SDF graph is not connected, then there is some concurrency between the disconnected parts that is not captured by the SDF rate parameters. In such cases, another model of computation (such as process networks) should be used to explicitly specify the concurrency. As such, the current SDF scheduler disallows disconnected graphs, and will throw an exception if you attempt to schedule such a graph. However, sometimes it is useful to avoid introducing another model of computation, so it is possible that a future scheduler will allow disconnected graphs with a default notion of concurrency.

Multiports. Notice that it is impossible to set a rate parameter on individual channels of a port. This is intentional, and all the channels of an actor are assumed to have the same rate. For example, when the AddSubtract actor fires under SDF, it will consume exactly one token from each channel of its input *plus* port, consume one token from each channel of its *minus* port, and produce one token the single channel of its *output* port. Notice that although the domain-polymorphic adder is written to be more general than this (it will consume *up to* one token on each channel of the input port), the SDF scheduler will ensure that there is always at least one token on each input port before the actor fires.

Dangling ports. All channels of a port are required to be connected to a remote port under the SDF domain. A regular port that is not connected will always result in an exception being thrown by the scheduler. However, the SDF scheduler detects multiports that are not connected to anything (and thus

1. The assumed values correspond to a homogeneous actor with no data delay. Input ports are assumed to have a consumption rate of one, output ports are assumed to have a production rate of one, and no tokens are produced during initialization.

have zero width). Such ports are interpreted to have no channels, and will be ignored by the SDF scheduler.

9.4.3 SDF ports and receivers

Unlike most domains, multirate SDF systems tend to produce and consume large blocks of tokens during each firing. Since there can be significant overhead in data transport for these large blocks, SDF receivers are optimized for sending and receiving a block of tokens *en masse*.

The SDFReceiver class implements the Receiver interface. Instead of using the FIFOQueue class to store data, which is based on a linked list structure, SDF receivers use the ArrayFIFOQueue class, which is based on a circular buffer. This choice is much more appropriate for SDF, since the size of the buffer is bounded, and can be determined statically[1].

The SDFIOPort class extends the TypedIOPort class. It exists mainly for convenience when creat-

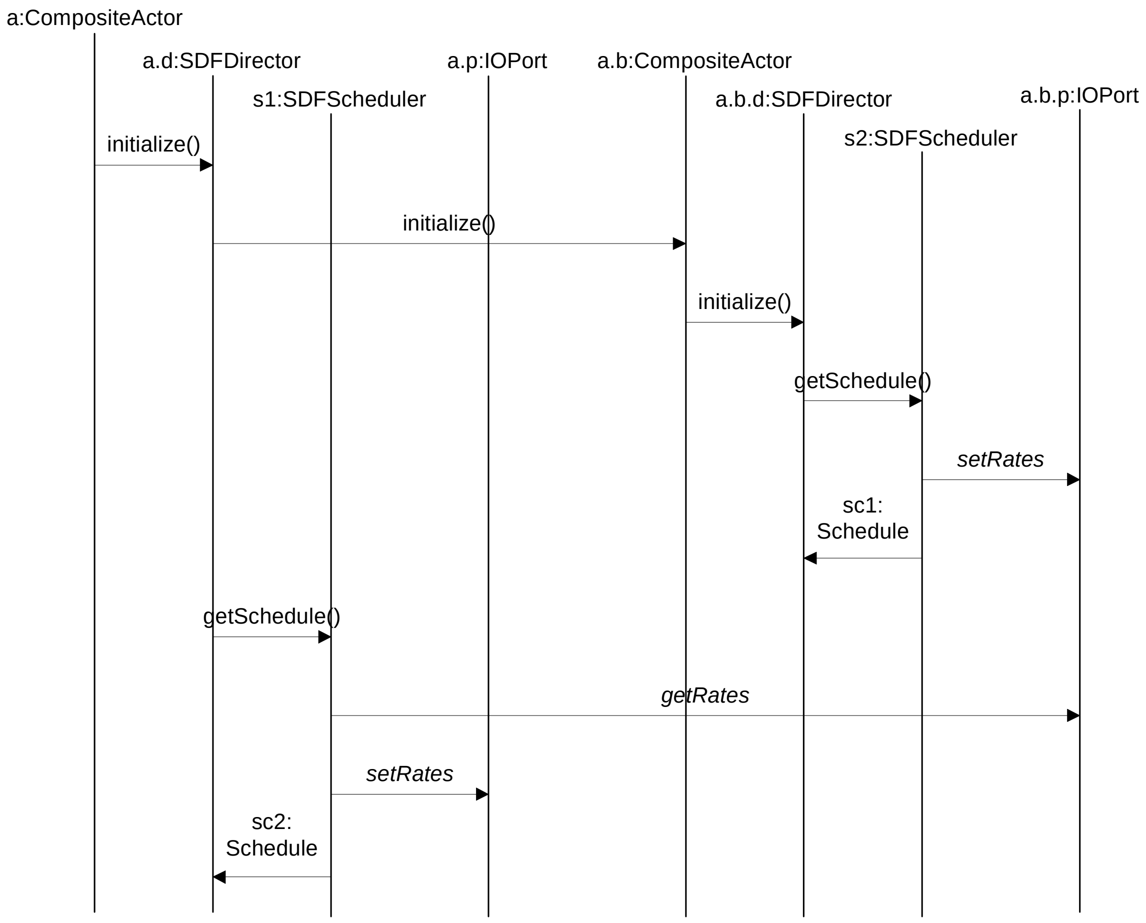


FIGURE 9.11. The sequence of method calls during scheduling of a hierarchical model.

1. Although the buffer sizes can be statically determined, the current mechanism for creating receivers does not easily support it. The SDF domain currently relies on the buffer expanding algorithm that the ArrayFIFOQueue uses to implement circular buffers of unbounded size. Although there is some overhead during the first iteration, the overhead is minimal during subsequent iterations (since the buffer is guaranteed never to grow larger).

ing actors in the SDF domain. It provides convenience methods for setting and accessing the rate parameters used by the SDF scheduler.

9.4.4 ArrayFIFOQueue

The ArrayFIFOQueue class implements a first in, first out (FIFO) queue by means of a circular array buffer[1]. Functionally it is very similar to the FIFOQueue class, although with different enqueue and dequeue performance. It provides a token history and an adjustable, possibly unspecified, bound on the number token it contains.

If the bound on the size is specified, then the array is exactly the size of the bound. In other words, the queue is full when the array becomes full. However, if the bound is unspecified, then the circular buffer is given a small starting size and allowed to grow. Whenever the circular buffer fills up, it is copied into a new buffer that is twice the original size.

9.5 Actors

Most domain-polymorphic actors can be used under the SDF domain. However, actors that depend on a notion of time may not work as expected. For example, in the case of a TimedPlotter actor, all data will be plotted at time zero when used in SDF. In general, domain-polymorphic actors (such as AddSubtract) are written to consume at most one token from each input port and produce exactly one token on each output port during each firing. Under SDF, such an actor will be assumed to have a rate of one on each port, and the actor will consume exactly one token from each input port during each firing. There is one actor that is normally only used in SDF: the Delay actor. The delay actor is provided to make it simple to build models with feedback, by automatically handling the *tokenInitProduction* parameter and providing a way to specify the tokens that are created.

Delay

Ports: *input* (Token), *output* (Token).
Parameters: *initialOutputs* (ArrayToken).

During initialization, create a token on the output for each token in the *initialOutputs* array. During each firing, consume one token on the input and produce the same token on the output.

1. Adding an array of objects to an ArrayFIFOQueue is implemented using the java.lang.system.arraycopy method. This method is capable of safely removing certain checks required by the Java language. On most Java implementations, this is significantly faster than a hand coded loop for large arrays. However, depending on the Java implementation it could actually be slower for small arrays. The cost is usually negligible, but can be avoided when the size of the array is small and known when the actor is written.

10

FSM Domain

Author: *Xiaojun Liu*

10.1 Introduction

Finite state machines (FSMs) have been used extensively in designing sequential control logic. There are two major reasons behind their use. First, FSMs are a very intuitive way to capture control logic and make it easier to communicate a design. Second, FSMs have been the subject of a long history of research work. Many formal analysis and verification methods have been developed for them.

In their simple flat form, FSM models have a key weakness: the number of states in an FSM model can get quite large even for a moderately complex system. Such models quickly become chaotic and incomprehensible when one tries to model a system having many concurrent activities. The problem can be solved by introducing hierarchical organization into FSM models and using them in combination with concurrency models. David Harel first used this approach when he introduced the *Statecharts* formalism [34].

The Statecharts formalism extends the conventional FSM model in three aspects: hierarchical decomposition of states, concurrent composition of FSMs in a synchronous-reactive fashion, and a broadcast communication mechanism between concurrent components. While how these extensions fit together was not completely specified in [34], Harel's work stimulated a lot of interest in the approach. Consequently, there is a proliferation of variants of the Statecharts formalism [7], each proposing a different way to make the extensions fit into a monolithic model. Unfortunately, in all these variants FSM is combined with a particular concurrency model. The applicability of the resulting models is often limited.

Based on the Ptolemy philosophy of hierarchical composition of heterogeneous models of computation, the **charts*[1] formalism [31] allows embedding hierarchical FSMs within a variety of concurrency models. If tight synchronization is possible and desirable, then FSMs can be composed by the

1. Pronounced "starcharts." The star represents a wildcard that can be interpreted as matching multiple concurrency models.

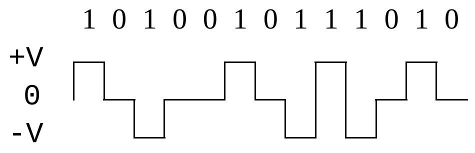
synchronous-reactive model. If the system has a global notion of time and components communicate by time-stamped events, then FSMs can be composed by the discrete-event model. The rest of this chapter focuses on how the FSM domain in Ptolemy II supports the *charts formalism.

10.2 Building FSMs in Vergil

An FSM model is contained by an instance of FSMActor. The FSM model reacts to inputs to the FSM actor by making state transitions. Actions such as sending tokens to the output ports of the FSM actor can be associated with state transitions. In this section, we show how to construct and run a model with an FSM actor in Vergil.

10.2.1 Alternate Mark Inversion Coder

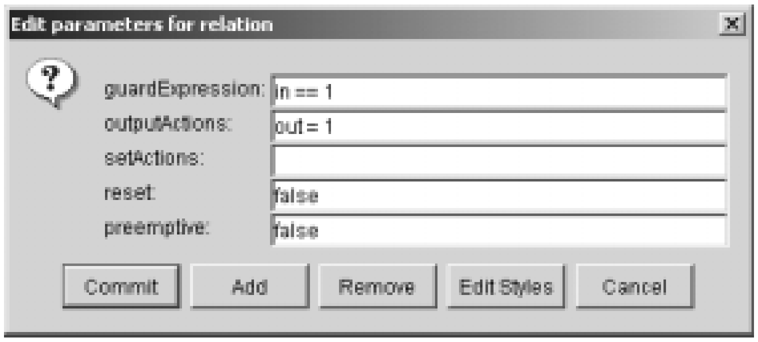
Alternate Mark Inversion (AMI) is a simple digital transmission technique that encodes a bit stream on a signal line as shown below:



The 0 bits are transmitted with voltage zero. The 1 bits are transmitted alternately with positive and negative voltages. On average, the resulting waveform will have no DC component.

We can model an AMI coder with a two-state FSM shown in figure 10.2. To construct a Ptolemy II model containing this coder, follow these steps:

1. Start Vergil, open a graph editor by selecting File -> New -> Graph Editor.
2. From utilities in the palette on the left, drag an FSM actor to the graph. Rename the FSM actor AMICoder.
3. Right click on AMICoder, select Configure Ports. Add an input port with name *in* and an output port with name *out* to AMICoder.
4. Right click on AMICoder, select Look Inside. This will open an FSM editor for AMICoder. Note that the ports of AMICoder are placed at the upper left corner of the graph panel.
5. From the palette on the left, drag a state to the graph, rename it Positive. Drag another state to the graph, rename it Negative.
6. Control-drag from the Positive state to the Negative state to create a transition.
7. Double click on the transition. This will bring up the dialog box shown in figure 10.1 for editing the parameters of the transition.
8. Set guardExpression to `in == 1`, and outputActions to `out = 1`.
9. Create a transition from the Positive state back to itself with guard expression `in == 0` and output action `out = 0`.
10. Create a transition from the Negative state back to itself with guard expression `in == 0` and output action `out = 0`.


Edit parameters for relation

guardExpression: in == 1
outputActions: out = 1
setActions:
reset: false
preemptive: false

FIGURE 10.1. The dialog box for editing parameters of a transition.

11. Create a transition from the Negative state to the Positive state with guard expression `in == 1` and output action `out = -1`.
12. Right click on the background of the graph panel. Select Edit Parameters from the context menu. This will bring up the dialog box for editing parameters of AMICoder. Set initialStateName to `Positive`.
13. The construction of AMICoder is complete. It will look like what is shown in figure 10.2.
14. Return to the graph editor opened in step 1.
15. Drag a Pulse actor (from actor library, sources), a SequencePlotter (from actor library, sinks), and an SDF director (from director library) to the graph.
16. Connect the actors as shown in figure 10.3.
17. Edit parameters of the Pulse actor: set indexes to `{0, 1, 2, 3, 4, 5}`; set values to `{0, 1, 1, 1, 0, 1}`.

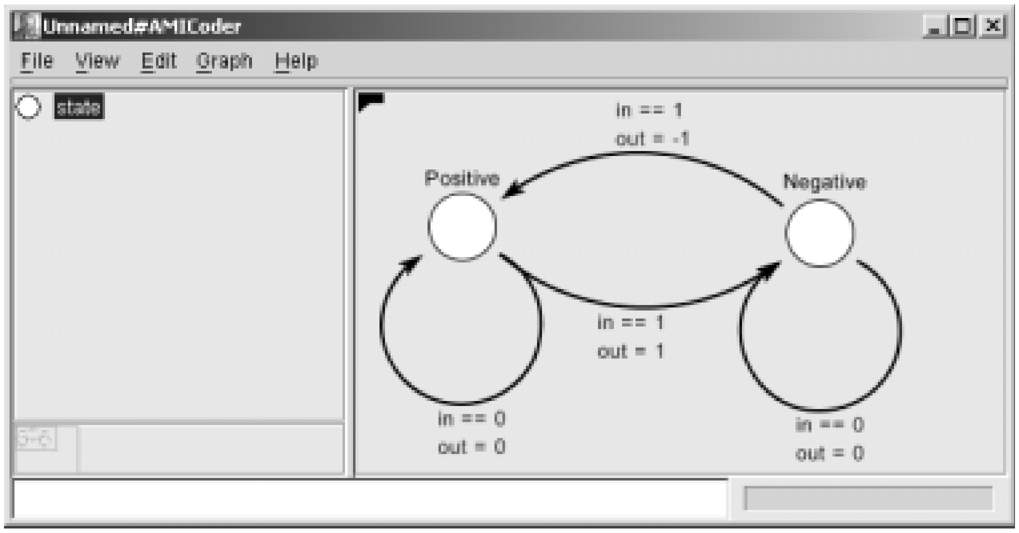


FIGURE 10.2. Vergil FSM editor showing the AMICoder.

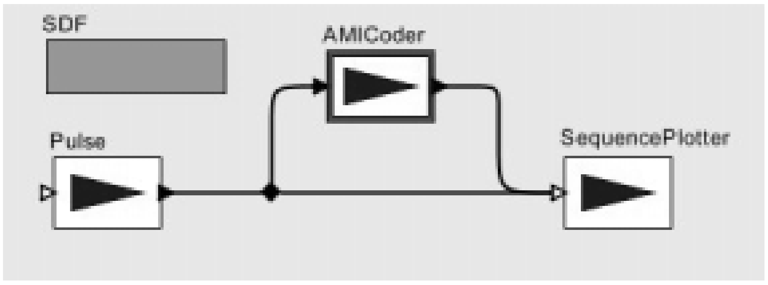


FIGURE 10.3. An SDF model with the AMICoder.

18. The model construction is complete.
19. Select View -> Run Window from the menu. Set director iterations to `6` and execute the model. For a better display of the result, open the set plot format dialog box, unselect connect and use various marks.

10.3 The Implementation of FSMActor

The FSMActor-related classes in the FSM kernel package are shown in figure 10.4.

The FSMActor class extends the CompositeEntity class and implements the TypedActor interface. An FSM actor contains states and transitions. The State class is a subclass of ComponentEntity. A State has two ports: incomingPort, which links to incoming transitions to the state, and outgoingPort, which links to transitions going out from the state. The Transition class is a subclass of ComponentRelation. A transition links to exactly two ports: the outgoing port of its source state, and the incoming port of its destination state.

10.3.1 Guard Expressions

The guard of a transition is specified by its *guardExpression* string attribute. Guard expressions are parsed and evaluated using the Ptolemy II expression language (see the Data chapter for details). Guard expressions should evaluate to a boolean value. A transition is enabled if its guard expression evaluates to true. Parameters of the FSM actor and input variables (defined below) can be used in guard expressions.

Input variables represent the status and input value for each input port of the FSM actor. If the input port is a single port, two variables are used: a status variable named *portName*`_isPresent`, and a value variable named *portName*. If the input port is a multiport of width *n*, *2n* variables are used, two for each channel: a status variable named *portName_channelIndex*`_isPresent`, and a value variable named *portName_channelIndex*. The status variables will have boolean value true if there is a token at the corresponding input, or false otherwise. The value variables have the same type as the corresponding input, and contain the token received from the input, or null if there is no token. All input variables are contained by the FSM actor.

In the following examples (and the examples in the next section), we assume that the FSM actor has two input ports: a single port *in1* and a multiport *in2* of width 2; an output port *out* that is a multiport of width 2; and a parameter *param*.

- Guard expression: `in2_0 + in2_1 > 10`. If the inputs from the two channels of port *in2* have a total greater than 10, the transition is enabled. Note that if one or both channels of port *in2* do not have a token when this expression is evaluated, an exception will be thrown.

- Guard expression: `in1_isPresent && in1 > param`. If there is input from port *in1* and the value of the input is greater than *param*, the transition is enabled.

10.3.2 Actions

A transition can have a set of actions that produce output tokens or set parameters of the FSM actor. To make FSM actors domain polymorphic (see section 3.5), especially for them to be operational

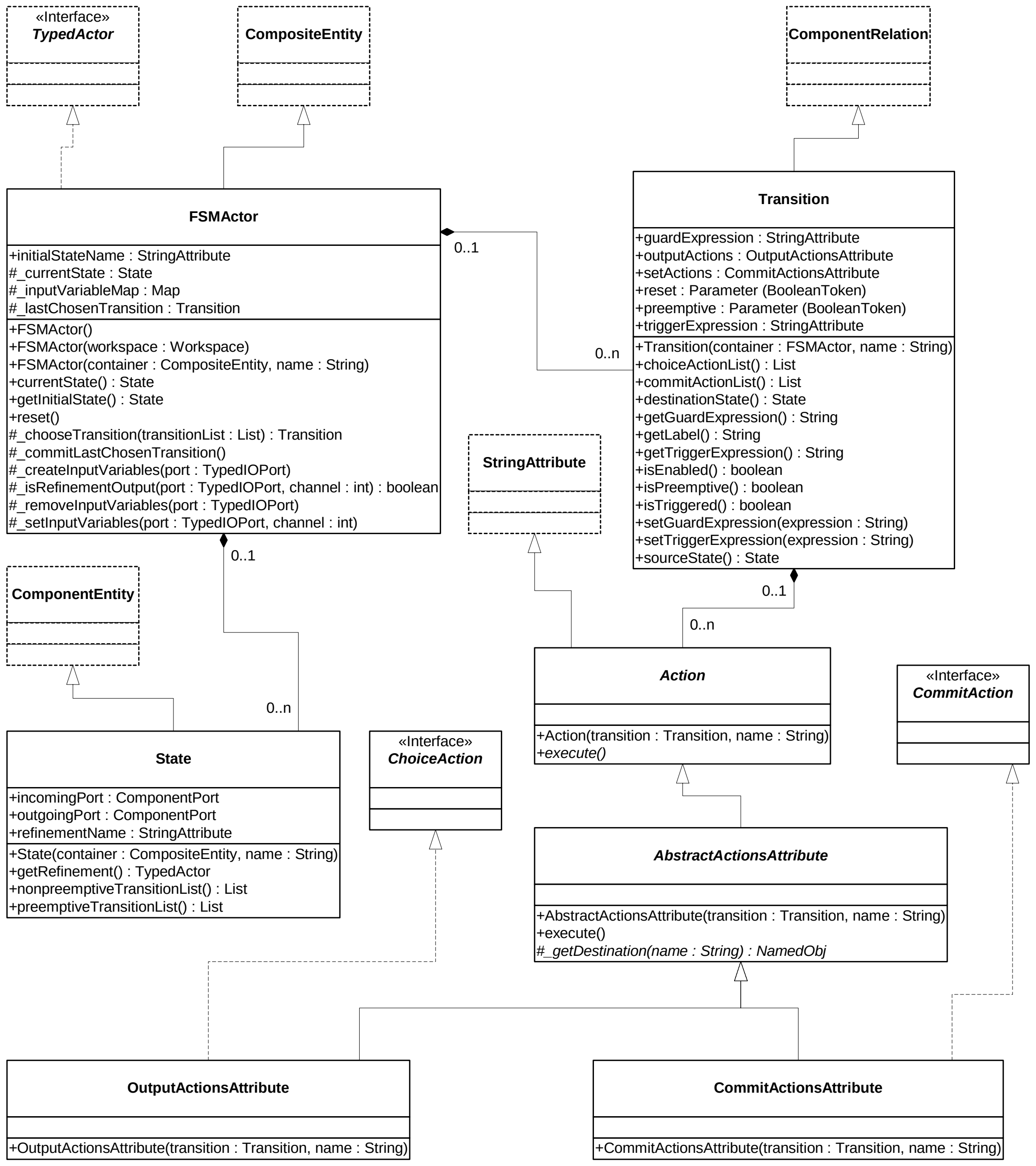

«Interface»
TypedActor
CompositeEntity
ComponentRelation
FSMActor
+initialStateName : StringAttribute
#_currentState : State
#_inputVariableMap : Map
#_lastChosenTransition : Transition
+FSMActor()
+FSMActor(workspace : Workspace)
+FSMActor(container : CompositeEntity, name : String)
+currentState() : State
+getInitialState() : State
+reset()
#_chooseTransition(transitionList : List) : Transition
#_commitLastChosenTransition()
#_createInputVariables(port : TypedIOPort)
#_isRefinementOutput(port : TypedIOPort, channel : int) : boolean
#_removeInputVariables(port : TypedIOPort)
#_setInputVariables(port : TypedIOPort, channel : int)
0..1
0..n
Transition
+guardExpression : StringAttribute
+outputActions : OutputActionsAttribute
+setActions : CommitActionsAttribute
+reset : Parameter (BooleanToken)
+preemptive : Parameter (BooleanToken)
+triggerExpression : StringAttribute
+Transition(container : FSMActor, name : String)
+choiceActionList() : List
+commitActionList() : List
+destinationState() : State
+getGuardExpression() : String
+getLabel() : String
+getTriggerExpression() : String
+isEnabled() : boolean
+isPreemptive() : boolean
+isTriggered() : boolean
+setGuardExpression(expression : String)
+setTriggerExpression(expression : String)
+sourceState() : State
StringAttribute
0..1
ComponentEntity
0..1
0..n
Action
+Action(transition : Transition, name : String)
+execute()
«Interface»
CommitAction
0..n
State
+incomingPort : ComponentPort
+outgoingPort : ComponentPort
+refinementName : StringAttribute
+State(container : CompositeEntity, name : String)
+getRefinement() : TypedActor
+nonpreemptiveTransitionList() : List
+preemptiveTransitionList() : List
«Interface»
ChoiceAction
AbstractActionsAttribute
+AbstractActionsAttribute(transition : Transition, name : String)
+execute()
#_getDestination(name : String) : NamedObj
OutputActionsAttribute
+OutputActionsAttribute(transition : Transition, name : String)
CommitActionsAttribute
+CommitActionsAttribute(transition : Transition, name : String)

FIGURE 10.4. The UML static structure diagram of FSMActor-related classes.

in domains having fixed-point semantics, two kinds of actions are defined: choice actions and commit actions. Choice actions do not modify the extended state[1] of the FSM actor. They are executed when the FSM actor is fired and the containing transition is enabled. Commit actions may modify the extended state of the FSM actor. They are executed in postfire() if the containing transition was enabled in the last firing of the FSM actor. Two marker interfaces are defined in the FSM kernel package: ChoiceAction, which is implemented by all choice action classes, and CommitAction, implemented by all commit action classes.

A transition has an *outputActions* attribute which is an instance of OutputActionsAttribute. The OutputActionsAttribute class allows the user to specify a list of semicolon separated output actions of the form `destination = expression`. The expression can use parameters of the FSM actor and input variables. The destination is either a port name, in which case the result token from evaluating the expression is broadcast to all channels of the port, or of the form `portName(channelIndex)`, in which case the result token is sent to the specified channel. Output actions are choice actions.

- outputActions: `out = in1_isPresent ? in1 : 0`. Broadcast the input from port *in1*, or 0 if there is no input from *in1*, to the two channels of *out*.
- outputActions: `out(0) = param; out(1) = param + 1`. Send the value of *param* to the first channel of *out*, and the value of *param* plus 1 to the second channel.

A transition has a *setActions* attribute which is an instance of CommitActionsAttribute. The CommitActionsAttribute class allows the user to specify a list of semicolon separated commit actions of the form `destination = expression`. The expression can use parameters of the FSM actor and input variables. The destination is a parameter name.

- setActions: `param = param + (in1_isPresent ? in1 : 0)`. The input values from port *in1* are accumulated in *param*.

It is worth noting that parameter values are persistent. If not properly initialized, the parameter *t* in the above example will retain its accumulated value from previous model executions. A useful approach is to build the FSM model such that the initial state has an outgoing transition with guard expression `true`, and use the set actions of this transition for parameter initialization.

10.3.3 Execution

The methods that define the execution of an FSM actor are implemented as follows:

- `preinitialize()`: create receivers and input variables for each input port; set current state to the initial state as specified by the *initialStateName* attribute.
- `initialize()`: perform domain-specific initialization by calling the initialize(Actor) method of the director. Note that in the example given in section 10.2.1, the director will be the SDF director.
- `prefire()`: always return true. An FSM actor is always ready to fire.
- `fire()`: set the values of input variables; choose the enabled transition among the outgoing transitions of the current state; execute the choice actions of the chosen transition.
- `postfire()`: execute the commit actions of the last chosen transition; change state to the destination state of that transition.

Non-deterministic FSMs are not allowed[2]. The fire() method checks whether there is more than one

1. The extended state of an FSM actor is the current state of the state machine it contains plus the set of current values of its parameters.
2. This may change in future developments.

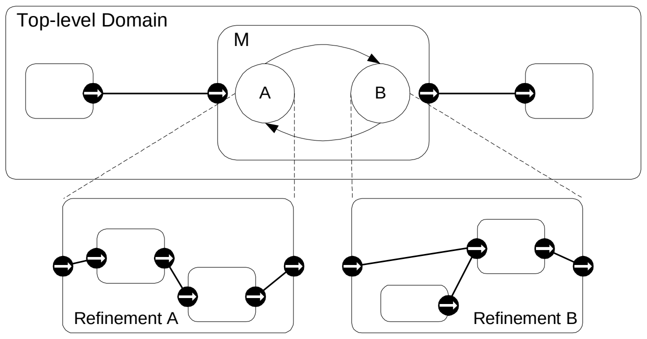


FIGURE 10.5. A modal model example.

enabled transition from the current state. An exception is thrown if there is. In the case when there is no enabled transition, the FSM will stay in its current state.

10.4 Modal Models

The FSM domain supports the *charts formalism with modal models. The concept of modal model is illustrated in figure 10.5. *M* is a modal model with two operation modes. The modes are represented by states of an FSM that controls mode switching. Each mode has a refinement that specifies the behavior of the mode. In Ptolemy II, a modal model[1] is constructed in a typed composite actor having the FSM director as local director. The composite actor contains a mode controller (an FSM actor) and a set of actors that model the refinements. The FSM director mediates the interaction with the outside domain, and coordinates the execution of the refinements with the mode controller.

10.4.1 A Schmidt Trigger Example

In this section, we will illustrate how to build a modal model in Ptolemy II with a simple Schmidt trigger example. The output from the Schmidt trigger will move from -1.0 to 1.0 when its input becomes greater than 0.3, and will move back to -1.0 once its input becomes less than -0.3.

1. Open a Vergil graph editor. From utilities, drag a typed composite actor to the graph, rename it SchmidtTrigger. Add an input port named *in* and an output port named *out* to it.
2. Look inside SchmidtTrigger. This will open a graph editor for it. In this graph editor, drag an FSM actor to the graph, rename it Controller. Drag a typed composite actor to the graph, rename it RefinementP. Drag another typed composite actor to the graph, rename it RefinementN.
3. Add an input port named *in* to Controller. Add an output port named *out* for both RefinementP and RefinementN.
4. Look inside Controller. This will open an FSM editor for it. In this FSM editor, construct a two-state FSM as shown in figure 10.6. Set the reset parameter of both transitions to `true`. Set refinement name of state P to `RefinementP`. Set refinement name of state N to `RefinementN`. Set initial state name of Controller to `N`.

1. The current software architecture that supports modal models is experimental. A new approach based on higher order functions is in progress.

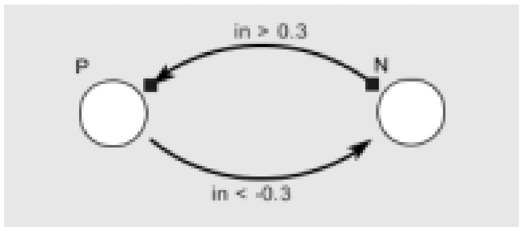


FIGURE 10.6. The mode controller for SchmidtTrigger.

5. Back to the graph editor for SchmidtTrigger. Look inside RefinementP. Build a model for it as shown in figure 10.7. Set the value of Const to `1.0`. Edit parameters of Pulse: set indexes to `{0, 1, 2, 3, 4}`, and values to `{-2.0, -1.6, -1.2, -0.8, -0.4}`.
6. Back to the graph editor for SchmidtTrigger. Look inside RefinementN. Build a model for it as shown in figure 10.7. Set the value of Const to `-1.0`. Edit parameters of Pulse: set indexes to `{0, 1, 2, 3, 4}`, and values to `{2.0, 1.6, 1.2, 0.8, 0.4}`.
7. Back to the graph editor for SchmidtTrigger. Drag an FSM director to the graph. Set its controllerName to `Controller`. Connect the actors as shown in figure 10.8.
8. Back to the graph editor opened in step 1. Build the model as shown in figure 10.9. The model generates an input signal (a sinusoid plus Gaussian noise) for the SchmidtTrigger and plots its output. Edit parameters of Ramp: set init to `-PI/2`, and step to `PI/20`. Edit parameters of Gaussian: set standardDeviation to `0.2`.
9. Run the model for 200 iterations. A sample result is shown in figure 10.10.

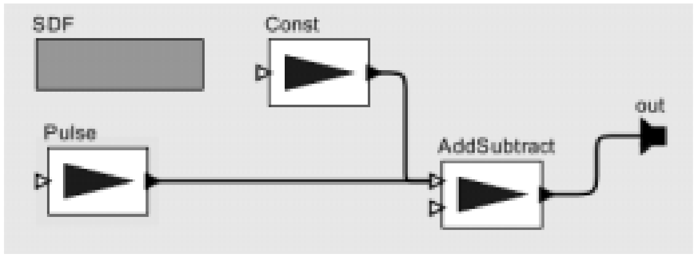


FIGURE 10.7. Model for the refinements in SchmidtTrigger.

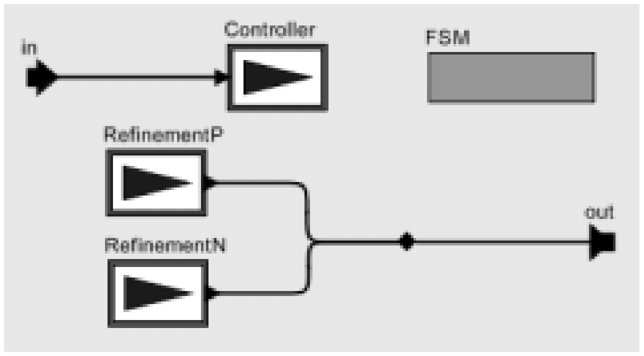


FIGURE 10.8. The SchmidtTrigger modal model.

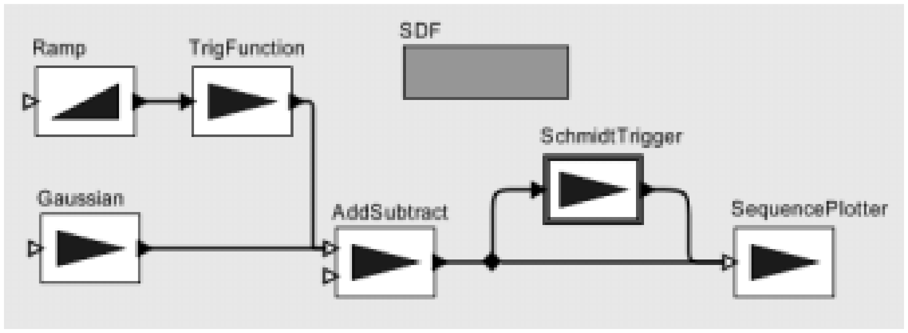


FIGURE 10.9. The top-level model with the SchmidtTrigger.

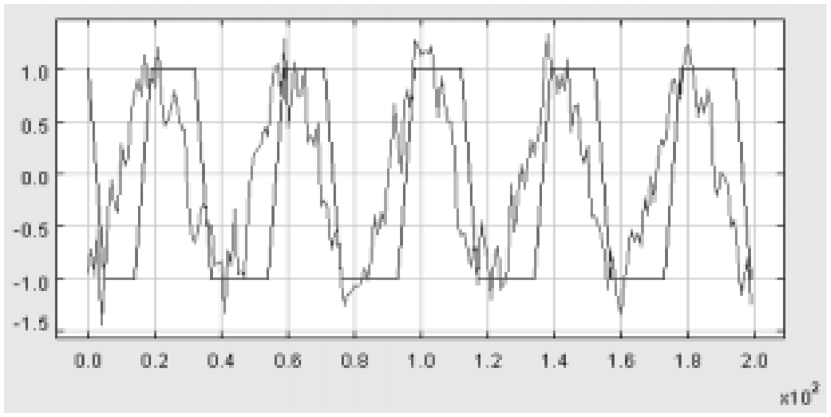


FIGURE 10.10. Sample result of the model shown in figure 10.9.

10.4.2 Implementation

The classes in the FSM kernel package that support modal models are shown in figure 10.11. The execution of a modal model is summarized below.

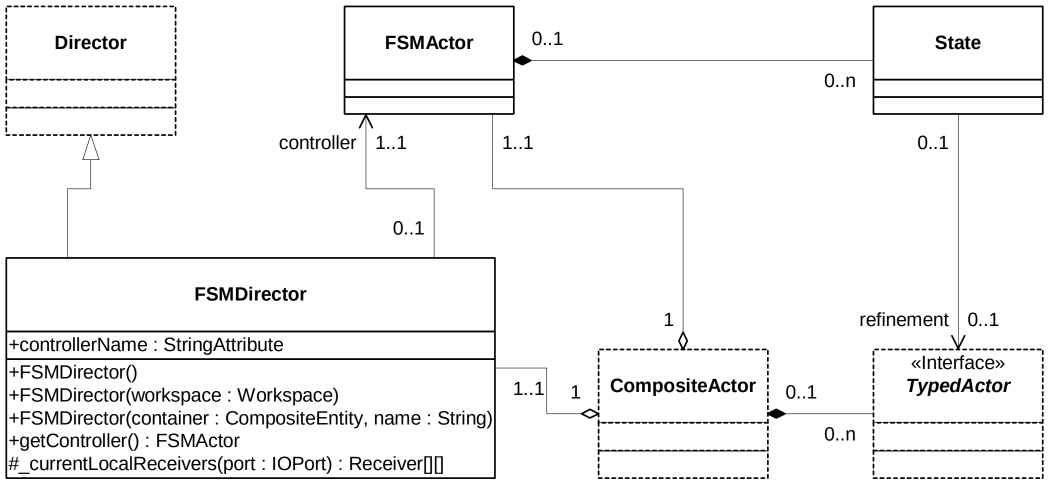

Director
FSMActor
State
0..1
0..n
controller
1..1
1..1
0..1
0..1
FSMDirector
+controllerName : StringAttribute
+FSMDirector()
+FSMDirector(workspace : Workspace)
+FSMDirector(container : CompositeEntity, name : String)
+getController() : FSMActor
#_currentLocalReceivers(port : IOPort) : Receiver[][]
1
refinement
0..1
1..1
1
CompositeActor
0..1
0..n
«Interface»
TypedActor

FIGURE 10.11. FSM kernel classes that support modal models.

When a modal model is fired:

1. The FSM director transfers the input tokens from the outside domain to the mode controller and to the refinement of its current state.
2. The preemptive transitions from the current state of the mode controller are examined. If there is an enabled transition, execute the choice actions of the transition, go to step 5.
3. Fire the refinement of the current state.
4. The non-preemptive transitions from the current state of the mode controller are examined. If there is an enabled transition, execute the choice actions of the transition.
5. Any output token produced by the mode controller or the refinement is transferred to the outside domain.

To make a transition preemptive, set its *preemptive* parameter to true. The mode controller does not change state during successive firings in one iteration in order to support outside domains that iterate to a fixed point. In postfire(), if there is an enabled transition in the latest firing:

1. Execute the commit actions of the transition.
2. Set the current state of the mode controller to the destination state of the transition.
3. If the value of the *reset* parameter of the transition is true, the refinement of the destination state is initialized.

10.4.3 Applications

Hybrid System Modeling. An HSDirector class that extends the FSMDirector class is created for modeling hybrid systems with FSMs and continuous-time (CT) models. An example is presented in section 7.6.3. Execution control is discussed in section 7.7.6.

Communication Protocol Modeling. Hierarchical FSMs are used to model protocol control logic. The timing characteristics of the communication channel are captured by discrete-event (DE) models. We have applied this approach to the alternating bit protocol. The detailed models can be found in the FSM domain demo directory ($PTII/ptolemy/domains/fsm/demo/ABP).

11 CSP Domain

Author: *Neil Smyth*
Contributors: *John S. Davis II*
Bilung Lee
Steve Neuendorffer

11.1 Introduction

The communicating sequential processes (CSP) domain in Ptolemy II models a system as a network of sequential processes that communicate by passing messages synchronously through channels. If a process is ready to send a message, it blocks until the receiving process is ready to accept the message. Similarly if a process is ready to accept a message, it blocks until the sending process is ready to send the message. This model of computation is non-deterministic as a process can be blocked waiting to send or receive on any number of channels. It is also highly concurrent.

The CSP domain is based on the model of computation (MoC) first proposed by Hoare [40][41] in 1978. In this MoC, a system is modeled as a network of processes communicate solely by passing messages through unidirectional channels. The transfer of messages between processes is via *rendezvous*, which means both the sending and receiving of messages from a channel are *blocking*: i.e. the sending or receiving process stalls until the message is transferred. Some of the notation used here is borrowed from Gregory Andrews' book on concurrent programming [4], which refers to rendezvous-based message passing as *synchronous message passing*.

Applications for the CSP domain include resource management and high level system modeling early in the design cycle. Resource management is often required when modeling embedded systems, and to further support this, a notion of time has been added to the model of computation used in the domain. This differentiates our CSP model from those more commonly encountered, which do not typically have any notion of time, although several versions of timed CSP have been proposed [38]. It might thus be more accurate to refer to the domain using our model of computation as the "Timed CSP" domain, but since it can be used with and without time, it is simply referred to as the CSP domain.

11.2 Using CSP

There are two basic issues that must be addressed when using the CSP domain:

- Unconditional vs. conditional rendezvous
- Time

11.2.1 Unconditional vs. Conditional Rendezvous

The basic communication statements send() and get() correspond to rendezvous communication in the CSP domain. Because of the domain framework, fact that a rendezvous is occurring on every communication is transparent to the actor code. However, this rendezvous is unconditional; an actor can only attempt to communicate on one port at a time. To realize the full power of the CSP domain, which allows non-deterministic rendezvous, it is necessary to write custom actors that use the conditional communication constructs in the CSPActor base class. There are three steps involved:

1) Create a ConditionalReceive or ConditionalSend branch for each guarded communication statement, depending on the communication. Pass each branch a unique integer identifier, starting from zero, when creating it.

2) Pass the branches to the chooseBranch() method in CSPActor. This method evaluates the guards, and decides which branch gets to rendezvous, performs the rendezvous and returns the identification number of the branch that succeeded. If all of the guards were false, -1 is returned.

3) Execute the statements for the guarded communication that succeeded.

A sample template for executing a conditional communication is shown in figure 11.1. This template corresponds to the CDO construct in CSP, described in section 11.3.2. In creating the ConditionalSend and ConditionalReceive branches, the first argument represents the guard. The second and third

```
boolean continueCDO = true;
while (continueCDO) {
   // step 1:
   ConditionalBranch[] branches = new ConditionalBranch[#branchesRequired];
   // Create a ConditionalReceive or a ConditionalSend for each branch
   // e.g. branches[0] = new ConditionalReceive((guard), input, 0, 0);

   // step 2:
   int result = chooseBranch(branches);

   // step 3:
   if (result == 0) {
      // execute statements associated with first branch
   } else if (result == 1) {
      // execute statements associated with second branch.
   } else if ... // continue for each branch ID

   } else if (result == -1) {
      // all guards were false so exit CDO.
      continueCDO = false;
   } else {
      // error
   }
}
```

FIGURE 11.1. Template for executing a CDO construct.

arguments represent the port and channel to send or receive the message on. The fourth argument is the identifier assigned to the branch. The choice of placing the guard in the constructor was made to keep the syntax of using guarded communication statements to the minimum, and to have the branch classes resemble the guarded communication statements they represent as closely as possible. This can give rise to the case where the Token specified in a ConditionalSend branch may not yet exist, but this has no effect because once the guard is false, the token in a ConditionalSend is never referenced.

The code for using a CIF is similar to that in figure 11.1 except that the surrounding while loop is omitted and the case when the identifier returned is -1 does nothing. At some stage the steps involved in using a CIF or a CDO may be automated using a pre-parser, but for now the user must follow the approach described above.

Figure 11.2 shows some actual code based on the template above that implements a buffer process. This process repeatedly rendezvous on its input port and its output port, buffering the data if the reading process is not yet ready for the writing process. It is worth pointing out that if most channels in a model are buffered in this way, it may be more reasonable to create the model in the PN domain which implicitly has an unbounded buffer on every channel.

11.2.2 Time

The CSP domain does not currently use the fireAt() mechanism to model time. If an actor wishes be delayed a certain amount of time during execution of the model, it must derive from CSPActor. each process in the CSP domain is able to delay itself, either for some period from the current model time or until the next occasion time deadlock is reached at the current model time. The two methods to call are delay() and waitForDeadlock(). If a process delays itself for zero time from the current time, the pro-

```
boolean guard = false;
boolean continueCDO = true;
ConditionalBranch[] branches = new ConditionalBranch[2];
while (continueCDO) {
   // step 1
   guard = (_size < depth);
   branches[0] = new ConditionalReceive(guard, input, 0, 0);
   guard = (_size > 0);
   branches[1] = new ConditionalSend(guard, output, 0, 1, _buffer[_readFrom]);

   // step 2
   int successfulBranch = chooseBranch(branches);

   // step 3
   if (successfulBranch == 0) {
      _size++;
      _buffer[_writeTo] = branches[0].getToken();
      _writeTo = ++_writeTo % depth;
   } else if (successfulBranch == 1) {
      _size--;
      _readFrom = ++_readFrom % depth;
   } else if (successfulBranch == -1) {
      // all guards false so exit CDO
      // Note this cannot happen in this case
      continueCDO = false;
   } else {
      throw new TerminateProcessException(getName() + ": " +
            "branch id returned during execution of CDO.");
   }
}
```

FIGURE 11.2. Code used to implement the buffer process described in figure 11.1.

cess will continue immediately. Thus delay(0.0) is not equivalent to waitForDeadlock()

As far as each process is concerned, time can only increase while it is blocked waiting to rendezvous or when it is delayed. A process can be aware of the current model time, but it should only ever affect the model time by delaying its execution, thus forcing time to advance. The method setCurrentTime() should never be called from a process. However, if no processes are delayed, it is possible to set the model time by calling the setCurrentTime() method of the director. However, this method is present only for composing CSP with other domains.

By default every model in the CSP domain is timed. To use CSP without a notion of time, simply do not use the delay() method. The infrastructure supporting time does not affect the model execution if this method is not used. For more information about the semantics of Timed CSP models, see section 11.3.4

11.3 Properties of the CSP Domain

At the core of CSP communication semantics are two fundamental ideas. First is the notion of atomic communication and second is the notion of nondeterministic choice. It is worth mentioning a related model of computation known as the calculus of communicating systems (CCS) that was independently developed by Robin Milner in 1980 [65]. The communication semantics of CSP are identical to those of CCS.

11.3.1 Atomic Communication: Rendezvous

Atomic communication is carried out via rendezvous and implies that the sending and receiving of a message occur simultaneously. During rendezvous both the sending and receiving processes block until the other side is ready to communicate; the act of sending and receiving is indistinguishable activities since one can not happen without the other. A real world analogy to rendezvous can be found in telephone communications (without answering machines). Both the caller and callee must be simultaneously present for a phone conversation to occur. Figure 11.3 shows the case where one process is ready to send before the other process is ready to receive. The communication of information in this way can be viewed as a distributed assignment statement.

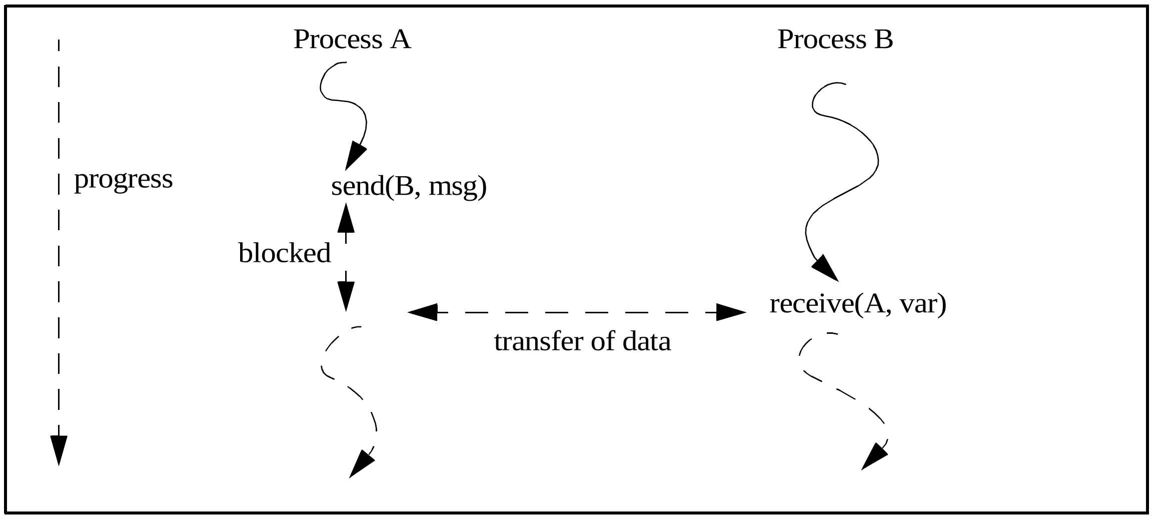


FIGURE 11.3. Illustrating how processes block waiting to rendezvous

The sending process places some data in the message that it wants to send. The receiving process assigns the data in the message to a local variable. Of course, the receiving process may decide to ignore the contents of the message and only concern itself with the fact that a message arrived.

11.3.2 Choice: Nondeterministic Rendezvous

Nondeterministic choice provides processes with the ability to randomly select between a set of possible atomic communications. We refer to this ability as nondeterministic rendezvous and herein lies much of the expressiveness of the CSP model of computation. The CSP domain implements nondeterministic rendezvous via *guarded communication statements.* A guarded communication statement has the form

```
guard; communication => statements;
```

The *guard* is only allowed to reference local variables, and its evaluation cannot change the state of the process. For example it is not allowed to assign to variables, only reference them. The *communication* must be a simple send or receive, i.e. another conditional communication statement cannot be placed here. *Statements* can contain any arbitrary sequence of statements, including more conditional communications.

If the guard is false, then the communication is not attempted and the statements are not executed. If the guard is true, then the communication is attempted, and if it succeeds, the following statements are executed. The guard may be omitted, in which case it is assumed to be true.

There are two conditional communication constructs built upon the guarded communication statements: **CIF** and **CDO**. These are analogous to the *if* and *while* statements in most programming languages. They should be read as "conditional if" and "conditional do". Note that each guarded communication statement represents one *branch* of the CIF or CDO. The communication statement in each branch can be either a send or a receive, and they can be mixed freely.

CIF: The form of a CIF is

```
CIF {
     G1;C1 => S1;
[]
     G2;C2 => S2;
[]
     ...
}
```

For each branch in the CIF, the guard (*G1*, *G2*,...) is evaluated. If it is true (or absent, which implies true), then the associated communication statement is enabled. If one or more branch is enabled, then the entire construct blocks until one of the communications succeeds. If more than one branch is enabled, the choice of which enabled branch succeeds with its communication is made nondeterministically. Once the successful communication is carried out, the associated statements are executed and the process continues. If all of the guards are false, then the process continues executing statements after the end of the CIF.

It is important to note that, although this construct is analogous to the common *if* programming construct, its behavior is very different. In particular, all guards of the branches are evaluated concurrently, and the choice of which one succeeds does not depend on its position in the construct. The notation "[]" is used to hint at the parallelism in the evaluation of the guards. In a common *if*, the branches are evaluated sequentially and the first branch that is evaluated to true is executed. The CIF construct also depends on the semantics of the communication between processes, and can thus stall the progress of the thread if none of the enabled branches is able to rendezvous.

CDO: The form of the CDO is

```
CDO {
     G1;C1 => S1;
[]
     G2;C2 => S2;
[]
     ...
}
```

The behavior of the CDO is similar to the CIF in that for each branch the guard is evaluated and the choice of which enabled communication to make is taken non-deterministically. However, the CDO repeats the process of evaluating and executing the branches until *all* the guards return false. When this happens the process continues executing statements after the CDO construct.

An example use of a CDO is in a buffer process which can both accept and send messages, but has to be ready to do both at any stage. The code for this would look similar to that in figure 11.4. Note that in this case both guards can never be simultaneously false so this process will execute the CDO forever.

11.3.3 Deadlock

A deadlock situation is one in which none of the processes can make progress: they are all either blocked trying to rendezvous or they are delayed (see the next section). Thus, two types of deadlock can be distinguished:

real deadlock - all active processes are blocked trying to communicate

time deadlock - all active processes are either blocked trying to communicate or are delayed, and at least one processes is delayed.

11.3.4 Time

In the CSP domain, *time* is centralized. That is, all processes in a model share the same time, referred to as the *current model time*. Each process can only choose to *delay* itself for some period relative to the current model time, or a process can wait for time deadlock to occur at the current model time. In both cases, a process is said to be *delayed*.

When a process delays itself for some length of time from the current model time, it is suspended

```
CDO {
    (room in buffer?); receive(input, beginningOfBuffer) => update pointer to beginning of buffer;
[]
    (messages in buffer?); send(output, endOfBuffer) => update pointer to end of buffer;
}
```

FIGURE 11.4. Example of how a CDO might be used in a buffer

until time has sufficiently advanced, at which stage it wakes up and continues. If the process delays itself for zero time, this will have no effect and the process will continue executing.

A process can also choose to delay its execution until the next occasion a time deadlock is reached. The process resumes at the same model time at which it delayed, and this is useful as a model can have several sequences of actions at the same model time. The next occasion time deadlock is reached, any processes delayed in this manner will continue, and time will not be advanced. An example of using time in this manner can be found in section 11.5.2.

Time may be *advanced* when all the processes are delayed or are blocked trying to rendezvous, and at least one process is delayed. If one or more processes are delaying until a time deadlock occurs, these processes are woken up and time is not advanced. Otherwise, the current model time is advanced just enough to wake up at least one process. Note that there is a semantic difference between a process delaying for zero time, which will have no effect, and a process delaying until the next occasion a time deadlock is reached.

Note also that time, as perceived by a single process, cannot change during its normal execution; only at rendezvous points or when the process delays can time change. A process can be aware of the centralized time, but it cannot influence the current model time except by delaying itself. The choice for modeling time was in part influenced by Pamela [30], a run time library that is used to model parallel programs.

11.3.5 Differences from Original CSP Model as Proposed by Hoare

The model of computation used by the CSP domain differs from the original CSP [40] model in two ways. First, a notion of time has been added. The original proposal had no notion of time, although there have been several proposals for timed CSP [38]. Second, as mentioned in section 11.3.2, it is possible to use both send and receive in guarded communication statements. The original model only allowed receives to appear in these statements, though Hoare subsequently extended their scope to allow both communication primitives [41].

One final thing to note is that in much of the CSP literature, send is denoted using a “!”, pronounced “bang”, and receive is denoted using a “?”, pronounced “query”. This syntax was what was used in the original CSP paper by Hoare. For example, the languages Occam [15] and Lotos [23] both follow this syntax. In the CSP domain in Ptolemy II we use *send* and *get*, the choice of which is influenced by the desire to maintain uniformity of syntax across domains in Ptolemy II that use message passing. This supports the heterogeneity principle in Ptolemy II which enables the construction and inter-operability of executable models that are built under a variety of models of computation. Similarly, the notation used in the CSP domain for conditional communication constructs differs from that commonly found in the CSP literature.

11.4 The CSP Software Architecture

11.4.1 Class Structure

In a CSP model, the director is an instance of *CSPDirector.* Since the model is controlled by a CSPDirector, all the receivers in the ports are *CSPReceivers*. The combination of the CSPDirector and CSPReceivers in the ports gives a model CSP semantics. The CSP domain associates each channel with exactly one receiver, located at the receiving end of the channel. Thus any process that sends or receives to any channel will rendezvous at a CSPReceiver. Figure 11.5 shows the static structure diagram of the five main classes in the CSP kernel, and a few of their associations. These are the classes that provide all the infrastructure needed for a CSP model.

CSPDirector: This gives a model CSP semantics. It takes care of starting all the processes and controls/responds to both real and time deadlocks. It also maintains and advances the model time when necessary.

CSPReceiver: This ensures that communication of messages between processes is via rendezvous.

CSPActor: This adds the notion of time and the ability to perform conditional communication.

ConditionalReceive, ConditionalSend: This is used to construct the guarded communication statements necessary for the conditional communication constructs.

11.4.2 Starting the model

The director creates a thread for each actor under its control in its initialize() method. It also invokes the initialize() method on each actor at this time. The director starts the threads in its prefire() method, and detects and responds to deadlocks in its fire() method. The thread for each actor is an instance of ProcessThread, which invokes the prefire(), fire() and postfire() methods for the actor until it finishes or is terminated. It then invokes the wrapup() method and the thread dies.

Figure 11.7 shows the code executed by the ProcessThread class. Note that it makes no assumption about the actor it is executing, so it can execute any domain-polymorphic actor as well as CSP domain-specific actors. In fact, any other domain actor that does not rely on the specifics of its parent domain can be executed in the CSP domain by the ProcessThread.

11.4.3 Detecting deadlocks:

For deadlock detection, the director maintains three counts:

- the number of *active* processes which are threads that have started but have not yet finished
- the number of *blocked* processes which is the number of processes that are blocked waiting to rendezvous, and

```
director.initialize() =>
   create a thread for each actor
   update count of active processes with the director
   call initialize() on each actor

director.prefire() => start the process threads =>
   calls actor.prefire()
   calls actor.fire()
   calls actor.postfire()
   repeat.

director.fire() => handle deadlocks until a real deadlock occurs.

director.postfire() =>
   return a boolean indicating if the execution of the model should continue for another iteration

director.wrapup() => terminate all the processes =>
   calls actor.wrapup()
   decrease the count of active processes with the director
```

FIGURE 11.6. Sequence of steps involved in setting up and controlling the model.

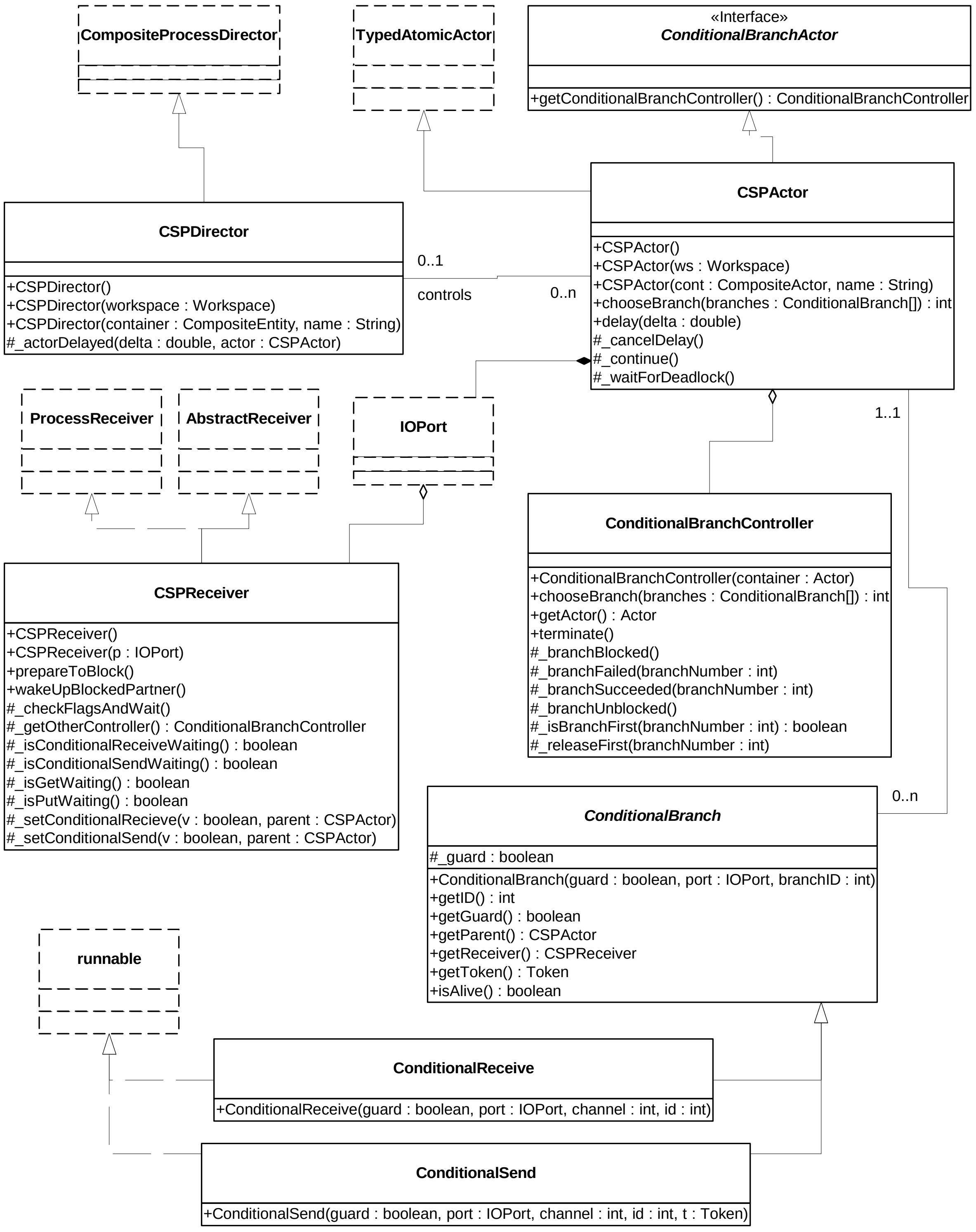

CompositeProcessDirector
TypedAtomicActor
«Interface»
ConditionalBranchActor
+getConditionalBranchController() : ConditionalBranchController
CSPActor
+CSPActor()
+CSPActor(ws : Workspace)
+CSPActor(cont : CompositeActor, name : String)
+chooseBranch(branches : ConditionalBranch[]) : int
+delay(delta : double)
#_cancelDelay()
#_continue()
#_waitForDeadlock()
CSPDirector
+CSPDirector()
+CSPDirector(workspace : Workspace)
+CSPDirector(container : CompositeEntity, name : String)
#_actorDelayed(delta : double, actor : CSPActor)
0..1
controls
0..n
1..1
ProcessReceiver
AbstractReceiver
IOPort
ConditionalBranchController
+ConditionalBranchController(container : Actor)
+chooseBranch(branches : ConditionalBranch[]) : int
+getActor() : Actor
+terminate()
#_branchBlocked()
#_branchFailed(branchNumber : int)
#_branchSucceeded(branchNumber : int)
#_branchUnblocked()
#_isBranchFirst(branchNumber : int) : boolean
#_releaseFirst(branchNumber : int)
CSPReceiver
+CSPReceiver()
+CSPReceiver(p : IOPort)
+prepareToBlock()
+wakeUpBlockedPartner()
#_checkFlagsAndWait()
#_getOtherController() : ConditionalBranchController
#_isConditionalReceiveWaiting() : boolean
#_isConditionalSendWaiting() : boolean
#_isGetWaiting() : boolean
#_isPutWaiting() : boolean
#_setConditionalRecieve(v : boolean, parent : CSPActor)
#_setConditionalSend(v : boolean, parent : CSPActor)
ConditionalBranch
0..n
#_guard : boolean
+ConditionalBranch(guard : boolean, port : IOPort, branchID : int)
+getID() : int
+getGuard() : boolean
+getParent() : CSPActor
+getReceiver() : CSPReceiver
+getToken() : Token
+isAlive() : boolean
runnable
ConditionalReceive
+ConditionalReceive(guard : boolean, port : IOPort, channel : int, id : int)
ConditionalSend
+ConditionalSend(guard : boolean, port : IOPort, channel : int, id : int, t : Token)

FIGURE 11.5. Static structure diagram for classes in the CSP kernel.

- the number of *delayed* processes, which is the number of processes waiting for time to advance plus the number of processes waiting for time deadlock to occur at the current model time.

When the number of blocked processes equals the number of active processes, then real deadlock has occurred and the fire method of the director returns. When the number of blocked plus the number of delayed processes equals the number of active processes, and at least one process is delayed, then time deadlock has occurred. If at least one process is delayed waiting for time deadlock to occur at the current model time, then the director wakes up all such processes and does not advance time. Otherwise the director looks at its list of processes waiting for time to advance, chooses the earliest one and advances time sufficiently to wake it up. It also wakes up any other processes due to be awakened at the new time. The director checks for deadlock each occasion a process blocks, delays or dies.

For the director to work correctly, these three counts need to be accurate at all stages of the model execution, so when they are updated becomes important. Keeping the active count accurate is relatively simple; the director increases it when it starts the thread, and decreases it when the thread dies. Likewise the count of delayed processes is straightforward; when a process delays, it increases the count of delayed processes, and the director keeps track of when to wake it up. The count is decreased when a delayed process resumes.

However, due to the conditional communication constructs, keeping the blocked count accurate requires a little more effort. For a basic send or receive, a process is registered as being blocked when it arrives at the rendezvous point before the matching communication. The blocked count is then decreased by one when the corresponding communication arrives. However what happens when an actor is carrying out a conditional communication construct? In this case the process keeps track of all of the branches for which the guards were true, and when all of those are blocked trying to rendezvous, it registers the process as being blocked. When one of the branches succeeds with a rendezvous, the process is registered as being unblocked.

```
public void run() {
   try {
      boolean iterate = true;
      while (iterate) {
         // container is checked for null to detect the termination
         // of the actor.
         iterate = false;
         if ((Entity)_actor).getContainer() != null && _actor.prefire()) {
            _actor.fire();
            iterate = _actor.postfire();
         }
      }
   } catch (TerminateProcessException t) {
      // Process was terminated early
   } catch (IllegalActionException e) {
      _manager.fireExecutionError(e);
   } finally {
      try {
         _actor.wrapup();
      } catch (IllegalActionExeption e) {
         _manager.fireExecutionError(e);
      }
      _director.decreaseActiveCount();
   }
}
```

FIGURE 11.7. Code executed by ProcessThread.run().

11.4.4 Terminating the model

A process can finish in one of two ways: either by returning false in its prefire() or postfire() methods, in which case it is said to have finished *normally*, or by being terminated *early* by a TerminateProcessException. For example, if a source process is intended to send ten tokens and then finish, it would exit its fire() method after sending the tenth token, and return false in its postfire() method. This causes the ProcessThread, see figure 11.7, representing the process, to exit the while loop and execute the finally clause. The finally clause calls wrapup() on the actor it represents, decreases the count of active processes in the director, and the thread representing the process dies.

A TerminateProcessException is thrown whenever a process tries to communicate via a channel whose receiver has its *finished* flag set to true. When a TerminateProcessException is caught in ProcessThread, the finally clause is also executed and the thread representing the process dies.

To terminate the model, the director sets the *finished* flag in each receiver. The next occasion a process tries to send to or receive from the channel associated with that receiver, a TerminateProcessException is thrown. This mechanism can also be used in a selective fashion to terminate early any processes that communicate via a particular channel. When the director controlling the execution of the model detects real deadlock, it returns from its fire() method. In the absence of hierarchy, this causes the wrapup() method of the director to be invoked. It is the wrapup() method of the director that sets the finished flag in each receiver. Note that the TerminateProcessException is a runtime exception so it does not need to be declared as being thrown.

There is also the option of abruptly terminating all the processes in the model by calling terminate() on the director. This method differs from the approach described in the previous paragraph in that it stops all the threads immediately and does not give them a chance to update the model state. After calling this method, the state of the model is unknown and so the model should be recreated after calling this method. This method is only intended for situations when the execution of the model has obviously gone wrong, and for it to finish normally would either take too long or could not happen. It should rarely be called.

11.4.5 Pausing/Resuming the Model

Pausing and resuming a model does not affect the outcome of a particular execution of the model, only the rate of progress. The execution of a model can be paused at any stage by calling the pause() method on the director. This method is blocking, and will only return when the model execution has been successfully paused. To pause the execution of a model, the director sets a *paused* flag in every receiver, and the next occasion a process tries to send to or receive from the channel associated with that receiver, it is paused. The whole model is paused when all the active processes are delayed, paused or blocked. To resume the model, the resume() method can similarly be called on the director This method resets the paused flag in every receiver and wakes up every process waiting on a receiver lock. If a process was paused, it sees that it is no longer paused and continues. The ability to pause and resume the execution of a model is intended primarily for user interface control.

11.5 Example CSP Applications

Several example applications have been developed which serve to illustrate the modeling capabilities of the CSP model of computation in Ptolemy II. Each demonstration incorporates several features of CSP and the general Ptolemy II framework. The applications are described here, but not the code. See the directory $PTII/ptolemy/domains/csp/demo for the code.

The first demonstration, *dining philosophers*, serves as a natural example of core CSP communication semantics. This demonstration models nondeterministic resource contention, e.g., five philosophers randomly accessing chopstick resources. Nondeterministic rendezvous serves as a natural modeling tool for this example. The second example, *hardware bus contention*, models deterministic resource contention in the context of time. As will be shown, the determinacy of this demonstration constrains the natural nondeterminacy of the CSP semantics and results in difficulties. Fortunately these difficulties can be smoothly circumvented by the timing model that has been integrated into the CSP domain.

11.5.1 Dining Philosophers

Nondeterministic Resource Contention. This implementation of the dining philosophers problem illustrates both time and conditional communication in the CSP domain. Five philosophers are seated at a table with a large bowl of food in the middle. Between each pair of philosophers is one chopstick, and to eat, a philosopher needs both the chopsticks beside him. Each philosopher spends his life in the following cycle: thinks for a while, gets hungry, picks up one of the chopsticks beside him, then the other, eats for a while and puts the chopsticks down on the table again. If a philosopher tries to grab a chopstick but it is already being used by another philosopher, then the philosopher waits until that chopstick becomes available. This implies that no neighboring philosophers can eat at the same time and at most two philosophers can eat at a time.

The Dining Philosophers problem was first proposed by Edsger W. Dijkstra in 1965. It is a classic concurrent programming problem that illustrates the two basic properties of concurrent programming:

Liveness. How can we design the program to avoid deadlock, where none of the philosophers can make progress because each is waiting for someone else to do something?

Fairness. How can we design the program to avoid starvation, where one of the philosophers could make progress but does not because others always go first?

This implementation uses an algorithm that lets each philosopher randomly chose which chopstick to pick up first (via a CDO), and all philosophers eat and think at the same rates. Each philosopher and each chopstick is represented by a separate process. Each chopstick has to be ready to be used by either philosopher beside it at any time, hence the use of a CDO. After it is grabbed, it blocks waiting for a message from the philosopher that is using it. After a philosopher grabs both the chopsticks next to him, he eats for a random time. This is represented by calling delay() with the random interval to eat for. The same approach is used when a philosopher is thinking. Note that because messages are passed by rendezvous, the blocking of a philosopher when it cannot obtain a chopstick is obtained for free.

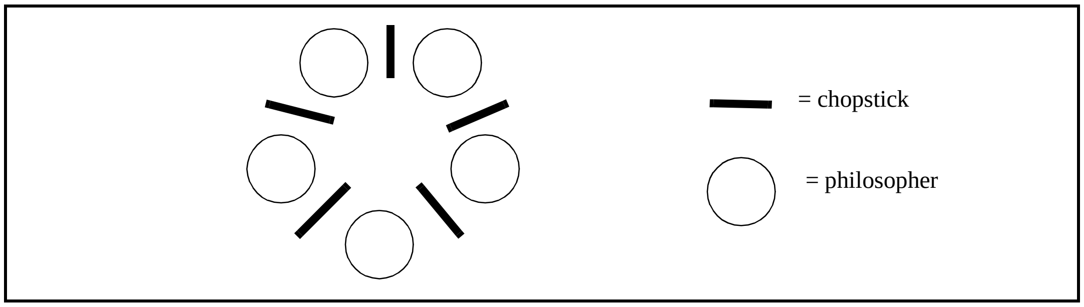


FIGURE 11.8. Illustration of the dining philosophers problem.

This algorithm is fair, as any time a chopstick is not being used, and both philosophers try to use it, they both have an equal chance of succeeding. However this algorithm does not guarantee the absence of deadlock, and if it is let run long enough this will eventually occur. The probability that deadlock occurs sooner increases as the thinking times are decreased relative to the eating times.

11.5.2 Hardware Bus Contention

Deterministic Resource Contention. This demonstration consists of a controller, *N* processors and a memory block, as shown in Figure 11.9. At randomly selected points in time, each processor requests permission from the controller to access the memory block. The processors each have priorities associated with them and in cases where there is a simultaneous memory access request, the controller grants permission to the processor with the highest priority. Due to the atomic nature of rendezvous, it is impossible for the controller to check priorities of incoming requests at the same time that requests are occurring. To overcome this difficulty, an alarm is employed. The alarm is started by the controller immediately following the first request for memory access at a given instant in time. It is awakened when a delay block occurs to indicate to the controller that no more memory requests will occur at the given point in time. Hence, the alarm uses CSP's notion of delay blocking to make deterministic an inherently non-deterministic activity.

11.6 Technical Details

11.6.1 Rendezvous Algorithm

In CSP, the locking point for all communication between processes is the receiver. Any occasion a process wishes to send or receive, it must first acquire the lock for the receiver associated with the channel it is communicating over. Two key facts to keep in mind when reading the following algorithms are that each channel has exactly one receiver associated with it and that at most one process can be trying to send to (or receive from) a channel at any stage. The constraint that each channel can have at most one process trying to send to (or receive from) a channel at any stage is not currently enforced, but an exception will be thrown if such a model is not constructed.

The rendezvous algorithm is *entirely symmetric* for the put() and the get(), except for the direction the token is transferred. This helps reduce the deadlock situations that could arise and also makes the interaction between processes more understandable and easier to explain. The algorithm controlling how a get() proceeds is shown in figure 11.10. The algorithm for a put() is exactly the same except that

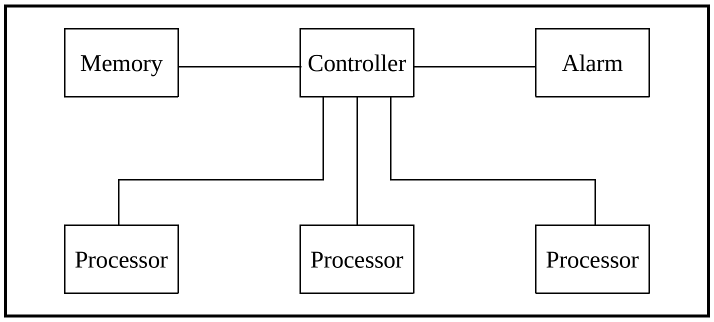


FIGURE 11.9. Illustration of the Hardware Bus Contention example.

put and get are swapped everywhere. Thus it suffices to explain what happens when a get() arrives at a receiver, i.e. when a process tries to receive from the channel associated with the receiver.

When a get() arrives at a receiver, a put() is either already waiting to rendezvous or it is not. Both the get() and put() methods are entirely synchronized on the receiver so they cannot happen simultaneously (only one thread can possess a lock at any given time). Without loss of generality assume a get() arrives before a put(). The rendezvous mechanism is basically three steps: a get() arrives, a put() arrives, the rendezvous completes.

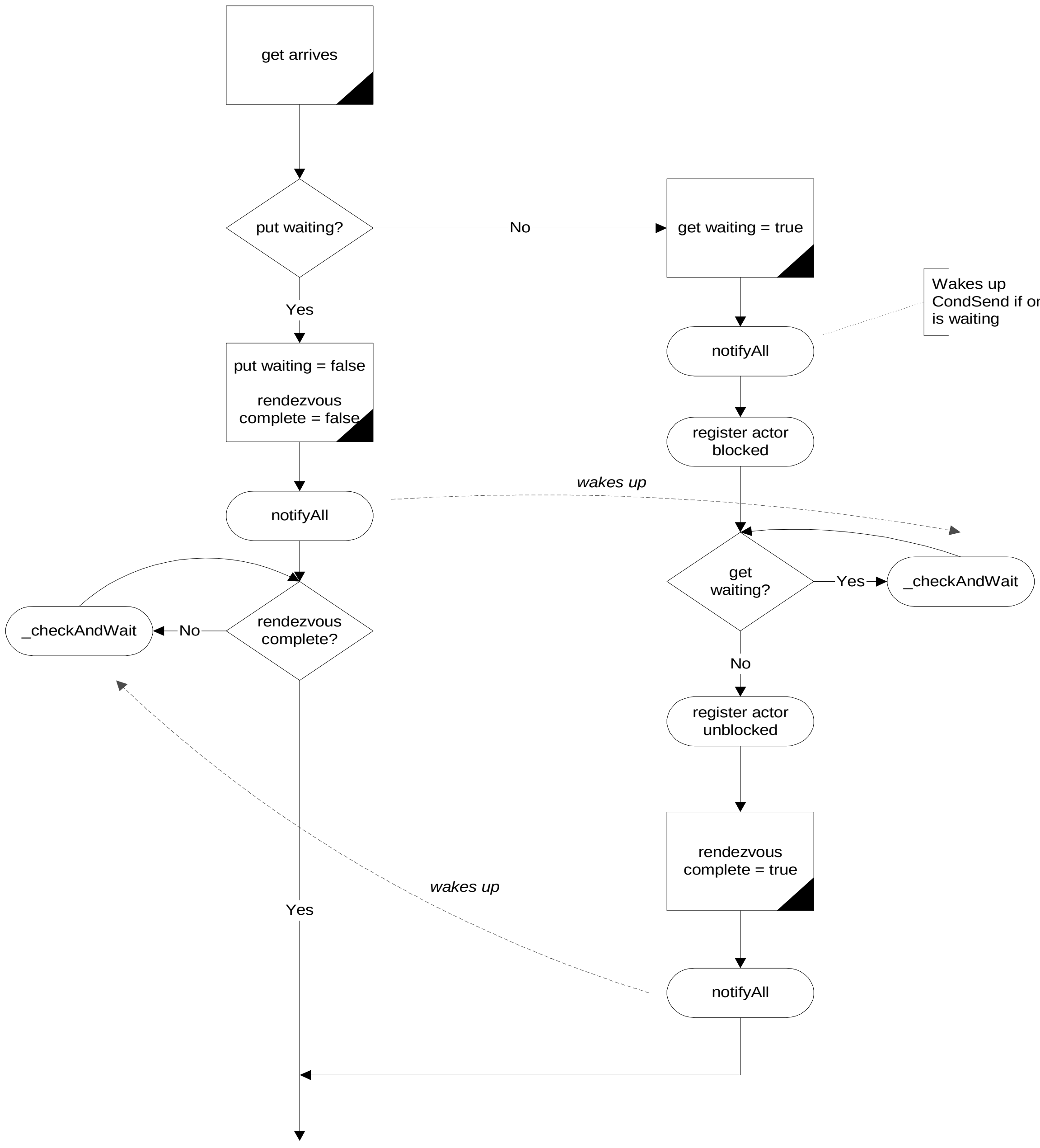


FIGURE 11.10. Rendezvous algorithm.

(1) When the get() arrives, it sees that it is first and sets a flag saying a get is waiting. It then waits on the receiver lock while the flag is still true,

(2) When a put() arrives, it sets the *getWaiting* flag to false, wakes up any threads waiting on the receiver (including the get), sets the *rendezvousComplete* flag to false and then waits on the receiver while the *rendezvousComplete* flag is false,

(3) The thread executing the get() wakes up, sees that a put() has arrived, sets the *rendezvousComplete* flag to true, wakes up any threads waiting on the receiver, and returns thus releasing the lock. The thread executing the put() then wakes up, acquires the receiver lock, sees that the rendezvous is complete and returns.

Following the rendezvous, the state of the receiver is exactly the same as before the rendezvous arrived, and it is ready to mediate another rendezvous. It is worth noting that the final step, of making sure the second communication to arrive does not return until the rendezvous is complete, is necessary to ensure that the correct token gets transferred. Consider the case again when a get() arrives first, except now the put() returns immediately if a get() is already waiting. A put() arrives, places a token in the receiver, sets the get waiting flag to false and returns. Now suppose another put() arrives before the get() wakes up, which will happen if the thread the put() is in wins the race to obtain the lock on the receiver. Then the second put() places a new token in the receiver and sets the put waiting flag to true. Then the get() wakes up, and returns with the wrong token! This is known as a *race condition*, which will lead to unintended behavior in the model. This situation is avoided by our design.

11.6.2 Conditional Communication Algorithm

There are two steps involved in executing a CIF or a CDO: first deciding which enabled branch succeeds, then carrying out the rendezvous.

Built on top of rendezvous:
When a conditional construct has more than one enabled branch (guard is true or absent), a new thread is spawned for each enabled branch. The job of the chooseBranch() method is to control these threads and to determine which branch should be allowed to successfully rendezvous. These threads and the mechanism controlling them are entirely separate from the rendezvous mechanism described in section 11.6.1, with the exception of one special case, which is described in section 11.6.3. Thus the conditional mechanism can be viewed as being built on top of basic rendezvous: conditional communication knows about and needs basic rendezvous, but the opposite is not true. Again this is a design decision which leads to making the interaction between threads easier to understand and is less prone to deadlock as there are fewer interaction possibilities to consider.

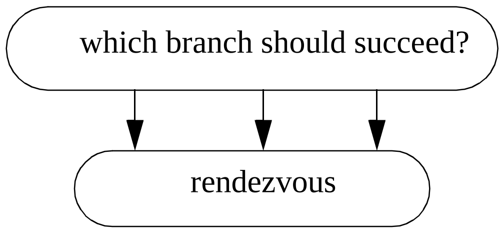


FIGURE 11.11. Conceptual view of how conditional communication is built on top of rendezvous.

Choosing which branch succeeds.
The manner in which the choice of which branch can rendezvous is worth explaining. The chooseBranch() method in CSPActor takes an array of branches as an argument. If all of the guards are false, it returns -1, which indicates that all the branches failed. If exactly one of the guards is true, it performs the rendezvous directly and returns the identification number of the successful branch. The interesting case is when more than one guard is true. In this case, it creates and starts a new thread for each branch whose guard is true. It then waits, on an internal lock, for one branch to succeed. At that point it gets woken up, sets a finished flag in the remaining branches and waits for them to fail. When all the threads representing the branches are finished, it returns the identification number of the successful branch. This approach is designed to ensure that exactly one of the branches created successfully performs a rendezvous.

Algorithm used by each branch:
Similar to the approach followed for rendezvous, the algorithm by which a thread representing a branch determines whether or not it can proceed is entirely *symmetrical* for a ConditionalSend and a ConditionalReceive. The algorithm followed by a ConditionalReceive is shown figure 11.12. Again the locking point is the receiver, and all code concerned with the communication is synchronized on the receiver. The receiver is also where all necessary flags are stored.

Consider three cases.

(1) a conditionalReceive arrives and a put is waiting.

In this case, the branch checks if it is the first branch to be ready to rendezvous, and if so, it is goes ahead and executes a get. If it is not the first, it waits on the receiver. When it wakes up, it checks if it is still alive. If it is not, it registers that it has failed and dies. If it is still alive, it starts again by trying to be the first branch to rendezvous. Note that a put cannot disappear.

(2) a conditionalReceive arrives and a conditionalSend is waiting

When both sides are conditional branches, it is up to the branch that arrives second to check whether the rendezvous can proceed. If both branches are the first to try to rendezvous, the conditionalReceive executes a get(), notifies its parent that it succeeded, issues a notifyAll() on the receiver and dies. If not, it checks whether it has been terminated by chooseBranch(). If it has, it registers with chooseBranch() that it has failed and dies. If it has not, it returns to the start of the algorithm and tries again. This is because a ConditionalSend could disappear. Note that the parent of the first branch to arrive at the receiver needs to be stored for the purpose of checking if both branches are the first to arrive.

This part of the algorithm is somewhat subtle. When the second conditional branch arrives at the rendezvous point it checks that *both* sides are the first to try to rendezvous for their respective processes. If so, then the conditionalReceive executes a get(), so that the conditionalSend is never aware that a conditionalReceive arrived: it only sees the get().

(3) a conditionalReceive arrives first.

It sets a flag in the receiver that it is waiting, then waits on the receiver. When it wakes up, it checks whether it has been killed by chooseBranch. If it has, it registers with chooseBranch that it has failed and dies. Otherwise it checks if a put is waiting. It only needs to check if a put is waiting because if a conditionalSend arrived, it would have behaved as in case (2) above. If a put is waiting, the branch checks if it is the first branch to be ready to rendezvous, and if so it is goes ahead and executes a get. If it is not the first, it waits on the receiver and tries again.

11.6.3 Modification of Rendezvous Algorithm

Consider the case when a conditional send arrives before a get. If all the branches in the conditional communication that the conditional send is a part of are blocked, then the process will register itself as blocked with the director. Then the get comes along, and even though a conditional send is waiting, it too would register itself as blocked. This leads to one too many processes being registered as blocked, which could lead to premature deadlock detection.

To avoid this, it is necessary to modify the algorithm used for rendezvous slightly. The change to

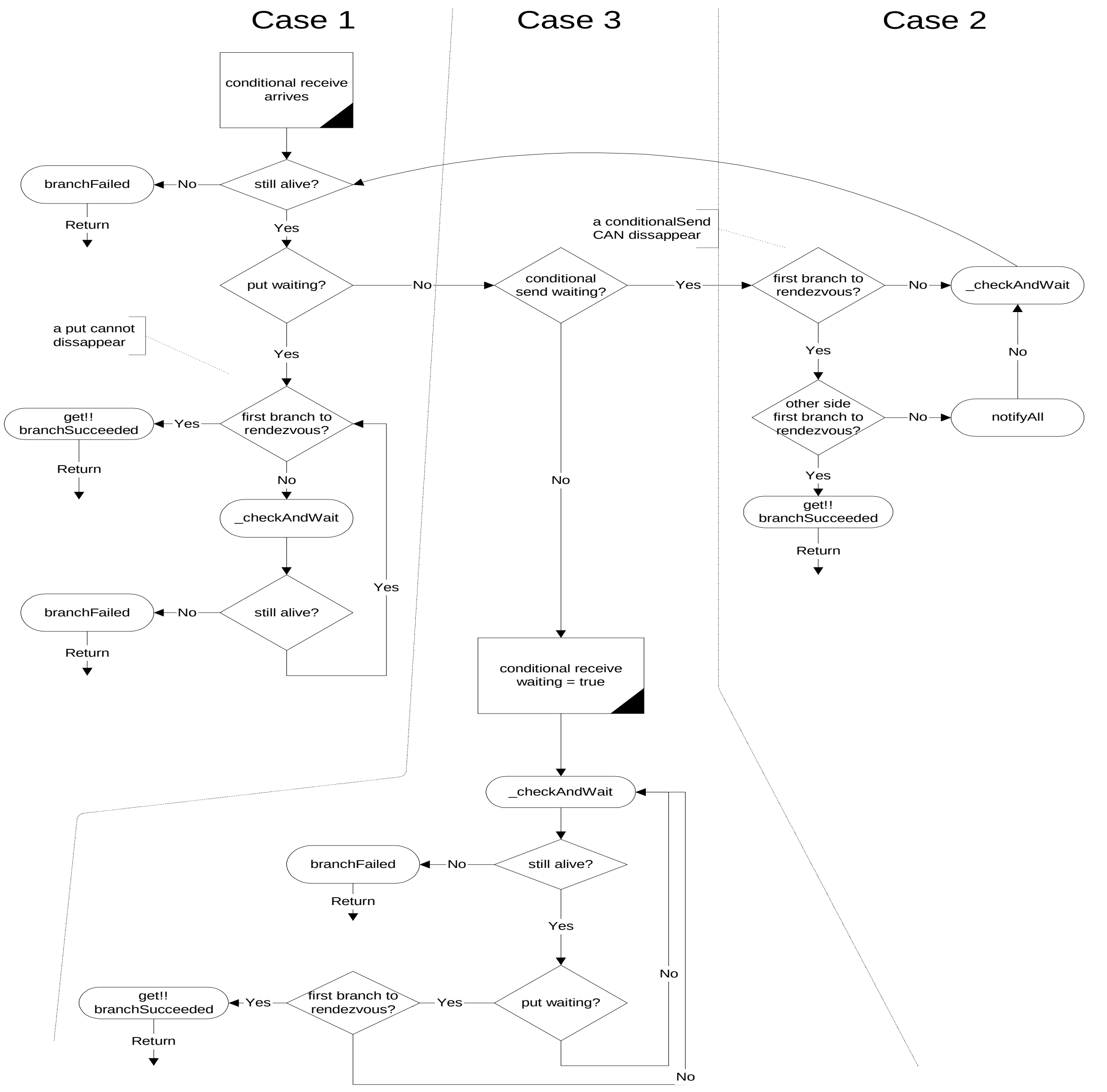

Case 1
Case 3
Case 2
conditional receive arrives
still alive?
branchFailed
No
Yes
Return
put waiting?
a put cannot dissappear
first branch to rendezvous?
get!! branchSucceeded
_checkAndWait
conditional send waiting?
a conditionalSend CAN dissappear
other side first branch to rendezvous?
notifyAll
conditional receive waiting = true

FIGURE 11.12. Algorithm used to determine if a conditional rendezvous branch succeeds or fails

the algorithm is shown in the dashed ellipse in figure 11.13. It does not affect the algorithm except in the case when a conditional send is waiting when a get arrives at the receiver. In this case the process that calls the get should wait on the receiver until the conditional send waiting flag is false. If the conditional send succeeded, and hence executed a put, then the get waiting flag and the conditional send waiting flag should both be false and the actor proceeds through to the third step of the rendezvous. If the conditional send failed, it will have reset the conditional send waiting flag and issued a notifyAll() on the receiver, thus waking up the get and allowing it to properly wait for a put.

The same reasoning also applies to the case when a conditional receive arrives at a receiver before a put.

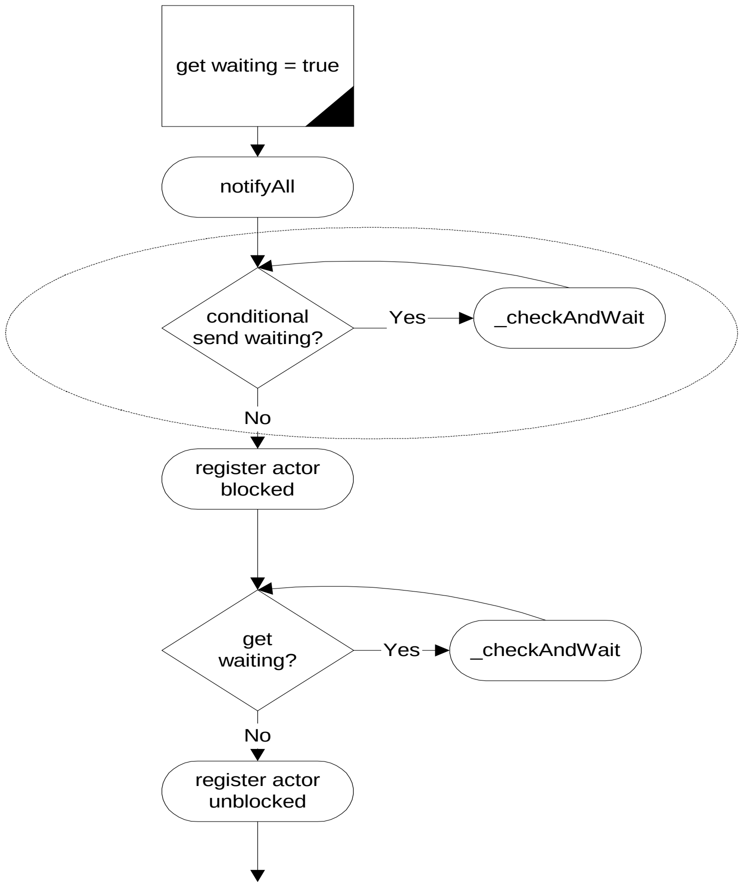


FIGURE 11.13. Modification of rendezvous algorithm, section 11.6.3, shown in ellipse.

12

DDE Domain

Author: John S. Davis II

12.1 Introduction

The distributed discrete event (DDE) model of computation incorporates a distributed notion of time into a dataflow style of communication. Time progresses in a DDE model when the actors in the model execute and communicate. Actors in a DDE model communicate by sending messages through bounded, FIFO channels. Time in a DDE model is distributed and localized, and the actors of a DDE model each maintain their own local notion of the current time. Local time information is shared between two connected actors whenever a communication between said actors occurs. Conversely, communication between two connected actors can occur only when constraints on the relative local time information of the actors are adhered to.

The DDE domain is based on distributed discrete event processing and leverages a wealth of research devoted to this topic. Several tutorial publications on this topic exist in [19][27][43][67]. The DDE domain implements a specific variant of distributed discrete event systems (DDES) as expounded by Chandy and Misra [19]. While the DDE domain has similarities with DDES, the distributed discrete event domain serves as a framework for studying DDES with two special emphases. First we consider DDES from a dataflow perspective; we view DDE as an implementation of the Kahn dataflow model [45] with distributed time added on top. Second we study DDES not with the goal of improving execution speed (as has been the case traditionally). Instead we study DDES to learn its usefulness in modeling and designing systems that are timed and distributed.

12.2 Using DDE

The DDE domain is typed so that actors used in a model must be derived from ptolemy/actor/TypedAtomicActor. The DDE domain is designed to use both DDE specific actors as well as polymorphic actors. DDE specific actors take advantage of DDEActor and DDEIOPort which are designed to provide convenient support for specifying time in the production and consumption of tokens. The DDE domain also has special restrictions on how feedback is specified in models.

12.2.1 DDEActor

The DDE model of computation makes one very strong assumption about the execution of an actor: *all input ports of an actor operating in a DDE model must be regularly polled to determine which input channel has the oldest pending event*. Any actor that adheres to this assumption can operate in a DDE model. Thus, many polymorphic actors found in ptolemy/actor/[lib, gui] are suitable for operation in DDE models. For convenience, DDEActor was developed to simplify the construction of actors that have DDE semantics. DDEActor has three key methods as follows:

getNextToken(). This method polls each input port of an actor and returns the (non-Null) token that represents the oldest event. This method blocks accordingly as outlined in section 12.3.1 (Communicating Time).

getLastPort(). This method returns the input IOPort from which the last (non-Null) token was consumed. This method presumes that getNextToken() is being used for token consumption.

12.2.2 DDEIOPort

DDEIOPort extends TypedIOPort with parameters for specifying time stamp values of tokens that are being sent to neighboring actors. Since DDEIOPort extends TypedIOPort, use of DDEIOPorts will not violate the type resolution protocol. DDEIOPort is not necessary to facilitate communication between actors executing in a DDE model; standard TypedIOPorts are sufficient in most communication. DDEIOPorts become useful when the time stamp to be associated with an outgoing token is greater than the current time of the sending actor. Hence, DDEIOPorts are only useful in conjunction with delay actors (see “Enabling Communication: Advancing Time” on page 12-3, for a definition of delay actor). Most polymorphic actors available for Ptolemy II are not delay actors.

12.2.3 Feedback Topologies

In order to execute feedback topologies that will not deadlock, FeedBackDelay actors must be used. FeedBackDelay is found in the DDE kernel package. FeedBackDelay actors do not perform computation, but instead increment the time stamps of tokens that flow through them by a specified delay. The delay value of a FeedBackDelay actor must be chosen to be less than the delta time of the feedback cycle in which the FeedBackDelay actor is contained. Elaborate delay values can be specified by overriding the getDelay() method in subclasses of FeedBackDelay. An example of such can be found in ptolemy/domains/dde/demo/LocalZeno/ZenoDelay.java.

A difficulty found in feedback cycles occurs in the initialization of a model’s execution. In figure 12.1 we see that even if Actor B is a FeedBackDelay actor, the system will deadlock if the first event is created by *A* since *C* will block on an event from *B*. To alleviate this problem a special time stamp

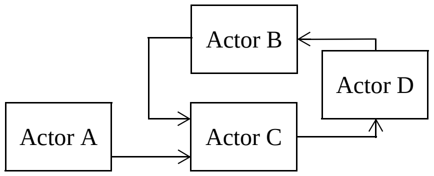


FIGURE 12.1. Initializing Feedback Topologies

value has been reserved: PrioritizedTimedQueue.IGNORE. When an actor encounters an event with a time stamp of IGNORE (an *ignore event*), the actor will ignore the event and the input channel it is associated with. The actor then considers the other input channels in determining the next available event. After a non-ignore event is encountered and consumed by the actor, all ignore events will be cleared from the receivers. If all of an actor's input channels contain ignore events, then the actor will clear all ignore events and then proceed with normal operation.

The initialize method of FeedBackDelay produces an ignore event. Thus, in figure 12.1, if *B* is a FeedBackDelay actor, the ignore event it produces will be sent to *C*'s upper input channel allowing *C* to consume the first event of *A*. The production of null tokens and feedback delays will then be sufficient to continue execution from that point on. Note that the production of an ignore event by a FeedBackDelay actor serves as a major distinction between it and all other actors. *If a delay is desired simply to represent the computational delay of a given model, a FeedBackDelay actor should not be used*.

The intricate operation of ignore events requires special consideration when determining the position of a FeedBackDelay actor in a feedback topology. A FeedBackDelay actor should be placed so that the ignore event it produces will be ignored in deference to the first real event that enters a feedback cycle. Thus, choosing actor *D* as a FeedBackDelay actor in figure 12.1 would not be useful given that the first real event entering the cycle is created by *A*.

12.3 Properties of the DDE domain

Operationally, the semantics of the DDE domain can be separated into two functionalities. The first functionality relates to how time advances during the communication of data and how communication proceeds via blocking reads and writes. The second functionality considers how a DDE model prevents deadlock due to local time dependencies. The technique for preventing deadlock involves the communication of *null messages* that consist solely of local time information.

12.3.1 Enabling Communication: Advancing Time

Communicating Tokens. A DDE model consists of a network of sequential actors that are connected via unidirectional, bounded, FIFO queues. Tokens are sent from a sending actor to a receiving actor by placing a token in the appropriate queue where the token is stored until the receiving actor consumes it. If a process attempts to read a token from a queue that is empty, then the process will block until a token becomes available on the channel. If a process attempts to write a token to a queue that is full, then the process will block until space becomes available for more tokens in that queue. Note that this blocking read/write paradigm is equivalent to the operational semantics found in non-timed process networks (PN) as implemented in Ptolemy II (see the PN Domain chapter).

If all processes in a DDE model simultaneously block, then the model deadlocks. Deadlock that is due to processes that are either waiting to read from an empty queue, *read blocks*, or waiting to write to a full queue, *write blocks,* then we say that the model has experienced *non-timed deadlock.* Non-timed deadlock is equivalent to the notion of deadlock found in bounded process networks scheduling problems as outlined by Parks [75]. If a non-timed deadlock is due to a model that consists solely of processes that are read blocked, then we say that a *real deadlock* has occurred and the model is terminated. If a non-timed deadlock is due to a model that consists of at least one process that is write blocked, then the capacity of the full queues are increased until deadlock no longer exists. Such deadlocks are called *artificial deadlock*, and the policy of increasing the capacity of full queues was shown by Parks to guarantee the execution of a model in bounded memory whenever possible.

Communicating Time. Each actor in a DDE model maintains a local notion of time. Any non-negative real number may serve as a valid value of time. As tokens are communicated between actors, time stamps are associated with each token. Whenever an actor consumes a token, the actor's *current time* is set to be equal to that of the consumed token's time stamp. The time stamp value applied to outgoing tokens of an actor is equivalent to that actor's *output time*. For actors that model a process in which there is delay between incoming time stamps and corresponding outgoing time stamps, then the output time is always greater than the current time; otherwise, the output time is equal to the current time. We refer to actors of the former case as *delay actors*.

For a given queue containing time stamped tokens, the time stamp of the first token currently contained by the queue is referred to as the *receiver time* of the queue. If a queue is empty, its receiver time is the value of the time stamp associated with the last token to flow through the queue, or 0.0 if no tokens have traveled through the queue. An actor may consume a token from an input queue given that the queue has a token available and the receiver time of the queue is less than the receiver times of all other input queues contained by the actor. If the queue with the smallest receiver time is empty, then the actor blocks until this queue receives a token, at which time the actor considers the updated receiver time in selecting a queue to read from. The *last time* of a queue is the time stamp of the last token to be placed in the queue. If no tokens have been placed in the queue, then the last time is 0.0

Figure 12.2 shows three actors, each with three input queues. Actor *A* has two tokens available on the top queue, no tokens available on the middle queue and one token available on the bottom queue. The receiver times of the top, middle and bottom queue are respectively, 17.0, 12.0 and 15.0. Since the queue with the minimum receiver time (the middle queue) is empty, *A* blocks on this queue before it proceeds. In the case of actor *B*, the minimum receiver time belongs to the bottom queue. Thus, *B* proceeds by consuming the token found on the bottom queue. After consuming this token, *B* compares all of its receiver times to determine which token it can consume next. Actor *C* is an example of an actor that contains multiple input queues with identical receiver times. To accommodate this situation, each actor assigns a unique priority to each input queue. An actor can consume a token from a queue if no other queue has a lower receiver time and if all queues that have an identical receiver time also have a lower priority.

Each receiver has a *completion time* that is set during the initialization of a model. The completion time of the receiver specifies the time after which the receiver will no longer operate. If the time stamp of the oldest token in a receiver exceeds the completion time, then that receiver will become *inactive*.

12.3.2 Maintaining Communication: Null Tokens

Deadlocks can occur in a DDE model in a form that differs from the deadlocks described in the previous section. This alternative form of deadlock occurs when an actor read blocks on an input port

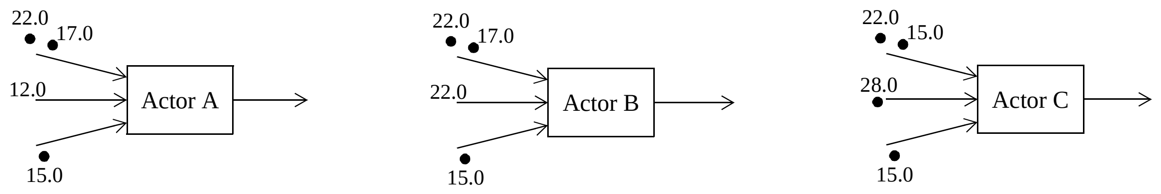


FIGURE 12.2. DDE actors and local time.

even though it contains other ports with tokens. The topology of a DDE model can lead to deadlock as read blocked actors wait on each other for time stamped tokens that will never appear. Figure 12.3 illustrates this problem. In this topology, consider a situation in which actor *A* only creates tokens on its lower output queue. This will lead to tokens being created on actor *C*'s output queue but no tokens will be created on *B*'s output queue (since *B* has no tokens to consume). This situation results in *D* read blocking indefinitely on its upper input queue even though it is clear that no tokens will ever flow through this queue. The result: *timed deadlock!* The situation shown in figure 12.3 is only one example of timed deadlock. In fact there are two types of timed deadlock: *feedforward* and *feedback*.

Figure 12.3 is an example of feedforward deadlock. Feedforward deadlock occurs when a set of connected actors are deadlocked such that all actors in the set are read blocked and at least one of the actors in the set is read blocked on an input queue that has a receiver time that is less than the local clock of the input queue's source actor. In the example shown above, the upper input queue of *B* has a receiver time of 0.0 even though the local clock of *A* has advanced to 8.0.

Feedback deadlock occurs when a set of cyclically connected actors are deadlocked such that all actors in the set are read blocked and at least one actor in the set, say actor *X*, is read blocked on an input queue that can read tokens which are directly or indirectly a result of output from that same actor (actor *X*). Figure 12.4 is an example of feedback timed deadlock. Note that *B* can not produce an output based on the consumption of the token timestamped at 5.0 because it must wait for a token on the upper input that depends on the output of *B*!

Preventing Feedforward Timed Deadlock. To address feedforward timed deadlock, *null tokens* are employed. A null token provides an actor with a means of communicating time advancement even though data (*real* tokens) are not being transmitted. Whenever an actor consumes a token, it places a null token on each of its output queues such that the time stamp of the null token is equal to the current time of the actor. Thus, if actor *A* of figure 12.3, produced a token on its lower output queue at time 5.0, it would also produce a null token on its upper output queue at time 5.0.

If an actor encounters a null token on one of its input queues, then the actor does the following. First it consumes the tokens of all other input queues it contains given that the other input queues have receiver times that are less than or equal to the time stamp of the null token. Next the actor removes the null token from the input queue and sets its current time to equal the time stamp of the null token. The actor then places null tokens time stamped to the current time on all output queues that have a last time

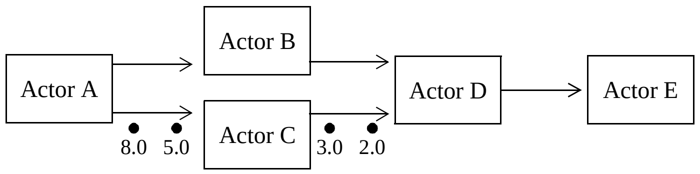


FIGURE 12.3. Timed deadlock (feedforward).

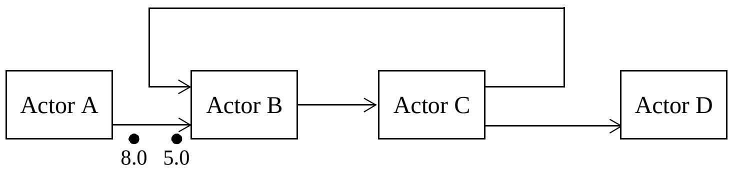


FIGURE 12.4. Timed Deadlock (Feedback)

that is less then the actor's current time. As an example, if *B* in figure 12.3 consumes a null token on its input with a time stamp of 5.0 then it would also produce a null token on its output with a time stamp of 5.0.

The result of using null tokens is that time information is evenly propagated through a model's topology. The beauty of null tokens is that they inform actors of inactivity in other components of a model without requiring centralized dissemination of this information. Given the use of null tokens, feedforward timed deadlock is prevented in the execution of DDE models. It is important to recognize that null tokens are used solely for the purpose of avoiding deadlocks. Null tokens do not represent any actual components of the physical system being modeled. Hence, we do not think of a null token as a real token. Furthermore, the production of a null token that is the direct result of the consumption of a null token is not considered computation from the standpoint of the system being modeled. The idea of null tokens was first espoused by Chandy and Misra [19].

Preventing Feedback Timed Deadlock. We address feedback timed deadlock as follows. All feedback loops are required to have a cumulative time stamp increment that is greater than zero. In other words, feedback loops are required to contain delay actors. Peacock, Wong and Manning [76] have shown that a necessary condition for feedback timed deadlock is that a feedback loop must contain no delay actors. The delay value (delay = output time - current time) of a delay actor must be chosen wisely; it must be less then the smallest delta time of all other actors contained in the same feedback loop. *Delta time* is the difference between the time stamps of a token that is consumed by an actor and the corresponding token that is produced in direct response. If a system being modeled has characteristics that prevent a fixed, positive lower bound on delta time from being specified, then our approach can not solve feedback timed deadlock. Such a situation is referred to as a *Zeno condition*. An application involving an approximated Zeno condition is discussed in section 12.5 below.

The DDE software architecture provides one delay actor for use in preventing feedback timed deadlock: *FeedBackDelay*. See "Feedback Topologies" on page 12-2 for further details about this actor.

12.3.3 Alternative Distributed Discrete Event Methods

The field of distributed discrete event simulation, also referred to as parallel discrete event simulation (PDES), has been an active area of research since the late 1970's [19][27][43][67][76]. Recently there has been a resurgence of activity [5][6][11]. This is due in part to the wide availability of distributed frameworks for hosting simulations and the application of parallel simulation techniques to non-research oriented domains. For example, several WWW search engines are based on network of workstation technology.

The field of distributed discrete event simulation can be cast into two camps that are distinguished by the blocking read approach taken by the actors. One camp was introduced by Chandy and Misra [19][27][67][76] and is known as *conservative* blocking. The second camp was introduced by David Jefferson through the Jet Propulsion Laboratory Time Warp system and is referred to as the *optimistic* approach [43][27]. In certain problems, the optimistic approach executes faster than the conservative approach, nevertheless, the gains in speed result in significant increases in program memory. The conservative approach does not perform faster than the optimistic approach but it executes efficiently for all classes of discrete event systems. Given the modeling semantics emphasis of Ptolemy II, performance (speed) is not considered a premium. Furthermore, Ptolemy II's embedded systems emphasis suggests that memory constraints are likely. For these reasons, the implementation found in the DDE domain follows the conservative approach.

12.4 The DDE Software Architecture

For a model to have DDE semantics, it must have a DDEDirector controlling it. This ensures that the receivers in the ports are DDEReceivers. Each actor in a DDE model is under the control of a DDEThread. DDEThreads contain a TimeKeeper that manages the local notion of time that is associated with the DDEThread's actor.

12.4.1 Local Time Management

The UML diagram of the local time management system of the DDE domain is shown in figure 12.5 and consists of PrioritizedTimedQueue, DDEReceiver, DDEThread and TimeKeeper. Since time is localized, the DDEDirector does not have a direct role in this process. Note that DDEReceiver is derived from PrioritizedTimedQueue. The primary purpose of PrioritizedTimedQueue is to keep track of a receiver's local time information. DDEReceiver adds blocking read/write functionality to PrioritizedTimedQueue.

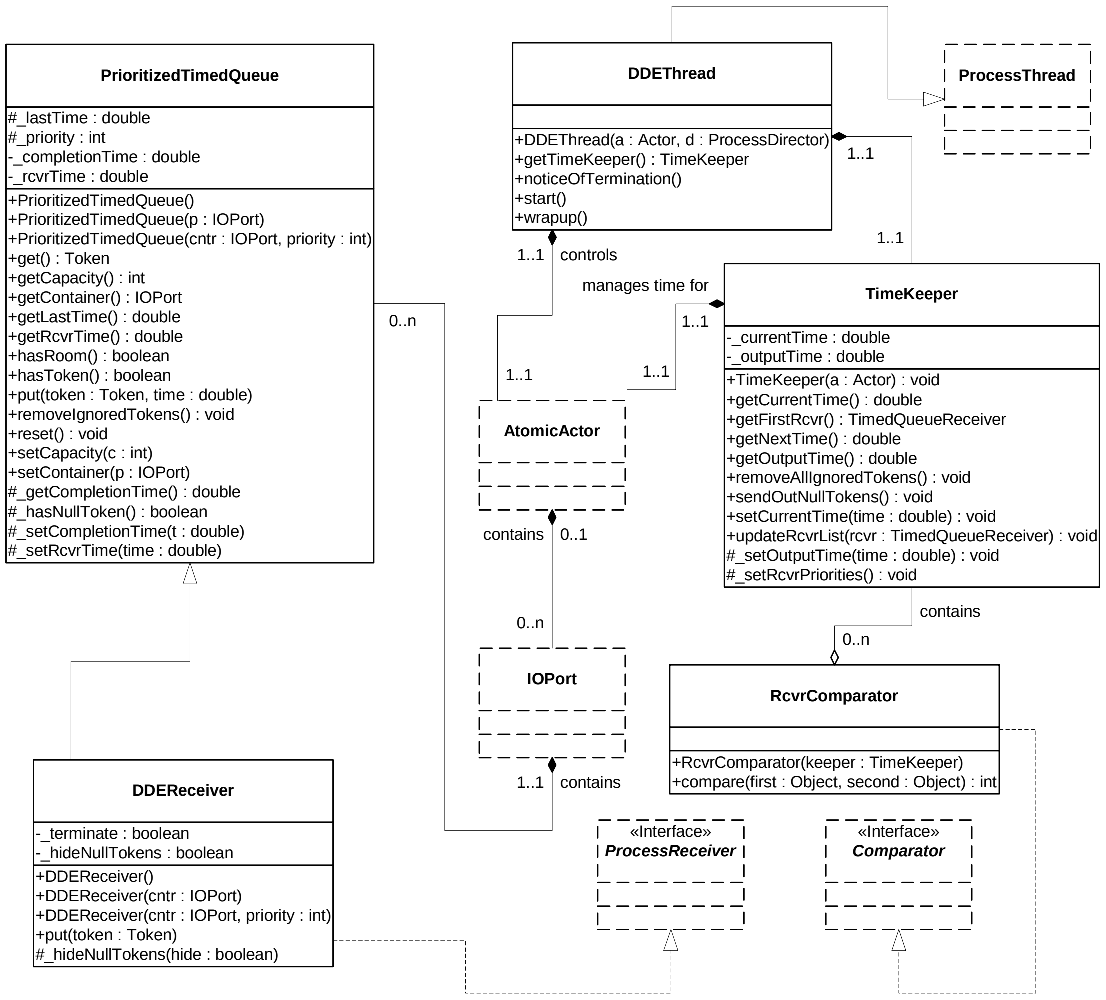

PrioritizedTimedQueue
#_lastTime : double
#_priority : int
-_completionTime : double
-_rcvrTime : double
+PrioritizedTimedQueue()
+PrioritizedTimedQueue(p : IOPort)
+PrioritizedTimedQueue(cntr : IOPort, priority : int)
+get() : Token
+getCapacity() : int
+getContainer() : IOPort
+getLastTime() : double
+getRcvrTime() : double
+hasRoom() : boolean
+hasToken() : boolean
+put(token : Token, time : double)
+removeIgnoredTokens() : void
+reset() : void
+setCapacity(c : int)
+setContainer(p : IOPort)
#_getCompletionTime() : double
#_hasNullToken() : boolean
#_setCompletionTime(t : double)
#_setRcvrTime(time : double)
DDEThread
+DDEThread(a : Actor, d : ProcessDirector)
+getTimeKeeper() : TimeKeeper
+noticeOfTermination()
+start()
+wrapup()
ProcessThread
1..1
1..1
1..1
controls
manages time for
1..1
0..n
1..1
1..1
TimeKeeper
-_currentTime : double
-_outputTime : double
+TimeKeeper(a : Actor) : void
+getCurrentTime() : double
+getFirstRcvr() : TimedQueueReceiver
+getNextTime() : double
+getOutputTime() : double
+removeAllIgnoredTokens() : void
+sendOutNullTokens() : void
+setCurrentTime(time : double) : void
+updateRcvrList(rcvr : TimedQueueReceiver) : void
#_setOutputTime(time : double) : void
#_setRcvrPriorities() : void
AtomicActor
contains
0..1
0..n
contains
0..n
IOPort
1..1
contains
RcvrComparator
+RcvrComparator(keeper : TimeKeeper)
+compare(first : Object, second : Object) : int
DDEReceiver
-_terminate : boolean
-_hideNullTokens : boolean
+DDEReceiver()
+DDEReceiver(cntr : IOPort)
+DDEReceiver(cntr : IOPort, priority : int)
+put(token : Token)
#_hideNullTokens(hide : boolean)
«Interface»
ProcessReceiver
«Interface»
Comparator

FIGURE 12.5. Key Classes for Locally Managing Time.

When a DDEDirector is initialized, it instantiates a DDEThread for each actor that the director manages. DDEThread is derived from ProcessThread. The ProcessThread class provides functionality that is common to all of the process domains (e.g., CSP, DDE and PN). The directors of all process domains (including DDE) assign a single actor to each ProcessThread. ProcessThreads take responsibility of their assigned actor's execution by invoking the iteration methods of the actor. The iteration methods are prefire(), fire() and postfire(); ProcessThreads also invoke wrapup() on the actors they control.

DDEThread extends the functionality of ProcessThread. Upon instantiation, a DDEThread creates a TimeKeeper object and assigns this object to the actor that it controls. The TimeKeeper gets access to each of the DDEReceivers that the actor contains. Each of the receivers can access the TimeKeeper and through the TimeKeeper the receivers can then determine their relative receiver times. With this information, the receivers are fully equipped to apply the appropriate blocking rules as they get and put time stamped tokens.

DDEReceivers use a dynamic approach to accessing the DDEThread and TimeKeeper. To ensure domain polymorphism, actors (DDE or otherwise) do not have static references to the TimeKeeper and DDEThread that they are controlled by. To ensure simplified mutability support, DDEReceivers do not have a static reference to TimeKeepers. Access to the local time management facilities is accomplished via the Java Thread.currentThread() method. Using this method, a DDEReceiver dynamically accesses the thread responsible for invoking it. Presumably the calling thread is a DDEThread and appropriate steps are taken if it is not. Once the DDEThread is accessed, the corresponding TimeKeeper can be accessed as well. The DDE domain uses this approach extensively in DDEReceiver.put(Token) and DDEReceiver.get().

DDEReceiver.put(Token) is derived from the Receiver interface and is accessible by all actors and domains. To facilitate local time advancement, DDEReceiver has a second put() method that has a time argument: DDEReceiver.put(Token, double). This second DDE-specific version of put() is taken advantage of without extensive code by using Thread.currentThread(). DDEReceiver.put() is shown below:

```
public void put(Token token)[1] {
   Thread thread = Thread.currentThread();
   double time = _lastTime;
   if( thread instanceof DDEThread ) {
      TimeKeeper timeKeeper = ((DDEThread)thread).getTimeKeeper();
      time = timeKeeper.getOutputTime();}
   put( token, time )[2];
}
```

Similar uses of Thread.currentThread() are found throughout DDEReceiver and DDEDirector. Note that while Thread.currentThread() can be quite advantageous, it means that if some methods are called by an inappropriate thread, problems may occur. Such an issue makes code testing difficult.

12.4.2 Detecting Deadlock

The other kernel classes of the DDE domain are shown in figure 12.6. The purpose of the DDEDirector is to detect and (if possible) resolve timed and/or non-timed deadlock of the model it controls.

1. DDEReceiver.put(Token) is equivalent to the put() signature of the ptolemy.actor.Receiver interface.
2. Polymorphic actors need not be aware of DDE-specific code such as DDEReceiver.put(Token, Double).

Whenever a receiver blocks, it informs the director. The director keeps track of the number of active processes, and the number of processes that are either blocked on a read or write. Artificial deadlocks are resolved by increasing the queue capacity of write-blocked receivers.

Note the distinction between internal and external read blocks in DDEDirector's package friendly methods. The current release of DDE assumes that actors that execute according to a DDE model of computation are atomic rather than composite. In a future Ptolemy II release, composite actors will be facilitated in the DDE domain. At that time, it will be important to distinguish internal and external read blocks. Until then, only internal read blocks are in use.

12.4.3 Ending Execution

Execution of a model ends if either an unresolvable deadlock occurs, the director's completion time is exceeded by all of the actors it manages, or early termination is requested (e.g., by a user interface button). The director's completion time is set via the public *stopTime* parameter of DDEDirector. The completion time is passed on to each DDEReceiver. If a receiver's receiver time exceeds the completion time, then the receiver becomes inactive. If all receivers of an actor become inactive and the actor is not a source actor, then the actor will end execution and its wrapup() method will be called. In such a scenario, the actor is said to have terminated *normally.*

Early terminations and unresolvable deadlocks share a common mechanism for ending execution. Each DDEReceiver has a boolean `_terminate` flag. If the flag is set to true, then the receiver will throw a `TerminateProcessException` the next time any of its methods are invoked. TerminateProcessExceptions are part of the `ptolemy/actor/process` package and ProcessThreads know to end

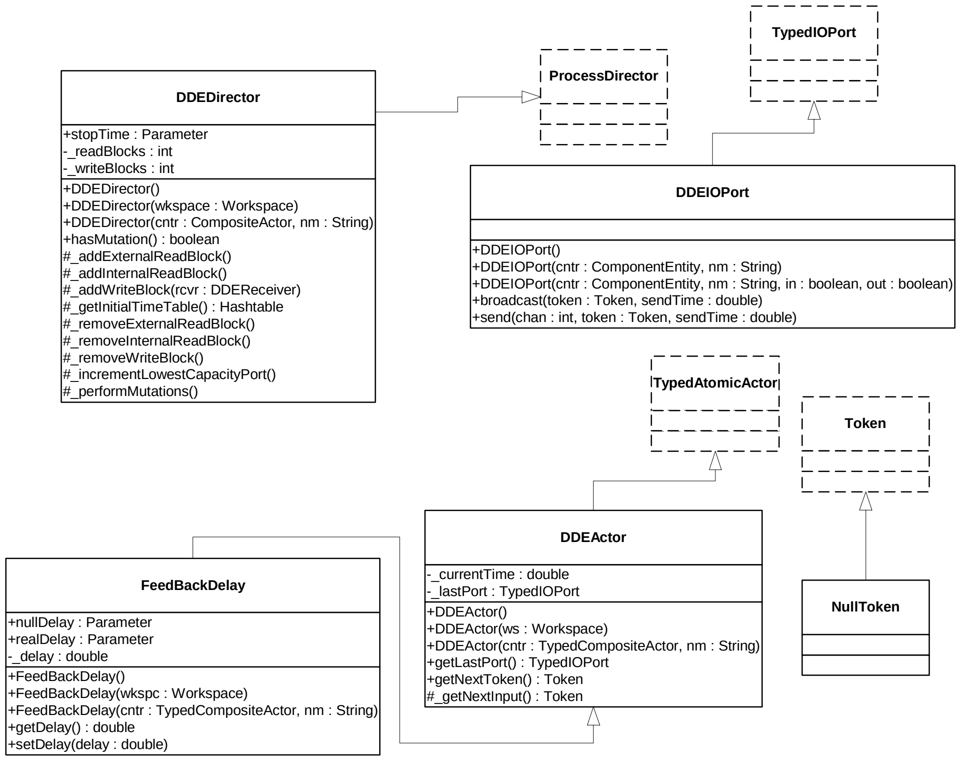

TypedIOPort
ProcessDirector
DDEDirector
+stopTime : Parameter
-_readBlocks : int
-_writeBlocks : int
+DDEDirector()
+DDEDirector(wkspace : Workspace)
+DDEDirector(cntr : CompositeActor, nm : String)
+hasMutation() : boolean
#_addExternalReadBlock()
#_addInternalReadBlock()
#_addWriteBlock(rcvr : DDEReceiver)
#_getInitialTimeTable() : Hashtable
#_removeExternalReadBlock()
#_removeInternalReadBlock()
#_removeWriteBlock()
#_incrementLowestCapacityPort()
#_performMutations()
DDEIOPort
+DDEIOPort()
+DDEIOPort(cntr : ComponentEntity, nm : String)
+DDEIOPort(cntr : ComponentEntity, nm : String, in : boolean, out : boolean)
+broadcast(token : Token, sendTime : double)
+send(chan : int, token : Token, sendTime : double)
TypedAtomicActor
Token
DDEActor
-_currentTime : double
-_lastPort : TypedIOPort
+DDEActor()
+DDEActor(ws : Workspace)
+DDEActor(cntr : TypedCompositeActor, nm : String)
+getLastPort() : TypedIOPort
+getNextToken() : Token
#_getNextInput() : Token
FeedBackDelay
+nullDelay : Parameter
+realDelay : Parameter
-_delay : double
+FeedBackDelay()
+FeedBackDelay(wkspc : Workspace)
+FeedBackDelay(cntr : TypedCompositeActor, nm : String)
+getDelay() : double
+setDelay(delay : double)
NullToken

FIGURE 12.6. Additional Classes in the DDE Kernel.

an actor’s execution if this exception is caught. In the case of unresolvable deadlock, the `_terminate` flag of all blocked receivers is set to true. The receivers are then awakened from blocking and they each throw the exception.

12.5 Example DDE Applications

To illustrate distributed discrete event execution, we have developed an applet that features a feedback topology and incorporates polymorphic as well as DDE specific actors. The model, shown in figure 12.7, consists of a single source actor (ptolemy/actor/lib/Clock) and an upper and lower branch of four actors each. The upper and lower branches have identical topologies and are fed an identical stream of tokens from the Clock source with the exception that in the lower branch ZenoDelay replaces FeedBackDelay.

As with all feedback topologies in DDE (and DE) models, a positive time delay is necessary in feedback loops to prevent deadlock. If the time delay of a given loop is lower bounded by zero but can not be guaranteed to be greater than a fixed positive value, then a Zeno condition occurs in which time will not advance beyond a certain point even though the actors of the feedback loop continue to execute without deadlocking. ZenoDelay extends FeedBackDelay and is designed so that a Zeno condition will be encountered. When execution of the model begins, both FeedBackDelay and ZenoDelay are used to feed back null tokens into Wire so that the model does not deadlock. After local time exceeds a preset value, ZenoDelay reduces its delay so that the lower branch approximates a Zeno condition.

In centralized discrete event systems, Zeno conditions prevent progress in the entire model. This is true because the feedback cycle experiencing the Zeno condition prevents time from advancing in the entire model. In contrast, distributed discrete event systems localize Zeno conditions as much as is possible based on the topology of the system. Thus, a Zeno condition can exist in the lower branch and the upper branch will continue its execution unimpeded. Localizing Zeno conditions can be useful in large scale modeling in which a Zeno condition may not be discovered until a great deal of time has been invested in execution of the model. In such situations, partial data collection may proceed prior to correction of the delay error that resulted in the Zeno condition.

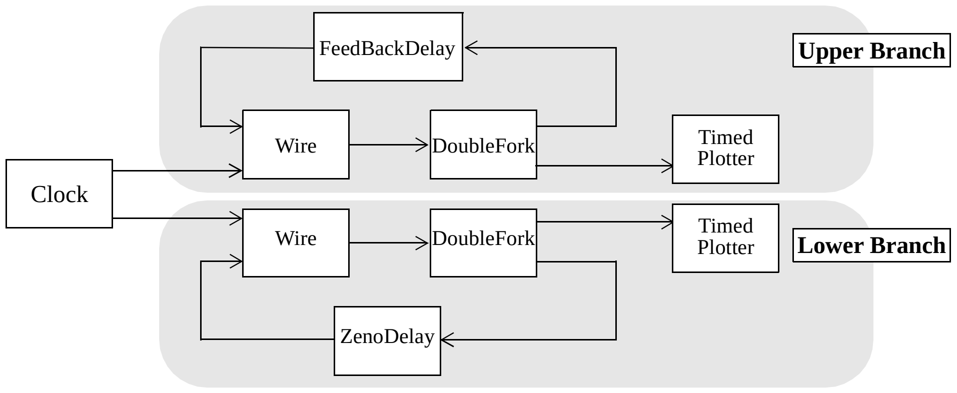


FIGURE 12.7. Localized Zeno condition topology.

13

PN Domain

Author: *Mudit Goel*
Contributor: *Steve Neuendorffer*

13.1 Introduction

The process networks (PN) domain in Ptolemy II models a system as a network of processes that communicate with each other by passing messages through unidirectional first-in-first-out (FIFO) channels. A process blocks when trying to read from an empty channel until a message becomes available on it. This model of computation is deterministic in the sense that the sequence of values communicated on the channels is completely determined by the model. Consequently, a process network can be evaluated using a complete parallel or sequential schedule and every schedule in between, always yielding the same output results for a given input sequence.

PN is a natural model for describing signal processing systems where infinite streams of data samples are incrementally transformed by a collection of processes executing in parallel. Embedded signal processing systems are good examples of such systems. They are typically designed to operate indefinitely with limited resources. This behavior is naturally described as a process network that runs forever but with bounded buffering on the communication channels whenever possible.[1]

PN can also be used to model concurrency in the various hardware components of an embedded system. The original process networks model of computation can model the functional behavior of these systems and test them for their functional correctness, but it cannot directly model their real-time behavior. To address the involvement of time, we have extended the PN model such that it can include the notion of time.

Some systems might display adaptive behavior like migrating code, agents, and arrivals and departures of processes. To support this adaptive behavior, we provide a mutation mechanism that supports addition, deletion, and changing of processes and channels. With untimed PN, this might display non-

1. In general, bounded buffers cannot be ensured for an arbitrary process network. An important part of the design of a process network concerns showing that the buffers are, in fact, bounded. Synchronous dataflow models are an important type of process network which always have bounded buffers.

determinism, while with timed-PN, it becomes deterministic.

The PN model of computation is a superset of the synchronous dataflow model of computation (see the SDF Domain chapter). Consequently, any SDF actor can be used within the PN domain. Similarly any domain-polymorphic actor can be used in the PN domain. However, the execution of the model is very different from SDF, since a separate process is created for each of actor. These processes are implemented as Java threads [72].

13.2 Using PN

There are two issues to be dealt with in the PN domain:

- Deadlock in feedback loops
- Designing actors

13.2.1 Deadlock in Feedback Loops

Feedback loops must be handled in much the same way as in the SDF actor. One of the actors in the feedback loop must create a number of tokens in its feedback loop in order to break the data dependency. Just like in the SDF domain, the SampleDelay actor can be used for this purpose. Remember, however that the PN domain does not (and cannot) statically analyze the model to determine the size of the delay necessary in the feedback loop. It is up to the designer of the model to specify the correct amount of delay.

13.2.2 Designing Actors

Because of the way the PN domain is implemented, it is not possible for an actor to check if data is present on an input port. The hasToken() method always returns true indicating that a token is present, and if a token is not actually present, then the get() method will block until one becomes available. This allows models to execute deterministically. However, actors that take inputs from more than one input can often be difficult to write. The common way of creating such an actor is similar to the way the Select actor works. Another input is read first, and the data from that port determines which input port to read from.

13.3 Properties of the PN domain

Two important properties of the PN domain implemented in Ptolemy II are that processes communicate asynchronously (by ordered queues) and that the memory used in the communication is bounded. The PN domain in Ptolemy II can be used with or without a notion of time.

13.3.1 Asynchronous Communication

Kahn and MacQueen [44][45] describe a model of computation where processes are connected by communication channels to form a network. Processes produce data elements or *tokens* and send them along a unidirectional communication channel where they are stored in a FIFO queue until the destination process consumes them. This is a form of asynchronous communication between processes. Communication channels are the *only* method processes may use to exchange information. A set of processes that communicate through a network of FIFO queues defines a *program*.

Kahn and MacQueen require that execution of a process be suspended when it attempts to get data from an empty input channel (*blocking reads*). Hence, a process may not poll a channel for presence or absence of data. At any given point, a process is either doing some computation (enabled) or it is blocked waiting for data (*read blocked*) on exactly one of its input channels; it cannot wait for data from more than one channel simultaneously. Systems that obey this model are determinate; the history of tokens produced on the communication channels does not depend on the execution order. Therefore, the results produced by executing a program are not affected by the scheduling of the various processes.

In case all the processes in a model are blocked while trying to read from some channel, then we have a *real deadlock*; none of the processes can proceed. Real deadlock is a program state that happens irrespective of the schedule chosen to schedule the processes in a model. This characteristic is guaranteed by the determinacy property of process networks.

13.3.2 Bounded Memory Execution

The high level of concurrency in process networks makes it an ideal match for embedded system software and for modeling hardware implementations. A characteristic of these embedded applications and hardware processes, is that they are intended to run indefinitely with a limited amount of memory. One problem with directly implementing the Kahn-MacQueen semantics is that bounded memory execution of a process network is not guaranteed, even if it is possible. Hence, bounded memory execution of process networks becomes crucial for its usefulness for hardware and embedded software.

Parks [75] addresses this aspect of process networks and provides an algorithm to make a process network application execute in bounded memory whenever possible. He provides an implementation of the Kahn-MacQueen semantics using *blocking writes* that assigns a fixed capacity to each FIFO channel and forces processes to block temporarily if a channel is full. Thus a process has now three states: *running* (*executing*), *read blocked*, or *write blocked* and a process may not poll a channel for either data or room.

In addition to the real deadlock described above, the introduction of a blocking write operation can cause an *artificial deadlock* of the process network. In this situation, all the processes in a model are blocked and at least one process is blocked on a write. However unlike after real deadlock, a program can continue after artificial deadlock by increasing the capacity of the channels on which processes are write blocked. In particular, Parks chooses to increase only the capacity of the channel with the smallest capacity among the channels on which processes are write blocked. This algorithm minimizes overall required memory in the channels and is used in the PN domain to handled artificial deadlock.

13.3.3 Time

In real-time systems and embedded applications, the real time behavior of a system is as important as the functional correctness. Process networks can be used to describe the functional properties of a system, but cannot describe temporal properties since the basic model lacks the notion of time. One solution is to use some other timed model of computation, such as DE, for describing temporal properties. Another solution is to extend the process networks model of computation with a notion time, as we have done in Ptolemy II. This extension is based on the Pamela model [30], which was originally developed for modeling the performance of parallel systems using Dykstra’s semaphores.

In the timed PN domain, time is global. All processes in a model share the same time, which is referred to as the *current time* or *model time*. A process can explicitly wait for time to advance, by *delaying* itself for some fixed amount of time. After being suspended for the specified amount of time, the process wakes up and continues to execute. If the process delays itself for zero time then the process simply continues to execute.

In the timed PN domain, time changes only at specific moments and never during the execution of a process. The time observed by a process can only advance when it is in one of the following two states:

1. The process is delayed and is explicitly waiting for time to advance (*delay block*).
2. The process is waiting for data to arrive on one of its input channels (*read block*).

When all the processes in a program are in one of these two states, then the program is in a state of *timed deadlock*. The fact that at least one process is delayed, distinguishes timed deadlock from other deadlocks. When timed deadlock is detected, the current time is advanced until at least one process can awaken from a delay block and the model continues executing.

13.3.4 Mutations

The PN domain tolerates mutations, which are run-time changes in the model structure. Normally, mutations are realized as *change requests* queued with the model. In PN there is no determinate point where mutations can occur other than a real deadlock. However, being able to perform mutations at this point is unlikely as a real deadlock might never occur. For example, a model with even one non-terminating source never experiences a real deadlock. Therefore mutations cannot be performed at determinate points since the processes in the network are not synchronized. Executing mutations at arbitrary times introduces non-determinism in PN, since the state of the processes is unknown.

In timed PN, however, the presence of timed deadlock provides a regular point at which the state of execution can be determined. This means that mutations in timed PN can be made deterministically. Implementation details are presented later in section 13.4.

13.4 The PN Software Architecture

The PN domain kernel is realized in package ptolemy.domains.pn.kernel. The structure diagram of the package is shown in figure .13.1.

13.4.1 BasePNDirector

This class extends the ProcessDirector base class to add Kahn process networks (PN) semantics. This director does not support mutations or a notion of time. It provides only a mechanism to perform blocking reads and writes using bounded memory execution whenever possible.

This director is capable of handling both real and artificial deadlocks. Artificial deadlock is resolved as soon as it arises using Parks's algorithm as explained in section 13.3.2. Real deadlock, however, cannot be handled locally and must rely on the external environment to provide more data for execution to continue.

13.4.2 PNDirector

PNDirector extends the BasePNDirector to handle mutations locally. This is only an optimization, since it allows a mutation to execute faster than it would otherwise, and does not add any interesting expressive capability to the model. Most importantly, the mutation is non-deterministic and can happen at any point during the execution of the model.

13.4.3 TimedPNDirector

TimedPNDirector extends the BasePNDirector to introduces a notion of global time to the model . It also provides for deterministic execution of mutations. Mutations are performed at the earliest timed-deadlock that occurs after they are queued. Since occurrence of timed-deadlock is deterministic, performing mutations at this point makes mutations deterministic.

13.4.4 PNQueueReceiver

The PNQueueReceiver implements the ProcessReceiver interface and contains a FIFO queue to represents a process network communications channel. These receivers are also responsible for implementing the blocking reads and blocking writes through the get() and put() methods.

When the get() method is called, the receiver first checks if a FIFO queue has any tokens. If not, then it reports to the director that the reading thread is blocked waiting for data. It also sets an internal flag to indicate that a thread is read blocked. Then the reading thread is suspended until some other thread puts a token into the FIFO queue. At this point, the flag of the receiver is reset to false, the director is notified that a process has unblocked, the reading process retrieves the first token from the FIFO queue and execution continues.

The put() method of the receiver works similarly by first checking whether the FIFO queue is full

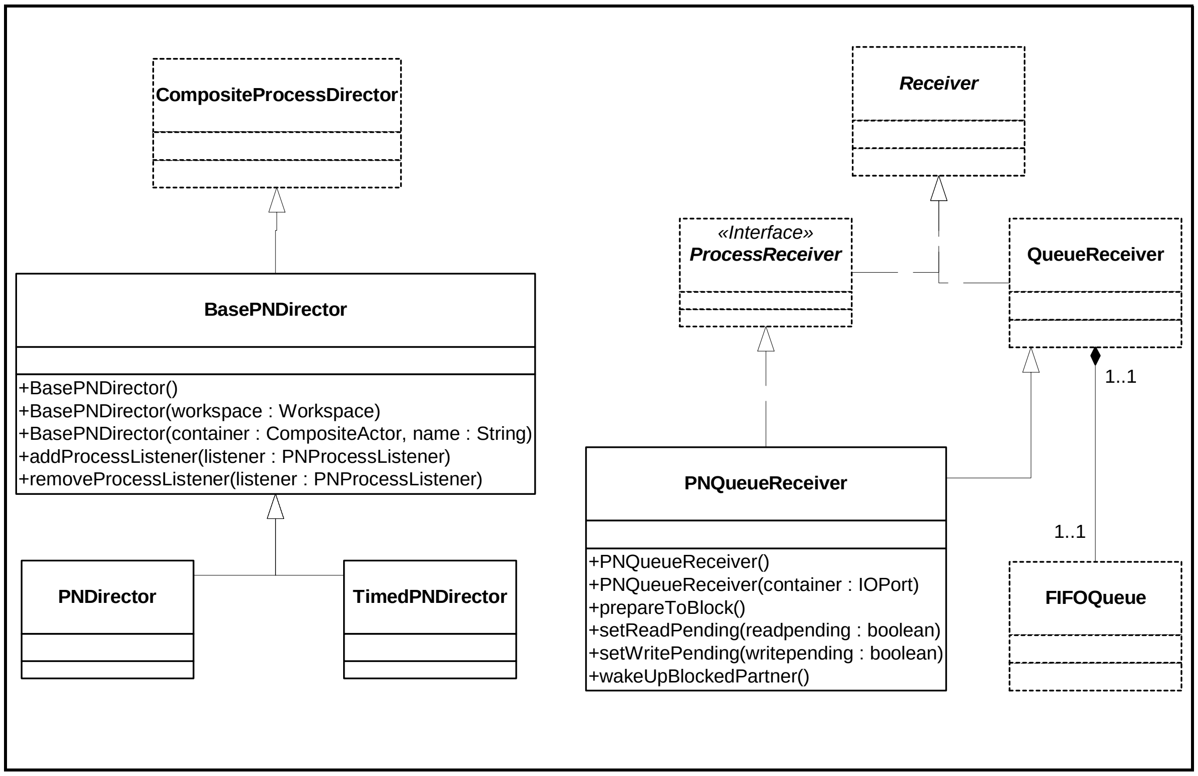

CompositeProcessDirector
BasePNDirector
+BasePNDirector()
+BasePNDirector(workspace : Workspace)
+BasePNDirector(container : CompositeActor, name : String)
+addProcessListener(listener : PNProcessListener)
+removeProcessListener(listener : PNProcessListener)
PNDirector
TimedPNDirector
Receiver
«Interface»
ProcessReceiver
QueueReceiver
1..1
PNQueueReceiver
+PNQueueReceiver()
+PNQueueReceiver(container : IOPort)
+prepareToBlock()
+setReadPending(readpending : boolean)
+setWritePending(writepending : boolean)
+wakeUpBlockedPartner()
1..1
FIFOQueue

FIGURE 13.1. Static structure of the PN kernel.

to capacity. If so, it reports to the director that the writing thread is blocked waiting for space in the queue. It also sets an internal flag to indicate that a thread is write blocked. The writing thread blocks until some other thread gets a token from the FIFO queue, or the size of the queue is increased by the director because the network reached an artificial deadlock. In either case, the director is notified that a writing process unblocks and the internal flag is reset. The writing thread is reawakened and its token is placed into the receiver.

13.4.5 Handling Deadlock

Every time an actor in PN blocks, the count of blocked actors is increased. If the total number of actors blocked or paused equals the total number of actors active in the simulation, a deadlock is detected. On detection of a deadlock, if one or more actors are blocked on a write, then this is an artificial deadlock. The channel with the smallest capacity among all the channels with actors blocked on a write is chosen and its capacity is incremented by 1. This implements the bounded memory execution as suggested by [75]. If a real deadlock is detected, then the fire() method of the director returns, allowing a containing model to present more data to the inputs of the process network.

13.4.6 Finite Iterations

An important aspect of Ptolemy II is that the firing of an actor, or an entire model is guaranteed to complete. In the process domains the end of a firing occurs when deadlock is reached. The deadlock can be real or timed deadlock. However, in a process network real deadlock may never actually happen. In this case, in order to manually stop execution or to execute mutations there needs to be a way to halt all the executing threads in the network. This is handled by the stopFire() method of the executable interface. The process director implements this method to set a flag in each process which causes the process to pause. Note that as with most domains, it is not possible to simply call the wrapup() method of the process director, since the fire method has not yet returned.

References

[1] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] G. A. Agha, “Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems,” in *Formal Methods for Open Object-based Distributed Systems*, IFIP Transactions, E. Najm and J.-B. Stefani, Eds., Chapman & Hall, 1997.

[3] 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.

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

[5] R. L. Bagrodia, “Parallel Languages for Discrete Event Simulation Models,” *IEEE Computational Science & Engineering*, vol. 5, no. 2, April-June 1998, pp 27-38.

[6] 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.

[7] 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, Sprinter-Verlag, 1994.

[8] 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.

[9] 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.

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

[11] 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.

[12] Randy Brown, “CalendarQueue: A Fast Priority Queue Implementation for The Simulation Event Set Problem”, Communications of the ACM, October 1998, Volume 31, Number 10.

[13] V. Bryant, “Metric Spaces,” Cambridge University Press, 1985.

[14] 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).

[15] A. Burns, *Programming in OCCAM 2*, Addison-Wesley, 1988.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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/)

[23] 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)

[24] P. A. Fishwick, *Simulation Model Design and Execution: Building Digital Worlds*, Prentice Hall, 1995.

[25] 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.

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

[27] R. M. Fujimoto, “Parallel Discrete Event Simulation,” *Communications of the ACM*, vol. 33, no. 10, October 1990, pp 30-53.

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

[29] C. W. Gear, “Numerical Initial Value Problems in Ordinary Differential Equations,” Prentice Hall Inc. 1971.

[30] 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.

[31] 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)

[32] 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)

[33] M. Grand, *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML*, John Wiley & Sons, 1998

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

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

[36] 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.

[37] 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.

[38] M. G. Hinchey and S. A. Jarvis, *Concurrent Systems: Formal Developments in CSP*, McGraw-Hill, 1995.

[39] 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.

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

[41] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[42] 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

[43] D. Jefferson, Brian Beckman, et al, "Distributed Simulation and the Time Warp Operating System," UCLA Computer Science Department: 870042, 1987.

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

[45] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[46] P. Laramie, R.S. Stevens, and M.Wan, "Kahn process networks in Java," ee290n class project report, Univ. of California at Berkeley, 1996.

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

[48] 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/)

[49] Edward A. Lee, "What's Ahead for Embedded Software?," IEEE Computer, September 2000, pp. 18-26.

[50] 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 (http://www.baltzer.nl.ansoft/). Also UCB/ERL Memorandum M98/7, March 4th 1998.(http://ptolemy.eecs.berkeley.edu/publications/papers/98/realtime)

[51] Edward A. Lee and Yuhong Xiong, "System-Level Types for Component-Based Design,"Technical Memorandum UCB/ERL M00/8, Electronics Research Lab, University o f California, Berkeley, CA 94720, USA, February 29, 2000.

[52] 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/)

[53] E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing,” *IEEE Trans. on Computers*, January, 1987.

[54] 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)

[55] 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/)

[56] M. A. Lemkin, *Micro Accelerometer Design with Digital Feedback Control*, Ph.D. dissertation, University of California, Berkeley, Fall 1997.

[57] 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/)

[58] Jie Liu and Edward 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

[59] 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.

[60] D. C. Luckham and J. Vera, “An Event-Based Architecture Definition Language,” *IEEE Transactions on Software Engineering*, 21(9), pp. 717-734, September, 1995.

[61] 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.

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

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

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

[65] R. Milner, *“A Calculus of Communicating Systems”*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

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

[67] J. Misra, “Distributed Discrete-Event Simulation,” *Computing Surveys*, vol. 18, no. 1, March 1986, pp. 39-65.

[68] 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/)

[69] 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.

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

[71] A. R. Newton and A. L. Sangiovanni-Vincentelli, “Relaxation-Based Electrical Simulation,” *IEEE Tr. on Electronic Devices*, Vol. ed-30, No. 9, Sept. 1983.

[72] S. Oaks and H. Wong, *Java Threads,* O’Reilly, 1997.

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

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

[75] 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/)

[76] 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.

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

[78] 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/)

[79] 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.

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

[81] R. C. Rosenberg and D.C. Karnopp, *Introduction to Physical System Dynamics*, McGraw-Hill, NY, 1983.

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

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

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

[85] S. Saracco, J. R. W. Smith, and R. Reed, *Telecommunications Systems Engineering Using SDL,* North-Holland - Elsevier, 1989.

[86] 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/)

[87] 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).

[88] Jeff Tsay, Christopher Hylands and Edward Lee, "A Code Generation Framework for Java Component-Based Designs," *CASES '00*, November 17-19, 2000, San Jose, CA.

[89] World Wide Web Consortium, *XML 1.0 Recommendation*, October 2000, http://www.w3.org/XML/

[90] World Wide Web Consortium, *Overview of SGML Resources*, August 2000, http://www.w3.org/MarkUp/SGML/

[91] Yuhong Xiong and Edward 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.

Glossary

abstract syntax	A conceptual data organization. cf. *concrete syntax.*
action methods	The methods initialize(), prefire(), fire(), postfire(), and wrapup() in the Executable interface.
actor	An executable entity. This was called a *block* in Ptolemy Classic.
anytype	The Ptolemy Classic name for *undeclared type*.
applet	A Java program that is downloaded from a web server by a browser and executed in the client's computer (usually within a plug-in for the browser). An applet has restricted access to local resources for security reasons.
application	A Java program that is executed as an ordinary program on a host computer. Unlike an applet, an application can have full access to local resources such as the file system.
atomic actor	A primitive actor. That is, one that is not a composite actor. This was called a *star* in Ptolemy Classic.
attribute	A named property associated with a named object in Ptolemy II. Also, in XML, a modifier to an element.
block	The Ptolemy Classic name for an *actor.*
browser	A program that renders HTML and accesses the worldwide web using the HTTP protocol.
channel	A path from an output port to an input port (via relations) that can transport a single stream of tokens.
clustered graph	A graph with hierarchy. Ptolemy II topologies are clustered graphs.
code generation	Translation of a model into efficient, standalone software for execution autonomously from the design environment. Code generation was a major emphasis of Ptolemy Classic. We are developing a code generation facility for Ptolemy II, but it is not included in the current release. For more information, see [87].
composite actor	An actor that is internally composed of other actors and relations. This was called a *galaxy* in Ptolemy Classic.
concrete syntax	A persistent representation of a data organization. cf. *abstract syntax.*
connection	A path from one port to another via relations and possibly transparent ports. A connection consists of one or more *relations* and two or more *links*.
container	An object that logically owns another. A Ptolemy II object can have at most one container.
dangling relation	A relation with only input ports or only output ports linked to it.
data polymorphism	Ability to operate with more than one token type.

deep traversals	Traversals of a clustered graph that see through transparent cluster boundaries (transparent composite entities and ports).
disconnected port	A port with no relation linked to it.
director	An object that controls the execution of a model or an opaque composite entity according to some *model of computation*.
domain	An implementation of a model of computation in Ptolemy II and Ptolemy Classic.
domain polymorphism	Ability to operate under more than one model of computation.
element	In XML, a portion of a document consisting of a begin tag, a body, and an end tag.
entity	A node in a Ptolemy II clustered graph. Also, in XML, a named text segment.
execution	One invocation of initialize(), followed by any number of *iterations*, followed by one invocation of wrapup().
executive director	From the perspective of an actor inside an opaque composite actor, the director of the container of the opaque composite actor.
galaxy	The Ptolemy Classic name for a *composite actor.*
immutable property	A property of an object that is set up when the object is constructed and that cannot be changed during the lifetime of the object.
iteration	One invocation of prefire(), followed by any number of invocations of fire(), followed by one invocation of postfire().
link	An association between a port and a relation.
manager	The top-level controller for the execution of a model.
model	A complete Ptolemy II application. This was called a *universe* in Ptolemy Classic.
model of computation	The rules that govern the interaction, communication, and control flow of a set of components.
MoML	Modeling markup language, an XML dialect for specifying component-based designs such those in Ptolemy II.
multiport	A port that can send or receive tokens over more than one channel.
opaque	For a composite entity or a port, an attribute that indicates that the inside should not be visible from the outside. That is, deep traversals of the topology do not see through an opaque boundary.
opaque composite actor	A composite actor with a local director. Such an actor appears to the outside domain to be atomic, but internally is composed of an interconnection of other actors. This was called a *wormhole* in Ptolemy Classic.
package	A collection of classes that forms a logical unit and occupies one directory in the source code tree.
parameter	An *attribute* with a value. This was called a *state* in Ptolemy Classic.
particle	The Ptolemy Classic name for a *token*.
port	A named interface of an entity to which connections be made.
Ptolemy Classic	A C++ software system for construction of concurrent models and

	implementation through code generation.
Ptolemy II	A Java software system for construction and execution of concurrent models.
Ptolemy Project	A research project at Berkeley that investigates modeling, simulation, and design of concurrent, networked, embedded systems.
relation	An object representing an interconnection between entities.
resolved type	A type for a port that is consistent with the type constraints of the actor and any port it is connected to. It is the result of type resolution.
servlet	A Java program that is executed on a web server and that produces results viewed remotely on a web browser.
star	The Ptolemy Classic name for an *atomic actor.*
state	The Ptolemy Classic name for a *parameter.*
subpackage	A package that is logically related to a parent package and occupies a subdirectory within the parent package in the source code tree.
tag	In XML, a portion of markup having the syntax <*tagname*>.
token	A unit of data that is communicated by actors. This was called a *parti-cle* in Ptolemy Classic.
topology	The structure of interconnections between entities (via relations) in a Ptolemy II model. See *clustered graph*.
transparent	For an entity or port, not opaque. That is, deep traversals of the topol-ogy pass right through its boundaries.
transparent composite actor	A composite actor with no local director.
transparent port	The port of a transparent composite entity. Deep traversals of the topology see right through such a port.
type constraints	The declared constraints on the token types that an actor can work with.
type resolution	The process of reconciling type constraints prior to running a model.
undeclared type	Capable of working with any type of token. This was called *anytype* in Ptolemy Classic.
universe	The Ptolemy Classic name for a *model*.
width of a port	The sum of the widths of the relations linked to it, or zero if there are none.
width of a relation	The number of channels supported by the relation.
wormhole	The Ptolemy Classic name for an *opaque composite actor.*