Chapter 5 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II) ," Technical Memorandum UCB/ERL M04/27, July 29, 2004, University of California, Berkeley, CA USA 94720.

# 5

# Designing Actors

*Authors:* *Christopher Brooks*
*Edward A. Lee*
*Jie Liu*
*Xiaojun Liu*
*Steve Neuendorffer*
*Yuhong Xiong*
*Winthrop Williams*

## 5.1 Overview

Ptolemy is about component-based design. The domains define the semantics of the interaction between components. This chapter explains the common, domain-independent principles in the design of components that are actors. Actors are components with input and output that at least conceptually operate concurrently with other actors.

The functionality of actors in Ptolemy II can be defined in a number of ways. The most basic mechanism is hierarchy, where an actor is defined as a composite of other actors. But composites are not always the most convenient. Using Expression actor, for instance, is often more convenient for involved mathematical relations. The functionality is defined using the expression language explained in an earlier chapter. Alternatively, you can use the MatlabExpression actor and give the behavior as a MATLAB script (assuming you have MATLAB installed). You can also define the behavior of an actor in Python, using the PythonActor or PythonScript actor. You can define the behavior in the experimental Cal actor definition language [35]. But the most flexible method is to define the actor in Java. This chapter explains how to do the latter. For the impatient, the appendix gives a tutorial walk through on the construction and use in Vergil of a simple actor.

As explained in the previous chapter, some actors are designed to be domain polymorphic, meaning that they can operate in multiple domains. Others are domain specific. Refer to the domain chapters in volume 3 for domain-specific information relevant to the design of actors. This chapter explains how to design actors so that they are maximally domain polymorphic. As also explained in the previous chapter, many actors are also data polymorphic. This means that they can operate on a wide variety of token types. Domain and data polymorphism help to maximize the reusability of actors and to minimize the amount of 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. Figure 4.1 shows a UML static structure diagram of the Ptolemy actor library. Three base classes, Source, Sink, and Transformer, exist to ensure consistent naming of ports and to avoid duplicating code associated with those ports. Since most actors in the library extend these base classes, users of the library can guess that an input port is named "input" and an output port is named "output," and they will probably be right. Using base classes avoids input ports named "in" or "inputSignal" or something else. This sort of consistency helps to promote re-use of actors because it makes them easier to use. Thus, we recommend using a reasonably deep class hierarchy to promote consistency.

## 5.2 Anatomy of an Actor

Each actor consists of a source code file written in Java. Sources are compiled to Java byte code as directed by the makefile in their directory. Thus, when creating a new actor, it is necessary to add its name to the local makefile. Vergil, described fully in its own chapter, is the graphical design tool commonly used to compose actors and other components into a complete program, a "Ptolemy model." To facilitate using an actor in Vergil, it must appear in one of the actor libraries. This permits it to be dragged from the library pallet onto the design canvas. The libraries are XML files. Many of the actor libraries are in the $PTII/ptolemy/actor/lib directory.

The basic structure of an actor is shown in figure 5.1. In that figure, keywords in bold are features of Ptolemy II that are briefly described here and described in more detail in the chapters of part 2. Italic text would be substituted with something pertinent in an actual actor definition.

We will go over this structure in detail in this chapter. The source code for existing Ptolemy II actors, located mostly in $PTII/ptolemy/actor/lib, should also be viewed as a key reference.

### 5.2.1 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 5.1 shows a single port *portName* that is an instance of TypedIOPort, declared in the line

```
public TypedIOPort portName;
```

Most ports in actors are instances of TypedIOPort, unless they require domain-specific services, in which case they may be instances of a domain-specific subclass, such as DEIOPort. The port is actually created in the constructor by the line

```
portName = new TypedIOPort(this, "portName", true, false);
```

The first argument to the constructor is the container of the port, this actor. The second is the name of the port, which can be any string, but by convention, is the same as the name of the public member. The third argument specifies whether the port is an input (it is in this example), and the fourth argument specifies whether it is an output (it is not in this example). There is no difficulty with having a

```
/** Javadoc comment for the actor class. */
public class ActorClassName extends BaseClass implements MarkerInterface {

   /** Javadoc comment for constructor. */
   public ActorClassName(CompositeEntity container, String name)
         throws NameDuplicationException, IllegalActionException  {
      super(container, name);
      // Create and configure ports, e.g. ...
      portName = new TypedIOPort(this, "portName", true, false);
      // Create and configure parameters, e.g. ...
      parameterName = new Parameter(this, "parameterName");
      parameterName.setExpression("0.0");
      parameterName.setTypeEquals(BaseType.DOUBLE);
   }

   ///////////////////////////////////////////////////////////////////
   ////                     ports and parameters                  ////

   /** Javadoc comment for port. */
    public TypedIOPort portName;

   /** Javadoc comment for parameter. */
   public Parameter parameterName;

   ///////////////////////////////////////////////////////////////////
   ////                         public methods                    ////

   /** Javadoc comment for fire method. */
   public void fire() {
      super.fire();
      ... read inputs and produce outputs ...
   }

   /** Javadoc comment for initialize method. */
   public void initialize() {
      super.initialize();
      ... initialize local variables ...
   }

   /** Javadoc comment for prefire method. */
   public boolean prefire() {
      ... determine whether firing should proceed and return false if not ...
      return super.prefire();
   }

   /** Javadoc comment for preinitialize method. */
   public void preinitialize() {
      super.preinitialize();
      ... set port types and/or scheduling information ...
   }

   /** Javadoc comment for postfire method. */
   public boolean postfire() {
      ... update persistent state ...
      ... determine whether firing should continue to next iteration and return false if not ...
      return super.postfire();
   }

   /** Javadoc comment for wrapup method. */
   public void wrapup() {
      super.wrapup();
      ... display final results ...
   }
}
```

FIGURE 5.1. Anatomy of an actor.

port that is both an input and an output, but it is rarely useful to have one that is neither.

*Multiports and Single Ports.* A port can be a single port or a multiport. By default, it is a single port. It can be declared to be a multiport with a statement like

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

All ports have a *width*, which corresponds to the number of channels the port represents. 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 with the syntax

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

where *channelNumber* is the number of the channel (beginning with 0 for the first channel). The width of the port, 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 complaining.

A token can be sent to all output channels of a port (or none if there are none) 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 contained value (if you know its type) by

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

You can query an input port to see whether such 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 most current domains, the answer is always true. Note that the get(), hasRoom() and hasToken() methods throw IllegalActionException if the channel is out of range, but send() just silently returns.

*Type Constraints.* Ptolemy II includes a sophisticated type system, described fully in the Type System chapter. This type system supports specification of type constraints in the form of inequalities between types. These inequalities can be easily understood as representing the possibility of lossless conversion. Type a is less than type b if an instance of a can be losslessly converted to an instance of b. For example, IntToken is less than DoubleToken, which is less than ComplexToken. However, LongToken is not less than DoubleToken, and DoubleToken is not less than LongToken, so these two types are said to be *incomparable*.

Suppose that you wish to ensure that the type of an output is greater than or equal to the type of a parameter. You can do so by putting the following statement in the constructor:

```
portName.setTypeAtLeast(parameterName);
```

This is called a *relative type constraint* because it constrains the type of one object relative to the type of another. Another form of relative type constraint forces two objects to have the same type, but without specifying what that type should be:

```
portName.setTypeSameAs(parameterName);
```

These constraints could be specified in the other order,

```
parameterName.setTypeSameAs(portName);
```

which obviously means the same thing, or

```
parameterName.setTypeAtLeast(portName);
```

which is not quite the same.

Another common type constraint is an *absolute type constraint*, which fixes the type of the port (i.e. making it monomorphic rather than polymorphic),

```
portName.setTypeEquals(BaseType.DOUBLE);
```

The above line declares that the port can only handle doubles. Another form of absolute type constraint imposes an upper bound on the type,

```
portName.setTypeAtMost(BaseType.COMPLEX);
```

which declares that any type that can be losslessly converted to ComplexToken is acceptable. By default, for any input port that has no declared type constraints, type constraints are automatically created that declares its type to be less than that of any output ports that have no declared type constraints.

If there are input ports with no constraints, but no output ports lacking constraints, then those input ports will be unconstrained. Conversely, if there are output ports with no constraints, but no input ports lacking constraints, then those output ports will be unconstrained. Of course, you can declare a port to be unconstrained by saying

```
portName.setTypeAtMost(BaseType.GENERAL);
```

For full details of the type system, see the Type System chapter in volume 2.

*Examples.* To be concrete, consider first the code segment shown in figure 5.2, from the Transformer class in the ptolemy.actor.lib package. This actor is a base class for actors with one input and one output. The code shows two ports, one that is an input and one that is an output. By convention, the Javadoc[1] comments indicate type constraints on the ports, if any. If the ports are multiports, then the Javadoc comment will indicate that. Otherwise, they are assumed to be single ports. Derived classes may change this, making the ports into multiports, in which case they should document this fact in the class comment. Derived classes may also set the type constraints on the ports.

An extension of Transformer is shown in figure 5.3, the SimplerScale actor, which is a simplified version of the Scale actor which is defined in $PTII/ptolemy/actor/lib/Scale.java. This actor produces

```
public class Transformer extends TypedAtomicActor {

    /** Construct an actor with the given container and name.
     *  @param container The container.
     *  @param name The name of this actor.
     *  @exception IllegalActionException If the actor cannot be contained
     *   by the proposed container.
     *  @exception NameDuplicationException If the container already has an
     *   actor with this name.
     */
    public Transformer(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        input = new TypedIOPort(this, "input", true, false);
        output = new TypedIOPort(this, "output", false, true);
    }

    ///////////////////////////////////////////////////////////////////
    ////                     ports and parameters                  ////

    /** The input port.  This base class imposes no type constraints except
     *  that the type of the input cannot be greater than the type of the
     *  output.
     */
    public TypedIOPort input;

    /** The output port. By default, the type of this output is constrained
     *  to be at least that of the input.
     */
    public TypedIOPort output;

}
```

FIGURE 5.2. Code segment showing the port definitions in the Transformer class.

1. Javadoc is a program that generates HTML documentation from Java files based on comments enclosed in "/** ... */".

```
import ptolemy.actor.lib.Transformer;
import ptolemy.data.IntToken;
import ptolemy.data.expr.Parameter;
import ptolemy.data.Token;
import ptolemy.kernel.util.*;
import ptolemy.kernel.CompositeEntity;

public class SimplerScale extends Transformer {
   ...
   public SimplerScale(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
      super(container, name);
      factor = new Parameter(this, "factor");
      factor.setExpression("1");

      // set the type constraints.
      output.setTypeAtLeast(input);
      output.setTypeAtLeast(factor);
    }

    ///////////////////////////////////////////////////////////////////
    ////                     ports and parameters                  ////

    /** The factor.
     *  This parameter can contain any token that supports multiplication.
     *  The default value of this parameter is the IntToken 1.
     */
    public Parameter factor;

    ///////////////////////////////////////////////////////////////////
    ////                         public methods                    ////

    /** Clone the actor into the specified workspace. This calls the
     *  base class and then sets the type constraints.
     *  @param workspace The workspace for the new object.
     *  @return A new actor.
     *  @exception CloneNotSupportedException If a derived class has
     *   an attribute that cannot be cloned.
     */
    public Object clone(Workspace workspace)
            throws CloneNotSupportedException {
        SimplerScale newObject = (SimplerScale)super.clone(workspace);
        newObject.output.setTypeAtLeast(newObject.input);
        newObject.output.setTypeAtLeast(newObject.factor);
        return newObject;
    }

    /** Compute the product of the input and the <i>factor</i>.
     *  If there is no input, then produce no output.
     *  @exception IllegalActionException If there is no director.
     */
    public void fire() throws IllegalActionException {
        if (input.hasToken(0)) {
            Token in = input.get(0);
            Token factorToken = factor.getToken();
            Token result = factorToken.multiply(in);
            output.send(0, result);
        }
    }
}
```

FIGURE 5.3. Code segment from the SimplerScale actor, showing the handling of ports and parameters.

an output token on each firing with a value that is equal to a scaled version of the input. The actor is polymorphic in that it can support any token type that supports multiplication by the *factor* parameter. In the constructor, the output type is constrained to be at least as general as both the input and the *factor* parameter.

Notice in figure 5.3 how the fire() method uses hasToken() to ensure that no output is produced if there is no input. Furthermore, only one token is consumed from each input channel, even if there is more than one token available. This is generally the behavior of domain-polymorphic actors. Notice also how it uses the multiply() method of the Token class. This method is polymorphic. Thus, this scale actor can operate on any token type that supports multiplication, including all the numeric types and matrices.

## 5.2.2 Port Rates and Dependencies Between Ports

Many Ptolemy II domains perform analysis of the topology of a model for the purposes of scheduling. SDF, for example, constructs a static schedule that sequences the invocations of actors. DE, SR, and CT all examine data dependencies between actors to prioritize reactions to events that are simultaneous. In all these cases, the director of the domain requires some additional information about the behavior of actors in order to perform the analysis. In this section, we explain what additional information you can provide in an actor that will ensure that it can be used in all these domains.

Suppose you are designing an actor that does not require a token at its input port in order to produce one on its output port. It is useful for the director to have access to this information. For example, the TimedDelay actor of the DE domain declares that its *output* port is independent of its *input* port by defining this method:

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

An output port has a function dependency on an input port if in its fire() method, it sends tokens on the output port that depend on tokens gotten from the input port. By default, actors declare that each output port depends on all input ports. If the actor writer does nothing, this is what a scheduler will assume. By overriding the pruneDependencies() method as above, the actor writer is asserting that for this particular actor, the output port named *output* does not depend on the input named *input* in any given firing. 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 Giotto and SDF), it is harmless to include the above the method. Thus, by making such declarations, you maximize the reuse potential of your actors.

Some domains (notably SDF) make use of information about production and consumption rates at the ports of actors. If the actor writer does nothing, the SDF will assume that an actor requires and consumes exactly one token on each input port when it fires and produces exactly one token on each output port. To override this assumption, the actor writer only needs to include a parameter (an instance of ptolemy.data.expr.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 the parameters of the actor. Including these parameters in the ports is harmless for domains that do not make use of this information, but including them makes such actors useful in SDF, and hence improves their reusability.

In addition to production and consumption rates, feedback loops in SDF require that at least one actor in the loop produce tokens in its initialize() method. To make the SDF scheduler aware that an actor does this, include a parameter in the output port that produces these tokens named “tokenInitProduction” with a value that is an integer specifying the number of tokens initially produced. The SDF scheduler will use this information to determine that a model with cycles does not deadlock.

## 5.2.3 Parameters

Like ports, parameters are public members of actors by convention. Figure 5.3 shows a parameter *factor* that is an instance of Parameter, declared in the line

```
public Parameter factor;
```

and created in the lines

```
factor = new Parameter(this, "factor");
factor.setExpression("2*PI");
```

The second line sets the default value of the parameter.

As with ports, you can specify type constraints on parameters. The most common type constraint is to fix the type, using

```
parameterName.setTypeEquals(BaseType.DOUBLE);
```

In fact, exactly the same relative or absolute type constraints that one can specify for ports can be specified for parameters as well. But in addition, arbitrary constraints on parameter values are possible, not just type constraints.

An actor is notified when a parameter value changes by having its attributeChanged() method called. Consider the example shown in figure 5.4, 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.

The attributeChanged() method is passed the parameter that changed. (Typically it is being changed by the user via the Configure dialog.) If this is *meanTime*, then this method checks to make sure that the specified value is positive, and if not, it throws an exception. This exception is presented to the user in a new dialog box. It shows up when the user attempts to commit a non-positive value. The new dialog requests that the user choose a new value or cancel the change.

A change in a parameter value sometimes has broader repercussions than just the local actor. It may, for example, affect the schedule of execution of actors. An actor can call the invalidateSchedule() method of the director, which informs the director that any statically computed schedule (if there is one) is no longer valid. This would be used, for example, if the parameter affects the number of tokens produced or consumed when an actor fires.

When the type of a parameter changes, the attributeTypeChanged() method in the actor containing that parameter will be called. The default implementation of this method in TypedAtomicActor is to invalidate type resolution. So parameter type change will cause type resolution to be performed in the model. This default implementation is suitable for most actors. In fact, most of the actors in the actor

```
public class PoissonClock extends TimedSource {

   public Parameter meanTime;
   public Parameter values;

   public PoissonClock(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
      super(container, name);
      meanTime = new Parameter(this, "meanTime");
      meanTime.setExpression("1.0");
      meanTime.setTypeEquals(BaseType.DOUBLE);
      ...
   }

   /** If the argument is the meanTime parameter, check that it is
     *  positive.
     *  @exception IllegalActionException If the meanTime value is
     *   not positive.
     */
   public void attributeChanged(Attribute attribute)
            throws IllegalActionException {
       if (attribute == meanTime) {
           double mean = ((DoubleToken)meanTime.getToken()).doubleValue();
           if (mean <= 0.0) {
               throw new IllegalActionException(this,
                       "meanTime is required to be positive.  meanTime given: "
                       + mean);
           }
       } else if (attribute == values) {
           ArrayToken val = (ArrayToken)(values.getToken());
           _length = val.length();
       } else {
           super.attributeChanged(attribute);
       }
    }
   ...
}
```

FIGURE 5.4. Code segment from the PoissonClock actor, showing the attributeChanged() method.

library do not override this method. However, if for some reason, an actor does not wish to redo type resolution upon parameter type change, the attributeTypeChanged() method can be overridden. But this is rarely necessary.

### 5.2.4 Constructors

We have seen already that the major task of the constructor is to create and configure ports and parameters. In addition, you may have noticed that it calls

```
super(container, name);
```

and that it declares that it throws NameDuplicationException and IllegalActionException. The latter is the most widely used exception, and many methods in actors declare that they can throw it. The former is thrown if the specified container already contains an actor with the specified name. For more details about exceptions, see the Kernel chapter.

### 5.2.5 Cloning

All actors are cloneable. A clone of an actor needs to be a new instance of the same class, with the same parameter values, but without any connections to other actors.

Consider the clone() method in figure 5.5, taken from the SimplerScale actor. This method begins with:

```
SimplerScale newObject = (SimplerScale)super.clone(workspace);
```

The convention in Ptolemy II is that each clone method begins the same way, so that cloning works its way up the inheritance tree until it ultimately uses the clone() method of the Java Object class. That method performs what is called a "shallow copy," which is not sufficient for our purposes. In particular, members of the class that are references to other objects, including public members such as ports and parameters, are copied by copying the references. The NamedObj and TypedAtomicActor base classes implement a "deep copy" so that all the contained objects are cloned, and public members reference the proper cloned objects[2].

Although the base classes neatly handle most aspects of the clone operation, there are subtleties involved with cloning type constraints. Absolute type constraints on ports and parameters are carried automatically into the clone, so clone() methods should never call setTypeEquals(). However, relative

```
public class SimplerScale extends Transformer {
   ...
   public SimplerScale(CompositeEntity container, String name)
           throws NameDuplicationException, IllegalActionException  {
      super(container, name);
      output.setTypeAtLeast(input);
      output.setTypeAtLeast(factor);
    }

    ///////////////////////////////////////////////////////////////////
    ////                     ports and parameters                  ////

    /** The factor.  The default value of this parameter is the integer 1. */
    public Parameter factor;

    ///////////////////////////////////////////////////////////////////
    ////                         public methods                    ////

    /** Clone the actor into the specified workspace. This calls the
     *  base class and then sets the type constraints.
     *  @param workspace The workspace for the new object.
     *  @return A new actor.
     *  @exception CloneNotSupportedException If a derived class has
     *   has an attribute that cannot be cloned.
     */
    public Object clone(Workspace workspace) throws CloneNotSupportedException {
       SimplerScale newObject = (SimplerScale)super.clone(workspace);
       newObject.output.setTypeAtLeast(newObject.input);
       newObject.output.setTypeAtLeast(newObject.factor);
       return newObject;
    }
   ...
}
```

FIGURE 5.5. Code segment from the SimplerScale actor, showing the clone() method.

2. Be aware that the implementation of the deep copy relies on a strict naming convention. Public members that reference ports and parameters must have the same name as the object that they are referencing in order to be properly cloned.

type constraints are not cloned automatically because of the difficulty of ensuring that the other object being referred to in a relative constraint is the intended one. Thus, in figure 5.5, the clone() method repeats the relative type constraints that were specified in the constructor:

```
newObject.output.setTypeAtLeast(newObject.input);
newObject.output.setTypeAtLeast(newObject.factor);
```

Note that at no time during cloning is any constructor invoked, so it is necessary to repeat in the clone() method any initialization in the constructor. For example, the clone() method in the Expression actor sets the values of a few private variables:

```
newObject._iterationCount = 1;
newObject._time = (Variable)newObject.getAttribute("time");
newObject._iteration =
   (Variable)newObject.getAttribute("iteration");
```

## 5.3 Action Methods

Figure 5.1 shows a set of public methods called the *action methods* because they specify the action performed by the actor. By convention, these are given in alphabetical order in Ptolemy II Java files, but we will discuss them here in the order that they are invoked. The first to be invoked is the preinitialize() method, which is invoked exactly once before any other action method is invoked. The preinitialize() method is often used to set type constraints. After the preinitialize() method is called, type resolution happens and all the type constraints are resolved. The initialize() method is invoked next, and is typically used to initialize state variables in the actor, which generally depends on type resolution.

After the initialize() method, the actor experiences some number of *iterations*, where an iteration is defined to be exactly one invocation of prefire(), some number of invocations of fire(), and at most one invocation of postfire().

### 5.3.1 Initialization

The initialize() method of the Average actor is shown in figure 5.6. This data- and domain-poly-

```
public class Average extends Transformer {
   ...
   public void initialize() throws IllegalActionException {
      super.initialize();
      _count = 0;
      _sum = null;
   }
   ...

   ///////////////////////////////////////////////////////////////////
   ////                         private members                   ////

   private Token _sum;
   private int _count = 0;
}
```

FIGURE 5.6. Code segment from the Average actor, showing the initialize() method.

morphic actor computes the average of tokens that have arrived. To do so, it keeps a running sum in a private variable `_sum`, and a running count of the number of tokens it has seen in a private variable `_count`. Both of these variables are initialized in the initialize() method. Notice that the actor also calls super.initialize(), allowing the base class to perform any initialization it expects to perform. This is essential because one of the base classes initializes the ports. An actor will almost certainly fail to run properly if super.initialize() is not called.

Note that the initialization of the Average actor does not affect, or depend on, type resolution. This means that the code to initialize this actor can be placed either in the preinitialize() method, or in the initialize() method. However, in some cases an actor may require part of its initialization to happen before type resolution, in the preinitialize() method, or part after type resolution, in the initialize() method. For example, an actor may need to dynamically create type constraints before each execution[3]. Such an actor must create its type constraints in preinitialize(). On the other hand, an actor may wish to produce (send or broadcast) an initial output token once at the beginning of an execution of a model. This production can only happen during initialize(), because data transport through ports depends on type resolution.

### 5.3.2 Prefire

The prefire() method is the only method that is invoked exactly once per iteration[4]. It returns a boolean that indicates to the director whether the actor wishes for firing to proceed. The fire() method of an actor should never be called until after its prefire() method has returned true. The most common use of this method is to test a condition to see whether the actor is ready to fire.

Consider for example an actor that reads from *trueInput* if a private boolean variable `_state` is *true*, and otherwise reads from *falseInput*. The prefire() method might look like this:

```
public boolean prefire() throws IllegalActionException {
  if (_state) {
    return trueInput.hasToken(0);
  } else {
    return falseInput.hasToken(0);
  }
}
```

It is good practice to check the superclass in case it has some reason to decline to be fired. The above example becomes:

```
public boolean prefire() throws IllegalActionException {
  if (_state) {
    return trueInput.hasToken(0) && super.prefire();
  } else {
    return falseInput.hasToken(0) && super.prefire();
```

3. The need for this is relatively rare, but important. Examples include higher-order functions, which are actors that replace themselves with other subsystems, and certain actors whose ports are not created at the time they are constructed, but rather are added later. In most cases, the type constraints of an actor do not change and are simply specified in the constructor.
4. Some domains invoke the fire() method only once per iteration, but others will invoke it multiple times (searching for global convergence to a solution, for example).

```
        }
    }
```

The prefire() method can also be used to perform an operation that will happen exactly once per iteration. Consider the prefire method of the Bernoulli actor in figure 5.7:

```
public boolean prefire() throws IllegalActionException {
   if (_random.nextDouble() <
        ((DoubleToken)(trueProbability.getToken())).doubleValue()) {
      _current = true;
   } else {
      _current = false;
   }
   return super.prefire();
}
```

This method selects a new boolean value that will correspond to the token sent during each firing in that iteration.

## 5.3.3 Fire

The fire() method is the main point of execution and is generally responsible for reading inputs and producing outputs. It may also read the current parameter values, and the output may depend on them. Things to remember when writing fire() methods are:

```
public class Bernoulli extends RandomSource {

   public Bernoulli(CompositeEntity container, String name)
           throws NameDuplicationException, IllegalActionException  {
      super(container, name);

      output.setTypeEquals(BaseType.BOOLEAN);

      trueProbability = new Parameter(this, "trueProbability");
      trueProbability.setExpression("0.5");
      trueProbability.setTypeEquals(BaseType.DOUBLE);
    }

   public Parameter trueProbability;

   public void fire() {
      super.fire();
      output.send(0, new BooleanToken(_current));
   }

   public boolean prefire() throws IllegalActionException {
      if (_random.nextDouble() < ((DoubleToken)(trueProbability.getToken())).doubleValue()) {
         _current = true;
      } else {
         _current = false;
      }
      return super.prefire();
   }

   private boolean _current;
}
```

FIGURE 5.7. Code for the Bernoulli actor, which is not data polymorphic.

- To get data polymorphism, use the methods of the Token class for arithmetic whenever possible (see the Data Package chapter). Consider for example the Average actor, shown in figure 5.8. Notice the use of the add() and divide() methods of the Token class to achieve data polymorphism.
- When data polymorphism is not practical or not desired, then it is usually easiest to use the set-TypeEquals() to define the type of input ports. The type system will assure that you can safely cast the tokens that you read to the type of the port. Consider again the Average actor shown in figure 5.8. This actor declares the type of its *reset* input port to be BaseType.BOOLEAN. In the fire() method, the input token is read and cast to a BooleanToken. The type system ensures that no cast error will occur. The same can be done with a parameter, as with the Bernoulli actor shown in figure 5.7.
- A domain-polymorphic actor cannot assume that there is data at all the input ports. Most domain-polymorphic actors will read at most one input token from each port, and if there are sufficient inputs, produce exactly one token on each output port.
- Some domains invoke the fire() method multiple times, working towards a converged solution. Thus, each invocation of fire() can be thought of as doing a tentative computation with tentative inputs and producing tentative outputs. Thus, the fire() method should not update persistent state. Instead, that should be done in the postfire() method, as discussed in the next section.

## 5.3.4 Postfire

The postfire() method has two tasks:

- updating persistent state, and
- determining whether the execution of an actor is complete.

Consider the fire() and postfire() methods of the Average actor in figure 5.8. Notice that the persistent state variables `_sum` and `_count` are not updated in fire(). Instead, they are shadowed by `_latestSum` and `_latestCount`, and updated in postfire().

The return value of postfire() is a boolean that indicates to the director whether execution of the actor is complete. By convention, the director should avoid iterating further an actor after its postfire() method returns false. In other words, the director won't call prefire(), fire(), or postfire() again during this execution of the model.

Consider the two examples shown in figure 5.9. These are base classes for source actors. SequenceSource is a base class for actors that output sequences. Its key feature is a parameter *firingCountLimit*, which specifies a limit on the number of iterations of the actor. When this limit is reached, the postfire() method returns false. Thus, this parameter can be used to define sources of finite sequences.

TimedSource is similar, except that instead of specifying a limit on the number of iterations, it specifies a limit on the current model time. When that limit is reached, the postfire() method returns false.

## 5.3.5 Wrapup

The wrapup() method is used typically for displaying final results. It is invoked exactly once at the end of an execution, including when an exception occurs that stops execution (as opposed to an exception occurring in, say, attributeChanged(), which does not stop execution). However, when an actor is removed from a model during execution, the wrapup() method is not called.

An actor may lock a resource (which it intends to release in wrapup() for example), or its designer

```
public class Average extends Transformer {

   ... constructor ...

    ///////////////////////////////////////////////////////////////////
    ////                     ports and parameters                  ////

    public TypedIOPort reset;

    ///////////////////////////////////////////////////////////////////
    ////                         public methods                    ////

   ... clone method ...

    public void fire() throws IllegalActionException {
        _latestSum = _sum;
        _latestCount = _count + 1;
        // Check whether to reset.
        for (int i = 0; i < reset.getWidth(); i++) {
            if (reset.hasToken(i)) {
                BooleanToken r = (BooleanToken)reset.get(0);
                if b(r.booleanValue()) {
                    // Being reset at this firing.
                    _latestSum = null;
                    _latestCount = 1;
                }
            }
        }
        if (input.hasToken(0)) {
            Token in = input.get(0);
            if (_latestSum == null) {
                _latestSum = in;
            } else {
                _latestSum = _latestSum.add(in);
            }
            Token out = _latestSum.divide(new IntToken(_latestCount));
            output.send(0, out);
        }
    }

    public void initialize() throws IllegalActionException {
        super.initialize();
        _count = 0;
        _sum = null;
    }

    public boolean postfire() throws IllegalActionException {
        _sum = _latestSum;
        _count = _latestCount;
        return super.postfire();
    }

    ///////////////////////////////////////////////////////////////////
    ////                         private members                   ////

    private Token _sum;
    private Token _latestSum;
    private int _count = 0;
    private int _latestCount;
}
```

FIGURE 5.8. Code segment from the Average actor, showing the action methods.

```
public class SequenceSource extends Source implements SequenceActor {

   public SequenceSource(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        firingCountLimit = new Parameter(this, "firingCountLimit");
        firingCountLimit.setExpression("0");
        firingCountLimit.setTypeEquals(BaseType.INT);
   }

   public Parameter firingCountLimit;

   ...
    public void attributeChanged(Attribute attribute)
            throws IllegalActionException {
        if (attribute == firingCountLimit) {
            _firingCountLimit =
                ((IntToken)firingCountLimit.getToken()).intValue();
        }
    }

   public boolean postfire() throws IllegalActionException {
        if (_firingCountLimit != 0) {
            _iterationCount++;
            if (_iterationCount == _firingCountLimit) {
                return false;
           }
       }
        return true;
   }

   protected int _firingCountLimit;
   protected int _iterationCount = 0;
}



public class TimedSource extends Source implements TimedActor {

   public TimedSource(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        stopTime = new Parameter(this, "stopTime");
        stopTime.setExpression("0.0");
        stopTime.setTypeEquals(BaseType.DOUBLE);
        ...
   }

   public Parameter stopTime;

   ...

   public boolean postfire() throws IllegalActionException {
        double time = ((DoubleToken)stopTime.getToken()).doubleValue();
        if (time > 0.0 && getDirector().getCurrentTime() >= time) {
            return false;
        }
        return true;
   }
}
```

FIGURE 5.9. Code segments from the SequenceSource and TimedSource base classes.

may have another reason to ensure that wrapup() always is called, even when the actor is removed from an executing model. This can be achieved by overriding the setContainer() method. In this case, the actor would have a setContainer() method which might look like this:

```
public void setContainer(CompositeEntity container)
        throws IllegalActionException, NameDuplicationException {
    if (container != getContainer()) {
      wrapup();
    }
  super.setContainer(container);
}
```

When overriding the setContainer() method in this way, it is best to make wrapup() idempotent (implying that it can be invoked many times without causing harm), because future implementations of the director might automatically unlock resources of removed actors, or call wrapup() on removed actors.

## 5.4 Coupled Port and Parameter

Often, in the design of an actor, it is hard to decide whether a quantity should be given by a port or by a parameter. Fortunately, you can design an actor to offer both options. An example of such an actor is shown in figure 5.10. This actor starts with an initial value, given by the *init* parameter, and increments it each time by the value of *step*. The value of *step* is given by either a parameter named *step* or a port named *step*. To use the parameter exclusively, the model builder simply leaves the port unconnected. If the port is connected, then the parameter provides the default value, used before anything arrives on the port. But after something arrives on the port, that is used.

When the model containing a Ramp actor is saved, only the parameter value is stored. No data that arrives on the port is stored. Thus, the default value given by the parameter is persistent, while the values that arrive on the port are transient.

To set up this arrangement, the Ramp actor creates an instance of the class PortParameter in its constructor, as shown in figure 5.10. This is a parameter that, when created, creates a coupled port. There is no need to explicitly create the port. The Ramp actor creates an instance of Parameter to specify the *init* value, since it makes less sense for the value of *init* to be provided via a port.

Referring to figure 5.10, the constructor, after creating *init* and *step*, sets up type constraints. These specify that the type of the output is at least as general as the types of *init* and *step*. The PortParameter class takes care of an additional constraint, which is that the type of the *step* parameter must match the type of the *step* port. The clone() method duplicates the type constraints that are given explicitly.

In the attributeChanged() method, the actor resets its state if *init* is the parameter that changed. This will work with an instance of Parameter, but it is not recommended for an instance of PortParameter. The reason is that each time you call getToken() on an instance of PortParameter, the method checks to see whether there is an input on the port, and consumes it if there is. Actors are expected to consume inputs in their action methods, fire() and postfire(), not in attributeChanged(). Some domains, like SDF, will be confused by the consumption of the token too early.

In the Ramp actor in figure 5.10, the fire() method simply outputs the current state, whereas the postfire() method calls getToken() on *step* and adds its value to the state. This follows the Ptolemy II convention that the state of the actor is not modified in fire(), but only in postfire(). Thus, this actor can be used in domains with fixed-point semantics, such as SR.

When using a PortParameter in an actor, care must be exercised to call update() exactly once per firing prior to calling getToken(). Each time update() is called, a new token will be consumed from the associated port (if the port is connected and has a token). If this is called multiple times in an iteration, it may result in consuming tokens that were intended for subsequent iterations. Thus, for example, update() should not be called in fire() and then again in postfire(). Moreover, in some domains (such

```
public class Ramp extends SequenceSource {

    public Ramp(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
        init = new Parameter(this, "init");
        init.setExpression("0");
        step = new PortParameter(this, "step");
        step.setExpression("1");

       // set the type constraints.
       output.setTypeAtLeast(init);
       output.setTypeAtLeast(step);
    }

    public Parameter init;
    public PortParameter step;

    public void attributeChanged(Attribute attribute) throws IllegalActionException {
       if (attribute == init) {
          _stateToken = init.getToken();
       } else {
          super.attributeChanged(attribute);
       }
    }

    public Object clone(Workspace workspace) throws CloneNotSupportedException {
       Ramp newObject = (Ramp)super.clone(workspace);
       newObject.output.setTypeAtLeast(newObject.init);
       newObject.output.setTypeAtLeast(newObject.step);
       ...
       return newObject;
    }

    public void fire() throws IllegalActionException {
       super.fire();
       output.send(0, _stateToken);
    }

    public void initialize() throws IllegalActionException {
       super.initialize();
       _stateToken = init.getToken();
    }

    ...

    public boolean postfire() throws IllegalActionException {
       step.update();
       _stateToken = _stateToken.add(step.getToken());
       return super.postfire();
    }

    private Token _stateToken = null;
}
```

FIGURE 5.10. Code segments from the Ramp actor.

as DE), it is essential that if a token is provided on a port, that it is consumed. In DE, the actor will be repeatedly fired until the token is consumed. Thus, it is an error to not call update() once per iteration.

It is important that the actor call getToken() exactly once in either the fire() method or in the postfire() method. In particular, it should not call it in both, because this could result in consumption of two tokens from the input port, inappropriately. Moreover, it should always call it, even if it has no use for the value. Otherwise, in the DE domain, the actor will be repeatedly fired if an input event is provided on the port but not consumed. Time cannot advance until that event is processed.

The way that the PortParameter class works is as follows. On each call to getToken(), the *step* instance of PortParameter first checks to see whether an input has arrived at the associated *step* port since the last setExpression() or setToken(), and if so, returns a token read from that port. Also, any call to get() on the associated port will result in the value of this parameter being updated, although normally an actor writer will not call get() on the port.

## 5.5 Iterate Method

Some domains (such as SDF) will always invoke prefire(), fire(), and postfire() in sequence, one after another, so there is no benefit from having their functionality separated into three methods. Moreover, in SDF this sequence of method invocations may be repeated a large number of times. An actor designer can improve execution efficiency by providing an iterate() method. Figure 5.11 shows the iterate() method of the Ramp actor. Its behavior is equivalent to invoking prefire(), and that returns true, then invoking fire() and postfire() in sequence. Moreover the iterate() method takes an integer argument, which specifies how many times this sequence of operations should be repeated. The return values are `NOT_READY`, `STOP_ITERATING`, or `COMPLETED`, which are constants defined in the Executable interface of the actor package. Returning `NOT_READY` is appropriate when prefire() would have returned false. Returning `STOP_ITERATING` is appropriate when postfire() would have returned false. Otherwise, the proper return value is `COMPLETED`.

## 5.6 Time

An actor can access current model time using:

```
double currentTime = getDirector().getCurrentTime();
```

Notice that although the director has a public method setCurrentTime(), an actor should never use it. Typically, only another enclosing director will call this method.

An actor can request an invocation at a future time using the fireAt() or fireAtCurrentTime() method of the director. These methods return immediately (for a correctly implemented director). The fireAt() method takes two arguments, an actor and a time. The fireAtCurrentTime() method takes only one argument, an actor. The director is responsible for performing one iteration of the specified actor at the specified time. These methods can be used to get a source actor started, and to keep it operating. In the actor's initialize() method, it can call fireAt() with a zero time. Then in each invocation of fire(), it calls fireAt() again.

Note that while fireAt() can safely be called by any of the actors action methods, code which executes asynchronously from the director should avoid calling fireAt(). Examples of such code include the private thread within the DatagramReader actor and the serialEvent() callback method of the SerialComm actor. Because these process hardware events, which can occur at any time, they instead use the fireAtCurrentTime() method. The fireAt() is incorrect because model time may advance between the call to getCurrentTime() and the call to fireAt(), which could result in fireAt() requesting a firing in the past. This will trigger an exception.

## 5.7 Icons

An actor designer can specify a custom icon when defining the actor. A (very primitive) icon editor is supplied with Ptolemy II version 4.0 and higher. To create an icon, in the File menu, select New and then Icon Editor. An editor opens that contains a gray box for reference that is the size of the default icon that will be supplied if you do not create a custom icon. To create a custom icon, drag in the visual elements from the library, set their parameters, and then save the icon in an XML file in the same directory with the actor definition. If the name of the actor class is Foo, then the name of the file should be FooIcon.xml. That is, simply append "Icon" to the class name and complete the file name with the extension ".xml".

One useful feature that is not immediately evident in the user interface is that when you specify a color to fill a geometric shape or to serve as the outline for the shape, you can make the color translus-

```
public class Ramp extends SequenceSource {

public Ramp(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException  {
        super(container, name);
       ...
       _resultArray = new Token[1];
   }

   ...
   public Object clone(Workspace workspace) throws CloneNotSupportedException {
       ...
       _resultArray = new Token[1];
       return newObject;
   }

   public int iterate(int count) throws IllegalActionException {
       // Check whether we need to reallocate the output token array.
       if (count > _resultArray.length) {
           _resultArray = new Token[count];
       }
       for (int i = 0; i < count; i++) {
           _resultArray[i] = _stateToken;
           step.update();
           _stateToken = _stateToken.add(step.getToken());
       }
       output.send(0, _resultArray, count);
       if (_firingCountLimit != 0) {
           _iterationCount += count;
           if (_iterationCount >= _firingCountLimit) {
               return STOP_ITERATING;
           }
       }
       return COMPLETED;
   }

   ...

   private Token[] _resultArray;
}
```

FIGURE 5.11. More code segments from the Ramp actor of figure 5.10, showing the iterate() method.

cent. To do that, first choose the color using the color chooser that is made available by the parameter editing dialog for the geometric shape, then note that the color gets specified as a four-tuple of real numbers that range from 0.0 to 1.0. The fourth of these numbers is the *alpha channel* for the color, which specifies transparency. A value of 1.0 makes the color opaque. A value of 0.0 makes the color completely transparent (no color will be visible, and the background will show through). For convenience, you can specify the color to be "none" in which case a fully transparent color is supplied.

### 5.7.1 The Older Method

Many actors in the library use an older method to specify the icon. For completeness, we document that method here. The Ramp actor, for instance, specifies the icon shown in figure 5.12 using the code shown in the constructor in figure 5.13. It uses a convenience method, `_attachText()`, which is a protected method defined in the base class NamedObj. This method creates an attribute named "_iconDescription" with a textual value, where in this case, the textual value is:

```
<svg>
  <rect x="-30" y="-20" width="60" height="40" style="fill:white"/>
  <polygon points="-20,10 20,-10 20,10" style="fill:grey"/>
</svg>
```

This is XML, using the schema SVG (scalable vector graphics). The Ptolemy II visual editor (Vergil) is built on top of a graphics package called Diva, which has limited support for SVG. As of this writing, the SVG elements that are supported are shown in figure 5.14. The positions in SVG are given by real numbers, where the values are increasing to the right and down from the origin, which is the nominal center of the figure. The Ramp icon contains a white rectangle and a polygon that forms a triangle.

Most of the elements in figure 5.14 support style attributes, as summarized in the table. A style

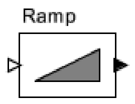

FIGURE 5.12. The Ramp icon.

```
public class Ramp extends SequenceSource {

public Ramp(CompositeEntity container, String name)
          throws NameDuplicationException, IllegalActionException  {
       super(container, name);
      ...
      _attachText("_iconDescription", "<svg>\n"
              + "<rect x=\"-30\" y=\"-20\" "
              + "width=\"60\" height=\"40\" "
              + "style=\"fill:white\"/>\n"
              + "<polygon points=\"-20,10 20,-10 20,10\" "
              + "style=\"fill:grey\"/>\n"
              + "</svg>\n");
   }

   ...

}
```

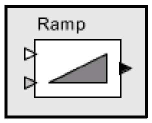

FIGURE 5.13. The Ramp actor defines a custom icon as shown.

attribute has value *keyword*:*value*. It can also have multiple *keyword*:*value* pairs, separated by semicolons. For example, the keywords currently supported by the *rect* element are "fill", "stroke" and "stroke-width." The "fill" gives the color of the body of the figure (for figures for which this makes sense), while the "stroke" gives the color of the outline. The supported colors are black, blue, cyan, darkgray, gray, green, lightgray, magenta, orange, pink, red, white, and yellow, plus any color supported by the Java Color class getColor() method. The "stroke-width" is a real number giving the thickness of the outline line, where the default is 1.0.

The image element, although tempting, is problematic in the current implementation. Images are very slow to load. It is not recommended.

| SVG element | Attributes |
|---|---|
| *rect* | *x*: horizontal position of the upper left corner<br>*y*: vertical position of the upper left corner<br>*width*: the width of the rectangle<br>*height*: the height of the rectangle<br>*style*: fill, stroke, stroke-width |
| *circle* | *cx*: horizontal position of the center of the circle<br>*cy*: vertical position of the center of the circle<br>*r*: radius of the circle<br>*style*: fill, stroke, stroke-width |
| *ellipse* | *cx*: horizontal position of the center of the ellipse<br>*cy*: vertical position of the center of the ellipse<br>*rx*: horizontal radius of the ellipse<br>*ry*: vertical radius of the ellipse<br>*style*: fill, stroke, stroke-width |
| *line* | *x1*: horizontal position of the start of the line<br>*y1*: vertical position of the start of the line<br>*x2*: horizontal position of the end of the line<br>*y2*: vertical position of the end of the line<br>*style*: stroke, stroke-width |
| *polyline* | *points*: List of x,y pairs of points, vertices of line segments, delimited by commas or spaces<br>*style*: stroke, stroke-width |
| *polygon* | *points*: List of x,y pairs of points, vertices of the polygon, delimited by commas or spaces<br>*style*: fill, stroke, stroke-width |
| *text* | *x*: horizontal position of the text<br>*y*: vertical position of the text<br>*style*: font-family, font-size, fill |
| *image* | *x*: horizontal position of the image<br>*y*: vertical position of the image<br>*width*: the width of the image<br>*height*: the height of the image<br>*xlink:href*: A URL for the image |

FIGURE 5.14. SVG subset currently supported by Diva, useful for creating custom icons.

# Appendix B: Creating and Using a Simple Actor

This appendix walks through the construction of a simple actor and the inclusion of that actor in the UserLibrary for use in Vergil. For this example, we are going to take the Ramp actor and change the default step from 1 to 2. The new actor will be called Ramp2. Note that this example commits a cardinal sin of software design: it copies code and makes small changes. It would be far better to subclass the actor and make the necessary changes using object-oriented techniques such as overriding. However, to illustrate the process, this provides a quick way to get some working code.

The instructions below assume that you have installed the Java Development Kit (JDK), which includes the javac binary, that you have make and other tools installed, that Ptolemy II has been installed, and that $PTII/configure and make have been run. In particular, the procedure will be different if you are using, for example, Eclipse as the software development environment. Under Windows XP with JDK1.4 and Cygwin, to do the initial setup, after installing the source code, do this:

```
bash-2.04$ PTII=c:/ptII4.0
bash-2.04$ export PTII
bash-2.04$ cd $PTII
bash-2.04$ ./configure
bash-2.04$ make fast >& make.out
```

This will usually generate a few warnings, but once it completes, you have compiled the Ptolemy II tree. Below are the steps to add an actor:

1. Create a directory in which to put the new actor. It is most convenient if that directory is in the classpath, which is most easily accomplished by putting it somewhere inside the $PTII directory. For this example, we will assume you do

```
cd $PTII
mkdir myActors
```

2. Create the new .java file that implements your actor:
   In this case, we are just copying a Ramp.java to Ramp.java

```
cd $PTII/ptolemy/actor/lib
cp -p Ramp.java $PTII/myActors/Ramp.java
```

3. Edit Ramp2.java and change:

```
package ptolemy.actor.lib;
```

   to

```
package myActors;
```

   You will need to add the imports:

```
import ptolemy.actor.lib.SequenceSource;
```

We also suggest that you change something about the actor, so that you can distinguish it from the original Ramp, such as changing:

```
step.setExpression("1");
```

to

```
step.setExpression("2");
```

4. Compile your actor:

```
cd $PTII/myActors
javac -classpath $PTII Ramp.java
```

5. Start Vergil.

```
vergil
```

If this does not work, you may not have created an alias for vergil. Try specifying the full path name, like this.

```
"$PTII/bin/vergil"
```

6. In Vergil, click on File -> New -> Graph Editor
7. In the graph editor window, select from the Graph menu “Instantiate Entity”. In the dialog that pops up, enter the classname for your new actor, which is “myActors.Ramp”. An instance of the actor will be created in the graph editor.
8. You can now also put this actor into the UserLibrary so that it will be available any time you start Vergil. To do that, right click on the actor icon and select “Save Actor in Library”. This will open a new window, which is a representation of the UserLibrary with the new actor added to it. Save the UserLibrary, and your new actor will henceforth appear in the UserLibrary whenever you run Vergil. You can edit the UserLibrary like any other Ptolemy II model. A convenient way to open it is to right click on its icon in the library window on the left and select “Open for Editing”.
9. To test the new Ramp actor:

   1. In a new graph editor window, drag the Ramp actor from UserLibrary to the main canvas on the right.

   2. Click on Actors -> Sinks -> GenericSinks and drag a Display actor over to the main canvas.

   3. Connect the two actors by clicking on the output of the Ramp actor and dragging over to the input of the Display actor.

4. Open the Directors library drag the SDF Director over to the right window.
5. Select View -> Run and change the number of iterations from 0 to 10, then hit the Run button.
6. You should see the numbers from 0 to 18 in the display.