This is a chapter from the book

System Design, Modeling, and Simulation using Ptolemy II

First Edition, Version 1.0

Please cite this book as:

Claudius Ptolemaeus, Editor,
System Design, Modeling, and Simulation using Ptolemy II, Ptolemy.org, 2014.
http://ptolemy.org/books/Systems.

12 Software Architecture

Christopher Brooks, Joseph Buck, Elaine Cheong, John S. Davis II, Patricia Derler, Thomas Huining Feng, Geroncio Galicia, Mudit Goel, Soonhoi Ha, Edward A. Lee, Jie Liu, Xiaojun Liu, David Messerschmitt, Lukito Muliadi, Stephen Neuendorffer, John Reekie, Bert Rodiers, Neil Smyth, Yuhong Xiong, Haiyang Zheng

Contents

This chapter provides an overview of the software architecture of Ptolemy II to enable the reader to create custom software extensions, such as new directors or custom actors. Additional detail can be found in Brooks et al. (2004) and in the Ptolemy source code, which is well documented and designed for easy readability. This chapter assumes some familiarity with Java, the language in which most of Ptolemy II is written, and with UML class diagrams, which are used to depict key properties of the architecture.

12.1 Package Structure

Ptolemy II is a collection of Java classes organized into multiple packages. The package structure, shown in Figure 12.1, is carefully designed to ensure separability of the pieces.

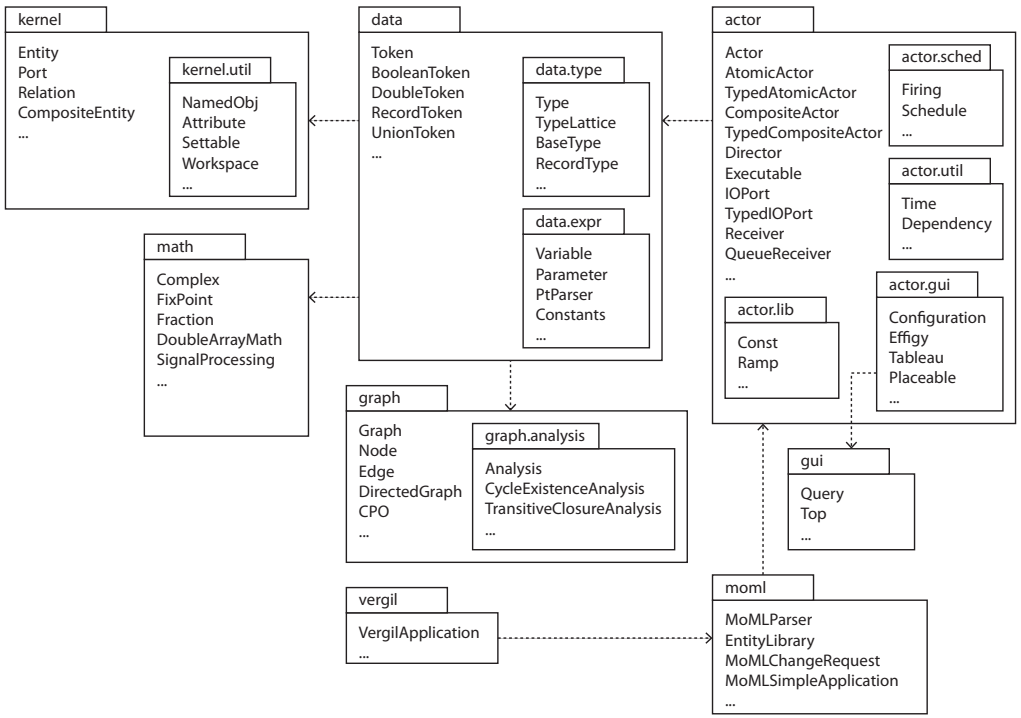

Figure 12.1: Key packages and their classes in Ptolemy II.

The kernel package. The kernel package and its subpackages are the heart of Ptolemy II. They contain the class definitions that serve as base classes for every part of a Ptolemy model. The kernel package itself is relatively small, and its design is described in Section 12.2. This package defines the structure of models; in particular, it specifies the hierarchy relationships between, for example, components and domains, and how the components of a model are interconnected.

The data package. The data package defines the classes that carry data from one component in a model to another. The Token class is of particular importance, because it is the base class for all units of data exchanged between components. The data.expr package defines the expression language, described in detail in Chapter 13. The expression language is used to assign values to parameters in a model and to establish interdependencies among parameters. The data.type package defines the type system (described in Chapter 14).

The math package. The math package contains mathematical functions and methods for operating on matrices and vectors. It also includes a complex number class, a class supporting fractions, and a set of classes supporting fixed-point numbers.

The graph package. The graph package and its subpackage, graph.analysis, provide algorithms for manipulating and analyzing mathematical graphs. This package supplies some of the core algorithms that are used in scheduling, in the type system, and in other model analysis tools.

The actor package. The actor package, described in more detail in Section 12.3, contains base classes for actors and I/O ports, where actors are defined as executable entities that receive and send data through I/O ports. The package also includes the base class Director that is customized for each domain to control model execution. The actor package contains several subpackages, including the following:

- The actor.lib package contains a large library of actors.
- The actor.sched package contains classes for representing and constructing schedules for executing actors.
- The actor.util package contains the core Time class, which implements model time, as described in Section 1.7.1. It also contains classes for keeping track of dependencies between output ports and input ports.
- The actor.gui package contains core classes for managing the user interface, including the Configuration class, which supports construction of customized, independently branded subsets of Ptolemy II. The Effigy and Tableau classes provide support for

opening and viewing models and submodels. The Placeable interface and associated classes provide support for actors with their own user interfaces.

The gui package. The gui package provides user interface components for interactively editing parameters of model components and for managing windows.

The moml package. The moml package provides a parser for **MoML** files (the **modeling markup language**, described in Lee and Neuendorffer (2000), which is the XML schema used to store Ptolemy II models.

The vergil package. The vergil package, which is quite large, provides the implementation of Vergil, the graphical user interface for Ptolemy II. Vergil is described further in Chapter 2.

There are many other packages and classes, but those discussed here provide a good overview of the overall system architecture. In the next section, we will explain how the kernel package defines the structure of models.

12.2 The Structure of Models

Computer scientists make a distinction between the syntax and the semantics of programming languages. The programming language syntax specifies the rules for constructing valid programs, and the semantics refers to the program's meaning. The same distinction is pertinent to models. The syntax of a model is how it is represented, while the semantics is what the model means.

Computer scientists further make a distinction between abstract syntax and concrete syntax. The **abstract syntax** of a model is the structure of its representation. For example, a model may be given as a **graph**, which is a collection of nodes and edges where the edges connect the nodes. Or it may be given as a **tree**, which is a type of graph that can be used to define a hierarchy. The abstract syntax of Ptolemy II models is (loosely) a tree (which represents the hierarchy of models) overlaid with a graph at each level of the hierarchy (for specifying the connections between components). The structure of a tree ensures that every node has exactly one **container**.

A **concrete syntax**, in contrast, is a specific notation for representing an abstract syntax. The block diagrams of Vergil are a concrete syntax. The MoML XML schema is a textual syntax (a syntax built from strings of characters) for the same models. The set of all

structures that a concrete syntax can represent is its abstract syntax. Whereas an abstract syntax constrains the structure of a model, a concrete syntax provides a description of the model in text or pictures.

Both abstract and concrete syntaxes can be formally defined. A textual concrete syntax, for example, might be given in **Backus-Naur form** (**BNF**), familiar to computer scientists. In fact, BNF is a concrete syntax for describing concrete syntaxes. Compiler toolkits, such as the classic Yacc parser generator, take BNF as an input to automatically create a **parser**, which converts a textual concrete syntax into data structures in the memory of a computer that represent the abstract syntax.

Abstract syntax is more fundamental than concrete syntax. Given two concrete syntaxes for the same abstract syntax, translation from one concrete syntax to the other is always possible. Hence, for example, every Vergil block diagram can be represented in MoML and vice versa.

In general, a **meta model** is a model of a modeling language or notation. In the world of modeling, engineers use meta models to precisely define abstract syntaxes. A meta model can for example be given as a **UML class diagrams**, a notation for object-oriented designs (UML stands for the unified modeling language). A meta model in UML for the Ptolemy II abstract syntax is shown in Figure 12.2. The figure shows the relationships between object-oriented classes that are instantiated by the MoML parser to create a data structure representing a Ptolemy model. Instances of these classes comprise a **Ptolemy II model**.

Every component in a Ptolemy II model is an instance of the NamedObj class, which has a name and a container (the container encloses part of the hierarchical model; it is null for the **top-level** object). There are four subclasses of NamedObj. These are called Attribute, Entity, Port, and Relation. Instances of these subclasses are shown in a Ptolemy model in Figure 12.3.

A model consists of a top-level entity that contains other entities. The entities have ports through which they interact. Their interactions are mediated by relations, which represent communication paths. All of these objects (entities, ports, and relations) can be assigned attributes, which define their parameters or annotations. Ports have **links** to relations, represented in the meta model as an association between the Relation class and the Port class.

A NamedObj contains a (possibly empty) list of instances of Attribute. An Entity also contains a (possibly empty) collection of instances of Port. Ports are associated with

instances of Relation, which define the connections between ports. A CompositeEntity is an Entity that contains instances of Entity and Relation. The resulting hierarchy of a model is illustrated in Figure 12.4. As described earlier, an actor is an executable entity, as indicated in Figure 12.2 by the fact that AtomicActor and CompositeActor implement the Executable interface. A director is an executable instance of the Director class, a subclass

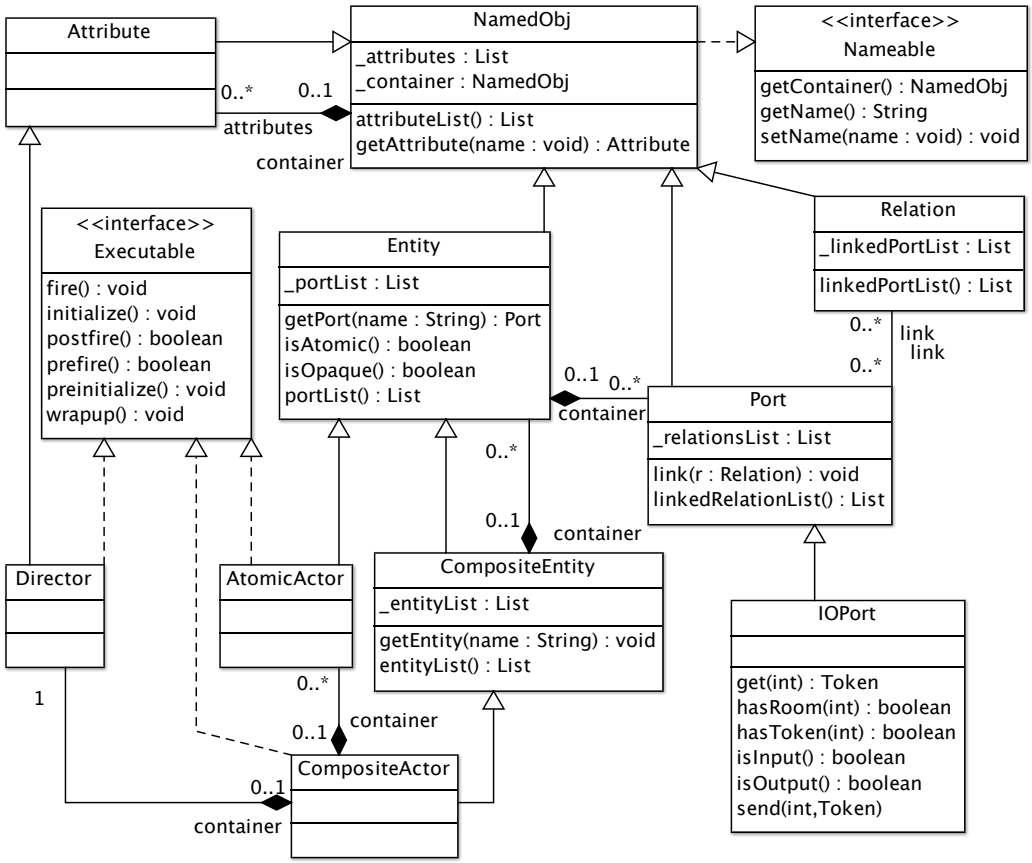

Figure 12.2: A meta model for Ptolemy II. This is a UML class diagram. The boxes represent classes with the class name, key members, and key methods shown. The lines with triangular arrowheads represent inheritance. The lines with diamond ends represent containment.

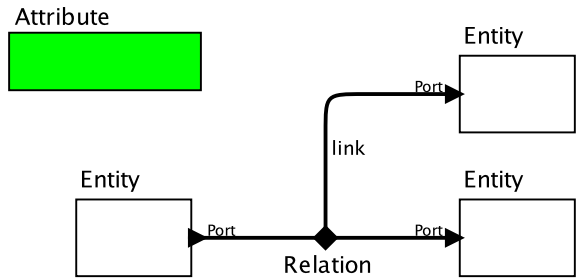

Figure 12.3: A Ptolemy II model showing the base meta-model class names for the objects in the model.

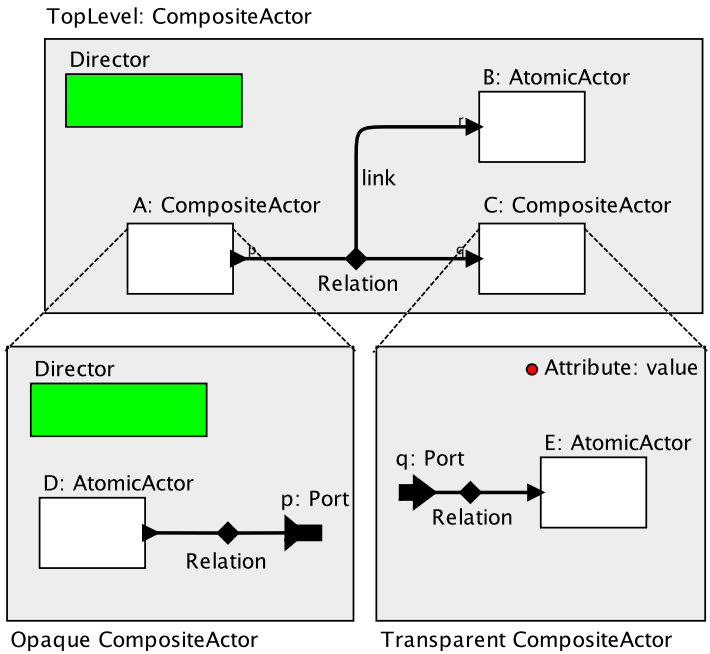

Figure 12.4: A hierarchical model showing meta-model class names for the objects in the model.

of Attribute. Each level of a hierarchical model has either one director or none; the top level always has one director.

Example 12.1: An example of a hierarchical Ptolemy II model is shown in Figure 12.4 using the concrete **visual syntax** of the Vergil visual editor.

The figure shows three distinct submodels and their hierarchical relationships. The top level of the hierarchy is labeled "TopLevel: CompositeActor," which means that its name is TopLevel and that it is an instance of CompositeActor. TopLevel contains an instance of Director, three actors, and one relation. Actors A and C are composite, whereas actor B is atomic. The ports of the three actors are linked to the relation. The ports of the composite actors appear twice in the diagram, once on the outside of the composite and once on the inside.

The block diagram in Figure 12.4 uses one of many possible concrete syntaxes for the same model. The model can also be defined in Java syntax, as shown in Figure 12.5, or in an XML schema known as MoML, as shown in Figure 12.6. All three syntaxes describe the model structure (which conforms to the abstract syntax). We will next give the structure some meaning (a semantics).

12.3 Actor Semantics and the MoC

Many Ptolemy II models are actor-oriented models; that is, they are based on connected groups of actors. In an actor-oriented model, actors execute concurrently and transfer data to each other via ports. What it means to "execute concurrently" and the manner in which data are passed between actors depend on the model of computation (MoC) in which the actor is running. In Ptolemy II, the model of computation is, in turn, defined by the director that is placed in that portion of the model.

An actor can itself be a Ptolemy II model, referred to as a composite actor. A composite actor that contains a director is said to be **opaque**; otherwise, it is **transparent**. An opaque composite actor behaves like a non-composite (i.e., atomic) actor, and its internal structure is not visible to the model in which it is used; it is a black box. In contrast, a transparent composite actor is fully visible from the outside, and is not executable on its

```
1  import ptolemy.actor.AtomicActor;
2  import ptolemy.actor.CompositeActor;
3  import ptolemy.actor.Director;
4  import ptolemy.actor.IOPort;
5  import ptolemy.actor.IORelation;
6  import ptolemy.kernel.Relation;
7  import ptolemy.kernel.util.IllegalActionException;
8  import ptolemy.kernel.util.NameDuplicationException;
9
10 public class TopLevel extends CompositeActor {
11    public TopLevel()
12            throws IllegalActionException,
13            NameDuplicationException {
14         super();
15         // Construct top level.
16         new Director(this, "Director");
17         CompositeActor A = new CompositeActor(this, "A");
18         IOPort p = new IOPort(A, "p");
19         AtomicActor B = new AtomicActor(this, "B");
20         IOPort r = new IOPort(B, "r");
21         CompositeActor C = new CompositeActor(this, "C");
22         IOPort q = new IOPort(C, "q");
23         Relation relation = connect(p, q);
24         r.link(relation);
25
26         // Populate composite actor A.
27         new Director(A, "Director");
28         AtomicActor D = new AtomicActor(A, "D");
29         IOPort D_p = new IOPort(D, "p");
30         Relation D_r = new IORelation(A, "r");
31         D_p.link(D_r);
32         p.link(D_r);
33
34         // Populate composite actor C.
35         AtomicActor E = new AtomicActor(C, "E");
36         IOPort E_p = new IOPort(E, "p");
37         Relation E_r = new IORelation(C, "r");
38         E_p.link(E_r);
39         q.link(E_r);
40     }
41 }
```

Figure 12.5: The model of Figure 12.4 given in the concrete syntax of Java.

```
<?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="TopLevel" class="ptolemy.actor.CompositeActor">
    <property name="Director" class="ptolemy.actor.Director"/>
    <entity name="A" class="ptolemy.actor.CompositeActor">
        <property name="Director" class="ptolemy.actor.Director"/>
        <port name="p" class="ptolemy.actor.IOPort"/>
        <entity name="D" class="ptolemy.actor.AtomicActor">
            <port name="p" class="ptolemy.actor.IOPort"/>
        </entity>
        <relation name="r" class="ptolemy.actor.IORelation"/>
        <link port="p" relation="r"/>
        <link port="D.p" relation="r"/>
    </entity>
    <entity name="B" class="ptolemy.actor.AtomicActor">
        <port name="r" class="ptolemy.actor.IOPort"/>
    </entity>
    <entity name="C" class="ptolemy.actor.CompositeActor">
        <property name="Attribute"
                  class="ptolemy.kernel.util.Attribute"/>
        <port name="q" class="ptolemy.actor.IOPort"/>
        <entity name="E" class="ptolemy.actor.AtomicActor">
            <port name="p" class="ptolemy.actor.IOPort"/>
        </entity>
        <relation name="r" class="ptolemy.actor.IORelation"/>
        <link port="q" relation="r"/>
        <link port="E.p" relation="r"/>
    </entity>
    <relation name="r" class="ptolemy.actor.IORelation"/>
    <link port="A.p" relation="r"/>
    <link port="B.r" relation="r"/>
    <link port="C.q" relation="r"/>
</entity>
```

Figure 12.6: The model of Figure 12.4 given in the concrete syntax of MoML.

own. Opaque composite actors — black boxes — are key to **hierarchical heterogeneity**, because they allow different models of computation to be nested within a single model.

Just as we make a distinction between abstract syntax and concrete syntax, we also make a distinction between **abstract semantics** and **concrete semantics**. Consider, for example, the communication between actors. The abstract semantics captures the fact that a communication occurs (that is, one actor sends a token to another), whereas a concrete semantics captures *how* the communication occurs (e.g., whether it is rendezvous communication, asynchronous message passing, a fixed point, etc.). A director realizes a concrete semantics; the interaction between directors across levels of the hierarchy is governed by the abstract semantics.

Ptolemy II provides a particular abstract semantics, called the **actor abstract semantics**, that is central to the interoperability of directors and the ability to build heterogeneous models. The actor abstract semantics defines three distinct aspects of the actor's behavior: execution control, communication, and a model of time, each of which is discussed in detail below.*

12.3.1 Execution Control

The overall execution of a model is controlled by an instance of the Manager class. An example execution sequence for a hierarchical model with an opaque composite actor is shown in Figure 12.7. Each opaque composite actor has a director. As shown in the meta model of Figure 12.2, a Director is an Attribute that implements the Executable interface. It is rendered in Vergil as a green rectangle, as shown in Figure 12.4. Inserting a Director into a composite actor makes the composite actor executable, since it implements the Executable interface. Atomic actors also implement this interface.

The Executable interface defines the actions of the actor abstract semantics that perform computation. These actions are divided into three phases: setup, iterate, and wrapup. Each of these phases is further divided into subphases (or actions), as described below.

The **setup** phase is divided into preinitialize and initialize actions, implemented by methods of the Executable interface. In the **preinitialize** action, an actor performs any operations that may influence static analysis (including scheduling, type inference and checking, code generation, etc.). A composite actor may alter its own internal structure — by

*In this book we describe the actor abstract semantics informally. A formal framework can be found in Tripakis et al. (2013) and Lee and Sangiovanni-Vincentelli (1998).

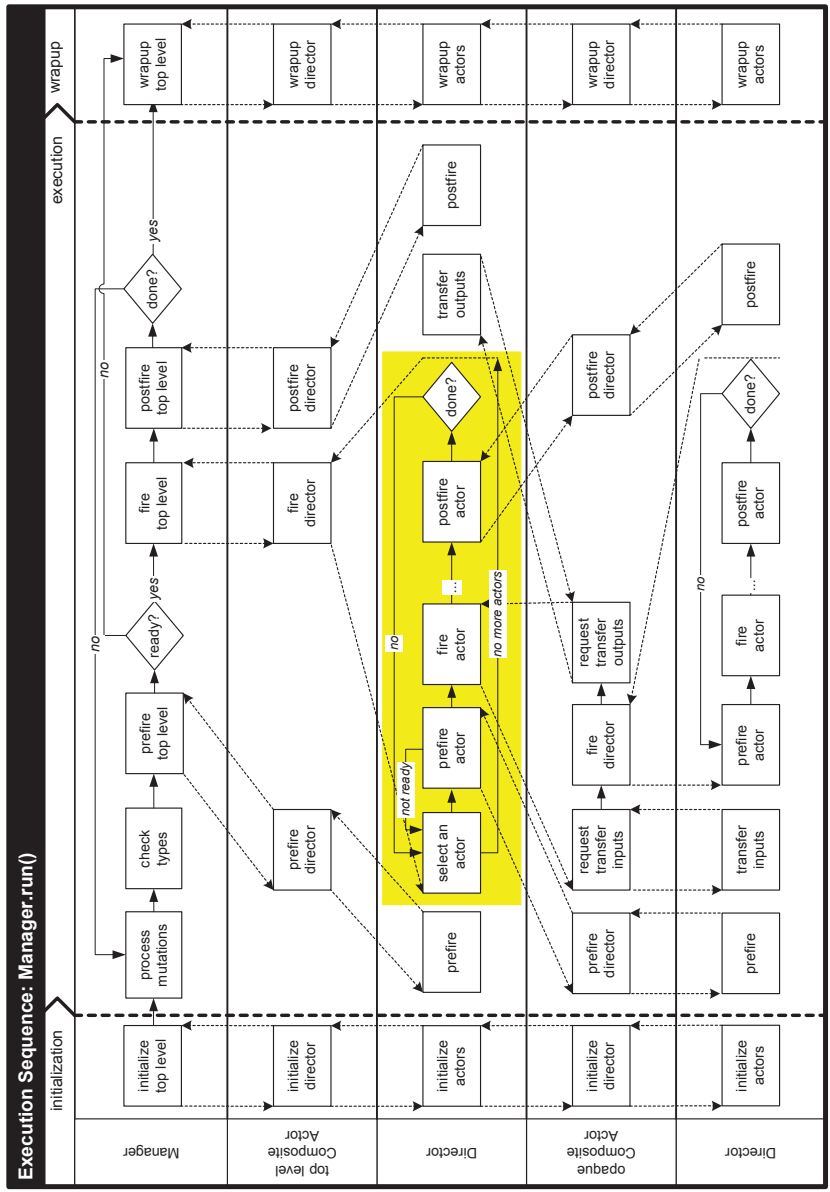

Figure 12.7: Execution of a hierarchical model with an opaque composite actor.

creating internal actors, for example — in this action. The **initialize** action of the setup phase initializes parameters, resets local state, and sends out any initial messages. Preinitialization of an actor is performed once, but initialization may be performed more than once during execution of a model. For example, initialization may be performed again if the semantics requires an actor to be re-initialized (as in the hybrid system formalism (Lee and Zheng, 2005)).

The **iterate** phase is the primary execution phase of a model. In this phase, each actor executes a sequence of **iteration**s, which are (typically finite) computations that lead the actor to a quiescent state. In a composite actor, the model of computation determines how the iteration of one actor relates to the iterations of other actors (whether they are concurrent or interleaved, how they are scheduled, etc.).

In order to coordinate the iterations among actors, an iteration is further broken into **prefire**, **fire**, and **postfire** actions. Prefire (optionally) tests the preconditions required for the actor to fire, such as the presence of sufficient inputs. The main computation of the actor is typically performed during the fire action, when it reads input data, performs computation, and produces output data. An actor may have persistent state that evolves during execution; the postfire action updates that state in response to any inputs. The fact that the state of an actor is updated only in postfire is an important part of the actor abstract semantics, as explained in the sidebar on page 433.

The **wrapup** phase is the final phase of execution. It is guaranteed to occur even if execution fails with an exception in a prior phase.

12.3.2 Communication

Like its execution control, an actor's communication capabilities are part of its abstract semantics. As described earlier, actors communicate via ports, which may be single ports or multiports. Each actor contains ports that are instances of IOPort, a subclass of Port, as shown in Figure 12.2. This subclass specifies whether a port is to be used for inputs or outputs. Two key methods that the IOPort subclass provides are `get` and `send`. As part of its fire action, an actor may use `get` to retrieve inputs, perform its computations, and use `send` to send the results to its output ports. For multiports, the integer arguments to `get` and `send` specify a channel. But what does it mean to get and send? Are communications

Sidebar: Why prefire, fire, and postfire?

Although it may not be immediately obvious, the division of each iteration of an actor's execution into prefire, fire, and postfire phases is essential for several Ptolemy II models of computation. As defined by the actor abstract semantics, the fire action reads inputs and produces outputs but does not change the state of the actor; state changes are only committed in the postfire phase. This approach is necessary for MoCs with a fixed-point semantics, which includes the synchronous-reactive (SR) and Continuous domains. Directors for such domains compute actor outputs by repeatedly firing the actors until a fixed point is reached. To ensure determinacy, it is essential that the state of each actor remain constant during these firings; the state of an actor can only be updated after the fixed point has been reached, at which point all the inputs to each actor are known. This does not occur until the postfire phase.

However, Ptolemy II does not strictly require every actor to follow this protocol. Goderis et al. (2009) classify actor-oriented MoCs into three categories of abstract semantics: **strict**, **loose**, and **loosest**. In the strict actor semantics, prefire, fire, and postfire are all finite computations, and only postfire changes the state. In the loose actor semantics, changes to the state may be made in the fire subphase. In the loosest actor semantics, the fire subphase may not even be finite; it may be a non-terminating computation.

An actor that conforms with the strict actor semantics is the most flexible type of actor in the sense that it may be used in any domain, including SR and Continuous. Such an actor is said to be **domain polymorphic**. Most actors in the library are domain polymorphic. An actor that conforms only with the loose actor semantics can be used with fewer directors (dataflow, for example). Those actors will be listed in domain-specific libraries. An actor that conforms only with the loosest actor semantics can be used with still fewer directors (process networks, for example). It is possible to define actors that will only work with a single type of director.

A director implements the same phases of execution as an actor. Thus, placing a director into a composite actor endows that composite actor with an executable semantics. If the director conforms to the strict actor semantics, then that composite actor is domain polymorphic. Such directors support the most flexible form of hierarchical heterogeneity in Ptolemy II because multiple directors with different MoCs may be combined hierarchically within a single model.

to be interpreted as a FIFO queue, a rendezvous communication, or other communication type? This meaning is specified by the director, not the actor.

A director determines how actors communicate by creating a **receiver** and placing it in an input port, with one receiver for each communication channel. A receiver is an object that implements the Receiver interface, as shown in Figure 12.8. That interface includes `put` and `get` methods. As illustrated in Figure 12.9, when one actor calls the `send` method of its output port, the output port delegates the call to the `put` method of the receiver in the designation input port(s). Similarly, when an actor calls the `get` method of an input port, the input port delegates the call to the `get` method of the receiver in the port. Thus, since the director provides the receiver, the director controls what it means to send and receive data.

Receivers can implement FIFO queues, mailboxes, proxies for a global queue, rendezvous, etc., all of which conform to the meta model shown in Figure 12.8. Directors provide receivers that implement a communication mechanism that is appropriate to the model of computation. Several receiver classes are shown in Figure 12.8.

Example 12.2: The PNReceiver, which is a subclass of QueueReceiver, is used by the process networks (PN) director. The `put` method of PNReceiver appends a data token t to a FIFO queue and then returns. When it returns, there is no assurance that the message has been received. The PN domain implements nonblocking writes; it delivers a token without waiting for the recipient to be ready to receive it, and thus will not block the model's continued execution.

In PN, every actor runs in its own Java thread. The actor receiving a message will therefore be running asynchronously in a different thread than the actor sending the token. The receiving actor will call the `get` method of its input port, which will delegate to the `get` method of the PNReceiver. The latter will block the calling thread until the FIFO queue has at least one token. It then returns the first token in the queue. Thus, the PNReceiver implements the blocking reads required by the PN domain. These blocking reads help ensure that a PN model is determinate.

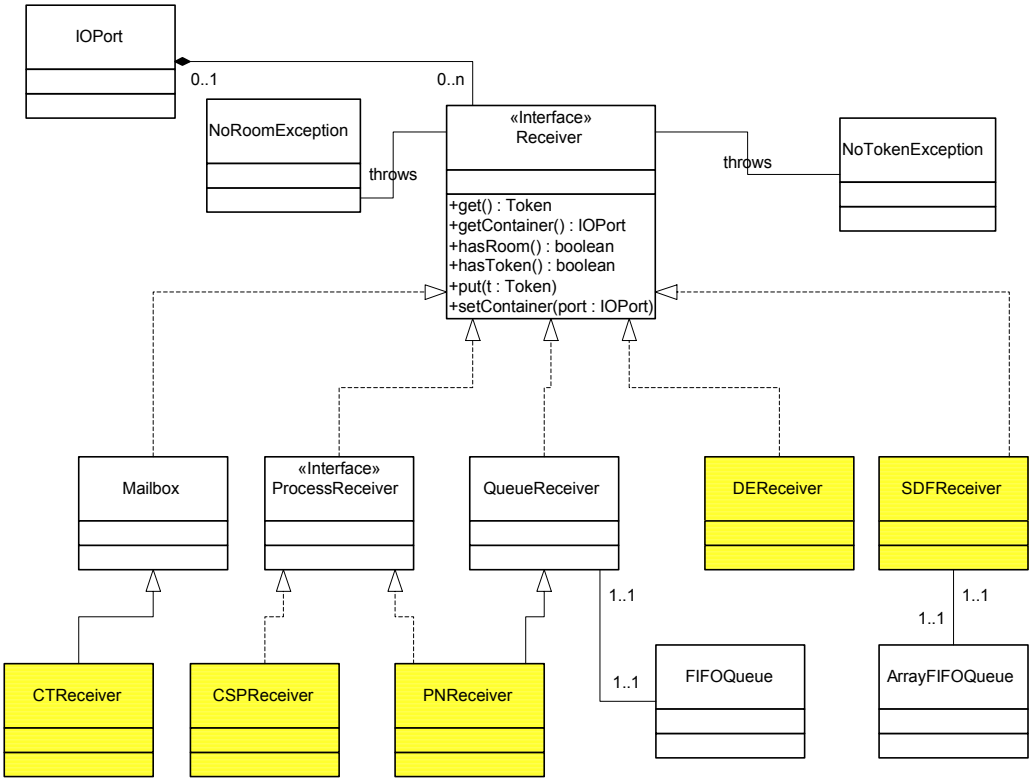

Figure 12.8: Meta model for communication in Ptolemy II.

12.3.3 Time

The final piece of the abstract actor semantics is the notion of time; that is, the way in which an actor views the passage of time when it is used in timed domains.

When an actor fires, it can ask its director for the current time. It does so with the following code:

```
Time currentTime = getDirector().getModelTime();
```

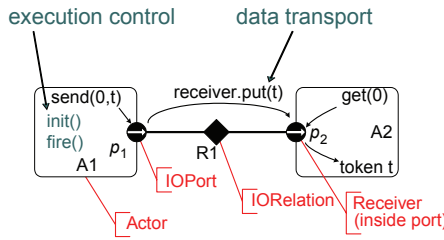

Figure 12.9: Communication mechanism in Ptolemy II.

The time returned by the director is called the **current** model time. If the actor requests the current model time only in its postfire subphase, then it is assured that the time value returned will be nondecreasing. If it requests the time in the fire subphase, however, then there is no such assurance, because some directors (such as the Continuous director, see Chapter 9) speculatively advance time while converging to a fixed point. In this case, the current model time may actually be *less* than its value in a prior invocation of `fire`.

An actor can request that it be fired at some future time by invoking the `fireAt` method of its director. The director is responsible for ensuring that the actor is fired at the requested future time. If it cannot honor the request, the director returns an alternative time at which it can fire the actor. Note, however, that the actor cannot assume that its *next* firing will be at that future time; there may be an arbitrary number of intervening firings. Moreover, if the execution of the model ends before time advances to the requested future time, then the actor will not be fired at the requested time.

The model hierarchy is central to the management of time. Typically, only the top-level director advances time. Other directors in a model obtain the current model time from their enclosing director. If the top-level director does not implement a timed model of computation, then time does not advance.

Perhaps counterintuitively, even untimed domains provide access to time. In a hierarchical model, unless the untimed domain is at the top level, it will delegate operations relating to time up the hierarchy to its container.

Timed and untimed models of computation may be interleaved in the hierarchy (see Section 1.7.1). There are certain combinations that do not make sense, however. For example, if the top-level director never advances time, and an actor requests a firing at a future time,

then the request cannot be honored. The director's `fireAt` method will return the time at which it can fire the actor, which will be time zero, since it never advances time. It is up to the actor to either accept this or to throw an exception to indicate that it is incompatible with an enclosing director.

Time can also advance non-uniformly in a model, as explained in Section 1.7.1. In particular, in modal models (Chapter 8), the advancement of time can be temporarily suspended (Lee and Tripakis, 2010). Within the submodel, there is a monotonically non-decreasing gap between the local time and the time in the enclosing environment. This mechanism is used to model temporary suspension of a submodel, as explained in Section 8.5.

12.4 Designing Actors in Java[†]

The functionality of actors in Ptolemy II can be defined in a number of ways. The most basic mechanism is the use of hierarchy, in which an actor is defined as the composite of other actors. For actors that implement complex mathematical functionality, however, it is often more convenient to use the Expression actor, whose functionality is defined using the expression language described in Chapter 13. Actors can also be created using the **MatlabExpression** actor, by defining the behavior as a MATLAB script. An actor can be defined in the Python language using the **PythonActor** or **PythonScript** actor, or can be defined using the **Cal** actor definition language (Eker and Janneck, 2003). But the most flexible method is to define the actor in Java, which is the focus of our discussion.

As described earlier, some actors are designed to be domain polymorphic, meaning that they can operate in multiple domains. Here, we focus on designing actors that are domain polymorphic. We will also focus on designing polymorphic actors that operate on a wide variety of token data types. Domain and data polymorphism help maximize the reusability of actors and minimize duplicated code when building an actor library.

Code duplication can also be avoided by using object-oriented inheritance. Inheritance can also help to enforce consistency across a set of actors. Most actors in the default library extend a common set of base classes that enforce uniform naming of commonly used ports and parameters. Using common base classes avoids unnecessary heterogeneity such as input ports named "in" vs. "inputSignal" vs. "input." This makes actors easier to use, and their code easier to maintain. To this end, we recommend using a reasonably deep

[†]This section assumes some familiarity with Java and object-oriented design.

class hierarchy to promote consistency. It is better to subclass and override an existing actor than to copy it and modify the copy.

Note that the Java source code for existing Ptolemy II actors, which can be viewed using the `Open Actor` context menu item, can provide a useful reference for defining new actors[‡].

Each actor consists of a source code file written in Java. An example of the source code for a simple actor is shown in Figure 12.10. This text can be placed in a Java file, compiled, instantiated in Vergil, and used as an actor. To create a new actor and use it in Vergil, choose a Java development environment (such as Eclipse), create a Java file, save the Java file in your classpath,[§] and instantiate the actor in Vergil. The latter can be accomplished by specifying the fully qualified class name in the dialog opened by the [`Graph→Instantiate Entity`] menu item. For example, you could copy the text in Figure 12.10 into a file named Count.java, save the file in the home directory of your Ptolemy installation, and then create an instance of this actor in Vergil using [`Graph→Instantiate Entity`].

In the source code shown in Figure 12.10, Lines 1 through 8 specify the Ptolemy classes on which this actor depends. The source code for those classes can also be viewed; Java development environments like Eclipse make it easy to view these files.

Line 10 defines a class named Count that subclasses TypedAtomicActor (the base class for most Ptolemy II actors whose ports and parameters have types). This particular actor could instead subclass Source or LimitedFiringSource, both of which would provide the needed ports. But here, for illustrative purposes, we include the port definitions. A particularly useful base class for actors is Transformer, shown in Figure 12.11. It is a reasonable choice for actors with one input and one output port.

Lines 12-20 give the constructor. The constructor is the Java procedure that creates instances of the class. The constructor takes arguments that define where the actor will be placed and the actor name. In the body of the constructor, line 15 creates an input port named *trigger*. The third and fourth arguments to the constructor for TypedIOPort designate that this port is an input and not an output. By convention, in Ptolemy II, every port

[‡] If you plan to contribute custom actors to the open source collection of actors in Ptolemy II, please be sure to follow the coding style given by Brooks and Lee (2003).

[§]The classpath is defined by an environment variable called CLASSPATH that Java uses to search for class definitions. By default, when you run Vergil, if you have a directory called ".ptolemyII" in your home directory, then that directory will be in your classpath. You can put Java class files there and Vergil will find them.

```
import ptolemy.actor.TypedAtomicActor;
import ptolemy.actor.TypedIOPort;
import ptolemy.data.IntToken;
import ptolemy.data.expr.Parameter;
import ptolemy.data.type.BaseType;
import ptolemy.kernel.CompositeEntity;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;

public class Count extends TypedAtomicActor {
    /** Constructor */
    public Count(CompositeEntity container, String name)
            throws NameDuplicationException,
            IllegalActionException {
        super(container, name);
        trigger = new TypedIOPort(this, "trigger", true, false);
        initial = new Parameter(this, "initial", new IntToken(0));
        initial.setTypeEquals(BaseType.INT);
        output = new TypedIOPort(this, "output", false, true);
        output.setTypeEquals(BaseType.INT);
    }
    /** Ports and parameters. */
    public TypedIOPort trigger, output;
    public Parameter initial;

    /** Action methods. */
    public void initialize() throws IllegalActionException {
        super.initialize();
        _count = ((IntToken)initial.getToken()).intValue();
    }
    public void fire() throws IllegalActionException {
        super.fire();
        if (trigger.getWidth() > 0 && trigger.hasToken(0)) {
             trigger.get(0);
        }
        output.send(0, new IntToken(_count + 1));
    }
    public boolean postfire() throws IllegalActionException {
        _count += 1;
        return super.postfire();
    }
    private int _count = 0; /** Local variable. */
}
```

Figure 12.10: A simple Count actor.

```
1  public class Transformer extends TypedAtomicActor {
2
3      /** Construct an actor with the given container and name.
4       *  @param container The container.
5       *  @param name The name of this actor.
6       *  @exception IllegalActionException If the actor
7       *   cannot be contained by the proposed container.
8       *  @exception NameDuplicationException If the container
9       *   alreadyhas an actor with this name.
10      */
11     public Transformer(CompositeEntity container, String name)
12             throws NameDuplicationException,
13             IllegalActionException {
14         super(container, name);
15         input = new TypedIOPort(this, "input", true, false);
16         output = new TypedIOPort(this, "output", false, true);
17     }
18
19     ////////////////////////////////////////////////
20     ////             ports and parameters        ////
21
22     /** The input port. This base class imposes no type
23      *  constraints except that the type of the input
24      *  cannot be greater than the type of the output.
25      */
26     public TypedIOPort input;
27
28     /** The output port. By default, the type of this output
29      *  is constrained to be at least that of the input.
30      */
31     public TypedIOPort output;
32 }
```

Figure 12.11: Transformer is a useful base class for actors with one input and one output.

is visible as a public field (defined in this case on line 22), and the name of the public field matches the name given as a constructor argument on line 15. Matching these names is important for actor-oriented classes, explained in Section 2.6, to work correctly.

Line 16 defines a parameter named *initial*. Again, by convention, parameters are public fields with matching names, as shown on line 23. Line 17 specifies the data type of the parameter, constraining its possible values.

Lines 18 and 19 create the output port and set its data type. Nothing in this actor constrains the type of the *trigger* input, so any data type is acceptable.

The `initialize` method, given on lines 26 to 29, initializes the private local variable `_count` to the value of the *initial* parameter. By convention in Ptolemy II, the names of private and protected variables begin with an underscore. The cast to *IntToken* is safe here because the parameter type is constrained to be an integer.

The `fire` method on lines 30-36 reads the input port if it is connected (that is, if it has a width greater than zero) and has a token. In some domains, such as DE, it is important to read input tokens even if they are not going to be used. In particular, the DE director will repeatedly fire an actor that has unconsumed tokens on its inputs; failure to read an input will result an infinite sequence of firings. Line 35 sends an output token.

The `postfire` method on lines 37-40 updates the state of the actor by incrementing the private variable `_count`. As explained above, updating the state in `postfire` rather than `fire` enables the use of this actor with directors such as SR and Continuous that repeatedly fire an actor until reaching a fixed point.

12.4.1 Ports

By convention, ports are public members of actors. They represent a set of input and output channels through which tokens may pass to other ports. Figure 12.10 shows how to define ports as public fields and instantiate them in the constructor. Here, we describe a few options that may be useful when creating ports.

Multiports and Single Ports

A port can be a single port or a multiport. By default, a port is a single port. It can be declared to be a multiport as follows:

```
portName.setMultiport(true);
```

Each port has a width, which corresponds to its number of channels. If a port is not connected, the width is zero. If a port is a single port, the width can be zero or one. If a port is a multiport, the width can be larger than one.

Reading and Writing

Data (encapsulated in a token) can be sent to a particular channel of an output port using the following syntax:

```
portName.send(channelNumber, token);
```

where `channelNumber` begins with 0 for the first channel. The width of the port (which is the number of channels) can be obtained by

```
int width = portName.getWidth();
```

If the port is unconnected, then the token is not sent anywhere. The `send` method will simply return. Note that in general, if the channel number refers to a channel that does not exist, the `send` method simply returns without issuing an exception. In contrast, attempting to read from a nonexistent input channel will usually result in an exception.

A token can be sent to all output channels of a port by

```
portName.broadcast(token);
```

You can generate a token from a value and then send this token by

```
portName.send(channelNumber, new IntToken(integerValue));
```

A token can be read from a channel by

```
Token token = portName.get(channelNumber);
```

You can read from channel 0 of a port and extract the data value (assuming the type is known) by

```
double variableName = ((DoubleToken)
        portName.get(0)).doubleValue();
```

You can query an input port to determine whether a `get` will succeed (whether a token is available) by

```
boolean tokenAvailable = portName.hasToken(channelNumber);
```

You can also query an output port to see whether a `send` will succeed using

```
boolean spaceAvailable = portName.hasRoom(channelNumber);
```

although with many domains (like SDF and PN), the answer is always true.

Dependencies Between Ports

Many Ptolemy II domains perform analysis of the topology of a model as part of the process of scheduling the model's execution. SDF, for example, constructs a static schedule that sequences the invocations of actors. DE, SR, and Continuous all examine data dependencies between actors to prioritize reactions to simultaneous events. In all of these cases, the director requires additional information about the behavior of actors in order to perform the analysis. In this section, we explain how to provide that additional information.

Suppose you are designing an actor that does not require a token at its input port in order to produce a token on its output port when it fires. It is useful for the director to have access to this information, which can be conveyed within the actor's Java code. For example, the MicrostepDelay actor declares that its output port is independent of its input port by defining this method:

```
public void declareDelayDependency()
        throws IllegalActionException {
    _declareDelayDependency(input, output, 0.0);
}
```

By default, each output port is assumed to have a dependency on all input ports. By defining the above method, the MicrostepDelay actor alerts the director that this default is not applicable. There is a delay between the ports *input* and *output*. The delay here is declared to be 0.0, which is interpreted as a microstep delay. The scheduler can use this information to sequence the execution of the actors and to resolve causality loops. For domains that do not use dependency information (such as SDF), it is harmless to include the above method. Thus, these declarations help maximize the ability to reuse actors in a variety of domains.

Port Production and Consumption Rates

Some domains (notably SDF) make use of information about production and consumption rates at the ports of actors. If the author of an actor makes no specific assertion, the SDF director will assume that upon firing, the actor requires and consumes exactly one token on each input port, and produces exactly one token on each output port. To override this assumption, the author needs to include a parameter in the port that is named either *tokenConsumptionRate* (for input ports) or *tokenProductionRate* (for output ports). The value of these parameters is an integer that specifies the number of tokens consumed or produced in a firing. As always, the value of these parameters can be given by an expression that depends on other parameters of the actor. As in the previous example, these parameters have no effect in domains that do not use this information, but they enable actors to be used within domains that do (such as SDF).

Feedback loops in SDF require at least one actor in the loop to produce tokens in its `initialize` method. To alert the SDF scheduler that an actor includes this capability, the relevant output port must include an integer-valued parameter (named *tokenInitProduction*) that specifies the number of tokens initially produced. The SDF scheduler will use this information to determine that a model with cycles does not deadlock.

12.4.2 Parameters

Like ports, parameters are public members of actors by convention, and the name of the public member is required to match the name passed to the constructor of the parameter. Type constraints on parameters are specified in the same way as for ports.

An actor is notified when a parameter value has changed by having its method `attributeChang` called. If the actor needs to check parameter values for validity, it can do so by overriding this method. Consider the example shown in Figure 12.12, taken from the PoissonClock actor. This actor generates timed events according to a Poisson process. One of its parameters is *meanTime*, which specifies the mean time between events. This must be a double, as asserted in the constructor, but equally importantly, it is required to be positive. The actor can enforce this requirement as shown in lines 21-25, which will throw an exception if a non-positive value is given.

The `attributeChanged` method may also be used to cache the current value of a parameter, as shown on lines 26-28.

```
1  public class PoissonClock extends TimedSource {
2       public PoissonClock(CompositeEntity container, String name)
3                 throws NameDuplicationException,
4                 IllegalActionException {
5            super(container, name);
6            meanTime = new Parameter(this, "meanTime");
7            meanTime.setExpression("1.0");
8            meanTime.setTypeEquals(BaseType.DOUBLE);
9            ...
10      }
11      public Parameter meanTime;
12      public Parameter values;
13
14      /** If the argument is the meanTime parameter,
15       *  check that it is positive.
16       */
17      public void attributeChanged(Attribute attribute)
18                throws IllegalActionException {
19           if (attribute == meanTime) {
20                double mean = ((DoubleToken)meanTime.getToken())
21                          .doubleValue();
22                if (mean <= 0.0) {
23                     throw new IllegalActionException(this,
24                          "meanTime is required to be positive."
25                          " Value given: " + mean);
26                     }
27                } else if (attribute == values) {
28                     ArrayToken val = (ArrayToken)
29                               (values.getToken());
30                     _length = val.length();
31                } else {
32                     super.attributeChanged(attribute);
33                }
34           }
35           ...
36      }
37      ...
38 }
```

Figure 12.12: Illustration of the use of `attributeChanged` to validate parameter values.

12.4.3 Coupled Port and Parameter

Often, in the design of an actor, it is hard to decide whether a quantity should be specified by a port or by a parameter. Fortunately, you can easily design an actor to offer both options. An example of such an actor is the Ramp actor, which uses the code shown in Figure 12.13. This actor starts with an initial value given by the *init* parameter, which is then incremented by the value of *step*. The value of *step* can be specified by either a parameter named *step* or by a port named *step*. If the port is left unconnected, then the value will always be set by the parameter. If the port is connected, then the parameter provides the initial default value, and this value is subsequently replaced by any value that arrives on the port.

The parameter value is stored with the model containing the Ramp actor when it is saved to a MoML file. In contrast, any data that arrives on the port during execution of the model is not stored. Thus, the default value given by the parameter is persistent, while the values that arrive on the port are transient.

To support the use of both a parameter and a port, the Ramp actor creates an instance of the class PortParameter in its constructor, as shown in Figure 12.13. This is a parameter that creates an associated port with the same name. The `postfire` method first calls `update` on *step*, and then adds its value to the state. Calling `update` has the side effect of reading from the associated input port, and if a token is present there, updating the value of the parameter. It is essential to call `update` before reading the value of a PortParameter in order to ensure that any input token that might be available on the associated input port is consumed.

12.5 Summary

This chapter has provided a brief introduction to the software architecture of Ptolemy II. It explains the overall layout of the classes that comprise Ptolemy II and how key classes in the kernel package define the structure of a model. It also explains how key classes in the actor package define the execution of a model. And finally, it gives a brief introduction to writing custom actors in Java.

```
1  public class Ramp extends SequenceSource {
2       public Ramp(CompositeEntity container, String name)
3                 throws NameDuplicationException,
4                 IllegalActionException {
5            super(container, name);
6            init = new Parameter(this, "init");
7            init.setExpression("0");
8            step = new PortParameter(this, "step");
9            step.setExpression("1");
10           ...
11      }
12      public Parameter init;
13      public PortParameter step;
14      public void attributeChanged(Attribute attribute)
15                throws IllegalActionException {
16           if (attribute == init) {
17                _stateToken = init.getToken();
18           } else {
19                super.attributeChanged(attribute);
20           }
21      }
22      public void initialize() throws IllegalActionException {
23           super.initialize();
24           _stateToken = init.getToken();
25      }
26      public void fire() throws IllegalActionException {
27           super.fire();
28           output.send(0, _stateToken);
29      }
30      ...
31      public boolean postfire() throws IllegalActionException {
32           step.update();
33           _stateToken = _stateToken.add(step.getToken());
34           return super.postfire();
35      }
36      private Token _stateToken = null;
37 }
```

Figure 12.13: Code segments from the Ramp actor.